



Estimation a posteriori

rapport scéance 3

Table des matières

1	Introduction	2
1.1	Objectifs du TP	2
1.2	Méthodes d'intégration étudiées	2
2	Étude de la fonction f_1 sur $[0, 1]$	3
2.1	Définition et propriétés	3
2.2	Représentation graphique	3
2.3	Étude de convergence	3
2.4	Analyse comparative	4
3	Étude de la fonction f_2 sur $(-1, 1)$	5
3.1	Définition et singularités	5
3.2	Représentation graphique	5
3.3	Étude de convergence	6
3.4	Analyse comparative	6
4	Comparaison globale des méthodes	7
4.1	Vitesses de convergence théoriques attendues	7
4.2	Résultats observés pour f_1 et f_2	7
4.3	Discussion : points forts et limites des méthodes	7
5	Conclusion	8
5.1	Bilan du TP	8
5.2	Perspectives	8
A	Annexe : Code Python	9

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

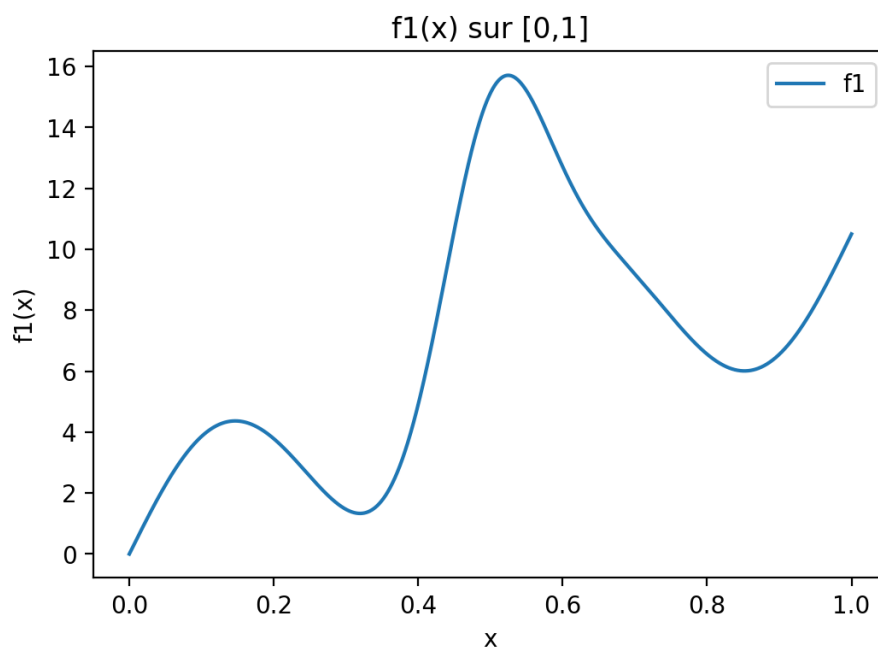


FIGURE 2.1 – Représentation de la fonction f_1 sur $[0, 1]$.

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 $1/\sqrt{N}$.

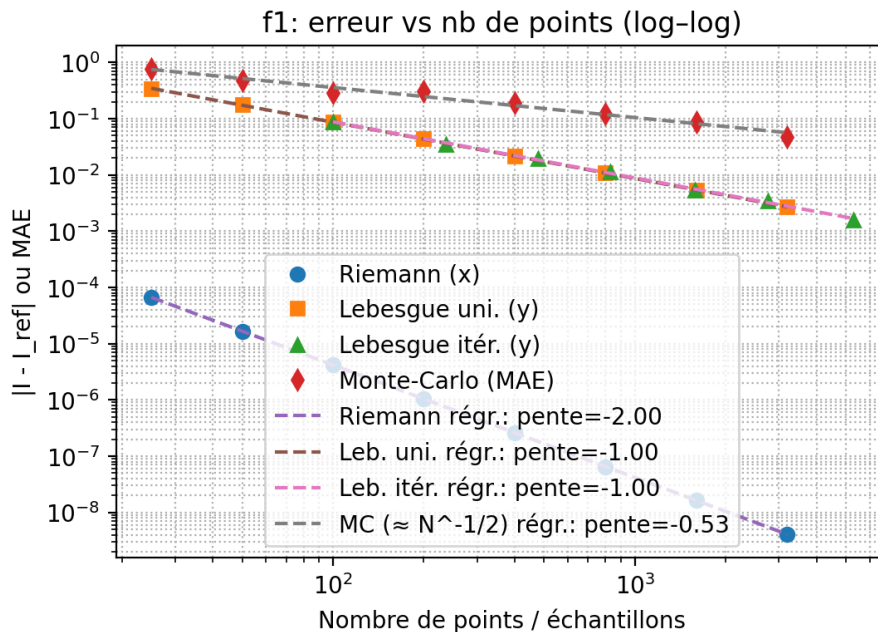


FIGURE 2.2 – Convergence des différentes méthodes appliquées à f_1 .

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

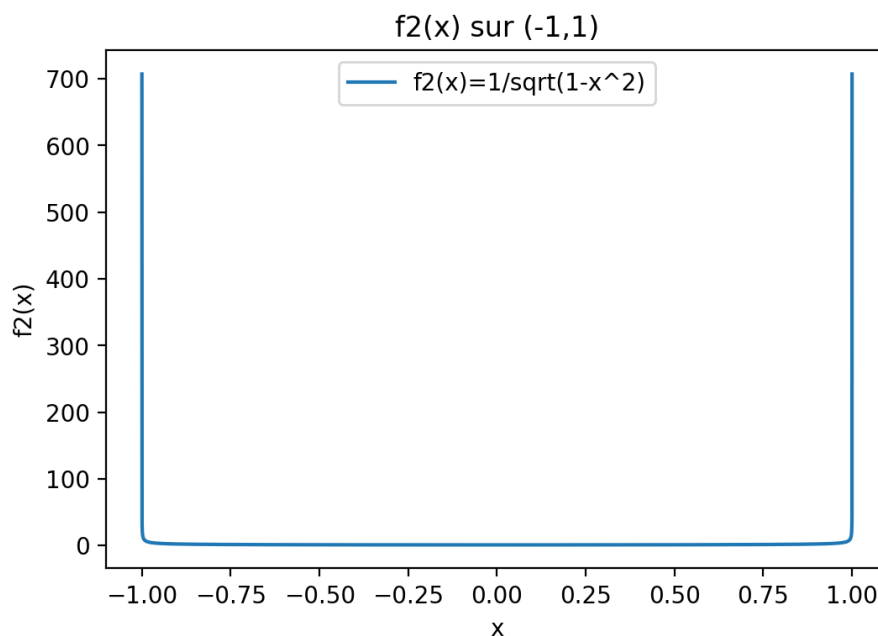


FIGURE 3.1 – Représentation de la fonction f_2 sur $(-1, 1)$.

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.

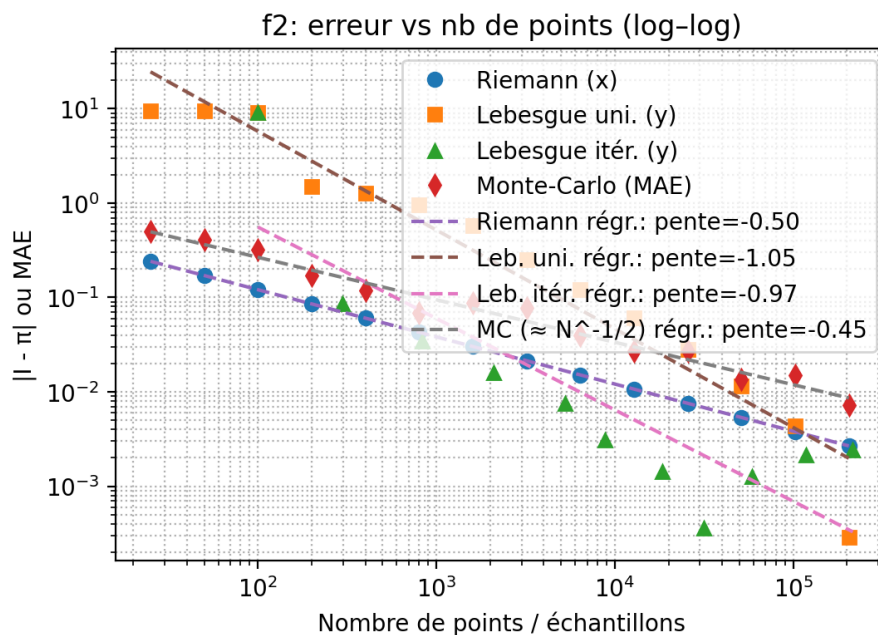


FIGURE 3.2 – Convergence des différentes méthodes appliquées à f_2 .

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

```
1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3  """
4  Extension "additions only" de Integration_Comparaison.py :
5  - Ajoute une deuxième fonction f2(x)=1/sqrt(1-x^2) sur (-1,1) (= ).
6  - Réutilise Riemann (milieux), Lebesgue uniforme et Lebesgue itératif non-
   uniforme.
7  - Ajoute les tracés de f1 et f2 et les graphes d'erreur vs nombre de points
   (log-log).
8  - Ajoute des droites de régression (log-log) pour visualiser les pentes.
9  - f2: plus de points dans les maillages pour clarifier la tendance.
10 - **NOUVEAU** : Ajout d'une méthode d'intégration Monte-Carlo uniforme sur x
   , avec calcul d'erreur
11 (MAE sur plusieurs essais) et ajout au graphe + régression linéaire (log-
   log).
12 """
13
14 import argparse
15 import numpy as np
16 import matplotlib.pyplot as plt
17 import pandas as pd
18 from typing import Callable, List, Tuple
19
20 # ===== (Copie des briques de l'utilisateur - inchangées) =====
21
22 Left, Right = 0.0, 1.0
23 a, b, c = 0.5, 10.0, 3.0
24 def f(x: np.ndarray) -> np.ndarray:
25     return a*x**2 + b*x + c*np.sin(4*np.pi*x) + 10*np.exp(-100*(x-0.5)**2)
26
27 def _simpson(f, a, b):
28     c = 0.5*(a+b); h = b - a
29     return (h/6.0)*(f(a) + 4.0*f(c) + f(b))
30
31 def _adaptive_simpson(f, a, b, eps, whole, depth, max_depth):
```

```

32     c = 0.5*(a+b)
33     left = _simpson(f, a, c)
34     right = _simpson(f, c, b)
35     if depth >= max_depth:
36         return left + right
37     if abs(left + right - whole) < 15*eps:
38         return left + right + (left + right - whole)/15.0
39     return (_adaptive_simpson(f, a, c, eps/2.0, left, depth+1, max_depth) +
40            _adaptive_simpson(f, c, b, eps/2.0, right, depth+1, max_depth))
41
42 def integral_reference(func: Callable[[float], float],
43                       a: float, b: float, eps: float = 1e-12, max_depth:
44                       int = 30) -> float:
45     whole = _simpson(func, a, b)
46     return _adaptive_simpson(func, a, b, eps, whole, 0, max_depth)
47
48 def riemann_sum_midpoint(func: Callable[[np.ndarray], np.ndarray],
49                           left: float, right: float, N: int) -> float:
50     h = (right - left) / N
51     x = left + h*(np.arange(N) + 0.5)
52     return h * np.sum(func(x))
53
54 def riemann_convergence(func, left, right, N0=25, levels=8, I_ref=None):
55     Ns = [N0 * (2**k) for k in range(levels)]
56     Is = np.array([riemann_sum_midpoint(func, left, right, N) for N in Ns],
57                   dtype=float)
58     if I_ref is None:
59         abs_err = np.empty_like(Is); abs_err[:] = np.nan
60     else:
61         abs_err = np.abs(Is - I_ref)
62     return np.array(Ns), Is, abs_err
63
64 def y_bounds(func, left, right, Nx=100000, pad_ratio=0.05,
65             avoid_singular_eps: float = 0.0):
66     if avoid_singular_eps > 0:
67         x = np.linspace(left + avoid_singular_eps, right -
68                          avoid_singular_eps, Nx)
69     else:
70         x = np.linspace(left, right, Nx)
71     y = func(x)
72     y_min = y.min(); y_max = y.max()
73     pad = pad_ratio * max(1.0, (float(y_max) - float(y_min)))
74     return (float(y_min) - pad), (float(y_max) + pad), x, y
75
76 def lebesgue_integral_from_samples(y_vals: np.ndarray, left: float, right:
77                                     float,
78                                     y_min: float, y_max: float, M: int):
79     Nxs = y_vals.size
80     L_edges = np.linspace(y_min, y_max, M+1)
81     counts, _ = np.histogram(y_vals, bins=L_edges)
82     measures = counts * (right - left) / float(Nxs)

```

```

78     IL = np.sum(measures * L_edges[:-1])
79     return IL, measures, L_edges, counts
80
81 def lebesgue_convergence(func, left, right, Nx=100000, M0=25, levels=8,
82     I_ref=None, avoid_singular_eps: float = 0.0):
83     y_min, y_max, x_grid, y_vals = y_bounds(func, left, right, Nx=Nx,
84     avoid_singular_eps=avoid_singular_eps)
85     Ms = [M0 * (2**k) for k in range(levels)]
86     ILs = []
87     for M in Ms:
88         IL, measures, L_edges, counts = lebesgue_integral_from_samples(
89         y_vals, left, right, y_min, y_max, M)
90         ILs.append(IL)
91     ILs = np.array(ILs, dtype=float)
92     if I_ref is None:
93         abs_err = np.full_like(ILs, np.nan)
94     else:
95         abs_err = np.abs(ILs - I_ref)
96     return (np.array(Ms), ILs, abs_err, y_min, y_max, x_grid, y_vals, Nx)
97
98 def refine_bins_by_load(edges: np.ndarray, counts: np.ndarray, Nx_total: int
99     , overrefine_factor: float = 2.0) -> np.ndarray:
100     N = len(edges) - 1
101     if N <= 0:
102         return edges.copy()
103     capacity = Nx_total / float(N)
104     new_edges: List[float] = [float(edges[0])]
105     for i in range(N):
106         left = float(edges[i]); right = float(edges[i+1])
107         ci = float(counts[i])
108         if ci > capacity:
109             k = int(np.ceil(overrefine_factor * ci / capacity))
110             k = max(1, k)
111             sub_edges = np.linspace(left, right, k+1)[1:]
112             new_edges.extend(list(sub_edges))
113         else:
114             new_edges.append(right)
115     return np.array(new_edges, dtype=float)
116
117 def lebesgue_nonuniform_iterative_refinement(
118     func: Callable[[np.ndarray], np.ndarray],
119     left: float, right: float,
120     Nx: int = 100000,
121     N0: int = 25,
122     n_iters: int = 6,
123     pad_ratio: float = 0.05,
124     save_prefix: str = "lebesgue_nonuni_iter",
125     show_plots: bool = True,
126     I_ref: float | None = None,
127     overrefine_factor: float = 2.0,
128     avoid_singular_eps: float = 0.0,

```

```

125 ) -> Tuple[np.ndarray, np.ndarray, np.ndarray, List[np.ndarray]]:
126     if avoid_singular_eps > 0:
127         x_grid = np.linspace(left + avoid_singular_eps, right -
128             avoid_singular_eps, Nx)
129     else:
130         x_grid = np.linspace(left, right, Nx)
131     y_vals = func(x_grid)
132     y_min = float(np.min(y_vals)); y_max = float(np.max(y_vals))
133     pad = pad_ratio * max(1.0, (y_max - y_min))
134     y_min -= pad; y_max += pad
135
136     edges = np.linspace(y_min, y_max, N0 + 1)
137     Nx_total = len(y_vals)
138
139     Ns, ILs, abs_err = [], [], []
140     edges_hist: List[np.ndarray] = []
141
142     def integral_from_edges(edges_arr: np.ndarray) -> Tuple[float, np.
143         ndarray]:
144         counts, _ = np.histogram(y_vals, bins=edges_arr)
145         measures = counts * (right - left) / float(Nx_total)
146         IL = float(np.sum(measures * edges_arr[:-1]))
147         return IL, counts
148
149     IL0, counts = integral_from_edges(edges)
150     Ns.append(len(edges) - 1); ILs.append(IL0)
151     abs_err.append(np.nan if I_ref is None else abs(IL0 - I_ref))
152     edges_hist.append(edges.copy())
153
154     for it in range(1, n_iters + 1):
155         new_edges = refine_bins_by_load(edges, counts, Nx_total,
156             overrefine_factor=overrefine_factor)
157         IL_next, counts_next = integral_from_edges(new_edges)
158
159         Ns.append(len(new_edges) - 1); ILs.append(IL_next)
160         abs_err.append(np.nan if I_ref is None else abs(IL_next - I_ref))
161
162         edges_hist.append(new_edges.copy())
163         edges, counts = new_edges, counts_next
164
165     Ns = np.array(Ns, dtype=int)
166     ILs = np.array(ILs, dtype=float)
167     abs_err = np.array(abs_err, dtype=float)
168
169     df = pd.DataFrame({"iteration": np.arange(len(Ns)), "N_bins": Ns, "IL":
170         ILs, "abs_err": abs_err})
171     df.to_csv(f"{save_prefix}_convergence.csv", index=False)
172
173     return Ns, ILs, abs_err, edges_hist
174
175 def save_show(fig_path: str, show: bool):

```

```

172     plt.savefig(fig_path, dpi=200, bbox_inches="tight")
173     if show: plt.show()
174     else: plt.close()
175
176 def regression_loglog(points: np.ndarray, errors: np.ndarray) -> Tuple[float
177     , float]:
178     mask = (points > 0) & np.isfinite(points) & (errors > 0) & np.isfinite(
179     errors)
180     x = np.log10(points[mask]); y = np.log10(errors[mask])
181     if x.size < 2:
182         return np.nan, np.nan
183     A = np.vstack([x, np.ones_like(x)]).T
184     slope, intercept = np.linalg.lstsq(A, y, rcond=None)[0]
185     return slope, intercept
186
187 def add_regression_line(points: np.ndarray, errors: np.ndarray, label_prefix
188     : str):
189     slope, intercept = regression_loglog(points, errors)
190     if np.isfinite(slope) and np.isfinite(intercept):
191         x_fit = np.array([points.min(), points.max()], dtype=float)
192         y_fit = 10.0**((intercept) * (x_fit**(slope)))
193         plt.loglog(x_fit, y_fit, linestyle="--", label=f"{label_prefix} régr
194         .: pente={slope:.2f}")
195     return slope, intercept
196
197 # ===== NOUVEAU : Monte-Carlo (uniforme sur x) =====
198
199 def monte_carlo_integral(func: Callable[[np.ndarray], np.ndarray],
200     left: float, right: float, N: int, rng: np.random.
201     Generator) -> float:
202     """Estimateur MC : (right-left) * mean(f(U)), où U ~ U([left,right])."""
203     U = rng.random(N) * (right - left) + left
204     return (right - left) * float(np.mean(func(U)))
205
206 def monte_carlo_convergence(func: Callable[[np.ndarray], np.ndarray],
207     left: float, right: float,
208     N0: int = 25, levels: int = 8,
209     trials: int = 21,
210     I_ref: float | None = None,
211     seed: int = 12345) -> Tuple[np.ndarray, np.
212     ndarray, np.ndarray]:
213     """
214     Retourne :
215     - Ns : tailles d'échantillon
216     - I_means : moyenne des estimations Monte-Carlo sur 'trials' essais
217     indépendants
218     - err_mae : MAE (mean absolute error) par N, c.-à-d. moyenne de |I_hat
219     - I_ref| sur les 'trials'.
220     Si I_ref est None, err_mae est rempli de NaN.
221     """
222     rng_master = np.random.default_rng(seed)

```

```

215     Ns = np.array([N0 * (2**k) for k in range(levels)], dtype=int)
216     I_means = np.zeros_like(Ns, dtype=float)
217     err_mae = np.full_like(Ns, np.nan, dtype=float)
218
219     for j, N in enumerate(Ns):
220         # Essais indépendants (graines dérivées)
221         estimates = []
222         for t in range(trials):
223             rng = np.random.default_rng(rng_master.integers(0, 2**63-1))
224             Ihat = monte_carlo_integral(func, left, right, int(N), rng)
225             estimates.append(Ihat)
226         estimates = np.array(estimates, dtype=float)
227         I_means[j] = float(np.mean(estimates))
228         if I_ref is not None:
229             err_mae[j] = float(np.mean(np.abs(estimates - I_ref)))
230
231     return Ns, I_means, err_mae
232
233 # ===== AJOUTS : deuxième fonction et orchestrateur =====
234
235 # f1 = f sur [0,1]
236 def run_for_f1(show_plots: bool = True, mc_trials: int = 21):
237     name = "f1"
238     L, R = 0.0, 1.0
239     func_vec = lambda X: f(X)
240     I_ref = integral_reference(lambda xx: float(func_vec(np.array([xx]))[0]),
241                               L, R, eps=1e-12, max_depth=30)
242
243     x_plot = np.linspace(L, R, 2000)
244     y_plot = func_vec(x_plot)
245     plt.figure(); plt.plot(x_plot, y_plot, label=name); plt.title(f"{name}(x) sur [0,1]")
246     plt.xlabel("x"); plt.ylabel(f"{name}(x)"); plt.legend()
247     save_show(f"{name}_function_plot.png", show_plots)
248
249     # Convergences (comme avant) + régression linéaire (log-log)
250     Ns_riem, Is_riem, err_riem = riemann_convergence(func_vec, L, R, N0=25,
251                                                       levels=8, I_ref=I_ref)
252     Ms_lebU, IIs_lebU, err_lebU, *_ = lebesgue_convergence(func_vec, L, R,
253                                                             Nx=100000, M0=25, levels=8, I_ref=I_ref)
254     Ns_iter, IIs_iter, err_iter, _ =
255     lebesgue_nonuniform_iterative_refinement(
256         func_vec, L, R, Nx=100000, N0=100, n_iters=6, show_plots=show_plots,
257         I_ref=I_ref
258     )
259
260     # Monte-Carlo
261     Ns_mc, I_means_mc, err_mc = monte_carlo_convergence(func_vec, L, R, N0
262                                                         =25, levels=8, trials=mc_trials, I_ref=I_ref)
263
264     plt.figure()

```

```

258     plt.loglog(Ns_riem, err_riem, marker="o", linestyle="None", label="
Riemann (x)")
259     plt.loglog(Ms_lebU, err_lebU, marker="s", linestyle="None", label="
Lebesgue uni. (y)")
260     plt.loglog(Ns_iter, err_iter, marker="^", linestyle="None", label="
Lebesgue itér. (y)")
261     plt.loglog(Ns_mc, err_mc, marker="d", linestyle="None", label="Monte-
Carlo (MAE)")
262     # Régressions (log-log)
263     add_regression_line(Ns_riem, err_riem, "Riemann")
264     add_regression_line(Ms_lebU, err_lebU, "Leb. uni.")
265     add_regression_line(Ns_iter, err_iter, "Leb. itér.")
266     add_regression_line(Ns_mc, err_mc, "MC ( $\approx N^{-1/2}$ ")
267     plt.title(f"{name}: erreur vs nb de points (log-log)")
268     plt.xlabel("Nombre de points / échantillons"); plt.ylabel("|I - I_ref|
ou MAE")
269     plt.grid(True, which="both", ls=":"); plt.legend()
270     save_show(f"{name}_errors_loglog.png", show_plots)
271
272     rows = []
273     for n, e in zip(Ns_riem, err_riem):
274         rows.append({"method": "Riemann (milieux, x)", "points": int(n), "
abs_error_or_MAE": float(e)})
275     for m, e in zip(Ms_lebU, err_lebU):
276         rows.append({"method": "Lebesgue uniforme (y)", "points": int(m), "
abs_error_or_MAE": float(e)})
277     for n, e in zip(Ns_iter, err_iter):
278         rows.append({"method": "Lebesgue non uniforme (itér.)", "points":
int(n), "abs_error_or_MAE": float(e)})
279     for n, e in zip(Ns_mc, err_mc):
280         rows.append({"method": f"Monte-Carlo (MAE, {mc_trials} essais)", "
points": int(n), "abs_error_or_MAE": float(e)})
281     pd.DataFrame(rows).to_csv(f"{name}_errors_vs_points.csv", index=False)
282
283     #  $f_2 = 1/\sqrt{1-x^2}$  sur  $(-1,1)$  ;  $I =$ 
284     def f2(x: np.ndarray) -> np.ndarray:
285         return 1.0 / np.sqrt(1.0 - x**2)
286
287     def run_for_f2(show_plots: bool = True, mc_trials: int = 21):
288         name = "f2"
289         L, R = -1.0, 1.0
290         func_vec = lambda X: f2(X)
291         I_ref = 3.141592653589793238462643383279
292
293         x_plot = np.linspace(L + 1e-6, R - 1e-6, 8000) # plus de points pour la
visualisation
294         y_plot = func_vec(x_plot)
295         plt.figure(); plt.plot(x_plot, y_plot, label=r"f2(x)=1/sqrt(1-x^2)")
296         plt.title(f"{name}(x) sur (-1,1)"); plt.xlabel("x"); plt.ylabel(f"{name}
(x)"); plt.legend()
297         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,
301     levels=14, I_ref=I_ref)
302     Ms_lebU, IIs_lebU, err_lebU, *_ = lebesgue_convergence(func_vec, L, R,
303     Nx=300000, M0=25, levels=14, I_ref=I_ref, avoid_singular_eps=1e-6)
304
305     # Itératif: plus d'itérations (et x-samples) pour lisser la courbe
306     Ns_iter, IIs_iter, err_iter, _ =
307     lebesgue_nonuniform_iterative_refinement(
308         func_vec, L, R, Nx=300000, N0=100, n_iters=10, show_plots=show_plots
309         , I_ref=I_ref, avoid_singular_eps=1e-6
310     )
311
312     # Monte-Carlo (attention aux singularités aux bords : l'échantillonnage
313     uniforme reste valable
314     # car l'intégrale est finie; on se contente d'éviter d'évaluer
315     exactement en 1 via  $U^*(L,R)$ ).
316     Ns_mc, I_means_mc, err_mc = monte_carlo_convergence(func_vec, L, R, N0
317     =25, levels=14, trials=mc_trials, I_ref=I_ref)
318
319     plt.figure()
320     plt.loglog(Ns_riem, err_riem, marker="o", linestyle="None", label="
321     Riemann (x)")
322     plt.loglog(Ms_lebU, err_lebU, marker="s", linestyle="None", label="
323     Lebesgue uni. (y)")
324     plt.loglog(Ns_iter, err_iter, marker="^", linestyle="None", label="
325     Lebesgue itér. (y)")
326     plt.loglog(Ns_mc, err_mc, marker="d", linestyle="None", label="Monte-
327     Carlo (MAE)")
328
329     # Ajout des droites de régression pour f2 (log-log)
330     s1, i1 = add_regression_line(Ns_riem, err_riem, "Riemann")
331     s2, i2 = add_regression_line(Ms_lebU, err_lebU, "Leb. uni.")
332     s3, i3 = add_regression_line(Ns_iter, err_iter, "Leb. itér.")
333     s4, i4 = add_regression_line(Ns_mc, err_mc, "MC ( $\approx N^{-1/2}$ ")
334
335     plt.title(f"{name}: erreur vs nb de points (log-log)")
336     plt.xlabel("Nombre de points / échantillons"); plt.ylabel("|I - | ou MAE
337     ")
338     plt.grid(True, which="both", ls=":"); plt.legend()
339     save_show(f"{name}_errors_loglog.png", show_plots)
340
341     # Exports CSV (incluant pentes)
342     rows = []
343     for n, e in zip(Ns_riem, err_riem):
344         rows.append({"method": "Riemann (milieux, x)", "points": int(n), "
345         abs_error_or_MAE": float(e)})
346     for m, e in zip(Ms_lebU, err_lebU):
347         rows.append({"method": "Lebesgue uniforme (y)", "points": int(m), "
348         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  $\approx$  {s1:.4f}\nLebesgue uniform slope  $\approx$  {s2
:.4f}\nLebesgue iterative slope  $\approx$  {s3:.4f}\nMonte-Carlo slope  $\approx$  {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()

```