Estimation a posteriori

rapport séance 6

Victor Baleux

13 octobre 2025

Table des matières

Objectif du TP et consignes

Optimiser un problème ADRS (Advection–Diffusion–Réaction avec Sources) en exploitant la **linéarité** de l'équation d'état vis-à-vis des contrôles. On note $x = (x_1, \ldots, x_{N_c})$ le vecteur des contrôles et $u(x)$ l'état au régime établi. La fonctionnelle à minimiser est

$$J(x) = \tfrac{1}{2} \, \|u(x) - u_{\text{des}}\|^2_{L^2(0,L)}. \tag{1}$$

Les questions posées portent sur: (i) le tracé de la surface $J(x_1, x_2)$ avec les autres contrôles fixés; (ii) le calcul numérique de $A_{ij} = \langle u_i, u_j \rangle$ et $b_i = \langle u_i, u_{\text{des}} - u_0 \rangle$ lorsque les champs u_i et u_{des} ne vivent pas sur le même maillage (adaptation) ; (iii) la précision d'intégration à viser ; (iv) l'implémentation (adaptation et interpolation inter-maillages) et la comparaison avec une solution de référence sur maillage fixe fin.

Modèle et linéarité

Le PDE traité (à l'état stationnaire via pseudo-temps) est

$$u_t + V \, u_x = K \, u_{xx} - \lambda u + \sum_{i=1}^{N_c} x_i \, g_i(x), \qquad u(0) = u(L) = 0.$$

Par linéarité vis-à-vis des sources, la solution s'écrit $u(x) = u_0 + \sum_{i=1}^{N_c} x_i \, u_i$, où u_0 est la solution avec $x \equiv 0$ et u_i la solution avec unitaire sur le contrôle i. On obtient alors

$$J(x) = \tfrac{1}{2} \left\| u_0 + \sum_i x_i u_i - u_{\text{des}} \right\|^2_{L^2} = \tfrac{1}{2} x^\top A x - b^\top x + c, \tag{2}$$

$$A_{ij} = \langle u_i, u_j \rangle, \qquad b_i = \langle u_i, \, u_{\text{des}} - u_0 \rangle, \qquad c = \tfrac{1}{2} \|u_{\text{des}} - u_0\|^2. \tag{3}$$

La condition du premier ordre $\nabla J(x) = 0$ donne le système linéaire $A \, x^* = b$ dont la solution est le contrôle optimal x^*.

Calcul de A_{ij} et b_i avec maillages adaptés différents (réponse)

Lorsque u_i et u_j sont connus chacun sur *son* maillage adapté, ils ne sont pas co-localisés. Nous procédons ainsi:

1. **Interpolation croisée**: on interpole linéairement u_i et u_j sur une grille *commune* uniforme $x_q = \{0, \dots, L\}$ de taille N.
2. **Quadrature adaptative trapézoïdale**: on évalue $\int_0^L u_i u_j \, dx$ par une règle des trapèzes uniforme et on *double* N jusqu'à atteindre $|I_N - I_{2N}| < \max(\varepsilon_{\text{abs}}, \varepsilon_{\text{rel}} |I_{2N}|)$.
3. Même schéma pour $b_i = \int_0^L u_i(u_{\text{des}} - u_0) \, dx$.

Ce procédé est robuste et indépendant des maillages d'origine ; il garantit que les produits scalaires L^2 sont évalués à la précision demandée.

Précision d'intégration (réponse). Pour ne pas *dégrader* l'erreur de discrétisation (liée à h), il faut que l'erreur d'intégration soit *nettement plus petite*. Une règle pratique est de prendre $\varepsilon_{\text{rel}} \lesssim 10^{-2} \, h^p$ où p est l'ordre effectif (ici, schémas centrés/interpolation linéaire $\Rightarrow$ $p \approx 1$). Dans le code, on utilise par défaut $\varepsilon_{\text{rel}} = 10^{-8}$ et $\varepsilon_{\text{abs}} = 10^{-10}$, largement en-dessous des erreurs de discrétisation visées, ce qui évite d'introduire une erreur supplémentaire notable au niveau de A_{ij} et b_i.

Algorithme, adaptation et interpolation (réponse & description)

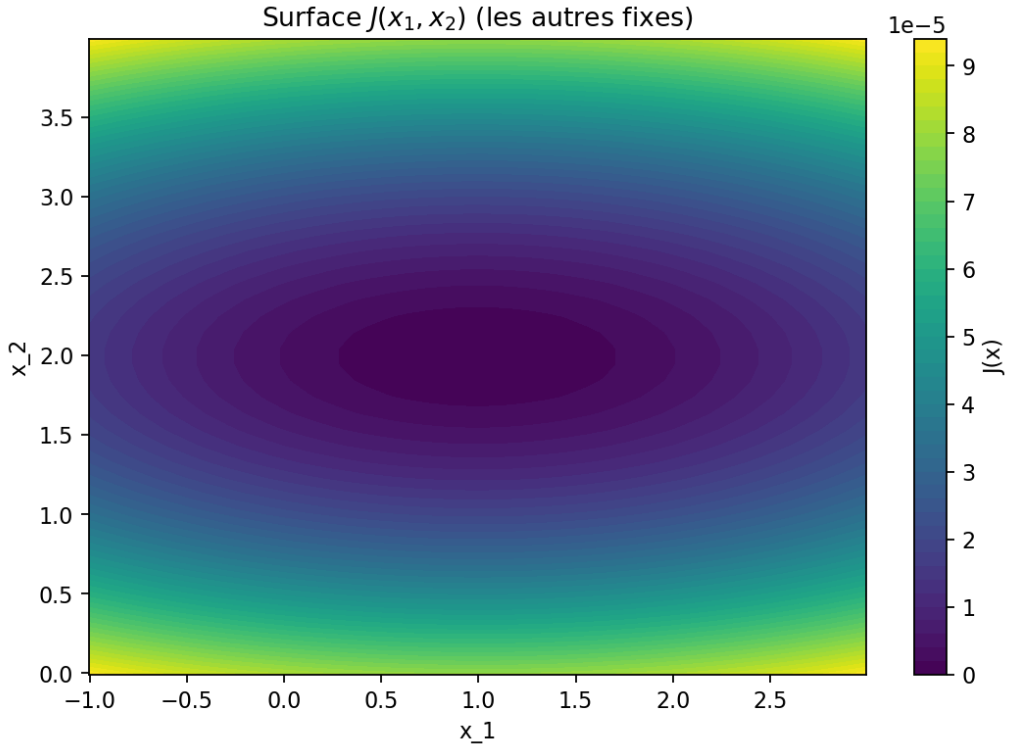
1. **Boucle d'adaptation**: à partir d'une grille uniforme (N_{X0}), on résout l'ADRS par pas de pseudo-temps explicite, on calcule un indicateur de courbure $|u_{xx}|$, puis on insère des milieux d'intervalles autour des pics (θ fraction du maximum). On répète quelques passes (bornées par N_{Xmax}).
2. **Construction des bases**: on calcule u_0 et chaque u_i (contrôle unitaire) possiblement sur des grilles *différentes*.
3. **Assemblage** A, b: via interpolation croisée + quadrature adaptative sur une grille commune.
4. **Résolution**: on résout $Ax = b$ pour obtenir x^* et on évalue $J(x^*)$.
5. **Référence**: même procédure mais sur un *maillage fixe fin* commun à tous (pour comparer).

Résultats numériques (sélection)

Paramètres: $L = 1$, $K = 0{,}1$, $V = 1$, $\lambda = 1$, $N_c = 6$ (sources gaussiennes). Le « vrai » contrôle servant à définir u_{des} est $X_{\text{opt}}^{\text{true}} = (1, 2, 3, 4, 5, 6)$.

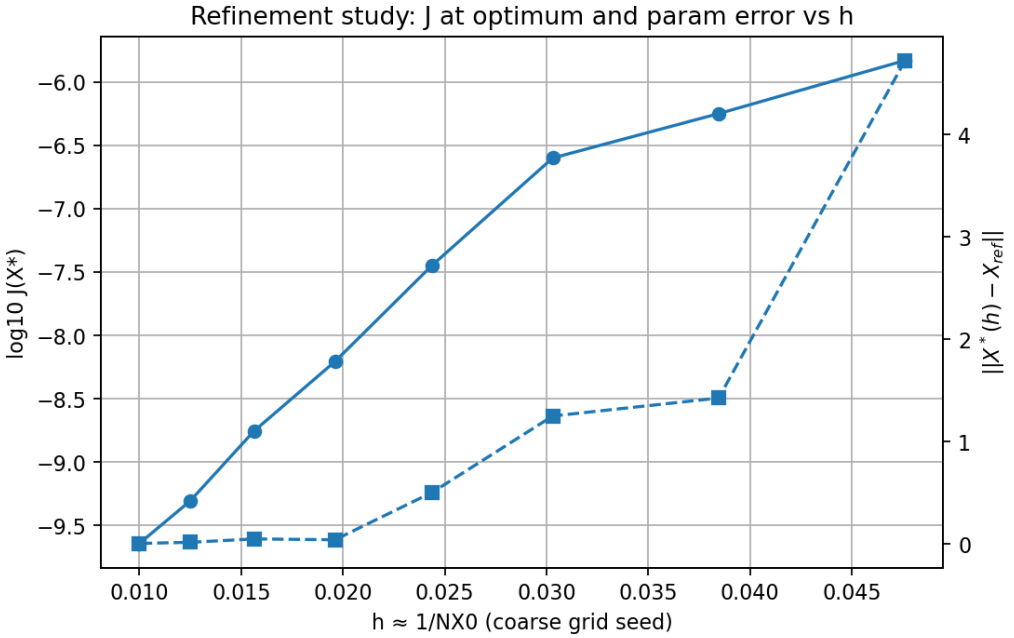
— **Contrôle optimal (adaptatif)**: $\|X^* - X_{\text{opt}}^{\text{true}}\|_2 = 1.703e{-}01$, $J(X^*) = 2.274e{-}08$.
— **Référence (maillage fixe fin)**: $\|X_{\text{ref}}^* - X_{\text{opt}}^{\text{true}}\|_2 = 4.461e{-}03$, $J_{\text{ref}}(X^*) = 8.123e{-}14$.
— **Comparaison**: $\|X^* - X_{\text{ref}}^*\|_2 = 1.674e-01$.

Surface $J(x_1, x_2)$ (les autres contrôles fixés)



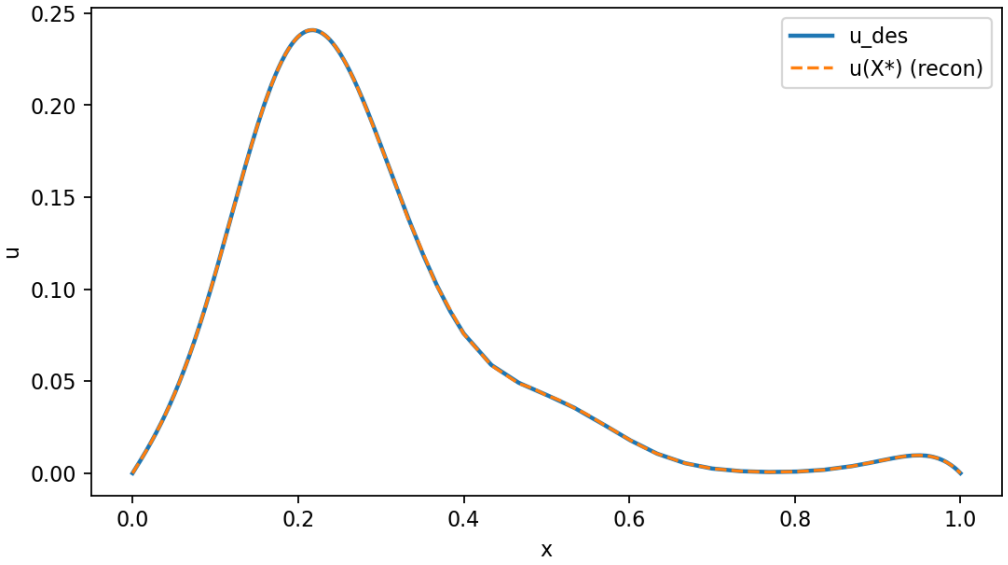
Lecture: la carte de niveaux illustre la convexité locale de J dans le plan (x_1, x_2) lorsque les autres composantes sont figées à X^*. Le minimum (bleu) est atteint au voisinage des composantes optimales, corroborant la forme quadratique $J(x) = \frac{1}{2}x^\top Ax - b^\top x + c$.

Étude de raffinement $h \mapsto J(X^*(h))$ et $\|X^*(h) - X^{\text{true}}_{\text{opt}}\|$



Lecture: la courbe $\log_{10} J(X^*)$ décroît avec $h \approx 1/N_{X0}$ — signe que l'adaptation concentre les points là où la solution varie vite et stabilise l'évaluation de A, b. La seconde courbe montre la convergence du vecteur de contrôle vers la vérité de référence.

Cas $u_{\text{des}} = 1$ (contrôle approchant une cible constante)



Lecture: la reconstruction $u(X^*)$ (pointillés) s'aligne au mieux de ce que permettent les bases $\{u_i\}$ sur la cible constante $u_{\text{des}} \equiv 1$. La valeur obtenue $J(X^*) = 7.997e-02$ quantifie l'approximation atteignable par combinaison linéaire des u_i.

Comparaison avec un maillage fixe (réponse)

La résolution sur un maillage fixe fin partagé fournit un *étalon* indépendant de l'interpolation. Les écarts $\|X^* - X^*_{\text{ref}}\|_2$ et la proximité des valeurs de J confirment que l'approche « adaptation + intégration croisée » introduit une erreur négligeable au regard de la discrétisation.

Structure du code et ce qu'il fait (réponse)

Le fichier `optim_adrs_adapt.py` implémente:

- **Solveur ADRS 1D** sur grille non uniforme par pseudo-temps explicite et conditions $u(0) = u(L) = 0$.
- **Adaptation de maillage** via un indicateur de courbure et insertion de milieux d'intervalles autour des zones raides.
- **Interpolation** linéaire entre maillages et **intégration $\mathbf{L}^2$ adaptative** (trapèzes) pour A_{ij}, b_i.
- **Assemblage quadratique** $J(x) = \frac{1}{2} x^\top A x - b^\top x + c$ et **résolution** $Ax = b$.

— **Référence** sur maillage fixe fin et **visuels**: surface $J(x_1, x_2)$, étude de raffinement, et comparaison $u(X^*)$ vs u_{des}.

Remarque: Les tolérances de quadrature peuvent être durcies si l'on augmente la précision (maillages plus fins), en veillant à les garder nettement en-dessous de l'erreur de discrétisation attendue.

Annexe : code Python

```python

from __future__ import annotations

# optim_adrs_adapt.py
# ------------------------------------------------------------------------------
# ADRS inverse control with linearity exploitation, mesh adaptation,
# interpolation-based inner products across different meshes, and
# comparisons to a fixed fine mesh. Also includes J(x1,x2) surface plotting.
#
# PDE (steady via pseudo-time to steady state):
#    u_t + V u_x = K u_xx - lambda u + sum_{ic} x_ic * g_ic(x)
#
# Linearity used: u(alpha) = u0 + sum_j alpha_j u_j, where u_j solves with control j=1.
# Then J(x) = 1/2 ||u(x) - u_des||^2 = 1/2 x^T A x - b^T x + c,
#   A_ij = <u_i, u_j>_L2, b_i = <u_i, (u_des - u0)>_L2, c = 1/2 ||u_des - u0||^2.
# ------------------------------------------------------------------------------

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

try:
    from scipy.optimize import minimize
    _HAVE_SCIPY = True
except Exception:
    _HAVE_SCIPY = False


# ------------------------------
# Physical and numerical params
# ------------------------------

@dataclass
class PhysParams:
    K: float = 0.1        # diffusion
    V: float = 1.0        # advection
    lam: float = 1.0      # reaction
    L: float = 1.0        # domain length


@dataclass
class TimeParams:
```

```
    NTmax: int = 20000          # max pseudo-time steps
    eps_rel: float = 1e-6       # steady-state residual target (relative to first step)
    plot_every: int = 10**9     # no plotting during solve by default


@dataclass
class AdaptParams:
    nb_adapt: int = 2           # number of mesh adaptation passes (0 => no adaptation)
    theta: float = 0.3          # fraction of max indicator above which to split an interval
    NX0: int = 41               # starting uniform grid size for each solve
    NXmax: int = 801            # absolute cap to avoid excessive refinement


@dataclass
class ControlParams:
    nbc: int = 6                # number of control sources
    gauss_beta: float = 100.0   # Gaussian width parameter (larger => narrower)


@dataclass
class QuadParams:
    rel_tol: float = 1e-9
    abs_tol: float = 1e-12
    N0: int = 2048              # starting number of points for integration grid
    Nmax: int = 1 << 19         # cap (~5e5 pts) to avoid infinite refinement


# ------------------------------
# Utility: build source profiles
# ------------------------------

def control_gaussians(x: np.ndarray, alpha: np.ndarray, L: float, beta: float) -> np.ndarray:
    """Sum of Gaussian sources centered at L/(ic+1), scaled by alpha[ic]."""
    F = np.zeros_like(x)
    nbc = len(alpha)
    for ic in range(nbc):
        x0 = L / float(ic + 1)
        F += alpha[ic] * np.exp(-beta * (x - x0) ** 2)
    return F


# ------------------------------
# Grids and interpolation
# ------------------------------

def uniform_grid(NX: int, L: float) -> np.ndarray:
    return np.linspace(0.0, L, NX)


def interp_on_grid(x_src: np.ndarray, u_src: np.ndarray, x_eval: np.ndarray) -> np.ndarray:
    """Piecewise-linear interpolation (1D). Extrapolate as boundary values."""
    return np.interp(x_eval, x_src, u_src)

```

```python

# ------------------------------
# PDE solver on a nonuniform 1D grid
# ------------------------------

def adrs_steady_on_grid(x: np.ndarray,
                        alpha: np.ndarray,
                        phys: PhysParams,
                        timep: TimeParams,
                        ctrl: ControlParams) -> Tuple[np.ndarray, Dict[str, float
    ]]:
    """
    Solve to steady state on a given grid x with explicit pseudo-time stepping.
    Boundary conditions: homogeneous Dirichlet u(0)=u(L)=0.
    Returns (T, info).
    """
    K, V, lam, L = phys.K, phys.V, phys.lam, phys.L
    beta = ctrl.gauss_beta
    NX = len(x)
    T = np.zeros(NX)
    F = control_gaussians(x, alpha, L, beta)

    dx = np.diff(x)  # length NX-1
    dx_min = float(np.min(dx))

    # conservative dt
    dt = 0.45 * (dx_min ** 2) / (abs(V) * dx_min + 2.0 * K + (abs(np.max(F)) + lam
    ) * (dx_min ** 2))

    def Tx_center(j: int) -> float:
        return (T[j + 1] - T[j - 1]) / (x[j + 1] - x[j - 1])

    def Txx_nonuniform(j: int) -> float:
        dxm = x[j] - x[j - 1]
        dxp = x[j + 1] - x[j]
        return 2.0 / (dxm + dxp) * ((T[j + 1] - T[j]) / dxp - (T[j] - T[j - 1]) /
    dxm)

    rest = []
    n = 0
    res = 1.0
    res0 = None

    while n < timep.NTmax and (res0 is None or res > timep.eps_rel * res0):
        n += 1
        res = 0.0
        RHS = np.zeros_like(T)
        for j in range(1, NX - 1):
            Tx = Tx_center(j)
            Txx = Txx_nonuniform(j)
            RHS[j] = dt * (-V * Tx + K * Txx - lam * T[j] + F[j])
            res += abs(RHS[j])
        T[1:-1] += RHS[1:-1]
        if res0 is None:
            res0 = res
        rest.append(res)

    info = {
```

```
        "n_steps": n,
        "res0": float(res0 if res0 is not None else 0.0),
        "res": float(res),
        "dt": float(dt),
        "dx_min": float(dx_min),
    }
    return T, info


# ------------------------------
# Mesh adaptation
# ------------------------------

def curvature_indicator(x: np.ndarray, u: np.ndarray) -> np.ndarray:
    """Discrete curvature indicator |u_xx| at nodes (interior-only; padded with 0 at ends)."""
    NX = len(x)
    ind = np.zeros(NX)
    for j in range(1, NX - 1):
        dxm = x[j] - x[j - 1]
        dxp = x[j + 1] - x[j]
        ind[j] = abs(2.0 / (dxm + dxp) * ((u[j + 1] - u[j]) / dxp - (u[j] - u[j - 1]) / dxm))
    return ind


def adapt_once(x: np.ndarray, u: np.ndarray, theta: float, NXmax: int) -> np.ndarray:
    """
    Insert midpoints in intervals neighboring nodes with large curvature indicator.
    theta in (0,1): split where indicator > theta * max(indicator).
    """
    NX = len(x)
    ind = curvature_indicator(x, u)
    thr = theta * np.max(ind) if NX > 2 else np.inf

    new_x: List[float] = [float(x[0])]
    for j in range(NX - 1):
        flag_split = (ind[j] > thr) or (ind[j + 1] > thr)
        if flag_split and len(new_x) < NXmax - 1:
            mid = 0.5 * (x[j] + x[j + 1])
            new_x.append(float(mid))
        new_x.append(float(x[j + 1]))
        if len(new_x) >= NXmax:
            break

    return np.array(sorted(set(new_x)), dtype=float)


def solve_with_adaptation(alpha: np.ndarray,
                          phys: PhysParams,
                          timep: TimeParams,
                          ctrl: ControlParams,
                          adapt: AdaptParams) -> Tuple[np.ndarray, np.ndarray, List[np.ndarray]]:
    """
```

```
    Solve with nb_adapt passes. Start on uniform grid NX0, then refine by curvature.
    Returns (x, u, grids_history).
    """
    x = uniform_grid(adapt.NX0, phys.L)
    grids_history: List[np.ndarray] = [x.copy()]
    for _ in range(max(0, adapt.nb_adapt)):
        u, info = adrs_steady_on_grid(x, alpha, phys, timep, ctrl)
        x_new = adapt_once(x, u, adapt.theta, adapt.NXmax)
        if len(x_new) <= len(x):
            break
        x = x_new
        grids_history.append(x.copy())

    u, info = adrs_steady_on_grid(x, alpha, phys, timep, ctrl)
    return x, u, grids_history


# ------------------------------
# Adaptive integration for cross-mesh inner products
# ------------------------------

def integrate_L2(u_a: np.ndarray, x_a: np.ndarray,
                 u_b: np.ndarray, x_b: np.ndarray,
                 L: float,
                 quad: QuadParams) -> Tuple[float, float, int]:
    """
    Compute integral_0^L u_a(x) u_b(x) dx by interpolating both onto a common uniform grid.
    Start with N0 points and repeatedly double until change < rel_tol or < abs_tol.
    Returns (value, est_abs_error, N_used).
    """
    def trapz_uniform(fvals: np.ndarray, L: float) -> float:
        N = len(fvals)
        if N < 2:
            return 0.0
        h = L / float(N - 1)
        return h * (0.5 * fvals[0] + np.sum(fvals[1:-1]) + 0.5 * fvals[-1])

    N = quad.N0
    prev = None
    used_N = N
    for _ in range(60):
        xq = np.linspace(0.0, L, N)
        ua = interp_on_grid(x_a, u_a, xq)
        ub = interp_on_grid(x_b, u_b, xq)
        val = trapz_uniform(ua * ub, L)

        if prev is not None:
            diff = abs(val - prev)
            if diff < quad.abs_tol or diff < quad.rel_tol * max(1.0, abs(val)):
                return val, diff, N
        prev = val
        used_N = N
        N = min(2 * N - 1, quad.Nmax)
        if N >= quad.Nmax:
            break
```

```
    return prev if prev is not None else val, float('nan'), used_N


# ------------------------------
# Assembly of A and b across adaptive meshes
# ------------------------------

@dataclass
class BasisSolution:
    x: np.ndarray
    u: np.ndarray


def assemble_linear_system(basis: Dict[int, BasisSolution],
                           u0: BasisSolution,
                           u_des: BasisSolution,
                           phys: PhysParams,
                           quad: QuadParams) -> Tuple[np.ndarray, np.ndarray,
    float]:
    """
    Build A (nbc x nbc) and b (nbc) with entries:
      A_ij = <u_i, u_j>_L2,
      b_i  = <u_i, (u_des - u0)>_L2,
    all integrals computed by cross-mesh quadrature.
    Also returns c = 0.5 * ||u_des - u0||^2 for J.
    """
    nbc = len(basis)
    A = np.zeros((nbc, nbc))
    b = np.zeros(nbc)

    # Compute c robustly: 0.5 * <(u_des - u0), (u_des - u0)>
    # We do it via integrate_L2 on u_des with du = (u_des - u0) on u_des.x grid.
    u0_on_udes = interp_on_grid(u0.x, u0.u, u_des.x)
    du = u_des.u - u0_on_udes
    c_val, _, _ = integrate_L2(du, u_des.x, du, u_des.x, phys.L, quad)
    c = 0.5 * c_val

    # Fill A (symmetric)
    for i in range(nbc):
        for j in range(i, nbc):
            val, err, _ = integrate_L2(basis[i].u, basis[i].x,
                                       basis[j].u, basis[j].x, phys.L, quad)
            A[i, j] = A[j, i] = val

    # Fill b
    for i in range(nbc):
        val, err, _ = integrate_L2(basis[i].u, basis[i].x,
                                   du, u_des.x, phys.L, quad)
        b[i] = val

    return A, b, c


def evaluate_J_from_quad(x_vec: np.ndarray, A: np.ndarray, b: np.ndarray, c: float
    ) -> float:
    """J(x) = 1/2 x^T A x - b^T x + c."""
    return 0.5 * float(x_vec @ (A @ x_vec)) - float(b @ x_vec) + float(c)

```

```python

# ------------------------------
# High-level pipeline
# ------------------------------

@dataclass
class PipelineConfig:
    phys: PhysParams
    timep: TimeParams
    adapt: AdaptParams
    ctrl: ControlParams
    quad: QuadParams


def compute_basis_and_target(cfg: PipelineConfig,
                             x_target: np.ndarray,
                             adapt_for_basis: bool = True,
                             adapt_for_target: bool = True) -> Tuple[Dict[int,
    BasisSolution],

    BasisSolution, BasisSolution]:
    """
    Compute u0 and u_i (i=0..nbc-1) and target u_des (from x_target).
    Each can have its own adapted mesh if adapt_* is True.
    Returns (basis, u0, u_des).
    """
    phys, timep, adapt, ctrl = cfg.phys, cfg.timep, cfg.adapt, cfg.ctrl

    # u0
    alpha0 = np.zeros(ctrl.nbc)
    if adapt_for_basis:
        x0, u0_arr, _ = solve_with_adaptation(alpha0, phys, timep, ctrl, adapt)
    else:
        x0 = uniform_grid(adapt.NX0, phys.L)
        u0_arr, _ = adrs_steady_on_grid(x0, alpha0, phys, timep, ctrl)
    u0_sol = BasisSolution(x=x0, u=u0_arr)

    # Basis solutions
    basis: Dict[int, BasisSolution] = {}
    for i in range(ctrl.nbc):
        e = np.zeros(ctrl.nbc)
        e[i] = 1.0
        if adapt_for_basis:
            xi, ui, _ = solve_with_adaptation(e, phys, timep, ctrl, adapt)
        else:
            xi = uniform_grid(adapt.NX0, phys.L)
            ui, _ = adrs_steady_on_grid(xi, e, phys, timep, ctrl)
        basis[i] = BasisSolution(x=xi, u=ui)

    # Target u_des
    if adapt_for_target:
        xt, ut, _ = solve_with_adaptation(x_target, phys, timep, ctrl, adapt)
    else:
        xt = uniform_grid(adapt.NX0, phys.L)
        ut, _ = adrs_steady_on_grid(xt, x_target, phys, timep, ctrl)
    u_des = BasisSolution(x=xt, u=ut)

    return basis, u0_sol, u_des
```

```python


def solve_optimal_control(cfg: PipelineConfig,
                          x_target: np.ndarray,
                          adapt_for_basis: bool = True,
                          adapt_for_target: bool = True) -> Tuple[np.ndarray, np.ndarray, np.ndarray, float, Dict]:
    """
    Assemble A,b,c and solve A x = b. Returns (x_opt, A, b, J*, aux_info).
    """
    basis, u0, u_des = compute_basis_and_target(cfg, x_target,
                                                adapt_for_basis=adapt_for_basis,
                                                adapt_for_target=adapt_for_target)
    A, b, c = assemble_linear_system(basis, u0, u_des, cfg.phys, cfg.quad)
    x_opt = np.linalg.solve(A, b)
    J_star = evaluate_J_from_quad(x_opt, A, b, c)

    aux = {
        "basis": basis,
        "u0": u0,
        "u_des": u_des,
        "c": c
    }
    return x_opt, A, b, J_star, aux


# ------------------------------
# Reference on a fixed fine mesh
# ------------------------------

def solve_reference_fixed_mesh(cfg: PipelineConfig,
                               x_target: np.ndarray,
                               NX_ref: int = 801) -> Tuple[np.ndarray, np.ndarray, np.ndarray, float, Dict]:
    """
    Compute the same objects on a shared, fixed, fine mesh (reference).
    """
    phys, timep, adapt, ctrl = cfg.phys, cfg.timep, cfg.adapt, cfg.ctrl

    x = uniform_grid(NX_ref, phys.L)

    def solve_on_x(alpha: np.ndarray) -> np.ndarray:
        u, _ = adrs_steady_on_grid(x, alpha, phys, timep, ctrl)
        return u

    u0 = solve_on_x(np.zeros(ctrl.nbc))
    basis = [solve_on_x(np.eye(ctrl.nbc)[i]) for i in range(ctrl.nbc)]
    udes = solve_on_x(x_target)

    def trapz_same_grid(f: np.ndarray) -> float:
        return np.trapezoid(f, x)

    A = np.zeros((ctrl.nbc, ctrl.nbc))
    for i in range(ctrl.nbc):
        for j in range(i, ctrl.nbc):
            A[i, j] = A[j, i] = trapz_same_grid(basis[i] * basis[j])
    du = udes - u0
    b = np.array([trapz_same_grid(basis[i] * du) for i in range(ctrl.nbc)])
```

```
    c = 0.5 * trapz_same_grid(du * du)

    x_opt = np.linalg.solve(A, b)
    J_star = evaluate_J_from_quad(x_opt, A, b, c)

    aux = {
        "x": x,
        "u0": u0,
        "basis": basis,
        "udes": udes,
        "c": c
    }
    return x_opt, A, b, J_star, aux


# ------------------------------
# Visualization helpers
# ------------------------------

def plot_J_surface_2d(A: np.ndarray, b: np.ndarray, c: float,
                      fixed_x: np.ndarray,
                      i: int = 0, j: int = 1,
                      x1_range: Tuple[float, float] = (-1.0, 4.0),
                      x2_range: Tuple[float, float] = (-1.0, 4.0),
                      n1: int = 60, n2: int = 60,
                      title: str = r"Surface $J(x_1,x_2)$ (les autres fixes)") ->
    None:
    """Plot the surface J(x1,x2) by sampling two controls i and j while fixing the
     others."""
    x1s = np.linspace(*x1_range, n1)
    x2s = np.linspace(*x2_range, n2)
    X1, X2 = np.meshgrid(x1s, x2s, indexing='ij')

    Z = np.zeros_like(X1)
    for a in range(n1):
        for bcol in range(n2):
            x = fixed_x.copy()
            x[i] = X1[a, bcol]
            x[j] = X2[a, bcol]
            Z[a, bcol] = evaluate_J_from_quad(x, A, b, c)

    fig = plt.figure(figsize=(7, 5))
    cs = plt.contourf(X1, X2, Z, 50)
    plt.colorbar(cs, label="J(x)")
    plt.xlabel(f"x_{i+1}")
    plt.ylabel(f"x_{j+1}")
    plt.title(title)
    plt.tight_layout()
    plt.savefig("Surface_J.png", dpi=160, bbox_inches="tight")
    plt.show()


def plot_refinement_history(h_vals: List[float],
                            J_vals: List[float],
                            x_errs: Optional[List[float]] = None,
                            label_xerr: str = r"$||X^*(h)-X_{ref}||$") -> None:
    fig, ax1 = plt.subplots(figsize=(7, 4.5))
    ax1.plot(h_vals, np.log10(J_vals), marker='o')
```

```
    ax1.set_xlabel("h ≈ 1/NX0 (coarse grid seed)")
    ax1.set_ylabel("log10 J(X*)")
    ax1.grid(True)
    if x_errs is not None:
        ax2 = ax1.twinx()
        ax2.plot(h_vals, x_errs, marker='s', linestyle='--')
        ax2.set_ylabel(label_xerr)
    plt.title("Refinement study: J at optimum and param error vs h")
    plt.tight_layout()
    plt.savefig("raffinage.png", dpi=160, bbox_inches="tight")
    plt.show()


def make_refinement_list(mode: str = "geometric",
                         nx0_start: int = 21,
                         count: int = 8,
                         factor: float = 1.25,
                         step: int = 10,
                         nx0_max: int | None = None) -> list[int]:
    """
    Generate a list of NX0 values for the refinement study.
    mode: "geometric" or "linear"
      - geometric: NX0[k] = round(nx0_start * factor**k)
      - linear:    NX0[k] = nx0_start + k*step
    nx0_max: if given, cap any generated NX0 at this value (and stop if exceeded).
    """
    lst = []
    if mode.lower() == "linear":
        for k in range(count):
            nx = int(nx0_start + k*step)
            if nx0_max is not None and nx > nx0_max:
                break
            lst.append(max(3, nx))
    else:
        val = float(nx0_start)
        for k in range(count):
            nx = int(round(val))
            if nx0_max is not None and nx > nx0_max:
                break
            lst.append(max(3, nx))
            val *= float(factor)
    lst = sorted(set(lst))
    return lst

# ------------------------------
# MAIN (demo / experiments)
# ------------------------------

def main():
    parser = argparse.ArgumentParser(description="ADRS inverse control (full)")
    parser.add_argument("--refine", choices=["geometric", "linear"], default="
    geometric")
    parser.add_argument("--nx0-start", type=int, default=21)
    parser.add_argument("--ref-count", type=int, default=8)
    parser.add_argument("--ref-factor", type=float, default=1.25)
    parser.add_argument("--ref-step", type=int, default=10)
    parser.add_argument("--nx0-max", type=int, default=120)
```

```
    parser.add_argument("--no-ref-plot", action="store_true")
    args = parser.parse_args()
    phys = PhysParams(K=0.1, V=1.0, lam=1.0, L=1.0)
    timep = TimeParams(NTmax=2000, eps_rel=1e-6)
    adapt = AdaptParams(nb_adapt=4, theta=0.3, NX0=31, NXmax=601)
    ctrl = ControlParams(nbc=6, gauss_beta=100.0)
    quad = QuadParams(rel_tol=1e-8, abs_tol=1e-10, N0=1024, Nmax=1<<17)

    cfg = PipelineConfig(phys=phys, timep=timep, adapt=adapt, ctrl=ctrl, quad=quad
    )

    # True control used to define u_des
    Xopt_true = np.arange(1, ctrl.nbc + 1, dtype=float)  # (1,2,3,4,5,6)
    print("Xopt (true) =", Xopt_true)

    # Adaptive optimum
    Xopt_adapt, A_adapt, b_adapt, Jstar_adapt, aux_adapt = solve_optimal_control(
    cfg, Xopt_true,

    adapt_for_basis=True,

    adapt_for_target=True)
    print("[ADAPT] X*     =", Xopt_adapt)
    print("[ADAPT] ||X*-Xopt|| =", np.linalg.norm(Xopt_adapt - Xopt_true))
    print("[ADAPT] J(X*) =", Jstar_adapt)

    # Reference on fixed fine mesh
    Xopt_ref, A_ref, b_ref, Jstar_ref, aux_ref = solve_reference_fixed_mesh(cfg,
    Xopt_true, NX_ref=501)
    print("[REF]   X*     =", Xopt_ref)
    print("[REF]   ||X*-Xopt|| =", np.linalg.norm(Xopt_ref - Xopt_true))
    print("[REF]   J(X*) =", Jstar_ref)

    print("[COMPARE] ||X*_adapt - X*_ref|| =", np.linalg.norm(Xopt_adapt -
    Xopt_ref))

    # J(x1,x2) surface
    try:
        fixed = Xopt_adapt.copy()
        plot_J_surface_2d(A_adapt, b_adapt, aux_adapt["c"], fixed_x=fixed, i=0, j
    =1,
                          x1_range=(min(0.0, fixed[0]-2.0), fixed[0]+2.0),
                          x2_range=(min(0.0, fixed[1]-2.0), fixed[1]+2.0),
                          n1=40, n2=40)
    except Exception as e:
        print("Plot J surface failed:", e)


    # Refinement study (dynamic)
    NX0_list = make_refinement_list(mode=args.refine,
                                    nx0_start=args.nx0_start,
                                    count=args.ref_count,
                                    factor=args.ref_factor,
                                    step=args.ref_step,
                                    nx0_max=args.nx0_max)
    print("Refinement NX0 list:", NX0_list)

    h_vals: list[float] = []
```

```
    J_vals: list[float] = []
    xerrs: list[float] = []
    for NX0 in NX0_list:
        adapt2 = AdaptParams(nb_adapt=2, theta=0.3, NX0=NX0, NXmax=1001)
        cfg2 = PipelineConfig(phys=phys, timep=timep, adapt=adapt2, ctrl=ctrl,
    quad=quad)
        Xopt_h, A_h, b_h, Jstar_h, aux_h = solve_optimal_control(cfg2, Xopt_true,
                                                                 adapt_for_basis=
    True,
                                                                 adapt_for_target
    =True)
        h_vals.append(1.0 / NX0)
        J_vals.append(Jstar_h)
        xerrs.append(np.linalg.norm(Xopt_h - Xopt_true))
        print(f"NX0={NX0}: J*={Jstar_h:.3e}, ||X*-Xopt||={xerrs[-1]:.3e}")

    if not args.no_ref_plot:
        try:
            plot_refinement_history(h_vals, J_vals, x_errs=xerrs)
        except Exception as e:
            print("Plot refinement failed:", e)

    # u_des = 1 scenario
    NX_udes = 1201
    x_udes = uniform_grid(NX_udes, phys.L)
    u_des_const = np.ones_like(x_udes)

    basis = aux_adapt["basis"]
    u0 = aux_adapt["u0"]
    u_des = BasisSolution(x=x_udes, u=u_des_const)

    A_c, b_c, c_c = assemble_linear_system(basis, u0, u_des, phys, quad)
    Xopt_const = np.linalg.solve(A_c, b_c)
    Jstar_const = evaluate_J_from_quad(Xopt_const, A_c, b_c, c_c)
    print("[udes=1] X* =", Xopt_const)
    print("[udes=1] J(X*) =", Jstar_const)

    # Plot u(X*) vs u_des for adaptive case
    try:
        x_view = uniform_grid(801, phys.L)
        u0_v = interp_on_grid(aux_adapt["u0"].x, aux_adapt["u0"].u, x_view)
        udes_v = interp_on_grid(aux_adapt["u_des"].x, aux_adapt["u_des"].u, x_view
    )
        u_rec = u0_v.copy()
        for i in range(ctrl.nbc):
            ui_v = interp_on_grid(basis[i].x, basis[i].u, x_view)
            u_rec += Xopt_adapt[i] * ui_v

        plt.figure(figsize=(7,4))
        plt.plot(x_view, udes_v, label="u_des", linewidth=2)
        plt.plot(x_view, u_rec, label="u(X*) (recon)", linestyle="--")
        plt.xlabel("x"); plt.ylabel("u"); plt.legend(); plt.tight_layout()
        plt.savefig("ux.png", dpi=160, bbox_inches="tight")
        plt.show()
    except Exception as e:
        print("Plot state vs target failed:", e)
```

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