



# Estimation a posteriori

*rapport scéance 1*

# Table des matières

<b>1</b>	<b>Méthode d'Euler explicite sur l'EDO <math>u'(t) = -\lambda u</math>, <math>u(0) = 1</math></b>	<b>2</b>
1.1	Problème et objectif . . . . .	2
1.2	Code Python utilisé . . . . .	2
1.3	Comparaison visuelle pour deux pas de temps . . . . .	9
1.4	Erreurs $L^2$ en fonction de $\Delta t$ . . . . .	10
1.5	Erreur ponctuelle vs norme de la dérivée exacte . . . . .	11
<b>2</b>	<b>Convection–diffusion avec source gaussienne</b>	<b>12</b>
2.1	Résumé . . . . .	12
2.2	L'équation et les données . . . . .	12
2.3	Discrétisation spatiale . . . . .	12
2.4	Schéma en temps (IMEX) . . . . .	13
2.5	Code Python utilisé . . . . .	13
2.6	Évaluation de l'erreur et post-traitement . . . . .	20
2.7	Le triptyque généré et son interprétation . . . . .	20
<b>3</b>	<b>Simulation numérique d'une équation de convection–diffusion–réaction 1D (conditions de Dirichlet–Neumann)</b>	<b>21</b>
3.1	Motivation . . . . .	21
3.2	Code Python . . . . .	21
3.3	Ce que fait le code et comment il le fait . . . . .	25
3.4	Résultat (figure) . . . . .	25
3.5	Commentaire sur la figure . . . . .	26

# Chapitre 1

## Méthode d'Euler explicite sur l'EDO

$$u'(t) = -\lambda u, \quad u(0) = 1$$

### 1.1 Problème et objectif

On considère l'équation différentielle ordinaire (EDO)

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

dont la solution exacte est  $u_{\text{ex}}(t) = e^{-\lambda t}$ . 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  $\Delta t$  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 ( $\log \Delta t$ ,  $\log$  erreur) 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  $\Delta t$  ( $\log$ - $\log$ ) 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

```

1
2 """
3 Euler_ODE_Errors.py -- version mise à jour
4 -----
5 EDO: u'(t) = -λ u(t), u(0)=u0.
6
7 Modifs (sept. 2025) :
8 1) Quadrature L2 : rectangles à gauche remplacés par trapèze / Simpson (si N pair)
9
10 2) Stabilité Euler : assertion robuste -- si λ=0 ne rien imposer ; sinon Δt ≤ (2
11    -)/λ.
12 3) Choix des pas : prend Δt = T/N avec N entier (échelonné log) pour éviter un "
13    dernier pas" irrégulier.
14
15 Figures générées (inchangées) :
16 1) Comparaison_visuelle.png
17    -> 2×2 : haut Δt=1 s, bas Δt=0.001 s ; (gauche) solutions, (droite) erreur
18 2) Erreur_vs_delta_temps.png
19    -> Erreurs L2 (u et u') en fonction de Δt (log-log)
20 3) Erreur_vs_derivé.png
21    -> Scatter de l'erreur ponctuelle |e(t_n)| en fonction de la norme
22    de la dérivée exacte |u'_{ex}(t_n)|, pour Δt=1 s et Δt=0.001 s.
23 """
24
25 import numpy as np
26 import matplotlib.pyplot as plt
27 from dataclasses import dataclass
28 from typing import Tuple, List, Optional
29
30 @dataclass
31 class Problem:
32     lam: float = 1.0    # λ
33     u0: float = 1.0     # condition initiale
34     T: float = 60.0     # horizon temporel (1 minute)
35
36 # -----
37 #   Intégrateur d'Euler explicite
38 # -----
39
40 def euler_explicite(pb: Problem, dt: Optional[float] = None, N: Optional[int] =
41     None, eps: float = 1e-12
42     ) -> Tuple[np.ndarray, np.ndarray, np.ndarray]:
43     """
44     Intègre u' = -λ u par Euler explicite avec pas Δt = T/N (N entier).
45     On peut fournir soit dt (approx.), soit directement N. Si dt est fourni,
46     on prend N = round(T/dt) puis Δt := T/N pour tomber EXACTEMENT à T.
47
48     Retourne:
49         t : instants (taille N+1, réguliers de 0 à T)
50         u : solution numérique aux instants t
51         dts: tableau de taille N rempli par Δt (tous les pas identiques)

```

```

50     """
51     T = pb.T
52     lam = pb.lam
53     u0 = pb.u0
54
55     if N is None:
56         if dt is None:
57             raise ValueError("Fournir soit dt, soit N.")
58         N = max(1, int(round(T / float(dt))))
59     else:
60         N = int(N)
61         if N <= 0:
62             raise ValueError("N doit être un entier positif.")
63
64     dt = T / N # pas EXACT et régulier
65     # Stabilité Euler robuste
66     if lam != 0.0 and lam > 0.0:
67         bound = (2.0 - eps) / lam
68         assert dt <= bound, (
69             f"Instable pour Euler explicite:  $\lambda \Delta t = \{lam*dt:.3g\}$  ; "
70             f"attendu  $\Delta t \leq (2-)/\lambda \approx \{bound:.6g\}$  ( $=\{eps:g\}$ ). "
71         )
72     # Si  $\lambda=0$  : ne rien imposer.
73
74     t = np.linspace(0.0, T, N + 1, dtype=float)
75     u = np.empty(N + 1, dtype=float)
76     u[0] = u0
77     for n in range(N):
78         u[n + 1] = u[n] + dt * (-lam * u[n])
79
80     dts = np.full(N, dt, dtype=float)
81     return t, u, dts
82
83
84 # -----
85 # Outils exacts / quadrature
86 # -----
87
88 def u_exact(t: np.ndarray, pb: Problem) -> np.ndarray:
89     return pb.u0 * np.exp(-pb.lam * t)
90
91
92 def _integrate_uniform(y: np.ndarray, dt: float) -> float:
93     """
94     Intègre une fonction tabulée y(t_n) sur [0,T] avec pas uniforme dt.
95     Utilise Simpson si le nombre d'intervalles N=len(y)-1 est pair, sinon trapèze.
96     Retourne l'intégrale numérique (pas la racine).
97     """
98     N = y.size - 1
99     if N <= 0:
100         return 0.0
101     if N % 2 == 0 and N >= 2:
102         # Simpson
103         s_odd = np.sum(y[1:N:2])
104         s_even = np.sum(y[2:N-1:2]) if N >= 3 else 0.0
105         return (dt / 3.0) * (y[0] + y[-1] + 4.0 * s_odd + 2.0 * s_even)
106     # Trapèze
107     return dt * (0.5 * y[0] + np.sum(y[1:-1]) + 0.5 * y[-1])

```

```

108
109
110 def l2_error_function(t: np.ndarray, u_num: np.ndarray, dts: np.ndarray, pb:
    Problem) -> float:
111     """
112     ||e||_{L2(0,T)} ≈ ( ∫_0^T |u_num(t)-u_ex(t)|^2 dt )^{1/2}
113     Quadrature: Simpson (si possible) sinon trapèze -- pas uniforme Δt = dts[0].
114     """
115     dt = float(dts[0])
116     e = u_num - u_exact(t, pb)
117     integ = _integrate_uniform(e**2, dt)
118     return float(np.sqrt(integ))
119
120
121 def l2_error_derivative(t: np.ndarray, u_num: np.ndarray, dts: np.ndarray, pb:
    Problem) -> float:
122     """
123     ||e'||_{L2(0,T)} ≈ ( ∫_0^T |u'_num(t) - u'_ex(t)|^2 dt )^{1/2}
124     - u'_num(t_n) : dérivée numérique aux noeuds via différences centrales (np.
    gradient)
125     - u'_ex(t_n) : -λ u_ex(t_n)
126     Quadrature: Simpson (si possible) sinon trapèze -- pas uniforme.
127     """
128     dt = float(dts[0])
129     uprime_num = np.gradient(u_num, dt) # central diff interne, 1er ordre aux
    bords
130     uprime_ex = -pb.lam * u_exact(t, pb)
131     eprime = uprime_num - uprime_ex
132     integ = _integrate_uniform(eprime**2, dt)
133     return float(np.sqrt(integ))
134
135
136 def convergence_slope(dts: np.ndarray, errs: np.ndarray) -> float:
137     x = np.log(dts)
138     y = np.log(errs)
139     A = np.vstack([x, np.ones_like(x)]).T
140     slope, _ = np.linalg.lstsq(A, y, rcond=None)[0]
141     return float(slope)
142
143
144 # -----
145 # Figures
146 # -----
147
148 def plot_part1_two_rows(pb: Problem,
149     dt_top: float = 1.0,
150     dt_bottom: float = 1e-3,
151     savepath: str = "Comparaison_visuelle.png") -> None:
152     """
153     Figure 2x2 : top = Δt=1 s ; bottom = Δt=0.001 s.
154     Colonnes: (gauche) solutions ; (droite) erreur ponctuelle.
155     Δt est projeté sur T/N (N entier) pour éviter un dernier pas irrégulier.
156     """
157     # TOP (Δt ≈ 1 s, mais forcé à T/N)
158     t1, u1, dts1 = euler_explicite(pb, dt_top)
159     ue1 = u_exact(t1, pb)
160     err1 = np.abs(u1 - ue1)
161

```

```

162 # BOTTOM ( $\Delta t \approx 0.001$  s, mais forcé à  $T/N$ )
163 t2, u2, dts2 = euler_explicite(pb, dt_bottom)
164 ue2 = u_exact(t2, pb)
165 err2 = np.abs(u2 - ue2)
166
167 fig, axes = plt.subplots(2, 2, figsize=(12, 8))
168 (ax00, ax01), (ax10, ax11) = axes
169
170 # Top-left: solutions  $\Delta t$ 
171 ax00.plot(t1, ue1, label="Solution exacte  $u_{\text{ex}}(t)$ ")
172 ax00.plot(t1, u1, marker="o", linestyle="--", label=fr"Euler  $\Delta t = \{dts1[0]:g\}$  s")
173 ax00.set_xlabel("Temps t (s)")
174 ax00.set_ylabel("Amplitude u(t)")
175 ax00.set_title("Solutions --  $\Delta t \approx 1$  s")
176 ax00.grid(True, alpha=0.3)
177 ax00.legend()
178
179 # Top-right: erreur  $\Delta t$ 
180 ax01.plot(t1, err1, marker="o", linestyle="--", label=r" $|e(t_n)|$ ")
181 ax01.set_xlabel("Temps t (s)")
182 ax01.set_ylabel("Erreur ponctuelle")
183 ax01.set_title("Erreur --  $\Delta t \approx 1$  s")
184 ax01.grid(True, alpha=0.3)
185 ax01.legend()
186
187 # Bottom-left: solutions  $\Delta t$ 
188 ax10.plot(t2, ue2, label="Solution exacte  $u_{\text{ex}}(t)$ ")
189 ax10.plot(t2, u2, linestyle="--", linewidth=1.0, label=fr"Euler  $\Delta t = \{dts2[0]:g\}$  s")
190 ax10.set_xlabel("Temps t (s)")
191 ax10.set_ylabel("Amplitude u(t)")
192 ax10.set_title("Solutions --  $\Delta t \approx 0.001$  s")
193 ax10.grid(True, alpha=0.3)
194 ax10.legend()
195
196 # Bottom-right: erreur  $\Delta t$ 
197 ax11.plot(t2, err2, linestyle="--", linewidth=1.0, label=r" $|e(t_n)|$ ")
198 ax11.set_xlabel("Temps t (s)")
199 ax11.set_ylabel("Erreur ponctuelle")
200 ax11.set_title("Erreur --  $\Delta t \approx 0.001$  s")
201 ax11.grid(True, alpha=0.3)
202 ax11.legend()
203
204 fig.suptitle("u'(t) = - $\lambda$  u, u(0)=1 -- Comparaison visuelle  $\Delta t$ ", y=0.98)
205 fig.tight_layout(rect=[0, 0, 1, 0.96])
206 fig.savefig(savepath, dpi=150)
207
208
209 def _logspace_integers(n_min: int, n_max: int, n_steps: int) -> np.ndarray:
210     """Valeurs entières échelonnées logarithmiquement dans [n_min, n_max]."""
211     vals = np.geomspace(max(1, n_min), max(1, n_max), max(2, n_steps))
212     ints = np.unique(np.clip(np.round(vals).astype(int), n_min, n_max))
213     return ints
214
215
216 def plot_part2(pb: Problem, n_steps: int = 20, dt_min: float = 1e-3, dt_max: float
    = 1.0,

```

```

217         savepath: str = "Erreur_vs_delta_temps.png") -> None:
218     """
219      $\Delta t$  définis via  $\Delta t = T/N$  avec N entier échelonné log entre
220     N_min = ceil(T/dt_max) et N_max = floor(T/dt_min).
221     Trace ||e||_L2 et ||e'||_L2 en fonction de  $\Delta t$  (log-log).
222     """
223     T = pb.T
224     N_min = int(np.ceil(T / dt_max))
225     N_max = int(np.floor(T / dt_min))
226     if N_max < max(2, N_min):
227         raise ValueError("Plage (dt_min, dt_max) trop étroite pour construire des
228         N entiers.")
229     N_list = _logspace_integers(N_min, N_max, n_steps)
230
231     dts_arr = []
232     errL2_u: List[float] = []
233     errL2_du: List[float] = []
234
235     for N in N_list:
236         t, u_num, dts = euler_explicite(pb, N=N)
237         dts_arr.append(float(dts[0]))
238         errL2_u.append(l2_error_function(t, u_num, dts, pb))
239         errL2_du.append(l2_error_derivative(t, u_num, dts, pb))
240
241     dts_arr = np.array(dts_arr, dtype=float)
242     errL2_u = np.array(errL2_u, dtype=float)
243     errL2_du = np.array(errL2_du, dtype=float)
244
245     slope_u = convergence_slope(dts_arr, errL2_u)
246     slope_du = convergence_slope(dts_arr, errL2_du)
247
248     fig, ax = plt.subplots(1, 1, figsize=(6.5, 4.5))
249     ax.loglog(dts_arr, errL2_u, marker="o", linestyle="-", label=r"$\|u_h-u_{ex}\|_{L^2(0,T)}$")
250     ax.loglog(dts_arr, errL2_du, marker="s", linestyle="--", label=r"$\|u'_h-u'_{ex}\|_{L^2(0,T)}$")
251     ax.set_xlabel("Pas de temps  $\Delta t$  (s)")
252     ax.set_ylabel("Erreur L2 sur [0, T]")
253     ax.set_title(f"Erreurs L2 vs  $\Delta t$  -- pentes  $\approx$  {slope_u:.2f} (u), {slope_du:.2f} (u')")
254     ax.grid(True, which="both", alpha=0.3)
255     ax.legend()
256     fig.tight_layout()
257     fig.savefig(savepath, dpi=150)
258
259     # Impression console (utile pour le rapport)
260     print(f"Pente de convergence (log-log) pour  $\|u_h - u_{ex}\|_{L2}$  : {slope_u:.4f}")
261     print(f"Pente de convergence (log-log) pour  $\|u'_h - u'_{ex}\|_{L2}$  : {slope_du:.4f}")
262
263 def plot_error_vs_exact_derivative(pb: Problem,
264                                   dts_to_show: List[float] = [1.0, 1e-3],
265                                   savepath: str = "Erreur_vs_derivé.png") -> None:
266     """
267     Scatter: erreur ponctuelle |e(t_n)| en fonction de |u'_{ex}(t_n)|,

```



```

268     pour plusieurs pas de temps (par défaut:  $\Delta t=1$  s et  $\Delta t=0.001$  s).
269     Deux sous-graphes: (gauche) axes linéaires, (droite) log-log.
270      $\Delta t$  est projeté sur  $T/N$  ( $N$  entier) pour éviter un dernier pas irrégulier.
271     """
272     fig, (ax_lin, ax_log) = plt.subplots(1, 2, figsize=(12, 4.5))
273
274     for dt in dts_to_show:
275         t, u_num, dts = euler_explicite(pb, dt)
276         ue = u_exact(t, pb)
277         err = np.abs(u_num - ue)
278         deriv_norm = np.abs(-pb.lam * ue)  # = pb.lam * |ue|
279
280         ax_lin.scatter(deriv_norm, err, s=10, alpha=0.6, label=fr"$\Delta t={dts[0]:g}$ s")
281         ax_log.loglog(deriv_norm + 1e-16, err + 1e-16, marker="o", linestyle="",
282 markersize=3, alpha=0.6, label=fr"$\Delta t={dts[0]:g}$ s")
283
284         ax_lin.set_xlabel(r"$|u'_{ex}(t_n)|$")
285         ax_lin.set_ylabel(r"$|e(t_n)|$")
286         ax_lin.set_title("Erreur vs norme de la dérivée exacte (linéaire)")
287         ax_lin.grid(True, alpha=0.3)
288         ax_lin.legend()
289
290         ax_log.set_xlabel(r"$|u'_{ex}(t_n)|$")
291         ax_log.set_ylabel(r"$|e(t_n)|$")
292         ax_log.set_title("Erreur vs norme de la dérivée exacte (log-log)")
293         ax_log.grid(True, which="both", alpha=0.3)
294         ax_log.legend()
295
296     fig.tight_layout()
297     fig.savefig(savepath, dpi=150)
298
299 def main():
300     pb = Problem(lam=1.0, u0=1.0, T=60.0)
301
302     # Partie 1 : deux rangées:  $\Delta t \approx 1$  s (haut) et  $\Delta t \approx 0.001$  s (bas),
303     # mais forcés à  $\Delta t = T/N$  exactement.
304     plot_part1_two_rows(pb, dt_top=1.0, dt_bottom=1e-3,
305 savepath="Comparaison_visuelle.png")
306
307     # Partie 2 : erreur L2 en fonction de  $\Delta t$  ( $N$  entiers échelonnés log entre
308     dt_max et dt_min)
309     plot_part2(pb, n_steps=20, dt_min=1e-3, dt_max=1.0,
310 savepath="Erreur_vs_delta_temps.png")
311
312     # Partie 3 : Erreur ponctuelle vs norme de la dérivée exacte
313     plot_error_vs_exact_derivative(pb, dts_to_show=[1.0, 1e-3],
314 savepath="Erreur_vs_derive.png")
315
316 if __name__ == "__main__":
317     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  $u_{\text{ex}}$  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.

$$u'(t) = -\lambda u, u(0)=1; \lambda=1 \text{ — Comparaison visuelle } \Delta t$$

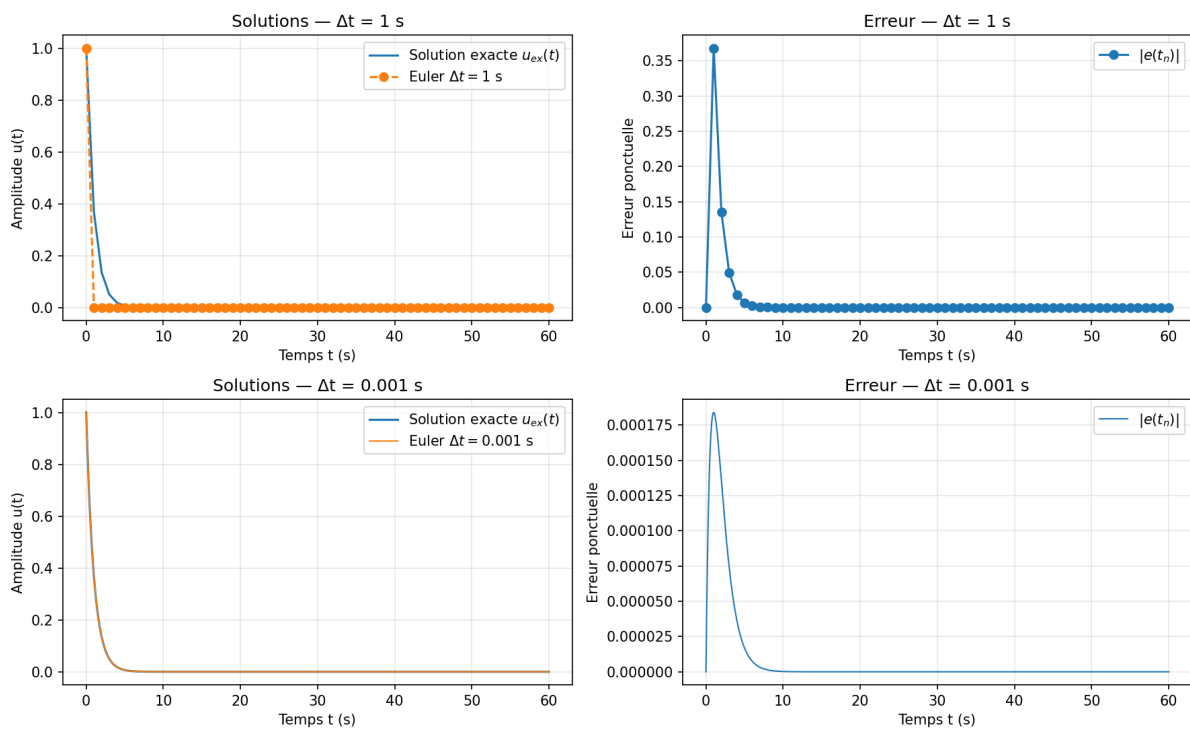


FIGURE 1.1 – Comparaison visuelle des solutions et de l'erreur ponctuelle pour  $\Delta t = 1$  s (haut) et  $\Delta t = 0,001$  s (bas).

## 1.4 Erreurs $L^2$ en fonction de $\Delta t$

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  $\approx 1.04$  pour  $u$  et  $\approx 1.06$  pour  $u'$ ), ce qui est conforme à l'ordre 1 attendu pour la méthode d'Euler explicite. Autrement dit, en divisant  $\Delta t$  par 10, l'erreur globale décroît approximativement d'un facteur 10.

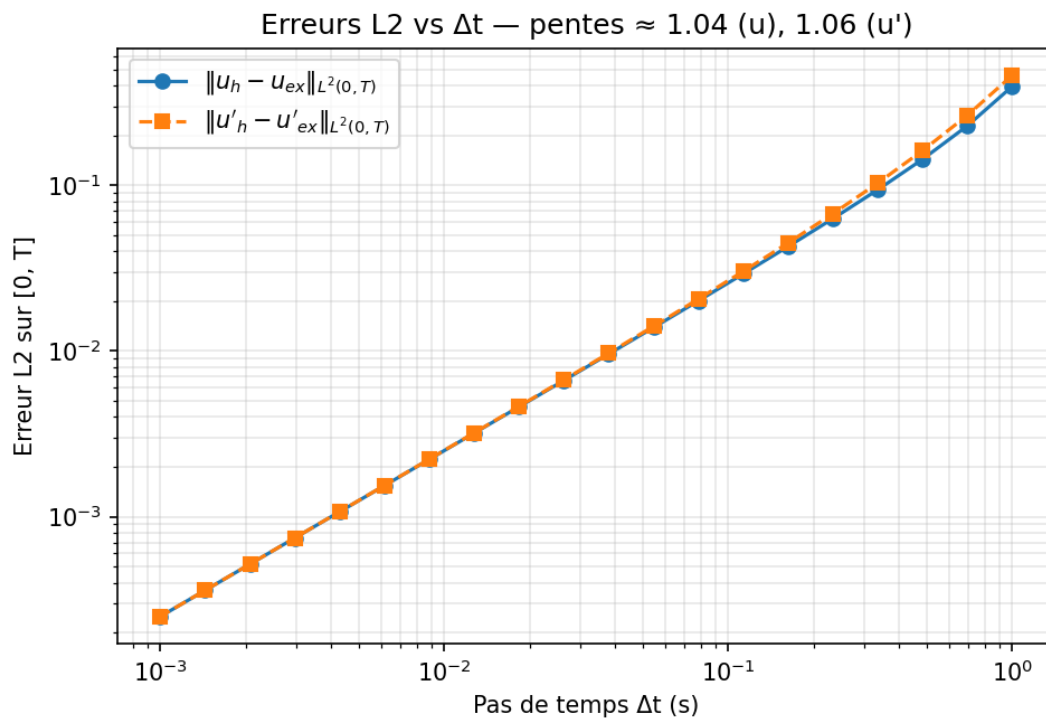


FIGURE 1.2 – Erreurs  $\|u_h - u_{ex}\|_{L^2(0,T)}$  et  $\|u'_h - u'_{ex}\|_{L^2(0,T)}$  versus  $\Delta t$ ; les pentes log-log mesurées sont proches de 1 (ordre d'Euler).

## 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.

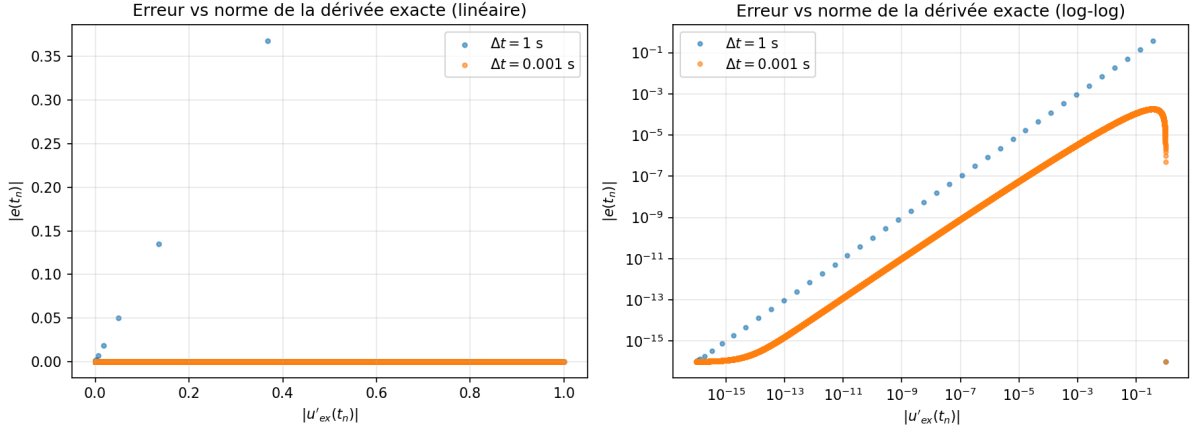


FIGURE 1.3 – Erreur ponctuelle  $|e(t_n)|$  en fonction de  $|u'_{\text{ex}}(t_n)|$  (échelles linéaire et log-log) pour deux pas de temps. La corrélation forte entre erreur et norme de la dérivée est manifeste; l’“amélioration” en fin d’intervalle provient du fait que  $|u'_{\text{ex}}|$  devient très 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  $\Delta t$ , la zone où la dérivée varie rapidement restera la plus difficile pour Euler explicite; diminuer  $\Delta t$  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

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

avec  $\mathbf{V} = (v_1, v_2)$ , la viscosité  $\nu$ , le terme de réaction  $\lambda$  (nul ici) et une source gaussienne  $f(x, y) = T_c \exp(-k\|(x, y) - \mathbf{s}_c\|^2)$  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  $61 \times 61$ ; 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  $\lambda u$ . 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  $u_{\text{in}}$ .

## 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  $\Delta t$ , on forme d'abord

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

puis on résout linéairement

$$\left(\frac{1}{\Delta t} + \lambda - \nu \Delta\right) u^{n+1} = \frac{1}{\Delta t} u^*,$$

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.

```
1
2  """
3  Convection-Diffusion(-Réaction) 2D
4  -----
5  - Diffusion : conditions de Neumann homogènes (u/n = 0) SUR TOUS LES BORDS,
6    traitées dans l'opérateur implicite via des stencils à point fantôme (facteur 2
7    sur le voisin intérieur).
8  - Advection : schéma d'amont (upwind) explicite. Les valeurs aux bords amont sont
9    passées **directement** via uL (gauche, taille Ny), uR (droite, Ny), uB (bas, Nx
10   ), uT (haut, Nx).
11   Pas de `repeat` ni de masques Dirichlet : l'amont impose les valeurs d'entrée.
12 - Réaction : terme  $-\lambda u$  implicite (lumped dans la diagonale).
13 - Sortie : un triptyque (solution, erreur u, erreur  $\|\nabla u\|$ ) et une courbe de
14   convergence spatiale
15    $\|e(u)\|$  et  $\|e(\nabla u)\|$  en fonction de h sur 2-3 raffinements (attendu  $\approx$  ordre 1 avec
16   amont + temps ordre 1).
17
18 Équation :  $u_t + V \cdot \nabla u - \nu \Delta u = -\lambda u + f(x,y)$ ,
19              $f(x,y) = Tc \cdot \exp(-k \cdot \|(x,y) - s_c\|^2)$ .
20 """
21
22 from io import BytesIO
23 from pathlib import Path
24 import numpy as np
25 import scipy.sparse as sp
26 import scipy.sparse.linalg as spla
27 import matplotlib.pyplot as plt
28
29 # ----- Utilitaires PDE -----
30
31 def assemble_operateur(Nx, Ny, dx, dy, dt, nu, lam):
32     """
33     Assemble  $M \approx (I/dt + \lambda) - \nu \Delta$  avec Neumann homogène sur le bord.
34     Discrétisation en différences finies sur grille cartésienne (indexing 'xy').
35     """
```

```

31     Aux bords : stencil à point fantôme coefficient *2 sur le voisin intérieur.
32     """
33     alpha = 1.0/dt + lam
34     N = Nx*Ny
35     rows, cols, vals = [], [], []
36
37     def idg(i,j): return i + Nx*j
38
39     cx = nu/(dx*dx) if Nx > 1 else 0.0
40     cy = nu/(dy*dy) if Ny > 1 else 0.0
41
42     for j in range(Ny):
43         for i in range(Nx):
44             p = idg(i,j)
45             diag = alpha
46
47             # --- x-direction with homogeneous Neumann ---
48             if Nx > 1:
49                 if i == 0:
50                     #  $u_{-1} = u_1$       $\Delta x u_0 \approx 2(u_1 - u_0)/dx^2$ 
51                     diag += 2*cx
52                     rows.append(p); cols.append(idg(i+1, j)); vals.append(-2*cx)
53                 elif i == Nx-1:
54                     diag += 2*cx
55                     rows.append(p); cols.append(idg(i-1, j)); vals.append(-2*cx)
56                 else:
57                     diag += 2*cx
58                     rows.append(p); cols.append(idg(i-1, j)); vals.append(-cx)
59                     rows.append(p); cols.append(idg(i+1, j)); vals.append(-cx)
60
61             # --- y-direction with homogeneous Neumann ---
62             if Ny > 1:
63                 if j == 0:
64                     diag += 2*cy
65                     rows.append(p); cols.append(idg(i, j+1)); vals.append(-2*cy)
66                 elif j == Ny-1:
67                     diag += 2*cy
68                     rows.append(p); cols.append(idg(i, j-1)); vals.append(-2*cy)
69                 else:
70                     diag += 2*cy
71                     rows.append(p); cols.append(idg(i, j-1)); vals.append(-cy)
72                     rows.append(p); cols.append(idg(i, j+1)); vals.append(-cy)
73
74             rows.append(p); cols.append(p); vals.append(diag)
75
76     return sp.csr_matrix((vals, (rows, cols)), shape=(N, N))
77
78 def advection_amont(u, v1, v2, dx, dy, uL, uR, uB, uT):
79     """
80     Flux d'advection explicite (amont).
81     Entrée :
82         - u : (Ny, Nx)
83         - v1, v2 : vitesses constantes V=(v1,v2)
84         - uL (Ny,), uR (Ny,), uB (Nx,), uT (Nx,)
85           valeurs aux bords gauche/droite/bas/haut UTILISÉES UNIQUEMENT
86           quand la vitesse entre dans le domaine par ce bord.
87     Sortie : - (v1 x u + v2 y u) évalué par différences amont.
88     """

```

```

89     Ny, Nx = u.shape
90     dudx = np.zeros_like(u)
91     dudy = np.zeros_like(u)
92
93     # x-direction
94     if v1 >= 0:
95         # amont = arrière
96         if Nx > 1:
97             dudx[:, 1:] = (u[:, 1:] - u[:, :-1]) / dx
98         # bord gauche utilise uL
99         dudx[:, 0] = (u[:, 0] - uL) / dx
100     else:
101         # amont = avant
102         if Nx > 1:
103             dudx[:, :-1] = (u[:, 1:] - u[:, :-1]) / dx
104         # bord droit utilise uR
105         dudx[:, -1] = (uR - u[:, -1]) / dx
106
107     # y-direction
108     if v2 >= 0:
109         if Ny > 1:
110             dudy[1:, :] = (u[1:, :] - u[:-1, :]) / dy
111         dudy[0, :] = (u[0, :] - uB) / dy
112     else:
113         if Ny > 1:
114             dudy[:-1, :] = (u[1:, :] - u[:-1, :]) / dy
115         dudy[-1, :] = (uT - u[-1, :]) / dy
116
117     return -(v1 * dudx + v2 * dudy)
118
119 def source_gauss(x, y, Tc, k, sc):
120     X, Y = np.meshgrid(x, y, indexing='xy')
121     return Tc * np.exp(-k * ((X - sc[0])**2 + (Y - sc[1])**2))
122
123 def norme_grad(u, dx, dy):
124     Ny, Nx = u.shape
125     dudx = np.zeros_like(u); dudy = np.zeros_like(u)
126     if Nx > 1:
127         dudx[:, 1:-1] = (u[:, 2:] - u[:, :-2]) / (2*dx)
128         dudx[:, 0] = (u[:, 1] - u[:, 0]) / dx
129         dudx[:, -1] = (u[:, -1] - u[:, -2]) / dx
130     if Ny > 1:
131         dudy[1:-1, :] = (u[2:, :] - u[:-2, :]) / (2*dy)
132         dudy[0, :] = (u[1, :] - u[0, :]) / dy
133         dudy[-1, :] = (u[-1, :] - u[-2, :]) / dy
134     return np.sqrt(dudx**2 + dudy**2)
135
136 def erreur_L2(champ, ref, dx, dy):
137     diff = champ - ref
138     return np.sqrt(np.sum(diff**2) * dx * dy)
139
140 # ----- Solveur IMEX -----
141
142 def resout_imex(ax,bx,ay,by,Nx,Ny,T,v1,v2,nu,lam,uL,uR,uB,uT,Tc,k,sc,cf1):
143     x = np.linspace(ax, bx, Nx)
144     y = np.linspace(ay, by, Ny)
145     dx = (bx-ax)/(Nx-1) if Nx>1 else 1.0
146     dy = (by-ay)/(Ny-1) if Ny>1 else 1.0

```



```

147
148     # CFL simple basé sur l'advection
149     limites = []
150     if abs(v1)>0 and Nx>1: limites.append(dx/abs(v1))
151     if abs(v2)>0 and Ny>1: limites.append(dy/abs(v2))
152     dt = cfl*min(limites) if limites else 0.02*min(dx,dy) # fallback
153     nsteps = int(np.ceil(T/dt)); dt = T/nsteps
154
155     M = assemble_opérateur(Nx, Ny, dx, dy, dt, nu, lam)
156     lu = spla.splu(M.tocsc())
157
158     f = source_gauss(x, y, Tc, k, sc)
159     u = np.zeros((Ny, Nx))
160
161     for _ in range(nsteps):
162         adv = advection_amont(u, v1, v2, dx, dy, uL, uR, uB, uT)
163         u_star = u + dt*(adv + f)
164         rhs = (u_star/dt).ravel()
165         u = lu.solve(rhs).reshape(Ny, Nx)
166
167     info = {"dt": dt, "nsteps": nsteps}
168     return x, y, u, info
169
170     # ----- Figures -----
171
172     def figure_as_image(plotter, figsize=(6,5), dpi=160):
173         """Crée une figure Matplotlib via la fonction plotter(ax) et renvoie son image
174         PIL en mémoire."""
175         import PIL.Image as Image
176         fig, ax = plt.subplots(figsize=figsize)
177         plotter(ax)
178         buf = BytesIO()
179         fig.tight_layout()
180         fig.savefig(buf, format="png", dpi=dpi)
181         plt.close(fig)
182         buf.seek(0)
183         return Image.open(buf).convert("RGB")
184
185     def collage_horizontal(images, outpath):
186         """Colle des images PIL horizontalement et enregistre le résultat."""
187         from PIL import Image
188         h = max(im.size[1] for im in images)
189         imgs = [im.resize((int(im.size[0]*h/im.size[1]), h)) for im in images]
190         W = sum(im.size[0] for im in imgs)
191         canvas = Image.new("RGB", (W, h), (255,255,255))
192         xoff = 0
193         for im in imgs:
194             canvas.paste(im, (xoff, 0)); xoff += im.size[0]
195         canvas.save(outpath)
196         return canvas
197
198     def convergence_spatiale(ax,bx,ay,by,N0,levels,T,v1,v2,nu,lam,u_in,Tc,k,sc,cfl,
199         outpath):
200         """
201         Calcule les erreurs L2 de u et  $\|\nabla u\|$  en fonction de h sur une hiérarchie emboî
202         tée :
203         N0 -> N1=2(N0-1)+1 -> N2=2(N1-1)+1 -> ...
204         Le niveau le plus fin sert de référence.

```

```

202     """
203     # Construire la liste des N
204     Ns = [N0]
205     for _ in range(1, levels):
206         Ns.append(2*(Ns[-1]-1)+1)
207
208     # Résolutions pour tous les niveaux
209     sols = []
210     dxy = []
211     for N in Ns:
212         Nx = Ny = N
213         # valeurs d'entrée (amont) constantes = u_in ; on peut remplacer par
214         # profils si besoin
215         uL = np.full(Ny, u_in)
216         uR = np.full(Ny, u_in)
217         uB = np.full(Nx, u_in)
218         uT = np.full(Nx, u_in)
219         x, y, u, info = resout_imex(ax,bx,ay,by,Nx,Ny,T,v1,v2,nu,lam,uL,uR,uB,uT,
220         Tc,k,sc,cfl)
221         dx = (bx-ax)/(Nx-1); dy = (by-ay)/(Ny-1)
222         sols.append(u)
223         dxy.append(max(dx,dy))
224
225     # Référence = plus fin
226     uF = sols[-1]
227     NF = Ns[-1]
228     dxF = (bx-ax)/(NF-1); dyF = (by-ay)/(NF-1)
229     from math import isclose
230
231     e_u = []
232     e_g = []
233     hs = []
234     for N, uC, hC in zip(Ns[:-1], sols[:-1], dxy[:-1]):
235         r = (NF-1)/(N-1) # ratio entier (emboîtement garanti)
236         assert (NF-1) % (N-1) == 0
237         uF_on_C = uF[:, :r, ::r]
238
239         # erreurs
240         dxC = (bx-ax)/(N-1); dyC = (by-ay)/(N-1)
241         e_u.append(erreur_L2(uC, uF_on_C, dxC, dyC))
242
243         gC = norme_grad(uC, dxC, dyC)
244         gF = norme_grad(uF, dxF, dyF)
245         gF_on_C = gF[:, :r, ::r]
246         e_g.append(erreur_L2(gC, gF_on_C, dxC, dyC))
247
248         hs.append(hC)
249
250     # fit pente
251     logh = np.log(hs)
252     logeu = np.log(e_u)
253     logeg = np.log(e_g)
254     pu = np.polyfit(logh, logeu, 1)[0]
255     pg = np.polyfit(logh, logeg, 1)[0]
256
257     # figure
258     plt.figure(figsize=(6.5,5))
259     plt.loglog(hs, e_u, marker='o', label=r"$\|e(u)\|_{L^2}$")

```

```

258 plt.loglog(hs, e_g, marker='s', label=r"$|e(\|\nabla u\|)\|_{L^2}$")
259 # ligne de pente 1 pour repère
260 c0 = e_u[0]/hs[0] # normalise pour passer par (h0, e_u0)
261 plt.loglog(hs, c0*np.array(hs), linestyle='--', label="pente 1 (réf)")
262 plt.gca().invert_xaxis()
263 plt.xlabel("pas h")
264 plt.ylabel("erreur L2")
265 plt.title(f"Convergence spatiale (p_u≈{pu:.2f}, p_grad≈{pg:.2f})")
266 plt.legend()
267 plt.tight_layout()
268 plt.savefig(outpath, dpi=160)
269 plt.show()
270 return {"N": Ns, "h": hs, "e_u": e_u, "e_grad": e_g, "p_u": pu, "p_grad": pg}
271
272 def run():
273     # ----- Paramètres par défaut -----
274     ax, bx, ay, by = 0.0, 1.0, 0.0, 1.0
275     Nx, Ny = 61, 61
276     T = 1.0
277     v1, v2 = 1.0, 0.3
278     nu, lam = 0.01, 0.0
279     u_in = 0.0
280     Tc, k = 1.0, 80.0
281     sc = (0.35, 0.55)
282     cfl = 0.45
283
284     # Dossier de sortie (même que le script)
285     outdir = Path(__file__).parent
286     outdir.mkdir(parents=True, exist_ok=True)
287     outfile_trip = outdir / "triple_panel.png"
288     outfile_conv = outdir / "convergence.png"
289
290     # ----- Résolution (grille "coarse") et référence fine -----
291     # Bords amont : valeurs constantes = u_in (peuvent être remplacées par profils
292     # )
293     uL = np.full(Ny, u_in)
294     uR = np.full(Ny, u_in)
295     uB = np.full(Nx, u_in)
296     uT = np.full(Nx, u_in)
297
298     x, y, uC, infoC = resout_imex(ax,bx,ay,by,Nx,Ny,T,v1,v2,nu,lam,uL,uR,uB,uT,Tc,
299     k,sc,cfl)
300     dxC, dyC = (bx-ax)/(Nx-1), (by-ay)/(Ny-1)
301
302     Nx_F, Ny_F = 2*(Nx-1)+1, 2*(Ny-1)+1
303     uL_F = np.full(Ny_F, u_in)
304     uR_F = np.full(Ny_F, u_in)
305     uB_F = np.full(Nx_F, u_in)
306     uT_F = np.full(Nx_F, u_in)
307     xF, yF, uF, infoF = resout_imex(ax,bx,ay,by,Nx_F,Ny_F,T,v1,v2,nu,lam,uL_F,uR_F,
308     uB_F,uT_F,Tc,k,sc,cfl)
309     uF_on_C = uF[:,::2, ::2]
310
311     # ----- Erreurs -----
312     e_u_L2 = erreur_L2(uC, uF_on_C, dxC, dyC)
313     gC = norme_grad(uC, dxC, dyC)
314     dxF, dyF = (bx-ax)/(Nx_F-1), (by-ay)/(Ny_F-1)
315     gF = norme_grad(uF, dxF, dyF)

```

```

313 gF_on_C = gF[:, :2, :2]
314 e_g_L2 = erreur_L2(gC, gF_on_C, dxC, dyC)
315
316 e_u_pw = np.abs(uC - uF_on_C)
317 e_g_pw = np.abs(gC - gF_on_C)
318 extent = [ax, bx, ay, by]
319
320 # ----- Triptyque -----
321 def plot_solution(axp):
322     im = axp.imshow(uC, extent=extent, origin='lower', aspect='auto')
323     axp.set_title(f"Solution u(x,y, T={T:.2f})\nV=({v1},{v2}), ν={nu}, λ={lam}"
324 ")
325     axp.set_xlabel("x"); axp.set_ylabel("y")
326     plt.colorbar(im, ax=axp, fraction=0.046, pad=0.04)
327
328 def plot_err_u(axp):
329     im = axp.imshow(e_u_pw, extent=extent, origin='lower', aspect='auto')
330     axp.set_title(f"Erreur ponctuelle |u - u_ref| (||e||={e_u_L2:.2e})")
331     axp.set_xlabel("x"); axp.set_ylabel("y")
332     plt.colorbar(im, ax=axp, fraction=0.046, pad=0.04)
333
334 def plot_err_grad(axp):
335     im = axp.imshow(e_g_pw, extent=extent, origin='lower', aspect='auto')
336     axp.set_title(f"Erreur ponctuelle ||∇u|| - ||∇u_ref|| (||e||={e_g_L2:.2e})")
337     axp.set_xlabel("x"); axp.set_ylabel("y")
338     plt.colorbar(im, ax=axp, fraction=0.046, pad=0.04)
339
340 img1 = figure_as_image(plot_solution)
341 img2 = figure_as_image(plot_err_u)
342 img3 = figure_as_image(plot_err_grad)
343
344 collage = collage_horizontal([img1, img2, img3], outfile_trip)
345
346 # ----- Convergence spatiale (3 niveaux par défaut) -----
347 conv = convergence_spatiale(ax,bx,ay,by,N0=41,levels=4,T=T,v1=v1,v2=v2,
348                             nu=nu,lam=lam,u_in=u_in,Tc=Tc,k=k,sc=sc,cfl=cfl,
349                             outpath=outfile_conv)
350
351 # Affichage rapide (triptyque)
352 plt.figure(figsize=(14,5))
353 plt.imshow(collage)
354 plt.axis("off")
355 plt.title("Triptyque : Solution -- Erreur u -- Erreur ||∇u||")
356 plt.show()
357
358 print("Triptyque enregistré dans :", outfile_trip)
359 print("Courbe de convergence enregistrée dans :", outfile_conv)
360 print("Pentes observées : p_u ≈ {:.2f}, p_grad ≈ {:.2f}".format(conv["p_u"],
361 conv["p_grad"]))
362
363 if __name__ == "__main__":
364     run()

```

## 2.6 Évaluation de l'erreur et post-traitement

Après avoir calculé  $u$  sur la grille grossière, une solution de *référence*  $u_{\text{ref}}$  est obtenue sur la grille fine. Elle est sous-échantillonnée sur la grille grossière pour comparer  $u$  et  $u_{\text{ref}}$ . 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  $|\|\nabla u\| - \|\nabla u_{\text{ref}}\||$ .

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

## 2.7 Le triptyque généré et son interprétation

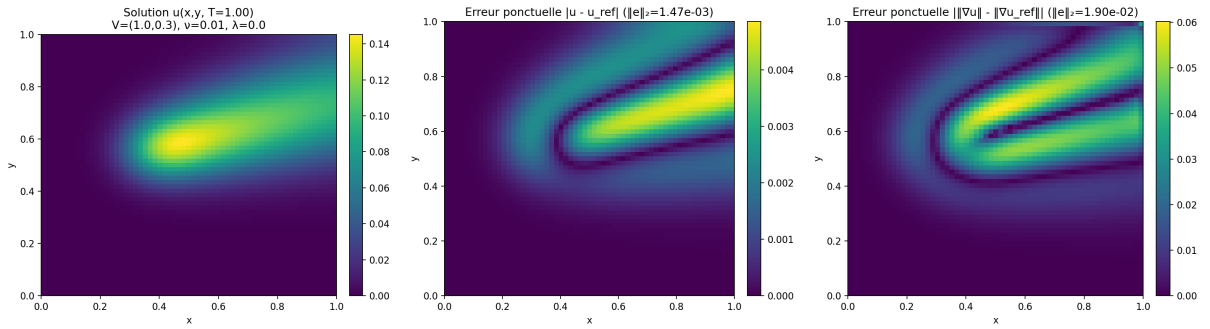


FIGURE 2.1 – De gauche à droite : solution  $u(x, y, T)$  ; erreur ponctuelle  $|u - u_{\text{ref}}|$  (la légende rappelle  $\|e\|_2$  typiquement  $\sim 1,5 \times 10^{-3}$  ici) ; erreur ponctuelle sur la norme du gradient  $|\|\nabla u\| - \|\nabla u_{\text{ref}}\||$  (avec  $\|e\|_2 \approx 1,9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

**Commentaire.** La solution présente une *bulle* créée par la source gaussienne, advectionnée 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

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

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

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3
4  # =====
5  # Paramètres du modèle (à ajuster)
6  # =====
7  L      = 1.0      # longueur du domaine [0, L]
8  v      = 0.750    # vitesse de convection (constante)
9  nu     = 1e-2     # diffusivité
10 lam    = 0.0      # lambda de réaction (>=0 typiquement)
11 T      = 1.0      # temps final
12 N      = 500      # N intervalles => N+1 points de grille
```

```

13 dt      = 5e-4      # pas de temps
14
15 # Choix du solveur: "sparse" (par défaut) ou "banded"
16 SOLVER = "sparse"    # "sparse" / "banded"
17
18 # Fonctions sources et CL (modifiez librement)
19 def f(t, x):
20     return 0.0
21
22 def ul(t):
23     return 0.0
24
25 def g(t):
26     return 0.0
27
28 # Paramètres de la gaussienne de départ
29 xc      = 0.2 * L
30 sigma   = 0.05 * L
31 A_amp   = 1.0
32 # =====
33
34
35 # =====
36 # u0(x) compatible avec u(0)=ul(0) et u_x(L)=g(0)
37 # =====
38 def build_u0(x, A=A_amp, xc=xc, sigma=sigma):
39     G0 = np.exp(-((x - xc)**2) / (2.0 * sigma**2))
40     G0_at_0 = np.exp(-((0.0 - xc)**2) / (2.0 * sigma**2))
41     G0_at_L = np.exp(-(L - xc)**2) / (2.0 * sigma**2))
42
43     B = g(0.0) - A * (-(L - xc) / sigma**2) * G0_at_L
44     C = ul(0.0) - A * G0_at_0
45     return A * G0 + B * x + C
46
47
48 # =====
49 # Assemblage tri-diagonal (trois diagonales seulement)
50 #  $(I - dt \cdot L) u^{n+1} = u^n + dt f^{n+1}$ 
51 #  $L = -v D1_{upwind} + nu D2 - lam I$ 
52 #  $\Rightarrow A = I + dt \cdot v \cdot D1_{upwind} - dt \cdot nu \cdot D2 + dt \cdot lam \cdot I$ 
53 # CL: Dirichlet à gauche, Neumann à droite.
54 # =====
55 def build_tridiag(N, L, dt, v, nu, lam):
56     dx = L / N
57     n = N + 1
58
59     main = np.zeros(n)      # diagonale principale A[i,i]
60     lower = np.zeros(n - 1) # sous-diagonale A[i, i-1]
61     upper = np.zeros(n - 1) # sur-diagonale A[i, i+1]
62
63     use_backward = (v >= 0.0)
64     alpha = dt * nu / dx**2
65     beta = dt * v / dx
66
67     # Lignes intérieures i=1..N-1
68     for i in range(1, N):
69         # Diffusion centrale
70         main[i] += 1.0 + dt*lam + 2.0 * alpha

```

```

71     lower[i-1] += -alpha
72     upper[i]   += -alpha
73
74     # Advection upwind implicite
75     if use_backward:
76         main[i] += beta
77         lower[i-1] += -beta
78     else:
79         main[i] += -beta
80         upper[i] += beta
81
82     # Bord gauche (Dirichlet): u(.,0) = ul
83     main[0] = 1.0
84     # upper[0] reste 0, lower[0] n'est pas utilisé
85
86     # Bord droit (Neumann): (u_N - u_{N-1})/dx = g
87     main[-1] = 1.0 / dx
88     lower[-1] = -1.0 / dx
89     # upper[-1] inexistant pour la dernière ligne
90
91     return lower, main, upper
92
93
94 def step_rhs(u_prev, t_next, x, dt):
95     b = u_prev.copy()
96     b[1:-1] += dt * np.array([f(t_next, xi) for xi in x[1:-1]])
97     # CL
98     b[0] = ul(t_next) # Dirichlet
99     b[-1] = g(t_next) # Neumann (cohérent avec la dernière ligne)
100    return b
101
102
103 def solve_pde(L, v, nu, lam, T, N, dt, solver="sparse"):
104     x = np.linspace(0.0, L, N + 1)
105
106     # Assemblage tri-diagonal
107     lower, main, upper = build_tridiag(N, L, dt, v, nu, lam)
108     n = N + 1
109
110     # Préparation du solveur choisi (factorisation réutilisée si possible)
111     if solver == "sparse":
112         # Matrice creuse CSC + LU sparse réutilisable
113         from scipy.sparse import diags
114         from scipy.sparse.linalg import splu
115
116         A_csc = diags([lower, main, upper], offsets=[-1, 0, 1],
117                       shape=(n, n), format="csc")
118         lu = splu(A_csc) # factorisation unique
119         def solve_A(rhs):
120             return lu.solve(rhs)
121
122     elif solver == "banded":
123         # Stockage bande (l=u=1) pour solve_banded
124         # ab[0,1:] = upper ; ab[1,:] = main ; ab[2,:-1] = lower
125         from scipy.linalg import solve_banded
126         ab = np.zeros((3, n), dtype=float)
127         ab[0, 1:] = upper
128         ab[1, :] = main

```



```

129     ab[2, :-1]= lower
130     def solve_A(rhs):
131         # Remarque: solve_banded refactorise à chaque appel
132         return solve_banded((1, 1), ab, rhs, overwrite_ab=False, overwrite_b=
False, check_finite=False)
133     else:
134         raise ValueError("solver doit valoir 'sparse' ou 'banded'.")
135
136     # État initial compatible
137     u = build_u0(x)
138
139     # Petits checks CL
140     dx = L / N
141     dL_num = (u[-1] - u[-2]) / dx
142     if abs(u[0] - u1(0.0)) > 1e-8 or abs(dL_num - g(0.0)) > 1e-6:
143         print("[Avertissement] u0 n'est pas parfaitement compatible numériquement
avec les CL.")
144
145     # Intégration en temps
146     times_to_store = np.linspace(0.0, T, 5)
147     snapshots = [(0.0, u.copy())]
148     t = 0.0
149     next_store_idx = 1
150
151     while t < T - 1e-12:
152         t_next = min(t + dt, T)
153         b = step_rhs(u, t_next, x, dt)
154         u = solve_A(b) # résolution via le solveur choisi
155         t = t_next
156
157         while next_store_idx < len(times_to_store) and t >= times_to_store[
next_store_idx] - 1e-12:
158             snapshots.append((times_to_store[next_store_idx], u.copy()))
159             next_store_idx += 1
160
161     return x, snapshots
162
163
164     # =====
165     # Lancement + visualisation
166     # =====
167     x, snapshots = solve_pde(L, v, nu, lam, T, N, dt, solver=SOLVER)
168
169     plt.figure()
170     for (ti, ui) in snapshots:
171         plt.plot(x, ui, label=f"t={ti:.3f}")
172     plt.xlabel("x")
173     plt.ylabel("u(t,x)")
174     plt.title(f"Convection-Diffusion-Réaction 1D -- solveur: {SOLVER}")
175     plt.legend()
176     plt.tight_layout()
177     plt.savefig("Convection_Diffusion_Reaction_1D.png", dpi=300, bbox_inches="tight")
178     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

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

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)

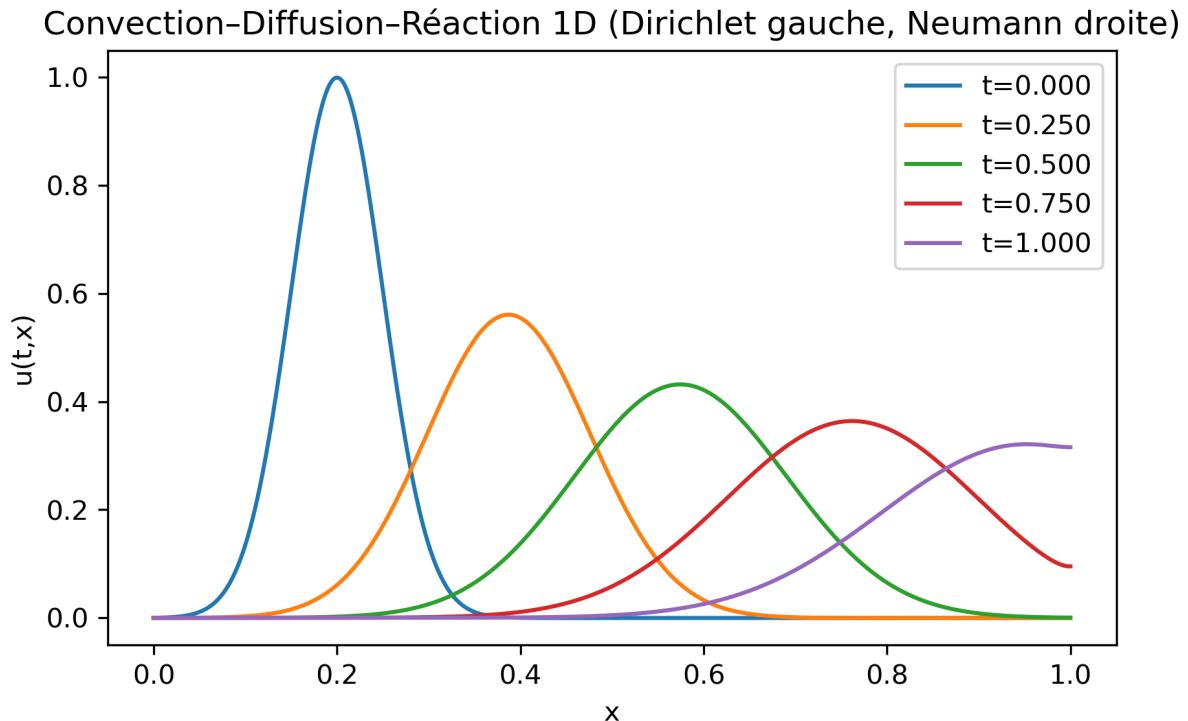


FIGURE 3.1 – Solution numérique  $u(t, x)$  pour  $v = 0.75$ ,  $\nu = 10^{-2}$ ,  $\lambda = 0$ ,  $L = 1$ , avec Dirichlet à gauche et Neumann (flux nul) à droite.

### 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.