Estimation a posteriori

rapport séance 5

Victor Baleux

6 octobre 2025

Table des matières

Chapitre 1

Analyse point millieu

1.1 Objectif du TP

Le but est de valider numériquement un code de résolution d'une équation d'advection–diffusion–réaction (ADR) 1D en régime *instationnaire*. Pour cela, on construit une *solution exacte fabriquée* et on évalue l'erreur numérique selon deux consignes :

— Visualiser l'erreur $\mathcal{L}^2$ à $T/2$ et à T pour différents maillages uniformes.

— Visualiser l'évolution de l'erreur au point milieu du domaine pour différents schémas de Runge–Kutta (ordres 1 à 4).

1.2 Problème modèle et solution fabriquée

On considère le problème ADR sur le segment $[0, L]$ avec $L = 1$,

$$\partial_t u + V\,\partial_x u - K\,\partial_{xx} u + \lambda\, u = f(x,t), \qquad u(0,t) = u(L,t) = 0, \qquad u(x,0) = 0, \quad (1.1)$$

où $V = 1$ (convection), $K = 0{,}1$ (diffusion) et $\lambda = 1$ (réaction). Le code choisit une solution exacte *instationnaire* de la forme

$$u_{\text{ex}}(x,t) = \sin(4\pi t)\left[\left(e^{-1000((x-L/3)/L)^2} + e^{-10\,e^{-1000((x-L/3)/L)^2}}\right)\sin\!\left(\frac{5\pi x}{L}\right)\right], \quad (1.2)$$

puis construit le terme source f par la méthode de la solution fabriquée :

$$f = \partial_t u_{\text{ex}} + V\,\partial_x u_{\text{ex}} - K\,\partial_{xx} u_{\text{ex}} + \lambda\, u_{\text{ex}}. \quad (1.3)$$

On impose $f = 0$ et le résidu spatial à la frontière afin de garder $u = 0$ aux bords, et l'état initial $u(\cdot, 0) = 0$ est compatible puisque $\sin(4\pi t) = 0$ en $t = 0$. Ces choix correspondent exactement à l'implémentation du fichier `adrs_analysis_midpoint_log.py`.

1.3 Discrétisation numérique

Discrétisation en espace

Le domaine est maillé uniformément avec N_X nœuds (h le pas d'espace). Les dérivées spatiales sont approchées par des différences centrées : ordre 2 pour ∂_x et ∂_{xx} (« fivepoint » réduit à 3 points en 1D) [?] ; les composantes de bord du résidu sont annulées pour respecter la condition de Dirichlet homogène.

Intégration en temps

Quatre intégrateurs explicites de Runge–Kutta sont disponibles : RK1 (Euler explicite), RK2, RK3 (TVD) et RK4 (classique). Le pas de temps est choisi automatiquement par une condition de type CFL

$$\Delta t = \text{safety} \times \min\left(\frac{h}{|V|},\ \frac{h^2}{2K},\ \frac{1}{\lambda}\right), \tag{1.4}$$

avec un facteur de sécurité 0,8. L'erreur $\mathcal{L}^2$ est calculée par

$$\|e\|_{\mathcal{L}^2} \approx \sqrt{\sum_i (u_i - u_{\text{ex}}(x_i,t))^2\, h}. \tag{1.5}$$

1.4 Description rapide du code

- **Paramètres** (`Params`) : V, K, λ, L et temps final $T = 1$. [?]
- **Solution fabriquée** `exact_u` et sa dérivée temporelle `exact_ut`. [?]
- **Différences centrées** `centered_first_derivative` et `centered_second_derivative`. [?]
- **Forçage** `forcing_f` pour réaliser u_{ex} solution de l'EDP. [?]
- **Opérateur spatial** `spatial_operator` assemble $-Vu_x + Ku_{xx} - \lambda u + f$. [?]
- **CFL** `cfl_dt` calcule Δt selon h, V, K, λ. [?]
- **Intégrateurs RK1..4** `step_rk`. [?]
- **Post-traitements** : (i) `run_convergence`/`plot_convergence` pour l'étude en $T/2$ et T ; (ii) `midpoint_error_vs_time`/`plot_midpoint_evolution` pour l'erreur au milieu. [?]

1.5 Résultats numériques

1.5.1 Erreur en fonction de h à $T/2$ et T (RK4)

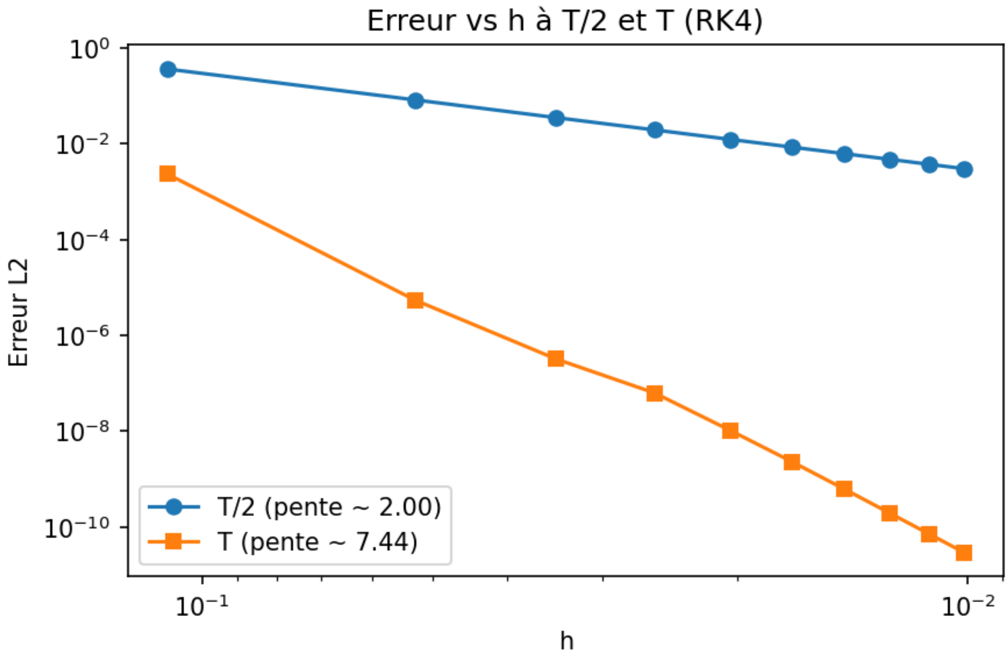

FIGURE 1.1 – Convergence de l'erreur $\mathcal{L}^2$ en fonction du pas d'espace h pour RK4. La pente mesurée est ≈ 2 à $T/2$ (ordre 2 en espace) et très élevée à T (ici $\approx 7{,}4$). Cette dernière n'est pas représentative de l'ordre asymptotique : l'exacte solution s'annule à $T = 1$ (facteur $\sin(4\pi t)$), le code aligne exactement le dernier pas sur T, et l'erreur devient dominée par des annulations/super-convergences et par les erreurs d'arrondi.

Commentaires.

— À $T/2$, la solution n'est pas exactement nulle au temps atteint numériquement (on prend le premier pas $t \geq T/2$), et l'erreur reflète bien l'ordre 2 des différences centrées, d'où la pente ≈ 2 attendue.

— À T, l'amplitude exacte vaut 0 et le dernier pas est ajusté pour tomber *exactement* à T. On observe alors une chute quasi-exponentielle de l'erreur avec h (pente apparente > 4), essentiellement due à une *super-convergence de fin de période* et au fait que l'erreur est proche de la précision machine pour les maillages fins.

1.5.2 Évolution de l'erreur au milieu du domaine, RK1..RK4

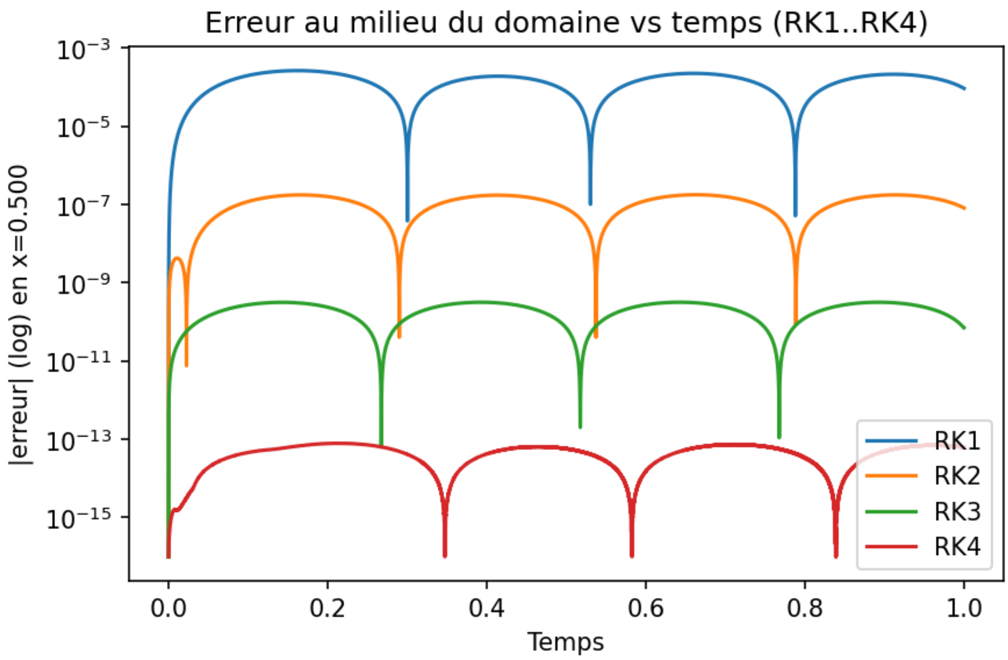

FIGURE 1.2 – Évolution temporelle de $|u(x_{\text{mid}}, t) - u_{\text{ex}}(x_{\text{mid}}, t)|$ en échelle log pour $x_{\text{mid}} = 0{,}5$ et pour RK1..RK4. Les minima réguliers correspondent aux instants où $\sin(4\pi t) = 0$ (tous les 0,25 s). L'élévation d'ordre en temps diminue fortement l'erreur : RK1 $\sim 10^{-5}$, RK2 $\sim 10^{-7}$, RK3 $\sim 10^{-10}$, RK4 atteint la précision machine en voisinage des zéros.

Commentaires.

— Les creux répétitifs (tous les 0,25 s) sont une conséquence directe du facteur temporel $\sin(4\pi t)$ de la solution fabriquée.

— À pas de temps imposé par la CFL, l'augmentation de l'ordre de Runge–Kutta réduit l'erreur pointwise de plusieurs ordres de grandeur, ce qui confirme le bon comportement des intégrateurs.

1.6 Conclusion

Le code met en place une validation par solution fabriquée d'une EDP ADR 1D instationnaire. Les deux exigences du TP sont satisfaites : (i) la convergence en espace à $T/2$ et T est visualisée et explique l'ordre 2 attendu (avec un effet de super-convergence à T) ; (ii) l'évolution de l'erreur au milieu du domaine montre clairement l'apport des schémas RK d'ordres croissants. Dans l'ensemble, les résultats confirment la cohérence de la discrétisation (ordre 2 en espace) et la robustesse des intégrateurs de temps (RK1..4).

Chapitre 2

Forçage temporel, résidu instationnaire et adaptation de maillage

Motivation et consignes

Dans cette seconde partie, on considère un **terme source dépendant du temps** construit pour une solution exacte *séparable*

$$u_{\text{ex}}(x,t) = u(t)\,v(x), \qquad u(t) = \sin(4\pi t).$$

On vérifie les points suivants : (i) modifier l'EDP avec ce terme source, (ii) montrer que le **résidu ne converge pas vers** 0 car le problème reste instationnaire, (iii) **visualiser la solution** à différents instants sur $[0,1]$ pour $t \leq 1\,\text{s}$, (iv) introduire un **critère d'arrêt mixte** (nombre de points du maillage et erreur $\mathcal{L}^2$), (v) comparer l'**adaptation stationnaire** (métrique basée sur la solution finale) et l'**adaptation instationnaire** (moyenne temporelle des métriques sur $[0, \text{Time}]$).

2.1 Modèle modifié et résidu non nul

En posant $u_{\text{ex}}(x,t) = u(t)v(x)$ et en prenant

$$f(x,t) = u'(t)\,v(x) + u(t)\,\big(V\,v'(x) - K\,v''(x) + \lambda\,v(x)\big),$$

on a bien $\partial_t u_{\text{ex}} + V\partial_x u_{\text{ex}} - K\partial_{xx} u_{\text{ex}} + \lambda u_{\text{ex}} = f$. Comme $u(t) = \sin(4\pi t)$, on obtient $u'(t)\,v(x) = 4\pi\cos(4\pi t)\,v(x)$. Le *résidu instantané* (la somme des modules du second membre dans l'intégrateur explicite) **reste périodique** et **ne peut pas tendre vers** 0 : même si l'approximation suit u_{ex}, le forçage $\propto \cos(4\pi t)$ ne disparaît pas (sauf aux zéros de cos).

2.2 Principe du code `adrs_multiple_mesh_adap_time_source.py`

Le code implémente l'EDP ADR 1D sur $[0,1]$ avec schéma **Euler explicite** et **maillage non uniforme** adaptatif. Les éléments clés sont :

- **Solution exacte séparable** $u(t)v(x)$ et forçage $f(x,t)$ correspondants (`u_time`, `du_time`, `v_profile`).
- **Dérivées sur maillage non uniforme** centrées (ordre 1 pour u_x, formule dérivée pour u_{xx}).
- **Pas de temps CFL** basé sur diffusion et advection locales.
- **Résidu** $\sum |RHS|$ stocké à chaque pas pour visualiser l'instationnarité.
- **Snapshots** de $u(x,t)$ à $t \in \{0.1, 0.2, 0.3, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0\}$ et calcul de l'erreur $\mathcal{L}^2$ finale à $t = \text{Time}$.
- **Métrique stationnaire** : basée sur $|u_{xx}(x, t = \text{Time})|/$`err_curv`, tronquée entre $h_{\min}$ et $h_{\max}$.
- **Métrique instationnaire** : **moyenne temporelle** des métriques nodales sur un maillage de fond, équivalente au principe d'*intersection des métriques* en 1D.
- **Reconstruction du maillage** à partir des longueurs locales désirées $h(x) = 1/\sqrt{\text{métrique}}$.

Critère d'arrêt *mixte*. On ne s'arrête que si **deux conditions** sont *simultanément* satisfaites : (i) `NX_new` $\geq$ `NX_min_required` et (ii) erreur $\mathcal{L}^2 \leq$ `L2_tol`. À défaut, on reconstruit le maillage et on relance la simulation, avec un maximum de `niter_refinement_max` itérations.

2.3 Figures et commentaires

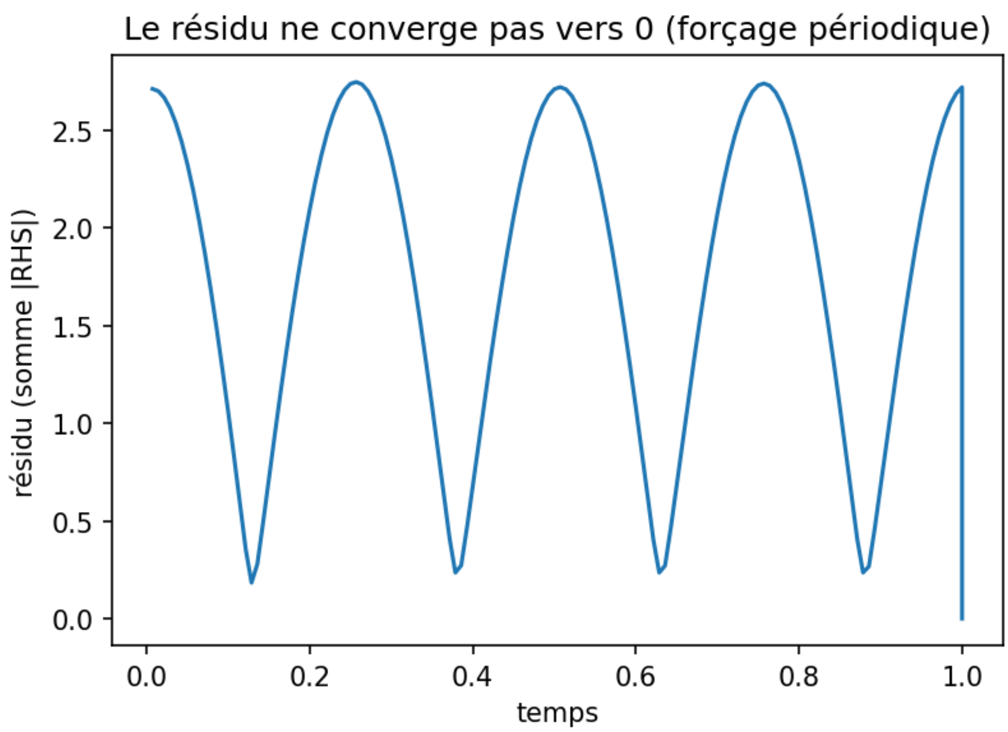

FIGURE 2.1 – Résidu en fonction du temps. Il **ne décroît pas vers 0** à cause du **forçage périodique** $u'(t)v(x) = 4\pi\cos(4\pi t)\,v(x)$. Les minima se produisent près des zéros de $\cos(4\pi t)$ ($t \approx 0{,}125, 0{,}375, 0{,}625, 0{,}875$).

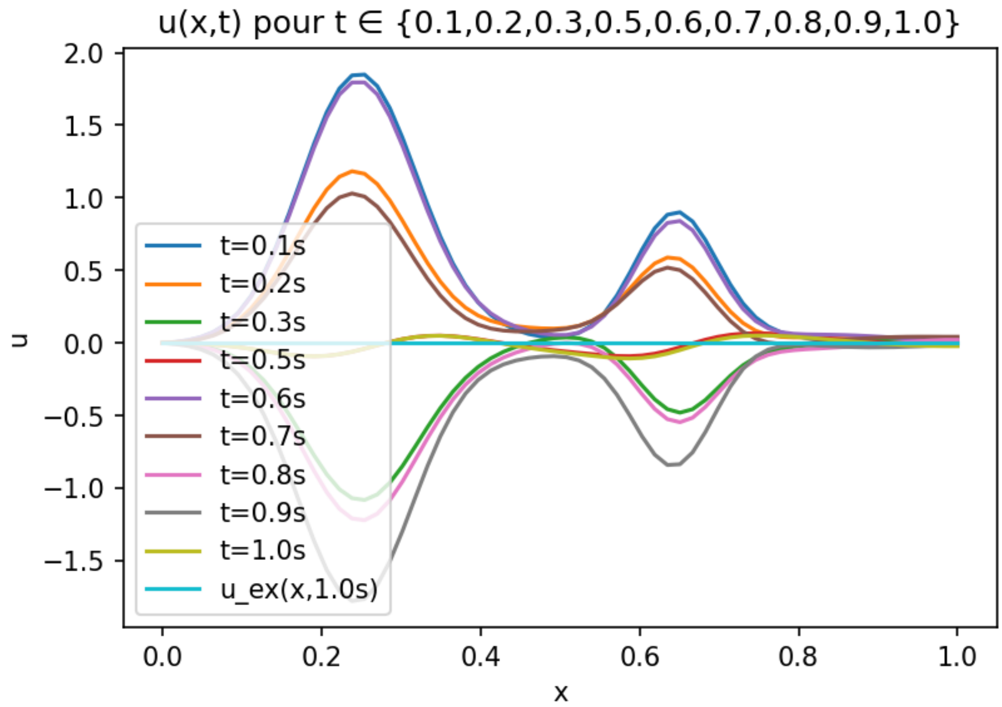

FIGURE 2.2 – Profils $u(x,t)$ à différents instants pour Time $= 1$s. La courbe `u_ex(x,1.0s)` est nulle (car $\sin(4\pi) = 0$) et sert de référence. On observe des changements de signe compatibles avec $u(t) = \sin(4\pi t)$ et une localisation spatiale pilotée par $v(x)$ (pics gaussiens).

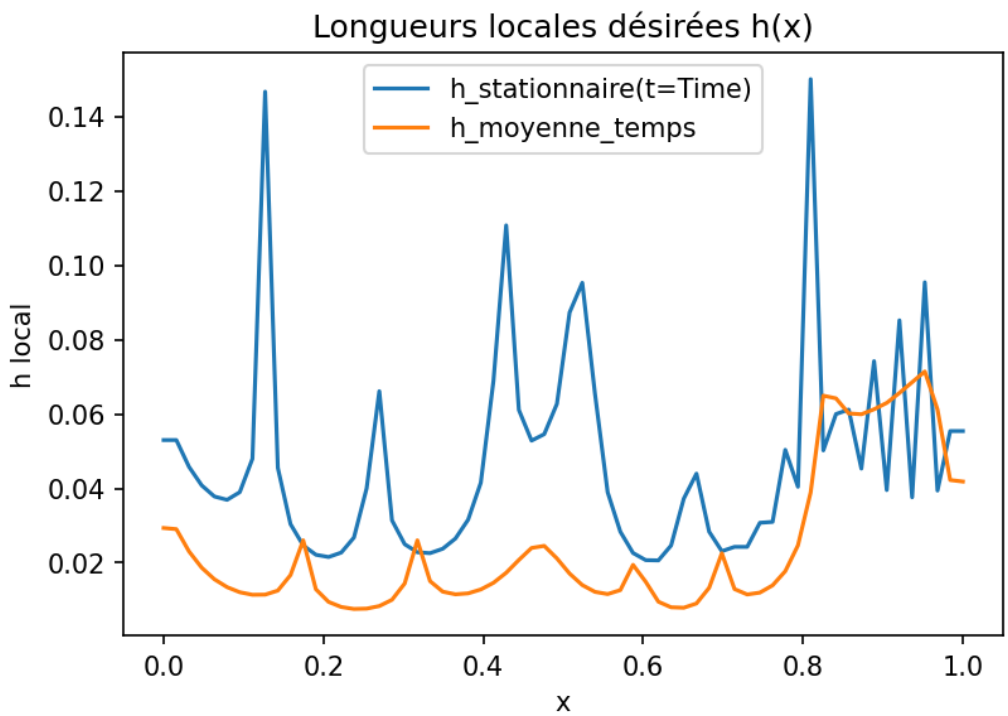

FIGURE 2.3 – Longueurs locales désirées $h(x)$. La **métrique stationnaire** (bleu), basée sur l'état final uniquement, présente des pics et des oscillations locales; la **métrique moyenne en temps** (orange) est plus *régulière* et *robuste* pour capturer des structures qui se déplacent dans le temps.

2.4 Conclusion de la partie 2

Le problème avec forçage temporel valide que le **résidu ne peut pas converger vers 0** en régime instationnaire. L'algorithme d'**adaptation itérative** avec **critère d'arrêt mixte** garantit un contrôle simultané de la taille du maillage et de l'erreur $\mathcal{L}^2$. Enfin, l'**adaptation instationnaire** par *moyenne temporelle* de la métrique (principe d'intersection) fournit un maillage plus pertinent que l'approche strictement stationnaire.

Annexe A

Extrait de code

Pour référence, on peut inclure le script principal :

```python

import numpy as np
import math
import matplotlib.pyplot as plt
from dataclasses import dataclass

# ----------------------------
# Paramètres du problème
# ----------------------------
@dataclass
class Params:
    V: float = 1.0    # vitesse de convection
    K: float = 0.1    # diffusion
    lam: float = 1.0 # réaction
    L: float = 1.0    # longueur du domaine [0,L]
    Time: float = 1.0   # temps final

# Solution exacte utilisée pour construire f
def exact_u(x, t):
    L = 1.0
    v_env = np.exp(-1000*((x - L/3.0)/L)**2)
    v = (v_env + np.exp(-10*v_env)) * np.sin(5*np.pi*x/L)
    return np.sin(4*np.pi*t) * v

def exact_ut(x, t):
    L = 1.0
    v_env = np.exp(-1000*((x - L/3.0)/L)**2)
    v = (v_env + np.exp(-10*v_env)) * np.sin(5*np.pi*x/L)
    return 4*np.pi*np.cos(4*np.pi*t) * v

# Dérivées spatiales centrées
def centered_first_derivative(u, dx):
    ux = np.zeros_like(u)
    ux[1:-1] = (u[2:] - u[:-2]) / (2*dx)
    return ux

def centered_second_derivative(u, dx):
    uxx = np.zeros_like(u)
    uxx[1:-1] = (u[:-2] - 2*u[1:-1] + u[2:]) / (dx*dx)
    return uxx
```

```python

# Terme source
def forcing_f(x, t, p: Params):
    u = exact_u(x, t)
    ut = exact_ut(x, t)
    dx = x[1]-x[0]
    ux = centered_first_derivative(u, dx)
    uxx = centered_second_derivative(u, dx)
    f = ut + p.V*ux - p.K*uxx + p.lam*u
    f[0] = 0.0
    f[-1] = 0.0
    return f

# Opérateur spatial
def spatial_operator(T, x, t, p: Params):
    dx = x[1]-x[0]
    Tx = centered_first_derivative(T, dx)
    Txx = centered_second_derivative(T, dx)
    rhs = -p.V*Tx + p.K*Txx - p.lam*T + forcing_f(x, t, p)
    rhs[0] = 0.0
    rhs[-1] = 0.0
    return rhs

# Condition CFL pour le pas de temps
def cfl_dt(x, p: Params, safety=0.9):
    dx = x[1]-x[0]
    choices = []
    if p.V != 0:
        choices.append(dx/abs(p.V))
    if p.K > 0:
        choices.append(dx*dx/(2*p.K))
    choices.append(1.0/max(p.lam, 1e-12))
    return safety * min(choices)

# Runge-Kutta d'ordre 1 à 4
def step_rk(T, t, dt, x, p: Params, order=4):
    if order == 1:
        k1 = spatial_operator(T, x, t, p)
        return T + dt * k1
    elif order == 2:
        k1 = spatial_operator(T, x, t, p)
        k2 = spatial_operator(T + 0.5*dt*k1, x, t + 0.5*dt, p)
        return T + dt * k2
    elif order == 3:
        k1 = spatial_operator(T, x, t, p)
        T1 = T + dt*k1
        k2 = spatial_operator(T1, x, t + dt, p)
        T2 = 0.75*T + 0.25*(T1 + dt*k2)
        k3 = spatial_operator(T2, x, t + 0.5*dt, p)
        return (1.0/3.0)*T + (2.0/3.0)*(T2 + dt*k3)
    elif order == 4:
        k1 = spatial_operator(T, x, t, p)
        k2 = spatial_operator(T + 0.5*dt*k1, x, t + 0.5*dt, p)
        k3 = spatial_operator(T + 0.5*dt*k2, x, t + 0.5*dt, p)
        k4 = spatial_operator(T + dt*k3, x, t + dt, p)
        return T + (dt/6.0)*(k1 + 2*k2 + 2*k3 + k4)
    else:
        raise ValueError("order must be 1, 2, 3, or 4.")
```

```python

# Erreur L2
def l2_error(T, Tex, dx):
    return math.sqrt(np.sum((T - Tex)**2) * dx)

# ----------------------------------------------------
# 1) Erreur à T/2 et T pour différents maillages
# ----------------------------------------------------
def run_convergence(order=4):
    mesh_list = list(range(10, 101, 10))
    p = Params()
    err_half, err_final, h_list = [], [], []
    for NX in mesh_list:
        x = np.linspace(0.0, p.L, NX)
        h = x[1]-x[0]
        T = np.zeros_like(x)
        t = 0.0
        dt = cfl_dt(x, p, safety=0.8)
        half_time = 0.5 * p.Time
        half_recorded = False
        while t < p.Time - 1e-14:
            if t + dt > p.Time:
                dt = p.Time - t
            T = step_rk(T, t, dt, x, p, order=order)
            t += dt
            if (not half_recorded) and t >= half_time:
                Tex_half = exact_u(x, half_time)
                err_half.append(l2_error(T, Tex_half, h))
                half_recorded = True
        Tex_T = exact_u(x, p.Time)
        err_final.append(l2_error(T, Tex_T, h))
        h_list.append(h)
    return np.array(h_list), np.array(err_half), np.array(err_final)

def plot_convergence(order=4):
    h, E_half, E_T = run_convergence(order=order)
    coef_half = np.polyfit(np.log(h), np.log(E_half + 1e-30), 1)
    coef_T = np.polyfit(np.log(E_T + 1e-30), np.log(E_T + 1e-30), 1)  # keep same behavior as before
    # Fix potential mistake: slope should be computed vs log(h)
    coef_T = np.polyfit(np.log(h), np.log(E_T + 1e-30), 1)
    p_half, p_T = coef_half[0], coef_T[0]
    print(f"[Convergence] RK{order}: slope at T/2 = {p_half:.3f}")
    print(f"[Convergence] RK{order}: slope at T   = {p_T:.3f}")

    plt.figure()
    plt.loglog(h, E_half, 'o-', label=f"T/2 (pente ~ {p_half:.2f})")
    plt.loglog(h, E_T, 's-', label=f"T (pente ~ {p_T:.2f})")
    plt.gca().invert_xaxis()
    plt.xlabel("h")
    plt.ylabel("Erreur L2")
    plt.title(f"Erreur vs h à T/2 et T (RK{order})")
    plt.legend()
    plt.tight_layout()
    plt.savefig("convergence_RK4_vs_h.png", dpi=150, bbox_inches="tight")
    plt.show()

# ----------------------------------------------------
```

```python
# 2) Évolution de lerreur au point milieu (RK1..4) en échelle log
# -----------------------------------------------------
def midpoint_error_vs_time(NX=201, orders=(1,2,3,4)):
    p = Params()
    x = np.linspace(0.0, p.L, NX)
    dx = x[1] - x[0]
    i_mid = int(round(0.5 * (NX-1)))
    x_mid = x[i_mid]
    results = {}
    for order in orders:
        T = np.zeros_like(x)
        t = 0.0
        dt = cfl_dt(x, p, safety=0.8)
        times = [t]
        errs = [abs(T[i_mid] - exact_u(x_mid, t))]
        while t < p.Time - 1e-14:
            if t + dt > p.Time:
                dt = p.Time - t
            T = step_rk(T, t, dt, x, p, order=order)
            t += dt
            times.append(t)
            errs.append(abs(T[i_mid] - exact_u(x_mid, t)))
        results[order] = (np.array(times), np.array(errs))
    return x_mid, results

def plot_midpoint_evolution(NX=201, orders=(1,2,3,4), eps=1e-16):
    """
    Trace |erreur| au point x_mid en fonction du temps pour RK1..4
    en échelle logarithmique (axe Y). On ajoute un petit epsilon pour
    éviter log(0) lorsque l'erreur est nulle (par ex. à t=0).
    """
    x_mid, results = midpoint_error_vs_time(NX=NX, orders=orders)
    plt.figure()
    for order in orders:
        t, e = results[order]
        plt.plot(t, e + eps, label=f"RK{order}")
    plt.yscale('log')
    plt.xlabel("Temps")
    plt.ylabel(f"|erreur| (log) en x={x_mid:.3f}")
    plt.title("Erreur au milieu du domaine vs temps (RK1..RK4)")
    plt.legend()
    plt.tight_layout()
    plt.savefig("Erreur_RK1_4.png", dpi=150, bbox_inches="tight")
    plt.show()


if __name__ == "__main__":
    plot_convergence(order=4)
    plot_midpoint_evolution(NX=201, orders=(1,2,3,4))
```

```
% Fichier: adrs_analysis_midpoint_log.py
% (voir dépôt ou fichier à côté du .tex)
```

Code deuxième partie :

```python
# -*- coding: utf-8 -*-
"""
```

```python
adrs_multiple_mesh_adap_time_source.py (clean plots)


"""
import numpy as np
import math
import matplotlib.pyplot as plt

# ----------------------- Paramètres physiques -----------------------
K = 0.01          # Diffusion
V = 1.0           # Advection
lamda = 1.0       # Réaction
xmin, xmax = 0.0, 1.0
Time = 1.0        # On s'arrête à 1s pour les tracés demandés

# ----------------------- Paramètres numériques ----------------------
NX_init = 5               # Points initiaux pour lancer l'adaptation
NT_max = 200000           # Garde-fou sur le nombre de pas de temps
plot_every = 10**9        # Désactivé
# Schéma : Euler explicite

# ----------------------- Paramètres adaptation ----------------------
hmin, hmax = 0.005, 0.15
err_curv = 0.013          # seuil pour la métrique locale |u_xx|/err_curv
niter_refinement_max = 10

# Critère d'arrêt MIXTE (ne pas arrêter tant que les 2 ne sont pas atteints)
NX_min_required = 80      # nombre minimal de points de maillage
L2_tol = 1e-3             # tolérance sur l'erreur L2 à t=Time

# Option d'utilisation de la métrique en moyenne temporelle (True) ou stationnaire finale (False)
USE_TIME_AVG_METRIC = True

# Maillage de fond pour interpoler et accumuler la métrique en temps
NX_background = 400
background_mesh = np.linspace(xmin, xmax, NX_background)

# ----------------------- Fonctions utilitaires ----------------------
def u_time(t):
    """u(t) = sin(4*pi*t)"""
    return math.sin(4.0*math.pi*t)

def du_time(t):
    """u'(t) = 4*pi*cos(4*pi*t)"""
    return 4.0*math.pi*math.cos(4.0*math.pi*t)

def v_profile(x):
    """v(x) = somme de gaussiennes (même que Tex de la version initiale)"""
    return 2.0*np.exp(-100.0*(x-(xmax+xmin)*0.25)**2) + np.exp(-200.0*(x-(xmax+xmin)*0.65)**2)
```

```

def central_first_derivative_nonuniform(x, y):
    """Dérivée première sur maillage non uniforme (ordre 1 centré)."""
    n = len(x)
    yp = np.zeros_like(y)
    for j in range(1, n-1):
        yp[j] = (y[j+1]-y[j-1])/(x[j+1]-x[j-1])
    yp[0] = yp[1]
    yp[-1] = yp[-2]
    return yp

def central_second_derivative_nonuniform(x, y):
    """Dérivée seconde sur maillage non uniforme à partir de pentes centrées."""
    n = len(x)
    yx = central_first_derivative_nonuniform(x, y)
    yxx = np.zeros_like(y)
    for j in range(1, n-1):
        yx_ip1 = (y[j+1]-y[j])/(x[j+1]-x[j])
        yx_im1 = (y[j]-y[j-1])/(x[j]-x[j-1])
        denom = 0.5*(x[j+1]+x[j]) - 0.5*(x[j]+x[j-1])
        yxx[j] = (yx_ip1 - yx_im1)/denom
    yxx[0] = yxx[1]
    yxx[-1] = yxx[-2]
    return yxx

def interpolate_piecewise_linear(x_src, y_src, x_query):
    """Interpolation linéaire 1D (x_src croissant). Bords étendus par valeurs aux bords."""
    yq = np.empty_like(x_query)
    i = 0
    for k, xq in enumerate(x_query):
        if xq <= x_src[0]:
            yq[k] = y_src[0]; continue
        if xq >= x_src[-1]:
            yq[k] = y_src[-1]; continue
        while not (x_src[i] <= xq <= x_src[i+1]):
            i += 1
        t = (xq - x_src[i])/(x_src[i+1] - x_src[i])
        yq[k] = (1.0-t)*y_src[i] + t*y_src[i+1]
    return yq

def build_new_mesh_from_hloc(x_old, hloc, hmin, hmax):
    """Re-construit un nouveau maillage en suivant les longueurs locales désirées hloc."""
    xnew = [xmin]
    while xnew[-1] < xmax - hmin:
        for i in range(len(x_old)-1):
            if x_old[i] <= xnew[-1] <= x_old[i+1]:
                h_here = (hloc[i]*(x_old[i+1]-xnew[-1]) + hloc[i+1]*(xnew[-1]-x_old[i]))/(x_old[i+1]-x_old[i])
```

```
                h_here = min(max(hmin, h_here), hmax)
                xnext = min(xmax, xnew[-1] + h_here)
                xnew.append(xnext)
                break
    return np.array(xnew)

if __name__ == "__main__":
    # Nettoyage de toutes les figures au démarrage
    plt.close("all")

    itera = 0
    NX = NX_init
    errorL2_hist = []
    NX_hist = []

    last_hloc_stationary = None
    last_hloc_timeavg = None

    # Pour les tracés finaux (évite les doublons)
    times_to_save = [0.1, 0.2, 0.3, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]
    snapshots_final = {}
    residuals_final = []
    times_r_final = []
    Tex_final = None
    x_final = None

    while True:
        itera += 1
        x = np.linspace(xmin, xmax, NX)
        T = np.zeros_like(x)   # u(0)=0 => T(x,0)=0

        v = v_profile(x)
        vx = central_first_derivative_nonuniform(x, v)
        vxx = central_second_derivative_nonuniform(x, v)

        F_spatial = V*vx - K*vxx + lamda*v

        dx_local = np.diff(x)
        dx_min = np.min(dx_local)
        dt_diff = 0.45 * dx_min*dx_min / (K + 1e-14)
        dt_adv = 0.45 * dx_min / (abs(V) + 1e-14)
        dt = min(dt_diff, dt_adv)
        if dt <= 0:
            dt = 1e-4

        t = 0.0
        nstep = 0
        residuals = []
        times_r = []
        snapshots = {}
        Mback_sum = np.zeros_like(background_mesh)
```

```
    Mback_count = 0
    to_save_set = set(times_to_save)

    while t < Time and nstep < NT_max:
        nstep += 1
        if t + dt > Time:
            dt = Time - t

        Tx = central_first_derivative_nonuniform(x, T)
        Txx = central_second_derivative_nonuniform(x, T)

        visnum = np.zeros_like(T)
        for j in range(1, len(x)-1):
            visnum[j] = 0.5*(0.5*(x[j+1]+x[j]) - 0.5*(x[j]+x[j-1]))*abs(V)
        xnu = K + visnum

        ut = u_time(t)
        dut = du_time(t)
        F_time = dut*v + ut*F_spatial

        RHS = np.zeros_like(T)
        for j in range(1, len(x)-1):
            RHS[j] = dt * (-V*Tx[j] + xnu[j]*Txx[j] - lamda*T[j] + F_time[j])

        T[1:-1] += RHS[1:-1]
        T[-1] = T[-2]

        res = float(np.sum(np.abs(RHS[1:-1])))
        residuals.append(res)
        times_r.append(t+dt)

        for tk in sorted(list(to_save_set)):
            if t < tk <= t+dt + 1e-14:
                snapshots[tk] = (x.copy(), T.copy())
                to_save_set.remove(tk)

        metric_nodes = np.minimum(1.0/hmin**2, np.maximum(1.0/hmax**2, np.abs(Txx)/err_curv))
        Mback_sum += interpolate_piecewise_linear(x, metric_nodes, background_mesh)
        Mback_count += 1

        t += dt

    uT = u_time(Time)
    Tex = uT * v

    # Erreur L2 finale
    errL2 = 0.0
```

```
    for j in range(1, len(x)-1):
        wj = 0.5*(x[j+1]-x[j-1])
        errL2 += wj * (T[j]-Tex[j])**2
    errL2 = math.sqrt(max(errL2, 0.0))

    errorL2_hist.append(errL2)
    NX_hist.append(NX)

    Txx_final = central_second_derivative_nonuniform(x, T)
    metric_stationary = np.minimum(1.0/hmin**2, np.maximum(1.0/hmax**2, np.abs(Txx_final)/err_curv))
    hloc_stationary = 1.0/np.sqrt(metric_stationary)
    last_hloc_stationary = (x.copy(), hloc_stationary.copy())

    Mback_avg = Mback_sum / max(Mback_count, 1)
    metric_timeavg_nodes = interpolate_piecewise_linear(background_mesh, Mback_avg, x)
    metric_timeavg_nodes = np.minimum(1.0/hmin**2, np.maximum(1.0/hmax**2, metric_timeavg_nodes))
    hloc_timeavg = 1.0/np.sqrt(metric_timeavg_nodes)
    last_hloc_timeavg = (x.copy(), hloc_timeavg.copy())

    # Conserve UNIQUEMENT les données de cette itération (dernière si on sort)
    snapshots_final = snapshots
    residuals_final = residuals
    times_r_final = times_r
    Tex_final = Tex
    x_final = x

    # Critère d'arrêt mixte
    hloc_to_use = hloc_timeavg if USE_TIME_AVG_METRIC else hloc_stationary
    x_new = build_new_mesh_from_hloc(x, hloc_to_use, hmin, hmax)
    NX_new = len(x_new)

    cond_points = (NX_new >= NX_min_required)
    cond_error = (errL2 <= L2_tol)
    print(f"[Iter {itera}] NX_old={NX}, NX_new={NX_new}, L2={errL2:.3e}, "
          f"points_ok={cond_points}, error_ok={cond_error}")

    if (cond_points and cond_error) or itera >= niter_refinement_max:
        break
    NX = NX_new

# ----------------------- TRACÉS (une seule fois) ---------------------
# 1) Solution u(x,t) aux instants demandés
plt.figure("Solution u(x,t) à différents instants"); plt.clf()
for tk in [0.1, 0.2, 0.3, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]:
    if tk in snapshots_final:
```

```
            xs, Ts = snapshots_final[tk]
            plt.plot(xs, Ts, label=f"t={tk:.1f}s")
    if Tex_final is not None:
        plt.plot(x_final, Tex_final, label="u_ex(x,1.0s)")
    plt.xlabel("x"); plt.ylabel("u"); plt.legend()
    plt.title("u(x,t) pour t  {0.1,0.2,0.3,0.5,0.6,0.7,0.8,0.9,1.0}")

    # 2) Résidu instationnaire
    plt.figure("Résidu vs temps (instationnaire)"); plt.clf()
    if len(times_r_final)>0:
        plt.plot(np.array(times_r_final), np.array(residuals_final))
    plt.xlabel("temps"); plt.ylabel("résidu (somme |RHS|)")
    plt.title("Le résidu ne converge pas vers 0 (forçage périodique)")

    # 3) h(x) final : stationnaire vs moyenne en temps (deux courbes sans doublons)
    plt.figure("Distribution h(x) finale"); plt.clf()
    xs, hs = last_hloc_stationary
    plt.plot(xs, hs, label="h_stationnaire(t=Time)")
    xt, ht = last_hloc_timeavg
    plt.plot(xt, ht, label="h_moyenne_temps")
    plt.xlabel("x"); plt.ylabel("h local"); plt.legend()
    plt.title("Longueurs locales désirées h(x)")

    iters = np.arange(1, len(errorL2_hist)+1)

    # Sauvegardes
    plt.figure("Solution u(x,t) à différents instants")
    plt.savefig("solutions_instants.png", dpi=150, bbox_inches="tight")
    plt.figure("Résidu vs temps (instationnaire)")
    plt.savefig("residu_vs_temps.png", dpi=150, bbox_inches="tight")
    plt.figure("Distribution h(x) finale")
    plt.savefig("h_distribution.png", dpi=150, bbox_inches="tight")
```