Assignment 5 - Equations differentielles ordinaires

May 27, 2025

1 Assignment 5: Èquations différentielles ordinaires

Avant de voir le code disponible de ce test et avant de commencer à rédiger vos réponses, prenez le temps de réfléchir à la manière dont vous pouvez organiser le travail.

- **Pensez à quelles parties du test utiliseront des fonctions écrites/résultats obtenus dans les parties précédentes.**
- **Réfléchissez à la structure de votre code (vous pouvez faire un brouillon sur papier).**
- **Réfléchissez aux sections du cours qui vous seront utiles pour l'analyse de vos résultats.**

Dans l'intervalle $[0, T]$ avec $T = 8$, on considère le système linéaire d'équations différentielles ordinaires (EDO) suivante

$$\begin{cases} y_1'(t) &= y_1(t) - 2y_2(t) + e^{\frac{t}{5T}} \\ y_2'(t) &= 9y_1(t) - 10y_2(t) - e^{\frac{t}{5T}} \end{cases}$$

En posant les conditions initiales $y_1(0) = 1$, $y_2(0) = 1$, ça s'écrit sous la forme

$$\begin{cases} \mathbf{y}'(t) = A\mathbf{y}(t) + \mathbf{b}(t) \quad \text{pour} \quad t \in (0, T] \ , \\ \mathbf{y}(0) = \mathbf{y}_0 \ , \end{cases}$$

où

$$\mathbf{y}(t) = \begin{bmatrix} y_1(t) \\ y_2(t) \end{bmatrix}, \quad A = \begin{bmatrix} 1 & -2 \\ 9 & -10 \end{bmatrix}, \quad \mathbf{b}(t) = e^{\frac{t}{5T}} \begin{bmatrix} 1 \\ -1 \end{bmatrix}, \quad \mathbf{y}_0 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}.$$

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.

[ ]:
```
import numpy as np
from scipy.linalg import eig
from scipy.integrate import simpson

import matplotlib.pyplot as plt
from matplotlib.collections import LineCollection
```

```
import matplotlib as mpl

from ODESystemLib import forwardEulerSystem, backwardEulerSystem

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

[ ]:
```
# ODE problem setup

t0 = 0
T = 8
tsp = [t0, T]

# REMARK: to be consistent with the implementation of the methods,
#         A and b are defined as time-dependent functions,
#         even if in this case they are constant
A = lambda t : np.array([[1, -2], [9 , -10]])
b = lambda t : np.array([np.exp(t/(5*T)), -np.exp(t/(5*T))])
f = lambda t, x : A(t)@ x +  b(t)

y0 = np.array([1, 1])
```

1.1 Partie 1

D'abord, on essaie résoudre le système d'EDO à l'aide de la méthode d'Euler progressive.

Sur la base de la théorie vue au cours, determiner le pas de temps maximale $h^* > 0$ pour lequel la méthode d'Euler progressive est stable. Justifier la réponse. (**Commentaire 1**)

Résoudre le systeme d'EDO pour $h = 0.1h^*$, $h = 0.9h^*$ et $h = h^*$, en utilisant la fonction suivante (disponible dans `ODESystemLib.py`)

```
def forwardEulerSystem( fun, interval, y0, N ) :
    """ Solve ordinary differential equations using the forward Euler method.

    Inputs: [fun, interval, y0, N]
        fun : right-hand side term. It must be a callable (e.g. function, lambda),
              that takes time (t) and solution (u) in input.
        interval: integration interval, list of the form [t0, T].
        y0: initial condition, list or numpy array.
        N: number of discrete timesteps

    Outputs: [t, u]
        t : vector of discrete time instants where the solution is approximated.
        u : solution approximation at the discrete time instants.
    """
```

Dessiner le comportement des deux composantes de la solution dans le temps et donner un commentaire sur la base des résultats obtenus. (**Commentaire 2**)

[ ]:
```
h_ref = ## COMPLETE HERE (maximal step size for which forward Euler is stable)
```

1.1.1 Commentaire 1

TODO

[ ]:
```
prop = 0.1
Nh = int(T/(prop*h_ref))
t01, u01 = ## COMPLETE HERE (call method)

prop = 0.9
Nh = int(T/(prop*h_ref))
t09, u09 = ## COMPLETE HERE (call method)

prop = 1.0
Nh = int(T/(prop*h_ref))
t10, u10 = ## COMPLETE HERE (call method)

prop = 2
Nh = int(T/(prop*h_ref))
t20, u20 = ## COMPLETE HERE (call method)
```

[ ]:
```
fig, axs = plt.subplots(1, 4, sharey=False, figsize=(13,5))

axs[0].plot(t01, u01[0,:], '-o')
axs[0].plot(t01, u01[1,:], '-o')
axs[0].set_title(r"$h = 0.1 \ h^*$")

axs[1].plot(t09, u09[0,:], '-o')
axs[1].plot(t09, u09[1,:], '-o')
axs[1].set_title(r"$h = 0.9 \ h^*$")

axs[2].plot(t10, u10[0,:], '-o')
axs[2].plot(t10, u10[1,:], '-o')
axs[2].set_title(r"$h = h^*$")

axs[3].plot(t20, u20[0,:], '-o')
axs[3].plot(t20, u20[1,:], '-o')
axs[3].set_title(r"$h = 2 \ h^*$")

for ax in axs:
    ax.set_xlabel('$t$', fontsize=16)
    ax.set_ylabel('$u_n$', fontsize=16)
    ax.legend(['$u_1$', '$u_2$'], fontsize=16)
    ax.grid(which='major', linestyle='--', linewidth=1)

plt.tight_layout()
```

```
plt.show()
```

[ ]:
```
fig, axs = plt.subplots(1, 4, sharey=False, figsize=(13,5))

def plot_state(t, u, ax, title):
    segments = np.concatenate([u.T[:-1, None], u.T[1:, None]], axis=1)
    norm = plt.Normalize(t0, T)
    lc = LineCollection(segments, cmap='plasma', norm=norm)
    lc.set_array(t)
    lc.set_linewidth(2)
    line = ax.add_collection(lc)

    ax.set_xlim(min(u[0]) - 0.05, max(u[0]) + 0.05)
    ax.set_ylim(min(u[1]) - 0.05, max(u[1]) + 0.05)
    ax.set_title(title)
    ax.set_xlabel('$u_1$', fontsize=16)
    ax.set_ylabel('$u_2$', fontsize=16)
    ax.grid(which='major', linestyle='--', linewidth=1)

    return

plot_state(t01, u01, axs[0], r"$h = 0.1 \ h^*$")
plot_state(t09, u09, axs[1], r"$h = 0.9 \ h^*$")
plot_state(t10, u10, axs[2], r"$h = h^*$")
plot_state(t20, u20, axs[3], r"$h = 2 \ h^*$")

# Add a shared colorbar outside the figure
cbar_ax = fig.add_axes([0.92, 0.15, 0.02, 0.7])
norm = plt.Normalize(t0, T)
sm = mpl.cm.ScalarMappable(cmap="plasma", norm=norm)
sm.set_array([])
cbar = fig.colorbar(sm, cax=cbar_ax)
cbar.set_label('Time', fontsize=16)

fig.tight_layout(rect=[0, 0, 0.9, 1])
plt.show()
```

1.1.2 Commentaire 2

TODO

1.2 Partie 2

Répeter la *Partie 1*, mais en considerant la méthode d'Euler rétrograde.

Pour approximer numeriquement la solution, la fonction suivante est mise à disposition dans la librairie `ODESystemLib.py`:

```
def backwardEulerSystem(A, b, interval, y0, N) :
    """ Solve ordinary differential equations using the backward Euler method.

    NOTE: this method is limited to the case of an affine right-hand side of the form
          f(t,x) = A(t)x + b(t).

    Inputs: [A, b, interval, y0, N]
        A :  callable (e.g. function, lambda), returning the matrix A at a given time instant.
        b :  callable (e.g. function, lambda), returning the vector b at a given time instant.
        interval : integration interval, list of the form [t0, T].
        y0 : initial condition, list or numpy array.
        N : number of discrete timesteps

    Outputs: [t, u]
        t : vector of discrete time instants where the solution is approximated.
        u : solution approximation at the discrete time instants.
    """
```

Donner un commentaire sur les résultats obtenus. En particulier, qu'est qu'on peut dire par rapport à la stabilité de cette méthode? (**Commentaire 3**)

[ ]:
```
prop = 0.1
Nh = int(T/(prop*h_ref))
t01, u01 = ## COMPLETE HERE (call method)

prop = 0.9
Nh = int(T/(prop*h_ref))
t09, u09 = ## COMPLETE HERE (call method)

prop = 1.0
Nh = int(T/(prop*h_ref))
t10, u10 = ## COMPLETE HERE (call method)

prop = 2
Nh = int(T/(prop*h_ref))
t20, u20 = ## COMPLETE HERE (call method)
```

[ ]:
```
fig, axs = plt.subplots(1, 4, sharey=True, figsize=(13,5))

axs[0].plot(t01, u01[0,:], '-o')
axs[0].plot(t01, u01[1,:], '-o')
axs[0].set_title(r"$h = 0.1 \ h^*$")

axs[1].plot(t09, u09[0,:], '-o')
axs[1].plot(t09, u09[1,:], '-o')
axs[1].set_title(r"$h = 0.9 \ h^*$")
```

```
axs[2].plot(t10, u10[0,:], '-o')
axs[2].plot(t10, u10[1,:], '-o')
axs[2].set_title(r"$h = h^*$")

axs[3].plot(t20, u20[0,:], '-o')
axs[3].plot(t20, u20[1,:], '-o')
axs[3].set_title(r"$h = 2 \ h^*$")

for ax in axs:
    ax.set_xlabel('$t$', fontsize=16)
    ax.set_ylabel('$u_n$', fontsize=16)
    ax.legend(['$u_1$', '$u_2$'], fontsize=16)
    ax.grid(which='major', linestyle='--', linewidth=1)

plt.tight_layout()
plt.show()
```

1.2.1 Commentaire 3

TODO

1.3 Partie 3

Èxrire une fonction qui implemente la méthode de Crank-Nicholson, pour résoudre des équations différentielles ordinaires linéaires, c-à-d avec $f(t,x) = A(t)x + b(t)$. La fonction doit avoir la structure suivante:

```
def CrankNicholsonSystem(A, b, interval, y0, N) :
    """ Solve ordinary differential equations using the Crank-Nicholson method.

    NOTE: this method is limited to the case of an affine right-hand side, of the form f(t,x) =

    Inputs: [A, b, interval, y0, N]
        A :  callable (e.g. function, lambda), returning the matrix A at a given time instant.
        b :  callable (e.g. function, lambda), returning the vector b at a given time instant.
        interval : integration interval, list of the form [t0, T].
        y0 : initial condition, list or numpy array.
        N : number of discrete timesteps

    Outputs: [t, u]
        t : vector of discrete time instants where the solution is approximated.
        u : solution approximation at the discrete time instants.
    """
```

On rappelle que, pour un generique terme de droite $f(t,y)$, la méthode de Crank-Nicholson s'écrit de la façon suivante:

$$\begin{cases} \mathbf{u}_{n+1} = \mathbf{u}_n + \dfrac{h}{2}\Big(\mathbf{f}(t_n, \mathbf{u}_n) + \mathbf{f}(t_{n+1}, \mathbf{u}_{n+1})\Big) \ , \\ \mathbf{u}_0 = \mathbf{u_0} \ , \end{cases}$$

étant $\mathbf{u_0}$ la donnée initiale.

[ ]:
```python
def CrankNicholsonSystem(A, b, interval, y0, N) :
    """ Solve ordinary differential equations using the Crank-Nicholson method.

    NOTE: this method is limited to the case of an affine right-hand side, of␣
 ↪the form f(t,x) = A(t)x + b(t).

    Inputs: [A, b, interval, y0, N]
        A :  callable (e.g. function, lambda), returning the matrix A at a␣
 ↪given time instant.
        b :  callable (e.g. function, lambda), returning the vector b at a␣
 ↪given time instant.
        interval : integration interval, list of the form [t0, T].
        y0 : initial condition, list or numpy array.
        N : number of discrete timesteps

    Outputs: [t, u]
        t : vector of discrete time instants where the solution is approximated.
        u : solution approximation at the discrete time instants.
    """

    # number of solution components
    d  = y0.size

    # time step
    h = (interval[1] - interval[0]) / N

    # time snapshots
    t = np.linspace(interval[0], interval[1], N+1)

    # initialize the solution vector
    u = np.empty([d, N+1])
    u[:, 0] = y0

    # time loop
    for n in range(N) :

        ## COMPLETE HERE: Crank-Nicholson method solution update

    return t, u
```

1.4 Partie 4

Choisir le pas de temps $h = h^*$, calculé dans la *Partie 1*. Analyser numeriquement la stabilité de la méthode de Crank-Nicholson, en résolvant l'EDO pour les mêmes valeurs de h considerées dans la *Partie 1*.

Donner un commentaire sur le résultats obtenus. Est-ce que la méthode est inconditionallement stable? (**Commentaire 4**)

[ ]:
```
prop = 0.1
Nh = int(T/(prop*h_ref))
t01, u01 = ## COMPLETE HERE (call method)

prop = 0.9
Nh = int(T/(prop*h_ref))
t09, u09 = ## COMPLETE HERE (call method)

prop = 1.0
Nh = int(T/(prop*h_ref))
t10, u10 = ## COMPLETE HERE (call method)

prop = 2.0
Nh = int(T/(prop*h_ref))
t20, u20 = ## COMPLETE HERE (call method)
```

[ ]:
```
fig, axs = plt.subplots(1, 4, sharey=True, figsize=(13,5))

axs[0].plot(t01, u01[0,:], '-o')
axs[0].plot(t01, u01[1,:], '-o')
axs[0].set_title(r"$h = 0.1 \ h^*$")

axs[1].plot(t09, u09[0,:], '-o')
axs[1].plot(t09, u09[1,:], '-o')
axs[1].set_title(r"$h = 0.9 \ h^*$")

axs[2].plot(t10, u10[0,:], '-o')
axs[2].plot(t10, u10[1,:], '-o')
axs[2].set_title(r"$h = h^*$")

axs[3].plot(t20, u20[0,:], '-o')
axs[3].plot(t20, u20[1,:], '-o')
axs[3].set_title(r"$h = 2 \ h^*$")

for ax in axs:
    ax.set_xlabel('$t$', fontsize=16)
    ax.set_ylabel('$u_n$', fontsize=16)
    ax.legend(['$u_1$', '$u_2$'], fontsize=16)
    ax.grid(which='major', linestyle='--', linewidth=1)
```

```
plt.tight_layout()
plt.show()
```

1.4.1 Commentaire 4

TODO

1.5 Partie 5

Finalement, on s'interesse à étudier l'ordre de convergence des differentes méthodes analysées dans ce test.

Résolvez l'EDO avec les méthodes de Euler progressif, Euler rétrograde et Crank-Nicholson pour $h = \left[\frac{h^*}{4}, \frac{h^*}{8}, \frac{h^*}{16}, \frac{h^*}{32}\right]$. Considerez comme solution exacte celle obtenu avec la méthode de Crank-Nicholson pour $h = \frac{h^*}{128}$.

Calculez les erreurs par rapport à la solution exacte et dérivez (graphiquement) l'ordre de convergence pour chaque méthode. Pour chaque valeur de h, l'erreur relative est calculée de la manière suivante

$$E_h = \left(\frac{\int_0^T ||\hat{\mathbf{u}}(t) - \mathbf{u}(t)||^2 \, dt}{\int_0^T ||\hat{\mathbf{u}}(t)||^2 \, dt} \right)^{\frac{1}{2}} ,$$

étant $||\cdot||$ la norme Euclidienne d'un vecteur, $\mathbf{u}(t)$ la solution obtenue numeriquement au temps t et $\hat{\mathbf{u}}(t)$ l'approximation (par la méthode de Crank-Nicholson) de la solution exacte au même temps. L'integrale qui apparaît dans la définition de l'erreur peut être calculé en utilisant la fonction `simpson(u, x=t)` de la librairie `scipy.integrate`, étant `u` la quantité à intégrer.

Donner un commentaire sur la base des résultats obtenus. Sont-ils en accord avec les attentes théoriques? (**Commentaire 5**)

Aide: pour calculer la norme Euclidienne de la solution $u \in \mathbb{R}^{2\times N_h}$ – étant N_h le nombre de pas temporels – utiliser la commande ***`norm = np.linalg.norm(u, axis=0)`***.

[ ]:
```
# reference solution, approximation of the exact solution
prop = 1 / 2**8
Nh = int(T/(prop*h_ref))
t_ex, u_ex = ## COMPLETE HERE (call method)
u_ex_int = ## COMPLETE HERE (compute norm of exact solution, use simpson and np.
 ↪linalg.norm)

# convergence test
err_FE, err_BE, err_CN = [], [], []

for k in range(1,6):
    prop = 1 / 2**k
    _Nh = int(T/(prop*h_ref))
```

```
    # subsampled exact solution --> USE THIS ONE FOR THE ERROR CALCULATION
    u_ex_cur = u_ex[:, ::2**(8-k)]

    t, u_FE = ## COMPLETE HERE (call method)
    err_FE.append()  ## COMPLETE HERE (compute norm of exact solution, use␣
↪simpson and np.linalg.norm)

    _, u_BE = ## COMPLETE HERE (call method)
    err_BE.append() ## COMPLETE HERE (compute norm of exact solution, use␣
↪simpson and np.linalg.norm)

    _, u_CN = ## COMPLETE HERE (call method)
    err_CN.append() ## COMPLETE HERE (compute norm of exact solution, use␣
↪simpson and np.linalg.norm)
```

[ ]:
```
dts = np.array([h_ref / 2**k for k in range(1,6)])

plt.figure()
plt.loglog(dts, err_FE, '-o', label="FE")
plt.loglog(dts, err_BE, '-o', label="BE")
plt.loglog(dts, err_CN, '-o', label="CN")

plt.loglog(dts, (0.5 * err_BE[0] / dts[0]) * dts, '--',
           color="black", label=r"$\sim h$")
plt.loglog(dts, (0.5 * err_CN[0] / dts[0]**2) * dts**2, '--',
           color="tab:gray", label=r"$\sim h^2$")

plt.grid(which='major', linestyle='--', linewidth=1)
plt.grid(which='minor', linestyle='--', linewidth=0.5)

plt.legend(fontsize=16)
plt.xlabel(r"$h$", fontsize=16)
plt.ylabel("Relative error", fontsize=16)

plt.show()
```

1.5.1 Commentaire 5

TODO

2 Quelques petites questions finales (pas évaluées)

- What types of collaboration strategies did your group use?
  - Work in pairs on different sections.
  - Work individually on different sections.

  - Work together on the same section with one notebook opened.
  - Work together on the same section with multiple notebooks opened.
  - Other (please specify).
- How effective was your collaboration strategy today? Please rate from 1 (not at all) to 5 (very effective).
- How supported did you feel by your TA during the session today? Please rate from 1 (not at all) to 5 (very effective).

Please report your answers here. **Thank you!**