Résolution approchée d'équations différentielles

I. Principe général

Soit $n \in \mathbb{N}^*$ et U un ouvert de $\mathbb{R} \times \mathbb{R}^n$. Une équation différentielle d'ordre 1 (résolue) est une équation de la forme

$$y'(t) = F(t, y(t))$$

où F est une application continue sur U à valeurs dans $\mathbb{R}^n$ et où la fonction inconnue y est de classe $\mathscr{C}^1$ sur un certain intervalle de $\mathbb{R}$, à valeurs dans $\mathbb{R}^n$.
Dans le cas où $n = 1$, on parle d'une équation différentielle scalaire ; dans le cas général, on obtient un système différentiel.
Si $(t_0, y_0) \in U$, résoudre le problème de Cauchy en (t_0, y_0), c'est trouver un couple (I, y) où I est un intervalle de $\mathbb{R}$ et où y est une fonction de classe $\mathscr{C}^1$ de I dans $\mathbb{R}^n$ tels que

$$y(t_0) = y_0 \quad \text{et} \quad \forall t \in I\ ,\ y'(t) = F(t, y(t))\,.$$

On peut démontrer (théorème de Cauchy-Lipschitz) que si pour tout t l'application $y \mapsto F(t, y)$ est de classe $\mathscr{C}^1$ (c'est-à-dire si toutes ses dérivées partielles existent et sont continues), alors il existe une unique solution maximale (I, y) au problème de Cauchy.

Les méthodes les plus simples de résolution approchée d'une telle équation différentielle consistent à approcher la solution y en un certain nombre de points de l'intervalle I. Plus précisément, on se donne un pas h, on part du point t_0 (où l'on connaît la valeur de y, c'est y_0), puis on calcule de proche en proche les valeurs de y aux points $t_k = t_0 + kh$ $(k \in \mathbb{N})$ en utilisant la relation

$$y(t_{k+1}) = y(t_k) + \int_{t_k}^{t_{k+1}} y'(u)\,\mathrm{d}u = y(t_k) + \int_{t_k}^{t_{k+1}} F(u, y(u))\,\mathrm{d}u\,.$$

Suivant la méthode de calcul approché de l'intégrale ci-dessus, on obtient diverses méthodes.

Remarques

- On ne traitera ici que des méthodes les plus courantes.
- On ne s'intéressera ici qu'à la partie programmation, et on ne discutera pas des problèmes posés par ces méthodes dans certains cas (convergence, précision, stabilité). C'est un sujet intéressant, mais qui prendrait beaucoup trop de temps.

II. Les équations scalaires d'ordre 1

II.1. Les méthodes explicites

II.1.1. La méthode d'Euler explicite

Cette méthode utilise la méthode des rectangles à gauche pour l'approximation de l'intégrale. Cela revient donc à écrire, avec les notations ci-dessus :

$$y(t_{k+1}) = y(t_k) + \int_{t_k}^{t_{k+1}} F(u, y(u))\,\mathrm{d}u \approx y(t_k) + hF(t_k, y(t_k))$$

(puisque $t_{k+1} - t_k = h$).
Dans le cas d'un équation différentielle scalaire, cela revient à faire une approximation de la courbe au point d'abscisse t_k par sa tangente en ce point. En effet l'équation de la tangente à la courbe au point d'abscisse t_k est :

$$Y = (X - t_k)y'(t_k) + y(t_k)$$

Erreur commise : si l'on se place sur un intervalle $[a\,;b]$ (avec $a = t_0$) subdivisé de façon régulière en n intervalles, soit $h = \dfrac{b-a}{n}$, l'erreur commise est un $O\left(\dfrac{1}{n}\right)$, soit $O(h)$. La méthode d'Euler est dite d'ordre 1.

☞ *Démonstration:*

Supposons F de classe $\mathscr{C}^1$. Alors la solution y est de classe $\mathscr{C}^2$. L'erreur e_k commise à la k-ième étape est :

$$e_k = \left|y(t_{k+1}) - \left[y(t_k) + hF(t_k, y(t_k))\right]\right| = \left|y(t_{k+1}) - y(t_k) - hy'(t_k)\right|,$$

et d'après l'inégalité de Taylor-Lagrange : $e_k \leqslant \dfrac{1}{2}h^2 \left\|y''\right\|_\infty$. L'erreur totale commise est alors :

$$\varepsilon = \sum_{k=0}^{n-1} e_k \leqslant \frac{n}{2} h^2 \left\| y'' \right\|_\infty = O\left(\frac{1}{n}\right).$$

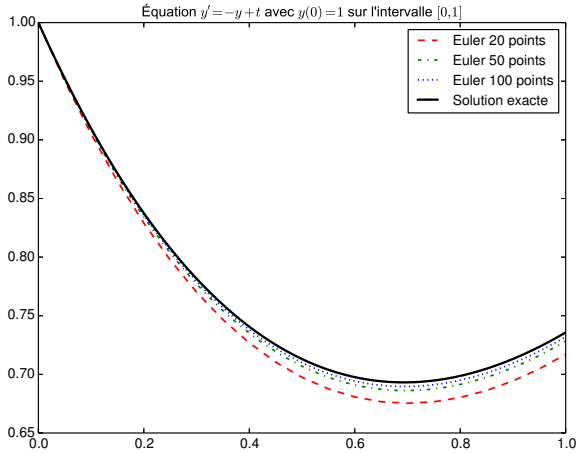
1er programme

1. Écrire une fonction `Euler(F,a,b,y0,n)` qui renvoie les deux listes formées des t_k et des $y(t_k)$.
2. On fera ensuite un test avec l'équation différentielle $y' = -y + t$ et la condition initiale $y(0) = 1$, dont la solution exacte est $t \mapsto 2\mathrm{e}^{-t} + t - 1$. Sur un même graphique on représentera les solutions obtenues sur l'intervalle $[0, 1]$ avec successivement 20, 50 et 100 points, ainsi que la solution exacte.

Corrigé :

```python
import numpy as np
import matplotlib.pyplot as plt
import math

def Euler(F, a, b, y_0, n):
    pas = (b - a)/n
    X = [a]
    t = a
    Y = [y_0]
    y = y_0
    for k in range(0, n):
        y = y + pas * F(t, y)
        t += pas
        X.append(t)
        Y.append(y)
    return X, Y

if __name__ == "__main__":
    F = lambda t, y: - y + t
    a = 0
    b = 1
    y_0 = 1
    # tracés approchés
    X, Y1 = Euler(F, a, b, y_0, 20)
    plt.plot(X, Y1, color='red',  linewidth = 1.5, linestyle = '--', label='Euler 20 points')
    X, Y2 = Euler(F, a, b, y_0, 50)
    plt.plot(X, Y2, color='green',  linewidth = 1.5, linestyle = '-.', label='Euler 50 points')
    X, Y3 = Euler(F, a, b, y_0, 100)
    plt.plot(X, Y3, color='blue',  linewidth = 1.5, linestyle = ':',  label='Euler 100 points')
    # tracé exact
    X = np.linspace(a, b, 1000)
    Z = 2 * np.exp(-X) + X - 1
    # calcul directement sur arrays; pour les listes il faudrait faire
    # Z = list(map(lambda t: 2 * m.exp(-t) + t - 1, X))
    plt.plot(X, Z, color='black', linewidth=2, label='Solution exacte')
    # ornements
    plt.xlim(a, b)
    plt.legend(fontsize = 'medium', loc='upper right')
    plt.title('Équation $y\'=-y+t$ avec $y(0)=1$ sur l\'intervalle $[0,1]$',\
              fontsize='medium')

    plt.show()
```

Voici la figure obtenue :



II.2. La méthode du point milieu (ou méthode de Heun)

Cette méthode utilise la méthode des rectangles au point milieu pour l'approximation de l'intégrale. Cela revient donc à écrire, avec les notations ci-dessus :

$$y(t_{k+1}) = y(t_k) + \int_{t_k}^{t_{k+1}} F(u, y(u))\,\mathrm{d}u \approx y(t_k) + hF\left(\frac{t_k+t_{k+1}}{2}, y\left(\frac{t_k+t_{k+1}}{2}\right)\right).$$

Mais à la k-ième étape, on ne connaît que la valeur approchée de $y(t_k)$ et non celle de $y\left(\frac{t_k+t_{k+1}}{2}\right)$; on approchera donc cette valeur à l'aide de la méthode d'Euler :

$$y\left(\frac{t_k+t_{k+1}}{2}\right) \approx y(t_k) + \frac{h}{2}F(t_k, y(t_k))$$

ce qui donne finalement le schéma :

$$y(t_{k+1}) \approx y(t_k) + hF\left(t_k + \frac{h}{2}, \alpha_k\right) \quad \text{avec} \quad \alpha_k = y(t_k) + \frac{h}{2}F(t_k, y(t_k)).$$

Erreur commise : si l'on se place sur un intervalle $[a,b]$ (avec $a = t_0$) subdivisé de façon régulière en n intervalles, soit $h = \dfrac{b-a}{n}$, l'erreur commise est un $O\left(\dfrac{1}{n^2}\right)$, soit un $O(h^2)$. La méthode de Heun est dite d'ordre 2.

☞ *Démonstration:*

Supposons F de classe $\mathscr{C}^2$. Alors la solution y est de classe $\mathscr{C}^3$. L'erreur e_k commise à la k-ième étape est ici :

$$e_k = |y(t_{k+1}) - y(t_k) - hd_k| \quad \text{avec} \quad d_k = F\left(t_k + \frac{h}{2}, y(t_k) + \frac{h}{2}F(t_k, y(t_k))\right).$$

Puisque d_k est une approximation de $y'\left(t_k + \frac{h}{2}\right)$, on écrira plutôt :

$$e_k \leqslant e'_k + e''_k \quad \text{avec} \quad \begin{cases} e'k = \left|y(t_{k+1}) - y(t_k) - hy'\left(t_k + \frac{h}{2}\right)\right| \\ e''_k = h\left|y'\left(t_k + \frac{h}{2}\right) - d_k\right| \end{cases}$$

En utilisant la formule de Taylor-Young pour y à l'ordre 3 et y' à l'ordre 2 :

$$\begin{aligned} y(t_{k+1}) - y(t_k) - hy'\left(t_k + \frac{h}{2}\right) &= \left[y(t_{k+1}) - y(t_k) - hy'(t_k)\right] - h\left[y'\left(t_k + \frac{h}{2}\right) - y'(t_k)\right] \\ &= \left[\frac{h^2}{2}y''(t_k) + O(h^3)\right] - h\left[\frac{h}{2}y''(t_k) + O(h^2)\right] = O(h^3), \end{aligned}$$

donc $e'_k = O(h^3)$. D'autre part, en appliquant la formule des accroissements finis à la deuxième application partielle de F :

$$\begin{aligned} \left|y'\left(t_k + \frac{h}{2}\right) - d_k\right| &= \left|F\left(t_k + \frac{h}{2}, y\left(t_k + \frac{h}{2}\right)\right) - F\left(t_k + \frac{h}{2}, y(t_k) + \frac{h}{2}F(t_k, y(t_k))\right)\right| \\ &= \left|\left[y\left(t_k + \frac{h}{2}\right) - y(t_k) - \frac{h}{2}F(t_k, y(t_k))\right]\frac{\partial F}{\partial y}\left(t_k + \frac{h}{2}, c_k\right)\right| \\ &\leqslant \left|y\left(t_k + \frac{h}{2}\right) - y(t_k) - \frac{h}{2}F(t_k, y(t_k))\right| \left\|\frac{\partial F}{\partial y}\right\|_\infty \end{aligned}$$

où c_k appartient au segment $\left[y\left(t_k + \frac{h}{2}\right), y(t_k) + \frac{h}{2}F(t_k, y(t_k))\right]$. En utilisant encore la formule de Taylor-Young, on obtient alors $\left|y'\left(t_k + \frac{h}{2}\right) - d_k\right| = O(h^2)$ et finalement $e''_k = O(h^3)$.

On en déduit $e_k = O(h^3) = O\left(\dfrac{1}{n^3}\right)$ et l'erreur totale ε est bien un $O\left(\dfrac{1}{n^2}\right)$.

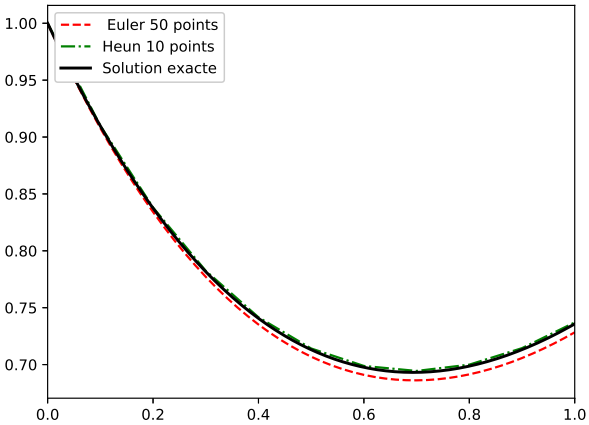
2ème programme

Implémenter la méthode (il suffit de modifier légèrement le programme précédent). Comparer.

Corrigé :

```python
import numpy as np
import matplotlib.pyplot as plt
import math

from P1EquationDifferentielleScalaireEuler import *

def Heun(F, a, b, y_0, n):
    pas = (b - a)/n
    X = [a]
    t = a
    Y = [y_0]
    y = y_0
    for k in range(0, n):
        alpha = y + pas * F(t, y) / 2
        y = y + pas * F(t+ pas / 2, alpha)
        t += pas
        X.append(t)
        Y.append(y)
    return X, Y

if __name__ == "__main__":
    F = lambda t,y: - y + t
    a = 0
    b = 1
    y_0 = 1
    X, Y1 = Euler(F, a, b, y_0, 50)
    plt.plot(X, Y1, color='red',  linewidth = 1.5, linestyle = '--', label=' Euler 50 points')
    X, Y2 = Heun(F, a, b, y_0, 10)
    plt.plot(X, Y2, color='green', linewidth = 1.5, linestyle = '-.', label='Heun 10 points')
    X = np.linspace(a, b, 1000) # tracé exact
    Z = 2 * np.exp(-X) + X - 1
    plt.plot(X, Z, color='black', linewidth=2, label='Solution exacte')
    plt.xlim(a, b)
    plt.legend(loc='upper left')

    plt.show()
```

Ci-dessous la figure obtenue.



II.3. Les méthodes implicites

II.3.1. La méthode d'Euler implicite

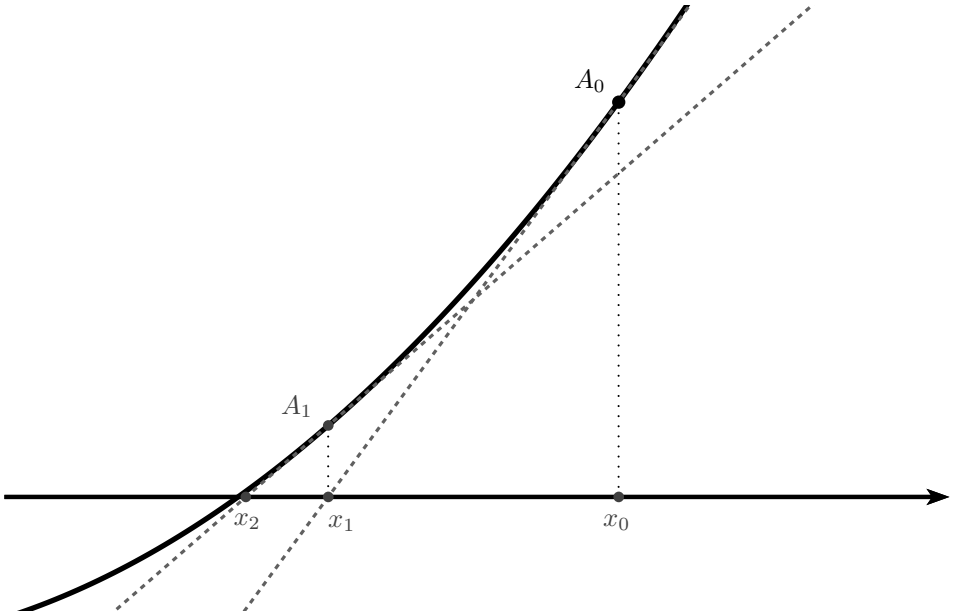
Cette méthode utilise la méthode des rectangles à droite pour l'approximation de l'intégrale. Cela revient donc à écrire, avec les notations ci-dessus :

$$y(t_{k+1}) = y(t_k) + \int_{t_k}^{t_{k+1}} F(u, y(u))\,\mathrm{d}u \approx y(t_k) + hF(t_{k+1}, y(t_{k+1}))$$

(puisque $t_{k+1} - t_k = h$).
$y(t_{k+1})$ est alors solution de l'équation $G(x) = 0$ où

$$G\colon x \mapsto y(t_k) + hF(t_{k+1}, x) - x\,.$$

On peut montrer que cette équation possède une unique solution pour h « assez petit ». On peut alors la résoudre par la méthode de Newton, illustrée ci-dessous.



La suite correspondante est donnée par la relation de récurrence :

$$x_{n+1} = x_n - \frac{G(x_n)}{G'(x_n)}\,.$$

et l'on approchera : $G'(x_n) \approx \dfrac{G(x_n + h) - G(x_n)}{h}$, ou mieux : $G'(x_n) \approx \dfrac{G(x_n + h) - G(x_n - h)}{2h}$ (je vous laisse justifier à l'aide de la formule de Taylor).

Erreur commise : si l'on se place sur un intervalle $[a\,;b]$ (avec $a = t_0$) subdivisé de façon régulière en n intervalles, soit $h = \dfrac{b-a}{n}$, on montre que l'erreur commise est encore un $O\left(\dfrac{1}{n}\right)$, soit un $O(h)$. L'avantage de la méthode par rapport à la méthode explicite est qu'elle est plus stable, c'est-à-dire risque moins de diverger.

3ème programme

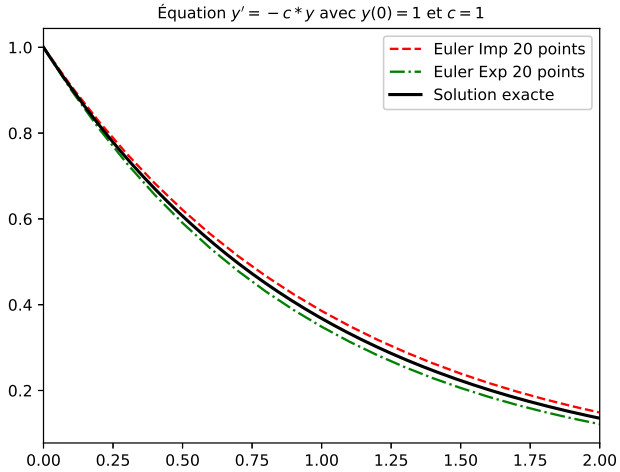
⌇ Implémenter la méthode, et comparer avec la méthode explicite.

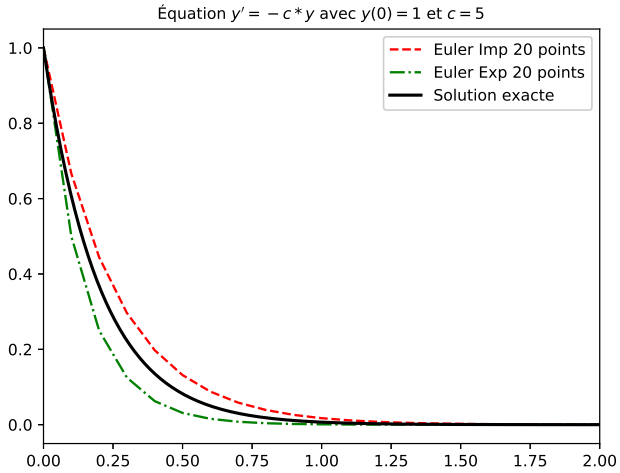
Corrigé :

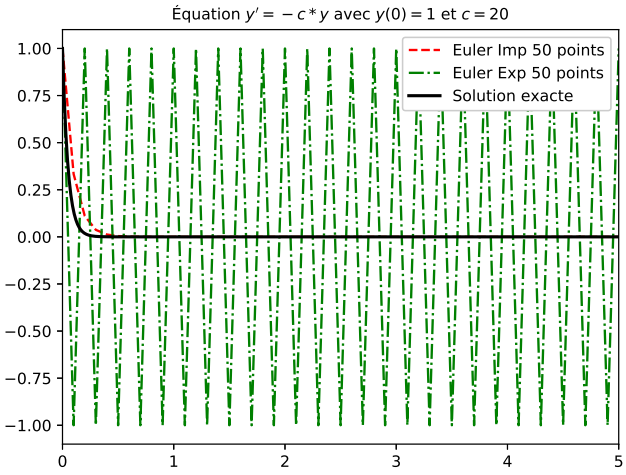
```
1  import numpy as np
2  import matplotlib.pyplot as plt
3  import math
4
5  def Newton(G, x0, pas):
6      err = 1
7      eps = 1e-10
8      while err > eps:
```

```
9           Gprime = (G(x0 + pas) - G(x0))/pas
10          x1 = x0 - G(x0)/Gprime
11          err = abs(x1 - x0)
12          x0 = x1
13      return x0
14
15  def EulerImplicite(F, a, b, y_0, n):
16      pas = (b - a)/n
17      X = [a]
18      t = a
19      Y = [y_0]
20      y = y_0
21      for k in range(0, n):
22          G = lambda x: y + pas * F( t + pas, x) - x
23          y = Newton(G, y, pas)
24          t += pas
25          X.append(t)
26          Y.append(y)
27      return X, Y
28
29  def EulerExplicite(F, a, b, y_0, n):
30      pas = (b - a)/n
31      X = [a]
32      t = a
33      Y = [y_0]
34      y = y_0
35      for k in range(0, n):
36          y = y + pas * F(t, y)
37          t += pas
38          X.append(t)
39          Y.append(y)
40      return X, Y
41
42  if __name__ == "__main__":
43      c = 20
44      nbpoints = 50
45      plt.figure(1)
46      F = lambda t,y: - c*y
47      a = 0
48      b = 3
49      y_0 = 1
50      X, Y1 = EulerImplicite(F, a, b, y_0, nbpoints)
51      plt.plot(X, Y1, color='red',  linewidth = 1.5, linestyle = '--', label='Euler Imp '+str(nbpoints)+' points')
52      X, Y2 = EulerExplicite(F, a, b, y_0, nbpoints)
53      plt.plot(X, Y2, color='green',  linewidth = 1.5, linestyle = '-.', label='Euler Exp '+str(nbpoints)+' points')
54
55      X = np.linspace(a, b, 1000) # tracé exact
56      Z = np.exp(-c*X)
57      plt.plot(X, Z, color='black', linewidth=2, label='Solution exacte')
58      plt.xlim(a, b)
59      plt.legend(fontsize = 'medium', loc='upper right')
60      plt.title('Équation $y\'=-c*y$ avec $y(0)=1$ et $c =$'+str(c),\
61                fontsize='medium')
62      plt.show()
```

Les figures ci-dessous montrent bien que la méthode d'Euler explicite n'est pas stable (ce serait évidemment meilleur avec plus de points !).







II.3.2. La méthode de Crank-Nicolson

Cette méthode utilise la méthode des trapèzes pour l'approximation de l'intégrale. Cela revient donc à écrire, avec les notations ci-dessus :

$$y(t_{k+1}) = y(t_k) + \int_{t_k}^{t_{k+1}} F(u, y(u))\,\mathrm{d}u \approx y(t_k) + \frac{h}{2}\left(F(t_k, y(t_k)) + F(t_{k+1}, y(t_{k+1}))\right).$$

$y(t_{k+1})$ est alors solution de l'équation $G(x) = 0$ où

$$G\colon x \mapsto y(t_k) + \frac{h}{2}\left(F(t_k, y(t_k)) + F(t_{k+1}, x)\right) - x\,,$$

que l'on résout encore par la méthode de Newton. On démontre qu'il s'agit d'une méthode d'ordre 2.

4ème programme

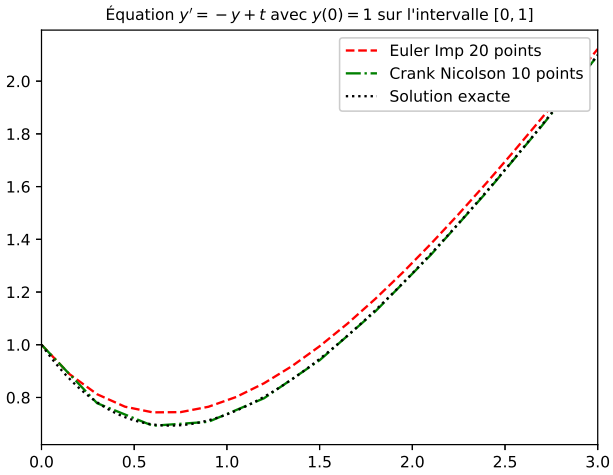
⌇ Implémenter la méthode, et comparer avec la méthode précédente.

Corrigé :

Le seul changement dans le code est le suivant :

```
def Crank(F, a, b, y_0, n):
    pas = (b - a)/n
    X = [a]
    t = a
    Y = [y_0]
    y = y_0
    for k in range(0, n):
        G = lambda x: y + pas/2 * (F( t + pas, x) + F(t,y)) - x
        y = Newton(G, y, pas)
        t += pas
        X.append(t)
        Y.append(y)
    return X, Y
```

Voici le résultat.



III. Les équations vectorielles (systèmes différentiels)

Tous les algorithmes vus ci-dessus s'adaptent directement au cas où la fonction F est définie sur $I \times \mathbb{R}^n$ et à valeurs dans $\mathbb{R}^n$. La solution de l'équation différentielle est alors une fonction vectorielle y définie sur I et à valeurs dans $\mathbb{R}^n$.

Pour la programmation, il suffira donc d'utiliser des variables de type `array` (module `numpy`).

III.1. Un exemple : équations de Lotka-Volterra (système proie-prédateur)

On s'intéresse à l'évolution au cours du temps d'un système biologique composé de deux espèces : des proies (lapins ou sardines) et des prédateurs (renards ou requins, respectivement !).
Pour cela, on note $x(t)$ et $y(t)$ le nombre de proies et de prédateurs au temps t. Pour la suite, on fera l'hypothèse de supposer ces fonctions dérivables.

En l'absence de prédateurs, les proies ont un taux de croissance constant (on suppose la nourriture abondante et l'absence de compétition), c'est-à-dire que le taux d'accroissement de la population $x'(t)$ vérifie $\dfrac{x'(t)}{x(t)} = a$ où a représente la différence entre le taux de natalité et le taux de mortalité naturelle hors prédateurs (modèle de Malthus).

De même, les prédateurs ont tendance à disparaître en l'absence de proies, faute de nourriture, de façon proportionnelle à leur nombre donc on aura une relation de la forme $\dfrac{y'(t)}{y(t)} = -c$.

Il reste à prendre en compte les interactions entre les deux espèces : le taux de prédation (taux de décroissance des proies dû aux prédateurs) est supposé proportionnel au nombre de prédateurs. De la même façon, le taux de variation du nombre de prédateurs est proportionnel à la quantité de nourriture à leur disposition, c'est-à-dire au nombre de proies. Ces considérations nous conduisent aux équations suivantes :

$$\begin{cases} \dfrac{x'(t)}{x(t)} &= a - by \\ \dfrac{y'(t)}{y(t)} &= -c + dx \end{cases} \qquad a, b, c, d > 0\,.$$

On obtient donc un système différentiel ; les conditions initiales correspondent aux populations à l'instant $t = 0$.

En notant $Y(t) = \begin{pmatrix} x(t) \\ y(t) \end{pmatrix}$, ce système peut aussi s'écrire sous la forme $Y'(t) = F(t, Y)$ où F est l'application de $\mathbb{R}_+ \times \mathbb{R}^2$ dans $\mathbb{R}^2$ définie par $F(t, Y) = \begin{pmatrix} x(a - by) \\ y(-c + dx) \end{pmatrix}$. Cette fonction F étant évidemment de classe $\mathscr{C}^1$, le théorème de Cauchy-Lipschitz s'applique et assure de l'existence et de l'unicité d'une solution.

5ème programme

Utiliser la méthode d'Euler pour résoudre le système différentiel précédent. Que constate-t-on ?

Utiliser alors la méthode de Heun.

Pour obtenir les figures ci-dessous, on pris les valeurs

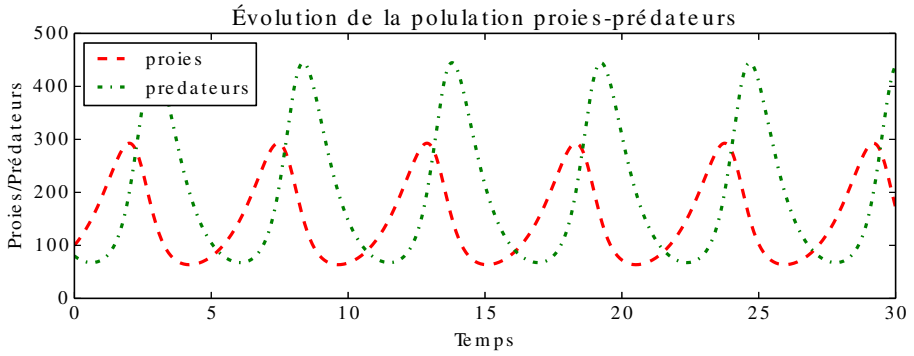
$$a = 1\ ,\ b = 0.005\ ,\ c = 1.5\ ,\ d = 0.01 \text{ et les conditions initiales } x_0 = 100\ ,\ y_0 = 80\,.$$

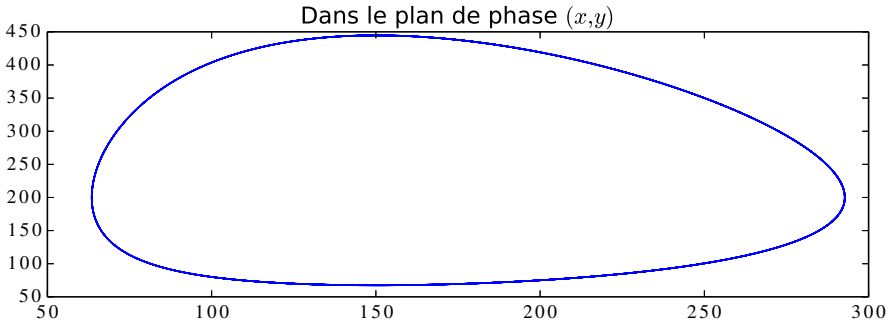
Corrigé :

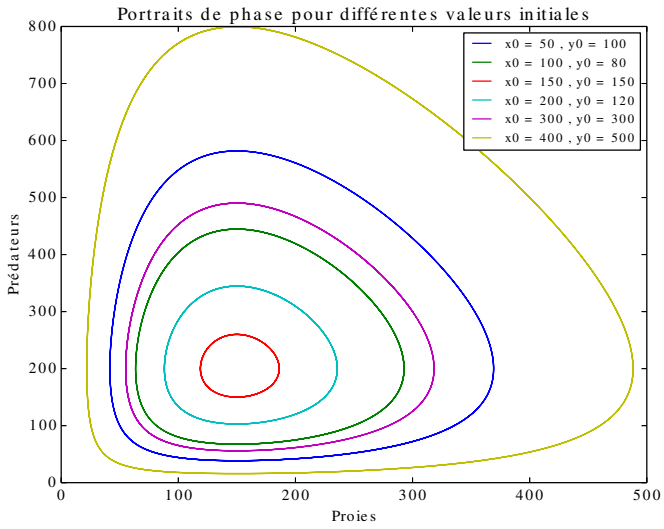
```python
import numpy as np
import matplotlib.pyplot as plt

def Heun(F, a, b, y_0, n):
    pas = (b - a)/n
    X = [a]
    t = a
    Y = [y_0]
    y = y_0
    for k in range(0, n):
        alpha = y + pas * F(t, y) / 2
        y = y +pas * F(t + pas / 2, alpha)
        t += pas
        X.append(t)
        Y.append(y)
    return X, Y

if __name__ == "__main__":

    def F(t,y):
        # a,b,c,d ici variables locales
        a = 1
        b = 0.005
        c = 1.5
        d = 0.01
        return np.array([ y[0] * (a - b * y[1]), y[1] * (d * y[0] - c)])

    a = 0
    b = 30
    y_0 = np.array([100, 80])
    nb_points = 10000
    X, Y = Heun(F, a, b, y_0, nb_points)

    fig1 = plt.figure(figsize = (9,7), num = 1)
    plt.subplots_adjust(hspace=0.5)
    plt.subplot(2,1,1)
    Y1 = [ Y[i][0] for i in range(0, nb_points+ 1)]
    plt.plot(X, Y1, color='red',  linestyle = '--', linewidth=2, label='proies')
```

```
    Y2 = [ Y[i][1] for i in range(0, nb_points+ 1)]
    plt.plot(X, Y2, color='green',  linestyle = '-.', linewidth=2, label='predateurs')
    plt.xlim(a, b)
    plt.ylim(0,500)
    plt.legend(loc='upper left', fontsize = 'medium')
    plt.title("Évolution de la population proies-prédateurs")
    plt.xlabel('Temps')
    plt.ylabel('Proies/Prédateurs')

    plt.subplot(2,1,2) # plan de phase
    plt.title("Dans le plan de phase $(x,y)$")
    plt.plot(Y1, Y2)

    plt.figure(2)
    #portraits de phase pour différentes valeurs initiales
    a = 0
    b = 20
    nb_points=10000
    # liste des valeurs initiales
    liste=[ np.array([50,100]), np.array([100,80]), np.array([150,150]), np.array([200,120]), \
            np.array([300,300]), np.array([400,500])]
    for y_0 in liste:
      X, Y = Heun(F, a, b, y_0, nb_points)
      Y1 = [ Y[i][0] for i in range(0, nb_points+ 1)]
      Y2 = [ Y[i][1] for i in range(0, nb_points+ 1)]
      plt.plot(Y1,Y2, label="x0 = "+str(y_0[0])+" , y0 = "+str(y_0[1]))
    plt.title("Portraits de phase pour différentes valeurs initiales")
    plt.legend(loc='upper right', fontsize='small')
    plt.xlabel('Proies')
    plt.ylabel('Prédateurs')
    plt.show()
```

Voici les figures obtenues.







6ème programme

Écrire une version qui n'utilise pas `numpy`, en gérant les deux coordonnées séparément.

Corrigé :

```
import matplotlib.pyplot as plt

def Heun(f, g, t0, t1, x0, y0, n):
    pas = (t1 - t0)/n
    T = [t0]
    t = t0
    X = [x0]
    x = x0
    Y = [y0]
    y = y0
    for k in range(0, n):
        alpha1 = x + pas * f(x, y) / 2
        alpha2 = y + pas * g(x, y) / 2
        x += pas * f(alpha1, alpha2)
        y += pas * g(alpha1, alpha2)
        X.append(x)
        Y.append(y)
        t += pas
        T.append(t)
    return T, X, Y

#a,b,c,d variables globales
a = 1
b = 0.005
c = 1.5
d = 0.01

def f(x, y): # 1ère équa diff x' = ...
    return a*x - b*x*y
def g(x, y): #2ème équa diff y' = ...
    return -c*y + d*x*y

x0 = 100
y0 = 80
t0 = 0
t1 = 30
nb_points = 10000
T, X, Y = Heun(f, g, t0, t1, x0, y0, nb_points)

fig1 = plt.figure(figsize = (9,7), num = 1)
plt.subplots_adjust(hspace=0.5)
plt.subplot(2,1,1)
plt.plot(T, X, color='red',  linestyle = '--', linewidth=2, label='proies')
```

```python
plt.plot(T, Y, color='green',  linestyle = '-.', linewidth=2, label='predateurs')
plt.xlim(t0, t1)
plt.ylim(0,500)
plt.legend(loc='upper left', fontsize = 'medium')
plt.title("Évolution de la population proies-prédateurs")
plt.xlabel('Temps')
plt.ylabel('Proies/Prédateurs')
plt.subplot(2,1,2) # plan de phase
plt.title("Dans le plan de phase $(x,y)$")
plt.plot(X, Y)

plt.figure(2)
#portraits de phase pour différentes valeurs initiales
t0 = 0
t1 = 20
nb_points=10000
# liste des valeurs initiales
liste=[ [50,100], [100,80], [150,150], [200,120], [300,300], [400,500]]
for CondInit in liste:
    x0, y0 = CondInit
    T, X, Y = Heun(f, g, t0, t1, x0, y0, nb_points)
    plt.plot(X, Y, label="x0 = "+str(x0)+" , y0 = "+str(y0))
plt.title("Portraits de phase pour différentes valeurs initiales")
plt.legend(loc='upper right', fontsize='small')
plt.xlabel('Proies')
plt.ylabel('Prédateurs')
plt.show()
```

III.2. Équations différentielles scalaires du second ordre

III.2.1. Un peu de théorie générale :

Considérons une équation différentielle linéaire scalaire du second ordre, de la forme :

$$x'' = a(t)x' + b(t)x + c(t) \quad (L)$$

où a, b et c sont des applications continues de I dans $\mathbb{K}$.
Matriciellement, cette équation peut s'écrire :

$$\begin{pmatrix} x'(t) \\ x''(t) \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ b(t) & a(t) \end{pmatrix} \begin{pmatrix} x(t) \\ x'(t) \end{pmatrix} + \begin{pmatrix} 0 \\ c(t) \end{pmatrix},$$

soit :

$$Y' = AY + B$$

avec

$$Y(t) = \begin{pmatrix} x(t) \\ x'(t) \end{pmatrix} \quad , \quad A(t) = \begin{pmatrix} 0 & 1 \\ b(t) & c(t) \end{pmatrix} \quad \text{et} \quad B(t) = \begin{pmatrix} 0 \\ c(t) \end{pmatrix}$$

ou encore $Y' = F(t, Y)$ avec $F(t, Y) = A(t).Y + B(t)$. Pour tout t, l'application $Y \mapsto A(t).Y + B(t)$ est de classe $\mathscr{C}^1$, donc le théorème de Cauchy-Lipschitz s'applique.
Pour résoudre cette équation, on pourra là encore utiliser les programmes vus précédemment.

III.2.2. Équations de Bessel

Les équations de Bessel sont des équations différentielles linéaires du second ordre de la forme

$$t^2x''(t) + tx'(t) + (t^2 - \alpha^2)x(t) = 0 \quad \text{où } \alpha \text{ est une constante.}$$

Leurs solutions (fonctions de Bessel) interviennent dans de nombreux domaines en Physique comme :

- la propagation de la chaleur dans un cylindre ;
- les ondes (électromagnétiques ou acoustiques) dans un guide cylindrique (antenne ou tuyau) ;
- les modes de vibration d'une fine membrane circulaire ou annulaire ;
- l'étude d'instruments optiques ;
- le pendule de Bessel ;
- dans les phénomènes de diffraction par une fente circulaire ;

En posant $Y(t) = \begin{pmatrix} x(t) \\ x'(t) \end{pmatrix}$, cette équation s'écrit aussi

$$Y'(t) = A(t)Y(t) \quad \text{avec} \quad A(t) = \begin{pmatrix} 0 & 1 \\ \frac{\alpha^2}{t^2} - 1 & -\frac{1}{t} \end{pmatrix}.$$

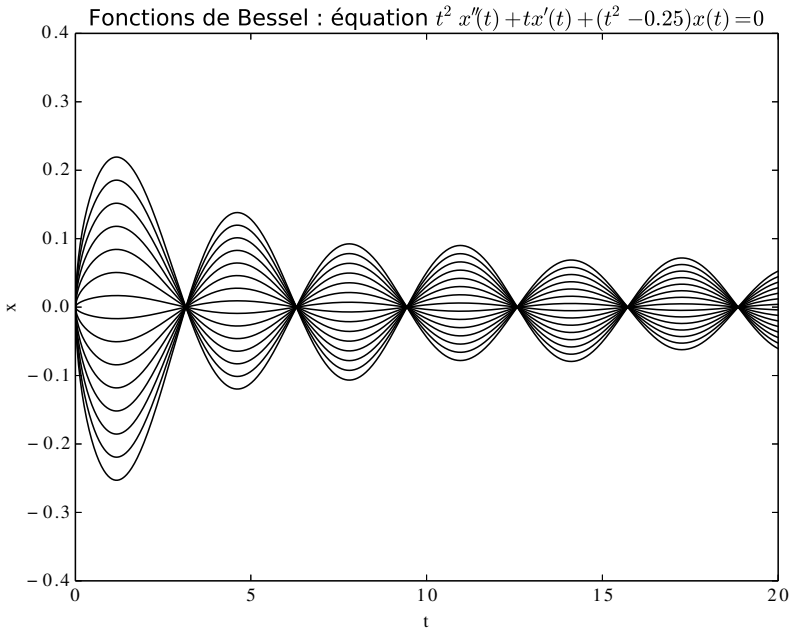
7ème programme

En utilisant la méthode de Heun, tracer plusieurs courbes intégrales de cette équation différentielle correspondant à différentes conditions initiales, par exemple dans le cas $\alpha = \dfrac{1}{2}$.

Corrigé :

```
import numpy as np
import matplotlib.pyplot as plt

from EquationDifferentielleVectorielle_Volterra import Heun

def F(t,y):
    alpha = 0.5
    return np.array([y[1], (alpha**2 / t**2 - 1) * y[0] - y[1] / t])

a = 0.01 #problèmes en 0
b = 20
pas = 0.01
nb_points = int( (b - a) / pas)
# liste des valeurs initiales
liste = [ np.array([0, x]) for x in np.arange(-3, 3, 0.4) ]
for y_0 in liste:
    X, Y = Heun(F, a, b, y_0, nb_points)
    Y1 = [ Y[i][0] for i in range(0, len(Y))]
    plt.plot(X, Y1, color='black')

plt.xlim(a, b)
plt.ylim(-0.4, 0.4)
plt.title("Fonctions de Bessel : équation $t^2x''(t) + tx'(t) +(t^2-0.25)x(t)=0$")
plt.xlabel('t')
plt.ylabel('x')
plt.show()
```

On obtient le graphique suivant :



III.2.3. L'équation du pendule

On considère une masse m suspendue par un fil rigide de longueur ℓ et de masse négligeable. On désigne par θ l'angle entre la verticale passant par le point O de suspension et la direction du fil, et g l'accélération de la pesanteur ;
De plus, on suppose que le pendule est soumis à un frottement visqueux (résistance de l'air) de coefficient k. On montre alors (voir votre cours de physique) que la fonction θ est solution de l'équation différentielle :

$$\theta''(t) + \frac{k}{m\ell^2}\theta'(t) + \frac{g}{\ell}\sin\big(\theta(t)\big) = 0\,.$$

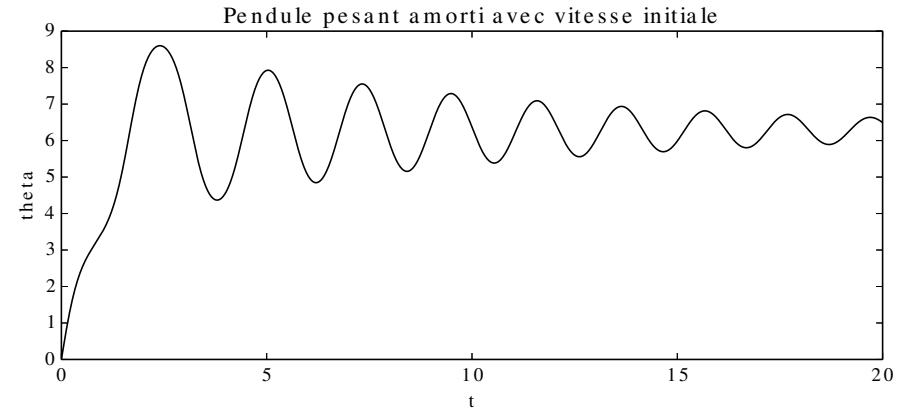
En posant $Y(t) = \begin{pmatrix}\theta(t)\\ \theta'(t)\end{pmatrix}$, cette équation s'écrit aussi

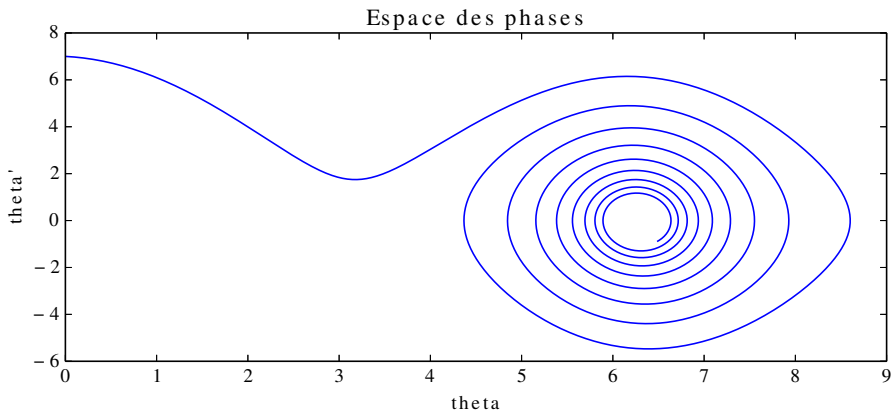
$$Y'(t) = F(t,Y) \quad \text{avec} \quad F\left(t, \begin{pmatrix}x\\ y\end{pmatrix}\right) = \begin{pmatrix} y \\ -\frac{k}{m\ell^2}y - \frac{g}{\ell}\sin(x)\end{pmatrix}\,.$$

Cette équation n'est pas linéaire,mais les mêmes méthodes restent applicables.

8ème programme

- Obtenir des graphes semblables à ceux ci-dessous.





Corrigé :

```python
import numpy as np
import matplotlib.pyplot as plt

def Euler(F, a, b, y_0, n):
    pas = (b - a)/n
    X = [a]
    t = a
    y = y_0
```

```
        Y = [y_0]
        for k in range(0, n):
            y = y + pas * F(t, y)
            t = t + pas
            X.append(t)
            Y.append(y)
        return X, Y

    if __name__ == "__main__":

        def F(t,y):
            g = 10
            m = 1
            k = 0.2
            l = 1
            return np.array([y[1], - g / l * np.sin(y[0]) - k / (m * l**2) * y[1]])
        a = 0
        b = 20
        nb_points = 10000

        y_0 = np.array([0, 7])

        fig = plt.figure(figsize=(9,8))
        plt.subplots_adjust(hspace=0.3)

        plt.subplot(2,1,1)
        X, Y = Euler(F, a, b, y_0, nb_points)
        Y1 = [ y[0] for y in Y ]
        plt.plot(X, Y1, color='black')
        plt.xlim(a, b)

        plt.title("Pendule pesant amorti avec vitesse initiale")
        plt.xlabel('t')
        plt.ylabel('theta')

        plt.subplot(2,1,2)
        Y2 = [ y[1] for y in Y ]
        plt.plot(Y1, Y2)
        plt.title("Espace des phases")
        plt.xlabel("theta")
        plt.ylabel("theta'")
        plt.show()
```

IV. La méthode des différences finies

IV.1. Approximation des dérivées

Soit f une application de classe $\mathscr{C}^n$ (avec n suffisamment grand) sur un intervalle I.
La formule de Taylor-Young en un point $x \in I$ donne :

$$f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + \ldots + \frac{h^{n-1}}{(n-1)!}f^{(n-1)}(x) + O(h^n).$$

Cette formule, appliqué en des points convenables, permet de démontrer sans difficulté :

$$f'(x) = \frac{f(x+h)-f(x)}{h} + O(h) \quad \text{ou} \quad f'(x) = \frac{f(x)-f(x-h)}{h} + O(h)$$

ce qui permet d'obtenir une approximation de $f'(x)$ par différence finie décentrée (en avant ou en arrière) d'ordre 1.
Pour améliorer la précision, on peut utiliser une différence finie centrée, qui sera d'ordre 2 :

$$f'(x) = \frac{f(x+h)-f(x-h)}{2h} + O(h^2)\,.$$

On peut de la même façon obtenir des approximations de la dérivée seconde. On utilise le plus souvent les différences finies centrées d'ordre 2 :

$$f''(x) = \frac{f(x+h)-2f(x)+f(x-h)}{h^2} + O(h^2)\,.$$

Si l'on veut augmenter la précision, on peut utiliser des différences finies centrées à l'ordre 4 (démontrer les égalités ci-dessous) :

$$f'(x) = \frac{f(x-2h) - 8f(x-h) + 8f(x+h) - f(x+2h)}{12h} + O(h^4)$$

et

$$f''(x) = \frac{-f(x-2h) + 16f(x-h) - 30f(x) + 16f(x+h) - f(x+2h)}{12h^2} + O(h^4)\,.$$

Notons enfin qu'on peut aussi obtenir, toujours à partir de la formule de Taylor, des approximations d'ordre 2 pour les dérivées troisième, quatrième etc.. si l'on a à résoudre des équations différentielles d'ordre supérieur. Le principe exposé ci-après reste valable pour ces équations, mais les calculs sont un peu plus compliqués.

IV.2. Cas d'une équation différentielle linéaire scalaire du second ordre

On considère une équation différentielle linéaire du second ordre, de la forme :

$$a(t)x''(t) + b(t)x'(t) + c(t)x(t) = d(t) \quad (L)$$

où a, b, c, d sont des applications continues définies sur un intervalle $[\alpha, \beta]$ à valeurs dans $\mathbb{R}$.

La discrétisation de cette équation consiste d'abord à considérer une subdivision régulière $(t_0, t_1, \ldots, t_{n+1})$ de l'intervalle $[\alpha, \beta]$, de pas $h = \dfrac{b-a}{n+1}$. C'est le *maillage*.

On écrit alors l'équation (L) aux points t_i pour $i \in [\![1\,;n]\!]$ en substituant aux dérivées les valeurs approchées à l'ordre 2 mentionnées au paragraphe précédent (on utilisera ici les différences centrées d'ordre 2, ce qui est le choix le plus courant. Le principe est le même mais les calculs sont un peu plus compliqués avec les différences d'ordre 4).

On obtient donc le système d'équations suivant, où l'on a noté pour simplifier $x_i = x(t_i)$ (donc $x_{i-1} = x(t_i - h)$ et $x_{i+1} = x(t_i + h)$) :

$$\forall i \in [\![1\,;n]\!]\ ,\ a(t_i)\frac{x_{i+1} - 2x_i + x_{i-1}}{h^2} + b(t_i)\frac{x_{i+1} - x_{i-1}}{2h} + c(t_i)x_i = d(t_i)$$

soit

$$\forall i \in [\![1\,;n]\!]\ ,\ \left[\frac{a(t_i)}{h^2} - \frac{b(t_i)}{2h}\right]x_{i-1} + \left[c(t_i) - 2\frac{a(t_i)}{h^2}\right]x_i + \left[\frac{a(t_i)}{h^2} + \frac{b(t_i)}{2h}\right]x_{i+1} = d(t_i) \qquad (1)$$

En considérant alors la matrice tridiagonale M, d'ordre n, telle que

$$M_{i,i-1} = \frac{a(t_i)}{h^2} - \frac{b(t_i)}{2h} \quad \text{pour } i \in [\![2\,;n]\!] \qquad M_{i,i} = c(t_i) - 2\frac{a(t_i)}{h^2} \quad \text{pour } i \in [\![1\,;n]\!]$$

$$\text{et } M_{i,i+1} = \frac{a(t_i)}{h^2} + \frac{b(t_i)}{2h} \quad \text{pour } i \in [\![2\,;n-1]\!]$$

(les autres coefficients étant nuls), les équations précédentes s'écrivent alors

$$\forall i \in [\![2\,;n-1]\!] \quad , \quad M_{i,i-1}x_{i-1} + M_{i,i}x_i + M_{i,i+1}x_{i+1} = d(t_i)\,.$$

Il faut de plus connaître les valeurs $x_0 = x(t_0)$ et $x_{n+1} = x(t_n)$ (la méthode des différences finies est donc particulièrement bien adapté aux problèmes où les conditions initiales sont les valeurs de la fonction aux bornes de l'intervalle, contrairement aux méthodes précédentes où il a fallu par exemple utiliser la méthode du tir).

En réécrivant alors (1) pour $i = 1$ et $i = n$, on obtient les équations supplémentaires

$$M_{1,1}x_1 + M_{1,2}x_2 = d(t_1) - \left[\frac{a(t_1)}{h^2} - \frac{b(t_1)}{2h}\right]x_0$$

et

$$M_{n,n-1}x_{n-1} + M_{n,n}x_n = d(t_n) - \left[\frac{a(t_n)}{h^2} + \frac{b(t_n)}{2h}\right]x_{n+1}\,.$$

En notant alors B le vecteur colonne $B = \begin{pmatrix} d(t_1) - \left[\dfrac{a(t_1)}{h^2} - \dfrac{b(t_1)}{2h}\right]x_0 \\ d(t_2) \\ \vdots \\ d(t_{n-1}) \\ d(t_n) - \left[\dfrac{a(t_n)}{h^2} + \dfrac{b(t_n)}{2h}\right]x_{n+1} \end{pmatrix}$ et X le vecteur colonne $X = {}^t(x_1, \ldots, x_n)$, le système précédent s'écrit : $MX = B$.

Il ne reste plus ensuite qu'à résoudre ce système. Puisqu'il est tridiagonal, cela pourra se faire simplement en deux étapes :

- on rend le système triangulaire supérieur par des combinaisons linéaires sur les lignes (méthode de Gauss)
- puis on résout le système obtenu « de proche en proche ».

Algorithme de Thomas (1949) pour la résolution d'un système tridiagonal :
Soit à résoudre un système $MX = D$ écrit sous la forme

$$\begin{pmatrix} b_1 & c_1 & 0 & & & 0 \\ a_2 & b_2 & c_2 & 0 & & 0 \\ 0 & a_3 & b_3 & c_3 & & \\ & & & \ddots & & \\ & & & \ddots & b_{n-1} & c_{n-1} \\ 0 & & & & a_n & b_n \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_{n-1} \\ x_n \end{pmatrix} = \begin{pmatrix} d_1 \\ d_2 \\ d_3 \\ \vdots \\ d_{n-1} \\ d_n \end{pmatrix}$$

Dans une première phase, on rend le système triangulaire supérieur :

```
pour k variant de 2 à n faire
```

⋮

puis on résout le système obtenu en commençant par la dernière équation :

⋮

Cet algorithme est de complexité $O(n)$ (il y a $8n - 7$ opérations en tout, faites le calcul...)

8ème programme

Écrire une fonction `TriDiag(A, B, C, D)` qui renvoie la solution X du système.
Attention : on rappelle qu'en Python, les listes sont indicées à partir de 0 ...

9ème programme

Le problème physique :
Considérons un fil élastique mono-dimensionnel dans le segment $[0, 1]$ maintenu en $x = 0$ et en $x = 1$ à l'altitude 0 et soumis à un chargement $f(x)$ perpendiculaire au segment, à l'équilibre.
Notons $u(x)$ l'altitude du fil à l'abscisse x. L'altitude du fil est solution du problème

$$\begin{cases} \forall x \in [0,1],\ -u''(x) + c(x)u(x) = f(x) \\ u(0) = u(1) = 0 \end{cases}$$

où $c(x)$ est donné par les caractéristiques du matériau qui constitue le fil (c'est le coefficient d'élasticité).

Tracer la solution obtenue par la méthode précédente, dans le cas $c(x) = cste = 1$ et $f(x) = \begin{cases} -1 & \text{si } x \in \left[0\,;\frac{1}{3}\right[\\ \frac{1}{2} & \text{si } x \in \left[\frac{1}{3}\,;\frac{2}{3}\right] \\ -2 & \text{si } x \in \left]\frac{2}{3}\,;1\right] \end{cases}$ (on comparera avec la solution exacte, que l'on aura calculée...)

Corrigé :

```python
import numpy as np
import matplotlib.pyplot as plt

def TriDiag(a, b, c, d):
    n = len(b)
    for k in range(0,n-1):
        m = a[k] / b[k]
        b[k+1] = b[k+1] - m * c[k]
        d[k+1] = d[k+1] - m * d[k]
    x = n * [0]
    x[n-1] = d[n-1] / b[n-1]
    for k in range(n-2, -1, -1):
```

```
13          x[k] = (d[k] - c[k] * x[k+1])/b[k]
14
15  def f(t):
16      if t < 1/3:
17          return -1
18      elif t < 2/3:
19          return 0.5
20      else:
21          return -2
22
23  n = 10000
24  a = 0
25  b = 1
26  x0 = 0
27  x1 = 0
28  h = (b-a)/ (n+1)
29
30  T = [i*h for i in range(0,n+2)]
31  # on pouvait aussi utiliser np.linspace, bien sûr
32  a = (n-1) * [-1/(h*h)]
33  c = (n-1) * [-1/(h*h)]
34  b = n * [1 + 2/(h*h)]
35  d = [f(i*h) for i in range(1, n+1)]
36
37  X = TriDiag(a,b,c,d)
38  X.append(x1)
39  X.insert(0,x0)
40  plt.plot(T,X)
41  plt.show()
```

Voici le résultat :