Équations différentielles ordinaires

Ahmed Ammar (ahmed.ammar@fst.utm.tn)

Institut Préparatoire aux Études Scientifiques et Techniques, Université de Carthage.

Feb 11, 2020

Contents

1 Introduction

Dans les domaines scientifiques et industriels, il est courant aujourd'hui d'étudier la nature ou les dispositifs technologiques au moyen de modèles sur ordinateur. Avec de tels modèles, l'ordinateur agit comme un laboratoire virtuel où les expériences peuvent être effectuées de manière rapide, fiable, sûre et économique.

Les équations différentielles constituent l'un des outils mathématiques les plus puissants pour comprendre et prédire le comportement des systèmes dynamiques de la nature, de l'ingénierie et de la société. Un système dynamique est un système avec un état, généralement exprimé par un ensemble de variables, évoluant dans le temps. Par exemple, un pendule oscillant, la propagation d'une maladie et les conditions météorologiques sont des exemples de systèmes dynamiques. Nous pouvons utiliser les lois fondamentales de la physique, ou l'intuition simple, pour exprimer des règles mathématiques qui régissent l'évolution du système dans le temps. Ces règles prennent la forme d'équations différentielles.

2 Loi de désintégration radioactive

Considérons la désintégration radioactive des noyaux. Le nombre de noyaux, N, suit l'équation différentielle ordinaire:

$$\frac{dN(t)}{dt} = -\frac{N(t)}{\tau} \tag{1}$$

où τ est la constante de temps de décroissance (on l'appelle aussi durée de vie moyenne). Cette équation peut être intégrée directement, avec la solution:

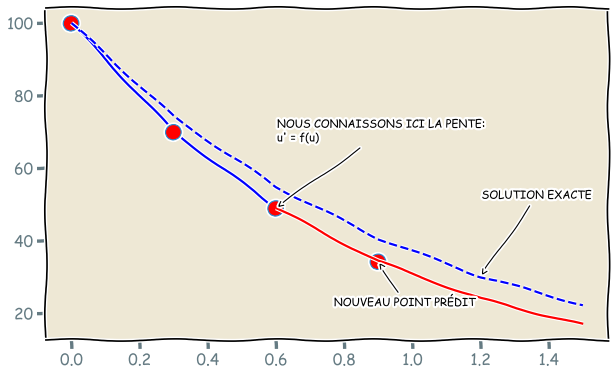
$$N(t) = N_0 e^{-t/\tau} \tag{2}$$

mais nous voulons essayer de résoudre l'équation numériquement.

L'approche la plus simple consiste à exprimer le nombre de noyaux à l'instant $t + \Delta t$ en termes de nombre à l'instant t:

$$N(t + \Delta t) = N(t) - \frac{N(t)}{\tau}\Delta t + \mathcal{O}(\Delta t^2) \tag{3}$$

Si nous commençons par N_0 noyaux à l'instant $t = 0$, alors à $t = \Delta t$ nous aurons $N(\Delta t) \approx N_0 - (N_0/\tau)\Delta t$; at $t = 2\Delta t$ nous aurons $N(2\Delta t) \approx N(\Delta t) - [N(\Delta t)/\tau]\Delta t$ etc. L'erreur de troncature est $\mathcal{O}(\Delta t^2)$. Par conséquent, si la taille du pas Δt est petite, nous nous attendons à ce que notre solution numérique soit proche de la solution exacte. Cette méthode d'intégration d'une équation différentielle ordinaire est connue sous le nom de **méthode d'Euler**.



Voici un programme qui implémentera cette méthode d'intégration de l'équation différentielle pour la désintégration radioactive:

```
## NOM DU PROGRAMME: desintegration.py
#%% IMPORTATION
import numpy as np
import matplotlib.pyplot as plt
#%% SOLUTION EXACTE
```

```python
def n_exact(t, noyaux0, tau):
    return noyaux0*np.exp(-t/tau)
#%% ENTRÉES
noyaux0 = int(input("nombre initial de noyaux: "))
tau = float(input('constante de temps de décroissance: '))
dt = float(input('pas de temps: '))
tmax = int(input('temps de fin de la simulation: '))
nsteps = int(tmax/dt)
noyaux = np.zeros(nsteps)
t = np.zeros(nsteps)
#%% VLEURS INITIALES
t[0] = 0.0
noyaux[0] =noyaux0
#%% BOUCLE PRINCIPALE: MÉTHODE D'EULER
for i in range(nsteps-1):
    t[i+1] = t[i] + dt
    noyaux[i+1] = noyaux[i] - noyaux[i]/tau*dt
#%% TRAÇAGE DU GRAPHIQUE
plt.figure(figsize=(8,5))
plt.plot(t, noyaux, '-r', label='Solution: Euler')
plt.plot(t, n_exact(t, noyaux0, tau), '--b', label='Solution exacte')
plt.xlabel('temps')
plt.ylabel('N(t)')
plt.title('Désintégration radioactive')
plt.grid()
plt.legend()
plt.tight_layout()
plt.savefig("desintegration.png")
plt.savefig("desintegration.pdf")
plt.show()
```

Le programme demande le nombre initial de noyaux, N_0, la constante de temps de décroissance τ, le pas de temps Δt et la durée totale de l'intégration t_{max}. Lorsque ce programme est exécuté avec les valeurs d'entrée sont; $N_0 = 100$, $\tau = 1$, $\Delta t = 0.04$ et $t_{max} = 5$, le programme produit le tracé présenté dans la Figure 1.

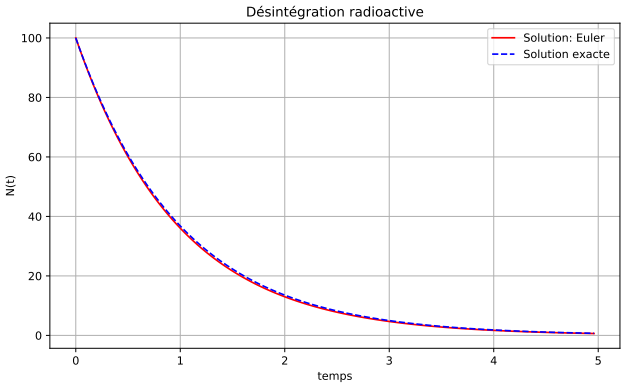


Figure 1: Résultat de l'exécution du programme *desintegration.py* avec entrée $N_0 = 100$, $\tau = 1$, $\Delta t = 0.04$ et $t_{max} = 5$.

Voyons maintenant à quel point notre programme est proche de la solution exacte. Vraisemblablement, lorsque le pas Δt est grand, l'erreur sera pire; aussi, les erreurs grandissent avec le temps. Pour voir cela, considérons une version modifiée de notre programme *desintegration.py* qui trace la différence fractionnaire entre le résultat numérique et le résultat exact donné par Eq. (2). Notre nouveau programme effectuera des évolutions numériques sur un certain nombre de **différentes valeurs du pas** afin que nous puissions voir comment l'erreur dépend du degré de raffinement de Δt.

```
## NOM DU PROGRAMME: desintegrationErr.py
#IMPORTATION
import numpy as np
import matplotlib.pyplot as plt
# ENTRÉES
noyaux0 = int(input("nombre initial de noyaux: "))
tau = float(input('constante de temps de décroissance: '))
dtbas = float(input('pas de temps de résolution le plus bas: '))
nres = int(input('nombre de raffinements de résolution: '))
tmax = int(input('temps de fin de la simulation: '))
# BOUCLE PRINCIPALE: CALCUL D'ERREURS
for n in range(nres):
    raffine = 10**n
    dt = dtbas/raffine
    nsteps = int(tmax/dt)
    noyaux = noyaux0
    err = np.zeros(nsteps)
    t = np.zeros(nsteps)
    # BOUCLE SECONDAIRE: MÉTHODE D'EULER
    for i in range(nsteps-1):
        t[i+1] = t[i] + dt
        noyaux = noyaux - noyaux/tau*dt
        exact = noyaux0 * np.exp(- t[i+1]/tau)
        err[i+1] = abs((noyaux - exact)/exact)
    # tracer l'erreur à cette résolution
    plt.loglog(t[raffine::raffine], err[raffine::raffine],
               '.-', label='dt = '+str(dt))

plt.legend(loc=4)
plt.xlabel('temps')
plt.ylabel('erreur fractionnaire')
plt.title("Erreur d'intégration de la désintégration radioactive")
plt.grid(linestyle='-', which='major')
plt.grid(which='minor')
plt.savefig("desintegrationErr.png")
plt.savefig("desintegrationErr.pdf")
plt.show()
```

Ce programme produit les résultats montrés à la Figure 2.

Les erreurs se rapprochent de manière linéaire avec le temps (les lignes du tracé logarithmique ont une pente approximativement égale à l'unité) et chaque facteur de 10 dans le raffinement diminue l'erreur fractionnaire d'un facteur 10. Pour comprendre cela, notez que le terme que nous avons jeté dans l'expansion de Taylor de notre équation différentielle ordinaire était le terme d^2N/dt^2, donc

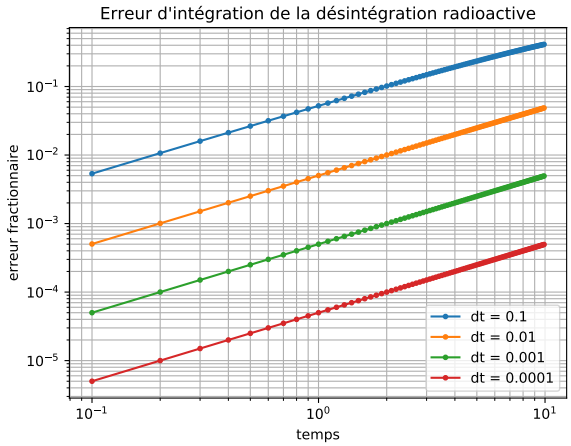


Figure 2: Résultat de l'exécution du programme *desintegrationErr.py* avec entrée $N_0 = 100$, $\tau = 1$, $\Delta t = 0.1$, $N_{res} = 4$ et $t_{max} = 10$.

chaque étape introduit une erreur de:

$$e_i \approx \frac{1}{2}\frac{d^2N(t_i)}{dt^2}\Delta t^2 = \frac{N(t_i)}{2\tau^2}\Delta t^2 \tag{4}$$

Ceci est connu sous le nom **d'erreur locale**. Si l'erreur locale d'un schéma d'intégration numérique est $\mathcal{O}(\Delta t^{p+1})$ comme $t \to 0$, alors on dit que c'est l'ordre p. La méthode d'Euler est donc un schéma d'intégration de premier ordre. **L'erreur globale** est l'erreur accumulée lorsque l'intégration est effectuée pendant une certaine durée T. Le nombre d'étapes requis est $n = T/\Delta t$ et chaque étape $i = 1...n$ accumule une erreur e_i, nous nous attendons donc à ce que l'erreur globale soit:

$$E_n \leq \sum_{i=1}^{n} e_i \leq T\frac{N_0}{2\tau^2}\Delta t \tag{5}$$

puisque $e_i \leq \frac{N_0}{2\tau^2}\Delta t^2$. Notez que pour un schéma d'intégration d'ordre p, l'erreur sera $\mathcal{O}(\Delta t^p)$; de plus, l'erreur grandit avec le temps T. Pour la méthode d'Euler, l'erreur croît de manière approximativement linéaire avec T et avec Δt, ce qui est ce que nous voyons sur la Figure 2.

Note
La méthode d'Euler n'est pas une méthode recommandée pour résoudre des équations différentielles ordinaires. S'agissant simplement du premier ordre, une précision souhaitée n'est obtenue que pour de très petites valeurs de Δt, de nombreuses étapes d'intégration sont donc nécessaires pour faire évoluer le système pour une durée donnée T. Mais le coût

en calcul de la méthode d'Euler n'est pas son seul inconvénient: elle n'est pas particulièrement stable non plus, comme nous le verrons plus loin dans ce chapitre..

3 Mouvement d'un projectile

Un autre exemple d'équation différentielle ordinaire est celui du mouvement du projectile, pour lequel les équations du mouvement sont:

$$\frac{dx}{dt} = v_x, \qquad \frac{dv_x}{dt} = 0, \tag{6}$$

$$\frac{dy}{dt} = v_y, \qquad \frac{dv_y}{dt} = -g, \tag{7}$$

où g est l'accélération de pesanteur. Nous pouvons utiliser la méthode d'Euler pour écrire chaque dérivée sous une forme de différence finie convenant à l'intégration numérique:

$$x_{i+1} = x_i + v_{x,i}\Delta t, \qquad v_{x,i+1} = v_{x,i}, \tag{8}$$

$$y_{i+1} = y_i + v_{y,i}\Delta t, \qquad v_{y,i+1} = v_{y,i} - g\Delta t, \tag{9}$$

Les trajectoires d'un projectile lancé avec une vitesse $v_0 = 10\ m\ s^{-1}$ à différents angles sont tracées par le programme `projectile.py` et sont tracées à la Fig. 3.

```
## NOM DU PROGRAMME: projectile.py
#%% IMPORTATION
import numpy as np
import matplotlib.pyplot as plt
#%% CONSTANTES
# accélération de la pesanteur(m/s^2)
g = 9.8
# angles de tire (deg)
angles = [30, 35, 40, 45, 50, 55]
# vitesse initiale (m/s())
v0 = 20.0
N = 10000 # Nombre de pas de temps
# pas de temps (s)
dt =0.001
#%% VLEURS INITIALES
x = np.zeros(N)
y = np.zeros(N)
vx = np.zeros(N)
vy = np.zeros(N)
for theta in angles:
    vx[0] = v0 * np.cos(theta*np.pi/180.0)
    vy[0] = v0 * np.sin(theta*np.pi/180.0)
    x[0], y[0]=0, 1
    # MÉTHODE D'EULER
```

```
    for i in range(N-1):
        x[i+1] = x[i] + vx[i] * dt
        y[i+1] = y[i] + vy[i] * dt
        vx[i+1] = vx[i]
        vy[i+1] = vy[i] - g * dt
    plt.plot(x, y, lw =2, label=str(theta)+' deg')
#%% GRAPHIQUE: TRAJECTOIRES
plt.xlabel('x [m]')
plt.ylabel('y [m]')
plt.suptitle("Trajectoire d'un projectile", weight='bold')
plt.title(r'Pas de temps: $\Delta_t$ = {:.3f} s'.format(dt))
plt.grid()
plt.legend()
plt.axis([0, 45, 0, 15])
# ENREGISTRER ET AFFICHER LA FIGURE
plt.savefig("projectile.png")
plt.savefig("projectile.pdf")
plt.show()
```

Nous voyons, comme prévu, que la plus grande plage est atteinte pour un angle de lancement de 45°.

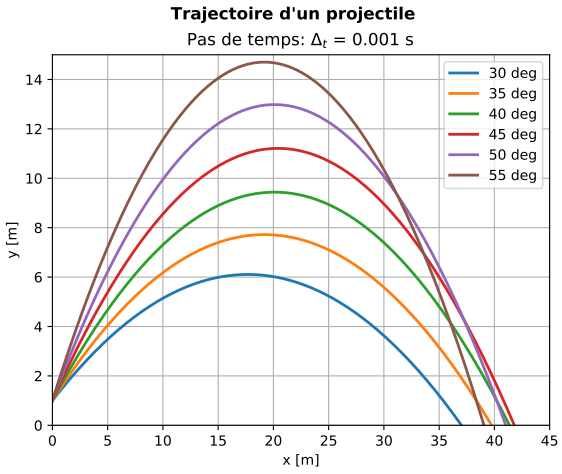


Figure 3: Résultats de l'exécution du programme `projectile.py`. On voit que la plus grande plage est atteinte avec un angle de lancement de $\theta = 45°$.

Trouver la trajectoire d'un projectile compte tenu de ses conditions initiales, $v_{x,i}$ et $v_{x,i}$ ou de manière équivalente v_0 et θ, est relativement simple. Cependant, supposons que nous voulons trouver l'angle de lancement θ requis pour atteindre une cible à une distance donnée avec une vitesse initiale v_0 donnée. Ceci est un exemple de problème de la *valeur aux limites* à deux points.Une approche pour résoudre un tel problème est connue comme *méthode de tir*.

L'idée est simple: devinez la valeur de θ, effectuez l'intégration, déterminez combien vous manquez votre note, puis affinez votre estimation de manière itérative jusqu'à ce que vous soyez suffisamment proche de la cible. Si $\Delta x(\theta)$ est

la quantité que vous manquez la cible avec l'angle de lancement θ alors l'objectif est de résoudre l'équation:

$$\Delta x(\theta) = 0 \tag{10}$$

pour θ. Ce problème général s'appelle **la recherche de racine**. Nous allons utiliser ici une méthode assez simple pour résoudre une racine appelée *méthode de bissection*. Supposons que nous savons que la racine de l'équation (10) se situe quelque part dans l'intervalle $\theta_1 < \theta < \theta_2$ et $\Delta x(\theta_1)$ a le signe opposé de $\Delta x(\theta_2)$ (c'est-à-dire si $\Delta x(\theta_1) < 0$ alors $\Delta x(\theta_2) > 0$, ou vice versa). On dit alors que θ_1 et θ_2 *encadrent* la racine.

Commençons par évaluer $\Delta x(\theta_1)$, $\Delta x(\theta_2)$ et $\Delta x(\theta_{deviner})$ avec $\theta_{deviner}$ au milieu entre θ_1 et θ_2 , $\theta_{deviner} = \frac{1}{2}(\theta_1 + \theta_2)$. Si le signe de $\Delta x(\theta_{deviner})$ est identique au signe de $\Delta x(\theta_1)$, alors nous savons que la racine doit être comprise entre $\theta_{deviner}$ et θ_2, nous assignons donc θ_1 à $\theta_{deviner}$ et faisons une nouvelle hypothèse à mi-chemin entre les nouveaux θ_1 et θ_2. Sinon, si le signe de $\Delta x(\theta_{deviner})$ est identique au signe de $\Delta x(\theta_2)$, nous savons que la racine doit être comprise entre θ_1 et $\theta_{deviner}$. Nous affectons donc θ_2 à $\theta_{deviner}$ et faisons une nouvelle hypothèse à mi-chemin entre θ_1 et le nouveau θ_2. Nous continuons cette itération jusqu'à ce que nous soyons *suffisamment proche*, c'est-à-dire $|\Delta x(\theta_{deviner})| < \epsilon$ pour une petite valeur de ϵ.

Pour le problème à résoudre, la cible doit être située à une distance x_{cible} et le point où le projectile touche le sol lorsqu'il est lancé à l'angle θ est $x_{sol}(\theta)$. Définir $\Delta x(\theta) = x_{sol}(\theta) - x_{cible}$ de telle sorte que $\Delta x(\theta) > 0$ si nous avons tiré trop loin et $\Delta x(\theta) < 0$ si nous avons tiré trop près. Ensuite, si $0 < x_{cible} < x_{max}$ où nous connaissons $x_{sol}(0°) = 0$ et $x_{sol}(45°) = x_{max}$, alors nous savons que $\theta_1 = 0°$ et $\theta_2 = 45°$ encadrent la racine. Le programme `tire.py` utilise la méthode de tir pour calculer la trajectoire d'un projectile lancé à partir de $x = 0$ avec une vitesse fixe et atterrissant au point $x = x_{sol}$. Le résultat de ce programme exécuté avec une vitesse initiale $v_0 = 10\ m\ s^{-1}$ et un emplacement cible $x_{cible} = 8\ m$ est présenté à la Fig. 4.

```
## NOM DU PROGRAMME: tire.py
#%% IMPORTATION
import numpy as np
import matplotlib.pyplot as plt
#%% CONSTANTES
# accélération de la pesanteur (m/s^2)
g = 9.8
# vitesse initiale (m/s)
v0 = 10.0
# plage cible (m)
xcible = 8.0
# comment nous devons nous rapprocher (m)
eps = 0.01
# pas de temps (s)
dt = 0.001
# angle (degrés) que le projectile tombe trop court
theta1 = 0.0
# angle (degrés) que le projectile tombe trop loin
theta2 = 45.0
# une valeur initiale> eps
```

```
dx = 2*eps
#%% BOUCLE PRINCIPALE: MÉTHODE DE TIRE
while abs(dx) > eps:
    # devinez à la valeur de thêta
    theta = (theta1+theta2)/2.0
    x = [0.0]
    y = [0.0]
    vx = [v0*np.cos(theta*np.pi/180.0)]
    vy = [v0*np.sin(theta*np.pi/180.0)]
    # MÉTHODE D'EULER
    i = 0
    while y[i] >= 0.0:
        # appliquer une différence finie approximative
        # aux équations du mouvement
        x += [x[i]+vx[i]*dt]
        y += [y[i]+vy[i]*dt]
        vx += [vx[i]]
        vy += [vy[i] - g*dt]
        i = i+1
    # nous avons touché le sol quelque part entre l'étape i-1 et i
    # interpoler pour trouver cet emplacement
    xsol = x[i - 1]+y[i - 1]*(x[i] - x[i - 1])/(y[i] - y[i - 1])
    # mettre à jour les limites encadrant la racine
    dx = xsol - xcible
    if dx < 0.0: # trop court: mettre à jour l'angle plus petit
        theta1 = theta
    else: # trop loin: mettre à jour un angle plus grand
        theta2 = theta
#%% GRAPHIQUE: TRAJECTOIRES
plt.plot(x, y, lw =2)
plt.plot([xcible], [0.0], 'o', ms=12)
plt.annotate('Cible', xy=(xcible, 0), xycoords='data', xytext=(5,5),
             textcoords='offset points')
plt.title(r"trajectoire d'un projectile avec $\theta$ = %.2f deg"% theta)
plt.xlabel('x [m]')
plt.ylabel('y [m]')
plt.ylim(ymin=0.0)
plt.grid()
# ENREGISTRER ET AFFICHER LA FIGURE
plt.savefig("tire.png")
plt.savefig("tire.pdf")
plt.show()
```

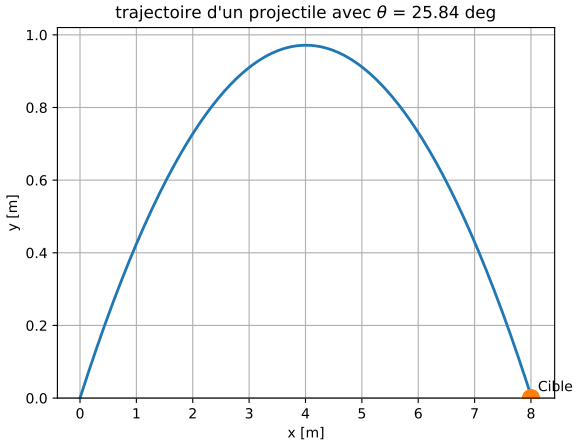


Figure 4: Résultats de l'exécution du programme tire.py avec une vitesse initiale $v_0 = 10\ m\ s^{-1}$ et l'emplacement cible $x_{cible} = 8$ m. L'angle requis pour atteindre la cible est $\theta = 25.84°$.

4 Convergence et de stabilité de la méthode d'Euler: Cas des systèmes linéaires

En mécanique classique, les équations du mouvement d'un système mécanique (systèmes de points matériels, système de solides) sont des équations différentielles du second ordre par rapport au temps. La connaissance des positions et des vitesses des points à l'instant $t = 0$ suffit à déterminer le mouvement pour $t > 0$.

Ces équations sont souvent non linéaires car les forces elles-mêmes le sont (par exemple la force de gravitation) et car l'accélération est souvent une fonction non linéaire des degrés de liberté. Dans ce cas, il est fréquent que l'on ne connaisse pas de solution analytique exacte. On est alors amené à rechercher une solution approchée par une méthode numérique.

Cette partie du cours explique le principe de ce type d'intégration numérique. On prendra l'exemple de l'oscillateur harmonique (dont la solution exacte est connue) auquel on appliquera la méthode numérique d'Euler. On abordera les notions importantes de *convergence* et de *stabilité*.

On verra aussi des variantes de la méthode d'Euler, qui peuvent être utilisées pour résoudre des systèmes conservatifs à N corps, par exemple en dynamique moléculaire.

De manière générale soit le système d'équations différentielles suivant:

$$\dot{\boldsymbol{u}} = f(\boldsymbol{u}) \tag{11}$$

Où $\boldsymbol{u}$ peut être un vecteur d'état et $f(\boldsymbol{u})$ peut être linéaire ou non linéaire.

Soit $f(\boldsymbol{u}) = \boldsymbol{A} \cdot \boldsymbol{u}$ avec $\boldsymbol{A}$ une matrice. Donc on peut écrire l'équation (4) comme suit:

$$\dot{\boldsymbol{u}} = \boldsymbol{A} \cdot \boldsymbol{u} \quad avec\ \boldsymbol{u}(t=0) = \boldsymbol{u}_0 \tag{12}$$

La solution analytique exacte d'un tel système est de la forme:

$$\boldsymbol{u}(t) = e^{\boldsymbol{A}t} \cdot \boldsymbol{u}_0 \tag{13}$$

On se propose d'appliquer différentes méthodes d'Euler au système (4).

4.1 La méthode d'Euler explicite (progressive)

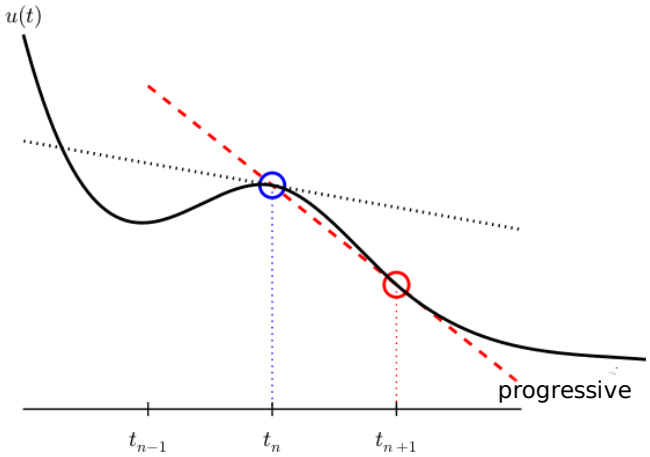


Figure 5: Illustration d'une approximation par différence progressive de la dérivée.

$$\frac{\boldsymbol{u}_{k+1} - \boldsymbol{u}_k}{\Delta t} \approx \dot{\boldsymbol{u}}_k = f(\boldsymbol{u}_k) \tag{14}$$

$$\boldsymbol{u}_{k+1} = \boldsymbol{u}_k + \Delta t f(\boldsymbol{u}_k) \tag{15}$$

Si $\dot{\boldsymbol{u}} = \boldsymbol{A} \cdot \boldsymbol{u}$ alors;

$$\boldsymbol{u}_{k+1} = \boldsymbol{u}_k + \Delta t \boldsymbol{A} \cdot \boldsymbol{u}_k = (\boldsymbol{I} + \Delta t \boldsymbol{A}) \cdot \boldsymbol{u}_k \tag{16}$$

Où $\boldsymbol{I}$ est la matrice identité.

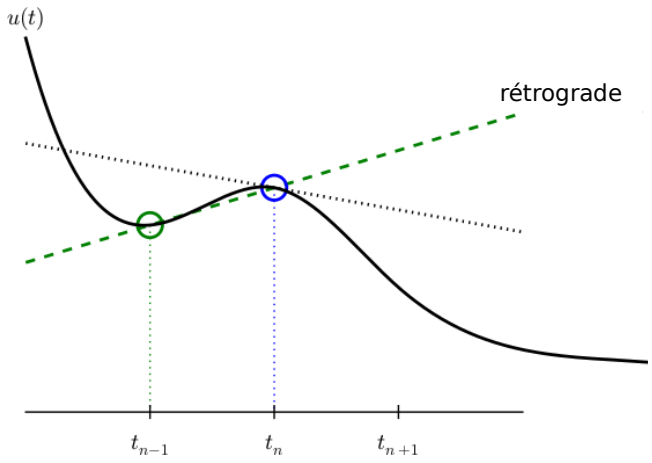


Figure 6: Illustration d'une approximation par différence rétrograde de la dérivée.

4.2 La méthode d'Euler implicite (rétrograde)

$$\frac{\boldsymbol{u}_{k+1} - \boldsymbol{u}_k}{\Delta t} \approx \dot{\boldsymbol{u}}_{k+1} = f(\boldsymbol{u}_{k+1}) \tag{17}$$

$$\boldsymbol{u}_{k+1} = \boldsymbol{u}_k + \Delta t f(\boldsymbol{u}_{k+1}) \tag{18}$$

Si $\dot{\boldsymbol{u}} = \boldsymbol{A} \cdot \boldsymbol{u}$ alors;

$$\boldsymbol{u}_{k+1} = \boldsymbol{u}_k + \Delta t \boldsymbol{A} \cdot \boldsymbol{u}_{k+1} \tag{19}$$

$$(\boldsymbol{I} - \Delta t \boldsymbol{A}) \cdot \boldsymbol{u}_{k+1} = \boldsymbol{u}_k \tag{20}$$

$$\boldsymbol{u}_{k+1} = (\boldsymbol{I} - \Delta t \boldsymbol{A})^{-1} \cdot \boldsymbol{u}_k \tag{21}$$

Où $\boldsymbol{I}$ est la matrice identité.

4.3 Exemple: Oscillateur libre amorti [masse, ressort, amortisseur]

Un bloc de masse m est lié à l'extrémité libre d'un ressort de raideur k , de longueur au repos l, de masse négligeable et d'élasticité parfaite, l'autre extrémité du ressort étant fixe. Le système est supposé dans l'espace (on néglige la force de pesanteur). Le seul mouvement possible pour le bloc est une translation suivant x; on assimilera le bloc à un point matériel M.

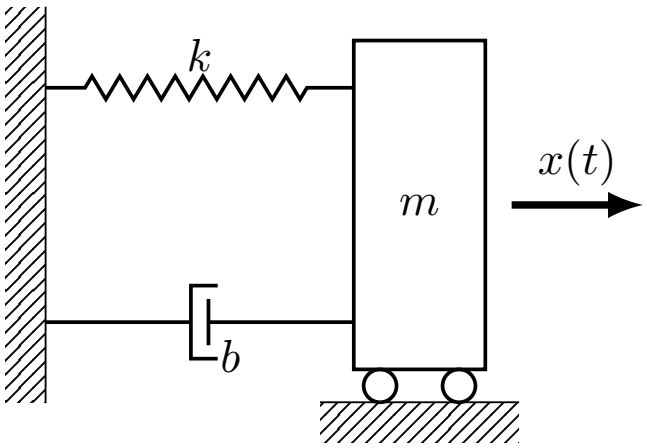


Figure 7: Schéma d'un système dynamique oscillant amorti unidimensionnel.

Bilan des forces.

- La force de rappel $\overrightarrow{F_r} = -k\overrightarrow{x}$ où k est un coefficient positif et $\overrightarrow{x}$ le vecteur position de M.
- Le système est amorti. L'amortissement de type visqueux est représenté par un amortisseur qui exerce la force dissipative (ou force d'amortissement visqueux) $\overrightarrow{F_a} = -b\overrightarrow{v}$ où b est un coefficient positif et $\overrightarrow{v}$ le vecteur vitesse de M.

Équation de mouvement. La deuxième loi de Newton pour le système peut être écrite avec l'accélération multipliée par la masse du côté gauche et la somme des forces du côté droit:

$$m\overrightarrow{a} = \overrightarrow{F_a} + \overrightarrow{F_r} \tag{22}$$

$$m\ddot{x} = -b\dot{x} - kx \tag{23}$$

$$m\ddot{x} + b\dot{x} + kx = 0 \tag{24}$$

On réécrit cette équation sous la forme canonique suivante :

$$\ddot{x} + 2\zeta\omega_0\dot{x} + \omega_0^2 x = 0 \tag{25}$$

avec $\omega_0 = \sqrt{\frac{k}{m}}$ désigne une *pulsation caractéristique* et $\zeta = \frac{b}{2\sqrt{km}}$ est une quantité positive sans dimension, appelée *taux d'amortissement*.

C'est une équation différentielle linéaire d'ordre 2 à coefficients constants.

On peut trouver numériquement la solution de l'équation (25) à l'aide des méthodes d'Euler à partir du système d'équations différentielles ordinaires suivant:

$$\dot{x} = v \tag{26}$$

$$\dot{v} = -2\zeta\omega_0 v - \omega_0^2 x \tag{27}$$

$$\tag{28}$$

$$\frac{d}{dt}\begin{pmatrix} x \\ v \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ -\omega_0^2 & -2\zeta\omega_0 \end{pmatrix} \cdot \begin{pmatrix} x \\ v \end{pmatrix} \tag{29}$$

L'équation (4.3) est de la forme: $\dot{\boldsymbol{u}} = \boldsymbol{A} \cdot \boldsymbol{u}$ avec:

$$\boldsymbol{A} = \begin{pmatrix} 0\&1 \\ -\omega_0^2 & -2\zeta\omega_0 \end{pmatrix}$$

et

$$\boldsymbol{u} = \begin{pmatrix} x \\ v \end{pmatrix}$$

Supposons que nous voulions résoudre le problème avec: $\omega_0 = 2\pi$, $\zeta = 0.25$, $\boldsymbol{u_0} = \begin{pmatrix} x(t=0) \\ v(t=0) \end{pmatrix} = \begin{pmatrix} 2 \\ 0 \end{pmatrix}$, $\Delta t = 0.01$ pour $t \in [0, 10]$. Ce sera une solution sinusoïdale amortie.

Solution avec la méthode d'Euler explicite. Nous implémentons l'expression explicite d'Euler montrée dans (16) dans le code python suivant:

```
## NOM DU PROGRAMME: OscillateurEulerExp.py
#% IMPORTATION
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d.axes3d import Axes3D
# SYSTÈME: OSCILLATEUR LIBRE AMORTI
w = 2*np.pi # fréquence propre
a = 0.25    # rapport d'amotissement
A = np.array([[0, 1], [-w**2, -2*a*w]])
dt = 0.01 # pas du temps
Tf = 10 # temps finale de la simulation
nsteps = int(Tf/dt)
# CONDITION INITIAL: à t = 0; x = 2, v = 0
u0 = np.array([2,0])
#%% ITÉRATION: EULER ExPLICITE
Texp = np.zeros(nsteps)
Uexp = np.zeros((2, nsteps))
Texp[0] = 0.0
```

```
Uexp[:,0] = u0
for k in range(nsteps-1):
    Texp[k+1] = Texp[k] + dt
    Uexp[:,k+1] = np.dot((np.eye(2) + dt * A), Uexp[:,k])

plt.figure(figsize=(10,5))
# PLOT POSITION vs TEMPS
plt.suptitle("Simulation d'un oscillateur libre amorti avec un pas d'intégration "+ r"$ \Delta t= %
             fontweight = "bold")
plt.subplot(1,2,1)
plt.plot(Texp,Uexp[0,:], linewidth=2, color ='k')
plt.xlabel("Temps")
plt.ylabel("Position")
plt.title("Trajectoire de la mass M (Euler explicite)")
# DIAGRAMME DE PHASE 2D
plt.subplot(1,2,2)
plt.plot(Uexp[0,:],Uexp[1,:], linewidth=2, color ='k')
plt.xlabel("Position")
plt.ylabel("Vitesse")
plt.title("Trajectoire de phase (Euler explicite)")
plt.savefig("EulerExp1D.png"); plt.savefig("EulerExp1D.pdf")
# DIAGRAMME DE PHASE 3D
plt.figure()
ax = plt.axes(projection="3d")
ax.plot(Texp, Uexp[0,:],Uexp[1,:], linewidth=2, color ='k')
ax.set_xlabel("Temps")
ax.set_ylabel("Position")
ax.set_zlabel("Vitesse")
ax.set_title("Trajectoire de phase (Euler explicite)")
plt.savefig("EulerExp3D.png"); plt.savefig("EulerExp3D.pdf")
plt.show()
```

La figure 8 est générée par le code `OscillateurEulerExp.py`, montrant la divergence et l'instabilité de la méthode Euler explicite. En effet, le pas d'intégration Δt agit considérablement sur la qualité de la simulation et donne un résultat inacceptable physiquement.

Dans le cas d'intégration avec la méthode d'Euler explicite, la figure 9 montre que nous avons un problème d'augmentation d'amplitude dans le cas d'un oscillateur non amorti (courbe bleue pour $\zeta = 0$). Plus le temps de simulation est long, plus l'amplitude augmente, ce qui n'est pas ce que nous attendons de l'évolution du système dans le temps. En d'autres termes, l'amplitude devrait être constante dans le temps pour un système oscillant non amorti.

Solution avec la méthode d'Euler implicite. Nous implémentons l'expression implicite d'Euler montrée dans (21) dans le code python suivant:

```
## NOM DU PROGRAMME: OscillateurEulerImp.py
#% IMPORTATION
import numpy as np
from numpy.linalg import inv
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d.axes3d import Axes3D
# SYSTÈME: OSCILLATEUR LIBRE AMORTI
w = 2*np.pi # fréquence propre
a = 0.25    # rapport d'amotissement
```

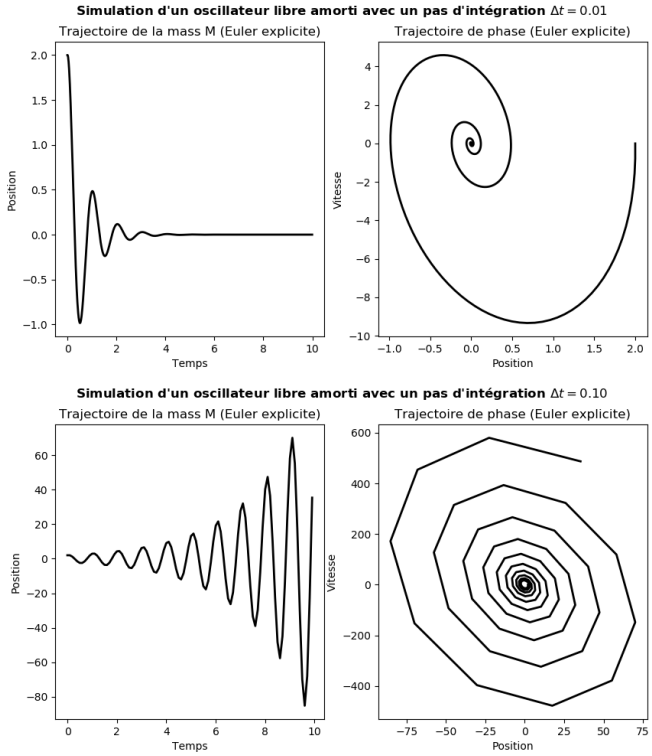


Figure 8: Simulation d'un système oscillant avec différents pas de temps; $\Delta t = 0.01$ et $\Delta t = 0.1$ et pour $\zeta = 0.25$.

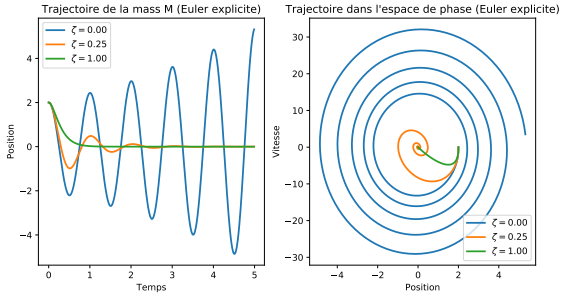


Figure 9: Simulation d'un système oscillant avec différentes valeurs de ζ et pour $\Delta t = 0.01$.

```
A = np.array([[0, 1], [-w**2, -2*a*w]])
dt = 0.1 # pas du temps
Tf = 10 # temps finale de la simulation
nsteps = int(Tf/dt)
# CONDITION INITIAL: à t = 0; x = 2, v = 0
```

```
u0 = np.array([2,0])
#%% ITÉRATION: EULER IMPLICITE
Timp = np.zeros(nsteps)
Uimp = np.zeros((2, nsteps))
Timp[0] = 0.0
Uimp[:,0] = u0
for k in range(nsteps-1):
    Timp[k+1] = Timp[k] + dt
    Uimp[:,k+1] = np.dot(inv(np.eye(2) - dt * A), Uimp[:,k])

plt.figure(figsize=(10,5))
# PLOT POSITION vs TEMPS
plt.suptitle("Simulation d'un oscillateur libre amorti avec un pas d'intégration "+ r"$ \Delta t= %
             fontweight = "bold")
plt.subplot(1,2,1)
plt.plot(Timp,Uimp[0,:], linewidth=2, color ='k')
plt.xlabel("Temps")
plt.ylabel("Position")
plt.title("Trajectoire de la mass M (Euler implicite)")
# DIAGRAMME DE PHASE 2D
plt.subplot(1,2,2)
plt.plot(Uimp[0,:],Uimp[1,:], linewidth=2, color ='k')
plt.xlabel("Position")
plt.ylabel("Vitesse")
plt.title("Trajectoire de phase (Euler implicite)")
plt.savefig("Eulerimp1D_2.png"); plt.savefig("Eulerimp1D_2.pdf")
# DIAGRAMME DE PHASE 3D
plt.figure()
ax = plt.axes(projection="3d")
ax.plot(Timp, Uimp[0,:],Uimp[1,:], linewidth=2, color ='k')
ax.set_xlabel("Temps")
ax.set_ylabel("Position")
ax.set_zlabel("Vitesse")
ax.set_title("Trajectoire de phase (Euler implicite)")
plt.savefig("Eulerimp3D_2.png"); plt.savefig("Eulerimp3D_2.pdf")
plt.show()
```

La figure 10 est générée par le code `OscillateurEulerImp.py`, montrant que la méthode d'Euler implicite est plus stable que la méthode Euler explicite. Nous remarquons toujours qu' il y a un effet du changement du pas d'intégration Δt sur la qualité de la simulation mais le résultat du calcul est désormais acceptable physiquement.

Même problème avec l'amplitude pour le cas d'intégration avec la méthode implicite d'Euler, la figure 11 montre que nous avons un problème de diminution d'amplitude dans le cas d'un oscillateur non amorti (courbe bleue pour $\zeta = 0$). Comme indiqué ci-dessus, l'amplitude devrait être constante dans le temps pour un système oscillant non amorti.

4.4 Conclusion

La conclusion ici est que la méthode Euler implicite est plus stable que celle explicite. Les deux méthodes posent un problème fondamental avec ses amplitudes croissantes et décroissantes, pour le cas d'oscillateur libre non amorti, et qu'un très petit Δt est nécessaire pour obtenir des résultats satisfaisants. Plus la simulation est longue, plus Δt doit être petit. Il est certainement temps de

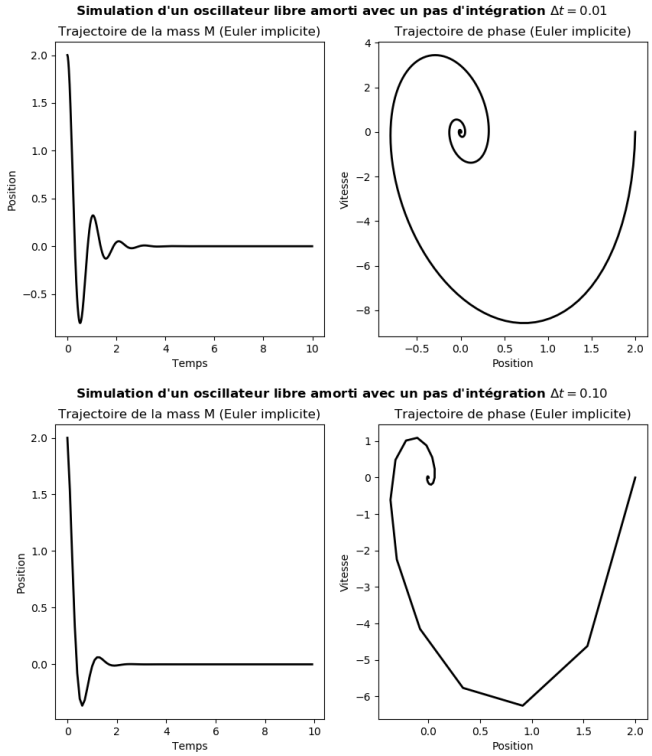


Figure 10: Simulation d'un système oscillant avec différents pas de temps; $\Delta t = 0.01$ et $\Delta t = 0.1$ et pour $\zeta = 0.25$.

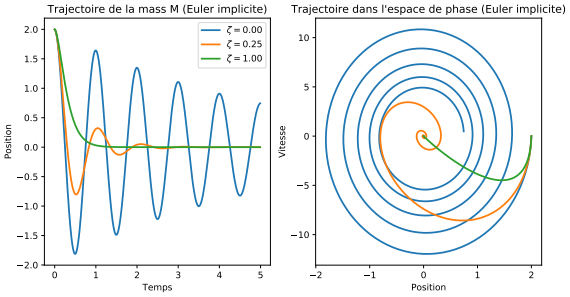


Figure 11: Simulation d'un système oscillant avec différentes valeurs de ζ et pour $\Delta t = 0.01$.

rechercher des méthodes numériques plus stables et plus efficaces tels que les méthodes de Runge-Kutta.