# Estimation a posteriori

*rapport séance 4*

Victor Baleux

29 septembre 2025

# Table des matières

# Chapitre 1

# Rapport de TP : adaptation de maillage et métriques

## 1.1 Lecture de §18.4.2 — contrôle local de métrique

Le *contrôle local de métrique* remplace la distance euclidienne par une métrique $\lambda(x)$ pour dimensionner localement la taille des éléments. En 1D, en s'appuyant sur l'erreur d'interpolation P1, on adopte

$$\lambda(x) = \min\Big( \max\Big(\tfrac{1}{\varepsilon}|u''(x)|, \tfrac{1}{h_{\min}^2}\Big), \tfrac{1}{h_{\max}^2}\Big), \tag{1.1}$$

et $h(x) \approx 1/\sqrt{\lambda(x)}$ pour l'adaptation.

## 1.2 Étapes de l'algorithme et correspondance dans le code

(1) initialiser le maillage ; (2) résoudre le problème ; (3) calculer l'indicateur (ici $u''$) ; (4) construire la métrique et adapter ; (5) tester l'arrêt sinon boucler.

- Maillage & boucle d'adaptation : `solve_adrs_adapt`.
- Résolution temporelle : `solve_adrs_on_mesh`.
- Indicateur $T_{xx} \approx u''$ : `compute_Txx`.
- Métrique $\lambda$ : `metric_from_Txx`.
- Reconstruction du maillage : `adapt_mesh_from_metric`.

## 1.3 Lois de métrique

Trois variantes (`metric_law`) :

**Loi 1** finale instantanée : $\lambda \propto |T_{xx}|/\varepsilon$ (au temps final).

**Loi 2** moyenne temporelle : moyenne de $\lambda^{\text{inst}}$ sur le temps.

**Loi 3** RMS temporelle : $\lambda \propto \text{RMS}(T_{xx})/\varepsilon$.

## 1.4 Implémentation de la loi 3

Accumulation de $T_{xx}^2$ dans `solve_adrs_on_mesh`, puis $T_{xx}^{\mathrm{RMS}} = \sqrt{\texttt{sum\_Txx2}/N_t}$ utilisé par `metric_from_Txx` ; conversion $h \approx 1/\sqrt{\lambda}$ avant la reconstruction.

## 1.5 Résultats numériques : $NX(\varepsilon)$

### 1.5.1 Comparaison globale des lois

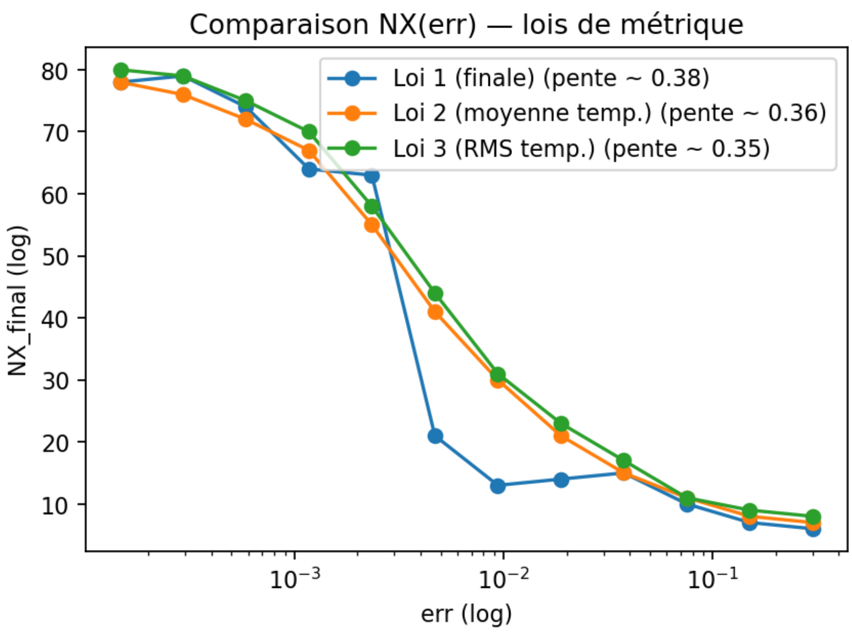


FIGURE 1.1 – Comparaison NX(err) — lois de métrique.

### 1.5.2 Valeurs demandées $\varepsilon \in \{0.04, 0.02, 0.01, 0.005, 0.0025\}$

On observe une loi de puissance $NX \sim \varepsilon^{-p}$ ; les pentes (log–log) estimées sont indiquées dans la légende de la figure et rappelées dans le texte.

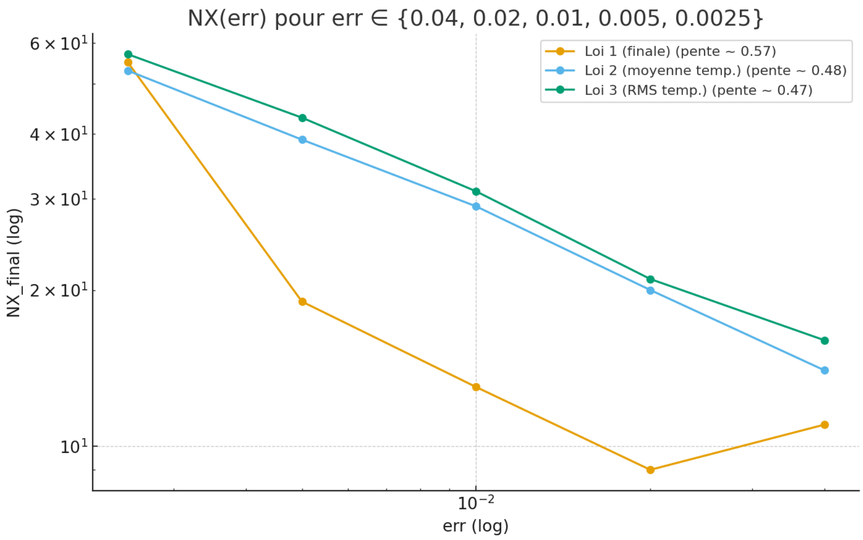


FIGURE 1.2 – $NX(\varepsilon)$ pour les valeurs imposées et comparaison des lois de métrique.

## 1.6 Évolution de $NX$ avec $\varepsilon$

Quand $\varepsilon$ diminue, $\lambda$ augmente et le maillage se raffine : $NX$ croît quasi-polynomialement. La loi 3 (RMS) est souvent la plus robuste car moins sensible aux pics transitoires.

## 1.7 Critères d'arrêt et critère mixte

Dans `solve_adrs_adapt`, on s'arrête uniquement si *deux* conditions sont satisfaites : (i) variation de $NX$ négligeable (`NX_tol`) *et* (ii) erreur $L^2$ sous le seuil `L2_target`.

## 1.8 Contraction sur maillage de fond

On interpole chaque solution sur un maillage background fin et on suit $I_k = \|T_{\mathrm{bg}}^{(k)} - T_{\mathrm{bg}}^{(k-1)}\|_{L^2}$. Une décroissance géométrique indique la contraction.

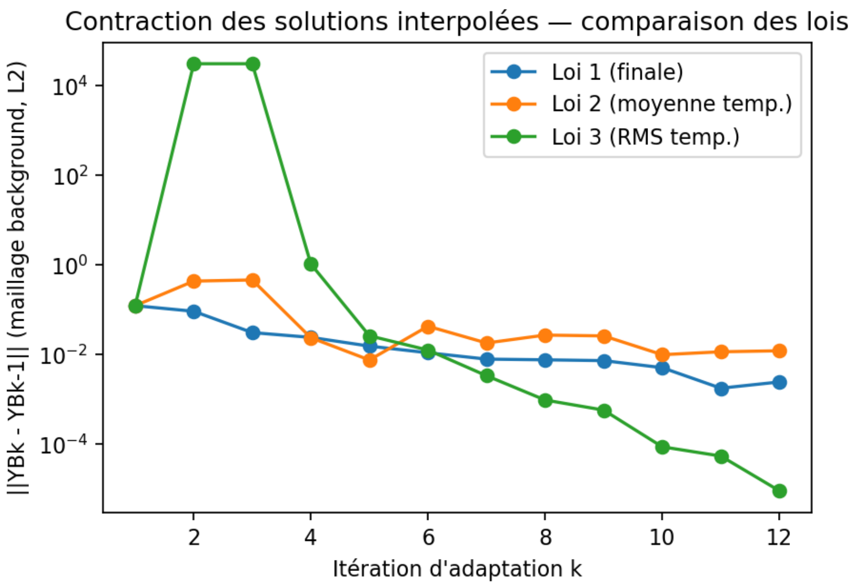


FIGURE 1.3 – Contraction des solutions interpolées — comparaison des lois.

# Annexe : code Python

```
# -*- coding: utf-8 -*-
import math
import numpy as np
import matplotlib.pyplot as plt
from dataclasses import dataclass
import pandas as pd
from typing import Tuple, List, Dict
from pathlib import Path

def interp_piecewise_linear(x_src: np.ndarray, y_src: np.ndarray, x_dst: np.
    ndarray) -> np.ndarray:
    y = np.zeros_like(x_dst, dtype=float)
    j = 0
    n = len(x_src)
    for i, xd in enumerate(x_dst):
        if xd <= x_src[0]:
            y[i] = y_src[0]; continue
        if xd >= x_src[-1]:
            y[i] = y_src[-1]; continue
        while j < n-2 and xd > x_src[j+1]:
            j += 1
        x0, x1 = x_src[j], x_src[j+1]
        y0, y1 = y_src[j], y_src[j+1]
        t = (xd - x0) / (x1 - x0)
        y[i] = (1.0 - t) * y0 + t * y1
    return y

@dataclass
class PhysParams:
    K: float = 0.01
    V: float = 1.0
    lamda: float = 1.0
    xmin: float = 0.0
```

```python
    xmax: float = 1.0
    Time: float = 2.0
    freq: float = 2*math.pi*3

@dataclass
class AdaptParams:
    hmin: float = 0.0125
    hmax: float = 0.25
    err: float = 0.005
    NX_init: int = 10
    NT_max: int = 600
    niter_refinement: int = 12
    NX_tol: int = 1
    L2_target: float = None
    dt_safety: float = 0.15
    metric_law: str = "loi3"

def exact_Tex(x: np.ndarray, phys: PhysParams) -> np.ndarray:
    return np.exp(-20.0*(x - 0.5*(phys.xmax + phys.xmin))**2)

def build_source_terms(x: np.ndarray, phys: PhysParams) -> Tuple[np.ndarray, float]:
    NX = len(x)
    Tex_arr = exact_Tex(x, phys)
    F = np.zeros(NX)
    dt = 1.0e30
    for j in range(1, NX-1):
        Tx = (Tex_arr[j+1] - Tex_arr[j-1]) / (x[j+1] - x[j-1])
        Txip1 = (Tex_arr[j+1] - Tex_arr[j]) / (x[j+1] - x[j])
        Txim1 = (Tex_arr[j] - Tex_arr[j-1]) / (x[j] - x[j-1])
        Txx = (Txip1 - Txim1) / (0.5*(x[j+1] + x[j]) - 0.5*(x[j] + x[j-1]))
        F[j] = phys.V*Tx - phys.K*Txx + phys.lamda*Tex_arr[j]
        local_h = (x[j+1] - x[j-1])
        denom = (abs(phys.V)*local_h + 4.0*phys.K + abs(F[j])*(local_h**2))
        if denom > 0.0:
            dt = min(dt, 0.25 * (local_h**2) / denom)
    return F, dt

def compute_error_L2_H1(x: np.ndarray, T: np.ndarray, phys: PhysParams) -> Tuple[float, float]:
    NX = len(x)
    Tex_arr = exact_Tex(x, phys)
    errL2 = 0.0
    errH1 = 0.0
    for j in range(1, NX-1):
        hx = 0.5*(x[j+1] + x[j]) - 0.5*(x[j] + x[j-1])
        Texx = (Tex_arr[j+1] - Tex_arr[j-1]) / (x[j+1] - x[j-1])
        Tx = (T[j+1] - T[j-1]) / (x[j+1] - x[j-1])
        errL2 += hx * (T[j] - Tex_arr[j])**2
        errH1 += hx * (Tx - Texx)**2
    return errL2, errL2 + errH1

def compute_Txx(T: np.ndarray, x: np.ndarray) -> np.ndarray:
    NX = len(x)
    Txx = np.zeros(NX)
    for j in range(1, NX-1):
        Txip1 = (T[j+1] - T[j]) / (x[j+1] - x[j])
        Txim1 = (T[j] - T[j-1]) / (x[j] - x[j-1])
```

```python
        hmid = (0.5*(x[j+1] + x[j]) - 0.5*(x[j] + x[j-1]))
        Txx[j] = (Txip1 - Txim1) / hmid
    Txx[0] = Txx[1]
    Txx[-1] = Txx[-2]
    return Txx

def metric_from_Txx(Txx: np.ndarray, ap: AdaptParams, accum_mode: str, n_time: int
    ) -> np.ndarray:
    hmin2 = 1.0 / (ap.hmin**2)
    hmax2 = 1.0 / (ap.hmax**2)
    if ap.metric_law == "loi1":
        raw = np.abs(Txx) / ap.err
        lam = np.minimum(np.maximum(raw, hmax2), hmin2)
    elif ap.metric_law == "loi2":
        lam = Txx.copy()
    elif ap.metric_law == "loi3":
        raw = Txx / ap.err
        lam = np.minimum(np.maximum(raw, hmax2), hmin2)
    else:
        raise ValueError("metric_law inconnu.")
    return lam

def adapt_mesh_from_metric(x: np.ndarray, T: np.ndarray, lam: np.ndarray, ap:
    AdaptParams) -> Tuple[np.ndarray, np.ndarray]:
    NX = len(x)
    h_loc = np.sqrt(1.0 / np.maximum(lam, 1e-16))
    h_loc = np.clip(h_loc, ap.hmin, ap.hmax)
    x_new = [x[0]]
    T_new = [T[0]]
    while x_new[-1] < x[-1] - ap.hmin:
        xi = x_new[-1]
        j = np.searchsorted(x, xi) - 1
        if j < 0: j = 0
        if j >= NX-1: j = NX-2
        hll = (h_loc[j]*(x[j+1]-xi) + h_loc[j+1]*(xi - x[j])) / (x[j+1]-x[j])
        hll = np.clip(hll, ap.hmin, ap.hmax)
        xn = min(x[-1], xi + hll)
        x_new.append(xn)
        t_interp = interp_piecewise_linear(x, T, np.array([xn]))[0]
        T_new.append(t_interp)
    return np.array(x_new), np.array(T_new)

def solve_adrs_on_mesh(x: np.ndarray, phys: PhysParams, ap: AdaptParams,
    collect_metric_stats: Dict) -> Tuple[np.ndarray, Dict]:
    NX = len(x)
    T = np.zeros(NX)
    Tex_arr = exact_Tex(x, phys)
    F, dt_base = build_source_terms(x, phys)
    dt = dt_base
    t = 0.0
    n = 0
    while n < ap.NT_max and t < phys.Time:
        n += 1
        dt = min(dt, phys.Time - t)
        t += dt
        RHS = np.zeros(NX)
        for j in range(1, NX-1):
```

```
            visnum = 0.25*(0.5*(x[j+1] + x[j]) - 0.5*(x[j] + x[j-1])) * abs(phys.V)
            xnu = phys.K + visnum
            Tx = (T[j+1] - T[j-1]) / (x[j+1] - x[j-1])
            Txip1 = (T[j+1] - T[j]) / (x[j+1] - x[j])
            Txim1 = (T[j] - T[j-1]) / (x[j] - x[j-1])
            Txx = (Txip1 - Txim1) / (0.5*(x[j+1] + x[j]) - 0.5*(x[j] + x[j-1]))
            src = F[j]*np.sin(phys.freq*t) + Tex_arr[j]*np.cos(phys.freq*t)*phys.freq
            RHS[j] = dt * (-phys.V*Tx + xnu*Txx - phys.lamda*T[j] + src)
        T[1:-1] += RHS[1:-1]
        T[0] = 0.0
        T[-1] = 2.0*T[-2] - T[-3]
        if ap.metric_law == "loi2":
            Txx_arr = compute_Txx(T, x)
            raw = np.abs(Txx_arr) / ap.err
            hmin2 = 1.0/(ap.hmin**2)
            hmax2 = 1.0/(ap.hmax**2)
            lam_inst = np.minimum(np.maximum(raw, hmax2), hmin2)
            collect_metric_stats["sum_lambda"] += lam_inst
            collect_metric_stats["count"] += 1
        elif ap.metric_law == "loi3":
            Txx_arr = compute_Txx(T, x)
            collect_metric_stats["sum_Txx2"] += (Txx_arr**2)
            collect_metric_stats["count"] += 1
    return T, collect_metric_stats

def one_adaptation_cycle(x: np.ndarray, T: np.ndarray, phys: PhysParams, ap: AdaptParams) -> Tuple[np.ndarray, np.ndarray, float, float, np.ndarray]:
    if ap.metric_law == "loi2":
        acc = {"sum_lambda": np.zeros_like(x), "count": 0}
    elif ap.metric_law == "loi3":
        acc = {"sum_Txx2": np.zeros_like(x), "count": 0}
    else:
        acc = {}
    T_final, acc = solve_adrs_on_mesh(x, phys, ap, acc)
    if ap.metric_law == "loi1":
        Txx = compute_Txx(T_final, x)
        lam = metric_from_Txx(Txx, ap, accum_mode="final", n_time=1)
    elif ap.metric_law == "loi2":
        count = max(1, acc["count"])
        lam_time_mean = acc["sum_lambda"] / float(count)
        lam = metric_from_Txx(lam_time_mean, ap, accum_mode="mean", n_time=count)
    elif ap.metric_law == "loi3":
        count = max(1, acc["count"])
        Txx_rms = np.sqrt(acc["sum_Txx2"] / float(count))
        lam = metric_from_Txx(Txx_rms, ap, accum_mode="rms", n_time=count)
    errL2, errH1 = compute_error_L2_H1(x, T_final, phys)
    x_new, T_new = adapt_mesh_from_metric(x, T_final, lam, ap)
    return x_new, T_new, errL2, errH1, lam

def solve_adrs_adapt(phys: PhysParams, ap: AdaptParams, background_N: int = 2001, return_history: bool = False):
    x = np.linspace(phys.xmin, phys.xmax, ap.NX_init)
    T = np.zeros_like(x)
    NX_prev = len(x)
    x_bg = np.linspace(phys.xmin, phys.xmax, background_N)
    T_bg_prev = interp_piecewise_linear(x, T, x_bg)
```

```
    hist = {"NX": [len(x)], "errL2": [], "errH1": [], "Ik": []}
    for k in range(ap.niter_refinement):
        x, T, eL2, eH1, lam = one_adaptation_cycle(x, T, phys, ap)
        hist["NX"].append(len(x))
        hist["errL2"].append(eL2)
        hist["errH1"].append(eH1)
        T_bg = interp_piecewise_linear(x, T, x_bg)
        Ik = np.sqrt(np.trapezoid((T_bg - T_bg_prev)**2, x_bg))
        hist["Ik"].append(Ik)
        T_bg_prev = T_bg.copy()
        stop_NX = abs(len(x) - NX_prev) <= ap.NX_tol
        stop_err = (ap.L2_target is not None) and (eL2 <= ap.L2_target)
        NX_prev = len(x)
        if stop_NX and stop_err:
            break
    if return_history:
        hist["x_bg"] = x_bg
        hist["T_bg"] = T_bg
        return x, T, hist
    else:
        return x, T

def study_NX_vs_err(err_list: List[float], phys: PhysParams, ap_template:
    AdaptParams) -> pd.DataFrame:
    rows = []
    for eps in err_list:
        ap = AdaptParams(
            hmin=ap_template.hmin, hmax=ap_template.hmax,
            err=eps, NX_init=ap_template.NX_init, NT_max=ap_template.NT_max,
            niter_refinement=ap_template.niter_refinement, NX_tol=ap_template.
    NX_tol,
            L2_target=None, dt_safety=ap_template.dt_safety, metric_law=
    ap_template.metric_law
        )
        x, T, hist = solve_adrs_adapt(phys, ap, return_history=True)
        rows.append({"err": eps, "NX_final": len(x)})
    df = pd.DataFrame(rows).sort_values("err", ascending=False).reset_index(drop=
    True)
    return df

def run_comparisons(output_dir: Path = Path(".")):
    phys = PhysParams()
    n_err = 12          # nombre de valeurs souhaité
    err_base = 0.3  # valeur de départ
    err_ratio = 0.5     # facteur de décroissance
    err_values = [err_base * (err_ratio ** k) for k in range(n_err)]
    laws = ["loi1", "loi2", "loi3"]
    labels_map = {"loi1": "Loi 1 (finale)", "loi2": "Loi 2 (moyenne temp.)", "loi3
    ": "Loi 3 (RMS temp.)"}
    ap_template = AdaptParams(
        hmin=0.0125, hmax=0.25, err=0.005, NX_init=12,
        NT_max=600, niter_refinement=8, NX_tol=1,
        L2_target=None, metric_law="loi3"
    )
    df_list = []
    slopes_info = {}
    for law in laws:
        ap_template.metric_law = law
```

```
        df = study_NX_vs_err(err_values, phys, ap_template)
        df.rename(columns={"NX_final": f"NX_{law}"}, inplace=True)
        df_list.append(df)
        log_err = np.log(np.array(df["err"]))
        log_NX = np.log(np.array(df[f"NX_{law}"]))
        A = np.vstack([log_err, np.ones_like(log_err)]).T
        slope, intercept = np.linalg.lstsq(A, log_NX, rcond=None)[0]
        slopes_info[law] = slope
    df_merged = df_list[0]
    for d in df_list[1:]:
        df_merged = df_merged.merge(d, on="err", how="inner")
    out_csv_compare = output_dir / "compare_NX_vs_err.csv"
    df_merged.to_csv(out_csv_compare, index=False)
    plt.figure()
    for law in laws:
        plt.plot(df_merged["err"], df_merged[f"NX_{law}"], marker='o', label=f"{
    labels_map[law]} (pente ~ {abs(slopes_info[law]):.2f})")
    plt.xscale("log");
    plt.xlabel("err (log)"); plt.ylabel("NX_final (log)")
    plt.title("Comparaison NX(err) -- lois de métrique")
    plt.legend()
    plt.savefig(output_dir / "NX_vs_err_compare.png", dpi=160, bbox_inches="tight"
    )
    plt.show()
    plt.close()
    ik_data = {}
    for law in laws:
        ap_c = AdaptParams(
            hmin=0.0125, hmax=0.25, err=0.005, NX_init=12,
            NT_max=600, niter_refinement=12, NX_tol=1,
            L2_target=None, metric_law=law
        )
        x, T, hist = solve_adrs_adapt(phys, ap_c, return_history=True)
        ik_data[law] = np.array(hist["Ik"], dtype=float)
    max_len = max(len(ik_data[law]) for law in laws)
    df_contr = pd.DataFrame({"iter": np.arange(1, max_len+1)})
    for law in laws:
        arr = ik_data[law]
        if len(arr) < max_len:
            arr = np.concatenate([arr, np.full(max_len-len(arr), np.nan)])
        df_contr[f"Ik_{law}"] = arr
    out_csv_contr = output_dir / "contraction_compare.csv"
    df_contr.to_csv(out_csv_contr, index=False)
    plt.figure()
    for law in laws:
        plt.plot(df_contr["iter"], df_contr[f"Ik_{law}"], marker='o', label=
    labels_map[law])
    plt.yscale('log')
    plt.xlabel("Itération d'adaptation k")
    plt.ylabel("||YBk - YBk-1|| (maillage background, L2)")
    plt.title("Contraction des solutions interpolées -- comparaison des lois")
    plt.legend()
    plt.savefig(output_dir / "contraction_compare.png", dpi=160, bbox_inches="
    tight")
    plt.show()
    plt.close()
    return {
        "compare_csv": str(out_csv_compare),
```

```python
            "nx_fig": str(output_dir / "NX_vs_err_compare.png"),
            "contr_csv": str(out_csv_contr),
            "contr_fig": str(output_dir / "contraction_compare.png"),
        }

if __name__ == "__main__":
    out = run_comparisons(output_dir=Path("."))
    print("Generated:")
    for k, v in out.items():
        print(f"- {k}: {v}")
```