Estimation a posteriori

rapport séance 3

Victor Baleux

22 septembre 2025

Table des matières

Chapitre 1

Introduction

1.1 Objectifs du TP

L'objectif de cette séance est d'analyser la convergence de différentes méthodes d'intégration numérique sur deux fonctions tests, une régulière (f_1) et une présentant des singularités (f_2). Nous comparerons :

- les sommes de Riemann (milieux),
- l'intégration de Lebesgue en partition uniforme,
- l'intégration de Lebesgue avec raffinement itératif non-uniforme,
- la méthode de Monte-Carlo.

1.2 Méthodes d'intégration étudiées

La méthode de Riemann repose sur l'échantillonnage de la fonction au centre de chaque sous-intervalle de partition. Les méthodes de Lebesgue consistent à discrétiser l'axe des ordonnées : dans la version uniforme on construit une grille régulière, tandis que dans la version itérative les intervalles sont raffinés en fonction de la densité locale des valeurs. La méthode de Monte-Carlo repose sur un échantillonnage aléatoire uniforme sur le domaine, et une estimation de l'intégrale par moyenne pondérée.

Chapitre 2

Étude de la fonction f_1 sur $[0, 1]$

2.1 Définition et propriétés

On considère la fonction $f_1(x)$ composée de polynômes, sinus et gaussienne centrée en 0.5.

2.2 Représentation graphique

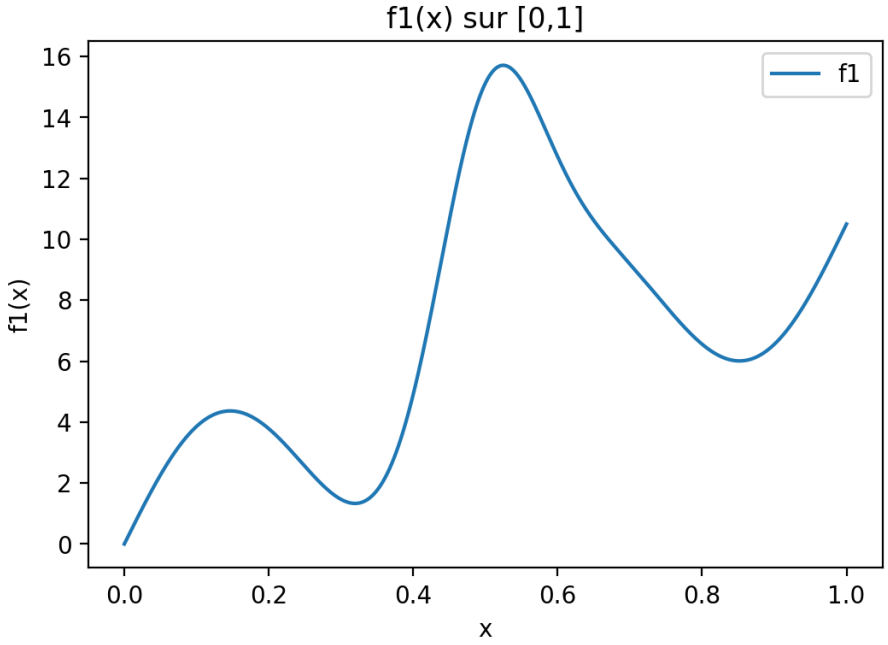

FIGURE 2.1 – Représentation de la fonction f_1 sur $[0, 1]$.

2.3 Étude de convergence

La méthode de Riemann (milieux) converge rapidement avec une pente proche de -2, ce qui confirme la convergence quadratique attendue. La méthode de Lebesgue uniforme

montre une pente proche de -1, traduisant une bonne convergence mais légèrement plus lente. La méthode de Lebesgue itérative présente une pente également voisine de -1, avec un comportement plus robuste sur les zones de variations rapides. Enfin, la méthode de Monte-Carlo a une pente voisine de $-1/2$, en accord avec la convergence théorique $1/\sqrt{N}$.

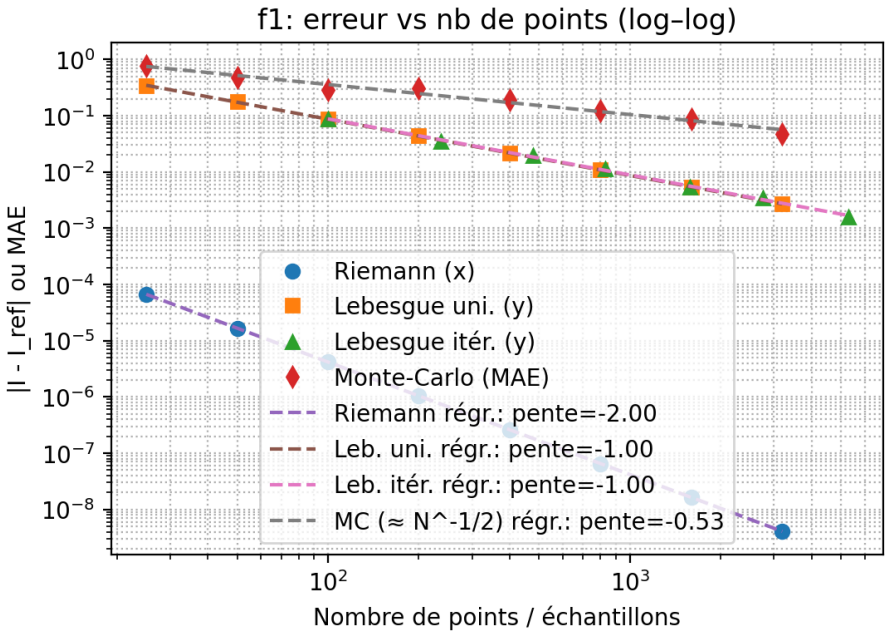

FIGURE 2.2 – Convergence des différentes méthodes appliquées à f_1.

2.4 Analyse comparative

Sur f_1, la méthode de Riemann est particulièrement efficace grâce à la régularité de la fonction, avec une vitesse de convergence nettement supérieure. Les méthodes de Lebesgue offrent une bonne alternative, notamment pour traiter des fonctions plus difficiles. La méthode de Monte-Carlo reste la moins performante en dimension 1, mais elle garde un intérêt pour des problèmes de grande dimension.

Chapitre 3

Étude de la fonction f_2 sur $(-1, 1)$

3.1 Définition et singularités

La fonction test est

$$f_2(x) = \frac{1}{\sqrt{1-x^2}},$$

dont l'intégrale exacte vaut π. Elle présente des singularités intégrables en $x = \pm 1$, ce qui complique l'analyse numérique.

3.2 Représentation graphique

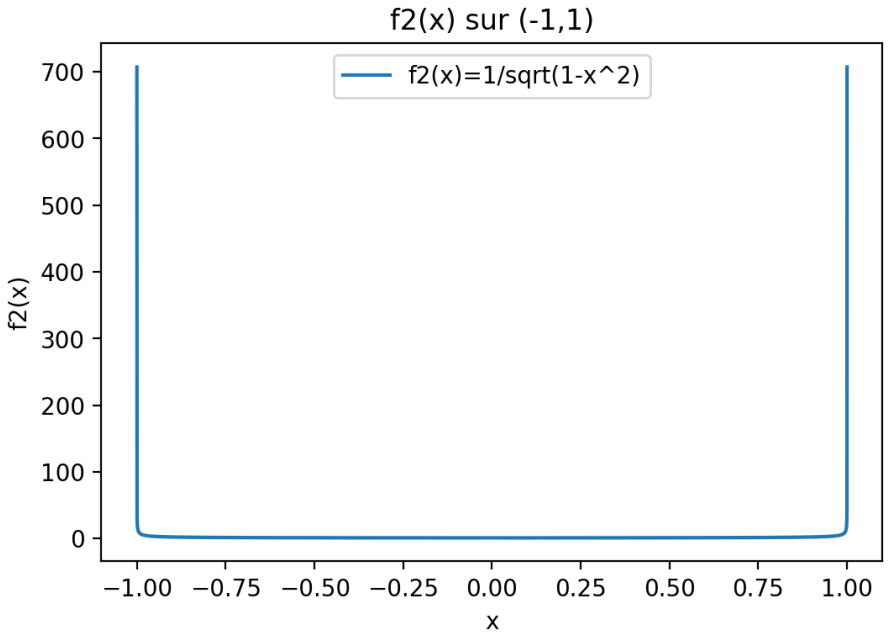

FIGURE 3.1 – Représentation de la fonction f_2 sur $(-1, 1)$.

3.3 Étude de convergence

Pour cette fonction, la méthode de Riemann conserve une pente proche de -0.5, beaucoup plus faible qu'en l'absence de singularités. La méthode de Lebesgue uniforme conserve une bonne vitesse de convergence, avec une pente d'environ -1.05. La version itérative de Lebesgue converge également avec une pente voisine de -0.97, confirmant l'intérêt d'une adaptation locale. La méthode de Monte-Carlo reste lente, avec une pente de l'ordre de -0.45, comme prévu.

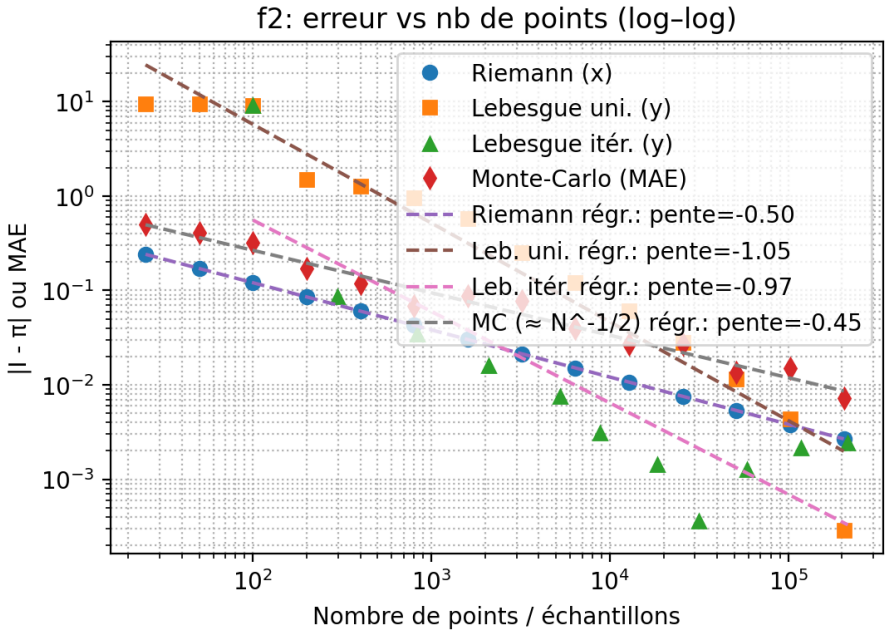

FIGURE 3.2 – Convergence des différentes méthodes appliquées à f_2.

3.4 Analyse comparative

Les singularités ralentissent fortement la convergence de Riemann et Monte-Carlo. Les méthodes de Lebesgue restent robustes et proches de la convergence optimale, ce qui illustre leur efficacité pour traiter des intégrales singulières.

Chapitre 4

Comparaison globale des méthodes

4.1 Vitesses de convergence théoriques attendues

— Riemann : $\mathcal{O}(1/N^2)$ pour une fonction régulière, dégradée en présence de singularités.
— Lebesgue (uniforme et itératif) : convergence de l'ordre de $\mathcal{O}(1/N)$.
— Monte-Carlo : convergence lente en $\mathcal{O}(1/\sqrt{N})$, mais insensible à la dimension.

4.2 Résultats observés pour f_1 et f_2

— f_1 (régulière) : Riemann domine avec une pente -2, suivi par les méthodes de Lebesgue (-1).
— f_2 (singulière) : Lebesgue uniforme (-1.05) et itératif (-0.97) restent efficaces, alors que Riemann et Monte-Carlo chutent à environ -0.5.

4.3 Discussion : points forts et limites des méthodes

Riemann est excellent sur des fonctions régulières mais se dégrade fortement avec des singularités. Les méthodes de Lebesgue offrent un compromis robuste et efficace. Monte-Carlo reste peu performant en 1D, mais sa robustesse et son indépendance à la dimension en font un candidat pour les intégrales multidimensionnelles.

Chapitre 5

Conclusion

5.1 Bilan du TP

Nous avons comparé quatre méthodes d'intégration numérique. Les résultats confirment la supériorité de Riemann sur des fonctions régulières et la robustesse des méthodes de Lebesgue face aux singularités. Monte-Carlo, bien que lent en 1D, conserve un intérêt en grande dimension.

5.2 Perspectives

Il serait intéressant d'étendre l'étude à des intégrales multidimensionnelles, où Monte-Carlo devient compétitif.

Annexe A

Annexe : Code Python

Listing A.1 – Code Python d'intégration et comparaison des méthodes

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Extension "additions only" de Integration_Comparaison.py :
- Ajoute une deuxième fonction f2(x)=1/sqrt(1-x^2) sur (-1,1) (= ).
- Réutilise Riemann (milieux), Lebesgue uniforme et Lebesgue itératif non-uniforme.
- Ajoute les tracés de f1 et f2 et les graphes d'erreur vs nombre de points (log-log).
- Ajoute des droites de régression (log-log) pour visualiser les pentes.
- f2: plus de points dans les maillages pour clarifier la tendance.
- **NOUVEAU** : Ajout d'une méthode d'intégration Monte-Carlo uniforme sur x , avec calcul d'erreur
  (MAE sur plusieurs essais) et ajout au graphe + régression linéaire (log-log).
"""

import argparse
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from typing import Callable, List, Tuple

# ====== (Copie des briques de l'utilisateur - inchangées) ======

Left, Right = 0.0, 1.0
a, b, c = 0.5, 10.0, 3.0
def f(x: np.ndarray) -> np.ndarray:
    return a*x**2 + b*x + c*np.sin(4*np.pi*x) + 10*np.exp(-100*(x-0.5)**2)

def _simpson(f, a, b):
    c = 0.5*(a+b); h = b - a
    return (h/6.0)*(f(a) + 4.0*f(c) + f(b))

def _adaptive_simpson(f, a, b, eps, whole, depth, max_depth):
```

```
    c = 0.5*(a+b)
    left = _simpson(f, a, c)
    right = _simpson(f, c, b)
    if depth >= max_depth:
        return left + right
    if abs(left + right - whole) < 15*eps:
        return left + right + (left + right - whole)/15.0
    return (_adaptive_simpson(f, a, c, eps/2.0, left, depth+1, max_depth) +
            _adaptive_simpson(f, c, b, eps/2.0, right, depth+1, max_depth))

def integral_reference(func: Callable[[float], float],
                       a: float, b: float, eps: float = 1e-12, max_depth:
    int = 30) -> float:
    whole = _simpson(func, a, b)
    return _adaptive_simpson(func, a, b, eps, whole, 0, max_depth)

def riemann_sum_midpoint(func: Callable[[np.ndarray], np.ndarray],
                         left: float, right: float, N: int) -> float:
    h = (right - left) / N
    x = left + h*(np.arange(N) + 0.5)
    return h * np.sum(func(x))

def riemann_convergence(func, left, right, N0=25, levels=8, I_ref=None):
    Ns = [N0 * (2**k) for k in range(levels)]
    Is = np.array([riemann_sum_midpoint(func, left, right, N) for N in Ns],
    dtype=float)
    if I_ref is None:
        abs_err = np.empty_like(Is); abs_err[:] = np.nan
    else:
        abs_err = np.abs(Is - I_ref)
    return np.array(Ns), Is, abs_err

def y_bounds(func, left, right, Nx=100000, pad_ratio=0.05,
    avoid_singular_eps: float = 0.0):
    if avoid_singular_eps > 0:
        x = np.linspace(left + avoid_singular_eps, right -
    avoid_singular_eps, Nx)
    else:
        x = np.linspace(left, right, Nx)
    y = func(x)
    y_min = y.min(); y_max = y.max()
    pad = pad_ratio * max(1.0, (float(y_max) - float(y_min)))
    return (float(y_min) - pad), (float(y_max) + pad), x, y

def lebesgue_integral_from_samples(y_vals: np.ndarray, left: float, right:
    float,
                                   y_min: float, y_max: float, M: int):
    Nxs = y_vals.size
    L_edges = np.linspace(y_min, y_max, M+1)
    counts, _ = np.histogram(y_vals, bins=L_edges)
    measures = counts * (right - left) / float(Nxs)
```

```
    IL = np.sum(measures * L_edges[:-1])
    return IL, measures, L_edges, counts

def lebesgue_convergence(func, left, right, Nx=100000, M0=25, levels=8, I_ref=None, avoid_singular_eps: float = 0.0):
    y_min, y_max, x_grid, y_vals = y_bounds(func, left, right, Nx=Nx, avoid_singular_eps=avoid_singular_eps)
    Ms = [M0 * (2**k) for k in range(levels)]
    ILs = []
    for M in Ms:
        IL, measures, L_edges, counts = lebesgue_integral_from_samples(y_vals, left, right, y_min, y_max, M)
        ILs.append(IL)
    ILs = np.array(ILs, dtype=float)
    if I_ref is None:
        abs_err = np.full_like(ILs, np.nan)
    else:
        abs_err = np.abs(ILs - I_ref)
    return (np.array(Ms), ILs, abs_err, y_min, y_max, x_grid, y_vals, Nx)

def refine_bins_by_load(edges: np.ndarray, counts: np.ndarray, Nx_total: int, overrefine_factor: float = 2.0) -> np.ndarray:
    N = len(edges) - 1
    if N <= 0:
        return edges.copy()
    capacity = Nx_total / float(N)
    new_edges: List[float] = [float(edges[0])]
    for i in range(N):
        left = float(edges[i]); right = float(edges[i+1])
        ci = float(counts[i])
        if ci > capacity:
            k = int(np.ceil(overrefine_factor * ci / capacity))
            k = max(1, k)
            sub_edges = np.linspace(left, right, k+1)[1:]
            new_edges.extend(list(sub_edges))
        else:
            new_edges.append(right)
    return np.array(new_edges, dtype=float)

def lebesgue_nonuniform_iterative_refinement(
    func: Callable[[np.ndarray], np.ndarray],
    left: float, right: float,
    Nx: int = 100000,
    N0: int = 25,
    n_iters: int = 6,
    pad_ratio: float = 0.05,
    save_prefix: str = "lebesgue_nonuni_iter",
    show_plots: bool = True,
    I_ref: float | None = None,
    overrefine_factor: float = 2.0,
    avoid_singular_eps: float = 0.0,
```

```
) -> Tuple[np.ndarray, np.ndarray, np.ndarray, List[np.ndarray]]:
    if avoid_singular_eps > 0:
        x_grid = np.linspace(left + avoid_singular_eps, right - avoid_singular_eps, Nx)
    else:
        x_grid = np.linspace(left, right, Nx)
    y_vals = func(x_grid)
    y_min = float(np.min(y_vals)); y_max = float(np.max(y_vals))
    pad = pad_ratio * max(1.0, (y_max - y_min))
    y_min -= pad; y_max += pad

    edges = np.linspace(y_min, y_max, N0 + 1)
    Nx_total = len(y_vals)

    Ns, ILs, abs_err = [], [], []
    edges_hist: List[np.ndarray] = []

    def integral_from_edges(edges_arr: np.ndarray) -> Tuple[float, np.ndarray]:
        counts, _ = np.histogram(y_vals, bins=edges_arr)
        measures = counts * (right - left) / float(Nx_total)
        IL = float(np.sum(measures * edges_arr[:-1]))
        return IL, counts

    IL0, counts = integral_from_edges(edges)
    Ns.append(len(edges) - 1); ILs.append(IL0)
    abs_err.append(np.nan if I_ref is None else abs(IL0 - I_ref))
    edges_hist.append(edges.copy())

    for it in range(1, n_iters + 1):
        new_edges = refine_bins_by_load(edges, counts, Nx_total, overrefine_factor=overrefine_factor)
        IL_next, counts_next = integral_from_edges(new_edges)

        Ns.append(len(new_edges) - 1); ILs.append(IL_next)
        abs_err.append(np.nan if I_ref is None else abs(IL_next - I_ref))

        edges_hist.append(new_edges.copy())
        edges, counts = new_edges, counts_next

    Ns = np.array(Ns, dtype=int)
    ILs = np.array(ILs, dtype=float)
    abs_err = np.array(abs_err, dtype=float)

    df = pd.DataFrame({"iteration": np.arange(len(Ns)), "N_bins": Ns, "IL": ILs, "abs_err": abs_err})
    df.to_csv(f"{save_prefix}_convergence.csv", index=False)

    return Ns, ILs, abs_err, edges_hist

def save_show(fig_path: str, show: bool):
```

```
    plt.savefig(fig_path, dpi=200, bbox_inches="tight")
    if show: plt.show()
    else: plt.close()

def regression_loglog(points: np.ndarray, errors: np.ndarray) -> Tuple[float, float]:
    mask = (points > 0) & np.isfinite(points) & (errors > 0) & np.isfinite(errors)
    x = np.log10(points[mask]); y = np.log10(errors[mask])
    if x.size < 2:
        return np.nan, np.nan
    A = np.vstack([x, np.ones_like(x)]).T
    slope, intercept = np.linalg.lstsq(A, y, rcond=None)[0]
    return slope, intercept

def add_regression_line(points: np.ndarray, errors: np.ndarray, label_prefix: str):
    slope, intercept = regression_loglog(points, errors)
    if np.isfinite(slope) and np.isfinite(intercept):
        x_fit = np.array([points.min(), points.max()], dtype=float)
        y_fit = 10.0**(intercept) * (x_fit**(slope))
        plt.loglog(x_fit, y_fit, linestyle="--", label=f"{label_prefix} régr.: pente={slope:.2f}")
    return slope, intercept

# ====== NOUVEAU : Monte-Carlo (uniforme sur x) ======

def monte_carlo_integral(func: Callable[[np.ndarray], np.ndarray],
                         left: float, right: float, N: int, rng: np.random.Generator) -> float:
    """Estimateur MC : (right-left) * mean(f(U)), où U ~ U([left,right])."""
    U = rng.random(N) * (right - left) + left
    return (right - left) * float(np.mean(func(U)))

def monte_carlo_convergence(func: Callable[[np.ndarray], np.ndarray],
                            left: float, right: float,
                            N0: int = 25, levels: int = 8,
                            trials: int = 21,
                            I_ref: float | None = None,
                            seed: int = 12345) -> Tuple[np.ndarray, np.ndarray, np.ndarray]:
    """
    Retourne :
      - Ns : tailles d'échantillon
      - I_means : moyenne des estimations Monte-Carlo sur 'trials' essais indépendants
      - err_mae : MAE (mean absolute error) par N, c.-à-d. moyenne de |I_hat - I_ref| sur les 'trials'.
    Si I_ref est None, err_mae est rempli de NaN.
    """
    rng_master = np.random.default_rng(seed)
```

```
    Ns = np.array([N0 * (2**k) for k in range(levels)], dtype=int)
    I_means = np.zeros_like(Ns, dtype=float)
    err_mae = np.full_like(Ns, np.nan, dtype=float)

    for j, N in enumerate(Ns):
        # Essais indépendants (graines dérivées)
        estimates = []
        for t in range(trials):
            rng = np.random.default_rng(rng_master.integers(0, 2**63-1))
            Ihat = monte_carlo_integral(func, left, right, int(N), rng)
            estimates.append(Ihat)
        estimates = np.array(estimates, dtype=float)
        I_means[j] = float(np.mean(estimates))
        if I_ref is not None:
            err_mae[j] = float(np.mean(np.abs(estimates - I_ref)))

    return Ns, I_means, err_mae

# ====== AJOUTS : deuxième fonction et orchestrateur ======

# f1 = f sur [0,1]
def run_for_f1(show_plots: bool = True, mc_trials: int = 21):
    name = "f1"
    L, R = 0.0, 1.0
    func_vec = lambda X: f(X)
    I_ref = integral_reference(lambda xx: float(func_vec(np.array([xx]))[0])
    , L, R, eps=1e-12, max_depth=30)

    x_plot = np.linspace(L, R, 2000)
    y_plot = func_vec(x_plot)
    plt.figure(); plt.plot(x_plot, y_plot, label=name); plt.title(f"{name}(x
    ) sur [0,1]")
    plt.xlabel("x"); plt.ylabel(f"{name}(x)"); plt.legend()
    save_show(f"{name}_function_plot.png", show_plots)

    # Convergences (comme avant) + régression linéaire (log-log)
    Ns_riem, Is_riem, err_riem = riemann_convergence(func_vec, L, R, N0=25,
    levels=8, I_ref=I_ref)
    Ms_lebU, ILs_lebU, err_lebU, *_ = lebesgue_convergence(func_vec, L, R,
    Nx=100000, M0=25, levels=8, I_ref=I_ref)
    Ns_iter, ILs_iter, err_iter, _ =
    lebesgue_nonuniform_iterative_refinement(
        func_vec, L, R, Nx=100000, N0=100, n_iters=6, show_plots=show_plots,
     I_ref=I_ref
    )
    # Monte-Carlo
    Ns_mc, I_means_mc, err_mc = monte_carlo_convergence(func_vec, L, R, N0
    =25, levels=8, trials=mc_trials, I_ref=I_ref)

    plt.figure()
```

```
    plt.loglog(Ns_riem, err_riem, marker="o", linestyle="None", label="Riemann (x)")
    plt.loglog(Ms_lebU, err_lebU, marker="s", linestyle="None", label="Lebesgue uni. (y)")
    plt.loglog(Ns_iter, err_iter, marker="^", linestyle="None", label="Lebesgue itér. (y)")
    plt.loglog(Ns_mc,  err_mc,  marker="d", linestyle="None", label="Monte-Carlo (MAE)")
    # Régressions (log-log)
    add_regression_line(Ns_riem, err_riem, "Riemann")
    add_regression_line(Ms_lebU, err_lebU, "Leb. uni.")
    add_regression_line(Ns_iter, err_iter, "Leb. itér.")
    add_regression_line(Ns_mc,  err_mc,  "MC (≈ N^-1/2)")
    plt.title(f"{name}: erreur vs nb de points (log-log)")
    plt.xlabel("Nombre de points / échantillons"); plt.ylabel("|I - I_ref| ou MAE")
    plt.grid(True, which="both", ls=":"); plt.legend()
    save_show(f"{name}_errors_loglog.png", show_plots)

    rows = []
    for n, e in zip(Ns_riem, err_riem):
        rows.append({"method": "Riemann (milieux, x)", "points": int(n), "abs_error_or_MAE": float(e)})
    for m, e in zip(Ms_lebU, err_lebU):
        rows.append({"method": "Lebesgue uniforme (y)", "points": int(m), "abs_error_or_MAE": float(e)})
    for n, e in zip(Ns_iter, err_iter):
        rows.append({"method": "Lebesgue non uniforme (itér.)", "points": int(n), "abs_error_or_MAE": float(e)})
    for n, e in zip(Ns_mc, err_mc):
        rows.append({"method": f"Monte-Carlo (MAE, {mc_trials} essais)", "points": int(n), "abs_error_or_MAE": float(e)})
    pd.DataFrame(rows).to_csv(f"{name}_errors_vs_points.csv", index=False)

# f2 = 1/sqrt(1-x^2) sur (-1,1) ; I =
def f2(x: np.ndarray) -> np.ndarray:
    return 1.0 / np.sqrt(1.0 - x**2)

def run_for_f2(show_plots: bool = True, mc_trials: int = 21):
    name = "f2"
    L, R = -1.0, 1.0
    func_vec = lambda X: f2(X)
    I_ref = 3.14159265358979323846264338327 9

    x_plot = np.linspace(L + 1e-6, R - 1e-6, 8000)  # plus de points pour la visualisation
    y_plot = func_vec(x_plot)
    plt.figure(); plt.plot(x_plot, y_plot, label=r"f2(x)=1/sqrt(1-x^2)")
    plt.title(f"{name}(x) sur (-1,1)"); plt.xlabel("x"); plt.ylabel(f"{name}(x)"); plt.legend()
    save_show(f"{name}_function_plot.png", show_plots)
```

```
298
299         # Plus de niveaux/points pour clarifier la tendance :
300         Ns_riem, Is_riem, err_riem = riemann_convergence(func_vec, L, R, N0=25,
        levels=14, I_ref=I_ref)
301         Ms_lebU, ILs_lebU, err_lebU, *_ = lebesgue_convergence(func_vec, L, R,
302             Nx=300000, M0=25, levels=14, I_ref=I_ref, avoid_singular_eps=1e-6)
303
304         # Itératif: plus d'itérations (et x-samples) pour lisser la courbe
305         Ns_iter, ILs_iter, err_iter, _ =
        lebesgue_nonuniform_iterative_refinement(
306             func_vec, L, R, Nx=300000, N0=100, n_iters=10, show_plots=show_plots
        , I_ref=I_ref, avoid_singular_eps=1e-6
307         )
308
309         # Monte-Carlo (attention aux singularités aux bords : l'échantillonnage
        uniforme reste valable
310         # car l'intégrale est finie; on se contente d'éviter d'évaluer
        exactement en 1 via U~(L,R)).
311         Ns_mc, I_means_mc, err_mc = monte_carlo_convergence(func_vec, L, R, N0
        =25, levels=14, trials=mc_trials, I_ref=I_ref)
312
313         plt.figure()
314         plt.loglog(Ns_riem, err_riem, marker="o", linestyle="None", label="
        Riemann (x)")
315         plt.loglog(Ms_lebU, err_lebU, marker="s", linestyle="None", label="
        Lebesgue uni. (y)")
316         plt.loglog(Ns_iter, err_iter, marker="^", linestyle="None", label="
        Lebesgue itér. (y)")
317         plt.loglog(Ns_mc,  err_mc,  marker="d", linestyle="None", label="Monte-
        Carlo (MAE)")
318
319         # Ajout des droites de régression pour f2 (log-log)
320         s1, i1 = add_regression_line(Ns_riem, err_riem, "Riemann")
321         s2, i2 = add_regression_line(Ms_lebU, err_lebU, "Leb. uni.")
322         s3, i3 = add_regression_line(Ns_iter, err_iter, "Leb. itér.")
323         s4, i4 = add_regression_line(Ns_mc,  err_mc,  "MC (≈ N^-1/2)")
324
325         plt.title(f"{name}: erreur vs nb de points (log-log)")
326         plt.xlabel("Nombre de points / échantillons"); plt.ylabel("|I - | ou MAE
        ")
327         plt.grid(True, which="both", ls=":"); plt.legend()
328         save_show(f"{name}_errors_loglog.png", show_plots)
329
330         # Exports CSV (incluant pentes)
331         rows = []
332         for n, e in zip(Ns_riem, err_riem):
333             rows.append({"method": "Riemann (milieux, x)", "points": int(n), "
        abs_error_or_MAE": float(e)})
334         for m, e in zip(Ms_lebU, err_lebU):
335             rows.append({"method": "Lebesgue uniforme (y)", "points": int(m), "
        abs_error_or_MAE": float(e)})
```

```
336     for n, e in zip(Ns_iter, err_iter):
337         rows.append({"method": "Lebesgue non uniforme (itér.)", "points":
        int(n), "abs_error_or_MAE": float(e)})
338     for n, e in zip(Ns_mc, err_mc):
339         rows.append({"method": f"Monte-Carlo (MAE, {mc_trials} essais)", "
        points": int(n), "abs_error_or_MAE": float(e)})
340     df = pd.DataFrame(rows)
341     df.to_csv(f"{name}_errors_vs_points.csv", index=False)
342 
343     with open(f"{name}_regression_slopes.txt", "w", encoding="utf-8") as fh:
344         fh.write(f"Riemann slope ≈ {s1:.4f}\nLebesgue uniform slope ≈ {s2
        :.4f}\nLebesgue iterative slope ≈ {s3:.4f}\nMonte-Carlo slope ≈ {s4:.4f
        }\n")
345 
346 def main():
347     parser = argparse.ArgumentParser()
348     parser.add_argument("--no-show", action="store_true", help="ne pas
        afficher les figures (seulement sauvegarder)")
349     parser.add_argument("--function", choices=["f1","f2","both"], default="
        both",
350                         help="choix de la fonction à traiter (défaut: both)"
        )
351     parser.add_argument("--mc-trials", type=int, default=21, help="nombre d'
        essais indépendants pour l'estimation MAE de Monte-Carlo")
352     args = parser.parse_args()
353 
354     show = not args.no_show
355     if args.function in ("f1", "both"):
356         run_for_f1(show_plots=show, mc_trials=args.mc_trials)
357     if args.function in ("f2", "both"):
358         run_for_f2(show_plots=show, mc_trials=args.mc_trials)
359 
360 if __name__ == "__main__":
361     main()
```