Estimation a posteriori

rapport scéance 1

Victor Baleux

25 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 -- version mise à jour
------------------------------------------------
EDO: u'(t) = -λ u(t), u(0)=u0.

Modifs (sept. 2025) :
1) Quadrature L2 : rectangles à gauche remplacés par trapèze / Simpson (si N pair).
2) Stabilité Euler : assertion robuste -- si λ=0 ne rien imposer ; sinon Δt ≤ (2−ε)/λ.
3) Choix des pas : prend Δt = T/N avec N entier (échelonné log) pour éviter un "dernier pas" irrégulier.

Figures générées (inchangées) :
1) Comparaison_visuelle.png
   -> 22 : 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.
"""

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


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


# ------------------------------
#   Intégrateur d'Euler explicite
# ------------------------------

def euler_explicite(pb: Problem, dt: Optional[float] = None, N: Optional[int] = None, eps: float = 1e-12
                    ) -> Tuple[np.ndarray, np.ndarray, np.ndarray]:
    """
    Intègre u' = -λ u par Euler explicite avec pas Δt = T/N (N entier).
    On peut fournir soit dt (approx.), soit directement N. Si dt est fourni,
    on prend N = round(T/dt) puis Δt := T/N pour tomber EXACTEMENT à T.

    Retourne:
        t : instants (taille N+1, réguliers de 0 à T)
        u : solution numérique aux instants t
        dts: tableau de taille N rempli par Δt (tous les pas identiques)
```

```python
    """
    T = pb.T
    lam = pb.lam
    u0 = pb.u0

    if N is None:
        if dt is None:
            raise ValueError("Fournir soit dt, soit N.")
        N = max(1, int(round(T / float(dt))))
    else:
        N = int(N)
        if N <= 0:
            raise ValueError("N doit être un entier positif.")

    dt = T / N  # pas EXACT et régulier
    # Stabilité Euler robuste
    if lam != 0.0 and lam > 0.0:
        bound = (2.0 - eps) / lam
        assert dt <= bound, (
            f"Instable pour Euler explicite: λΔt={lam*dt:.3g} ; "
            f"attendu Δt ≤ (2−ε)/λ ≈ {bound:.6g} (ε={eps:g})."
        )
    # Si λ=0 : ne rien imposer.

    t = np.linspace(0.0, T, N + 1, dtype=float)
    u = np.empty(N + 1, dtype=float)
    u[0] = u0
    for n in range(N):
        u[n + 1] = u[n] + dt * (-lam * u[n])

    dts = np.full(N, dt, dtype=float)
    return t, u, dts


# ------------------------------
#   Outils exacts / quadrature
# ------------------------------

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


def _integrate_uniform(y: np.ndarray, dt: float) -> float:
    """
    Intègre une fonction tabulée y(t_n) sur [0,T] avec pas uniforme dt.
    Utilise Simpson si le nombre d'intervalles N=len(y)-1 est pair, sinon trapèze.
    Retourne l'intégrale numérique (pas la racine).
    """
    N = y.size - 1
    if N <= 0:
        return 0.0
    if N % 2 == 0 and N >= 2:
        # Simpson
        s_odd = np.sum(y[1:N:2])
        s_even = np.sum(y[2:N-1:2]) if N >= 3 else 0.0
        return (dt / 3.0) * (y[0] + y[-1] + 4.0 * s_odd + 2.0 * s_even)
    # Trapèze
    return dt * (0.5 * y[0] + np.sum(y[1:-1]) + 0.5 * y[-1])
```

```


def l2_error_function(t: np.ndarray, u_num: np.ndarray, dts: np.ndarray, pb:
    Problem) -> float:
    """
    ||e||_{L2(0,T)} ≈ ( ∫_0^T |u_num(t)-u_ex(t)|^2 dt )^{1/2}
    Quadrature: Simpson (si possible) sinon trapèze -- pas uniforme Δt = dts[0].
    """
    dt = float(dts[0])
    e = u_num - u_exact(t, pb)
    integ = _integrate_uniform(e**2, dt)
    return float(np.sqrt(integ))


def l2_error_derivative(t: np.ndarray, u_num: np.ndarray, dts: np.ndarray, pb:
    Problem) -> float:
    """
    ||e'||_{L2(0,T)} ≈ ( ∫_0^T |u'_num(t) - u'_ex(t)|^2 dt )^{1/2}
    - u'_num(t_n) : dérivée numérique aux noeuds via différences centrales (np.
    gradient)
    - u'_ex(t_n)  : -λ u_ex(t_n)
    Quadrature: Simpson (si possible) sinon trapèze -- pas uniforme.
    """
    dt = float(dts[0])
    uprime_num = np.gradient(u_num, dt)  # central diff interne, 1er ordre aux
    bords
    uprime_ex = -pb.lam * u_exact(t, pb)
    eprime = uprime_num - uprime_ex
    integ = _integrate_uniform(eprime**2, dt)
    return float(np.sqrt(integ))


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)


# ------------------------------
#   Figures
# ------------------------------

def plot_part1_two_rows(pb: Problem,
                        dt_top: float = 1.0,
                        dt_bottom: float = 1e-3,
                        savepath: str = "Comparaison_visuelle.png") -> None:
    """
    Figure 2x2 : top = Δt=1 s ; bottom = Δt=0.001 s.
    Colonnes: (gauche) solutions ; (droite) erreur ponctuelle.
    Δt est projeté sur T/N (N entier) pour éviter un dernier pas irrégulier.
    """
    # TOP (Δt ≈ 1 s, mais forcé à T/N)
    t1, u1, dts1 = euler_explicite(pb, dt_top)
    ue1 = u_exact(t1, pb)
    err1 = np.abs(u1 - ue1)

```

```python
    # BOTTOM (Δt ≈ 0.001 s, mais forcé à T/N)
    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
    ax00.plot(t1, ue1, label="Solution exacte $u_{ex}(t)$")
    ax00.plot(t1, u1, marker="o", linestyle="--", label=fr"Euler $\Delta t={dts1[0]:g}$ 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
    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
    ax10.plot(t2, ue2, label="Solution exacte $u_{ex}(t)$")
    ax10.plot(t2, u2, linestyle="-", linewidth=1.0, label=fr"Euler $\Delta t={dts2[0]:g}$ 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
    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 -- Comparaison visuelle Δt", y=0.98)
    fig.tight_layout(rect=[0, 0, 1, 0.96])
    fig.savefig(savepath, dpi=150)


def _logspace_integers(n_min: int, n_max: int, n_steps: int) -> np.ndarray:
    """Valeurs entières échelonnées logarithmiquement dans [n_min, n_max]."""
    vals = np.geomspace(max(1, n_min), max(1, n_max), max(2, n_steps))
    ints = np.unique(np.clip(np.round(vals).astype(int), n_min, n_max))
    return ints


def plot_part2(pb: Problem, n_steps: int = 20, dt_min: float = 1e-3, dt_max: float = 1.0,
```

```
               savepath: str = "Erreur_vs_delta_temps.png") -> None:
    """
    Δt définis via Δt = T/N avec N entier échelonné log entre
    N_min = ceil(T/dt_max) et N_max = floor(T/dt_min).
    Trace ||e||_L2 et ||e'||_L2 en fonction de Δt (log-log).
    """
    T = pb.T
    N_min = int(np.ceil(T / dt_max))
    N_max = int(np.floor(T / dt_min))
    if N_max < max(2, N_min):
        raise ValueError("Plage (dt_min, dt_max) trop étroite pour construire des N entiers.")
    N_list = _logspace_integers(N_min, N_max, n_steps)

    dts_arr = []
    errL2_u: List[float] = []
    errL2_du: List[float] = []

    for N in N_list:
        t, u_num, dts = euler_explicite(pb, N=N)
        dts_arr.append(float(dts[0]))
        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_arr, 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 = "Erreur_vs_derivé.png") -> None:
    """
    Scatter: erreur ponctuelle |e(t_n)| en fonction de |u'_{ex}(t_n)|,
```

```python
    pour plusieurs pas de temps (par défaut: Δt=1 s et Δt=0.001 s).
    Deux sous-graphes: (gauche) axes linéaires, (droite) log-log.
    Δt est projeté sur T/N (N entier) pour éviter un dernier pas irrégulier.
    """
    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"$\Delta t={dts[0]:g}$ s")
        ax_log.loglog(deriv_norm + 1e-16, err + 1e-16, marker="o", linestyle="", markersize=3, alpha=0.6, label=fr"$\Delta t={dts[0]:g}$ 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),
    # mais forcés à Δt = T/N exactement.
    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 (N entiers échelonnés log entre dt_max et dt_min)
    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_derive.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.

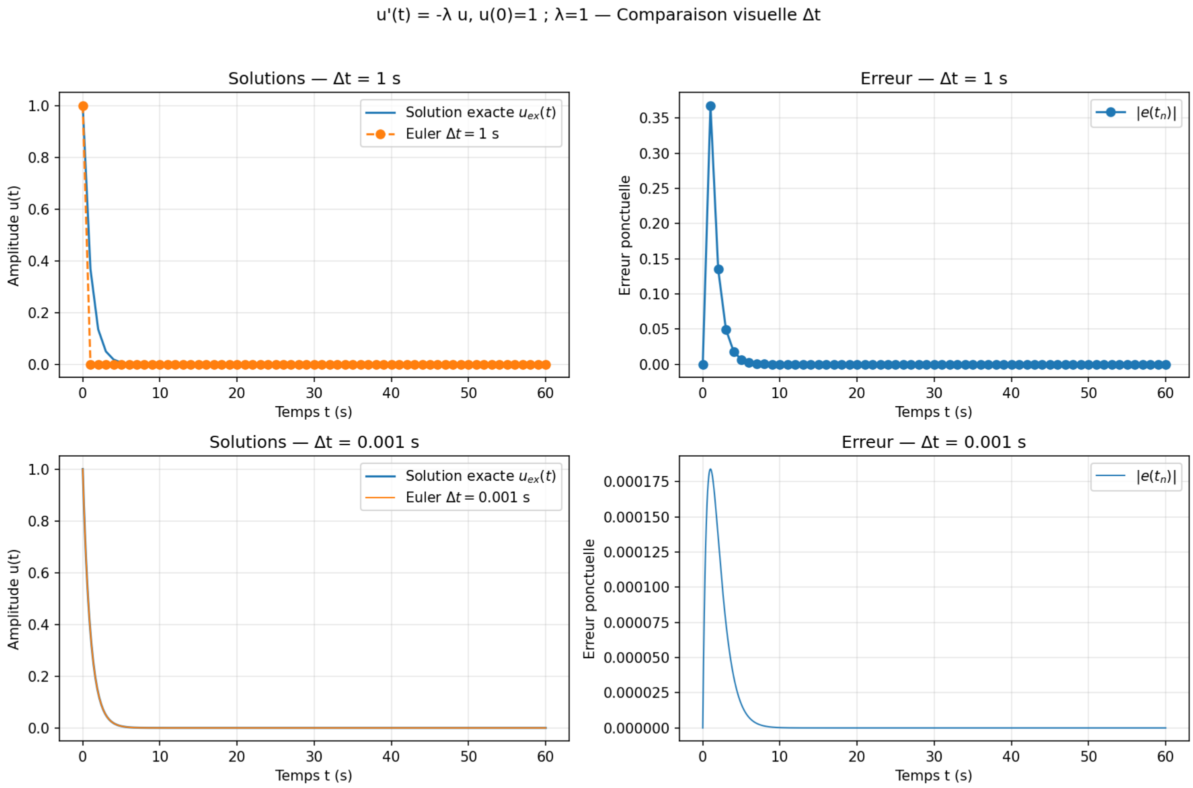

FIGURE 1.1 – Comparaison visuelle des solutions et de l'erreur ponctuelle pour $\Delta t = 1$ s (haut) et $\Delta t = 0{,}001$ s (bas).

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.

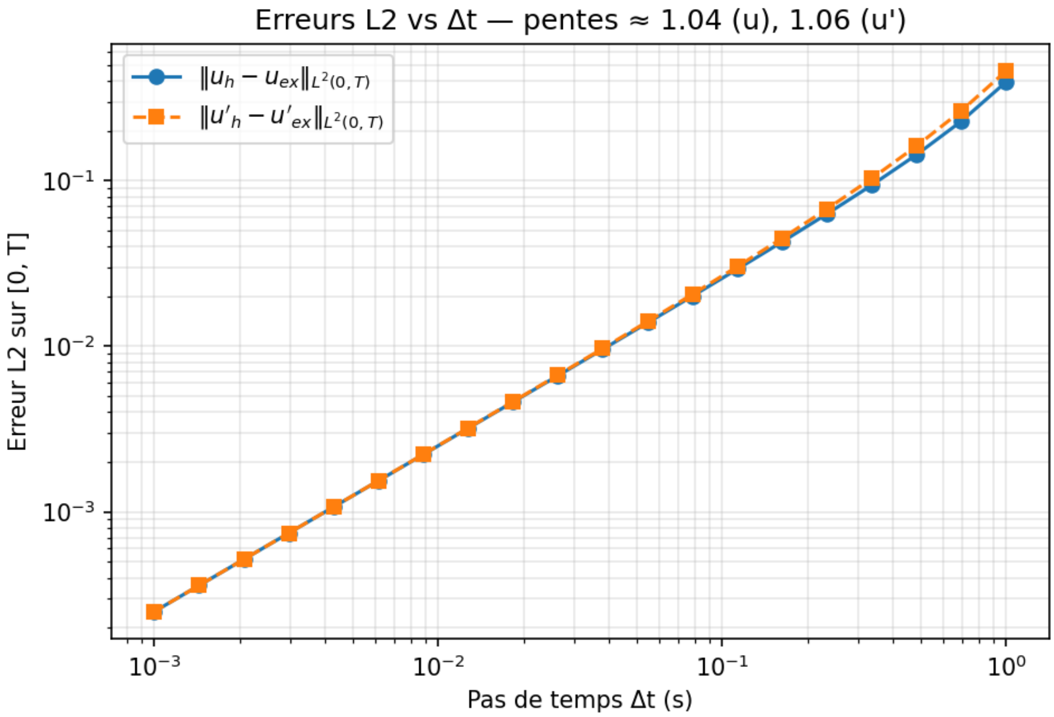

FIGURE 1.2 – Erreurs $\|u_h - u_{\text{ex}}\|_{L^2(0,T)}$ et $\|u'_h - u'_{\text{ex}}\|_{L^2(0,T)}$ versus Δt; les pentes log–log mesurées sont proches de 1 (ordre d'Euler).

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 $|u'_{\text{ex}}(t_n)|$ 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.

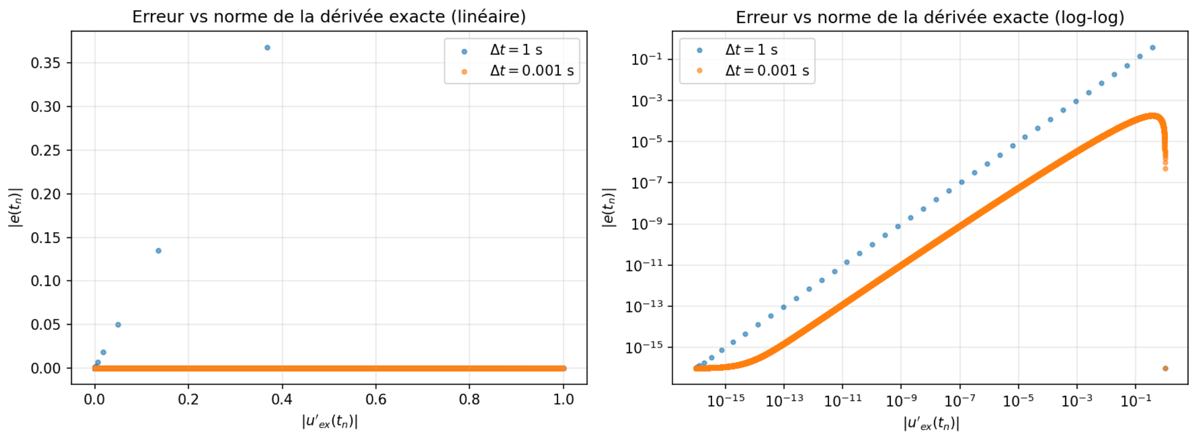

FIGURE 1.3 – Erreur ponctuelle $|e(t_n)|$ en fonction de $|u'_{\text{ex}}(t_n)|$ (échelles linéaire et log–log) pour deux pas de temps. La corrélation forte entre erreur et norme de la dérivée est manifeste; l'"amélioration" en fin d'intervalle provient du fait que $|u'_{\text{ex}}|$ devient très petit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

```

"""
Convection-Diffusion(-Réaction) 2D
----------------------------------
- Diffusion : conditions de Neumann homogènes (∂u/∂n = 0) SUR TOUS LES BORDS,
  traitées dans l'opérateur implicite via des stencils à point fantôme (facteur 2
    sur le voisin intérieur).
- Advection : schéma d'amont (upwind) explicite. Les valeurs aux bords amont sont
  passées **directement** via uL (gauche, taille Ny), uR (droite, Ny), uB (bas, Nx
    ), uT (haut, Nx).
  Pas de `repeat` ni de masques Dirichlet : l'amont impose les valeurs d'entrée.
- Réaction : terme -λ u implicite (lumped dans la diagonale).
- Sortie : un triptyque (solution, erreur u, erreur ‖∇u‖) et une courbe de
    convergence spatiale
  ‖e(u)‖₂ et ‖e(∇u)‖₂ en fonction de h sur 2-3 raffinements (attendu ≈ ordre 1
    avec amont + temps ordre 1).

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

# --------------------- Utilitaires PDE ---------------------

def assemble_operateur(Nx, Ny, dx, dy, dt, nu, lam):
    """
    Assemble M ≈ (I/dt + lam) - nu * Δ  avec Neumann homogène sur le bord.
    Discrétisation en différences finies sur grille cartésienne (indexing 'xy').
```

```
    Aux bords : stencil à point fantôme ⇒ coefficient *2 sur le voisin intérieur.
    """
    alpha = 1.0/dt + lam
    N = Nx*Ny
    rows, cols, vals = [], [], []

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

    cx = nu/(dx*dx) if Nx > 1 else 0.0
    cy = nu/(dy*dy) if Ny > 1 else 0.0

    for j in range(Ny):
        for i in range(Nx):
            p = idg(i,j)
            diag = alpha

            # --- x-direction with homogeneous Neumann ---
            if Nx > 1:
                if i == 0:
                    # u_{-1} = u_{1}  ⇒  Δx u_0 ≈ 2(u_1 - u_0)/dx²
                    diag += 2*cx
                    rows.append(p); cols.append(idg(i+1, j)); vals.append(-2*cx)
                elif i == Nx-1:
                    diag += 2*cx
                    rows.append(p); cols.append(idg(i-1, j)); vals.append(-2*cx)
                else:
                    diag += 2*cx
                    rows.append(p); cols.append(idg(i-1, j)); vals.append(-cx)
                    rows.append(p); cols.append(idg(i+1, j)); vals.append(-cx)

            # --- y-direction with homogeneous Neumann ---
            if Ny > 1:
                if j == 0:
                    diag += 2*cy
                    rows.append(p); cols.append(idg(i, j+1)); vals.append(-2*cy)
                elif j == Ny-1:
                    diag += 2*cy
                    rows.append(p); cols.append(idg(i, j-1)); vals.append(-2*cy)
                else:
                    diag += 2*cy
                    rows.append(p); cols.append(idg(i, j-1)); vals.append(-cy)
                    rows.append(p); cols.append(idg(i, j+1)); vals.append(-cy)

            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):
    """
    Flux d'advection explicite (amont).
    Entrée :
      - u : (Ny, Nx)
      - v1, v2 : vitesses constantes V=(v1,v2)
      - uL (Ny,), uR (Ny,), uB (Nx,), uT (Nx,)
        valeurs aux bords gauche/droite/bas/haut UTILISÉES UNIQUEMENT
        quand la vitesse entre dans le domaine par ce bord.
    Sortie : - (v1 ∂x u + v2 ∂y u) évalué par différences amont.
    """
```

```python
    Ny, Nx = u.shape
    dudx = np.zeros_like(u)
    dudy = np.zeros_like(u)

    # x-direction
    if v1 >= 0:
        # amont = arrière
        if Nx > 1:
            dudx[:, 1:] = (u[:, 1:] - u[:, :-1]) / dx
        # bord gauche utilise uL
        dudx[:, 0] = (u[:, 0] - uL) / dx
    else:
        # amont = avant
        if Nx > 1:
            dudx[:, :-1] = (u[:, 1:] - u[:, :-1]) / dx
        # bord droit utilise uR
        dudx[:, -1] = (uR - u[:, -1]) / dx

    # y-direction
    if v2 >= 0:
        if Ny > 1:
            dudy[1:, :] = (u[1:, :] - u[:-1, :]) / dy
        dudy[0, :] = (u[0, :] - uB) / dy
    else:
        if Ny > 1:
            dudy[:-1, :] = (u[1:, :] - u[:-1, :]) / dy
        dudy[-1, :] = (uT - 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)
    if Nx > 1:
        dudx[:, 1:-1] = (u[:, 2:] - u[:, :-2]) / (2*dx)
        dudx[:, 0]  = (u[:, 1] - u[:, 0]) / dx
        dudx[:, -1] = (u[:, -1] - u[:, -2]) / dx
    if Ny > 1:
        dudy[1:-1, :] = (u[2:, :] - u[:-2, :]) / (2*dy)
        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)

# --------------------- Solveur IMEX ---------------------

def resout_imex(ax,bx,ay,by,Nx,Ny,T,v1,v2,nu,lam,uL,uR,uB,uT,Tc,k,sc,cfl):
    x = np.linspace(ax, bx, Nx)
    y = np.linspace(ay, by, Ny)
    dx = (bx-ax)/(Nx-1) if Nx>1 else 1.0
    dy = (by-ay)/(Ny-1) if Ny>1 else 1.0
```

```

    # CFL simple basé sur l'advection
    limites = []
    if abs(v1)>0 and Nx>1: limites.append(dx/abs(v1))
    if abs(v2)>0 and Ny>1: limites.append(dy/abs(v2))
    dt = cfl*min(limites) if limites else 0.02*min(dx,dy)  # fallback
    nsteps = int(np.ceil(T/dt)); dt = T/nsteps

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

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

    for _ in range(nsteps):
        adv = advection_amont(u, v1, v2, dx, dy, uL, uR, uB, uT)
        u_star = u + dt*(adv + f)
        rhs = (u_star/dt).ravel()
        u = lu.solve(rhs).reshape(Ny, Nx)

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

# --------------------- Figures ---------------------

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 convergence_spatiale(ax,bx,ay,by,N0,levels,T,v1,v2,nu,lam,u_in,Tc,k,sc,cfl,
    outpath):
    """
    Calcule les erreurs L2 de u et ‖∇u‖ en fonction de h sur une hiérarchie emboî
    tée :
    N0 -> N1=2(N0-1)+1 -> N2=2(N1-1)+1 -> ...
    Le niveau le plus fin sert de référence.
```

```
    """
    # Construire la liste des N
    Ns = [N0]
    for _ in range(1, levels):
        Ns.append(2*(Ns[-1]-1)+1)

    # Résolutions pour tous les niveaux
    sols = []
    dxy = []
    for N in Ns:
        Nx = Ny = N
        # valeurs d'entrée (amont) constantes = u_in ; on peut remplacer par
    profils si besoin
        uL = np.full(Ny, u_in)
        uR = np.full(Ny, u_in)
        uB = np.full(Nx, u_in)
        uT = np.full(Nx, u_in)
        x, y, u, info = resout_imex(ax,bx,ay,by,Nx,Ny,T,v1,v2,nu,lam,uL,uR,uB,uT,
    Tc,k,sc,cfl)
        dx = (bx-ax)/(Nx-1); dy = (by-ay)/(Ny-1)
        sols.append(u)
        dxy.append(max(dx,dy))

    # Référence = plus fin
    uF = sols[-1]
    NF = Ns[-1]
    dxF = (bx-ax)/(NF-1); dyF = (by-ay)/(NF-1)
    from math import isclose

    e_u = []
    e_g = []
    hs  = []
    for N, uC, hC in zip(Ns[:-1], sols[:-1], dxy[:-1]):
        r = (NF-1)//(N-1)  # ratio entier (emboîtement garanti)
        assert (NF-1) % (N-1) == 0
        uF_on_C = uF[::r, ::r]

        # erreurs
        dxC = (bx-ax)/(N-1); dyC = (by-ay)/(N-1)
        e_u.append(erreur_L2(uC, uF_on_C, dxC, dyC))

        gC = norme_grad(uC, dxC, dyC)
        gF = norme_grad(uF, dxF, dyF)
        gF_on_C = gF[::r, ::r]
        e_g.append(erreur_L2(gC, gF_on_C, dxC, dyC))

        hs.append(hC)

    # fit pente
    logh = np.log(hs)
    logeu = np.log(e_u)
    logeg = np.log(e_g)
    pu = np.polyfit(logh, logeu, 1)[0]
    pg = np.polyfit(logh, logeg, 1)[0]

    # figure
    plt.figure(figsize=(6.5,5))
    plt.loglog(hs, e_u, marker='o', label=r"$\|e(u)\|_{L^2}$")
```

```
    plt.loglog(hs, e_g, marker='s', label=r"$\|e(\|\nabla u\|)\|_{L^2}$")
    # ligne de pente 1 pour repère
    c0 = e_u[0]/hs[0]  # normalise pour passer par (h0, e_u0)
    plt.loglog(hs, c0*np.array(hs), linestyle='--', label="pente 1 (réf)")
    plt.gca().invert_xaxis()
    plt.xlabel("pas h")
    plt.ylabel("erreur L2")
    plt.title(f"Convergence spatiale (p_u≈{pu:.2f}, p_grad≈{pg:.2f})")
    plt.legend()
    plt.tight_layout()
    plt.savefig(outpath, dpi=160)
    plt.show()
    return {"N": Ns, "h": hs, "e_u": e_u, "e_grad": e_g, "p_u": pu, "p_grad": pg}

def run():
    # ----- Paramètres par défaut -----
    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

    # Dossier de sortie (même que le script)
    outdir = Path(__file__).parent
    outdir.mkdir(parents=True, exist_ok=True)
    outfile_trip = outdir / "triple_panel.png"
    outfile_conv = outdir / "convergence.png"

    # ---------- Résolution (grille "coarse") et référence fine ----------
    # Bords amont : valeurs constantes = u_in (peuvent être remplacées par profils
    )
    uL = np.full(Ny, u_in)
    uR = np.full(Ny, u_in)
    uB = np.full(Nx, u_in)
    uT = np.full(Nx, u_in)

    x, y, uC, infoC = resout_imex(ax,bx,ay,by,Nx,Ny,T,v1,v2,nu,lam,uL,uR,uB,uT,Tc,
    k,sc,cfl)
    dxC, dyC = (bx-ax)/(Nx-1), (by-ay)/(Ny-1)

    NxF, NyF = 2*(Nx-1)+1, 2*(Ny-1)+1
    uL_F = np.full(NyF, u_in)
    uR_F = np.full(NyF, u_in)
    uB_F = np.full(NxF, u_in)
    uT_F = np.full(NxF, u_in)
    xF, yF, uF, infoF = resout_imex(ax,bx,ay,by,NxF,NyF,T,v1,v2,nu,lam,uL_F,uR_F,
    uB_F,uT_F,Tc,k,sc,cfl)
    uF_on_C = uF[::2, ::2]

    # ---------- Erreurs ----------
    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]

    # ---------- Triptyque ----------
    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‖₂={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‖₂={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 = collage_horizontal([img1, img2, img3], outfile_trip)

    # ---------- Convergence spatiale (3 niveaux par défaut) ----------
    conv = convergence_spatiale(ax,bx,ay,by,N0=41,levels=4,T=T,v1=v1,v2=v2,
                                nu=nu,lam=lam,u_in=u_in,Tc=Tc,k=k,sc=sc,cfl=cfl,
                                outpath=outfile_conv)

    # Affichage rapide (triptyque)
    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_trip)
    print("Courbe de convergence enregistrée dans :", outfile_conv)
    print("Pentes observées : p_u ≈ {:.2f}, p_grad ≈ {:.2f}".format(conv["p_u"],
    conv["p_grad"]))


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

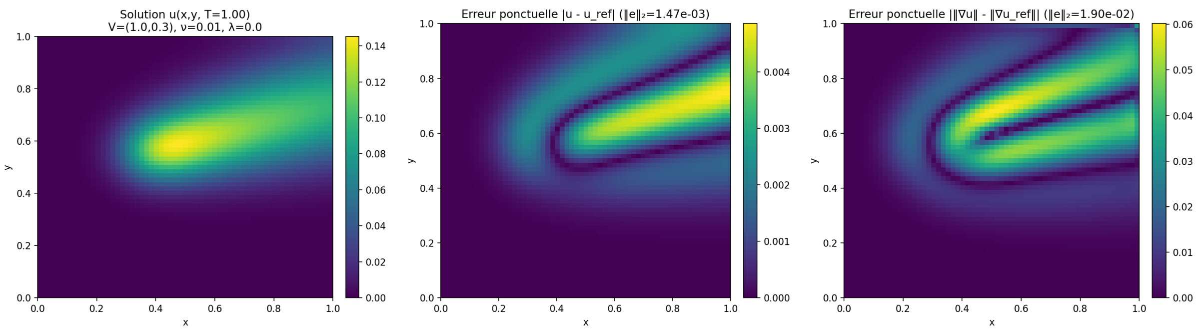

FIGURE 2.1 – De gauche à droite : solution $u(x, y, T)$; erreur ponctuelle $|u - u_{\text{ref}}|$ (la légende rappelle $\|e\|_2$ typiquement $\sim 1{,}5 \times 10^{-3}$ ici) ; erreur ponctuelle sur la norme du gradient $\big|\|\nabla u\| - \|\nabla u_{\text{ref}}\|\big|$ (avec $\|e\|_2 \approx 1{,}9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

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`

```python
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

# Choix du solveur: "sparse" (par défaut) ou "banded"
SOLVER = "sparse"    # "sparse" | "banded"

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

def ul(t):
    return 0.0

def g(t):
    return 0.0

# Paramètres de la gaussienne de départ
xc      = 0.2 * L
sigma   = 0.05 * L
A_amp   = 1.0
# ==========================================================


# ==========================================================
# u0(x) compatible avec u(0)=ul(0) et u_x(L)=g(0)
# ==========================================================
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
    return A * G0 + B * x + C


# ==========================================================
# Assemblage tri-diagonal (trois diagonales seulement)
# (I - dt*L) u^{n+1} = u^n + dt f^{n+1}
# L = -v D1_upwind + nu D2 - lam I
# => A = I + dt*v*D1_upwind - dt*nu*D2 + dt*lam*I
# CL: Dirichlet à gauche, Neumann à droite.
# ==========================================================
def build_tridiag(N, L, dt, v, nu, lam):
    dx = L / N
    n  = N + 1

    main  = np.zeros(n)      # diagonale principale A[i,i]
    lower = np.zeros(n - 1)  # sous-diagonale      A[i, i-1]
    upper = np.zeros(n - 1)  # sur-diagonale       A[i, i+1]

    use_backward = (v >= 0.0)
    alpha = dt * nu / dx**2
    beta  = dt * v  / dx

    # Lignes intérieures i=1..N-1
    for i in range(1, N):
        # Diffusion centrale
        main[i]   += 1.0 + dt*lam + 2.0 * alpha
```

```
        lower[i-1] += -alpha
        upper[i]   += -alpha

        # Advection upwind implicite
        if use_backward:
            main[i]  +=  beta
            lower[i-1] += -beta
        else:
            main[i]  += -beta
            upper[i] +=  beta

    # Bord gauche (Dirichlet): u(.,0) = ul
    main[0] = 1.0
    # upper[0] reste 0, lower[0] n'est pas utilisé

    # Bord droit (Neumann): (u_N - u_{N-1})/dx = g
    main[-1]  =  1.0 / dx
    lower[-1] = -1.0 / dx
    # upper[-1] inexistant pour la dernière ligne

    return lower, main, upper


def step_rhs(u_prev, t_next, x, dt):
    b = u_prev.copy()
    b[1:-1] += dt * np.array([f(t_next, xi) for xi in x[1:-1]])
    # CL
    b[0]  = ul(t_next)  # Dirichlet
    b[-1] = g(t_next)   # Neumann (cohérent avec la dernière ligne)
    return b


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

    # Assemblage tri-diagonal
    lower, main, upper = build_tridiag(N, L, dt, v, nu, lam)
    n = N + 1

    # Préparation du solveur choisi (factorisation réutilisée si possible)
    if solver == "sparse":
        # Matrice creuse CSC + LU sparse réutilisable
        from scipy.sparse import diags
        from scipy.sparse.linalg import splu

        A_csc = diags([lower, main, upper], offsets=[-1, 0, 1],
                      shape=(n, n), format="csc")
        lu = splu(A_csc)        # factorisation unique
        def solve_A(rhs):
            return lu.solve(rhs)

    elif solver == "banded":
        # Stockage bande (l=u=1) pour solve_banded
        # ab[0,1:] = upper ; ab[1,:] = main ; ab[2,:-1] = lower
        from scipy.linalg import solve_banded
        ab = np.zeros((3, n), dtype=float)
        ab[0, 1:] = upper
        ab[1, :]  = main
```

```
        ab[2, :-1]= lower
        def solve_A(rhs):
            # Remarque: solve_banded refactorise à chaque appel
            return solve_banded((1, 1), ab, rhs, overwrite_ab=False, overwrite_b=False, check_finite=False)
    else:
        raise ValueError("solver doit valoir 'sparse' ou 'banded'.")

    # État initial compatible
    u = build_u0(x)

    # Petits checks CL
    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.")

    # Intégration en temps
    times_to_store = np.linspace(0.0, T, 5)
    snapshots = [(0.0, u.copy())]
    t = 0.0
    next_store_idx = 1

    while t < T - 1e-12:
        t_next = min(t + dt, T)
        b = step_rhs(u, t_next, x, dt)
        u = solve_A(b)  # résolution via le solveur choisi
        t = t_next

        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, solver=SOLVER)

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(f"Convection-Diffusion-Réaction 1D -- solveur: {SOLVER}")
plt.legend()
plt.tight_layout()
plt.savefig("Convection_Diffusion_Reaction_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)

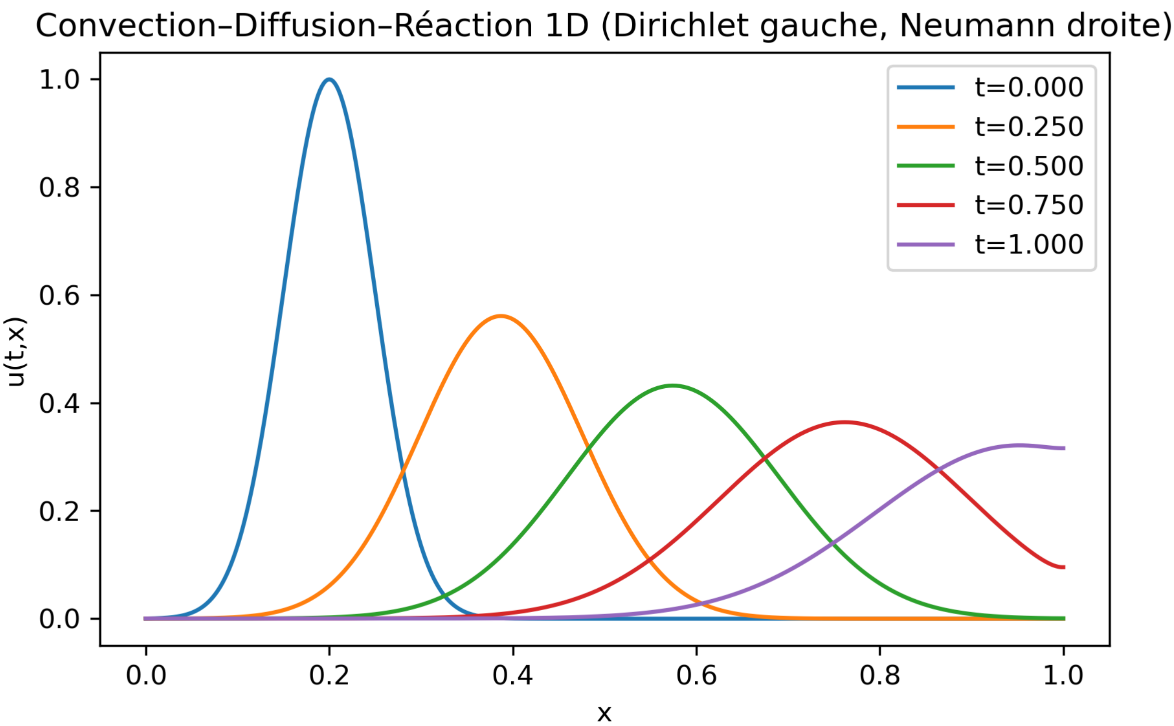

FIGURE 3.1 – Solution numérique $u(t, x)$ pour $v = 0.75$, $\nu = 10^{-2}$, $\lambda = 0$, $L = 1$, avec Dirichlet à gauche et Neumann (flux nul) à droite.

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.