Estimation a posteriori

rapport scéance 1

Victor Baleux

14 septembre 2025

Table des matières

Chapitre 1

Méthode d'Euler explicite sur l'EDO $u'(t) = -\lambda u$, $u(0) = 1$

1.1 Problème et objectif

On considère l'équation différentielle ordinaire (EDO)

$$u'(t) = -\lambda\, u(t), \qquad u(0) = 1, \qquad \lambda = 1, \quad t \in [0, T],\ T = 60 \text{ s},$$

dont la solution exacte est $u_{\text{ex}}(t) = \mathrm{e}^{-\lambda t}$. Nous appliquons la méthode d'Euler explicite et nous analysons (i) la solution numérique et (ii) les erreurs (ponctuelle et intégrée en norme L^2) lorsque le pas de temps Δt varie.

1.2 Code Python utilisé

Le script suivant (`Euler_ODE_Errors.py`) génère les figures et calcule les erreurs. Il intègre l'EDO par Euler explicite, mesure les erreurs L^2 sur $[0, T]$ pour u et pour sa dérivée, et produit trois figures sauvegardées sous: `Comparaison_visuelle.png`, `Erreur_vs_delta_temps.png` et `Erreur_vs_derive.png`.

Ce que fait le code.

- **Intégration numérique.** La fonction `euler_explicite(pb, dt)` réalise l'intégration $u_{n+1} = u_n + \Delta t\,(-\lambda u_n)$ jusqu'à T, en ajustant le tout dernier pas pour tomber exactement à T.
- **Erreurs L^2.** `l2_error_function` calcule $\|u_h - u_{\text{ex}}\|_{L^2(0,T)}$ en intégrant au sens des rectangles à gauche sur chaque intervalle. `l2_error_derivative` calcule $\|u_h' - u_{\text{ex}}'\|_{L^2(0,T)}$ en différenciant u_h sur chaque intervalle et en comparant à la dérivée exacte au point milieu.
- **Pente de convergence.** `convergence_slope` effectue une régression linéaire sur le nuage $\big(\log \Delta t, \log \text{erreur}\big)$ pour estimer la pente (ordre numérique).
- **Figures.** `plot_part1_two_rows` produit une comparaison visuelle (solutions et erreurs) pour deux pas de temps. `plot_part2` trace les erreurs L^2 en fonction de Δt (log–log) et affiche les pentes estimées. `plot_error_vs_exact_derivative`

représente l'erreur ponctuelle $|e(t_n)|$ en fonction de $|u'_{\text{ex}}(t_n)|$, en échelle linéaire puis log–log.

Listing du code:

Listing 1.1 – Script Python `Euler_ODE_Errors.py` générant les figures et les erreurs

```python
"""
Euler_ODE_Errors.py
----------------------------------------------
EDO: u'(t) = -λ u(t), u(0)=u0, λ=1.

Figures générées :
1) Comparaison_visuelle.png
   -> 2×2 : haut Δt=1 s, bas Δt=0.001 s ; (gauche) solutions, (droite) erreur
2) Erreur_vs_delta_temps.png
   -> Erreurs L2 (u et u') en fonction de Δt (log-log)
3) Erreur_vs_derivé.png
   -> Scatter de l'erreur ponctuelle |e(t_n)| en fonction de la norme
      de la dérivée exacte |u'_{ex}(t_n)|, pour Δt=1 s et Δt=0.001 s.
      Sous-figure gauche: axes linéaires ; droite: log-log.
"""

import numpy as np
import matplotlib.pyplot as plt
from dataclasses import dataclass
from typing import Tuple, List


@dataclass
class Problem:
    lam: float = 1.0    # λ
    u0: float = 1.0     # condition initiale
    T: float = 60.0     # horizon temporel (1 minute)


def euler_explicite(pb: Problem, dt: float) -> Tuple[np.ndarray, np.ndarray, np.ndarray]:
    """
    Intègre u' = -λ u par Euler explicite avec pas nominal dt jusqu'à T.
    Le dernier pas est ajusté (si nécessaire) pour tomber exactement à T.
    Retourne:
        t : instants (taille N+1)
        u : solution numérique aux instants t
        dts: taille de chaque intervalle (longueur N), avec dernier dt possiblement ajusté
    """
    T = pb.T
    lam = pb.lam
    u0 = pb.u0

    N_full = int(np.floor(T / dt))
    t_list = [0.0]
    u_list = [u0]

    assert dt < 2.0/lam, f"Instable pour Euler explicite: λΔt={lam*dt:.3g} (attendu < 2)."

    # Pas réguliers de taille dt
    for _ in range(N_full):
```

```
        un = u_list[-1]
        un1 = un + dt * (-lam * un)
        u_list.append(un1)
        t_list.append(t_list[-1] + dt)

    # Dernier pas ajusté pour atteindre exactement T (si besoin)
    t_current = t_list[-1]
    dt_last = T - t_current
    dts = [dt] * N_full  # liste des pas
    if dt_last > 1e-14:
        un = u_list[-1]
        un1 = un + dt_last * (-lam * un)
        u_list.append(un1)
        t_list.append(T)
        dts.append(dt_last)

    t = np.array(t_list, dtype=float)
    u = np.array(u_list, dtype=float)
    dts = np.array(dts, dtype=float)
    return t, u, dts


def u_exact(t: np.ndarray, pb: Problem) -> np.ndarray:
    return pb.u0 * np.exp(-pb.lam * t)


def l2_error_function(t: np.ndarray, u_num: np.ndarray, dts: np.ndarray, pb: Problem) -> float:
    ue = u_exact(t, pb)
    e_left = u_num[:-1] - ue[:-1]      # erreur évaluée au bord gauche de chaque intervalle
    return float(np.sqrt(np.sum((e_left**2) * dts)))


def l2_error_derivative(t: np.ndarray, u_num: np.ndarray, dts: np.ndarray, pb: Problem) -> float:
    # dérivée numérique par intervalle
    du = np.diff(u_num)
    uprime_num = du / dts  # taille N
    # points milieux de chaque intervalle
    t_mid = t[:-1] + 0.5 * dts
    # dérivée exacte aux milieux
    uprime_ex = -pb.lam * u_exact(t_mid, pb)
    eprime = uprime_num - uprime_ex
    return float(np.sqrt(np.sum((eprime**2) * dts)))


def convergence_slope(dts: np.ndarray, errs: np.ndarray) -> float:
    x = np.log(dts)
    y = np.log(errs)
    A = np.vstack([x, np.ones_like(x)]).T
    slope, _ = np.linalg.lstsq(A, y, rcond=None)[0]
    return float(slope)


def plot_part1_two_rows(pb: Problem,
                        dt_top: float = 1.0,
                        dt_bottom: float = 1e-3,
```

```
                         savepath: str = "Visual_comparison.png") -> None:
    """
    Figure 2x2 : top = Δt=1 s ; bottom = Δt=0.001 s.
    Colonnes: (gauche) solutions ; (droite) erreur ponctuelle.
    """
    # TOP (Δt = 1 s)
    t1, u1, dts1 = euler_explicite(pb, dt_top)
    ue1 = u_exact(t1, pb)
    err1 = np.abs(u1 - ue1)

    # BOTTOM (Δt = 0.001 s)
    t2, u2, dts2 = euler_explicite(pb, dt_bottom)
    ue2 = u_exact(t2, pb)
    err2 = np.abs(u2 - ue2)

    fig, axes = plt.subplots(2, 2, figsize=(12, 8))
    (ax00, ax01), (ax10, ax11) = axes

    # Top-left: solutions Δt=1 s
    ax00.plot(t1, ue1, label="Solution exacte $u_{ex}(t)$")
    ax00.plot(t1, u1, marker="o", linestyle="--", label=r"Euler $\Delta t=1$ s")
    ax00.set_xlabel("Temps t (s)")
    ax00.set_ylabel("Amplitude u(t)")
    ax00.set_title("Solutions -- Δt = 1 s")
    ax00.grid(True, alpha=0.3)
    ax00.legend()

    # Top-right: erreur Δt=1 s
    ax01.plot(t1, err1, marker="o", linestyle="-", label=r"$|e(t_n)|$")
    ax01.set_xlabel("Temps t (s)")
    ax01.set_ylabel("Erreur ponctuelle")
    ax01.set_title("Erreur -- Δt = 1 s")
    ax01.grid(True, alpha=0.3)
    ax01.legend()

    # Bottom-left: solutions Δt=0.001 s
    ax10.plot(t2, ue2, label="Solution exacte $u_{ex}(t)$")
    ax10.plot(t2, u2, linestyle="-", linewidth=1.0, label=r"Euler $\Delta t=0.001$ s")
    ax10.set_xlabel("Temps t (s)")
    ax10.set_ylabel("Amplitude u(t)")
    ax10.set_title("Solutions -- Δt = 0.001 s")
    ax10.grid(True, alpha=0.3)
    ax10.legend()

    # Bottom-right: erreur Δt=0.001 s
    ax11.plot(t2, err2, linestyle="-", linewidth=1.0, label=r"$|e(t_n)|$")
    ax11.set_xlabel("Temps t (s)")
    ax11.set_ylabel("Erreur ponctuelle")
    ax11.set_title("Erreur -- Δt = 0.001 s")
    ax11.grid(True, alpha=0.3)
    ax11.legend()

    fig.suptitle("u'(t) = -λ u, u(0)=1 ; λ=1 -- Comparaison visuelle Δt", y=0.98)
    fig.tight_layout(rect=[0, 0, 1, 0.96])
    fig.savefig(savepath, dpi=150)


```

```python
def plot_part2(pb: Problem, n_steps: int = 20, dt_min: float = 1e-3, dt_max: float = 1.0,
               savepath: str = "L2_error_vs_deta_time.png") -> None:
    """
    Δt décroissants de 1 s à 0.001 s (échelle logarithmique), 20 valeurs.
    Trace ||e||_L2 et ||e'||_L2 en fonction de Δt (log-log).
    """
    dts_list = np.logspace(np.log10(dt_max), np.log10(dt_min), n_steps)
    errL2_u: List[float] = []
    errL2_du: List[float] = []

    for dt in dts_list:
        t, u_num, dts = euler_explicite(pb, dt)
        errL2_u.append(l2_error_function(t, u_num, dts, pb))
        errL2_du.append(l2_error_derivative(t, u_num, dts, pb))

    dts_arr = np.array(dts_list, dtype=float)
    errL2_u = np.array(errL2_u, dtype=float)
    errL2_du = np.array(errL2_du, dtype=float)

    slope_u = convergence_slope(dts_arr, errL2_u)
    slope_du = convergence_slope(dts_arr, errL2_du)

    fig, ax = plt.subplots(1, 1, figsize=(6.5, 4.5))
    ax.loglog(dts_arr, errL2_u, marker="o", linestyle="-", label=r"$\|u_h-u_{ex}\|_{L^2(0,T)}$")
    ax.loglog(dts_arr, errL2_du, marker="s", linestyle="--", label=r"$\|u'_h-u'_{ex}\|_{L^2(0,T)}$")
    ax.set_xlabel("Pas de temps Δt (s)")
    ax.set_ylabel("Erreur L2 sur [0, T]")
    ax.set_title(f"Erreurs L2 vs Δt -- pentes ≈ {slope_u:.2f} (u), {slope_du:.2f} (u')")
    ax.grid(True, which="both", alpha=0.3)
    ax.legend()
    fig.tight_layout()
    fig.savefig(savepath, dpi=150)

    # Impression console (utile pour le rapport)
    print(f"Pente de convergence (log-log) pour ||u_h - u_ex||_L2 : {slope_u:.4f}")
    print(f"Pente de convergence (log-log) pour ||u'_h - u'_ex||_L2 : {slope_du:.4f}")


def plot_error_vs_exact_derivative(pb: Problem,
                                   dts_to_show: List[float] = [1.0, 1e-3],
                                   savepath: str = "Error_vs_exact_derivative.png") -> None:
    """
    Scatter: erreur ponctuelle |e(t_n)| en fonction de |u'_{ex}(t_n)|,
    pour plusieurs pas de temps (par défaut: Δt=1 s et Δt=0.001 s).
    Deux sous-graphes: (gauche) axes linéaires, (droite) log-log.
    """
    fig, (ax_lin, ax_log) = plt.subplots(1, 2, figsize=(12, 4.5))

    for dt in dts_to_show:
        t, u_num, dts = euler_explicite(pb, dt)
        ue = u_exact(t, pb)
```

```
        err = np.abs(u_num - ue)
        deriv_norm = np.abs(-pb.lam * ue)  # = pb.lam * |ue|

        ax_lin.scatter(deriv_norm, err, s=10, alpha=0.6, label=fr"$\Delta t={dt:g}$ s")
        ax_log.loglog(deriv_norm + 1e-16, err + 1e-16, marker="o", linestyle="", markersize=3, alpha=0.6, label=fr"$\Delta t={dt:g}$ s")

    ax_lin.set_xlabel(r"$|u'_{ex}(t_n)|$")
    ax_lin.set_ylabel(r"$|e(t_n)|$")
    ax_lin.set_title("Erreur vs norme de la dérivée exacte (linéaire)")
    ax_lin.grid(True, alpha=0.3)
    ax_lin.legend()

    ax_log.set_xlabel(r"$|u'_{ex}(t_n)|$")
    ax_log.set_ylabel(r"$|e(t_n)|$")
    ax_log.set_title("Erreur vs norme de la dérivée exacte (log-log)")
    ax_log.grid(True, which="both", alpha=0.3)
    ax_log.legend()

    fig.tight_layout()
    fig.savefig(savepath, dpi=150)


def main():
    pb = Problem(lam=1.0, u0=1.0, T=60.0)

    # Partie 1 : deux rangées: Δt = 1 s (haut) et Δt = 0.001 s (bas)
    plot_part1_two_rows(pb, dt_top=1.0, dt_bottom=1e-3,
                        savepath="Comparaison_visuelle.png")

    # Partie 2 : erreur L2 en fonction de Δt (20 valeurs entre 1 et 1e-3)
    plot_part2(pb, n_steps=20, dt_min=1e-3, dt_max=1.0,
               savepath="Erreur_vs_delta_temps.png")

    # Partie 3 : Erreur ponctuelle vs norme de la dérivée exacte
    plot_error_vs_exact_derivative(pb, dts_to_show=[1.0, 1e-3],
                                   savepath="Erreur_vs_derivé.png")


if __name__ == "__main__":
    main()
```

1.3 Comparaison visuelle pour deux pas de temps

La figure 1.1 montre, en haut, la solution exacte et la solution d'Euler pour $\Delta t = 1$ s, ainsi que l'erreur ponctuelle correspondante; en bas, les mêmes quantités pour $\Delta t = 0{,}001$ s. On observe :

— pour $\Delta t = 1$ s, une dissipation numérique très forte dès les premiers instants: le schéma est stable (car $\lambda\Delta t = 1 < 2$) mais peu précis; l'erreur atteint un maximum au début puis décroît ensuite car u_{ex} décroît;

— pour $\Delta t = 0{,}001$ s, la solution d'Euler colle à la solution exacte, et l'erreur ponctuelle est de l'ordre de 10^{-6}–10^{-4} au début puis s'éteint très rapidement au fur et à mesure que la solution exacte décroît.

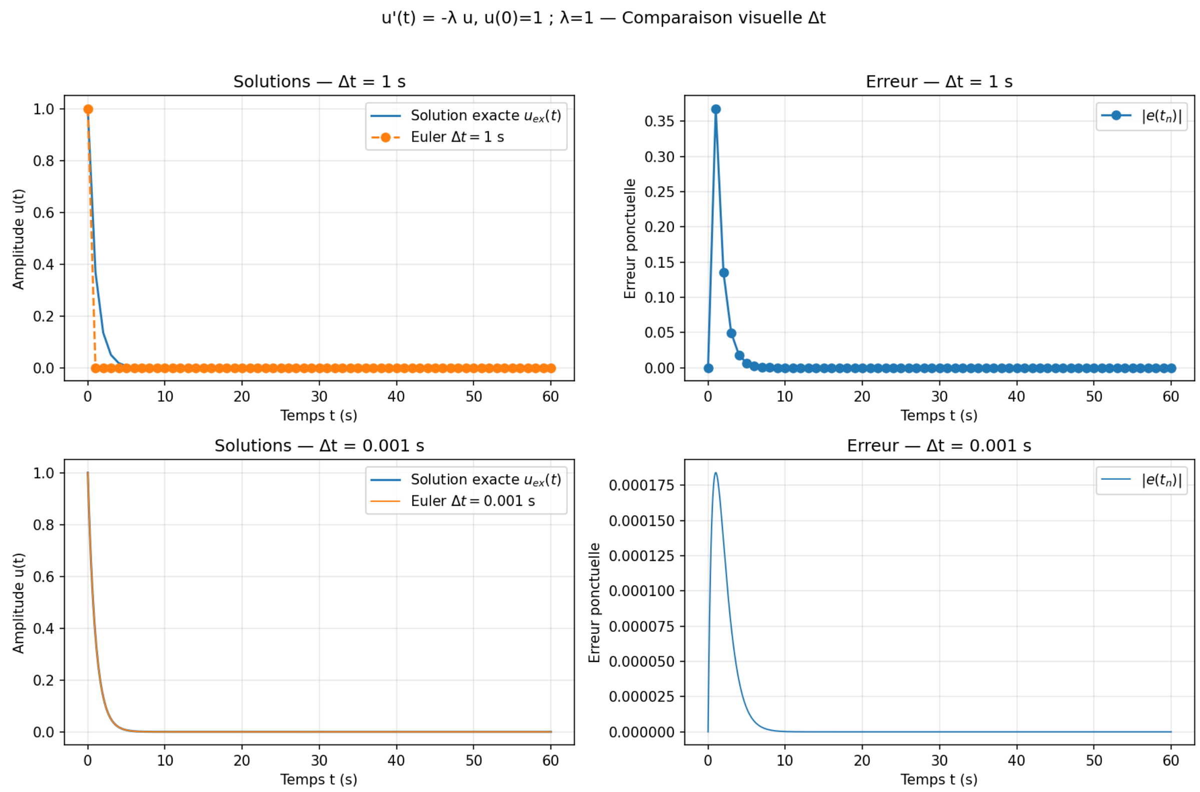

FIGURE 1.1 – Comparaison visuelle des solutions et de l'erreur ponctuelle pour $\Delta t = 1$ s (haut) et $\Delta t = 0{,}001$ s (bas).

1.4 Erreurs L^2 en fonction de Δt

La figure 1.2 présente les erreurs L^2 (pour u en trait plein et pour u' en tirets) en fonction du pas de temps, sur une échelle log–log. La régression linéaire renvoie des pentes voisines de 1 (ici ≈ 1.04 pour u et ≈ 1.06 pour u'), ce qui est conforme à l'*ordre 1* attendu pour la méthode d'Euler explicite. Autrement dit, en divisant Δt par 10, l'erreur globale décroît approximativement d'un facteur 10.

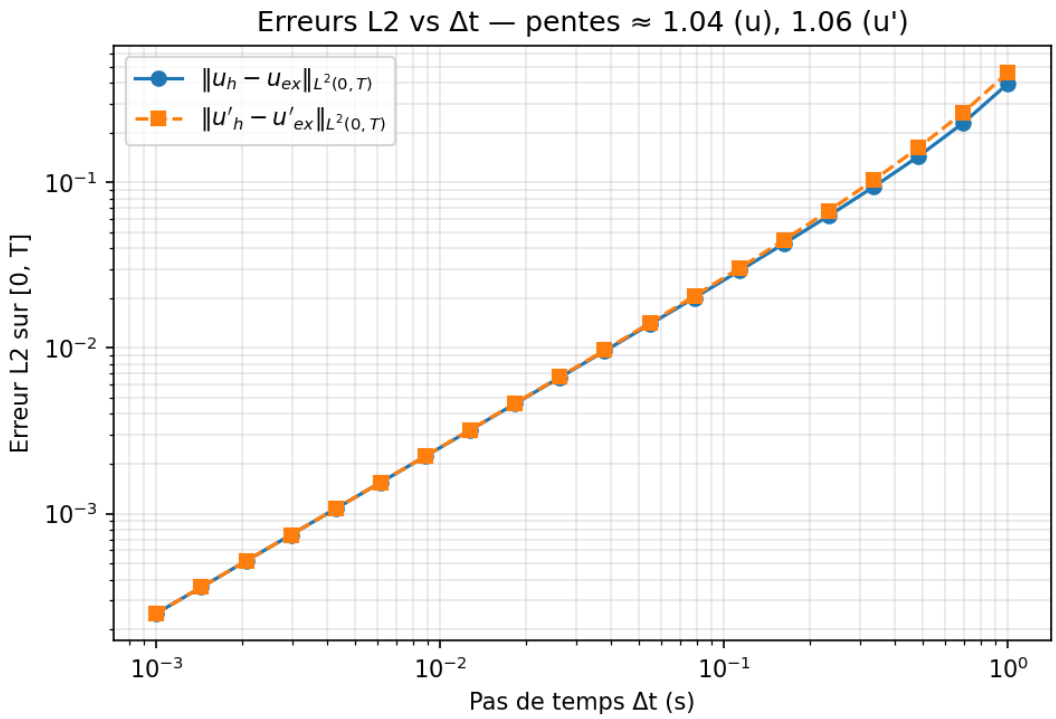

FIGURE 1.2 – Erreurs $\|u_h - u_{\mathrm{ex}}\|_{L^2(0,T)}$ et $\|u'_h - u'_{\mathrm{ex}}\|_{L^2(0,T)}$ versus Δt; les pentes log–log mesurées sont proches de 1 (ordre d'Euler).

1.5 Erreur ponctuelle vs norme de la dérivée exacte

Enfin, la figure 1.3 représente le nuage de points *erreur ponctuelle* $|e(t_n)|$ en fonction de la quantité $|u'_{\mathrm{ex}}(t_n)|$ pour deux pas de temps ($\Delta t = 1$ s et $\Delta t = 10^{-3}$ s), à gauche en échelle linéaire et à droite en échelle log–log.

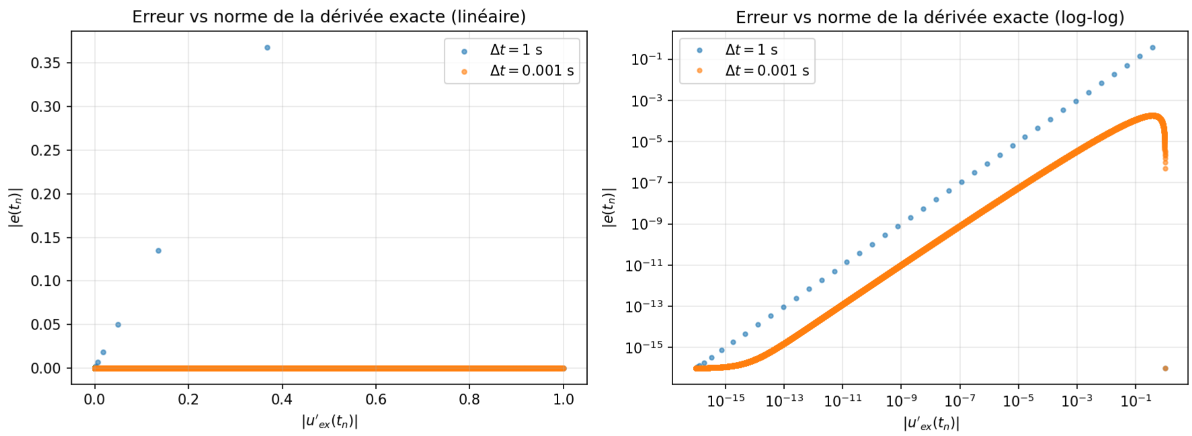

FIGURE 1.3 – Erreur ponctuelle $|e(t_n)|$ en fonction de $|u'_{\mathrm{ex}}(t_n)|$ (échelles linéaire et log–log) pour deux pas de temps. La corrélation forte entre erreur et norme de la dérivée est manifeste; l'"amélioration" en fin d'intervalle provient du fait que $|u'_{\mathrm{ex}}|$ devient très petit près du bord, pas d'un changement d'ordre de la méthode.

Point important à retenir. Cette figure met en évidence une limite inhérente au schéma d'Euler : **la précision locale dépend directement de la norme de la dérivée de la solution**. Lorsque $|u'_{\mathrm{ex}}(t_n)|$ est grand (c'est le cas aux premiers instants lorsque u est encore loin de zéro), l'erreur ponctuelle est la plus élevée, *même si l'on raffine le pas de temps*. En revanche, lorsque $|u'_{\mathrm{ex}}(t_n)|$ devient très petit (vers la fin de l'intervalle, la solution est proche de zéro), l'erreur semble "s'améliorer" naturellement. Il ne s'agit pas d'un gain miraculeux de la méthode, mais d'un effet de *bord* : la condition imposée au bord (solution qui tend vers 0 et pas final ajusté) rend $|u'_{\mathrm{ex}}|$ petit, donc l'erreur locale diminue mécaniquement. La conclusion pratique est que, pour un même Δt, la zone où la dérivée varie rapidement restera la plus difficile pour Euler explicite; diminuer Δt réduit l'erreur globale mais ne supprime pas cette dépendance à $|u'|$.

Chapitre 2

Convection–diffusion avec source gaussienne

2.1 Résumé

Ce document explique le fonctionnement du script `convection_diffusion.py` et commente l'image `triple_panel.png` qu'il génère. Le problème résolu est une équation de convection–diffusion–(réaction) 2D sur le carré unité, avec conditions de Dirichlet imposées *uniquement* sur les bords **entrants** au sens de l'advection, et conditions de type Neumann homogène ailleurs. Le script produit une solution numérique, une solution de référence plus fine, et des cartes d'erreur ponctuelle ainsi que des normes L^2.

2.2 L'équation et les données

On considère

$$\partial_t u + \mathbf{V} \cdot \nabla u - \nu \Delta u = -\lambda\, u + f(x, y), \qquad (x, y) \in (0, 1)^2,\ t \in (0, T],$$

avec $\mathbf{V} = (v_1, v_2)$, la viscosité ν, le terme de réaction λ (nul ici) et une source gaussienne $f(x, y) = T_c \exp\big(-k\|(x, y) - \mathbf{s}_c\|^2\big)$ centrée en $\mathbf{s}_c = (0{,}35, 0{,}55)$. Les paramètres utilisés par défaut sont $T = 1$, $\mathbf{V} = (1,\ 0{,}3)$, $\nu = 10^{-2}$, $u_{\text{in}} = 0$, $T_c = 1$, $k = 80$, sur une grille grossière 61×61; une référence est calculée sur une grille deux fois plus fine dans chaque direction.

2.3 Discrétisation spatiale

- **Diffusion / réaction** : schéma à 5 points pour $-\nu\Delta u$ et ajout de λu. Les nœuds marqués Dirichlet (bords entrants) sont traités en imposant la diagonale unité dans la matrice (ligne identité), ce qui fixe $u = u_{\text{in}}$ à ces endroits. Sur les bords non-Dirichlet, une formule à un seul voisin implémente de facto une condition de Neumann homogène.
- **Advection** : dérivées amont (*upwind*) en x et y. Les valeurs aux bords nécessaires par l'amont sont prises égales à u_{in}.

2.4 Schéma en temps (IMEX)

Le script utilise un schéma **IMEX** : l'advection et la source f sont traitées explicitement, tandis que diffusion et réaction sont implicites. À chaque pas de temps Δt, on forme d'abord

$$u^\star = u^n + \Delta t \,(\text{advection}(u^n) + f),$$

puis on résout linéairement

$$\Big(\tfrac{1}{\Delta t} + \lambda - \nu\Delta\Big)u^{n+1} = \tfrac{1}{\Delta t}\, u^\star,$$

avec les lignes Dirichlet déjà mises à l'identité. La matrice creuse est factorisée une seule fois (`splu`), puis réutilisée à chaque pas. Le pas de temps est choisi par une condition CFL advection (min de $\Delta x/|v_1|$ et $\Delta y/|v_2|$) et ajusté pour tomber exactement à T.

2.5 Code Python utilisé

Listing 2.1 – `convection_diffusion.py` : résolution IMEX, calcul des erreurs et génération du triptyque.

```python
"""
Convection-Diffusion(-Réaction) 2D (Dirichlet uniquement aux bords entrants).
- Génère **uniquement** un triptyque (collage) des 3 vues :
  solution, erreur sur u, erreur sur la norme du gradient.
- Sauvegarde **dans le même dossier que ce script** et affiche le triptyque.

Équation : u_t + V·∇u − ν Δu = −λ u + f(x,y),
           f(x,y) = Tc · exp(−k · ‖(x,y) − s_c‖²).
"""

from io import BytesIO
from pathlib import Path
import numpy as np
import scipy.sparse as sp
import scipy.sparse.linalg as spla
import matplotlib.pyplot as plt

def bords_entrants(v1, v2):
    """Côtés entrants (V·n < 0) pour V=(v1,v2)."""
    return (v1 > 0), (v1 < 0), (v2 > 0), (v2 < 0)  # gauche, droite, bas, haut

def masque_dirichlet(Nx, Ny, inflow):
    in_left, in_right, in_bot, in_top = inflow
    m = np.zeros((Ny, Nx), dtype=bool)
    if in_left:  m[:, 0]  = True
    if in_right: m[:, -1] = True
    if in_bot:   m[0, :]  = True
    if in_top:   m[-1, :] = True
    return m

def assemble_operateur(Nx, Ny, dx, dy, dt, nu, lam, mD):
    alpha = 1.0/dt + lam
    N = Nx*Ny
    rows, cols, vals = [], [], []
```

```python

    def idg(i,j): return i + Nx*j

    for j in range(Ny):
        for i in range(Nx):
            p = idg(i,j)
            if mD[j,i]:
                rows.append(p); cols.append(p); vals.append(1.0)
                continue

            diag = alpha
            # x
            if i == 0:
                diag += nu*(1.0/dx**2)
                rows.append(p); cols.append(idg(i+1,j)); vals.append(-nu*(1.0/dx**2))
            else:
                rows.append(p); cols.append(idg(i-1,j)); vals.append(-nu*(1.0/dx**2))
                diag += nu*(1.0/dx**2)
            if i == Nx-1:
                diag += nu*(1.0/dx**2)
                rows.append(p); cols.append(idg(i-1,j)); vals.append(-nu*(1.0/dx**2))
            else:
                rows.append(p); cols.append(idg(i+1,j)); vals.append(-nu*(1.0/dx**2))
                diag += nu*(1.0/dx**2)
            # y
            if j == 0:
                diag += nu*(1.0/dy**2)
                rows.append(p); cols.append(idg(i,j+1)); vals.append(-nu*(1.0/dy**2))
            else:
                rows.append(p); cols.append(idg(i,j-1)); vals.append(-nu*(1.0/dy**2))
                diag += nu*(1.0/dy**2)
            if j == Ny-1:
                diag += nu*(1.0/dy**2)
                rows.append(p); cols.append(idg(i,j-1)); vals.append(-nu*(1.0/dy**2))
            else:
                rows.append(p); cols.append(idg(i,j+1)); vals.append(-nu*(1.0/dy**2))
                diag += nu*(1.0/dy**2)

            rows.append(p); cols.append(p); vals.append(diag)

    return sp.csr_matrix((vals, (rows, cols)), shape=(N, N))

def advection_amont(u, v1, v2, dx, dy, uL, uR, uB, uT):
    Ny, Nx = u.shape
    dudx = np.zeros_like(u)
    dudy = np.zeros_like(u)
    # x
    if v1 >= 0:
        dudx[:, 1:] = (u[:, 1:] - u[:, :-1]) / dx
        dudx[:, 0]  = (u[:, 0]  - uL[:, 0]) / dx
```

```python
    else:
        dudx[:, :-1] = (u[:, 1:] - u[:, :-1]) / dx
        dudx[:, -1]  = (uR[:, -1] - u[:, -1]) / dx
    # y
    if v2 >= 0:
        dudy[1:, :] = (u[1:, :] - u[:-1, :]) / dy
        dudy[0,  :] = (u[0,  :] - uB[0, :]) / dy
    else:
        dudy[:-1, :] = (u[1:, :] - u[:-1, :]) / dy
        dudy[-1,  :] = (uT[-1, :] - u[-1, :]) / dy
    return -(v1*dudx + v2*dudy)

def source_gauss(x, y, Tc, k, sc):
    X, Y = np.meshgrid(x, y, indexing='xy')
    return Tc * np.exp(-k * ((X - sc[0])**2 + (Y - sc[1])**2))

def norme_grad(u, dx, dy):
    Ny, Nx = u.shape
    dudx = np.zeros_like(u); dudy = np.zeros_like(u)
    dudx[:, 1:-1] = (u[:, 2:] - u[:, :-2]) / (2*dx)
    dudy[1:-1, :] = (u[2:, :] - u[:-2, :]) / (2*dy)
    dudx[:, 0]  = (u[:, 1] - u[:, 0]) / dx
    dudx[:, -1] = (u[:, -1] - u[:, -2]) / dx
    dudy[0, :]  = (u[1, :] - u[0, :]) / dy
    dudy[-1, :] = (u[-1, :] - u[-2, :]) / dy
    return np.sqrt(dudx**2 + dudy**2)

def erreur_L2(champ, ref, dx, dy):
    diff = champ - ref
    return np.sqrt(np.sum(diff**2) * dx * dy)

def resout_imex(ax,bx,ay,by,Nx,Ny,T,v1,v2,nu,lam,u_in,Tc,k,sc,cfl):
    x = np.linspace(ax, bx, Nx)
    y = np.linspace(ay, by, Ny)
    dx = (bx-ax)/(Nx-1); dy = (by-ay)/(Ny-1)

    inflow = bords_entrants(v1, v2)
    mD = masque_dirichlet(Nx, Ny, inflow)

    limites = []
    if abs(v1)>0: limites.append(dx/abs(v1))
    if abs(v2)>0: limites.append(dy/abs(v2))
    dt = cfl*min(limites) if limites else 0.02*min(dx,dy)
    nsteps = int(np.ceil(T/dt)); dt = T/nsteps

    M = assemble_operateur(Nx, Ny, dx, dy, dt, nu, lam, mD)
    lu = spla.splu(M.tocsc())

    f = source_gauss(x, y, Tc, k, sc)
    u = np.zeros((Ny, Nx))

    uL = np.full((Ny, 1), u_in)
    uR = np.full((Ny, 1), u_in)
    uB = np.full((1, Nx), u_in)
    uT = np.full((1, Nx), u_in)

    for _ in range(nsteps):
        adv = advection_amont(u, v1, v2, dx, dy,
```

```
                                np.repeat(uL, Nx, axis=1)[:, :1],
                                np.repeat(uR, Nx, axis=1)[:, -1:],
                                np.repeat(uB, Ny, axis=0)[:1, :],
                                np.repeat(uT, Ny, axis=0)[-1:, :])
        u_star = u + dt*(adv + f)
        rhs = (u_star/dt).ravel()
        rhs[mD.ravel()] = u_in
        u = lu.solve(rhs).reshape(Ny, Nx)

    return x, y, u, {"dt": dt, "nsteps": nsteps}

def figure_as_image(plotter, figsize=(6,5), dpi=160):
    """Crée une figure Matplotlib via la fonction plotter(ax) et renvoie son image
     PIL en mémoire."""
    import PIL.Image as Image
    fig, ax = plt.subplots(figsize=figsize)
    plotter(ax)
    buf = BytesIO()
    fig.tight_layout()
    fig.savefig(buf, format="png", dpi=dpi)
    plt.close(fig)
    buf.seek(0)
    return Image.open(buf).convert("RGB")

def collage_horizontal(images, outpath):
    """Colle des images PIL horizontalement et enregistre le résultat."""
    from PIL import Image
    h = max(im.size[1] for im in images)
    imgs = [im.resize((int(im.size[0]*h/im.size[1]), h)) for im in images]
    W = sum(im.size[0] for im in imgs)
    canvas = Image.new("RGB", (W, h), (255,255,255))
    xoff = 0
    for im in imgs:
        canvas.paste(im, (xoff, 0)); xoff += im.size[0]
    canvas.save(outpath)
    return canvas

def run():
    # ----- Paramètres -----
    ax, bx, ay, by = 0.0, 1.0, 0.0, 1.0
    Nx, Ny = 61, 61
    T = 1.0
    v1, v2 = 1.0, 0.3
    nu, lam = 0.01, 0.0
    u_in = 0.0
    Tc, k = 1.0, 80.0
    sc = (0.35, 0.55)
    cfl = 0.45

    # >>> Enregistrer dans le **même dossier que le script** <<<
    outdir = Path(__file__).parent
    outdir.mkdir(parents=True, exist_ok=True)
    outfile = outdir / "triple_panel.png"

    # Solve coarse
    x, y, uC, infoC = resout_imex(ax,bx,ay,by,Nx,Ny,T,v1,v2,nu,lam,u_in,Tc,k,sc,
    cfl)
    dxC, dyC = (bx-ax)/(Nx-1), (by-ay)/(Ny-1)
```

```python

    # Reference fine
    NxF, NyF = 2*(Nx-1)+1, 2*(Ny-1)+1
    xF, yF, uF, infoF = resout_imex(ax,bx,ay,by,NxF,NyF,T,v1,v2,nu,lam,u_in,Tc,k,
    sc,cfl)
    uF_on_C = uF[::2, ::2]

    # Errors
    e_u_L2 = erreur_L2(uC, uF_on_C, dxC, dyC)
    gC = norme_grad(uC, dxC, dyC)
    dxF, dyF = (bx-ax)/(NxF-1), (by-ay)/(NyF-1)
    gF = norme_grad(uF, dxF, dyF)
    gF_on_C = gF[::2, ::2]
    e_g_L2 = erreur_L2(gC, gF_on_C, dxC, dyC)

    e_u_pw = np.abs(uC - uF_on_C)
    e_g_pw = np.abs(gC - gF_on_C)
    extent = [ax, bx, ay, by]

    # Figures en mémoire
    def plot_solution(axp):
        im = axp.imshow(uC, extent=extent, origin='lower', aspect='auto')
        axp.set_title(f"Solution u(x,y, T={T:.2f})\nV=({v1},{v2}), ν={nu}, λ={lam}
    ")
        axp.set_xlabel("x"); axp.set_ylabel("y")
        plt.colorbar(im, ax=axp, fraction=0.046, pad=0.04)

    def plot_err_u(axp):
        im = axp.imshow(e_u_pw, extent=extent, origin='lower', aspect='auto')
        axp.set_title(f"Erreur ponctuelle |u - u_ref| (‖e‖={e_u_L2:.2e})")
        axp.set_xlabel("x"); axp.set_ylabel("y")
        plt.colorbar(im, ax=axp, fraction=0.046, pad=0.04)

    def plot_err_grad(axp):
        im = axp.imshow(e_g_pw, extent=extent, origin='lower', aspect='auto')
        axp.set_title(f"Erreur ponctuelle |‖∇u‖ - ‖∇u_ref‖| (‖e‖={e_g_L2:.2e})")
        axp.set_xlabel("x"); axp.set_ylabel("y")
        plt.colorbar(im, ax=axp, fraction=0.046, pad=0.04)

    img1 = figure_as_image(plot_solution)
    img2 = figure_as_image(plot_err_u)
    img3 = figure_as_image(plot_err_grad)

    # Collage (unique fichier écrit)
    collage = collage_horizontal([img1, img2, img3], outfile)

    # Affichage
    plt.figure(figsize=(14,5))
    plt.imshow(collage)
    plt.axis("off")
    plt.title("Triptyque : Solution -- Erreur u -- Erreur ‖∇u‖")
    plt.show()

    print("Triptyque enregistré dans :", outfile)

if __name__ == "__main__":
    run()
```

2.6 Évaluation de l'erreur et post-traitement

Après avoir calculé u sur la grille grossière, une solution de *référence* u_{ref} est obtenue sur la grille fine. Elle est sous-échantillonnée sur la grille grossière pour comparer u et u_{ref}. Le script calcule :

— la norme L^2 de l'erreur sur u,

— la norme du gradient $\|\nabla u\|$ (par différences centrées) et la norme L^2 de l'erreur correspondante,

— deux cartes d'erreur ponctuelle : $|u - u_{\text{ref}}|$ et $\big|\|\nabla u\| - \|\nabla u_{\text{ref}}\|\big|$.

Trois figures sont produites en mémoire puis rassemblées en un *triptyque* sauvegardé sous `triple_panel.png`.

2.7 Le triptyque généré et son interprétation

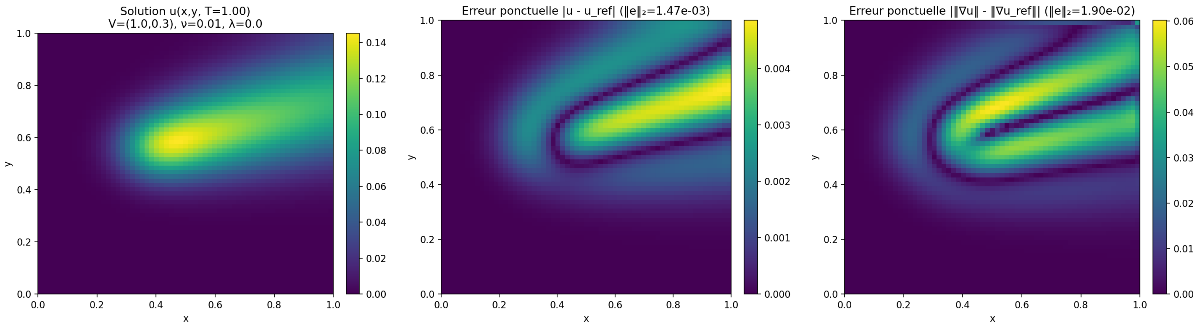

FIGURE 2.1 – De gauche à droite : solution $u(x, y, T)$; erreur ponctuelle $|u - u_{\text{ref}}|$ (la légende rappelle $\|e\|_2$ typiquement $\sim 1{,}5 \times 10^{-3}$ ici) ; erreur ponctuelle sur la norme du gradient $\big|\|\nabla u\| - \|\nabla u_{\text{ref}}\|\big|$ (avec $\|e\|_2 \approx 1{,}9 \times 10^{-2}$).

Commentaire. La solution présente une *bulle* créée par la source gaussienne, advectée le long des lignes de courant de $\mathbf{V} = (1,\ 0{,}3)$ (trajectoire oblique montante), et légèrement *étalée* par la diffusion $\nu > 0$. Les erreurs maximales suivent les zones à fort *cisaillement* (fronts de solution) et les traces obliques sont typiques de l'advection amont : l'erreur sur u reste faible et lisse, tandis que l'erreur sur $\|\nabla u\|$ est plus marquée le long des pentes raides où la diffusion numérique et le pas de maille influent davantage.

Chapitre 3

Simulation numérique d'une équation de convection–diffusion–réaction 1D (conditions de Dirichlet–Neumann)

3.1 Motivation

On souhaite illustrer le comportement d'une quantité scalaire $u(t,x)$ transportée par un flux constant, tout en subissant diffusion et (éventuellement) réaction. Le phénomène est modélisé par l'EDP

$$\partial_t u + v\,\partial_x u - \nu\,\partial_{xx} u + \lambda\, u = f(t,x), \quad x \in (0,L),\ t \in (0,T],$$

avec une condition de Dirichlet à gauche $u(t,0) = u_\ell(t)$, une condition de Neumann à droite $u_x(t,L) = g(t)$, et une condition initiale $u(0,x) = u_0(x)$. Cette étude numérique permet d'observer l'advection d'une bosse initiale, son étalement diffusif et l'influence des conditions aux bords.

3.2 Code Python

Le code ci-dessous met en place un schéma implicite en temps, avec advection *upwind*, diffusion centrale et terme de réaction.

Fichier : `convection_diffusion_1D.py`

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

# ============================================================
# Paramètres du modèle (à ajuster)
# ============================================================
L      = 1.0        # longueur du domaine [0, L]
v      = 0.750        # vitesse de convection (constante)
nu     = 1e-2       # diffusivité
lam    = 0.0        # lambda de réaction (>=0 typiquement)
T      = 1.0        # temps final
N      = 500        # N intervalles => N+1 points de grille
```

```python
dt      = 5e-4        # pas de temps

# Fonctions sources et CL (modifiez librement)
def f(t, x):
    # source volumique (exemple: nulle)
    return 0.0

def ul(t):
    # Dirichlet à gauche (exemple: constant et non nul)
    return 0.0

def g(t):
    # Neumann à droite u_x(t,L)=g(t) (exemple: constant et non nul)
    return 0.0

# Paramètres de la gaussienne de départ (utilisée dans u0)
xc      = 0.2 * L       # centre
sigma   = 0.05 * L      # largeur
A_amp   = 1.0           # amplitude libre de la gaussienne
# ==========================================================


# ==========================================================
# Construction de u0(x) compatible avec u(0)=ul(0) et u_x(L)=g(0)
#     u0(x) = A*G(x) + B*x + C
#     G(x) = exp(-(x-xc)^2/(2*sigma^2))
# Conditions:
#     u0(0)=ul(0)   => C = ul(0) - A*G(0)
#     u0'(L)=g(0)   => u0'(x)=A*(-(x-xc)/sigma^2)G(x) + B
#                      B = g(0) - A*(-(L-xc)/sigma^2)*G(L)
# ==========================================================
def build_u0(x, A=A_amp, xc=xc, sigma=sigma):
    G0 = np.exp(-((x - xc)**2) / (2.0 * sigma**2))
    G0_at_0 = np.exp(-((0.0 - xc)**2) / (2.0 * sigma**2))
    G0_at_L = np.exp(-((L   - xc)**2) / (2.0 * sigma**2))

    B = g(0.0) - A * (-(L - xc) / sigma**2) * G0_at_L
    C = ul(0.0) - A * G0_at_0

    u0 = A * G0 + B * x + C
    return u0


# ==========================================================
# Assemblage du système linéaire implicite:
# (I - dt * L) u^{n+1} = u^n + dt f^{n+1}
# avec L = -v D1_upwind + nu D2 - lam I
# On encode directement A = I + dt*v*D1_upwind - dt*nu*D2 + dt*lam*I
# et on remplace les deux lignes de bord par les CL (Dirichlet/Neumann).
# ==========================================================
def build_matrix(N, L, dt, v, nu, lam):
    dx = L / N
    size = N + 1
    A = np.zeros((size, size))

    # Upwind pour l'advection (au niveau des points i=1..N-1)
    # D1_upwind u_i ~ (u_i - u_{i-1})/dx si v>0
    #               ~ (u_{i+1} - u_i)/dx si v<0
```

```python
    use_backward = (v >= 0.0)

    # Remplissage lignes intérieures i=1..N-1
    for i in range(1, N):
        # Diffusion (D2 central): u_{i-1} - 2 u_i + u_{i+1} sur dx^2
        A[i, i]   += 1.0 + dt * lam + dt * nu * (2.0 / dx**2)
        A[i, i-1] += - dt * nu * (1.0 / dx**2)
        A[i, i+1] += - dt * nu * (1.0 / dx**2)

        # Advection implicite upwind
        if use_backward:
            # D1 ≈ (u_i - u_{i-1})/dx
            A[i, i]   += dt * v * ( 1.0 / dx)
            A[i, i-1] += dt * v * (-1.0 / dx)
        else:
            # D1 ≈ (u_{i+1} - u_i)/dx
            A[i, i]   += dt * v * (-1.0 / dx)
            A[i, i+1] += dt * v * ( 1.0 / dx)

    # Bord gauche (Dirichlet) : u(t,0) = ul(t)
    A[0, :] = 0.0
    A[0, 0] = 1.0

    # Bord droit (Neumann) : (u_N - u_{N-1})/dx = g(t)
    A[N, :] = 0.0
    A[N, N-1] = -1.0 / dx
    A[N, N]   =  1.0 / dx

    return A


def step_rhs(u_prev, t_next, x, dt):
    # RHS b = u^n + dt f(t^{n+1}, x) pour points intérieurs
    b = u_prev.copy()
    b[1:-1] += dt * np.array([f(t_next, xi) for xi in x[1:-1]])

    # CL de Dirichlet au bord gauche
    b[0] = ul(t_next)

    # CL de Neumann au bord droit
    b[-1] = g(t_next)

    return b


def solve_pde(L, v, nu, lam, T, N, dt):
    x = np.linspace(0.0, L, N + 1)
    A = build_matrix(N, L, dt, v, nu, lam)

    # Pré-calcul: factorisation possible (si SciPy dispo), ici solve dense simple
    # Création de u^0 compatible
    u = build_u0(x)

    # Petits checks de compatibilité (informative, tolérance ~1e-8)
    dx = L / N
    dL_num = (u[-1] - u[-2]) / dx
    if abs(u[0] - ul(0.0)) > 1e-8 or abs(dL_num - g(0.0)) > 1e-6:
        print("[Avertissement] u0 n'est pas parfaitement compatible numériquement
```

```
    avec les CL.")

    times_to_store = np.linspace(0.0, T, 5)  # 5 instants (dont t=0 et t=T)
    snapshots = [(0.0, u.copy())]
    t = 0.0
    next_store_idx = 1  # le prochain instant à mémoriser (sauter t=0 déjà stocké)

    while t < T - 1e-12:
        t_next = min(t + dt, T)
        b = step_rhs(u, t_next, x, dt)
        u = np.linalg.solve(A, b)
        t = t_next

        # stocker si on a franchi le prochain jalon
        while next_store_idx < len(times_to_store) and t >= times_to_store[next_store_idx] - 1e-12:
            snapshots.append((times_to_store[next_store_idx], u.copy()))
            next_store_idx += 1

    return x, snapshots


# ==========================================================
# Lancement + visualisation
# ==========================================================
x, snapshots = solve_pde(L, v, nu, lam, T, N, dt)

plt.figure()
for (ti, ui) in snapshots:
    plt.plot(x, ui, label=f"t={ti:.3f}")
plt.xlabel("x")
plt.ylabel("u(t,x)")
plt.title("Convection-Diffusion-Réaction 1D (Dirichlet gauche, Neumann droite)")
plt.legend()
plt.tight_layout()
plt.savefig("Convection_Diffusion_Réaction_1D.png", dpi=300, bbox_inches="tight")
plt.show()
```

3.3 Ce que fait le code et comment il le fait

- **Discrétisation spatiale.** Le domaine $[0, L]$ est maillé uniformément en $N+1$ points. L'advection est discrétisée par un opérateur *upwind* (retard si $v \geq 0$, avance sinon) pour stabiliser le transport. La diffusion utilise une approximation centrale d'ordre 2. Le terme de réaction est traité linéairement.
- **Avancement en temps (implicite).** À chaque pas, on résout

$$(I + \Delta t\, v\, D_1^{\mathrm{up}} - \Delta t\, \nu\, D_2 + \Delta t\, \lambda\, I)\, u^{n+1} = u^n + \Delta t\, f^{n+1}.$$

  La matrice A correspond au membre de gauche; le second membre incorpore aussi les conditions aux limites.
- **Conditions aux limites.** Dirichlet en $x{=}0$ imposée en remplaçant la première ligne de A. Neumann en $x{=}L$ imposée via une différence finie pour $u_x(t, L) = g(t)$.

— **Condition initiale compatible.** $u_0(x)$ est la somme d'une gaussienne et d'une partie affine choisie pour satisfaire $u(0,0) = u_\ell(0)$ et $u_x(0,L) = g(0)$, afin d'éviter des incohérences numériques dès t=0.
— **Sortie graphique.** Cinq *snapshots* $t \in \{0, 0.25, 0.5, 0.75, 1\}$ sont tracés et sauvegardés dans un fichier image.

3.4 Résultat (figure)

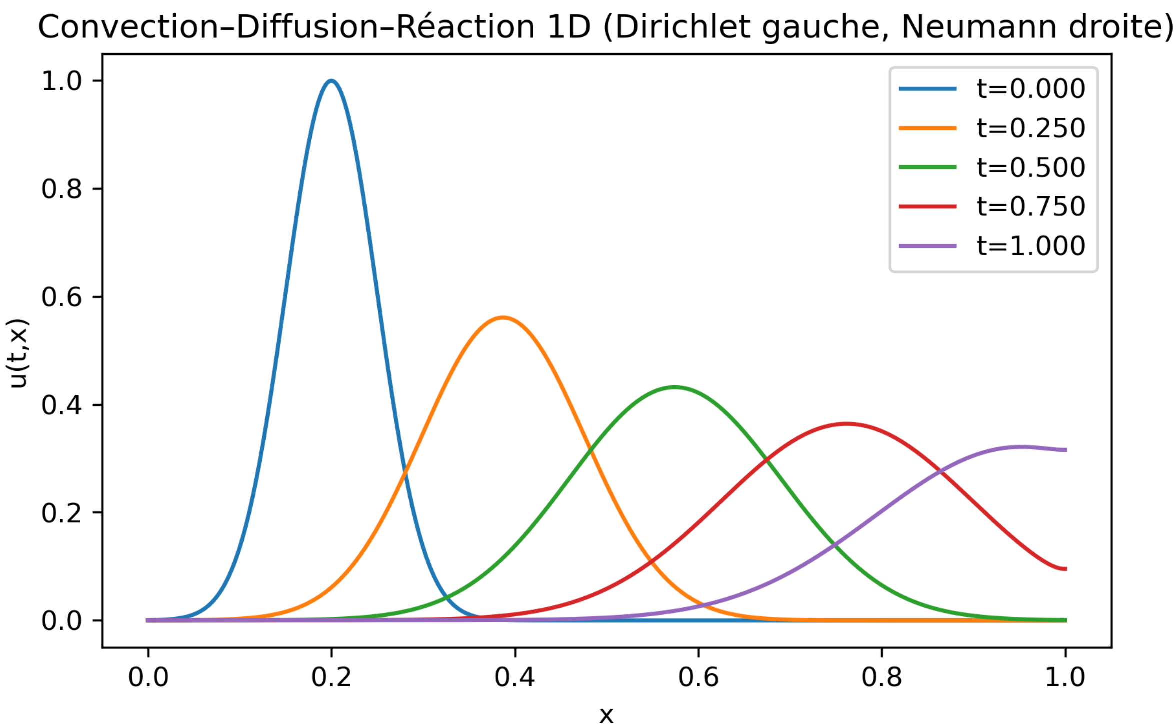

FIGURE 3.1 – Solution numérique $u(t,x)$ pour $v = 0.75$, $\nu = 10^{-2}$, $\lambda = 0$, $L = 1$, avec Dirichlet à gauche et Neumann (flux nul) à droite.

3.5 Commentaire sur la figure

La bosse initiale centrée près de $x \simeq 0.2$ est *advectée* vers la droite à vitesse constante v, tandis que la diffusion l'élargit et en réduit l'amplitude au fil du temps. La condition de Dirichlet maintient la solution proche de zéro à gauche, alors que la condition de Neumann impose un flux nul au bord droit, ce qui se traduit par une pente $\partial_x u \simeq 0$ quand l'onde atteint $x = L$. Avec $\lambda = 0$, l'amplitude décroît uniquement par diffusion; si $\lambda > 0$ on observerait en plus une décroissance exponentielle globale.