Serie13

Simone Deparis

June 5, 2025

Contents

[1]:
```
# importing libraries used in this book
import numpy as np
import matplotlib.pyplot as plt

# In my case, the  OrdinaryDifferentialEquationsLib is in the parent directory,
# therfore have have to add the aprent directory to path :
import sys
sys.path.append('..')

from ODESystemLib import forwardEulerSystem, backwardEulerSystem,␣
↪CrankNicolsonSystem
```

[2]:
```
# scipy.linalg.eig : eigenvalues of a  matrix
from scipy.linalg import eig
```

1 Exemple Système ODE

Le système linéaire

$$\begin{cases} y_1'(t) &= -2y_1(t) + y_2(t) + e^{-t} \\ y_2'(t) &= 3y_1(t) - 4y_2(t) \end{cases}$$

avec les conditions initiales $y_1(0) = y_{10}$, $y_2(0) = y_{20}$, s'écrit sous la forme

$$\begin{cases} \mathbf{y}'(t) = A\mathbf{y}(t) + \mathbf{b}(t) \quad t > 0, \\ \mathbf{y}(0) = \mathbf{y}_0, \end{cases},$$

où

$$\mathbf{y}(t) = \begin{bmatrix} y_1(t) \\ y_2(t) \end{bmatrix}, \quad A = \begin{bmatrix} -2 & 1 \\ 3 & -4 \end{bmatrix}, \quad \mathbf{b}(t) = \begin{bmatrix} e^{-t} \\ 0 \end{bmatrix}, \quad \mathbf{y}_0 = \begin{bmatrix} y_{10} \\ y_{20} \end{bmatrix}.$$

Soit $h > 0$ le pas de temps. Pour $n \in \mathbb{N}$, on pose $t_n = nh$, $\mathbf{b}_n = \mathbf{b}(t_n)$ et on désigne par $\mathbf{u}_n$ une valeur approchée de la solution exacte $\mathbf{y}(t_n)$ au temps t_n.

1.1 Ecrire les Schémas d'Euler progressive, retrograde et Crank-Nicolson pour ce problème.

Les schémas d'Euler progressif, d'Euler rétrograde et de Crank-Nicolson pour approcher la solution $\mathbf{y}(t)$ de s'écrivent respectivement:

Euler progressif
$$\begin{cases} \mathbf{u}_{n+1} = \mathbf{u}_n + hA\mathbf{u}_n + h\mathbf{b}_n = (I + hA)\mathbf{u}_n + h\mathbf{b}_n \\ \mathbf{u}_0 = \mathbf{y}_0 \end{cases}$$

Euler rétrograde
$$\begin{cases} (I - hA)\mathbf{u}_{n+1} = \mathbf{u}_n + h\mathbf{b}_{n+1} \\ \mathbf{u}_0 = \mathbf{y}_0 \end{cases}$$

Crank-Nicolson
$$\begin{cases} (I - \frac{h}{2}A)\mathbf{u}_{n+1} = (I + \frac{h}{2}A)\mathbf{u}_n + \frac{h}{2}(\mathbf{b}_n + \mathbf{b}_{n+1}) \\ \mathbf{u}_0 = \mathbf{y}_0 \end{cases}$$

$$n = 0, 1, ..., N_h - 1$$

Il faut remarquer qu'à chaque étape des méthodes de ER et CN, il faut résoudre un système linéaire avec pour matrice $I - hA$ et $I - \frac{h}{2}A$ respectivement (il s'agit de méthodes implicites).

1.2 Setup

Nous allons définir la matrice A comme dépendante du temps pour généraliser, même si ici elle est constante

[3]:
```
A = lambda t : np.array([[-2, 1 ], [ 3 , -4 ]]);
b = lambda t : np.array([np.exp(-t), 0]);

y0 = np.array([1,1])
hb = 2/5;
t0 = 0; T = 8; tsp = [t0, T];
```

```
f = lambda t,x : ( A(t)@ x +  b(t) )
```

1.3 Stabilité d'Euler Progressive

Il faut calculer les valeurs propres de A

[4]:
```
lk, v = eig(A(0))
print(lk)
# j représente le nombre complexe tq j^2 = -1
```

```
[-1.+0.j -5.+0.j]
```

La méthode d'Euler progressive est explicite (il n'y a pas de système linéaire à résoudre), par contre elle est seulement conditionellement stable. Dans notre cas, les valeurs propres de A sont $\lambda_1 = -1$ et $\lambda_2 = -5$; elles sont bien négatives, donc la condition de stabilité sur h s'applique: comme $\rho(A) = 5$, cette condition de stabilité est

$$h < \bar{h} = \frac{2}{5}.$$

1.4 Approximez la solution en utilisant `ODESystemLib`

```
from ODESystemLib import forwardEulerSystem, backwardEulerSystem, CrankNicolsonSystem
```

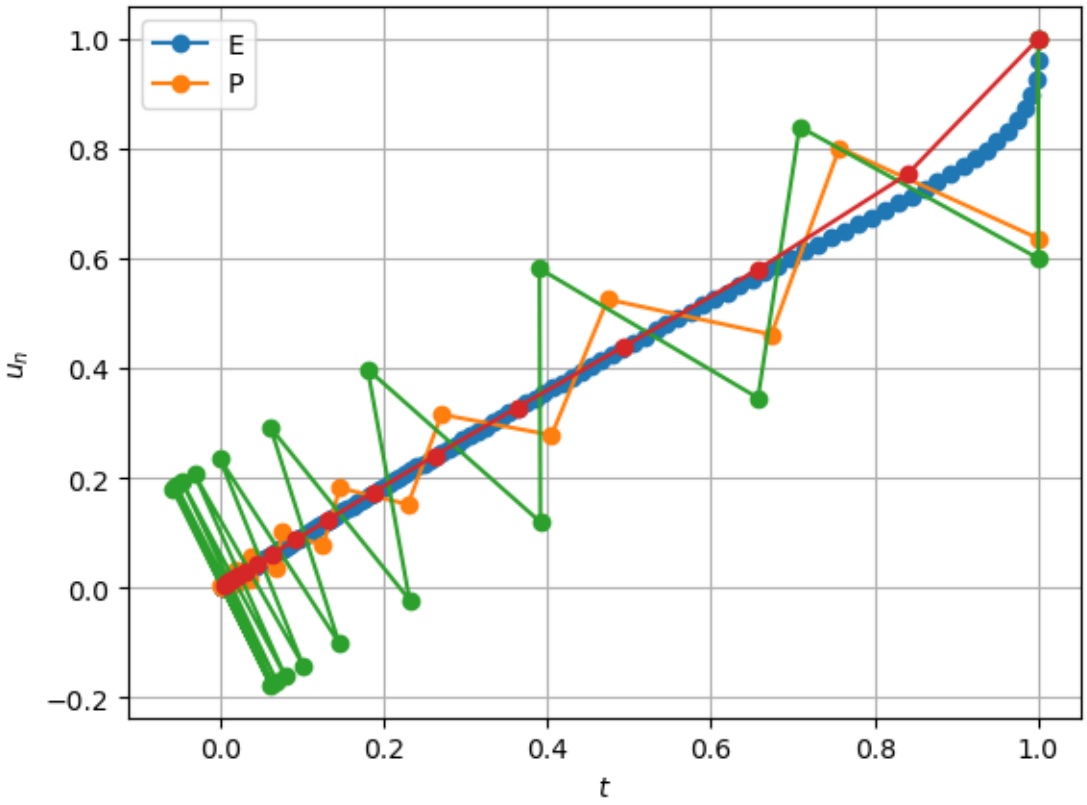
(Regardez dans `ODESystemLib` comment appeler ces fonctions)

[5]:
```
prop = 0.1
Nh = int(T/(prop*hb));
t01, u01 = forwardEulerSystem(f, tsp, y0, Nh);

prop = 1
Nh = int(T/(prop*hb));
t10, u10 = forwardEulerSystem(f, tsp, y0, Nh);

prop = 0.9
Nh = int(T/(prop*hb));
t09, u09 = forwardEulerSystem(f, tsp, y0, Nh);

prop = 1.2
Nh = int(T/(prop*hb));
tbe, ube = backwardEulerSystem(A, b, tsp, y0, Nh);
```

Pour tracer le graphe de la solution numérique en fonction du temps, il faut se rappeller que u_n est la valeur approchée.

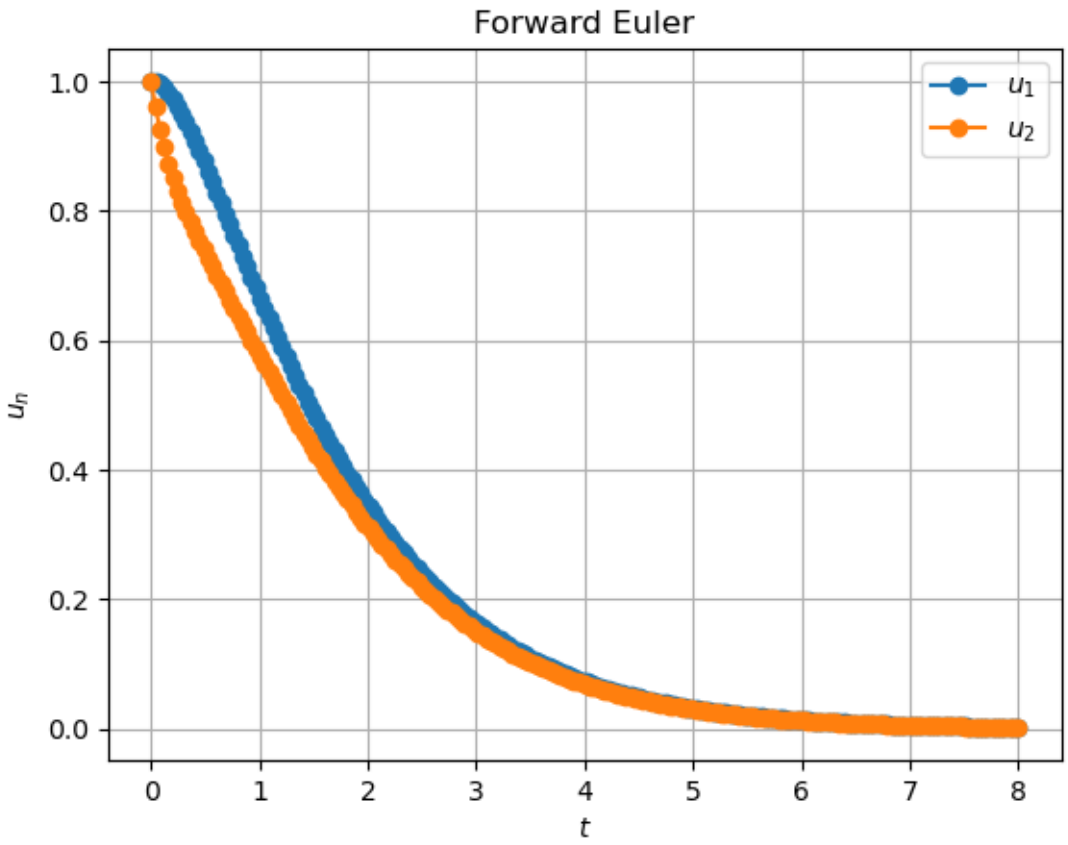
[6]:
```
plt.plot(u01[0,:],u01[1,:],'o-')
plt.plot(u09[0,:],u09[1,:],'o-')
plt.plot(u10[0,:],u10[1,:],'o-')

plt.plot(ube[0,:],ube[1,:],'o-')
```

```
# labels, title, legend
plt.xlabel('$t$'); plt.ylabel('$u_n$')
plt.legend('EP')
plt.grid(True)
plt.show()
```

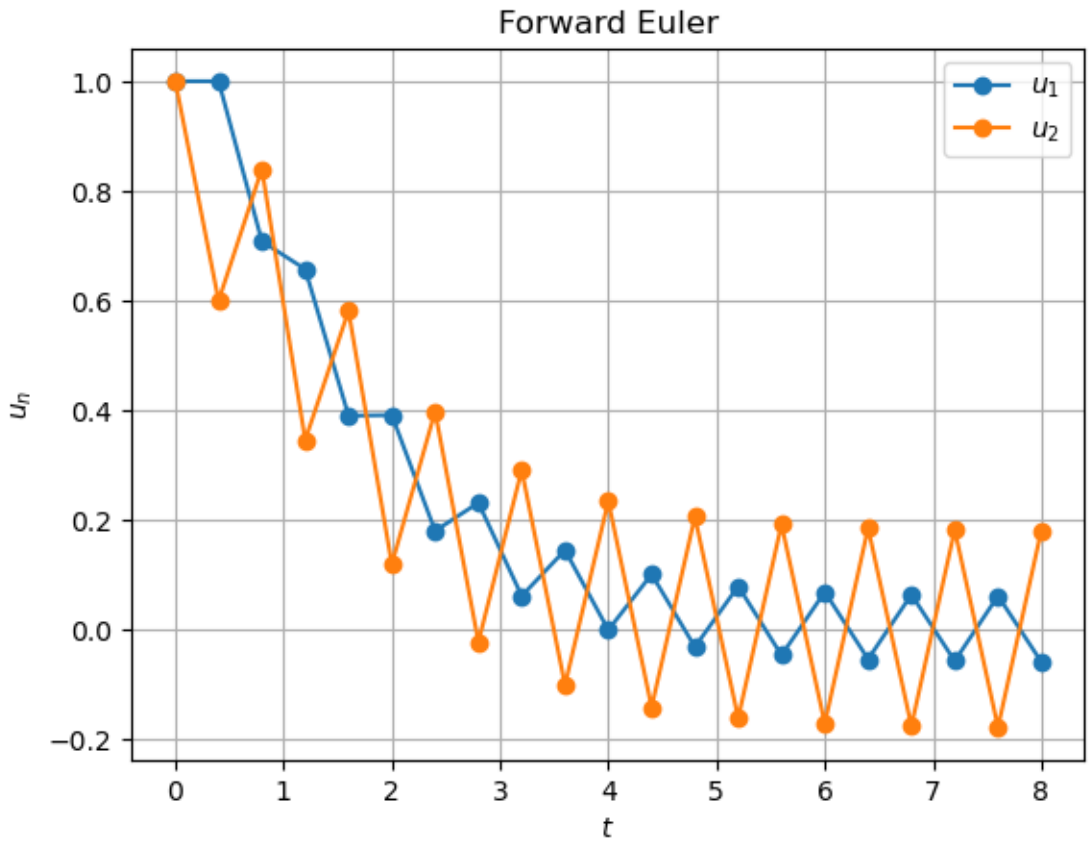


On peut aussi déssiner le comportement des deux variables

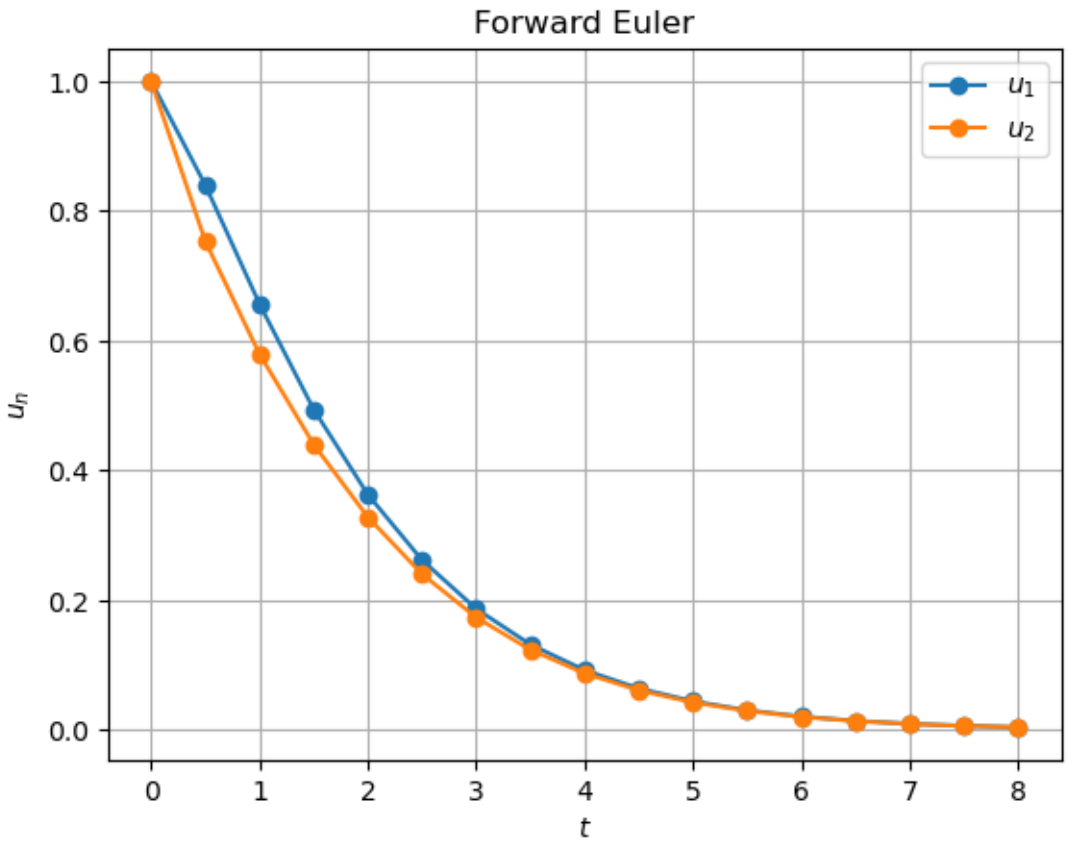
[7]:
```
plt.plot(t01,u01[0,:],'o-')
plt.plot(t01,u01[1,:],'o-')

# labels, title, legend
plt.title("Forward Euler")
plt.xlabel('$t$'); plt.ylabel('$u_n$')
plt.legend(['$u_1$', '$u_2$'])
plt.grid(True)
plt.show()
```



[8]:
```
plt.plot(t10,u10[0,:],'o-')
plt.plot(t10,u10[1,:],'o-')

# labels, title, legend
plt.title("Forward Euler")
plt.xlabel('$t$'); plt.ylabel('$u_n$')
plt.legend(['$u_1$', '$u_2$'])
plt.grid(True)
plt.show()
```



[9]:
```
plt.plot(tbe,ube[0,:],'o-')
plt.plot(tbe,ube[1,:],'o-')

# labels, title, legend
plt.title("Forward Euler")
plt.xlabel('$t$'); plt.ylabel('$u_n$')
plt.legend(['$u_1$', '$u_2$'])
plt.grid(True)
plt.show()
```



[10]:
```
plt.plot(tbe,ucn[0,:],'o-')
plt.plot(tbe,ucn[1,:],'o-')

# labels, title, legend
plt.title("Backward Euler")
plt.xlabel('$t$'); plt.ylabel('$u_n$')
plt.legend(['$u_1$', '$u_2$'])
plt.grid(True)
plt.show()
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[10], line 1
----> 1 plt.plot(tbe,ucn[0,:],'o-')
      2 plt.plot(tbe,ucn[1,:],'o-')
      4 # labels, title, legend

NameError: name 'ucn' is not defined
```

[ ]:

[ ]:

2 Exemple Système ODE

Le système non-linéaire

$$y_1'(t) = -2y_1(t) + \sin(y_2(t)) + e^{-t}\sin(t)$$
$$y_2'(t) = \cos(y_1(t)) - 4y_2(t)$$

avec les conditions initiales $y_1(0) = y_{10}$, $y_2(0) = y_{20}$,

s'écrit sous la forme

$$\mathbf{y}'(t) = \mathbf{F}(t, \mathbf{y}(t)),$$

où

$$\mathbf{F}(t, \mathbf{y}(t)) = \begin{bmatrix} -2y_1(t) + \sin(y_2(t)) + e^{0.1t}\sin(t) \\ \cos(y_1(t)) - 4y_2(t) \end{bmatrix}.$$

Soit $h > 0$ le pas de temps. Pour $n \in \mathbb{N}$, on pose $t_n = nh$, $\mathbf{b}_n = \mathbf{b}(t_n)$ et on désigne par $\mathbf{u}_n$ une valeur approchée de la solution exacte $\mathbf{y}(t_n)$ au temps t_n.

2.1 Ecrire les Schémas d'Euler progressive, retrograde et Crank-Nicolson pour ce problème.

Les schémas d'Euler progressif, d'Euler rétrograde et de Crank-Nicolson pour approcher la solution $\mathbf{y}(t)$ de s'écrivent respectivement:

Euler progressif
$$\begin{cases} \mathbf{u}_{n+1} = \mathbf{u}_n + hF(t_n, \mathbf{u}_n) \\ \mathbf{u}_0 = \mathbf{y}_0 \end{cases}$$

Euler rétrograde
$$\begin{cases} \mathbf{u}_{n+1} = \mathbf{u}_n + hF(t_{n+1}, \mathbf{u}_{n+1}) \\ \mathbf{u}_0 = \mathbf{y}_0 \end{cases}$$

Crank-Nicolson
$$\begin{cases} \mathbf{u}_{n+1} = \mathbf{u}_n + \frac{h}{2}\left(F(t_n, \mathbf{u}_n) + F(t_{n+1}, \mathbf{u}_{n+1})\right) \\ \mathbf{u}_0 = \mathbf{y}_0 \end{cases}$$
$$n = 0, 1, ..., N_h - 1$$

Il faut remarquer qu'à chaque étape des méthodes de ER et CN, il faut résoudre un système non-linéaire (il s'agit de méthodes implicites).

La méthode d'Euler progressive est explicite (il n'y a pas de système linéaire à résoudre), par contre elle est seulement conditionellement stable. Dans notre cas, les valeurs propres de A sont $\lambda_1 = -1$ et $\lambda_2 = -5$; elles sont bien négatives, donc la condition de stabilité sur h s'applique: comme $\rho(A) = 5$, cette condition de stabilité est

$$h < \bar{h} = \frac{2}{5}.$$

2.2 Approximer la solution en utilisant `forwardEulerSystem`

`from ODESystemLib import forwardEulerSystem`

(Regardez dans `ODESystemLib` comment appeler ces fonctions)

[11]:
```
# importing libraries used in this book
import numpy as np
```

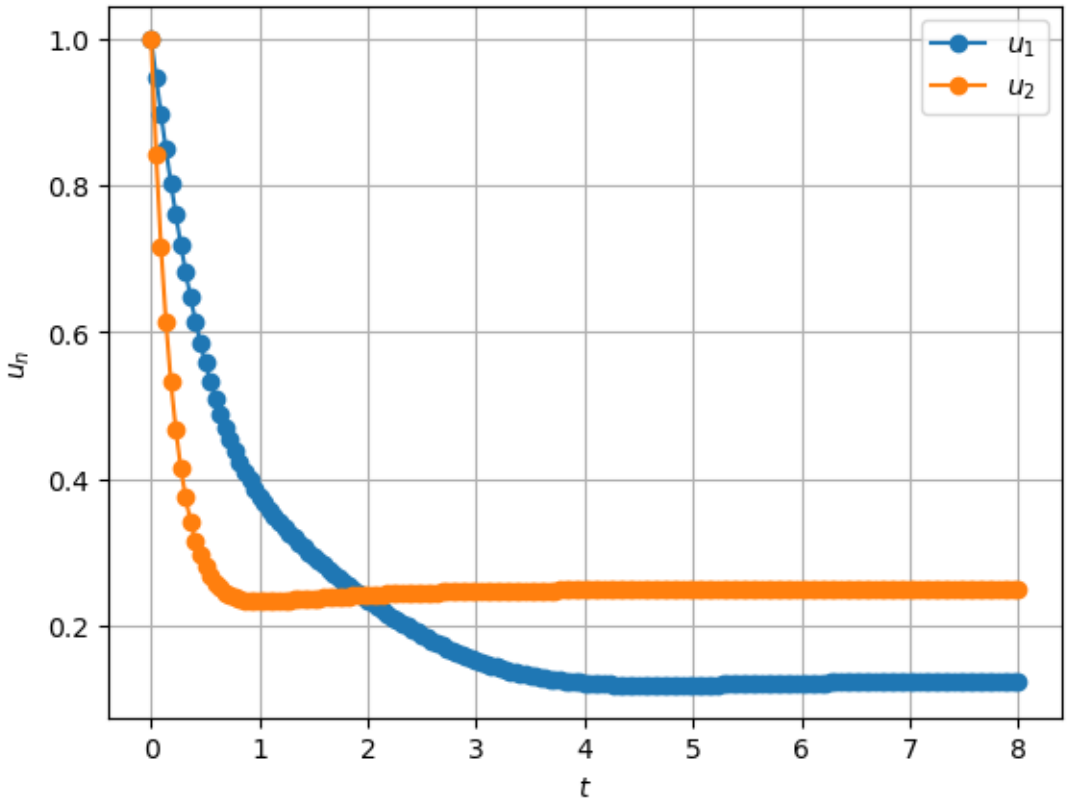
```
import matplotlib.pyplot as plt

# In my case, the  OrdinaryDifferentialEquationsLib is in the parent directory,
# therfore have have to add the aprent directory to path :
import sys
sys.path.append('..')

from ODESystemLib import forwardEulerSystem, backwardEulerSystem,␣
 ↪CrankNicolsonSystem
```

[12]:
```
y0 = np.array([1,1])
hb = 2/(3+np.sqrt(2));
t0 = 0; T = 8; tsp = [t0, T];

f = lambda t,x : np.array([ -2*x[0] + np.sin(x[1]) + np.exp(-t)*np.sin(t) ,
                            np.cos(x[0]) - 4*x[1] ])

prop = 0.1
Nh = int(T/(prop*hb));
t01, u01 = forwardEulerSystem(f, tsp, y0, Nh);

prop = 1.2
Nh = int(T/(prop*hb));
t12, u12 = forwardEulerSystem(f, tsp, y0, Nh);

prop = 0.9
Nh = int(T/(prop*hb));
t09, u09 = forwardEulerSystem(f, tsp, y0, Nh);
```

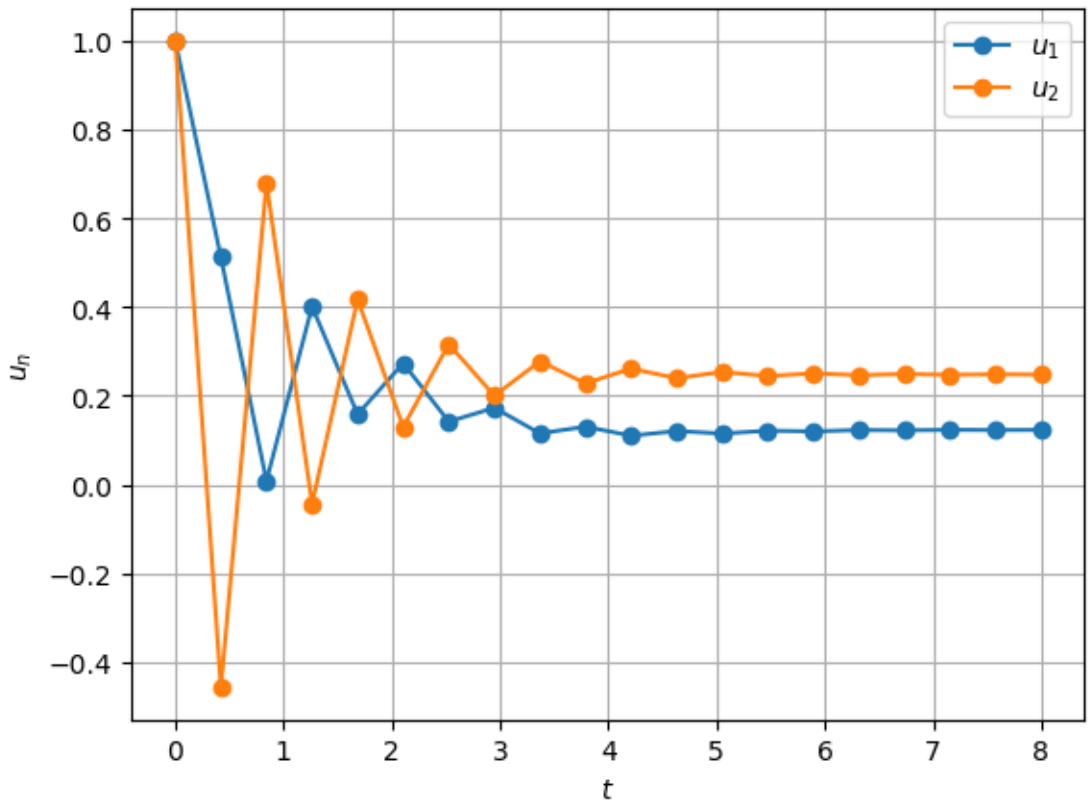
Pour tracer le graphe de la solution numérique en fonction du temps, il faut se rappeller que u_n est la valeur approchée.

On peut aussi déssiner le comportement des deux variables

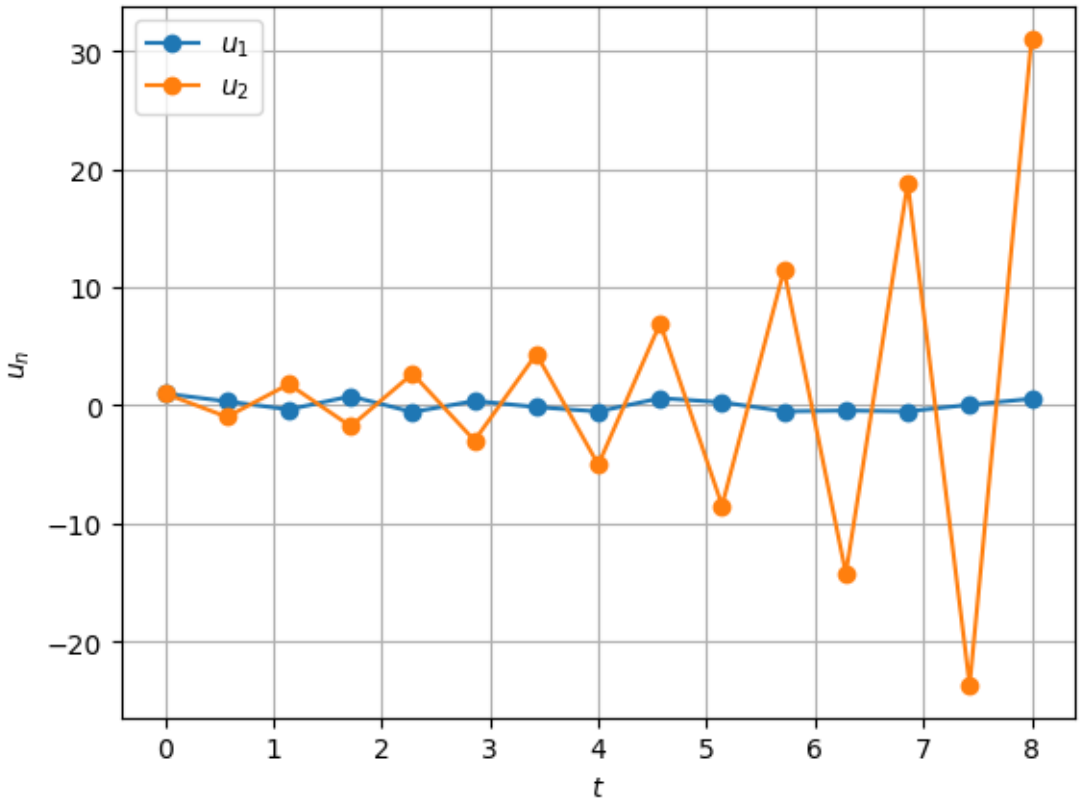
[13]:
```
plt.plot(t01,u01[0,:],'o-')
plt.plot(t01,u01[1,:],'o-')

# labels, title, legend
plt.xlabel('$t$'); plt.ylabel('$u_n$')
plt.legend(['$u_1$', '$u_2$'])
plt.grid(True)
plt.show()
```



[14]:
```
plt.plot(t09,u09[0,:],'o-')
plt.plot(t09,u09[1,:],'o-')

# labels, title, legend
plt.xlabel('$t$'); plt.ylabel('$u_n$')
plt.legend(['$u_1$', '$u_2$'])
plt.grid(True)
plt.show()
```



[15]:
```
plt.plot(t12,u12[0,:],'o-')
plt.plot(t12,u12[1,:],'o-')

# labels, title, legend
plt.xlabel('$t$'); plt.ylabel('$u_n$')
plt.legend(['$u_1$', '$u_2$'])
plt.grid(True)
plt.show()
```


u_1
u_2
30
20
10
0
−10
−20
u_n
0
1
2
3
4
5
6
7
8
t

[ ]:

[ ]:

3 Dynamique de population

On considère une population de y individus dans un environnement où au plus $B = 1000$ individus peuvent coexister. On suppose qu'initialement le nombre d'individus est $y_0 = 100$ et que le facteur de croissance est égal à une constante C. Le modèle de l'évolution de la population considérée sur 100 années est le suivant:

$$y'(t) = Cy(t)\left(1 - \frac{y(t)}{B}\right),\ t \in (0, 100), \quad y(0) = y_0,$$

où t est mesuré *en années*, et $C = 2/15$ an^{-1}.

Soit u_n l'approximation de $y(t_n)$, où $t_n = nh$, $n = 0, 1, 2, \ldots, N_h$, $h = 100/N_h$ étant le pas de temps, N_h étant le nombre de pas temporels. Plus N_h est grand, plus le pas de temps est petit et plus l'approximation sera précise.

1. Ecrire les schémas d'Euler progressif (explicite) et rétrograde (implicite) pour calculer une approximation u_n de $y(t_n)$ (donner les équations de récurrence définissant la suite u_n dans les deux cas).
2. On prend $h = 1/12$ an; donner la valeur u_1 qui approche le nombre d'individus $y(t_1)$ au temps t_1 (donc après *1 mois*), obtenue *i)* par la méthode d'Euler progressive, puis *ii)* par la méthode d'Euler rétrograde. Suggestion: resoudre à la main l'équation pour trouver la nouvelle valeur u_{i+1}.
3. En utilisant la méthode d'Euler progressif (copier et modifier la fonction forwardEuler.m créé précédemment), calculer les valeurs approchées u_n, $n = 0, 1, \ldots, N_h$ en Python, pour $N_h = 20$.
4. Tracer un graphe de la solution numérique trouvée en fonction du temps.
5. Faire à nouveau le calcul précédent, mais en prenant $N_h = 1000$; tracer la nouvelle solution numérique.
6. D'après les résultats trouvés, combien d'années sont nécessaires pour que la population atteigne le nombre de 900 individus? *Suggestion: utiliser la commande* `find`.

Partie 1 En général, un schéma numérique pour le calcul des approximations u_n des valeurs $y(t_n)$, s'écrit sous la forme d'une équation *de récurrence* définissant u_{n+1} en fonction de la valeur u_n trouvée au pas précedent. Dans notre cas, le schéma d'Euler progressif est donné par

$$\begin{cases} u_{n+1} = u_n + hCu_n\left(1 - \dfrac{u_n}{B}\right), \\ u_0 = 100, \end{cases}$$

tandis que celui d'Euler rétrograde s'écrit

$$\begin{cases} u_{n+1} = u_n + hCu_{n+1}\left(1 - \dfrac{u_{n+1}}{B}\right), \\ u_0 = 100. \end{cases}$$

Partie 2 Avec les valeurs données, aprés un mois la valeur approchée du nombre d'individus que l'on calcule par la méthode d'Euler progressive est

$$u_1 = 100 + \frac{1}{12}\frac{2}{15}100\left(1 - \frac{100}{1000}\right) = 101.$$

Pour calculer l'approximation selon le schéma d'Euler rétrograde, il faut résoudre une équation avec u_1 comme inconnue:

$$hC\frac{1}{B}u_1^2 + (1 - hC)u_1 - u_0 = 0.$$

Dans ce cas, la solution n'est pas unique. On a deux racines:

$$u_1 = \frac{B}{2hC}\left[-(1 - hC) \pm \sqrt{(1 - hC)^2 + 4hC\frac{u_0}{B}}\right].$$

Néanmoins, seule la solution positive est acceptable (on rappelle que l'on cherche à calculer un nombre d'individus). Celle-ci est donc

$$u_1 = 1000 \cdot 45\left[-\left(1 - \frac{1}{90}\right) + \sqrt{\left(1 - \frac{1}{90}\right)^2 + \frac{2}{450}}\right] = 101.008958.$$

Partie 3 On sait que le schéma d'Euler progressif pour le problème considéré est

$$\begin{cases} u_{n+1} = u_n + hCu_n\left(1 - \dfrac{u_n}{B}\right), \\ u_0 = 100. \end{cases}$$

En Python, ceci s'écrit

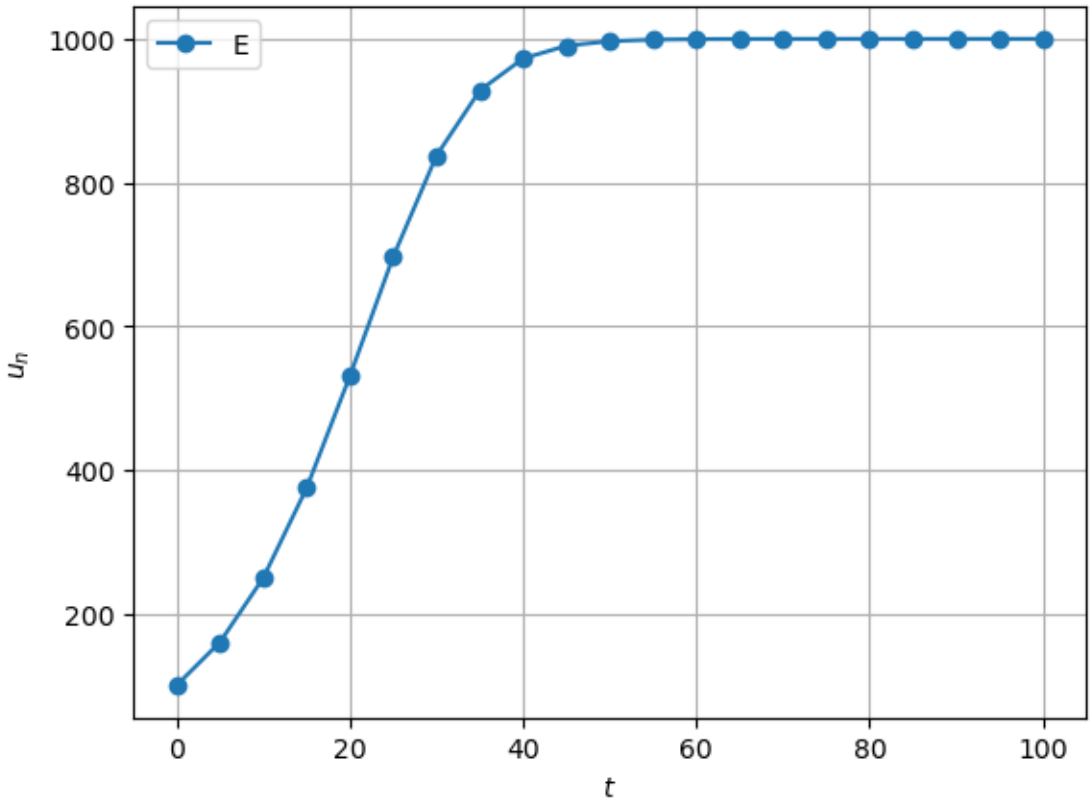
[16]:
```
# importing libraries used in this book
import numpy as np
import matplotlib.pyplot as plt

# In my case, the  OrdinaryDifferentialEquationsLib is in the parent directory,
# therfore have have to add the aprent directory to path :
import sys
sys.path.append('..')

from OrdinaryDifferentialEquationsLib import forwardEuler, backwardEuler
```

[17]:
```
C = 2/15; B=1000;         # valeurs des paramtres C et B
y0 = 100
Nh = 20;                  # valeur de Nh
t0 = 0; T = 100.; tsp = [t0, T]
h = (T-t0)/Nh;               # pas de temps

f = lambda t,x : ( C * x * (1-x/B) )
t, u = forwardEuler(f, tsp, y0, Nh);
```

où on a suivi la suggestion de l'énoncé.

Partie 4 Pour tracer le graphe de la solution numérique en fonction du temps, il faut se rappeller que u_n est la valeur approchée au temps $t_n = nh$. Donc on peut taper

[18]:
```
plt.plot(t,u,'o-')

# labels, title, legend
plt.xlabel('$t$'); plt.ylabel('$u_n$')
plt.legend('EP')
plt.grid(True)
plt.show()
```



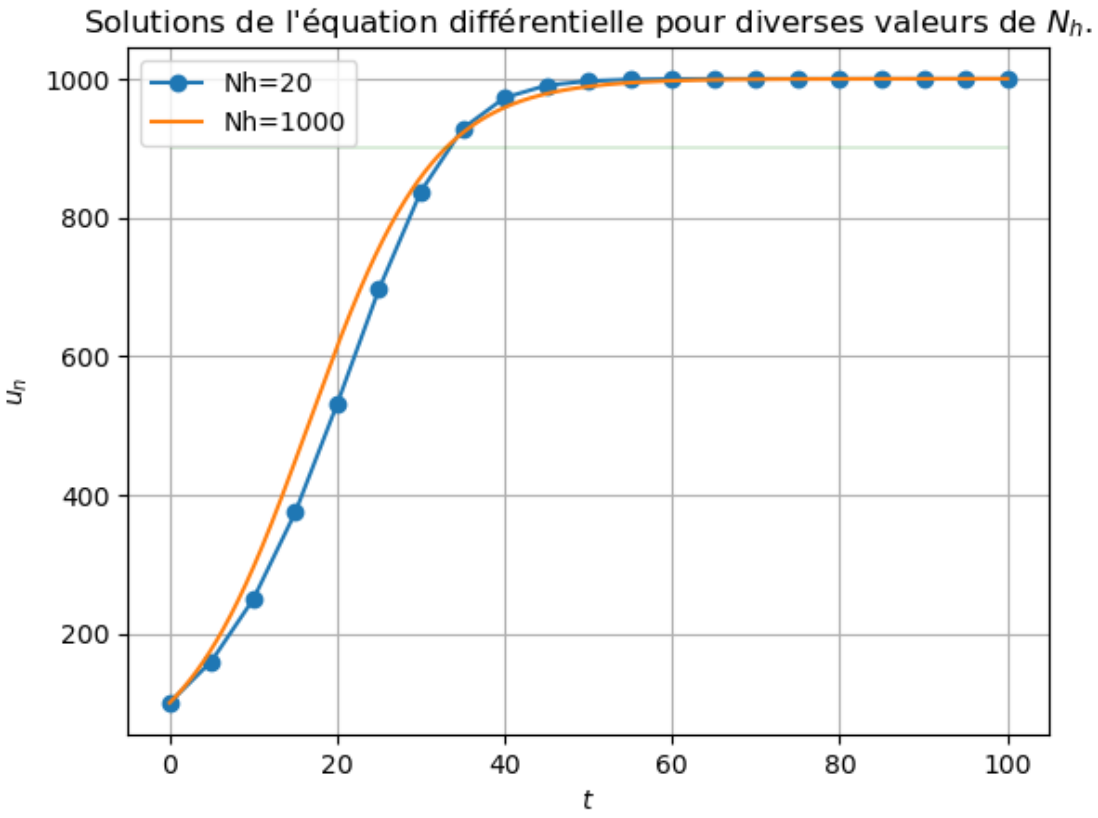
pour obtenir le graphe cherché.

Partie 5 On utilise les mêmes commandes qu'au point a), mais on change la valeur de `Nh`:

[19]:
```
C = 2/15; B=1000;         # valeurs des paramtres C et B
y0 = 100
Nh = 1000;                  # valeur de Nh
t0 = 0; T = 100.; tsp = [t0, T]
h = (T-t0)/Nh;               # pas de temps

f = lambda t,x : ( C * x * (1-x/B) )
t1, u1 = forwardEuler(f, tsp, y0, Nh);
```

Ensuite on trace le nouveau graphe avec

[20]:
```
plt.plot(t,u,'o-')
plt.plot(t1,u1,'-')

# labels, title, legend
plt.xlabel('$t$'); plt.ylabel('$u_n$')
plt.legend(['Nh=20','Nh=1000'])
plt.grid(True)
plt.title("Solutions de l'équation différentielle pour diverses valeurs de $N_h$.
 ↪")

plt.plot([t0,T], [900,900], 'g-',linewidth=0.2)

plt.savefig("EX020_fig.pdf", dpi=150)
plt.show()
```



On s'attend à ce que la solution correspondant à $N_h = 1000$ soit la plus précise, donc la plus proche de l'évolution exacte de la population.

Partie 6 On peut répondre soit en analysant le graphe de la solution numérique la plus précise ($N_h = 1000$), en tracent la droite correspondent à `u=900` (avec la commande `plt.plot([t0,T], [900,900], 'g-',linewidth=0.2)`), soit en utilisant la commande `np.where` de numpy. On donne

ici un exemple de comment utiliser cette fonction (*Remarque*: si `u` est un vecteur, alors `(u>=900)` est aussi un vecteur, dont la composante i-ème est égale à 1 si $u(i) \geq 900$, zéro autrement. La commande `where` retourne alors le vecteur des indices non-nuls de `(u>=900)`; finalement, le premier élément du vecteur résultant nous donne le plus petit de ces indices) :

[21]:
```
index = np.where(u1 >= 900)

i = index[0][0];          # le plus petit des indices i
                          # tels que u(i)> =900
T = h*i
print(T)
```

```
33.0
```

Donc la population atteint le nombre de 900 individus après $T = 33$ ans.

[ ]:

[ ]:

4 Proie-prédateur

Le système non-linéaire

$$y_1'(t) = 0.08\, y_1(t) - 0.004\, y_1(t) y_2(t),$$
$$y_2'(t) = -0.06\, y_2(t) + 0.002\, y_1(t) y_2(t).$$

avec les conditions initiales $y_1(0) = 40$, $y_2(0) = 20$,

s'écrit sous la forme

$$\mathbf{y}'(t) = \mathbf{F}(t, \mathbf{y}(t)),$$

où

$$\mathbf{F}(t, \mathbf{x}) = \begin{bmatrix} 0.08\, x_1 - 0.004\, x_1 x_2 \\ -0.06\, x_2 + 0.002\, x_1 x_2 \end{bmatrix}.$$

Soit $h > 0$ le pas de temps. Pour $n \in \mathbb{N}$, on pose $t_n = nh$, $\mathbf{b}_n = \mathbf{b}(t_n)$ et on désigne par $\mathbf{u}_n$ une valeur approchée de la solution exacte $\mathbf{y}(t_n)$ au temps t_n.

4.1 Ecrire les Schémas d'Euler progressive, retrograde et Crank-Nicolson pour ce problème.

Les schémas d'Euler progressif, d'Euler rétrograde et de Crank-Nicolson pour approcher la solution $\mathbf{y}(t)$ de s'écrivent respectivement:

Euler progressif
$$\begin{cases} \mathbf{u}_{n+1} = \mathbf{u}_n + hF(t_n, \mathbf{u}_n) \\ \mathbf{u}_0 = \mathbf{y}_0 \end{cases}$$

Euler rétrograde
$$\begin{cases} \mathbf{u}_{n+1} = \mathbf{u}_n + hF(t_{n+1}, \mathbf{u}_{n+1}) \\ \mathbf{u}_0 = \mathbf{y}_0 \end{cases}$$

Crank-Nicolson
$$\begin{cases} \mathbf{u}_{n+1} = \mathbf{u}_n + \frac{h}{2}\left(F(t_n, \mathbf{u}_n) + F(t_{n+1}, \mathbf{u}_{n+1})\right) \\ \mathbf{u}_0 = \mathbf{y}_0 \end{cases}$$

$$n = 0, 1, ..., N_h - 1$$

Il faut remarquer qu'à chaque étape des méthodes de ER et CN, il faut résoudre un système non-linéaire (il s'agit de méthodes implicites).

4.2 Approximez la solution en utilisant `forwardEulerSystem`

```
from ODESystemLib import forwardEulerSystem
```

(Regardez dans `ODESystemLib` comment appeler ces fonctions)

[22]:
```
# importing libraries used in this book
import numpy as np
import matplotlib.pyplot as plt

# In my case, the  OrdinaryDifferentialEquationsLib is in the parent directory,
# therfore have have to add the aprent directory to path :
import sys
sys.path.append('..')
```

```
from ODESystemLib import forwardEulerSystem, backwardEulerSystem,␣
↪CrankNicolsonSystem
```

[23]:
```
# scipy.linalg.eig : eigenvalues of a  matrix
from scipy.linalg import eig
# scipy.linalg.lu : LU decomposition
from scipy.linalg import lu
# scipy.linalg.cholesky : Cholesky decomposition
from scipy.linalg import cholesky
# scipy.linalg.hilbert : Hilbert matrix
from scipy.linalg import hilbert

np.set_printoptions(precision=4, suppress=True, linewidth=120)
```

[24]:
```
y0 = np.array([40,20])
hb = 2/10.0; # Ici nous n'avons pas d'idée sur la valeur de lambda_max, on␣
↪invente en on prend lambda_max = 5
t0 = 0; T = 1200; tsp = [t0, T];

f = lambda t,x : np.array([  0.08*x[0] - 0.004*x[0]*x[1],
                           -0.06*x[1] + 0.002*x[0]*x[1] ])
```

4.3 Stabilité d'Euler progressive

La Jacobienne de F est la suivante

$$\frac{\partial \mathbf{F}}{\partial x}(t, \mathbf{x}) = \begin{bmatrix} 0.08 - 0.004\, x_2 & -0.004\, x_1 \\ 0.002\, x_2 & -0.06 + 0.002\, x_1 \end{bmatrix}.$$

On peut dessiner le countour des valeurs propres dependemment de (x_1, x_2).

[25]:
```
df = lambda t,x : np.array([ [ 0.08- 0.004*x[1], - 0.004*x[0]] ,
                           [ 0.002*x[1],-0.06 + 0.002*x[0]] ])
```

4.4 Stabilité d'Euler progressive

La méthode d'Euler progressive est explicite (il n'y a pas de système linéaire à résoudre), par contre elle est seulement conditionellement stable.

Il faudrait à chaque pas de temps calculer les valeurs propres et choisir h en conséquence. Nous allons le faire même si cela n'est pas réaliste.

De plus, comme l'on verra plus bas, le système physique alterne entre moment stables et enstables.

[26]:
```
prop = 0.1
Nh = int(T/(prop*hb));
t01, u01 = forwardEulerSystem(f, tsp, y0, Nh);
```

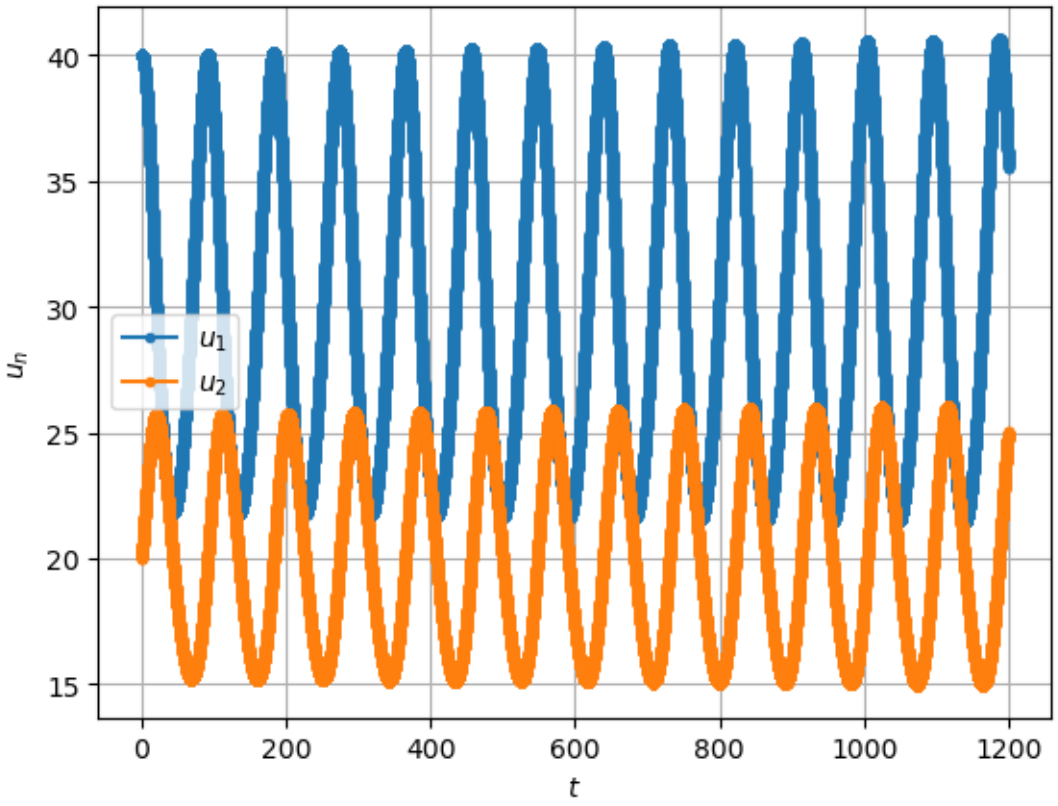
[27]:
```
prop = 0.9
Nh = int(T/(prop*hb));
t09, u09 = forwardEulerSystem(f, tsp, y0, Nh);
```

[28]:
```
prop = 1.2
Nh = int(T/(prop*hb));
t12, u12 = forwardEulerSystem(f, tsp, y0, Nh);
```

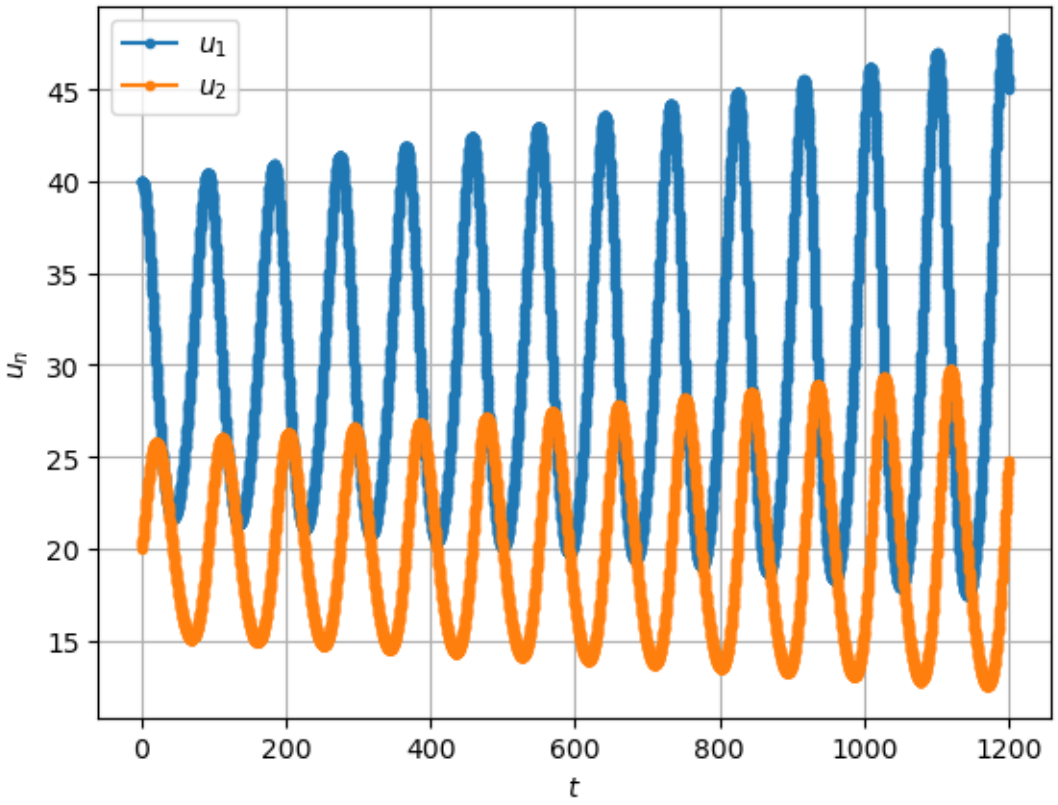
Pour tracer le graphe de la solution numérique en fonction du temps, il faut se rappeller que u_n est la valeur approchée.

On peut aussi déssiner le comportement des deux variables

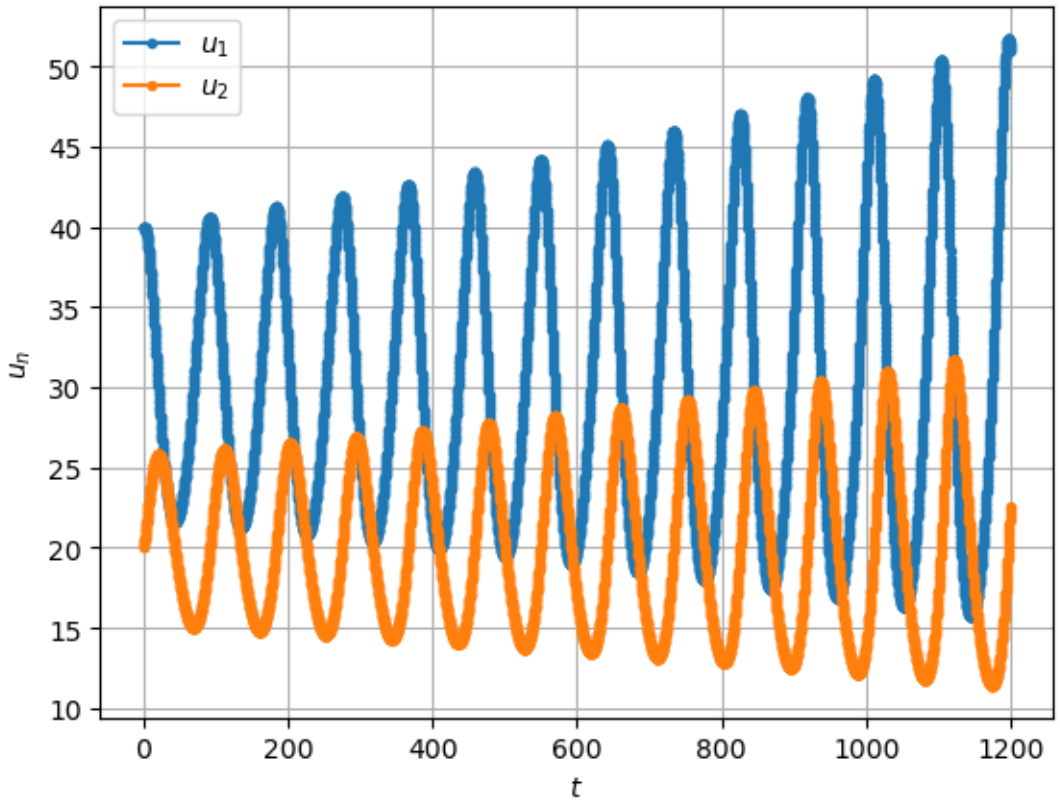
[29]:
```
plt.plot(t01,u01[0,:],'.-')
plt.plot(t01,u01[1,:],'.-')

# labels, title, legend
plt.xlabel('$t$'); plt.ylabel('$u_n$')
plt.legend(['$u_1$', '$u_2$'])
plt.grid(True)
plt.show()
```



[30]:
```
plt.plot(t09,u09[0,:],'.-')
plt.plot(t09,u09[1,:],'.-')

# labels, title, legend
plt.xlabel('$t$'); plt.ylabel('$u_n$')
plt.legend(['$u_1$', '$u_2$'])
plt.grid(True)
plt.show()
```



[31]:
```
plt.plot(t12,u12[0,:],'.-')
plt.plot(t12,u12[1,:],'.-')

# labels, title, legend
plt.xlabel('$t$'); plt.ylabel('$u_n$')
plt.legend(['$u_1$', '$u_2$'])
plt.grid(True)
plt.show()
```



4.5 Stabilité de la solution

Dans le cas général (pas vu en cours), le système est stable quand la partie réelle des valeurs propres de $\partial_x F$ est négative, autrement le système est instable.

On peut calculer $\partial_x F$ le long de la solution numérique et identifier des moments où la configuration est stable ou pas.

On peut le voir en dessinant la trajectoire $(u_1, u_2$, avec une élevation z égale au max des valeurs propres de $\lambda_1(\partial_x F)$.

4.6 Calcul numérique des valeurs propres de $\partial_x F$

[32]:
```
lam, v = eig(df(0,u01[:,0]))
```

4.7 Calcul analytique des valeurs propres de $\partial_x F$

Dans ce cas particulier, on peut aussi calculer des valeurs propres de manière analytique :

[33]:
```
def lam1(u):
  x = u[0,:]
  y = u[1,:]
```

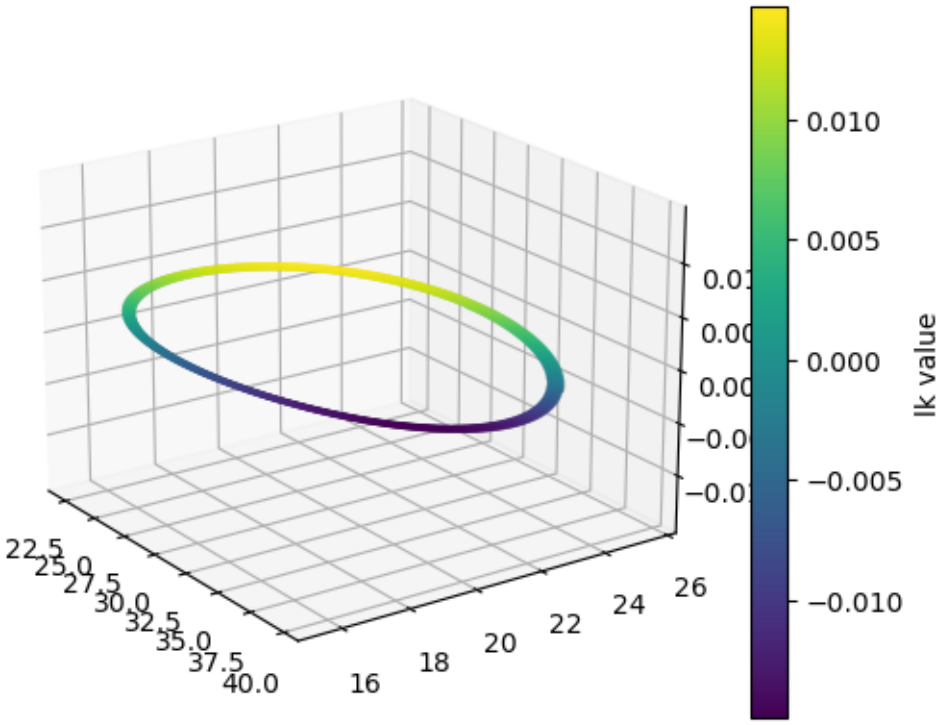
```
  return np.real(x - 2 *y - np.sqrt(x**2 - 4*y*x - 140*x + 4*y**2 - 280*y +␣
↪4900,  dtype=complex) + 10)/1000
def lam2(u):
  x = u[0,:]
  y = u[1,:]
  return np.real(x - 2 *y + np.sqrt(x**2 - 4*y*x - 140*x + 4*y**2 - 280*y +␣
↪4900,  dtype=complex) + 10)/1000
```

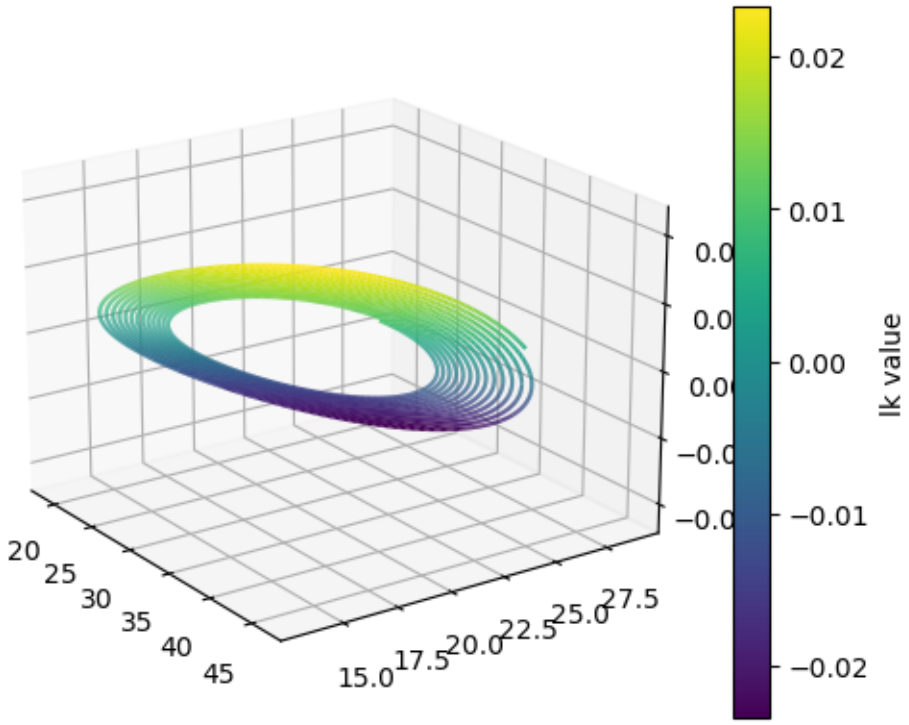
[34]:
```
## With eig fonction
Nh = u01[1,:].shape[0] -1
lk01 = np.zeros(Nh+1)
for k in range(0,Nh+1) :
    lkk, v = eig(df(0,u01[:,k]))
    lk01[k] = np.real(lkk[1])
```

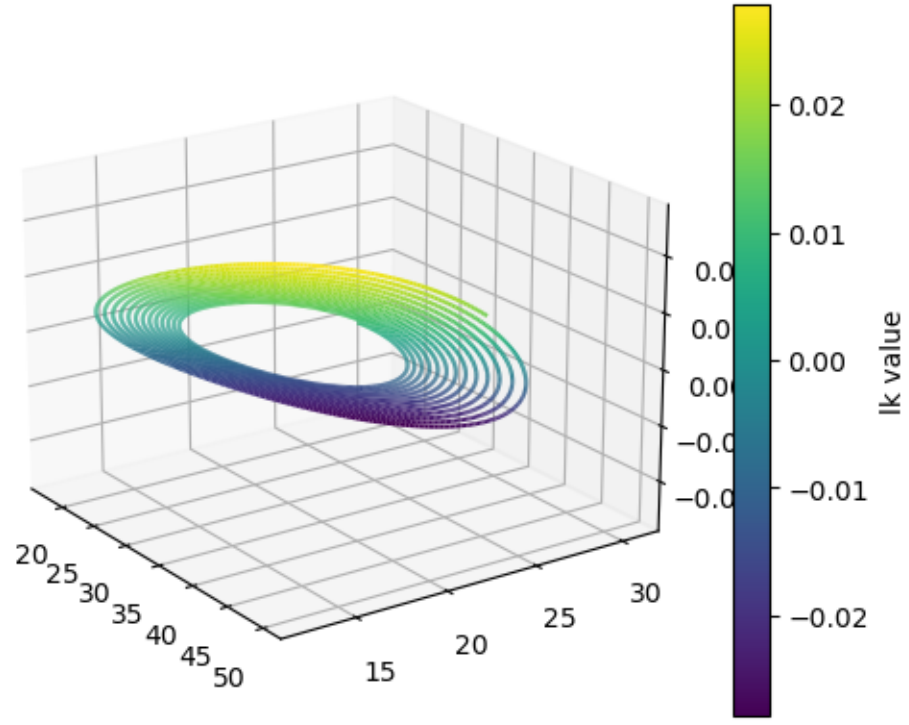
[35]:
```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d.art3d import Line3DCollection

def plotLambda(u,lk) :

    # Assume u01 is shape (2, N) and lk is shape (N,)
    x = np.ravel(u[0, :])
    y = np.ravel(u[1, :])
    z = np.ravel(lk)

    # Check they have the same length
    assert x.shape == y.shape == z.shape, "x, y, z must be 1D arrays of the same␣
↪length"

    # Stack into (N, 3) array of points
    points = np.vstack([x, y, z]).T  # shape (N, 3)

    # Build line segments between each consecutive point
    segments = np.array([points[i:i+2] for i in range(len(points)-1)])

    # Use average lk value per segment for coloring (length = len(segments))
    lk_segment = (z[:-1] + z[1:]) / 2

    # Create line collection
    lc = Line3DCollection(segments, cmap='viridis', norm=plt.
↪Normalize(lk_segment.min(), lk_segment.max()))
    lc.set_array(lk_segment)

    # Plot
    fig = plt.figure()
    ax = fig.add_subplot(projection='3d')
```

```
    ax.add_collection3d(lc)

    # Set axis limits
    ax.set_xlim(x.min(), x.max())
    ax.set_ylim(y.min(), y.max())
    ax.set_zlim(z.min(), z.max())

    ax.view_init(elev=20., azim=-35, roll=0)
    plt.colorbar(lc, ax=ax, label='lk value')

    plt.show()
```

[36]:
```
plotLambda(u01,lam2(u01))
plotLambda(u09,lam2(u09))
plotLambda(u12,lam2(u12))
```




0.02
0.01
0.00
−0.01
−0.02
Ik value
20
25
30
35
40
45
15.0
17.5
20.0
22.5
25.0
27.5


0.02
0.01
0.00
−0.01
−0.02
Ik value
20
25
30
35
40
45
50
15
20
25
30

[ ]:

[ ]: