

膜宇宙論 条件14/eta検証 解析スクリプト全文カタログ

2026年4月11日午後セッション / 坂口 忍
スクリプト 6 本、総行数 3,585 行

目次・スクリプト一覧

#	ファイル名	目的	結果	行数
1	sparc_kappa_estimation.py	kappa=const 推定	X級 (否定)	723
2	sparc_kappa_nonlinear.py	非線形8モデル系統テスト	M3有意 (B-級)	743
3	sparc_kappa_circularity.py	循環参照5段階検証	真の信号確認	511
4	sparc_M3_M6_independence.py	M3/M6独立性7段階	M6冗長	607
5	epsilon_c_derivation.py	epsilon_c理論導出	X級 (不成功)	531
6	sparc_eta_derivation.py	eta第一原理導出	部分的成功	470

検証パイプラインの流れ: Script 1 (kappa=const否定) -> Script 2 (非線形M3有意) -> Script 3 (循環参照テスト通過) -> Script 4 (M6冗長判明) -> Script 5 (epsilon_c導出不成功) -> Script 6 (eta導出部分的成功)

実行環境: ローカル Windows + Claude Code (VS Code拡張)。コマンド: `uv run --with scipy --with matplotlib python [script]`。データ: SPARC sparc_gc.csv + Rotmod_LTG/ (175銀河の回転曲線)。Script 5のみデータ不要 (純粋理論計算)。

Script 1: sparc_kappa_estimation.py

目的

kappa=const (線形膜剛性) モデルの推定

膜ラグランジアン $L = U(\epsilon; c) + \kappa(d\epsilon/dr)^2$ の κ を定数として推定する。gc = gc_deep x (1 + kappa x E_grad/E_local) を log 空間で最小二乗フィットし、bootstrap CI (N=2000)、普遍性検証 (kappa_i vs 銀河パラメータ)、hR 偏相関改善、循環参照チェック (シャッフルテスト) を実施。

結果

X級 (否定) : kappa x median(ratio) = -0.006 (0.6%補正)、dAIC = +1.1 (悪化)、kappa_i vs Log(hR) で rho=-0.474 (普遍定数でない)。線形パラメータ化は膜の非線形応答を捉えられない。

実行方法

```
uv run --with scipy --with matplotlib python sparc_kappa_estimation.py
```

入力データ: sparc_gc.csv + Rotmod_LTG/*.dat

出力: kappa_estimation_results.png + 標準出力

ソースコード全文

```
1 | #!/usr/bin/env python3
2 | """
3 | sparc_kappa_estimation.py
4 | =====
5 | ##### K #####
6 |
7 | ###: gc = gc_deep x (1 + kappa x E_grad/E_local)
8 | - gc_deep = deep-MOND limit gc (x<1 #####)
9 | - epsilon(r) = sqrt(gN(r) / gc_deep)
10 | - E_grad = mean[(de/dr)^2] (#####radial bin##)
11 | - E_local = mean[epsilon^2]
12 | - ratio = E_grad / E_local
13 |
14 | #####:
15 | (1) K ##### + bootstrap CI
16 | (2) K ##### (## vs #####)
17 | (3) K ##### (a bending rigidity)
18 | (4) gc_corrected ## hR #####
19 | (5) ##13%#####
20 |
21 | ##: uv run --with scipy --with matplotlib python sparc_kappa_estimation.py
22 | """
23 |
24 | import os, sys, glob
25 | import numpy as np
26 | from scipy.optimize import minimize_scalar, minimize
27 | from scipy.stats import spearmanr, pearsonr
28 | import warnings
29 | warnings.filterwarnings('ignore')
30 |
31 | # =====
32 | # 0. #####
33 | # =====
34 | BASE = r"D:\#####\#####\#####\#####\#####"
35 | ROTMOD = os.path.join(BASE, "Rotmod_LTG")
36 | GC_CSV = os.path.join(BASE, "sparc_gc.csv")
37 |
38 | a0 = 1.2e-10 # m/s^2, MOND#####
39 |
40 | # =====
41 | # 1. sparc_gc.csv #####
42 | # =====
43 | def load_gc_csv():
44 |     """sparc_gc.csv #####gc##### (m/s^2#####)"""
45 |     import csv
46 |     data = {}
47 |     with open(GC_CSV, 'r') as f:
48 |         reader = csv.DictReader(f)
49 |         for row in reader:
50 |             name = row.get('galaxy') or row.get('Galaxy') or row.get('name')
51 |             if name is None:
52 |                 # #####
53 |                 for k in row:
54 |                     if 'gal' in k.lower() or 'name' in k.lower():
55 |                         name = row[k]
56 |                         break
57 |             if name is None:
58 |                 continue
59 |             name = name.strip()
```

```

60 |
61 |     # gc
62 |     gc_val = None
63 |     for k in ['gc', 'gc_obs', 'g_c', 'gc_a0']:
64 |         if k in row and row[k]:
65 |             try:
66 |                 gc_val = float(row[k])
67 |                 break
68 |             except:
69 |                 pass
70 |     if gc_val is None:
71 |         continue
72 |
73 |     # vflat, hR
74 |     vflat = None
75 |     for k in ['vflat', 'Vflat', 'v_flat']:
76 |         if k in row and row[k]:
77 |             try:
78 |                 vflat = float(row[k])
79 |                 break
80 |             except:
81 |                 pass
82 |
83 |     hR = None
84 |     for k in ['hR', 'Reff', 'h_R', 'hr']:
85 |         if k in row and row[k]:
86 |             try:
87 |                 hR = float(row[k])
88 |                 break
89 |             except:
90 |                 pass
91 |
92 |     Yd = None
93 |     for k in ['Yd', 'yd', 'Y_d', 'SPS']:
94 |         if k in row and row[k]:
95 |             try:
96 |                 Yd = float(row[k])
97 |                 break
98 |             except:
99 |                 pass
100 |
101 |     data[name] = {'gc': gc_val, 'vflat': vflat, 'hR': hR, 'Yd': Yd}
102 |
103 | # gc = a0 * M / s^2
104 | gc_vals = [d['gc'] for d in data.values()]
105 | median_gc = np.median(gc_vals)
106 | if median_gc < 1e-5:
107 |     # m/s^2
108 |     gc_unit = 'mps2'
109 | else:
110 |     # a0 → m/s^2
111 |     gc_unit = 'a0'
112 |     for name in data:
113 |         data[name]['gc'] *= a0
114 |
115 | print(f"sparc_gc.csv: {len(data)} galaxies loaded (gc unit detected: {gc_unit})")
116 | return data
117 |
118 | # =====
119 | # 2. Rotmod_LTG
120 | # =====
121 | def load_rotcurve(galaxy_name):
122 |     """
123 |     Rotmod_LTG<galaxy>_rotmod.dat
124 |     : Rad(kpc), Vobs, errV, Vgas, Vdisk, Vbul, SBdisk, SBbul
125 |     """
126 |     fname = os.path.join(ROTMOD, f"{galaxy_name}_rotmod.dat")
127 |     if not os.path.exists(fname):
128 |         return None
129 |
130 |     rad, vobs, vgas, vdisk, vbul = [], [], [], [], []
131 |     with open(fname, 'r') as f:
132 |         for line in f:
133 |             line = line.strip()
134 |             if not line or line.startswith('#'):
135 |                 continue
136 |             parts = line.split()
137 |             if len(parts) < 6:
138 |                 continue
139 |             try:
140 |                 r = float(parts[0])
141 |                 vo = float(parts[1])
142 |                 vg = float(parts[3])
143 |                 vd = float(parts[4])
144 |                 vb = float(parts[5])
145 |                 rad.append(r)
146 |                 vobs.append(vo)
147 |                 vgas.append(vg)
148 |                 vdisk.append(vd)
149 |                 vbul.append(vb)
150 |             except:
151 |                 continue

```

```

152 |         return None
153 |
154 |     return {
155 |         'r': np.array(rad),          # kpc
156 |         'vobs': np.array(vobs),     # km/s
157 |         'vgas': np.array(vgas),     # km/s
158 |         'vdisk': np.array(vdisk),   # km/s (sqrt(Yd) * gc)
159 |         'vbul': np.array(vbul),     # km/s
160 |     }
161 |
162 | # =====
163 | # 3. gN * gc + ε(r) + E_grad/E_local
164 | # =====
165 | def compute_strain_quantities(rc, gc_deep, Yd=0.5):
166 |     """
167 |     Compute strain quantities:
168 |     gN(r) = (Yd*vdisk^2 + vgas^2 + vbul^2) / r (in m/s^2)
169 |     ε(r) = sqrt(gN(r) / gc_deep)
170 |     E_grad = mean[(dε/dr)^2]
171 |     E_local = mean[ε^2]
172 |     ratio = E_grad / E_local
173 |     """
174 |     r_kpc = rc['r']
175 |     r_m = r_kpc * 3.086e19 # kpc -> m
176 |
177 |     # gN = V_bar^2 / r (centripetal acceleration from baryons)
178 |     v_bar2 = Yd * rc['vdisk']**2 + rc['vgas']**2 + rc['vbul']**2 # (km/s)^2
179 |     v_bar2_mps2 = v_bar2 * 1e6 # -> (m/s)^2
180 |
181 |     # gN in m/s^2
182 |     gN = v_bar2_mps2 / r_m
183 |
184 |     # gc_deep < 0 (deep MOND)
185 |     if gc_deep <= 0:
186 |         return None
187 |
188 |     # ε(r) = sqrt(gN / gc_deep), gN < 0
189 |     gN_pos = np.maximum(gN, 0)
190 |     ratio_gN = gN_pos / gc_deep
191 |     epsilon = np.sqrt(ratio_gN)
192 |
193 |     # ε > 1 (Newtonian)
194 |     # deep-MOND ε > 1
195 |
196 |     # dε/dr (finite difference)
197 |     if len(r_m) < 3:
198 |         return None
199 |
200 |     # central difference (forward/backward)
201 |     deps_dr = np.zeros_like(epsilon)
202 |     deps_dr[0] = (epsilon[1] - epsilon[0]) / (r_m[1] - r_m[0])
203 |     deps_dr[-1] = (epsilon[-1] - epsilon[-2]) / (r_m[-1] - r_m[-2])
204 |     for i in range(1, len(epsilon)-1):
205 |         deps_dr[i] = (epsilon[i+1] - epsilon[i-1]) / (r_m[i+1] - r_m[i-1])
206 |
207 |     E_grad = np.mean(deps_dr**2)
208 |     E_local = np.mean(epsilon**2)
209 |
210 |     if E_local == 0:
211 |         return None
212 |
213 |     return {
214 |         'r_kpc': r_kpc,
215 |         'gN': gN,
216 |         'epsilon': epsilon,
217 |         'deps_dr': deps_dr,
218 |         'E_grad': E_grad,
219 |         'E_local': E_local,
220 |         'ratio': E_grad / E_local,
221 |         'mean_epsilon': np.mean(epsilon),
222 |         'max_epsilon': np.max(epsilon),
223 |     }
224 |
225 | # =====
226 | # 4. gc_deep * gc (x < 1) (deep MOND)
227 | # =====
228 | def compute_gc_deep(rc, Yd=0.5):
229 |     """
230 |     Deep-MOND limit: gobs = sqrt(gN * gc)
231 |     → gc = gobs^2 / gN (in m/s^2)
232 |     x = gN/gc < 1 (deep MOND)
233 |     → median gc_deep
234 |     """
235 |     r_m = rc['r'] * 3.086e19
236 |     v_bar2 = (Yd * rc['vdisk']**2 + rc['vgas']**2 + rc['vbul']**2) * 1e6
237 |     v_obs2 = (rc['vobs'] * 1e3)**2
238 |
239 |     gN = v_bar2 / r_m
240 |     gobs = v_obs2 / r_m
241 |
242 |     # gc_point = gobs^2 / gN (deep MOND)
243 |     mask = gN > 0

```

```

244 |     gc_points = gobs[mask]**2 / gN[mask]
245 |
246 |     if len(gc_points) < 3:
247 |         return None
248 |
249 |     # x = gN / gc &lt; 1 0000 (iterative: 000 median gc 000)
250 |     gc_med = np.median(gc_points)
251 |     if gc_med &lt;= 0:
252 |         return None
253 |
254 |     x_vals = gN[mask] / gc_med
255 |     deep_mask = x_vals &lt; 1.0
256 |
257 |     if np.sum(deep_mask) < 3:
258 |         # x&lt;1 00000 → 00 median 000
259 |         return gc_med
260 |
261 |     return np.median(gc_points[deep_mask])
262 |
263 | # =====
264 | # 5. 00000
265 | # =====
266 | def main():
267 |     print("=" * 70)
268 |     print("k0000000000")
269 |     print("=" * 70)
270 |
271 |     # --- 000000 ---
272 |     gc_data = load_gc_csv()
273 |
274 |     # --- 00000 ---
275 |     results = []
276 |
277 |     galaxies = sorted(gc_data.keys())
278 |     n_loaded = 0
279 |     n_skipped = 0
280 |
281 |     for gname in galaxies:
282 |         gd = gc_data[gname]
283 |         gc_obs = gd['gc'] # m/s^2
284 |         vflat = gd.get('vflat')
285 |         hR = gd.get('hR')
286 |         Yd = gd.get('Yd') or 0.5
287 |
288 |         if gc_obs &lt;= 0 or vflat is None or hR is None:
289 |             n_skipped += 1
290 |             continue
291 |         if vflat &lt;= 0 or hR &lt;= 0:
292 |             n_skipped += 1
293 |             continue
294 |
295 |         rc = load_rotcurve(gname)
296 |         if rc is None:
297 |             n_skipped += 1
298 |             continue
299 |
300 |         # gc_deep
301 |         gc_deep = compute_gc_deep(rc, Yd=Yd)
302 |         if gc_deep is None or gc_deep &lt;= 0:
303 |             n_skipped += 1
304 |             continue
305 |
306 |         # 000
307 |         sq = compute_strain_quantities(rc, gc_deep, Yd=Yd)
308 |         if sq is None:
309 |             n_skipped += 1
310 |             continue
311 |
312 |         results.append({
313 |             'name': gname,
314 |             'gc_obs': gc_obs,
315 |             'gc_deep': gc_deep,
316 |             'vflat': vflat,
317 |             'hR': hR,
318 |             'Yd': Yd,
319 |             'ratio': sq['ratio'],
320 |             'E_grad': sq['E_grad'],
321 |             'E_local': sq['E_local'],
322 |             'mean_eps': sq['mean_epsilon'],
323 |             'n_points': len(rc['r']),
324 |             'rmax_hR': rc['r'][-1] / hR if hR > 0 else 0,
325 |         })
326 |         n_loaded += 1
327 |
328 |     print(f"\nLoaded: {n_loaded}, Skipped: {n_skipped}")
329 |
330 |     if n_loaded < 20:
331 |         print("ERROR: 000000000000000000000000")
332 |         sys.exit(1)
333 |
334 |     # --- numpy000 ---
335 |     gc_obs = np.array([r['gc_obs'] for r in results])

```

```

336 | gc_deep = np.array([r['gc_deep'] for r in results])
337 | ratio = np.array([r['ratio'] for r in results])
338 | vflat = np.array([r['vflat'] for r in results])
339 | hR = np.array([r['hR'] for r in results])
340 | Yd = np.array([r['Yd'] for r in results])
341 | rmax_hR = np.array([r['rmax_hR'] for r in results])
342 | names = [r['name'] for r in results]
343 |
344 | # =====
345 | # (1) K oooo
346 | # =====
347 | print("\n" + "=" * 70)
348 | print("(1) K ooooo")
349 | print("=" * 70)
350 |
351 | # ooo: gc_obs = gc_deep * (1 + k * ratio)
352 | # logoo: log(gc_obs) = log(gc_deep) + log(1 + k * ratio)
353 | # -> oooo: minimize  $\sum[\log(gc\_obs/gc\_deep) - \log(1 + k*ratio)]^2$ 
354 |
355 | log_excess = np.log10(gc_obs / gc_deep)
356 |
357 | # oo gc_obs/gc_deep ooooo
358 | med_excess = np.median(gc_obs / gc_deep)
359 | print(f"gc_obs/gc_deep: median={med_excess:.3f}, "
360 |       f"mean={np.mean(gc_obs/gc_deep):.3f}, "
361 |       f"std(log)={np.std(log_excess):.3f} dex")
362 | print(f"ratio (E_grad/E_local): median={np.median(ratio):.4e}, "
363 |       f"range=[{np.min(ratio):.4e}, {np.max(ratio):.4e}]")
364 |
365 | def chi2_kappa(kappa):
366 |     model = np.log10(1 + kappa * ratio)
367 |     return np.sum((log_excess - model)**2)
368 |
369 | # grid search for initial value
370 | kappa_grid = np.linspace(-1e20, 1e20, 1000)
371 | # ratio ooooooooooooooooooooo
372 | ratio_scale = np.median(ratio)
373 | if ratio_scale > 0:
374 |     kappa_max = 10.0 / ratio_scale
375 | else:
376 |     kappa_max = 1e20
377 |
378 | kappa_grid = np.linspace(-kappa_max, kappa_max, 2000)
379 | chi2_vals = [chi2_kappa(k) for k in kappa_grid]
380 | k_init = kappa_grid[np.argmin(chi2_vals)]
381 |
382 | # optimize
383 | from scipy.optimize import minimize_scalar
384 | res = minimize_scalar(chi2_kappa, bounds=(-kappa_max*2, kappa_max*2), method='bounded')
385 | kappa_best = res.x
386 | chi2_best = res.fun
387 |
388 | # null model (k=0)
389 | chi2_null = np.sum(log_excess**2)
390 |
391 | N = len(gc_obs)
392 | dof = N - 1
393 | dAIC = chi2_best - chi2_null + 2 # +1 parameter
394 |
395 | print(f"\nk_best = {kappa_best:.6e}")
396 | print(f"ratio_scale (median) = {ratio_scale:.6e}")
397 | print(f"k * median(ratio) = {kappa_best * ratio_scale:.4f}")
398 | print(f"chi2(k=0) = {chi2_null:.3f}, chi2(k_best) = {chi2_best:.3f}")
399 | print(f"dAIC = {dAIC:.1f} (negative = Koooo)")
400 |
401 | # R² improvement
402 | SS_tot = chi2_null
403 | SS_res = chi2_best
404 | R2 = 1 - SS_res / SS_tot
405 | print(f"R²(Koo) = {R2:.4f}")
406 |
407 | # gc_corrected
408 | gc_corrected = gc_deep * (1 + kappa_best * ratio)
409 | residual_corrected = np.log10(gc_obs / gc_corrected)
410 | residual_uncorrected = log_excess
411 | print(f"residual std: uncorrected={np.std(residual_uncorrected):.4f} dex, "
412 |       f"corrected={np.std(residual_corrected):.4f} dex")
413 |
414 | # =====
415 | # (1b) Bootstrap CI for k
416 | # =====
417 | print("\n--- Bootstrap CI (N=2000) ---")
418 | np.random.seed(42)
419 | kappa_boot = []
420 | for _ in range(2000):
421 |     idx = np.random.randint(0, N, N)
422 |     le = log_excess[idx]
423 |     ra = ratio[idx]
424 |     def chi2_b(k):
425 |         return np.sum((le - np.log10(np.maximum(1 + k * ra, 1e-10)))**2)
426 |     rb = minimize_scalar(chi2_b, bounds=(-kappa_max*2, kappa_max*2), method='bounded')
427 |     kappa_boot.append(rb.x)

```

```

428 |
429 | kappa_boot = np.array(kappa_boot)
430 | ci_lo, ci_hi = np.percentile(kappa_boot, [2.5, 97.5])
431 | print(f"k = {kappa_best:.4e}, 95%CI = [{ci_lo:.4e}, {ci_hi:.4e}]")
432 | print(f"CI includes 0? {'YES' if ci_lo &lt;= 0 &lt;= ci_hi else 'NO'}")
433 |
434 | # =====
435 | # (2) k oooooooooo
436 | # =====
437 | print("\n" + "=" * 70)
438 | print("(2) k oooooo")
439 | print("=" * 70)
440 |
441 | # ooooo k_i = (gc_obs/gc_deep - 1) / ratio
442 | mask_ratio = ratio &gt; 0
443 | kappa_individual = np.full(N, np.nan)
444 | kappa_individual[mask_ratio] = (gc_obs[mask_ratio] / gc_deep[mask_ratio] - 1) / ratio[mask_ratio]
445 |
446 | valid = ~np.isnan(kappa_individual) & np.isfinite(kappa_individual)
447 | # outlier clip (|k_i| &gt; 100 * median)
448 | ki_valid = kappa_individual[valid]
449 | med_ki = np.median(ki_valid)
450 | mad_ki = np.median(np.abs(ki_valid - med_ki))
451 | clip = np.abs(ki_valid - med_ki) &lt; 10 * (mad_ki + 1e-30)
452 | ki_clipped = ki_valid[clip]
453 |
454 | print(f"oook_i: N={len(ki_clipped)}, median={np.median(ki_clipped):.4e}, "
455 |       f"MAD={mad_ki:.4e}, CV={mad_ki/(abs(med_ki)+1e-30):.2f}")
456 |
457 | # k_i vs galaxy parameters
458 | # (ratio &gt; 0 & clip oooooooooo)
459 | idx_valid = np.where(valid)[0]
460 | idx_valid = idx_valid[clip]
461 |
462 | params_test = {
463 |     'log(vflat)': np.log10(vflat[idx_valid]),
464 |     'log(hR)': np.log10(hR[idx_valid]),
465 |     'log(Yd)': np.log10(np.maximum(Yd[idx_valid], 0.01)),
466 |     'rmax/hR': rmax_hR[idx_valid],
467 |     'log(gc_deep)': np.log10(gc_deep[idx_valid]),
468 | }
469 |
470 | print("\nk_i vs oooooo (Spearman):")
471 | for pname, pvals in params_test.items():
472 |     finite = np.isfinite(pvals) & np.isfinite(ki_clipped)
473 |     if np.sum(finite) &lt; 10:
474 |         continue
475 |     rho, p = spearmanr(pvals[finite], ki_clipped[finite])
476 |     sig = "****" if p &lt; 0.001 else "***" if p &lt; 0.01 else "" if p &lt; 0.05 else ""
477 |     print(f" {pname:15s}: rho={rho:+.3f}, p={p:.4f} {sig}")
478 |
479 | # =====
480 | # (2b) Two-parameter model: k = k0 + k1 * log(param)
481 | # =====
482 | print("\n--- k ooooooo ---")
483 |
484 | for dep_name, dep_vals in [(('log_vflat', np.log10(vflat)),
485 |                             ('log_hR', np.log10(hR)),
486 |                             ('log_Yd', np.log10(np.maximum(Yd, 0.01))))]:
487 |     def chi2_2p(params):
488 |         k0, k1 = params
489 |         kappa_var = k0 + k1 * dep_vals
490 |         model = np.log10(np.maximum(1 + kappa_var * ratio, 1e-10))
491 |         return np.sum((log_excess - model)**2)
492 |
493 |     r2p = minimize(chi2_2p, [kappa_best, 0], method='Nelder-Mead')
494 |     dAIC_2p = r2p.fun - chi2_best + 2 # +1 extra param
495 |     print(f" k = k0 + k1*{dep_name}: k0={r2p.x[0]:.4e}, k1={r2p.x[1]:.4e}, "
496 |           f"dAIC vs const-k = {dAIC_2p:+.1f}")
497 |
498 | # =====
499 | # (3) hR ooooooo
500 | # =====
501 | print("\n" + "=" * 70)
502 | print("(3) hR oooooo")
503 | print("=" * 70)
504 |
505 | log_gc_obs = np.log10(gc_obs)
506 | log_gc_deep = np.log10(gc_deep)
507 | log_gc_corr = np.log10(np.maximum(gc_corrected, 1e-30))
508 | log_vflat = np.log10(vflat)
509 | log_hR = np.log10(hR)
510 |
511 | # ooo: p(gc, hR | vflat)
512 | from numpy.polynomial.polynomial import polyfit
513 |
514 | def partial_corr(x, y, z):
515 |     """p(x,y|z) by residualization"""
516 |     cx = np.polyfit(z, x, 1)
517 |     cy = np.polyfit(z, y, 1)
518 |     rx = x - np.polyval(cx, z)
519 |     ry = y - np.polyval(cy, z)

```

```

520 |         return spearmanr(rx, ry)
521 |
522 | rho_obs, p_obs = partial_corr(log_gc_obs, log_hR, log_vflat)
523 | rho_deep, p_deep = partial_corr(log_gc_deep, log_hR, log_vflat)
524 | rho_corr, p_corr = partial_corr(log_gc_corr, log_hR, log_vflat)
525 |
526 | print(f"p(gc_obs, hR | vflat)      = {rho_obs:+.3f} (p={p_obs:.4f})")
527 | print(f"p(gc_deep, hR | vflat)    = {rho_deep:+.3f} (p={p_deep:.4f})")
528 | print(f"p(gc_corrected, hR | vflat) = {rho_corr:+.3f} (p={p_corr:.4f})")
529 | print(f"====: |p| {abs(rho_obs):.3f} - {abs(rho_corr):.3f} "
530 |       f"({(1-abs(rho_corr)/abs(rho_obs))*100:.1f}% reduction)")
531 |
532 | # x_outer ===
533 | log_ratio = np.log10(np.maximum(ratio, 1e-30))
534 |
535 | def partial_corr_multi(target, y, controls):
536 |     """=====
537 |     from numpy.linalg import lstsq
538 |     X = np.column_stack(controls)
539 |     X = np.column_stack([X, np.ones(len(X))])
540 |     ct, _, _, _ = lstsq(X, target, rcond=None)
541 |     cy, _, _, _ = lstsq(X, y, rcond=None)
542 |     rt = target - X @ ct
543 |     ry = y - X @ cy
544 |     return spearmanr(rt, ry)
545 |
546 | rho_full, p_full = partial_corr_multi(
547 |     log_gc_corr, log_hR, [log_vflat, rmax_hR])
548 | print(f"p(gc_corrected, hR | vflat, rmax/hR) = {rho_full:+.3f} (p={p_full:.4f})")
549 |
550 | # =====
551 | # (4) =====
552 | # =====
553 | print("\n" + "-" * 70)
554 | print("(4) =====")
555 | print("=" * 70)
556 |
557 | # ratio o hR ===
558 | rho_rh, p_rh = spearmanr(ratio, hR)
559 | print(f"p(ratio, hR) = {rho_rh:+.3f} (p={p_rh:.4f})")
560 |
561 | # ratio o gc_obs ===
562 | rho_rg, p_rg = spearmanr(ratio, gc_obs)
563 | print(f"p(ratio, gc_obs) = {rho_rg:+.3f} (p={p_rg:.4f})")
564 |
565 | # ratio o hR =====?
566 | # E_grad/E_local o hR ===== (ε(r) = sqrt(gN/gc_deep))
567 | # === gN o 1/r =====
568 | rho_rh_vf, p_rh_vf = partial_corr(log_ratio, log_hR, log_vflat)
569 | print(f"p(ratio, hR | vflat) = {rho_rh_vf:+.3f} (p={p_rh_vf:.4f})")
570 | print("(E_grad/E_local o hR =====)")
571 |
572 | # =====: ratio =====
573 | np.random.seed(42)
574 | rho_shuffle = []
575 | for _ in range(1000):
576 |     ratio_s = np.random.permutation(ratio)
577 |     gc_s = gc_deep * (1 + kappa_best * ratio_s)
578 |     log_gc_s = np.log10(np.maximum(gc_s, 1e-30))
579 |     rho_s, _ = partial_corr(log_gc_s, log_hR, log_vflat)
580 |     rho_shuffle.append(rho_s)
581 | rho_shuffle = np.array(rho_shuffle)
582 | pctile = np.mean(rho_shuffle &lt;= rho_corr) * 100
583 | print(f"\n=====: real ρ={rho_corr:+.3f}, "
584 |       f"shuffle mean={np.mean(rho_shuffle):+.3f}, "
585 |       f"percentile={pctile:.1f}%")
586 |
587 | # =====
588 | # (5) =====
589 | # =====
590 | print("\n" + "-" * 70)
591 | print("(5) κ =====")
592 | print("=" * 70)
593 |
594 | print(f"====")
595 | =====: L[ε] = U(ε;c) + κ(dε/dr)2
596 |
597 | κ_best = {kappa_best:.4e}
598 | κ * median(ratio) = {kappa_best * ratio_scale:.4f}
599 | → ===== {abs(kappa_best * ratio_scale)*100:.1f}% o gc ==
600 |
601 | =====:
602 | - κ > 0: ===== gc ===== → =====
603 | - κ < 0: ===== gc =====
604 | - |κ×ratio| &lt;&lt; 1: =====
605 |
606 | == bending rigidity ==:
607 | D_bend = κ × h_membrane3 / 12(1-ν2) (====)
608 | == h_membrane =====ν =====
609 | """)
610 |
611 | # =====

```



```
704 |         ax.axvline(ci_lo, color='orange', ls='--', lw=1, label=f'95%CI')
705 |         ax.axvline(ci_hi, color='orange', ls='--', lw=1)
706 |         ax.axvline(0, color='grey', ls='--', lw=1)
707 |         ax.set_xlabel('kappa (bootstrap)')
708 |         ax.set_ylabel('count')
709 |         ax.set_title('Bootstrap distribution of kappa')
710 |         ax.legend()
711 |
712 |         plt.tight_layout()
713 |         figpath = os.path.join(BASE, 'kappa_estimation_results.png')
714 |         plt.savefig(figpath, dpi=150)
715 |         print(f"\n====: {figpath}")
716 |
717 |     except Exception as e:
718 |         print(f"\n=====: {e}")
719 |
720 |     print("\n====")
721 |
722 | if __name__ == '__main__':
723 |     main()
```

Script 2: sparc_kappa_nonlinear.py

目的

条件14 非線形膜剛性モデル8種の系統的テスト

kappa=const の否定を受け、8種の非線形モデルを AICc + L00-CV + シャッフルで系統評価。M1:

kappa=const (再現)、M2: $\kappa_0 \times \text{ratio}^n$ 、M3: epsilon閾値 (塑性/弾性二相)、M4: epsilon-modulated ratio、M5: 塑性指標 Φ 、M6: $u_{\text{strain}}/E_{\text{kin}}$ 、M7: 勾配非一様性、M8: epsilon曲率 $d^2 \text{epsilon}/dr^2$ 。過適合制御は AICc (小サンプル補正) + L00-CV (全銀河) + シャッフル順列 (1000回) の3重。

結果

M3 (epsilon閾値, dAICc=-5.1, L00=0.351) が最良。epsilon_c=0.072,

beta=-0.23。M6 (dAICc=-5.7)、M8 (dAICc=-5.6) も有意だが、後に M3 と冗長と判明。M1/M4/M5/M7 は不支持。

実行方法

```
uv run --with scipy --with matplotlib python sparc_kappa_nonlinear.py
```

入力データ: sparc_gc.csv + Rotmod_LTG/*.dat

出力: kappa_nonlinear_results.png + 標準出力

ソースコード全文

```
1 | #!/usr/bin/env python3
2 | """
3 | sparc_kappa_nonlinear.py
4 | =====
5 | 0014 0000: 0000000000000000
6 |
7 | 00:
8 | - kappa=const 000 (dAIC=+1.1, 0.6%00, 000000)
9 | - E_grad/E_local ratio 0 18% (B-0) 0000
10 | - 00: ratio -> K 0000000000
11 |
12 | 00: K 000000000000 0 00000
13 | - 00/000000 (0014) 0000000
14 |
15 | 000000:
16 | M0: gc = gc_deep (null, 0 param)
17 | M1: gc = gc_deep * (1 + kappa_0 * ratio) (const kappa, 1 param) [00]
18 | M2: gc = gc_deep * (1 + kappa_0 * ratio^n) (power-law, 2 param)
19 | M3: gc = gc_deep * f(epsilon_max) (epsilon, 2 param)
20 | M4: gc = gc_deep * (1 + kappa_0 * ratio * g(epsilon)) (epsilon-modulated, 2 param)
21 | M5: gc = gc_deep * (1 + kappa_0 * Phi_plastic) (0000, 1 param)
22 | M6: gc = gc_deep * h(E_strain/E_kinetic) (00-000, 1-2 param)
23 |
24 | 00000:
25 | - AICc (000000AIC)
26 | - L00-CV (leave-one-out cross-validation)
27 | - Shuffle permutation (10000)
28 | - hR0000000
29 |
30 | 00: uv run --with scipy --with matplotlib python sparc_kappa_nonlinear.py
31 | """
32 |
33 | import os, sys
34 | import numpy as np
35 | from scipy.optimize import minimize, minimize_scalar
36 | from scipy.stats import spearmanr
37 | import warnings
38 | warnings.filterwarnings('ignore')
39 |
40 | # =====
41 | # 0. 00000
42 | # =====
43 | BASE = r"D:\000000\000000\00000\000000\00000"
44 | ROTMOD = os.path.join(BASE, "Rotmod_LTG")
45 | GC_CSV = os.path.join(BASE, "sparc_gc.csv")
46 | a0 = 1.2e-10 # m/s^2
47 |
48 | # =====
49 | # 1. 000000 (000000000)
50 | # =====
51 | def load_gc_csv():
52 |     import csv
53 |     data = {}
54 |     with open(GC_CSV, 'r') as f:
55 |         reader = csv.DictReader(f)
56 |         for row in reader:
57 |             name = None
```

```

58 |         for k in row:
59 |             if 'gal' in k.lower() or 'name' in k.lower():
60 |                 name = row[k].strip(); break
61 |
62 |         if not name: continue
63 |         gc_val = None
64 |         for k in ['gc', 'gc_obs', 'g_c', 'gc_a0']:
65 |             if k in row and row[k]:
66 |                 try: gc_val = float(row[k]); break
67 |                 except: pass
68 |         if gc_val is None: continue
69 |         vflat, hR, Yd = None, None, None
70 |         for k in ['vflat', 'vflat', 'v_flat']:
71 |             if k in row and row[k]:
72 |                 try: vflat = float(row[k]); break
73 |                 except: pass
74 |         for k in ['hR', 'Reff', 'h_R', 'hr']:
75 |             if k in row and row[k]:
76 |                 try: hR = float(row[k]); break
77 |                 except: pass
78 |         for k in ['Yd', 'yd', 'Y_d', 'SPS']:
79 |             if k in row and row[k]:
80 |                 try: Yd = float(row[k]); break
81 |                 except: pass
82 |         data[name] = {'gc': gc_val, 'vflat': vflat, 'hR': hR, 'Yd': Yd}
83 |     gc_vals = [d['gc'] for d in data.values()]
84 |     if np.median(gc_vals) > 1e-5:
85 |         for name in data: data[name]['gc'] *= a0
86 |     print(f"sparc_gc.csv: {len(data)} galaxies")
87 |     return data
88 |
89 | def load_rotcurve(galaxy_name):
90 |     fname = os.path.join(ROTMOD, f"{galaxy_name}_rotmod.dat")
91 |     if not os.path.exists(fname): return None
92 |     rad, vobs, vgas, vdisk, vbul = [], [], [], [], []
93 |     with open(fname, 'r') as f:
94 |         for line in f:
95 |             line = line.strip()
96 |             if not line or line.startswith('#'): continue
97 |             parts = line.split()
98 |             if len(parts) < 6: continue
99 |             try:
100 |                 rad.append(float(parts[0]))
101 |                 vobs.append(float(parts[1]))
102 |                 vgas.append(float(parts[3]))
103 |                 vdisk.append(float(parts[4]))
104 |                 vbul.append(float(parts[5]))
105 |             except: continue
106 |     if len(rad) < 5: return None
107 |     return {k: np.array(v) for k, v in
108 |            zip(['r', 'vobs', 'vgas', 'vdisk', 'vbul'], [rad, vobs, vgas, vdisk, vbul])}
109 |
110 | def compute_gc_deep(rc, Yd=0.5):
111 |     r_m = rc['r'] * 3.086e19
112 |     v_bar2 = (Yd * rc['vdisk']**2 + rc['vgas']**2 + rc['vbul']**2) * 1e6
113 |     v_obs2 = (rc['vobs'] * 1e3)**2
114 |     gN = v_bar2 / r_m
115 |     gobs = v_obs2 / r_m
116 |     mask = gN > 0
117 |     gc_pts = gobs[mask]**2 / gN[mask]
118 |     if len(gc_pts) < 3: return None
119 |     gc_med = np.median(gc_pts)
120 |     if gc_med <= 0: return None
121 |     x = gN[mask] / gc_med
122 |     deep = x < 1.0
123 |     return np.median(gc_pts[deep]) if np.sum(deep) >= 3 else gc_med
124 |
125 | # =====
126 | # 2. =====
127 | # =====
128 | def compute_extended_strain(rc, gc_deep, Yd=0.5):
129 |     """
130 |     radial profile
131 |     """
132 |     r_kpc = rc['r']
133 |     r_m = r_kpc * 3.086e19
134 |     v_bar2 = (Yd * rc['vdisk']**2 + rc['vgas']**2 + rc['vbul']**2) * 1e6
135 |     v_obs2 = (rc['vobs'] * 1e3)**2
136 |     gN = np.maximum(v_bar2 / r_m, 0)
137 |     gobs = v_obs2 / r_m
138 |
139 |     epsilon = np.sqrt(gN / gc_deep)
140 |
141 |     # de/dr
142 |     deps_dr = np.zeros_like(epsilon)
143 |     deps_dr[0] = (epsilon[1] - epsilon[0]) / (r_m[1] - r_m[0])
144 |     deps_dr[-1] = (epsilon[-1] - epsilon[-2]) / (r_m[-1] - r_m[-2])
145 |     for i in range(1, len(epsilon)-1):
146 |         deps_dr[i] = (epsilon[i+1] - epsilon[i-1]) / (r_m[i+1] - r_m[i-1])
147 |
148 |     E_grad = np.mean(deps_dr**2)
149 |     E_local = np.mean(epsilon**2)
150 |     ratio = E_grad / E_local if E_local > 0 else 0

```

```

150 |
151 | # --- oooo ---
152 | eps_max = np.max(epsilon)
153 | eps_mean = np.mean(epsilon)
154 | eps_outer = np.mean(epsilon[-max(1, len(epsilon)//3):]) # oo1/3ooo
155 |
156 | # oooo: ε &gt; ε_C (oo) ooooo
157 | # ε_C oooooooooo
158 |
159 | # oooooooooo u_strain o ε2 (oo) Or ε3 (ooo)
160 | u_strain_elastic = np.mean(epsilon**2)
161 | u_strain_cubic = np.mean(epsilon**3)
162 |
163 | # ooooooo proxy: v_obs2
164 | E_kin = np.mean(v_obs2) # (m/s)2
165 |
166 | # oo-ooo
167 | strain_kin_ratio = u_strain_elastic / (E_kin / (3.086e19)**2) if E_kin &gt; 0 else 0
168 |
169 | # εoooooooo: std(dε/dr) / mean(|dε/dr|)
170 | grad_nonuniform = np.std(deps_dr) / (np.mean(np.abs(deps_dr)) + 1e-50)
171 |
172 | # ε profile ooo: d2ε/dr2
173 | if len(epsilon) &gt;= 5:
174 |     d2eps = np.gradient(np.gradient(epsilon, r_m), r_m)
175 |     curvature = np.mean(d2eps**2)
176 | else:
177 |     curvature = 0
178 |
179 | return {
180 |     'ratio': ratio,
181 |     'E_grad': E_grad,
182 |     'E_local': E_local,
183 |     'eps_max': eps_max,
184 |     'eps_mean': eps_mean,
185 |     'eps_outer': eps_outer,
186 |     'epsilon_profile': epsilon,
187 |     'u_elastic': u_strain_elastic,
188 |     'u_cubic': u_strain_cubic,
189 |     'strain_kin': strain_kin_ratio,
190 |     'grad_nonuniform': grad_nonuniform,
191 |     'curvature': curvature,
192 |     'n_points': len(epsilon),
193 | }
194 |
195 | # =====
196 | # 3. oooooooooo
197 | # =====
198 | def build_dataset():
199 |     gc_data = load_gc_csv()
200 |     results = []
201 |     for gname in sorted(gc_data.keys()):
202 |         gd = gc_data[gname]
203 |         gc_obs = gd['gc']
204 |         vflat, hR, Yd = gd.get('vflat'), gd.get('hR'), gd.get('Yd') or 0.5
205 |         if gc_obs &lt;= 0 or not vflat or not hR or vflat &lt;= 0 or hR &lt;= 0:
206 |             continue
207 |         rc = load_rotcurve(gname)
208 |         if rc is None: continue
209 |         gc_deep = compute_gc_deep(rc, Yd=Yd)
210 |         if gc_deep is None or gc_deep &lt;= 0: continue
211 |         sq = compute_extended_strain(rc, gc_deep, Yd=Yd)
212 |         if sq is None or sq['ratio'] == 0: continue
213 |         results.append({
214 |             'name': gname, 'gc_obs': gc_obs, 'gc_deep': gc_deep,
215 |             'vflat': vflat, 'hR': hR, 'Yd': Yd,
216 |             **{k: sq[k] for k in sq if k != 'epsilon_profile'},
217 |         })
218 |     print(f"Dataset: {len(results)} galaxies\n")
219 |     return results
220 |
221 | # =====
222 | # 4. ooooo + oooooooooo
223 | # =====
224 | def aicc(chi2, n, k):
225 |     """Small-sample corrected AIC"""
226 |     aic = chi2 + 2*k
227 |     if n - k - 1 &gt; 0:
228 |         return aic + 2*k*(k+1) / (n - k - 1)
229 |     return aic
230 |
231 | def loo_cv(log_excess, predictor_func, param_fitter, N):
232 |     """Leave-one-out cross-validation: returns mean squared error"""
233 |     errors = []
234 |     for i in range(N):
235 |         mask = np.ones(N, dtype=bool)
236 |         mask[i] = False
237 |         params = param_fitter(mask)
238 |         if params is None:
239 |             continue
240 |         pred = predictor_func(i, params)
241 |         errors.append((log_excess[i] - pred)**2)

```

```

242 |     return np.mean(errors) if errors else 999
243 |
244 | class ModelTester:
245 |     def __init__(self, data):
246 |         self.N = len(data)
247 |         self.gc_obs = np.array([d['gc_obs'] for d in data])
248 |         self.gc_deep = np.array([d['gc_deep'] for d in data])
249 |         self.log_excess = np.log10(self.gc_obs / self.gc_deep)
250 |         self.ratio = np.array([d['ratio'] for d in data])
251 |         self.eps_max = np.array([d['eps_max'] for d in data])
252 |         self.eps_mean = np.array([d['eps_mean'] for d in data])
253 |         self.eps_outer = np.array([d['eps_outer'] for d in data])
254 |         self.u_elastic = np.array([d['u_elastic'] for d in data])
255 |         self.u_cubic = np.array([d['u_cubic'] for d in data])
256 |         self.strain_kin = np.array([d['strain_kin'] for d in data])
257 |         self.grad_nonuniform = np.array([d['grad_nonuniform'] for d in data])
258 |         self.curvature = np.array([d['curvature'] for d in data])
259 |         self.vflat = np.array([d['vflat'] for d in data])
260 |         self.hR = np.array([d['hR'] for d in data])
261 |         self.rmax_hR = np.array([d.get('rmax_hR', 0) for d in data])
262 |         self.names = [d['name'] for d in data]
263 |
264 |         # null chi2
265 |         self.chi2_null = np.sum(self.log_excess**2)
266 |         self.mse_null = np.mean(self.log_excess**2)
267 |
268 |         print(f"log(gc_obs/gc_deep): mean={np.mean(self.log_excess):.4f}, "
269 |               f"std={np.std(self.log_excess):.4f}")
270 |
271 |     def test_model(self, name, n_params, fit_func, predict_func,
272 |                   shuffle_test=True, loo=True):
273 |         """
274 |         oooooooooo
275 |         fit_func(mask=None) -> params (mask=Noneoooooooo)
276 |         predict_func(params) -> array of model predictions (log10 scale)
277 |         """
278 |         print(f"\n{'-'*60}")
279 |         print(f"Model: {name} ({n_params} params)")
280 |         print(f"{'-'*60}")
281 |
282 |         # Fit on all data
283 |         params = fit_func()
284 |         if params is None:
285 |             print(" FIT FAILED")
286 |             return None
287 |
288 |         pred = predict_func(params)
289 |         resid = self.log_excess - pred
290 |         chi2 = np.sum(resid**2)
291 |         R2 = 1 - chi2 / self.chi2_null
292 |         _aicc = aicc(chi2, self.N, n_params)
293 |         _aicc_null = aicc(self.chi2_null, self.N, 0)
294 |         dAICc = _aicc - _aicc_null
295 |
296 |         print(f"  params: {params}")
297 |         print(f"  chi2={chi2:.3f}, R2={R2:.4f}, dAICc={dAICc:+.1f}")
298 |         print(f"  resid std={np.std(resid):.4f} (null={np.std(self.log_excess):.4f}")
299 |
300 |         # hR partial correlation
301 |         log_vf = np.log10(self.vflat)
302 |         log_hR = np.log10(self.hR)
303 |         gc_model = self.gc_deep * 10**pred
304 |         log_gc_m = np.log10(np.maximum(gc_model, 1e-30))
305 |
306 |         def pcorr(x, y, z):
307 |             cx = np.polyfit(z, x, 1)
308 |             cy = np.polyfit(z, y, 1)
309 |             return spearmanr(x - np.polyval(cx, z), y - np.polyval(cy, z))
310 |
311 |         rho_obs, _ = pcorr(np.log10(self.gc_obs), log_hR, log_vf)
312 |         rho_model, p_model = pcorr(log_gc_m, log_hR, log_vf)
313 |
314 |         print(f"  hRooo: gc_obs={rho_obs:+.3f}, model={rho_model:+.3f} "
315 |               f"({(1-abs(rho_model)/abs(rho_obs))*100:.1f}%)")
316 |
317 |         # Shuffle test
318 |         if shuffle_test:
319 |             np.random.seed(42)
320 |             better = 0
321 |             for _ in range(1000):
322 |                 idx = np.random.permutation(self.N)
323 |                 le_s = self.log_excess[idx]
324 |                 chi2_s = np.sum((le_s - pred)**2)
325 |                 if chi2_s < chi2:
326 |                     better += 1
327 |             print(f"  Shuffle: {better}/1000 better (p={better/1000:.3f}")
328 |
329 |         # LOO-CV
330 |         if loo and self.N <= 200:
331 |             loo_errors = []
332 |             for i in range(self.N):
333 |                 mask = np.ones(self.N, dtype=bool)

```

```

334 |         mask[i] = False
335 |         p_loo = fit_func(mask)
336 |         if p_loo is None: continue
337 |         pred_i = predict_func(p_loo, idx=i)
338 |         loo_errors.append((self.log_excess[i] - pred_i)**2)
339 |     mse_loo = np.mean(loo_errors)
340 |     print(f" L00-CV MSE={mse_loo:.6f} (null={self.mse_null:.6f}, "
341 |           f"ratio={mse_loo/self.mse_null:.4f})")
342 |
343 |     return {
344 |         'name': name, 'n_params': n_params, 'chi2': chi2,
345 |         'R2': R2, 'dAICc': dAICc, 'rho_hr': rho_model,
346 |         'params': params, 'resid_std': np.std(resid),
347 |     }
348 |
349 | # =====
350 | # M1:  $\kappa$  const (□□)
351 | # =====
352 | def test_M1(self):
353 |     ratio = self.ratio
354 |     le = self.log_excess
355 |     def fit(mask=None):
356 |         r, e = (ratio, le) if mask is None else (ratio[mask], le[mask])
357 |         def chi2(k):
358 |             return np.sum((e - np.log10(np.maximum(1+k*r, 1e-10)))**2)
359 |         rsc = np.median(r)
360 |         if rsc > 0:
361 |             km = 5.0/rsc
362 |         else:
363 |             km = 1e40
364 |         res = minimize_scalar(chi2, bounds=(-km, km), method='bounded')
365 |         return [res.x]
366 |     def pred(params, idx=None):
367 |         k = params[0]
368 |         r = ratio if idx is None else np.array([ratio[idx]])
369 |         return np.log10(np.maximum(1 + k*r, 1e-10))
370 |     return self.test_model("M1:  $\kappa$ =const", 1, fit, pred)
371 |
372 | # =====
373 | # M2:  $\kappa \times \text{ratio}^n$  (power-law□□□)
374 | # =====
375 | def test_M2(self):
376 |     ratio = self.ratio
377 |     le = self.log_excess
378 |     log_ratio = np.log10(np.maximum(ratio, 1e-50))
379 |
380 |     def fit(mask=None):
381 |         r, e = (ratio, le) if mask is None else (ratio[mask], le[mask])
382 |         lr = np.log10(np.maximum(r, 1e-50))
383 |         def chi2(p):
384 |             k0, n = p
385 |             val = k0 * np.abs(r)**n * np.sign(r)
386 |             return np.sum((e - np.log10(np.maximum(1 + val, 1e-10)))**2)
387 |         best = None
388 |         for n0 in [0.3, 0.5, 1.0, 1.5, 2.0]:
389 |             rsc = np.median(r)
390 |             k0_init = 0.01 / (np.median(np.abs(r))**n0 + 1e-50)
391 |             for k_sign in [1, -1]:
392 |                 res = minimize(chi2, [k_sign*k0_init, n0], method='Nelder-Mead',
393 |                               options={'maxiter': 5000, 'xatol': 1e-10})
394 |                 if best is None or res.fun < best.fun:
395 |                     best = res
396 |             return list(best.x) if best else None
397 |
398 |     def pred(params, idx=None):
399 |         k0, n = params
400 |         r = ratio if idx is None else np.array([ratio[idx]])
401 |         val = k0 * np.abs(r)**n * np.sign(r)
402 |         return np.log10(np.maximum(1 + val, 1e-10))
403 |
404 |     return self.test_model("M2:  $\kappa \times \text{ratio}^n$ ", 2, fit, pred)
405 |
406 | # =====
407 | # M3:  $\epsilon_{\text{max}} / \epsilon_{\text{c}}$  (□□/□□□□)
408 | # gc = gc_deep  $\times$  (1 + A  $\times$  f_plastic)
409 | # f_plastic = fraction of points with  $\epsilon$  >  $\epsilon_{\text{c}}$ 
410 | # =====
411 | def test_M3(self):
412 |     eps_profiles = self.eps_mean # use per-galaxy  $\epsilon_{\text{mean}}$  as proxy
413 |     eps_max = self.eps_max
414 |     le = self.log_excess
415 |
416 |     # f_plastic( $\epsilon_{\text{c}}$ ) = fraction with  $\epsilon$  >  $\epsilon_{\text{c}}$  → use eps_max vs threshold
417 |     # □□□: eps_max >  $\epsilon_{\text{c}}$  □□ "plastic-dominated"
418 |     # gc = gc_deep  $\times$  ( $\epsilon_{\text{max}} / \epsilon_{\text{c}}$ ) $^{\beta}$  for  $\epsilon_{\text{max}} > \epsilon_{\text{c}}$ , else gc = gc_deep
419 |
420 |     def fit(mask=None):
421 |         em = eps_max if mask is None else eps_max[mask]
422 |         e = le if mask is None else le[mask]
423 |         lem = np.log10(np.maximum(em, 1e-10))
424 |
425 |     def chi2(p):

```

```

426 |         log_ec, beta = p
427 |         excess = np.maximum(lem - log_ec, 0) # max(0, log(eps_max/eps_c))
428 |         model = beta * excess
429 |         return np.sum((e - model)**2)
430 |
431 |     best = None
432 |     for lec in np.linspace(np.percentile(lem, 10), np.percentile(lem, 90), 10):
433 |         for b in [-1, -0.5, 0.5, 1]:
434 |             res = minimize(chi2, [lec, b], method='Nelder-Mead',
435 |                             options={'maxiter': 3000})
436 |             if best is None or res.fun < best.fun:
437 |                 best = res
438 |     return list(best.x) if best else None
439 |
440 |     def pred(params, idx=None):
441 |         log_ec, beta = params
442 |         em = eps_max if idx is None else np.array([eps_max[idx]])
443 |         lem = np.log10(np.maximum(em, 1e-10))
444 |         excess = np.maximum(lem - log_ec, 0)
445 |         return beta * excess
446 |
447 |     return self.test_model("M3:  $\epsilon_0(\sigma/\sigma_0)$ ", 2, fit, pred)
448 |
449 | # =====
450 | # M4:  $\epsilon$ -modulated ratio
451 | #  $gc = gc_{deep} \times (1 + \kappa_0 \times ratio \times (\epsilon_{mean}/\epsilon_{ref})^m)$ 
452 | # =====
453 |     def test_M4(self):
454 |         ratio = self.ratio
455 |         eps_mean = self.eps_mean
456 |         le = self.log_excess
457 |
458 |     def fit(mask=None):
459 |         r = ratio if mask is None else ratio[mask]
460 |         em = eps_mean if mask is None else eps_mean[mask]
461 |         e = le if mask is None else le[mask]
462 |
463 |         eps_ref = np.median(em)
464 |
465 |     def chi2(p):
466 |         k0, m = p
467 |         mod = k0 * r * (em / eps_ref)**m
468 |         return np.sum((e - np.log10(np.maximum(1 + mod, 1e-10)))**2)
469 |
470 |         rsc = np.median(r)
471 |         k_init = 0.01 / (rsc + 1e-50)
472 |         best = None
473 |         for m0 in [-2, -1, 0, 1, 2]:
474 |             for ks in [1, -1]:
475 |                 res = minimize(chi2, [ks*k_init, m0], method='Nelder-Mead',
476 |                                 options={'maxiter': 5000})
477 |                 if best is None or res.fun < best.fun:
478 |                     best = res
479 |     return list(best.x) if best else None
480 |
481 |     def pred(params, idx=None):
482 |         k0, m = params
483 |         r = ratio if idx is None else np.array([ratio[idx]])
484 |         em = eps_mean if idx is None else np.array([eps_mean[idx]])
485 |         eps_ref = np.median(self.eps_mean)
486 |         mod = k0 * r * (em / eps_ref)**m
487 |         return np.log10(np.maximum(1 + mod, 1e-10))
488 |
489 |     return self.test_model("M4:  $\kappa_0 \times ratio \times (\epsilon/\epsilon_{ref})^m$ ", 2, fit, pred)
490 |
491 | # =====
492 | # M5:  $\sigma^3$  ( $\sigma_{14}$ )
493 | #  $\sigma \epsilon$   $gc$ 
494 | #  $gc = gc_{deep} \times (1 + A \times f_{high\_strain})$ 
495 | #  $f_{high\_strain} = mean[\epsilon^2 \times I(\epsilon > median_\epsilon)]$ 
496 | # =====
497 |     def test_M5(self):
498 |         #  $\sigma^3 \epsilon$   $\sigma^3$ :  $u_{cubic} / u_{elastic} = \epsilon^3 / \epsilon^2 = \epsilon$ ;  $\epsilon$ ;  $\epsilon_{weighted}$ 
499 |         #  $\rightarrow \sigma \epsilon$ 
500 |         plastic_indicator = self.u_cubic / (self.u_elastic + 1e-50)
501 |         le = self.log_excess
502 |
503 |     def fit(mask=None):
504 |         pi = plastic_indicator if mask is None else plastic_indicator[mask]
505 |         e = le if mask is None else le[mask]
506 |         log_pi = np.log10(np.maximum(pi, 1e-10))
507 |
508 |     def chi2(p):
509 |         a, = p
510 |         model = a * log_pi
511 |         return np.sum((e - model)**2)
512 |
513 |     # linear regression
514 |     from numpy.linalg import lstsq
515 |     X = log_pi.reshape(-1, 1)
516 |     sol, _, _, _ = lstsq(X, e, rcond=None)
517 |     return [sol[0]]

```

```

518 |
519 |     def pred(params, idx=None):
520 |         a = params[0]
521 |         pi = plastic_indicator if idx is None else np.array([plastic_indicator[idx]])
522 |         return a * np.log10(np.maximum(pi, 1e-10))
523 |
524 |     return self.test_model("M5:  $\epsilon^3$  & /&  $\epsilon^2$ ", 1, fit, pred)
525 |
526 | # =====
527 | # M6:  $\sigma_{\text{kin}} / \sigma_{\text{excess}}$ 
528 | # =====
529 |     def test_M6(self):
530 |         sk = self.strain_kin
531 |         le = self.log_excess
532 |
533 |         def fit(mask=None):
534 |             s = sk if mask is None else sk[mask]
535 |             e = le if mask is None else le[mask]
536 |             log_s = np.log10(np.maximum(s, 1e-50))
537 |             finite = np.isfinite(log_s)
538 |             if np.sum(finite) < 10: return None
539 |
540 |             from numpy.linalg import lstsq
541 |             X = log_s[finite].reshape(-1, 1)
542 |             sol, _, _, _ = lstsq(X, e[finite], rcond=None)
543 |             return [sol[0]]
544 |
545 |         def pred(params, idx=None):
546 |             a = params[0]
547 |             s = sk if idx is None else np.array([sk[idx]])
548 |             return a * np.log10(np.maximum(s, 1e-50))
549 |
550 |     return self.test_model("M6:  $\sigma_{\text{strain}} / E_{\text{kin}}$ ", 1, fit, pred)
551 |
552 | # =====
553 | # M7:  $\sigma_{\text{kin}} / \sigma_{\text{excess}}$ 
554 | #  $K \sigma_{\text{kin}} / \sigma_{\text{excess}}$  GC
555 | # =====
556 |     def test_M7(self):
557 |         gn = self.grad_nonuniform
558 |         le = self.log_excess
559 |
560 |         def fit(mask=None):
561 |             g = gn if mask is None else gn[mask]
562 |             e = le if mask is None else le[mask]
563 |             log_g = np.log10(np.maximum(g, 1e-10))
564 |             finite = np.isfinite(log_g)
565 |             if np.sum(finite) < 10: return None
566 |             from numpy.linalg import lstsq
567 |             X = log_g[finite].reshape(-1, 1)
568 |             sol, _, _, _ = lstsq(X, e[finite], rcond=None)
569 |             return [sol[0]]
570 |
571 |         def pred(params, idx=None):
572 |             a = params[0]
573 |             g = gn if idx is None else np.array([gn[idx]])
574 |             return a * np.log10(np.maximum(g, 1e-10))
575 |
576 |     return self.test_model("M7:  $\sigma_{\text{kin}} / \sigma_{\text{excess}}$ ", 1, fit, pred)
577 |
578 | # =====
579 | # M8:  $\epsilon_{\text{curv}} (d^2\epsilon/dr^2)$ 
580 | # =====
581 |     def test_M8(self):
582 |         curv = self.curvature
583 |         le = self.log_excess
584 |
585 |         def fit(mask=None):
586 |             c = curv if mask is None else curv[mask]
587 |             e = le if mask is None else le[mask]
588 |             log_c = np.log10(np.maximum(c, 1e-80))
589 |             finite = np.isfinite(log_c)
590 |             if np.sum(finite) < 10: return None
591 |             from numpy.linalg import lstsq
592 |             X = log_c[finite].reshape(-1, 1)
593 |             sol, _, _, _ = lstsq(X, e[finite], rcond=None)
594 |             return [sol[0]]
595 |
596 |         def pred(params, idx=None):
597 |             a = params[0]
598 |             c = curv if idx is None else np.array([curv[idx]])
599 |             return a * np.log10(np.maximum(c, 1e-80))
600 |
601 |     return self.test_model("M8:  $\epsilon_{\text{curv}} d^2\epsilon/dr^2$ ", 1, fit, pred)
602 |
603 | # =====
604 | # 5.  $\sigma_{\text{kin}} / \sigma_{\text{excess}}$ 
605 | # =====
606 |     def main():
607 |         print("=" * 70)
608 |         print("14  $\sigma_{\text{kin}} / \sigma_{\text{excess}}$ ")
609 |         print("=" * 70)

```

```

610 |
611 | data = build_dataset()
612 | mt = ModelTester(data)
613 |
614 | results = []
615 | results.append(mt.test_M1()) #  $\kappa$  const (baseline)
616 | results.append(mt.test_M2()) #  $\kappa_0 \times \text{ratio}^n$ 
617 | results.append(mt.test_M3()) #  $\epsilon_{00}$ 
618 | results.append(mt.test_M4()) #  $\epsilon$ -modulated ratio
619 | results.append(mt.test_M5()) #  $\epsilon_{000}$ 
620 | results.append(mt.test_M6()) #  $u_{\text{strain}}/E_{\text{kin}}$ 
621 | results.append(mt.test_M7()) #  $\epsilon_{00000}$ 
622 | results.append(mt.test_M8()) #  $\epsilon_{00}$ 
623 |
624 | results = [r for r in results if r is not None]
625 |
626 | # =====
627 | #  $\epsilon_{00}$ 
628 | # =====
629 | print("\n" + "-" * 70)
630 | print("#####")
631 | print("-" * 70)
632 | print(f"{'Model':&lt;&35s} {'k':&lt;&2s} {'dAICc':&lt;&7s} {'R2':&lt;&7s} "
633 |       f"{' $\rho$ (hR)':&lt;&7s} {'std':&lt;&7s}")
634 | print("-" * 70)
635 | print(f"{'M0: null (gc=gc_deep)':&lt;&35s} {' $\theta$ ':&lt;&2s} {' $\theta_0$ ':&lt;&7s} {' $\theta_0$ ':&lt;&7s} "
636 |       f"{'--':&lt;&7s} {'np.std(mt.log_excess):&lt;&7.4f}")
637 | for r in sorted(results, key=lambda x: x['dAICc']):
638 |     print(f"{'r[name]':&lt;&35s} {'r[n_params]':&lt;&2d} {'r['dAICc']':&lt;&7.1f} "
639 |           f"{'r[R2']':&lt;&7.4f} {'r[rho_hR]':&lt;&7.3f} {'r[resid_std]':&lt;&7.4f}")
640 |
641 | #  $\epsilon_{0000}$ 
642 | best = min(results, key=lambda x: x['dAICc'])
643 | print(f"\n#####: {best['name']} (dAICc={best['dAICc']:+.1f})")
644 |
645 | if best['dAICc'] &gt;= 0:
646 |     print("-> null##### (gc = gc_deep) #####")
647 | elif best['dAICc'] &gt; -2:
648 |     print("-> #####")
649 | elif best['dAICc'] &gt; -6:
650 |     print("-> #####")
651 | else:
652 |     print("-> #####")
653 |
654 | # =====
655 | #  $\epsilon_{00}$ 
656 | # =====
657 | print("\n" + "-" * 70)
658 | print("#####")
659 | print("-" * 70)
660 | print("")
661 | E_grad/E_local  $\epsilon_{00}$ 18% (B- $\epsilon_{00}$ ) #####
662 |  $\epsilon_{00}$ #####
663 |
664 | #####:
665 | dAICc &lt; -6 -> #####
666 | dAICc &lt; -2 -> #####
667 | dAICc &gt;= 0 -> #####
668 |
669 | #####:
670 | -> E_grad/E_local #####
671 | ->  $\epsilon_{00}$ 14 $\epsilon_{00}$  $\rho_{\text{eff}} = u_{\text{strain}}/c^2$ #####
672 | -> #####
673 |
674 | #####:
675 | ->  $\epsilon_{00}$ 14#####
676 | ->  $\kappa(\epsilon)$  #####
677 | """)
678 |
679 | # =====
680 | #  $\epsilon_{00}$ 
681 | # =====
682 | try:
683 |     import matplotlib
684 |     matplotlib.use('Agg')
685 |     import matplotlib.pyplot as plt
686 |
687 |     fig, axes = plt.subplots(2, 2, figsize=(12, 10))
688 |     fig.suptitle('Non-linear membrane stiffness models', fontsize=14)
689 |
690 |     # (a) dAICc comparison
691 |     ax = axes[0, 0]
692 |     names = [r['name'].split(':')[0] for r in results]
693 |     daics = [r['dAICc'] for r in results]
694 |     colors = ['green' if d &lt; -6 else 'orange' if d &lt; -2 else 'red' for d in daics]
695 |     ax.barh(range(len(names)), daics, color=colors, alpha=0.7)
696 |     ax.set_yticks(range(len(names)))
697 |     ax.set_yticklabels(names, fontsize=8)
698 |     ax.axvline(0, color='black', ls='--', lw=1)
699 |     ax.axvline(-2, color='grey', ls=':', lw=1)
700 |     ax.set_xlabel('dAICc vs null')
701 |     ax.set_title('Model comparison')

```

```

702 |
703 |     # (b) R2 comparison
704 |     ax = axes[0, 1]
705 |     r2s = [r['R2'] for r in results]
706 |     ax.barh(range(len(names)), r2s, color='steelblue', alpha=0.7)
707 |     ax.set_yticks(range(len(names)))
708 |     ax.set_yticklabels(names, fontsize=8)
709 |     ax.set_xlabel('R2')
710 |     ax.set_title('Variance explained')
711 |
712 |     # (c) hR partial correlation
713 |     ax = axes[1, 0]
714 |     rhos = [abs(r['rho_hR']) for r in results]
715 |     ax.barh(range(len(names)), rhos, color='coral', alpha=0.7)
716 |     rho_obs_val = abs(results[0]['rho_hR']) if results else 0.35
717 |     ax.axvline(rho_obs_val, color='black', ls='--', lw=1, label='gc_obs baseline')
718 |     ax.set_yticks(range(len(names)))
719 |     ax.set_yticklabels(names, fontsize=8)
720 |     ax.set_xlabel('|ρ(gc_model, hR | vflat)|')
721 |     ax.set_title('hR partial correlation (lower=better)')
722 |     ax.legend(fontsize=8)
723 |
724 |     # (d) scatter: best model prediction vs excess
725 |     ax = axes[1, 1]
726 |     if best:
727 |         ax.scatter(mt.log_excess, np.zeros(mt.N), s=10, alpha=0.3, label='null residual')
728 |         pred_best = best.get('params')
729 |         ax.set_xlabel('log10(gc_obs/gc_deep)')
730 |         ax.set_ylabel('model prediction')
731 |         ax.set_title(f'Best model: {best["name"]}')
```

```

732 |
733 |     plt.tight_layout()
734 |     figpath = os.path.join(BASE, 'kappa_nonlinear_results.png')
735 |     plt.savefig(figpath, dpi=150)
736 |     print(f"\n====: {figpath}")
737 |     except Exception as e:
738 |         print(f"\n=====: {e}")
739 |
740 |     print("\n====")
741 |
742 | if __name__ == '__main__':
743 |     main()
```

Script 3: sparc_kappa_circularity.py

目的

M3/M6/M8 の $R^2=0.55$ が gc_deep 共有ノイズのアーティファクトでないかの検証

$\epsilon = \sqrt{gN/gc_deep}$ なので gc_deep が予測変数と応答変数の両方に共有される。gc_deep 推定ノイズが見かけの R^2 を生む可能性を 5 段階で検証。T1: Split-half gc_deep (データ点を2分割、独立に gc_deep 推定、交差使用)。T2: gc_deep シャッフル (銀河間でシャッフル、1000回)。T3: gc_obs 直接予測 (gc_deep を介さない proxy で予測)。T4: gc_deep 攪乱 ($\pm 0.05 \sim 0.5$ dex ノイズ追加)。T5: log_excess vs log(strain_kin) の代数的関係分析。

結果

真の物理信号と確認。T1: split-half で R^2 が 93-96% 維持。T2: シャッフルで完全消失 (mean=0.117, p=0.0000)。T3: 幾何平均法則を超える独立情報 ($dR^2=+0.119$)。

実行方法

```
uv run --with scipy --with matplotlib python sparc_kappa_circularity.py
```

入力データ: sparc_gc.csv + Rotmod_LTG/*.dat

出力: 標準出力 (図なし)

ソースコード全文

```
1 | #!/usr/bin/env python3
2 | """
3 | sparc_kappa_circularity.py
4 | =====
5 | M3/M6/M8  $R^2=0.55$   $\epsilon$ ? gc_deep
6 |
7 |  $\epsilon$ :
8 |  $y = \log(gc\_obs / gc\_deep)$ 
9 |  $x = f(\epsilon)$ ,  $\epsilon = \sqrt{gN / gc\_deep}$ 
10 |  $\rightarrow gc\_deep \propto y \propto x$ 
11 |  $\rightarrow gc\_deep \propto y \propto x \rightarrow R^2$ 
12 |
13 |  $\epsilon$ :
14 | T1: Split-half gc_deep -  $gc\_deep$ 
15 |  $gc\_deep \rightarrow R^2$ 
16 | T2: gc_deep - gc_deep
17 | gc_deep -  $R^2$ 
18 | T3: gc_obs -  $y = \log(gc\_obs)$  (gc_deep)
19 |  $\rightarrow gc\_deep$ 
20 | T4: gc_deep - gc_deep  $\pm 0.2$  dex
21 |  $\rightarrow R^2$ 
22 |
23 |  $\epsilon$ : uv run --with scipy --with matplotlib python sparc_kappa_circularity.py
24 | """
25 |
26 | import os, sys
27 | import numpy as np
28 | from scipy.optimize import minimize_scalar, minimize
29 | from scipy.stats import spearmanr
30 | import warnings
31 | warnings.filterwarnings('ignore')
32 |
33 | BASE = r"D:\.....\.....\.....\....."
34 | ROTMOD = os.path.join(BASE, "Rotmod_LTG")
35 | GC_CSV = os.path.join(BASE, "sparc_gc.csv")
36 | a0 = 1.2e-10
37 |
38 | # =====
39 | # (.....)
40 | # =====
41 | def load_gc_csv():
42 |     import csv
43 |     data = {}
44 |     with open(GC_CSV, 'r') as f:
45 |         reader = csv.DictReader(f)
46 |         for row in reader:
47 |             name = None
48 |             for k in row:
49 |                 if 'gal' in k.lower() or 'name' in k.lower():
50 |                     name = row[k].strip(); break
51 |             if not name: continue
52 |             gc_val = None
53 |             for k in ['gc', 'gc_obs', 'g_c', 'gc_a0']:
54 |                 if k in row and row[k]:
55 |                     try: gc_val = float(row[k]); break
56 |                     except: pass
57 |             if gc_val is None: continue
```

```

58 |         vflat, hR, Yd = None, None, None
59 |         for k in ['vflat', 'Vflat', 'v_flat']:
60 |             if k in row and row[k]:
61 |                 try: vflat = float(row[k]); break
62 |             except: pass
63 |         for k in ['hR', 'Reff', 'h_R', 'hr']:
64 |             if k in row and row[k]:
65 |                 try: hR = float(row[k]); break
66 |             except: pass
67 |         for k in ['Yd', 'yd', 'Y_d', 'SPS']:
68 |             if k in row and row[k]:
69 |                 try: Yd = float(row[k]); break
70 |             except: pass
71 |         data[name] = {'gc': gc_val, 'vflat': vflat, 'hR': hR, 'Yd': Yd}
72 |     gc_vals = [d['gc'] for d in data.values()]
73 |     if np.median(gc_vals) > 1e-5:
74 |         for n in data: data[n]['gc'] *= a0
75 |     return data
76 |
77 | def load_rotcurve(galaxy_name):
78 |     fname = os.path.join(ROTMOD, f"{galaxy_name}_rotmod.dat")
79 |     if not os.path.exists(fname): return None
80 |     cols = {k: [] for k in ['r', 'vobs', 'vgas', 'vdisk', 'vbul']}
81 |     with open(fname, 'r') as f:
82 |         for line in f:
83 |             line = line.strip()
84 |             if not line or line.startswith('#'): continue
85 |             p = line.split()
86 |             if len(p) < 6: continue
87 |             try:
88 |                 cols['r'].append(float(p[0]))
89 |                 cols['vobs'].append(float(p[1]))
90 |                 cols['vgas'].append(float(p[3]))
91 |                 cols['vdisk'].append(float(p[4]))
92 |                 cols['vbul'].append(float(p[5]))
93 |             except: continue
94 |     if len(cols['r']) < 5: return None
95 |     return {k: np.array(v) for k, v in cols.items()}
96 |
97 | def compute_gc_deep_from_indices(rc, Yd, indices):
98 |     """oooooooooooooooooo gc_deep ooo"""
99 |     r_m = rc['r'][indices] * 3.086e19
100 |     v_bar2 = (Yd * rc['vdisk'][indices]**2 + rc['vgas'][indices]**2 + rc['vbul'][indices]**2) * 1e6
101 |     v_obs2 = (rc['vobs'][indices] * 1e3)**2
102 |     gN = v_bar2 / r_m
103 |     gobs = v_obs2 / r_m
104 |     mask = gN > 0
105 |     if np.sum(mask) < 3: return None
106 |     gc_pts = gobs[mask]**2 / gN[mask]
107 |     gc_med = np.median(gc_pts)
108 |     if gc_med <= 0: return None
109 |     x = gN[mask] / gc_med
110 |     deep = x < 1.0
111 |     return np.median(gc_pts[deep]) if np.sum(deep) >= 3 else gc_med
112 |
113 | def compute_gc_deep(rc, Yd=0.5):
114 |     return compute_gc_deep_from_indices(rc, Yd, np.arange(len(rc['r'])))
115 |
116 | def compute_quantities(rc, gc_deep, Yd=0.5):
117 |     """M6 (u_strain/E_kin) o M3 (eps_max) o M8 (curvature) ooo"""
118 |     r_m = rc['r'] * 3.086e19
119 |     v_bar2 = (Yd * rc['vdisk']**2 + rc['vgas']**2 + rc['vbul']**2) * 1e6
120 |     v_obs2 = (rc['vobs'] * 1e3)**2
121 |     gN = np.maximum(v_bar2 / r_m, 0)
122 |     epsilon = np.sqrt(gN / gc_deep)
123 |
124 |     u_elastic = np.mean(epsilon**2)
125 |     E_kin = np.mean(v_obs2)
126 |     strain_kin = u_elastic / (E_kin / (3.086e19)**2) if E_kin > 0 else 0
127 |
128 |     eps_max = np.max(epsilon)
129 |
130 |     if len(epsilon) >= 5:
131 |         d2eps = np.gradient(np.gradient(epsilon, r_m), r_m)
132 |         curvature = np.mean(d2eps**2)
133 |     else:
134 |         curvature = 0
135 |
136 |     return {'strain_kin': strain_kin, 'eps_max': eps_max, 'curvature': curvature}
137 |
138 | # =====
139 | # M6 oooooo
140 | # =====
141 | def fit_M6(log_excess, log_sk):
142 |     """M6: y = a * log(strain_kin) - 1-param OLS"""
143 |     finite = np.isfinite(log_sk) & np.isfinite(log_excess)
144 |     if np.sum(finite) < 10: return None, 999
145 |     from numpy.linalg import lstsq
146 |     X = log_sk[finite].reshape(-1, 1)
147 |     sol, _, _, _ = lstsq(X, log_excess[finite], rcond=None)
148 |     pred = sol[0] * log_sk
149 |     pred[~finite] = 0

```

```

150 |     chi2 = np.sum((log_excess - pred)**2)
151 |     return sol[0], chi2
152 |
153 | def fit_M3(log_excess, log_eps_max):
154 |     """M3:  $y = \beta \cdot \max(0, \log(\text{eps\_max}) - \log(\text{eps\_c}))$ """
155 |     finite = np.isfinite(log_eps_max)
156 |     lem = log_eps_max[finite]
157 |     e = log_excess[finite]
158 |     best = None
159 |     for lec in np.linspace(np.percentile(lem, 10), np.percentile(lem, 90), 20):
160 |         for b in np.linspace(-1, 1, 20):
161 |             excess = np.maximum(lem - lec, 0)
162 |             chi2 = np.sum((e - b * excess)**2)
163 |             if best is None or chi2 < best[0]:
164 |                 best = (chi2, lec, b)
165 |         if best is None: return None, None, 999
166 |     # refine
167 |     def chi2_f(p):
168 |         exc = np.maximum(lem - p[0], 0)
169 |         return np.sum((e - p[1] * exc)**2)
170 |     res = minimize(chi2_f, [best[1], best[2]], method='Nelder-Mead')
171 |     return res.x[0], res.x[1], res.fun
172 |
173 | # =====
174 | # oooo
175 | # =====
176 | def main():
177 |     print("-" * 70)
178 |     print("oooooo: M3/M6/M8 o R2=0.55 oooo?")
179 |     print("-" * 70)
180 |
181 |     gc_data = load_gc_csv()
182 |
183 |     # ooooo
184 |     galaxies = []
185 |     for gname in sorted(gc_data.keys()):
186 |         gd = gc_data[gname]
187 |         gc_obs, vflat, hR = gd['gc'], gd.get('vflat'), gd.get('hR')
188 |         Yd = gd.get('Yd') or 0.5
189 |         if gc_obs &lt;= 0 or not vflat or not hR or vflat &lt;= 0 or hR &lt;= 0:
190 |             continue
191 |         rc = load_rotcurve(gname)
192 |         if rc is None or len(rc['r']) &lt; 8: continue # need enough points for split
193 |         gc_deep = compute_gc_deep(rc, Yd)
194 |         if gc_deep is None or gc_deep &lt;= 0: continue
195 |         sq = compute_quantities(rc, gc_deep, Yd)
196 |         if sq['strain_kin'] &lt;= 0: continue
197 |         galaxies.append({
198 |             'name': gname, 'gc_obs': gc_obs, 'gc_deep': gc_deep,
199 |             'vflat': vflat, 'hR': hR, 'Yd': Yd, 'rc': rc, **sq
200 |         })
201 |
202 |     N = len(galaxies)
203 |     print(f"Galaxies: {N}\n")
204 |
205 |     gc_obs = np.array([g['gc_obs'] for g in galaxies])
206 |     gc_deep = np.array([g['gc_deep'] for g in galaxies])
207 |     strain_kin = np.array([g['strain_kin'] for g in galaxies])
208 |     eps_max = np.array([g['eps_max'] for g in galaxies])
209 |     curvature = np.array([g['curvature'] for g in galaxies])
210 |     vflat = np.array([g['vflat'] for g in galaxies])
211 |     hR = np.array([g['hR'] for g in galaxies])
212 |
213 |     log_excess = np.log10(gc_obs / gc_deep)
214 |     log_sk = np.log10(np.maximum(strain_kin, 1e-50))
215 |     log_em = np.log10(np.maximum(eps_max, 1e-10))
216 |     log_curv = np.log10(np.maximum(curvature, 1e-80))
217 |
218 |     # --- Baseline ---
219 |     chi2_null = np.sum(log_excess**2)
220 |     a_m6, chi2_m6 = fit_M6(log_excess, log_sk)
221 |     R2_m6 = 1 - chi2_m6/chi2_null
222 |     print(f"Baseline M6: a={a_m6:.4f}, R2={R2_m6:.4f}")
223 |
224 |     lec_m3, beta_m3, chi2_m3 = fit_M3(log_excess, log_em)
225 |     R2_m3 = 1 - chi2_m3/chi2_null
226 |     print(f"Baseline M3:  $\epsilon_c$ ={10**lec_m3:.4f},  $\beta$ ={beta_m3:.3f}, R2={R2_m3:.4f}")
227 |
228 |     # =====
229 |     # T1: Split-half gc_deep (oooo)
230 |     # =====
231 |     print("\n" + "-" * 70)
232 |     print("T1: Split-half gc_deep")
233 |     print("-" * 70)
234 |     print("oooooooooooo/oooooooooooogc_deepoooo")
235 |     print("gc_deep_A o  $\epsilon$  ooooogc_deep_B o excess ooo  $\rightarrow$  ooooo\n")
236 |
237 |     np.random.seed(42)
238 |     R2_split_m6 = []
239 |     R2_split_m3 = []
240 |
241 |     for trial in range(100):

```

```

242 |         gc_deep_A = np.zeros(N)
243 |         gc_deep_B = np.zeros(N)
244 |         valid = np.ones(N, dtype=bool)
245 |
246 |         for i, g in enumerate(galaxies):
247 |             rc = g['rc']
248 |             n_pts = len(rc['r'])
249 |             perm = np.random.permutation(n_pts)
250 |             half = n_pts // 2
251 |             idxA, idxB = perm[:half], perm[half:]
252 |
253 |             gA = compute_gc_deep_from_indices(rc, g['Yd'], idxA)
254 |             gB = compute_gc_deep_from_indices(rc, g['Yd'], idxB)
255 |
256 |             if gA is None or gB is None or gA &lt;= 0 or gB &lt;= 0:
257 |                 valid[i] = False
258 |                 continue
259 |             gc_deep_A[i] = gA
260 |             gc_deep_B[i] = gB
261 |
262 |         if np.sum(valid) &lt; 50:
263 |             continue
264 |
265 |         # ε from gc_deep_A, excess from gc_deep_B
266 |         # → ○○○○○○
267 |         le_B = np.log10(gc_obs[valid] / gc_deep_B[valid])
268 |
269 |         # M6: strain_kin using gc_deep_A
270 |         sk_A = []
271 |         for i in np.where(valid)[0]:
272 |             sq = compute_quantities(galaxies[i]['rc'], gc_deep_A[i], galaxies[i]['Yd'])
273 |             sk_A.append(sq['strain_kin'])
274 |         sk_A = np.array(sk_A)
275 |         log_sk_A = np.log10(np.maximum(sk_A, 1e-50))
276 |
277 |         a_s, chi2_s = fit_M6(le_B, log_sk_A)
278 |         if a_s is not None:
279 |             R2_s = 1 - chi2_s / np.sum(le_B**2)
280 |             R2_split_m6.append(R2_s)
281 |
282 |         # M3: eps_max using gc_deep_A
283 |         em_A = []
284 |         for i in np.where(valid)[0]:
285 |             sq = compute_quantities(galaxies[i]['rc'], gc_deep_A[i], galaxies[i]['Yd'])
286 |             em_A.append(sq['eps_max'])
287 |         em_A = np.array(em_A)
288 |         log_em_A = np.log10(np.maximum(em_A, 1e-10))
289 |
290 |         _, _, chi2_s3 = fit_M3(le_B, log_em_A)
291 |         R2_s3 = 1 - chi2_s3 / np.sum(le_B**2)
292 |         R2_split_m3.append(R2_s3)
293 |
294 |         R2_split_m6 = np.array(R2_split_m6)
295 |         R2_split_m3 = np.array(R2_split_m3)
296 |
297 |         print(f"M6 split-half R²: mean={np.mean(R2_split_m6):.4f} "
298 |               f"(baseline={R2_m6:.4f}, ratio={np.mean(R2_split_m6)/R2_m6:.2f})")
299 |         print(f"M3 split-half R²: mean={np.mean(R2_split_m3):.4f} "
300 |               f"(baseline={R2_m3:.4f}, ratio={np.mean(R2_split_m3)/R2_m3:.2f})")
301 |
302 |         if np.mean(R2_split_m6) / R2_m6 &lt; 0.3:
303 |             print("○ M6: R² ○ 70%+ ○○ → gc_deep ○○○○○○○○")
304 |         elif np.mean(R2_split_m6) / R2_m6 &lt; 0.7:
305 |             print("○ M6: R² ○ 30-70% ○○ → ○○○○○○○○")
306 |         else:
307 |             print("○ M6: R² ○○○○○○ → ○○○○○○○○")
308 |
309 |         # =====
310 |         # T2: gc_deep ○○○○○
311 |         # =====
312 |         print("\n" + "=" * 70)
313 |         print("T2: gc_deep ○○○○○ (1000○)")
314 |         print("=" * 70)
315 |
316 |         np.random.seed(42)
317 |         R2_shuffle_m6 = []
318 |         for _ in range(1000):
319 |             perm = np.random.permutation(N)
320 |             gc_deep_s = gc_deep[perm]
321 |             le_s = np.log10(gc_obs / gc_deep_s)
322 |
323 |             sk_s = []
324 |             for i in range(N):
325 |                 sq = compute_quantities(galaxies[i]['rc'], gc_deep_s[i], galaxies[i]['Yd'])
326 |                 sk_s.append(sq['strain_kin'])
327 |             sk_s = np.array(sk_s)
328 |             log_sk_s = np.log10(np.maximum(sk_s, 1e-50))
329 |
330 |             a_s, chi2_s = fit_M6(le_s, log_sk_s)
331 |             if a_s is not None:
332 |                 R2_shuffle_m6.append(1 - chi2_s / np.sum(le_s**2))
333 |

```

```

334 | R2_shuffle_m6 = np.array(R2_shuffle_m6)
335 | p_val = np.mean(R2_shuffle_m6 >= R2_m6)
336 | print(f"M6 shuffle R2: mean={np.mean(R2_shuffle_m6):.4f}, "
337 |       f"max={np.max(R2_shuffle_m6):.4f}")
338 | print(f"Real R2={R2_m6:.4f}, p={p_val:.4f}")
339 |
340 | if np.mean(R2_shuffle_m6) > 0.3:
341 |     print("o oooooo R2 ooo → gc_deep ooooooo")
342 | else:
343 |     print("o oooooo R2 oo → ooooooo")
344 |
345 | # =====
346 | # T3: gc_obs oooo (gc_deep ooo)
347 | # =====
348 | print("\n" + "=" * 70)
349 | print("T3: gc_obs oooo (gc_deep ooooo)")
350 | print("=" * 70)
351 |
352 | # M6ooo gc_deep ooooo:
353 | # gN_mean = mean(V_bar2/r) (gc_deep oo)
354 | # E_kin_mean = mean(V_obs2)
355 | # proxy_ratio = gN_mean / E_kin_mean (ooooooooo)
356 |
357 | log_gc_obs = np.log10(gc_obs)
358 |
359 | # gc_deep-free oo
360 | gN_mean_arr = []
361 | E_kin_arr = []
362 | for g in galaxies:
363 |     rc = g['rc']
364 |     r_m = rc['r'] * 3.086e19
365 |     v_bar2 = (g['Yd'] * rc['vdisk']**2 + rc['vgas']**2 + rc['vbul']**2) * 1e6
366 |     v_obs2 = (rc['vobs'] * 1e3)**2
367 |     gN = np.maximum(v_bar2 / r_m, 0)
368 |     gN_mean_arr.append(np.mean(gN))
369 |     E_kin_arr.append(np.mean(v_obs2))
370 |
371 | gN_mean = np.array(gN_mean_arr)
372 | E_kin = np.array(E_kin_arr)
373 |
374 | # gc_deep-free proxy: gN_mean / E_kin (~ ε2 o proxy)
375 | proxy_nofree = gN_mean / (E_kin / (3.086e19)**2)
376 | log_proxy = np.log10(np.maximum(proxy_nofree, 1e-50))
377 |
378 | # gc_obs o log_proxy ooo
379 | from numpy.linalg import lstsq
380 | finite = np.isfinite(log_proxy)
381 | X = np.column_stack([log_proxy[finite], np.ones(np.sum(finite))])
382 | sol, _, _, _ = lstsq(X, log_gc_obs[finite], rcond=None)
383 | pred_direct = sol[0] * log_proxy + sol[1]
384 | chi2_direct = np.sum((log_gc_obs - pred_direct)**2)
385 | chi2_null_direct = np.sum((log_gc_obs - np.mean(log_gc_obs))**2)
386 | R2_direct = 1 - chi2_direct / chi2_null_direct
387 |
388 | print(f"gc_obs ~ proxy (gc_deep-free): R2={R2_direct:.4f}")
389 | print(f" (cf. M6 R2={R2_m6:.4f} uses gc_deep in both sides)")
390 |
391 | # gc_obs o vflat2/hR ooo (oooooo proxy)
392 | log_GSigma = np.log10(vflat**2 / hR)
393 | X2 = np.column_stack([log_GSigma, np.ones(N)])
394 | sol2, _, _, _ = lstsq(X2, log_gc_obs, rcond=None)
395 | pred_gs = sol2[0] * log_GSigma + sol2[1]
396 | R2_gs = 1 - np.sum((log_gc_obs - pred_gs)**2) / chi2_null_direct
397 | print(f"gc_obs ~ vflat2/hR (oooo): R2={R2_gs:.4f}")
398 |
399 | # gc_obs o gc_deep-free proxy + vflat2/hR ooo
400 | X3 = np.column_stack([log_proxy[finite], log_GSigma[finite], np.ones(np.sum(finite))])
401 | sol3, _, _, _ = lstsq(X3, log_gc_obs[finite], rcond=None)
402 | pred_comb = sol3[0]*log_proxy + sol3[1]*log_GSigma + sol3[2]
403 | R2_comb = 1 - np.sum((log_gc_obs - pred_comb)**2) / chi2_null_direct
404 | print(f"gc_obs ~ proxy + vflat2/hR: R2={R2_comb:.4f}")
405 | print(f" proxy ooo dR2 = {R2_comb - R2_gs:.4f}")
406 |
407 | if R2_comb - R2_gs < 0.02:
408 |     print("o proxy o vflat2/hR ooooooooooooo")
409 | else:
410 |     print(f"o proxy o vflat2/hR o +{(R2_comb-R2_gs)*100:.1f}% ooooooo")
411 |
412 | # =====
413 | # T4: gc_deep ooooo
414 | # =====
415 | print("\n" + "=" * 70)
416 | print("T4: gc_deep ooooo")
417 | print("=" * 70)
418 |
419 | np.random.seed(42)
420 | perturbations = [0.05, 0.1, 0.2, 0.3, 0.5] # dex
421 | for sigma in perturbations:
422 |     R2_perturbed = []
423 |     for _ in range(100):
424 |         noise = 10**(np.random.normal(0, sigma, N))
425 |         gc_deep_p = gc_deep * noise

```

```

426 |         le_p = np.log10(gc_obs / gc_deep_p)
427 |
428 |         sk_p = []
429 |         for i in range(N):
430 |             sq = compute_quantities(galaxies[i]['rc'], gc_deep_p[i], galaxies[i]['Yd'])
431 |             sk_p.append(sq['strain_kin'])
432 |         sk_p = np.array(sk_p)
433 |         log_sk_p = np.log10(np.maximum(sk_p, 1e-50))
434 |
435 |         a_p, chi2_p = fit_M6(le_p, log_sk_p)
436 |         if a_p is not None:
437 |             R2_perturbed.append(1 - chi2_p / np.sum(le_p**2))
438 |
439 |         R2_p = np.array(R2_perturbed)
440 |         print(f"   sigma={sigma:.2f} dex: R^2={np.mean(R2_p):.4f} "
441 |               f"(baseline={R2_m6:.4f}, ratio={np.mean(R2_p)/R2_m6:.2f})")
442 |
443 |         # =====
444 |         # T5:  $\sigma - \epsilon^2 = gN/gc\_deep$   $\square$   $y = \log(gc\_obs/gc\_deep)$   $\square$ 
445 |         #    $\square\square\square\square\square\square$ 
446 |         # =====
447 |         print("\n" + "=" * 70)
448 |         print("T5:  $\square\square\square\square\square$ ")
449 |         print("=" * 70)
450 |
451 |         # M6  $\square\square\square$ :
452 |         #  $y = a \times \log(\text{strain\_kin}) = a \times \log(\epsilon^2) / E\_kin\_scaled$ 
453 |         #  $\epsilon^2 = gN/gc\_deep$ 
454 |         #  $y = \log(gc\_obs/gc\_deep)$ 
455 |         #
456 |         # deep MOND:  $gc\_obs \approx \sqrt{gN \times gc\_deep\_true}$  / something
457 |         #  $\rightarrow \log(gc\_obs/gc\_deep) = \log(gc\_obs) - \log(gc\_deep)$ 
458 |         #  $\rightarrow$  if  $gc\_deep$  has error  $\delta$ :  $y = y\_true - \delta$ 
459 |         #  $\rightarrow \epsilon^2 = gN/gc\_deep = (gN/gc\_deep\_true) \times 10^{\delta}$ 
460 |         #  $\rightarrow \log(\epsilon^2) = \log(\epsilon^2\_true) + \delta$ 
461 |         #  $\rightarrow y = y\_true - \delta$ ,  $x = x\_true + \delta$   $\rightarrow$  anti-correlation from  $\delta$ 
462 |
463 |         #  $\square\square\square\square$ :  $\log\_excess$  vs  $\log(\epsilon^2)$   $\square\square\square$ 
464 |         log_eps2_mean = np.log10(np.array([g['strain_kin'] for g in galaxies]) *
465 |                                   E_kin / (3.086e19)**2 + 1e-50)
466 |         # Actually  $\log(\epsilon^2) = u\_elastic$ 
467 |         u_el = np.array([np.mean(np.sqrt(np.maximum(
468 |             (g['Yd']*g['rc']['vdisk']**2 + g['rc']['vgas']**2 + g['rc']['vbul']**2)*1e6 /
469 |             (g['rc']['r']*3.086e19) / g['gc_deep'])),**2
470 |             for g in galaxies]) #  $\epsilon\_mean^2$  proxy
471 |
472 |         # simpler: just compute correlation of excess with  $\epsilon$  quantities
473 |         rho_direct, p_direct = spearmanr(log_excess, np.log10(np.maximum(strain_kin, 1e-50)))
474 |         print(f"p(log_excess, log_strain_kin) = {rho_direct:+.3f} (p={p_direct:.2e})")
475 |
476 |         # If this is  $\sim -0.7$  or stronger, it's suspicious
477 |         if abs(rho_direct) > 0.6:
478 |             print("  $\square$   $\square\square\square$   $\rightarrow$  gc_deep  $\square\square\square\square\square$ ")
479 |         elif abs(rho_direct) > 0.3:
480 |             print("  $\square$   $\square\square\square$   $\rightarrow$   $\square\square\square\square\square\square\square\square$ ")
481 |         else:
482 |             print("  $\square$   $\square\square\square$   $\rightarrow$  gc_deep  $\square\square\square\square\square$ ")
483 |
484 |         # =====
485 |         #  $\square\square\square$ 
486 |         # =====
487 |         print("\n" + "=" * 70)
488 |         print("  $\square\square\square$ ")
489 |         print("=" * 70)
490 |         print(f"")
491 |          $\square\square\square\square$ :
492 |         T1 Split-half: M6  $R^2$  {np.mean(R2_split_m6):.4f} / {R2_m6:.4f} = {np.mean(R2_split_m6)/R2_m6:.2f}
493 |         T1 Split-half: M3  $R^2$  {np.mean(R2_split_m3):.4f} / {R2_m3:.4f} = {np.mean(R2_split_m3)/R2_m3:.2f}
494 |         T2 Shuffle: mean  $R^2$ ={np.mean(R2_shuffle_m6):.4f}, p={p_val:.4f}
495 |         T3 Direct: gc_deep-free  $R^2$ ={R2_direct:.4f},  $dR^2$  over geometric mean = {R2_comb-R2_gs:.4f}
496 |         T5 Correlation:  $\rho$ ={rho_direct:+.3f}
497 |
498 |          $\square\square\square\square$ :
499 |         Split-half ratio > 0.7 AND shuffle p < 0.01 AND  $dR^2(T3)$  > 0.02
500 |          $\rightarrow$   $\square\square\square\square$ 
501 |
502 |         Split-half ratio < 0.3 OR shuffle  $R^2$  > 0.3
503 |          $\rightarrow$  gc_deep  $\square\square\square\square\square\square\square$ 
504 |
505 |          $\square\square$   $\rightarrow$   $\square\square\square\square\square\square\square\square$ 
506 |         """)
507 |
508 |         print("  $\square\square\square$ ")
509 |
510 | if __name__ == '__main__':
511 |     main()

```

Script 4: sparc_M3_M6_independence.py

目的

M3 (epsilon閾値) と M6 (u_strain/E_kin) が相補的か冗長かの判定

7段階テスト: T1: M3残差 vs M6残差の相関。T2: M3+M6 結合モデル (3パラメータ) の dAICc。T2b: L00-CV で結合モデルが単独より改善するか。T3: 増分 R^2 (M3残差を M6 で予測 / M6残差を M3 で予測)。T4: hR 偏相関の段階的改善。T5: M3指標と M6指標の物理的独立性 (vflat統制下の相関)。T6: M3/M6/M8 の三角関係。T7: シャッフル独立性テスト。

結果

M6 は M3 の冗長な代替指標。T3: M6 → M3残差 $R^2=0.0000$ (独立情報ゼロ)。T7: シャッフル $p=0.76$ (偶然と区別不可)。T5: vflat 統制下で $\rho=0.90$ (ほぼ同一)。条件14の証拠は「3系列」から「M3の1系列のみ」に縮小。

実行方法

```
uv run --with scipy --with matplotlib python sparc_M3_M6_independence.py
```

入力データ: sparc_gc.csv + Rotmod_LTG/*.dat

出力: M3_M6_independence.png + 標準出力

ソースコード全文

```
1 | #!/usr/bin/env python3
2 | """
3 | sparc_M3_M6_independence.py
4 | =====
5 | M3 (ε) vs M6 (u_strain/E_kin) ??????
6 |
7 | ???
8 | T1: M3 vs M6 →
9 | T2: M3+M6 combined model (3 param) - dAICc
10 | T3: M3 vs M6 / M6 vs M3 →  $R^2$ 
11 | T4: : M3 vs M6 vs hR
12 | T5: : M3 vs M6
13 |
14 | uv run --with scipy --with matplotlib python sparc_M3_M6_independence.py
15 | """
16 |
17 | import os, sys
18 | import numpy as np
19 | from scipy.optimize import minimize, minimize_scalar
20 | from scipy.stats import spearmanr, pearsonr
21 | import warnings
22 | warnings.filterwarnings('ignore')
23 |
24 | BASE = r"D:\????\????\????\????\????"
25 | ROTMOD = os.path.join(BASE, "Rotmod_LTG")
26 | GC_CSV = os.path.join(BASE, "sparc_gc.csv")
27 | a0 = 1.2e-10
28 |
29 | # =====
30 | # ( )
31 | # =====
32 | def load_gc_csv():
33 |     import csv
34 |     data = {}
35 |     with open(GC_CSV, 'r') as f:
36 |         reader = csv.DictReader(f)
37 |         for row in reader:
38 |             name = None
39 |             for k in row:
40 |                 if 'gal' in k.lower() or 'name' in k.lower():
41 |                     name = row[k].strip(); break
42 |             if not name: continue
43 |             gc_val = None
44 |             for k in ['gc', 'gc_obs', 'g_c', 'gc_a0']:
45 |                 if k in row and row[k]:
46 |                     try: gc_val = float(row[k]); break
47 |                     except: pass
48 |             if gc_val is None: continue
49 |             vflat, hR, Yd = None, None, None
50 |             for k in ['vflat', 'vflat', 'v_flat']:
51 |                 if k in row and row[k]:
52 |                     try: vflat = float(row[k]); break
53 |                     except: pass
54 |             for k in ['hR', 'Reff', 'h_R', 'hr']:
55 |                 if k in row and row[k]:
```

```

56 |         try: hR = float(row[k]); break
57 |         except: pass
58 |     for k in ['Yd', 'yd', 'Y_d', 'SPS']:
59 |         if k in row and row[k]:
60 |             try: Yd = float(row[k]); break
61 |             except: pass
62 |     data[name] = {'gc': gc_val, 'vflat': vflat, 'hR': hR, 'Yd': Yd}
63 | gc_vals = [d['gc'] for d in data.values()]
64 | if np.median(gc_vals) > 1e-5:
65 |     for n in data: data[n]['gc'] *= a0
66 | return data
67 |
68 | def load_rotcurve(gname):
69 |     fname = os.path.join(ROTMOD, f"{gname}_rotmod.dat")
70 |     if not os.path.exists(fname): return None
71 |     cols = {k: [] for k in ['r', 'vobs', 'vgas', 'vdisk', 'vbul']}
72 |     with open(fname, 'r') as f:
73 |         for line in f:
74 |             line = line.strip()
75 |             if not line or line.startswith('#'): continue
76 |             p = line.split()
77 |             if len(p) < 6: continue
78 |             try:
79 |                 cols['r'].append(float(p[0]))
80 |                 cols['vobs'].append(float(p[1]))
81 |                 cols['vgas'].append(float(p[3]))
82 |                 cols['vdisk'].append(float(p[4]))
83 |                 cols['vbul'].append(float(p[5]))
84 |             except: continue
85 |     if len(cols['r']) < 5: return None
86 |     return {k: np.array(v) for k, v in cols.items()}
87 |
88 | def compute_gc_deep(rc, Yd=0.5):
89 |     r_m = rc['r'] * 3.086e19
90 |     v_bar2 = (Yd * rc['vdisk']**2 + rc['vgas']**2 + rc['vbul']**2) * 1e6
91 |     v_obs2 = (rc['vobs'] * 1e3)**2
92 |     gN = v_bar2 / r_m
93 |     gobs = v_obs2 / r_m
94 |     mask = gN > 0
95 |     if np.sum(mask) < 3: return None
96 |     gc_pts = gobs[mask]**2 / gN[mask]
97 |     gc_med = np.median(gc_pts)
98 |     if gc_med <= 0: return None
99 |     x = gN[mask] / gc_med
100 |     deep = x < 1.0
101 |     return np.median(gc_pts[deep]) if np.sum(deep) >= 3 else gc_med
102 |
103 | def compute_quantities(rc, gc_deep, Yd=0.5):
104 |     r_m = rc['r'] * 3.086e19
105 |     v_bar2 = (Yd * rc['vdisk']**2 + rc['vgas']**2 + rc['vbul']**2) * 1e6
106 |     v_obs2 = (rc['vobs'] * 1e3)**2
107 |     gN = np.maximum(v_bar2 / r_m, 0)
108 |     epsilon = np.sqrt(gN / gc_deep)
109 |
110 |     u_elastic = np.mean(epsilon**2)
111 |     E_kin = np.mean(v_obs2)
112 |     strain_kin = u_elastic / (E_kin / (3.086e19)**2) if E_kin > 0 else 0
113 |     eps_max = np.max(epsilon)
114 |
115 |     if len(epsilon) >= 5:
116 |         d2eps = np.gradient(np.gradient(epsilon, r_m), r_m)
117 |         curvature = np.mean(d2eps**2)
118 |     else:
119 |         curvature = 0
120 |
121 |     # E_grad/E_local (ratio from previous sessions)
122 |     deps_dr = np.zeros_like(epsilon)
123 |     deps_dr[0] = (epsilon[1] - epsilon[0]) / (r_m[1] - r_m[0])
124 |     deps_dr[-1] = (epsilon[-1] - epsilon[-2]) / (r_m[-1] - r_m[-2])
125 |     for i in range(1, len(epsilon)-1):
126 |         deps_dr[i] = (epsilon[i+1] - epsilon[i-1]) / (r_m[i+1] - r_m[i-1])
127 |     E_grad = np.mean(deps_dr**2)
128 |     E_local = np.mean(epsilon**2)
129 |     ratio = E_grad / E_local if E_local > 0 else 0
130 |
131 |     return {
132 |         'strain_kin': strain_kin, 'eps_max': eps_max, 'curvature': curvature,
133 |         'ratio': ratio, 'eps_mean': np.mean(epsilon),
134 |         'n_points': len(epsilon), 'rmax_hR': rc['r'][-1],
135 |     }
136 |
137 | # =====
138 | # □□□□□
139 | # =====
140 | def build_dataset():
141 |     gc_data = load_gc_csv()
142 |     results = []
143 |     for gname in sorted(gc_data.keys()):
144 |         gd = gc_data[gname]
145 |         gc_obs, vflat, hR = gd['gc'], gd.get('vflat'), gd.get('hR')
146 |         Yd = gd.get('Yd') or 0.5
147 |         if gc_obs <= 0 or not vflat or not hR or vflat <= 0 or hR <= 0:

```

```

148 |         continue
149 |         rc = load_rotcurve(gname)
150 |         if rc is None: continue
151 |         gc_deep = compute_gc_deep(rc, Yd)
152 |         if gc_deep is None or gc_deep <= 0: continue
153 |         sq = compute_quantities(rc, gc_deep, Yd)
154 |         if sq['strain_kin'] <= 0: continue
155 |         results.append({
156 |             'name': gname, 'gc_obs': gc_obs, 'gc_deep': gc_deep,
157 |             'vflat': vflat, 'hR': hR, 'Yd': Yd,
158 |             'rmax_hR': rc['r'][-1] / hR if hR > 0 else 0,
159 |             **{k: sq[k] for k in ['strain_kin', 'eps_max', 'curvature', 'ratio', 'eps_mean']},
160 |         })
161 |     print(f"Dataset: {len(results)} galaxies\n")
162 |     return results
163 |
164 | # =====
165 | # oooooooo
166 | # =====
167 | def fit_M3(log_excess, log_eps_max, mask=None):
168 |     """M3: y = beta * max(0, log(eps_max) - log_ec)"""
169 |     if mask is None: mask = np.ones(len(log_excess), dtype=bool)
170 |     lem = log_eps_max[mask]
171 |     e = log_excess[mask]
172 |     best = None
173 |     for lec in np.linspace(np.percentile(lem, 5), np.percentile(lem, 95), 30):
174 |         for b in np.linspace(-1, 1, 30):
175 |             exc = np.maximum(lem - lec, 0)
176 |             chi2 = np.sum((e - b * exc)**2)
177 |             if best is None or chi2 < best[0]:
178 |                 best = (chi2, lec, b)
179 |     if best is None: return None
180 |     res = minimize(lambda p: np.sum((e - p[1]*np.maximum(lem-p[0],0))**2),
181 |                   [best[1], best[2]], method='Nelder-Mead')
182 |     return res.x # [log_ec, beta]
183 |
184 | def pred_M3(params, log_eps_max):
185 |     log_ec, beta = params
186 |     return beta * np.maximum(log_eps_max - log_ec, 0)
187 |
188 | def fit_M6(log_excess, log_sk, mask=None):
189 |     """M6: y = a * log(strain_kin)"""
190 |     if mask is None: mask = np.ones(len(log_excess), dtype=bool)
191 |     finite = np.isfinite(log_sk[mask])
192 |     from numpy.linalg import lstsq
193 |     X = log_sk[mask][finite].reshape(-1, 1)
194 |     sol, _, _, _ = lstsq(X, log_excess[mask][finite], rcond=None)
195 |     return sol[0] # a
196 |
197 | def pred_M6(a, log_sk):
198 |     return a * log_sk
199 |
200 | # =====
201 | # AICc
202 | # =====
203 | def aicc(chi2, n, k):
204 |     aic = chi2 + 2*k
205 |     if n - k - 1 > 0:
206 |         return aic + 2*k*(k+1) / (n - k - 1)
207 |     return aic
208 |
209 | def partial_corr(x, y, z):
210 |     """rho(x,y|z)"""
211 |     cx = np.polyfit(z, x, 1)
212 |     cy = np.polyfit(z, y, 1)
213 |     return spearmanr(x - np.polyval(cx, z), y - np.polyval(cy, z))
214 |
215 | def partial_corr_multi(target, y, controls):
216 |     from numpy.linalg import lstsq
217 |     X = np.column_stack([*controls, np.ones(len(target))])
218 |     ct, _, _, _ = lstsq(X, target, rcond=None)
219 |     cy, _, _, _ = lstsq(X, y, rcond=None)
220 |     return spearmanr(target - X @ ct, y - X @ cy)
221 |
222 | # =====
223 | # ooo
224 | # =====
225 | def main():
226 |     print("-" * 70)
227 |     print("M3 vs M6 ooooo")
228 |     print("-" * 70)
229 |
230 |     data = build_dataset()
231 |     N = len(data)
232 |
233 |     gc_obs = np.array([d['gc_obs'] for d in data])
234 |     gc_deep = np.array([d['gc_deep'] for d in data])
235 |     strain_kin = np.array([d['strain_kin'] for d in data])
236 |     eps_max = np.array([d['eps_max'] for d in data])
237 |     curvature = np.array([d['curvature'] for d in data])
238 |     ratio = np.array([d['ratio'] for d in data])
239 |     vflat = np.array([d['vflat'] for d in data])

```

```

240 |     hR = np.array([d['hR'] for d in data])
241 |     rmax_hR = np.array([d['rmax_hR'] for d in data])
242 |
243 |     log_excess = np.log10(gc_obs / gc_deep)
244 |     log_sk = np.log10(np.maximum(strain_kin, 1e-50))
245 |     log_em = np.log10(np.maximum(eps_max, 1e-10))
246 |     log_curv = np.log10(np.maximum(curvature, 1e-80))
247 |     log_vf = np.log10(vflat)
248 |     log_hR = np.log10(hR)
249 |
250 |     chi2_null = np.sum(log_excess**2)
251 |
252 |     # --- Fit individual models ---
253 |     params_M3 = fit_M3(log_excess, log_em)
254 |     pred_m3 = pred_M3(params_M3, log_em)
255 |     resid_m3 = log_excess - pred_m3
256 |     chi2_m3 = np.sum(resid_m3**2)
257 |
258 |     a_M6 = fit_M6(log_excess, log_sk)
259 |     pred_m6 = pred_M6(a_M6, log_sk)
260 |     resid_m6 = log_excess - pred_m6
261 |     chi2_m6 = np.sum(resid_m6**2)
262 |
263 |     print(f"M3: log_ec={params_M3[0]:.3f} (eps_c={10**params_M3[0]:.4f}), "
264 |           f"beta={params_M3[1]:.3f}, R2={1-chi2_m3/chi2_null:.4f}")
265 |     print(f"M6: a={a_M6:.4f}, R2={1-chi2_m6/chi2_null:.4f}")
266 |
267 |     # =====
268 |     # T1: 0000
269 |     # =====
270 |     print("\n" + "=" * 70)
271 |     print("T1: M300 vs M600 0000")
272 |     print("=" * 70)
273 |
274 |     rho_resid, p_resid = spearmanr(resid_m3, resid_m6)
275 |     rho_resid_p, p_resid_p = pearsonr(resid_m3, resid_m6)
276 |     print(f"Spearman rho(resid_M3, resid_M6) = {rho_resid:+.4f} (p={p_resid:.2e})")
277 |     print(f"Pearson r(resid_M3, resid_M6) = {rho_resid_p:+.4f} (p={p_resid_p:.2e})")
278 |
279 |     if abs(rho_resid) > 0.8:
280 |         print("-&gt; 0000: M300M60000")
281 |     elif abs(rho_resid) > 0.5:
282 |         print("-&gt; 00000: 000000000000")
283 |     else:
284 |         print("-&gt; 0000: M300M60000")
285 |
286 |     # =====
287 |     # T2: Combined model M3+M6 (3 params)
288 |     # =====
289 |     print("\n" + "=" * 70)
290 |     print("T2: Combined model M3+M6 (3 params)")
291 |     print("=" * 70)
292 |
293 |     def fit_combined(le, lem, lsk, mask=None):
294 |         if mask is None: mask = np.ones(len(le), dtype=bool)
295 |         e = le[mask]
296 |         em = lem[mask]
297 |         sk = lsk[mask]
298 |
299 |         def chi2_comb(p):
300 |             log_ec, beta, a = p
301 |             pred = beta * np.maximum(em - log_ec, 0) + a * sk
302 |             return np.sum((e - pred)**2)
303 |
304 |         best = None
305 |         for lec_init in np.linspace(np.percentile(em, 10), np.percentile(em, 90), 10):
306 |             for beta_init in [-0.3, -0.1, 0.1, 0.3]:
307 |                 for a_init in [-0.01, -0.005, 0.005, 0.01]:
308 |                     res = minimize(chi2_comb, [lec_init, beta_init, a_init],
309 |                                   method='Nelder-Mead', options={'maxiter': 5000})
310 |                     if best is None or res.fun < best.fun:
311 |                         best = res
312 |         return best.x, best.fun
313 |
314 |     params_comb, chi2_comb = fit_combined(log_excess, log_em, log_sk)
315 |     R2_comb = 1 - chi2_comb / chi2_null
316 |
317 |     aicc_null = aicc(chi2_null, N, 0)
318 |     aicc_m3 = aicc(chi2_m3, N, 2)
319 |     aicc_m6 = aicc(chi2_m6, N, 1)
320 |     aicc_comb = aicc(chi2_comb, N, 3)
321 |
322 |     print(f"Combined: log_ec={params_comb[0]:.3f}, beta={params_comb[1]:.3f}, "
323 |           f"a_M6={params_comb[2]:.4f}")
324 |     print("\n\nModel comparison:")
325 |     print(f" {'Model':&lt;&lt;25s} {'k':&lt;&lt;3s} {'chi2':&lt;&lt;8s} {'R2':&lt;&lt;7s} {'AICc':&lt;&lt;8s} {'dAICc':&lt;&lt;8s}")
326 |     print(f" {'-'*60}")
327 |     print(f" {'M0: null':&lt;&lt;25s} {'0':&lt;&lt;3s} {chi2_null:&lt;&lt;8.2f} {'0.000':&lt;&lt;7s} "
328 |           f" {aicc_null:&lt;&lt;8.1f} {'0.0':&lt;&lt;8s}")
329 |     print(f" {'M6: u_strain/E_kin':&lt;&lt;25s} {'1':&lt;&lt;3s} {chi2_m6:&lt;&lt;8.2f} "
330 |           f" {1-chi2_m6/chi2_null:&lt;&lt;7.4f} {aicc_m6:&lt;&lt;8.1f} {aicc_m6-aicc_null:&lt;&lt;+8.1f}")
331 |     print(f" {'M3: eps threshold':&lt;&lt;25s} {'2':&lt;&lt;3s} {chi2_m3:&lt;&lt;8.2f} "

```

```

332 |         f"{1-chi2_m3/chi2_null:&gt;7.4f} {aicc_m3:&gt;8.1f} {aicc_m3-aicc_null:&gt;+8.1f}")
333 |     print(f"    {'M3+M6: combined':&lt;&lt;25s} {'3':&gt;&gt;3s} {chi2_comb:&gt;8.2f} "
334 |           f"{'R2_comb:&gt;7.4f} {aicc_comb:&gt;8.1f} {aicc_comb-aicc_null:&gt;+8.1f}")
335 |
336 |     dAICc_comb_vs_m3 = aicc_comb - aicc_m3
337 |     dAICc_comb_vs_m6 = aicc_comb - aicc_m6
338 |     print(f"\n    M3+M6 vs M3 alone: dAICc = {dAICc_comb_vs_m3:+.1f}")
339 |     print(f"    M3+M6 vs M6 alone: dAICc = {dAICc_comb_vs_m6:+.1f}")
340 |
341 |     if dAICc_comb_vs_m3 &lt; -2 and dAICc_comb_vs_m6 &lt; -2:
342 |         print("    -&gt; 00000000: M3M60000")
343 |     elif dAICc_comb_vs_m3 &gt;= 0 and dAICc_comb_vs_m6 &gt;= 0:
344 |         print("    -&gt; 00000000: 000000")
345 |     else:
346 |         best_single = "M3" if chi2_m3 &lt; chi2_m6 else "M6"
347 |         print(f"    -&gt; {best_single} 0000000000")
348 |
349 |     # =====
350 |     # T2b: LOO-CV for combined model
351 |     # =====
352 |     print("\n--- LOO-CV for combined model ---")
353 |     loo_errors_comb = []
354 |     loo_errors_m3 = []
355 |     loo_errors_m6 = []
356 |     for i in range(N):
357 |         mask = np.ones(N, dtype=bool)
358 |         mask[i] = False
359 |
360 |         # Combined
361 |         try:
362 |             p_c, _ = fit_combined(log_excess, log_em, log_sk, mask)
363 |             pred_i = p_c[1] * max(log_em[i] - p_c[0], 0) + p_c[2] * log_sk[i]
364 |             loo_errors_comb.append((log_excess[i] - pred_i)**2)
365 |         except:
366 |             pass
367 |
368 |         # M3
369 |         p3 = fit_M3(log_excess, log_em, mask)
370 |         if p3 is not None:
371 |             pred_i3 = pred_M3(p3, np.array([log_em[i]]))[0]
372 |             loo_errors_m3.append((log_excess[i] - pred_i3)**2)
373 |
374 |         # M6
375 |         a6 = fit_M6(log_excess, log_sk, mask)
376 |         pred_i6 = a6 * log_sk[i]
377 |         loo_errors_m6.append((log_excess[i] - pred_i6)**2)
378 |
379 |     mse_null = np.mean(log_excess**2)
380 |     mse_comb = np.mean(loo_errors_comb) if loo_errors_comb else 999
381 |     mse_m3 = np.mean(loo_errors_m3) if loo_errors_m3 else 999
382 |     mse_m6 = np.mean(loo_errors_m6) if loo_errors_m6 else 999
383 |
384 |     print(f"    LOO-CV MSE: null={mse_null:.5f}, M6={mse_m6:.5f}, "
385 |           f"{'M3':{mse_m3:.5f}, M3+M6={mse_comb:.5f}")
386 |     print(f"    LOO ratio: M6={mse_m6/mse_null:.3f}, M3={mse_m3/mse_null:.3f}, "
387 |           f"{'M3+M6':{mse_comb/mse_null:.3f}")
388 |
389 |     if mse_comb &lt; min(mse_m3, mse_m6) * 0.95:
390 |         print("    -&gt; Combined 0 5%+ 00: 00000000")
391 |     else:
392 |         print("    -&gt; Combined 000000: 000000")
393 |
394 |     # =====
395 |     # T3: 00R^2 (0000)
396 |     # =====
397 |     print("\n" + "=" * 70)
398 |     print("T3: 00 R2 000")
399 |     print("=" * 70)
400 |
401 |     # M3000M600000
402 |     from numpy.linalg import lstsq
403 |     X_m6 = log_sk.reshape(-1, 1)
404 |     sol_m6on3, _, _, _ = lstsq(X_m6, resid_m3, rcond=None)
405 |     pred_m6on3 = sol_m6on3[0] * log_sk
406 |     R2_m6on3 = 1 - np.sum((resid_m3 - pred_m6on3)**2) / np.sum(resid_m3**2)
407 |
408 |     # M6000M300000
409 |     exc_m3 = np.maximum(log_em - params_M3[0], 0)
410 |     X_m3 = exc_m3.reshape(-1, 1)
411 |     sol_m3on6, _, _, _ = lstsq(X_m3, resid_m6, rcond=None)
412 |     pred_m3on6 = sol_m3on6[0] * exc_m3
413 |     R2_m3on6 = 1 - np.sum((resid_m6 - pred_m3on6)**2) / np.sum(resid_m6**2)
414 |
415 |     print(f"{'M6 on M3 residuals: R2 = {R2_m6on3:.4f} "
416 |           f"({'M6 adds info' if R2_m6on3 &gt; 0.05 else 'negligible'})")
417 |     print(f"{'M3 on M6 residuals: R2 = {R2_m3on6:.4f} "
418 |           f"({'M3 adds info' if R2_m3on6 &gt; 0.05 else 'negligible'})")
419 |
420 |     # =====
421 |     # T4: hR000000000
422 |     # =====
423 |     print("\n" + "=" * 70)

```

```

424 | print("T4: hR#####")
425 | print("=" * 70)
426 |
427 | gc_m3 = gc_deep * 10**pred_m3
428 | gc_m6 = gc_deep * 10**pred_m6
429 | gc_comb = gc_deep * 10**(params_comb[1]*np.maximum(log_em-params_comb[0],0)
430 |           + params_comb[2]*log_sk)
431 |
432 | log_gc_obs = np.log10(gc_obs)
433 | log_gc_m3 = np.log10(np.maximum(gc_m3, 1e-30))
434 | log_gc_m6 = np.log10(np.maximum(gc_m6, 1e-30))
435 | log_gc_comb = np.log10(np.maximum(gc_comb, 1e-30))
436 |
437 | rho_obs, _ = partial_corr(log_gc_obs, log_hR, log_vf)
438 | rho_m3, _ = partial_corr(log_gc_m3, log_hR, log_vf)
439 | rho_m6, _ = partial_corr(log_gc_m6, log_hR, log_vf)
440 | rho_comb, _ = partial_corr(log_gc_comb, log_hR, log_vf)
441 |
442 | print(f" rho(gc, hR | vflat):")
443 | print(f"   gc_obs:      {rho_obs:+.4f}")
444 | print(f"   gc_M3:       {rho_m3:+.4f} (={1-abs(rho_m3)/abs(rho_obs)}*100:.1f}%)")
445 | print(f"   gc_M6:       {rho_m6:+.4f} (={1-abs(rho_m6)/abs(rho_obs)}*100:.1f}%)")
446 | print(f"   gc_M3+M6:    {rho_comb:+.4f} (={1-abs(rho_comb)/abs(rho_obs)}*100:.1f}%)")
447 |
448 | # + rmax/hR ==
449 | rho_obs2, _ = partial_corr_multi(log_gc_obs, log_hR, [log_vf, rmax_hR])
450 | rho_comb2, _ = partial_corr_multi(log_gc_comb, log_hR, [log_vf, rmax_hR])
451 | print(f"\n rho(gc, hR | vflat, rmax/hR):")
452 | print(f"   gc_obs:      {rho_obs2:+.4f}")
453 | print(f"   gc_M3+M6:    {rho_comb2:+.4f} (={1-abs(rho_comb2)/abs(rho_obs2)}*100:.1f}%)")
454 |
455 | # =====
456 | # T5: ##### - #####
457 | # =====
458 | print("\n" + "=" * 70)
459 | print("T5: M3 vs M6 #####")
460 | print("=" * 70)
461 |
462 | rho_indicators, p_ind = spearmanr(log_em, log_sk)
463 | print(f" rho(log_eps_max, log_strain_kin) = {rho_indicators:+.4f} (p={p_ind:.2e})")
464 |
465 | # M3 indicator = max(0, log_em - log_ec)
466 | m3_indicator = np.maximum(log_em - params_M3[0], 0)
467 | rho_m3_m6, p_m3_m6 = spearmanr(m3_indicator, log_sk)
468 | print(f" rho(M3_indicator, log_strain_kin) = {rho_m3_m6:+.4f} (p={p_m3_m6:.2e})")
469 |
470 | # ==: vflat==
471 | rho_ind_vf, p_ind_vf = partial_corr(log_em, log_sk, log_vf)
472 | print(f" rho(log_eps_max, log_strain_kin | vflat) = {rho_ind_vf:+.4f} (p={p_ind_vf:.2e})")
473 |
474 | if abs(rho_indicators) > 0.8:
475 |     print(" -> #####: M3M6#####")
476 | elif abs(rho_indicators) > 0.5:
477 |     print(" -> ==: #####")
478 | else:
479 |     print(" -> ==: #####")
480 |
481 | # =====
482 | # T6: M8 (curvature) #####
483 | # =====
484 | print("\n" + "=" * 70)
485 | print("T6: M3/M6/M8 #####")
486 | print("=" * 70)
487 |
488 | rho_36, _ = spearmanr(log_em, log_sk)
489 | rho_38, _ = spearmanr(log_em, log_curv)
490 | rho_68, _ = spearmanr(log_sk, log_curv)
491 | print(f" rho(M3, M6) = {rho_36:+.3f}")
492 | print(f" rho(M3, M8) = {rho_38:+.3f}")
493 | print(f" rho(M6, M8) = {rho_68:+.3f}")
494 |
495 | if abs(rho_68) > 0.9:
496 |     print(" -> M6M8#####")
497 | if abs(rho_36) < abs(rho_68):
498 |     print(" -> M3M6/M8#####")
499 |
500 | # =====
501 | # T7: #####
502 | # =====
503 | print("\n" + "=" * 70)
504 | print("T7: #####")
505 | print("=" * 70)
506 | print("M3#####M6#####R2#####")
507 |
508 | np.random.seed(42)
509 | R2_shuffle = []
510 | for _ in range(1000):
511 |     sk_s = np.random.permutation(log_sk)
512 |     sol_s, _, _, _ = lstsq(sk_s.reshape(-1,1), resid_m3, rcond=None)
513 |     pred_s = sol_s[0] * sk_s
514 |     R2_s = 1 - np.sum((resid_m3 - pred_s)**2) / np.sum(resid_m3**2)
515 |     R2_shuffle.append(R2_s)

```

```

516 |
517 | R2_shuffle = np.array(R2_shuffle)
518 | p_shuffle = np.mean(R2_shuffle &gt;= R2_m6on3)
519 | print(f" Real R2(M6 on M3 residuals) = {R2_m6on3:.4f}")
520 | print(f" Shuffle: mean={np.mean(R2_shuffle):.4f}, max={np.max(R2_shuffle):.4f}")
521 | print(f" p = {p_shuffle:.4f}")
522 | if p_shuffle &lt; 0.01:
523 |     print(" -&gt; M6=M3oooooooooooooooo")
524 | else:
525 |     print(" -&gt; M6=M3oooooooooooooooo")
526 |
527 | # =====
528 | # oooo
529 | # =====
530 | print("\n" + "=" * 70)
531 | print("oooo")
532 | print("=" * 70)
533 |
534 | print(f"""
535 | oooo:
536 | T1 oooo: rho = {rho_resid:+.4f}
537 | T2 Combined dAICc: vs M3 = {dAICc_comb_vs_m3:+.1f}, vs M6 = {dAICc_comb_vs_m6:+.1f}
538 | T2b L00-CV ratio: M3={mse_m3/mse_null:.3f}, M6={mse_m6/mse_null:.3f}, Comb={mse_comb/mse_null:.3f}
539 | T3 oR2: M6-&gt;M3oo = {R2_m6on3:.4f}, M3-&gt;M6oo = {R2_m3on6:.4f}
540 | T4 hRooo: obs={rho_obs:+.4f}, M3={rho_m3:+.4f}, M6={rho_m6:+.4f}, Comb={rho_comb:+.4f}
541 | T5 oooo: rho(M3,M6) = {rho_indicators:+.4f}
542 | T6 oooo: M6-M8 = {rho_68:+.3f}
543 | T7 oooo: p = {p_shuffle:.4f}
544 |
545 | oooo:
546 | ooo: oooo &lt; 0.5, Combined dAICc &lt; -2 vs both, ooR2 &gt; 0.05, p(T7) &lt; 0.01
547 | oo: oooo &gt; 0.8, Combined dAICc &gt;= 0 vs best, ooR2 &lt; 0.02
548 | oo: oo
549 | """)
550 |
551 | # =====
552 | # o
553 | # =====
554 | try:
555 |     import matplotlib
556 |     matplotlib.use('Agg')
557 |     import matplotlib.pyplot as plt
558 |
559 |     fig, axes = plt.subplots(2, 2, figsize=(12, 10))
560 |     fig.suptitle('M3 vs M6 Independence Test', fontsize=14)
561 |
562 |     # (a) M3 residual vs M6 residual
563 |     ax = axes[0, 0]
564 |     ax.scatter(resid_m3, resid_m6, s=10, alpha=0.5, c='steelblue')
565 |     ax.set_xlabel('M3 residual')
566 |     ax.set_ylabel('M6 residual')
567 |     ax.set_title(f'Residual correlation (rho={rho_resid:+.3f})')
568 |     lim = max(abs(resid_m3).max(), abs(resid_m6).max()) * 1.1
569 |     ax.set_xlim(-lim, lim)
570 |     ax.set_ylim(-lim, lim)
571 |     ax.plot([-lim, lim], [-lim, lim], 'k--', lw=0.5)
572 |
573 |     # (b) M3 indicator vs M6 indicator
574 |     ax = axes[0, 1]
575 |     ax.scatter(log_em, log_sk, s=10, alpha=0.5, c='steelblue')
576 |     ax.set_xlabel('log(eps_max) [M3]')
577 |     ax.set_ylabel('log(strain_kin) [M6]')
578 |     ax.set_title(f'Indicator correlation (rho={rho_indicators:+.3f})')
579 |
580 |     # (c) hR partial correlation comparison
581 |     ax = axes[1, 0]
582 |     labels = ['gc_obs', 'M3', 'M6', 'M3+M6']
583 |     rhos = [abs(rho_obs), abs(rho_m3), abs(rho_m6), abs(rho_comb)]
584 |     colors_bar = ['grey', 'steelblue', 'coral', 'green']
585 |     ax.bar(labels, rhos, color=colors_bar, alpha=0.7)
586 |     ax.set_ylabel('|rho(gc, hR | vflat)|')
587 |     ax.set_title('hR partial correlation (lower = better)')
588 |
589 |     # (d) L00-CV comparison
590 |     ax = axes[1, 1]
591 |     labels2 = ['null', 'M6', 'M3', 'M3+M6']
592 |     mses = [mse_null, mse_m6, mse_m3, mse_comb]
593 |     ax.bar(labels2, mses, color=['grey', 'coral', 'steelblue', 'green'], alpha=0.7)
594 |     ax.set_ylabel('L00-CV MSE')
595 |     ax.set_title('Cross-validation (lower = better)')
596 |
597 |     plt.tight_layout()
598 |     figpath = os.path.join(BASE, 'M3_M6_independence.png')
599 |     plt.savefig(figpath, dpi=150)
600 |     print(f"\nFigure saved: {figpath}")
601 | except Exception as e:
602 |     print(f"\nFigure error: {e}")
603 |
604 |     print("\noooo")
605 |
606 | if __name__ == '__main__':
607 |     main()

```

Script 5: epsilon_c_derivation.py

目的

M3 の $\epsilon_c \approx 0.072$ を $U(\epsilon;c)$ の構造から理論的に導出する試み

11仮説を系統テスト: H1: U 変曲点 $U'=0$, H2: 非調和応答開始, H3: 降伏点, H4: Duffing 非線形パラメータ, H5: MOND 補間展開, H6: 有限ディスク端効果, H7: c 分布, H8: 理論定数照合, H9: $x_c=1/200$, H10: 遷移構造, H11: Taylor 弾性限界。データ不要の純粋理論計算 (コンテナ内実行)。

結果

全て不成功 (X級)。根本原因: $\epsilon_{proxy} = \sqrt{gN/gc}$ と ϵ_{fold} の非線形写像により、 $U(\epsilon;c)$ の全特異点が $\epsilon_{proxy} \approx 0$ に帰着する。H9 のみ保留: $1/(10 \sqrt{2}) = 0.0707$ vs 観測 0.072 (差 1.8%) だが理論的根拠なし。 ϵ_c は経験的パラメータとして記録。

実行方法

```
python epsilon_c_derivation.py or
```

入力データ: なし (純粋理論計算)

出力: $\epsilon_c_analysis.png$ + 標準出力

ソースコード全文

```
1 | #!/usr/bin/env python3
2 | # -*- coding: utf-8 -*-
3 | """
4 | epsilon_c_derivation.py
5 | =====
6 | M3  $\epsilon_c \approx 0.072$   $U(\epsilon;c)$  ??????????
7 |
8 |  $\epsilon_{proxy} = \sqrt{gN/gc\_deep}$   $U(\epsilon;c)$   $\epsilon_{fold}$ 
9 |  $x = gN/gc = (c-1+\epsilon_{fold}^2)/(1-\epsilon_{fold})$ 
10 |  $\epsilon_{proxy} = \sqrt{x} = \sqrt{(c-1+\epsilon_{fold}^2)/(1-\epsilon_{fold})}$ 
11 |
12 |  $\epsilon_c$ :
13 | H1: U  $U'=0$ 
14 | H2: (anharmonic/harmonic ratio)
15 | H3:
16 | H4: free energy landscape
17 | H5: MOND
18 | H6: +
19 | """
20 |
21 | import numpy as np
22 | from scipy.optimize import brentq, minimize_scalar
23 | import matplotlib
24 | matplotlib.use('Agg')
25 | import matplotlib.pyplot as plt
26 |
27 | # =====
28 | # 1.  $U(\epsilon;c)$ 
29 | # =====
30 | def U(eps, c):
31 |     """Free energy"""
32 |     return -eps - eps**2/2 - c * np.log(1 - eps)
33 |
34 | def U1(eps, c):
35 |     """U' = dU/dε"""
36 |     return -1 - eps + c / (1 - eps)
37 |
38 | def U2(eps, c):
39 |     """U'' = d²U/dε²"""
40 |     return -1 + c / (1 - eps)**2
41 |
42 | def U3(eps, c):
43 |     """U''' = d³U/dε³"""
44 |     return 2 * c / (1 - eps)**3
45 |
46 | def U4(eps, c):
47 |     """U'''' = d⁴U/dε⁴"""
48 |     return 6 * c / (1 - eps)**4
49 |
50 | #  $U'(\epsilon_{eq}) = 0 \rightarrow \epsilon_{eq}$ 
51 | def eps_eq(c):
52 |     """ $U'(\epsilon;c) = 0$ """
53 |     if c >= 1:
54 |         return 0.0
55 |     #  $-1 - \epsilon + c/(1-\epsilon) = 0 \rightarrow \epsilon^2 - \epsilon + (c-1) = 0 \dots$  no
56 |     # Actually:  $c/(1-\epsilon) = 1+\epsilon \rightarrow c = (1-\epsilon)(1+\epsilon) = 1-\epsilon^2$ 
57 |     #  $\rightarrow \epsilon = \sqrt{1-c}$ 
```

```

58 |     return np.sqrt(max(1 - c, 0))
59 |
60 | # x = gN/gc from equilibrium
61 | def x_from_eps_fold(eps_f, c):
62 |     """gN/gc = (c - 1 + ε²)/(1 - ε)"""
63 |     return (c - 1 + eps_f**2) / (1 - eps_f)
64 |
65 | # ε_proxy = sqrt(x)
66 | def eps_proxy_from_fold(eps_f, c):
67 |     x = x_from_eps_fold(eps_f, c)
68 |     return np.sqrt(max(x, 0))
69 |
70 | # ε_fold from ε_proxy (inverse)
71 | def eps_fold_from_proxy(eps_p, c):
72 |     """x = eps_p² = (c-1+ε²)/(1-ε)
73 |     → ε² + eps_p²ε + (c-1-eps_p²) = 0"""
74 |     x = eps_p**2
75 |     # ε² + xε - (1-c+x) = 0 ... let me redo
76 |     # (c-1+ε²)/(1-ε) = x
77 |     # c - 1 + ε² = x(1-ε) = x - xε
78 |     # ε² + xε + (c-1-x) = 0
79 |     a, b, cc2 = 1, x, (c - 1 - x)
80 |     disc = b**2 - 4*a*cc2
81 |     if disc <= 0:
82 |         return None
83 |     return (-b + np.sqrt(disc)) / (2*a)
84 |
85 | print("=" * 70)
86 | print("ε_c ===== U(ε;c) =====")
87 | print("=" * 70)
88 |
89 | # =====
90 | # 2. ===== U(ε;c) =====
91 | # =====
92 | print("\n--- ===== ---")
93 | for c in [0.1, 0.24, 0.5, 0.8, 1.0]:
94 |     eeq = eps_eq(c)
95 |     u2 = U2(eeq, c) if eeq <= 0.99 else float('inf')
96 |     print(f" c={c:.2f}: ε_eq={eeq:.4f}, U'(ε_eq)={u2:.4f}")
97 |
98 | # =====
99 | # H1: U''=0 =====
100 | # =====
101 | print("\n" + "=" * 70)
102 | print("H1: U ===== U''(ε;c) = 0")
103 | print("=" * 70)
104 | print("U'(ε;c) = -1 + c/(1-ε)²")
105 | print("U'' = 0 → ε_infl = 1 - sqrt(c)")
106 |
107 | for c in [0.1, 0.24, 0.5, 0.8, 1.0]:
108 |     eps_infl = 1 - np.sqrt(c)
109 |     eps_p = eps_proxy_from_fold(eps_infl, c) if eps_infl >= 0 else 0
110 |     print(f" c={c:.2f}: ε_fold_infl={eps_infl:.4f}, ε_proxy_infl={eps_p:.4f}")
111 |
112 | print("\n → ===== ε_proxy ◻ c =====0.072 =====H1 =====")
113 |
114 | # =====
115 | # H2: ===== |U''δε| / |U''| = 1 =====
116 | # =====
117 | print("\n" + "=" * 70)
118 | print("H2: =====")
119 | print("=" * 70)
120 | print("ε_eq ===== δε ◻ |U''(ε_eq)δε/U''(ε_eq)| = 1 ===== δε")
121 |
122 | for c in [0.1, 0.24, 0.5, 0.8, 1.0]:
123 |     eeq = eps_eq(c)
124 |     if eeq >= 0.99: continue
125 |     u2v = U2(eeq, c)
126 |     u3v = U3(eeq, c)
127 |     if abs(u3v) >= 0:
128 |         delta_eps = abs(u2v / u3v)
129 |         # ===== ε_fold
130 |         eps_anh = eeq - delta_eps # ε < ε_eq ◻
131 |         if eps_anh >= 0:
132 |             eps_p = eps_proxy_from_fold(eps_anh, c)
133 |         else:
134 |             eps_p = None
135 |         ep_str = f"{eps_p:.4f}" if eps_p is not None else "N/A"
136 |         print(f" c={c:.2f}: eps_eq={eeq:.4f}, delta_eps={delta_eps:.4f}, "
137 |               f"eps_fold_anh={eps_anh:.4f}, eps_proxy={ep_str}")
138 |
139 | # =====
140 | # H3: =====
141 | # =====
142 | print("\n" + "=" * 70)
143 | print("H3: ===== (stress-strain yield)")
144 | print("=" * 70)
145 | print("σ(ε) = U'(ε;c) ===== |σ_max| =====")
146 | print("dσ/dε = U''(ε;c) = 0 → ε_yield = 1 - sqrt(c) (= H1=====)")
147 | print("→ H1=====")
148 |
149 | # =====99%=====

```

```

150 | print("\n      :  $\sigma(\epsilon)/\sigma_{\text{linear}}$       10%")
151 | for c in [0.1, 0.24, 0.5, 0.8, 1.0]:
152 |     eeq = eps_eq(c)
153 |     if eeq >= 0.99: continue
154 |     u2v = U2(eeq, c)
155 |
156 |     #  $\sigma_{\text{linear}}(\delta\epsilon) = U'(\epsilon_{\text{eq}}) \times \delta\epsilon$ 
157 |     #  $\sigma_{\text{actual}}(\delta\epsilon) = U'(\epsilon_{\text{eq}} + \delta\epsilon) - U'(\epsilon_{\text{eq}}) = U'(\epsilon_{\text{eq}} + \delta\epsilon)$  [since  $U'(\epsilon_{\text{eq}})=0$ ]
158 |     # deviation =  $|\sigma_{\text{actual}} - \sigma_{\text{linear}}| / |\sigma_{\text{linear}}| = 10\%$ 
159 |
160 |     for delta in np.linspace(0.001, 0.5, 1000):
161 |         eps_test = eeq - delta # going toward smaller  $\epsilon$  (larger gN)
162 |         if eps_test <= 0.01: break
163 |         sigma_lin = u2v * (-delta)
164 |         sigma_act = U1(eps_test, c)
165 |         if abs(sigma_lin) > 0:
166 |             dev = abs(sigma_act - sigma_lin) / abs(sigma_lin)
167 |             if dev > 0.1:
168 |                 x_yield = x_from_eps_fold(eps_test, c)
169 |                 eps_p_yield = np.sqrt(max(x_yield, 0))
170 |                 print(f" c={c:.2f}: 10% at  $\delta\epsilon={\text{delta}:.4f}$ , "
171 |                       f" $\epsilon_{\text{fold}}={\text{eps\_test}:.4f}$ ,  $x={x\_yield:.4f}$ ,  $\epsilon_{\text{proxy}}={\text{eps\_p\_yield}:.4f}$ ")
172 |                 break
173 |
174 | # =====
175 | # H4:  $U''^2/(U' \times U''') = 1$  (Duffing)
176 | # =====
177 | print("\n" + "=" * 70)
178 | print("H4: Duffing  $\gamma = U''^2/(U' \times U''')$ ")
179 | print("=" * 70)
180 |
181 | for c in [0.1, 0.24, 0.5, 0.8, 1.0]:
182 |     eeq = eps_eq(c)
183 |     if eeq >= 0.99: continue
184 |     u2v = U2(eeq, c)
185 |     u3v = U3(eeq, c)
186 |     u4v = U4(eeq, c)
187 |     gamma = u3v**2 / (u2v * u4v) if u2v * u4v != 0 else 0
188 |     print(f" c={c:.2f}:  $\gamma(\epsilon_{\text{eq}}) = {\text{gamma}:.4f}$ ")
189 |
190 | print("  $\gamma \propto c$  ")
191 |
192 | # =====
193 | # H5: MOND
194 | # =====
195 | print("\n" + "=" * 70)
196 | print("H5: MOND")
197 | print("=" * 70)
198 | print("g_eff/gc = (x + sqrt(x^2 + 4x))/2  $\square$  x= $\epsilon_{\text{proxy}}^2$  ")
199 | print("  $\rightarrow$  g_eff/gc  $\approx$  sqrt(x) + x/(4sqrt(x)) - x^2/(32x^(3/2)) + ...")
200 | print("  $\rightarrow$  = sqrt(x)(1 + sqrt(x)/4 - x/(32) + ...)")
201 | print("3/2 = 1  $\square$  x:")
202 | # 2nd order correction: x/4 at order sqrt(x)
203 | # ratio of 2nd to 1st: sqrt(x)/4
204 | # ratio = 1  $\rightarrow$  sqrt(x) = 4  $\rightarrow$  x = 16 (Newtonian regime, not relevant)
205 | # ratio = 0.1  $\rightarrow$  sqrt(x) = 0.4  $\rightarrow$  x = 0.16
206 |
207 | # More carefully:
208 | # v(x) = (1 + sqrt(1+4/x))/2 (MOND interpolation)
209 | # For x << 1; 1: v  $\approx$  1/sqrt(x) - 1/(2x) + ... so g_eff = gN * v  $\approx$  sqrt(gN*gc)
210 | # The correction to deep-MOND:
211 | # g_eff = sqrt(gN*gc) * (1 + sqrt(gN/gc)/4 + ...)
212 | # The relative correction = sqrt(x)/4
213 |
214 | # At  $\epsilon_{\text{proxy}} = 0.072$ : x = 0.00518, sqrt(x) = 0.072
215 | # correction = 0.072/4 = 0.018 = 1.8%
216 |
217 | print(f"\n  $\epsilon_{\text{proxy}}=0.072 \rightarrow x=0.00518$ ")
218 | print(f" MOND  $\square$  = sqrt(x)/4 = {0.072/4:.4f} = 1.8%")
219 | print(f"  $\rightarrow$  MOND")
220 |
221 | # =====
222 | # H6:
223 | # =====
224 | print("\n" + "=" * 70)
225 | print("H6:")
226 | print("=" * 70)
227 | print("gN(r) = vflat^2/r *  $\Sigma_{\square}$ ")
228 | print("  $\epsilon_{\text{proxy}}(r) = \text{sqrt}(gN(r)/gc)$ ")
229 | print("  $\epsilon_{\text{proxy}_c} = 0.072 \rightarrow gN/gc = 0.005$ ")
230 | print("  $\rightarrow$  r/hR ")
231 |
232 | # For exponential disk: gN  $\square$  exp(-r/hR) *  $\square$  / r
233 | # At large r: gN  $\rightarrow$  0 monotonically
234 | # gN(r)/gc  $\approx$  (vflat^2/r) * (r/hR)^2 * exp(-r/hR) / gc ( $\square$ )
235 | # The peak of gN is around r  $\sim$  2.2 hR
236 | # gN_peak / gc  $\approx$  typical gc_obs/a0  $\approx$  0.24 means gN_peak  $\approx$  0.24*a0*f
237 | # where f depends on the galaxy
238 |
239 | print("  $\epsilon_{\text{proxy}_c} = 0.072 \square$  gN/gc  $\approx$  0.005 ")
240 | print(" (gc=0.24 a0) r/hR  $\sim$  5-8 ")
241 | print("  $\rightarrow$  ")

```



```

334 |
335 |    $\epsilon_c^2 \approx 1/200 \approx 0.005$ 
336 |    $\rightarrow \epsilon_c \approx 1/\sqrt{200} = 1/(10\sqrt{2}) \approx 0.0707$ 
337 |   """)
338 |
339 |   val_theory = 1/(10*np.sqrt(2))
340 |   print(f"   : 1/(10sqrt(2)) = {val_theory:.5f}")
341 |   print(f"   :  $\epsilon_c \approx 0.072$ ")
342 |   print(f"   : {abs(0.072 - val_theory):.5f} ({abs(0.072-val_theory)/0.072*100:.1f}%)")
343 |
344 |   # But 1/(10sqrt(2)) is ad hoc. Let's look deeper.
345 |
346 |   # =====
347 |   # H10:   gc_deep vs gc_obs   :
348 |   # =====
349 |   print("\n" + "=" * 70)
350 |   print("H10: deep-MOND  $\rightarrow$  full MOND :")
351 |   print("=" * 70)
352 |   print("")
353 |   gc_obs = gc_deep * ( )
354 |   M3 :  $\epsilon_{\max}$  &gt;  $\epsilon_c$  = gc = 23%
355 |
356 |    $\epsilon_{\max} = \sqrt{gN_{\max} / gc_{\text{deep}}} \approx \sqrt{v_{\text{flat}}^2 / (hR \times gc_{\text{deep}})}$  [r-hR]
357 |
358 |    $\epsilon_{\max}$  &gt;  $\epsilon_c$  :
359 |    $gN_{\max} / gc_{\text{deep}}$  &gt;  $\epsilon_c^2$ 
360 |    $\rightarrow gN_{\max}$  &gt;  $\epsilon_c^2 \times gc_{\text{deep}}$ 
361 |
362 |   gc_deep - gc_obs (deep MOND) -  $\eta \times \sqrt{a_0 \times v_{\text{flat}}^2 / hR}$  :
363 |    $gN_{\max} \sim v_{\text{flat}}^2 / hR$  (hR)
364 |   gc_deep  $\sim \eta \times \sqrt{a_0 \times v_{\text{flat}}^2 / hR}$ 
365 |
366 |    $\epsilon_{\max}^2 = gN_{\max} / gc_{\text{deep}} \sim (v_{\text{flat}}^2 / hR) / (\eta \times \sqrt{a_0 \times v_{\text{flat}}^2 / hR})$ 
367 |   =  $\sqrt{v_{\text{flat}}^2 / hR} / (\eta \times \sqrt{a_0})$ 
368 |   =  $\sqrt{G \Sigma_0 / a_0} / \eta$ 
369 |
370 |    $\epsilon_{\max}$  &gt;  $\epsilon_c$  :
371 |    $\sqrt{G \Sigma_0 / a_0}$  &gt;  $\eta \times \epsilon_c$ 
372 |    $G \Sigma_0 / a_0$  &gt;  $\eta^2 \times \epsilon_c^2$ 
373 |
374 |    $\eta \sim 0.74$ ,  $\epsilon_c \sim 0.072$  :
375 |    $\eta^2 \times \epsilon_c^2 = 0.74^2 \times 0.072^2 = 0.548 \times 0.00518 = 0.00284$ 
376 |    $\rightarrow G \Sigma_0$  &gt;  $0.00284 \times a_0$ 
377 |    $\rightarrow G \Sigma_0$  &gt;  $3.4 \times 10^{-13} \text{ m/s}^2$ 
378 |   """)
379 |
380 |   # This gives a surface density threshold
381 |   GSigma_threshold = 0.74**2 * 0.072**2 * 1.2e-10
382 |   print(f"   :  $G \Sigma_0$  &gt; {GSigma_threshold:.2e} m/s2")
383 |   print(f"   :  $v_{\text{flat}}^2 / hR$  :  $v_{\text{flat}}^2 / hR$  &gt; {GSigma_threshold:.2e} m/s2")
384 |   print(f"   :  $v_{\text{flat}} \sim 100 \text{ km/s}$ ,  $hR \sim 3 \text{ kpc}$  :")
385 |   vflat_test = 100e3 # m/s
386 |   hR_test = 3 * 3.086e19 # m
387 |   GSigma_test = vflat_test**2 / hR_test
388 |   print(f"   :  $G \Sigma_0 = \{GSigma\_test:.2e\} \text{ m/s}^2$  {'&gt;'} if  $G \Sigma_0$  &gt;  $G \Sigma_0$  else {'&lt;'} :")
389 |   print(f"   :  $\rightarrow$  SPARC :")
390 |
391 |   # =====
392 |   # H11:   Taylor :
393 |   # =====
394 |   print("\n" + "=" * 70)
395 |   print("H11: U( $\epsilon$ ;c) = Taylor :")
396 |   print("=" * 70)
397 |
398 |   print("")
399 |    $\epsilon_{\text{fold}} = \epsilon_{\text{eq}} - \delta\epsilon$  ( )
400 |
401 |    $U(\epsilon_{\text{eq}} - \delta\epsilon; c) = U(\epsilon_{\text{eq}}) + (1/2)U''(\epsilon_{\text{eq}})\delta\epsilon^2 - (1/6)U'''(\epsilon_{\text{eq}})\delta\epsilon^3 + \dots$ 
402 |
403 |   :
404 |    $|U''\delta\epsilon / (3U'')| \ll 0.1$ 
405 |    $\rightarrow \delta\epsilon \ll 0.3 |U''/U''|$ 
406 |
407 |   + gN :
408 |    $x = gN/gc = U'(\epsilon_{\text{eq}}) \times \delta\epsilon / [gc \times (\text{derivative mapping})]$  ( )
409 |   """)
410 |
411 |   print("\n c : delta_eps_elastic : eps_proxy:")
412 |   header = " :&gt;6s :&gt;8s :&gt;12s :&gt;10s :&gt;14s :&gt;10s".format(
413 |       "c", "eps_eq", "|U2/U3|", "delt_10%", "eps_fold_lim", "eps_proxy")
414 |   print(header)
415 |
416 |   for c in [0.10, 0.15, 0.20, 0.24, 0.30, 0.40, 0.50, 0.70, 0.90]:
417 |       eeq = eps_eq(c)
418 |       if eeq &gt;= 0.99: continue
419 |       u2v = U2(eeq, c)
420 |       u3v = U3(eeq, c)
421 |       ratio_u = abs(u2v / u3v) if u3v != 0 else 999
422 |       delta_10 = 0.3 * ratio_u
423 |       ef_lim = eeq - delta_10
424 |       if ef_lim &gt; 0.01 and ef_lim &lt; 0.99:
425 |           ep = eps_proxy_from_fold(ef_lim, c)

```

```

426 |     else:
427 |         ep = None
428 |         ep_str = f"{ep:.4f}" if ep is not None else "N/A"
429 |         print(f" {c:6.3f} {eeq:8.4f} {ratio_u:12.4f} {delta_10:10.4f} "
430 |               f"{ef_lim:14.4f} {ep_str}&gt;10s}")
431 |
432 | # =====
433 | # oooo
434 | # =====
435 | print("\n" + "=" * 70)
436 | print("oooo")
437 | print("=" * 70)
438 |
439 | print("")
440 | ooooooo:
441 |
442 | H1: Uooo → Coooooooooooo
443 | H2: oooo → ooooooooooε_proxy ooooooooooooo
444 | H3: ooo → H1oooooooo
445 | H4: Duffing → ooooooooooooo
446 | H5: MONDoo → 2ooo 1.8% oooooooooo
447 | H6: ooooo → ooooooooooooo
448 | H7: Coo → ooooooooooooo
449 | H8: oooo → ooooooooooooo
450 | H9: x_c=1/200 → ooooooooooooooooooooo
451 | H10: oooo → ooooooooooooooooooooo
452 | H11: Tayloroooo → Coooooooooooo
453 |
454 | ε_c ≈ 0.072 □ U(ε;c) ooooooooooooo
455 | ooooooooooooo
456 |
457 | ooooooo:
458 | (a) ε_c □ U(ε;c) ooooooooooooooooooooo
459 | (b) ε_c ooooooooooooooooomax/hR ooooooo
460 | (c) ε_c ooo14oooooooooU ooooooooooooo
461 |
462 | oo: ε_c ooooooooooooooooooooo
463 | oooo ε_c ≈ 0.07 +/- ? ooooooooooooooooooooo
464 | "")
465 |
466 | # =====
467 | # □
468 | # =====
469 | fig, axes = plt.subplots(2, 2, figsize=(12, 10))
470 |
471 | # (a) U(ε;c) for various c
472 | ax = axes[0, 0]
473 | eps_range = np.linspace(0.01, 0.95, 200)
474 | for c in [0.1, 0.24, 0.5, 0.8]:
475 |     ax.plot(eps_range, U(eps_range, c), label=f'c={c}')
476 | ax.set_xlabel('epsilon_fold')
477 | ax.set_ylabel('U(epsilon;c)')
478 | ax.set_title('Free energy U(epsilon;c)')
479 | ax.legend()
480 | ax.set_ylim(-5, 2)
481 |
482 | # (b) U''(ε;c) – stiffness
483 | ax = axes[0, 1]
484 | for c in [0.1, 0.24, 0.5, 0.8]:
485 |     ax.plot(eps_range, U2(eps_range, c), label=f'c={c}')
486 | ax.axhline(0, color='grey', ls='--')
487 | ax.set_xlabel('epsilon_fold')
488 | ax.set_ylabel('U''(epsilon;c)')
489 | ax.set_title('Local stiffness')
490 | ax.legend()
491 | ax.set_ylim(-2, 20)
492 |
493 | # (c) ε_proxy vs ε_fold for various c
494 | ax = axes[1, 0]
495 | for c in [0.1, 0.24, 0.5, 0.8]:
496 |     eeq = eps_eq(c)
497 |     ef_range = np.linspace(0.01, eeq-0.01, 100)
498 |     ep = [eps_proxy_from_fold(ef, c) for ef in ef_range]
499 |     ax.plot(ef_range, ep, label=f'c={c}')
500 | ax.axhline(0.072, color='red', ls='--', lw=2, label='epsilon_c=0.072')
501 | ax.set_xlabel('epsilon_fold')
502 | ax.set_ylabel('epsilon_proxy = sqrt(gN/gc)')
503 | ax.set_title('epsilon_proxy vs epsilon_fold')
504 | ax.legend()
505 |
506 | # (d) Non-linearity ratio
507 | ax = axes[1, 1]
508 | for c in [0.1, 0.24, 0.5, 0.8]:
509 |     eeq = eps_eq(c)
510 |     ef_range = np.linspace(0.01, eeq-0.01, 100)
511 |     ratios = []
512 |     ep_vals = []
513 |     for ef in ef_range:
514 |         ep = eps_proxy_from_fold(ef, c)
515 |         if ep &gt; 0:
516 |             nl = abs(U3(ef, c) * (eeq - ef) / U2(ef, c)) if U2(ef, c) != 0 else 0
517 |             ratios.append(nl)

```

```
518 |         ep_vals.append(ep)
519 |     ax.plot(ep_vals, ratios, label=f'c={c}')
520 | ax.axvline(0.072, color='red', ls='--', lw=2)
521 | ax.axhline(1, color='grey', ls='--')
522 | ax.set_xlabel('epsilon_proxy')
523 | ax.set_ylabel('|U3 * delta_eps / U2|')
524 | ax.set_title('Non-linearity ratio')
525 | ax.legend()
526 | ax.set_ylim(0, 3)
527 |
528 | plt.tight_layout()
529 | plt.savefig('/home/claude/epsilon_c_analysis.png', dpi=150)
530 | print("\nFigure saved: epsilon_c_analysis.png")
531 | print("\n====")
```

Script 6: sparc_eta_derivation.py

目的

eta の第一原理導出: 深MOND領域のガス分率仮説の検証

核心的洞察: 深MOND極限で $d \ln(gc)/d \ln(Yd) = -(1 - f_{\text{gas_deep}})$ 。eta $\sim Yd^{-0.41}$ の観測から $f_{\text{gas_deep}} = 0.59$ を予測。テスト項目: (1) SPARC 各銀河の深MOND領域で $f_{\text{gas_deep}}$ を直接計算。(2) 銀河ごとの $\beta_{\text{predicted}}$ vs η_{obs} 。(3) $\eta_{\text{predicted}}$ vs η_{obs} (per-galaxy vs fixed exponent)。(4) $f_{\text{gas_deep}}$ の決定因子。(5) Yd 微分の直接テスト ($Yd \pm 10\%$ の数値微分)。(6) 導出精度評価。

結果

部分的成功: $d \ln(gc)/d \ln(Yd) = -(1-f_{\text{gas}})$ は銀河内で成立 ($r=0.616$, $p=7e-18$)。しかし銀河間回帰 $\beta=-0.41$ の予測には失敗。 $f_{\text{gas_deep}}$ median=0.284 (予測 0.59 の半分)。原因: f_{gas} と Yd の負相関 ($\rho=-0.46$) による Simpson's paradox 的效果。A-1 (eta導出) は A 級維持だが「運動学的に制約された」ことは記録。

実行方法

```
uv run --with scipy --with matplotlib python sparc_eta_derivation.py
```

入力データ: sparc_gc.csv + Rotmod_LTG/*.dat

出力: eta_derivation_results.png + 標準出力

ソースコード全文

```
1 | #!/usr/bin/env python3
2 | # -*- coding: utf-8 -*-
3 | """
4 | sparc_eta_derivation.py
5 | =====
6 | eta oooooo: oMONDooooooooo
7 |
8 | ■ ooooo:
9 |
10 | oMONDoo: gc_deep(r) = g_eff(r)^2 / gN(r)
11 |
12 | gN(r) = Yd x v_disk(r)^2/r + v_gas(r)^2/r + v_bul(r)^2/r
13 |
14 | Yd oooooooooo gc_deep ooo:
15 |
16 | d ln(gc_deep) / d ln(Yd) = -Yd x v_disk^2 / (Yd x v_disk^2 + v_gas^2 + v_bul^2)
17 |                               = -(1 - f_gas_deep)
18 |
19 | ooo f_gas_deep = (v_gas^2 + v_bul^2) / gN ooMONDooooYdooooooooo
20 |
21 | η = gc / sqrt(a0 x G Sigma0) o G Sigma0 = vflat^2/hR o Yd oooooo:
22 |
23 | d ln(η) / d ln(Yd) = d ln(gc) / d ln(Yd) = -(1 - f_gas_deep)
24 |
25 | oo: η o Yd^(-0.41) → d ln(η)/d ln(Yd) = -0.41
26 |
27 | oo: f_gas_deep = 1 - 0.41 = 0.59
28 |
29 | ooo: oMONDooooo(+bulge)oooo59%ooo
30 | η o Yd^(-0.41) oooooo
31 |
32 | ■ ooo:
33 | (1) SPARC oooooMONDooo f_gas ooooo
34 | (2) median f_gas_deep ≈ 0.59 oooooo
35 | (3) ooooo f_gas_deep vs Yd ooooo
36 | (4) eta_predicted = C x Yd^(-(1-f_gas_deep)) oooooo
37 |
38 | oo: uv run --with scipy --with matplotlib python sparc_eta_derivation.py
39 | """
40 |
41 | import os, sys
42 | import numpy as np
43 | from scipy.optimize import minimize
44 | from scipy.stats import spearmanr, pearsonr
45 | import warnings
46 | warnings.filterwarnings('ignore')
47 |
48 | BASE = r"D:\ooooo\ooooo\ooooo\ooooo\ooooo"
49 | ROTMOD = os.path.join(BASE, "Rotmod_LTG")
50 | GC_CSV = os.path.join(BASE, "sparc_gc.csv")
51 | a0 = 1.2e-10
52 |
53 | # =====
54 | # oooooo
55 | # =====
```

```

56 | def load_gc_csv():
57 |     import csv
58 |     data = {}
59 |     with open(GC_CSV, 'r') as f:
60 |         reader = csv.DictReader(f)
61 |         for row in reader:
62 |             name = None
63 |             for k in row:
64 |                 if 'gal' in k.lower() or 'name' in k.lower():
65 |                     name = row[k].strip(); break
66 |             if not name: continue
67 |             gc_val = None
68 |             for k in ['gc', 'gc_obs', 'g_c', 'gc_a0']:
69 |                 if k in row and row[k]:
70 |                     try: gc_val = float(row[k]); break
71 |                 except: pass
72 |             if gc_val is None: continue
73 |             vflat, hR, Yd = None, None, None
74 |             for k in ['vflat', 'vflat', 'v_flat']:
75 |                 if k in row and row[k]:
76 |                     try: vflat = float(row[k]); break
77 |                 except: pass
78 |             for k in ['hR', 'Reff', 'h_R', 'hr']:
79 |                 if k in row and row[k]:
80 |                     try: hR = float(row[k]); break
81 |                 except: pass
82 |             for k in ['Yd', 'yd', 'Y_d', 'SPS']:
83 |                 if k in row and row[k]:
84 |                     try: Yd = float(row[k]); break
85 |                 except: pass
86 |             data[name] = {'gc': gc_val, 'vflat': vflat, 'hR': hR, 'Yd': Yd}
87 |             gc_vals = [d['gc'] for d in data.values()]
88 |             if np.median(gc_vals) > 1e-5:
89 |                 for n in data: data[n]['gc'] *= a0
90 |             return data
91 |
92 | def load_rotcurve(gname):
93 |     fname = os.path.join(ROTMOD, f"{gname}_rotmod.dat")
94 |     if not os.path.exists(fname): return None
95 |     cols = {k: [] for k in ['r', 'vobs', 'vgas', 'vdisk', 'vbul']}
96 |     with open(fname, 'r') as f:
97 |         for line in f:
98 |             line = line.strip()
99 |             if not line or line.startswith('#'): continue
100 |             p = line.split()
101 |             if len(p) < 6: continue
102 |             try:
103 |                 cols['r'].append(float(p[0]))
104 |                 cols['vobs'].append(float(p[1]))
105 |                 cols['vgas'].append(float(p[3]))
106 |                 cols['vdisk'].append(float(p[4]))
107 |                 cols['vbul'].append(float(p[5]))
108 |             except: continue
109 |             if len(cols['r']) < 5: return None
110 |             return {k: np.array(v) for k, v in cols.items()}
111 |
112 | def compute_gc_deep(rc, Yd=0.5):
113 |     r_m = rc['r'] * 3.086e19
114 |     v_bar2 = (Yd * rc['vdisk']**2 + rc['vgas']**2 + rc['vbul']**2) * 1e6
115 |     v_obs2 = (rc['vobs'] * 1e3)**2
116 |     gN = v_bar2 / r_m
117 |     gobs = v_obs2 / r_m
118 |     mask = gN > 0
119 |     if np.sum(mask) < 3: return None
120 |     gc_pts = gobs[mask]**2 / gN[mask]
121 |     gc_med = np.median(gc_pts)
122 |     if gc_med <= 0: return None
123 |     x = gN[mask] / gc_med
124 |     deep = x < 1.0
125 |     return np.median(gc_pts[deep]) if np.sum(deep) >= 3 else gc_med
126 |
127 | # =====
128 | #   ooo
129 | # =====
130 | def main():
131 |     print("=" * 70)
132 |     print("eta oooooo: oMOND oooooo")
133 |     print("=" * 70)
134 |
135 |     print("""
136 |     ■ ooooo:
137 |     eta ~ Yd^beta where beta = -(1 - f_gas_deep)
138 |     o: beta = -0.41 → f_gas_deep = 0.59
139 |
140 |     f_gas_deep = &lt;v_gas^2 + v_bul^2&gt; / &lt;Yd*v_disk^2 + v_gas^2 + v_bul^2&gt;
141 |     ooo &lt;...&gt; oMONDooo x = gN/gc &lt;1oooooo
142 |     """)
143 |
144 |     gc_data = load_gc_csv()
145 |
146 |     results = []
147 |     for gname in sorted(gc_data.keys()):

```

```

148 |     gd = gc_data[gname]
149 |     gc_obs, vflat, hR = gd['gc'], gd.get('vflat'), gd.get('hR')
150 |     Yd = gd.get('Yd') or 0.5
151 |     if gc_obs &lt;= 0 or not vflat or not hR or vflat &lt;= 0 or hR &lt;= 0:
152 |         continue
153 |     rc = load_rotcurve(gname)
154 |     if rc is None: continue
155 |     gc_deep = compute_gc_deep(rc, Yd)
156 |     if gc_deep is None or gc_deep &lt;= 0: continue
157 |
158 |     # ===== gN =====
159 |     r_m = rc['r'] * 3.086e19
160 |     v_disk2 = rc['vdisk']**2 * 1e6 # (m/s)^2
161 |     v_gas2 = rc['vgas']**2 * 1e6
162 |     v_bul2 = rc['vbul']**2 * 1e6
163 |
164 |     gN_disk = Yd * v_disk2 / r_m # Yd=====
165 |     gN_gas = v_gas2 / r_m # Yd=====
166 |     gN_bul = v_bul2 / r_m # Yd=====
167 |     gN_total = gN_disk + gN_gas + gN_bul
168 |
169 |     # MOND===== (x = gN/gc &lt;= 1)
170 |     x = gN_total / gc_deep
171 |     deep_mask = (x &lt;= 1.0) &amp; (gN_total &gt; 0)
172 |
173 |     if np.sum(deep_mask) &lt;= 3:
174 |         continue
175 |
176 |     # f_gas_deep = (Yd) / (gN) at deep MOND points
177 |     gN_nonYd = gN_gas[deep_mask] + gN_bul[deep_mask]
178 |     gN_tot_deep = gN_total[deep_mask]
179 |
180 |     # =====gN =====
181 |     f_gas_deep = np.sum(gN_nonYd) / np.sum(gN_tot_deep)
182 |
183 |     # gN^2 =====gc_deep =====
184 |     weights = gN_tot_deep**2 # gc_deep = gobs^2/gN gN^2 =====
185 |     f_gas_deep_w = np.sum(gN_nonYd * weights) / np.sum(gN_tot_deep * weights)
186 |
187 |     # ===== f_gas =====
188 |     f_gas_pts = gN_nonYd / gN_tot_deep
189 |
190 |     # eta_obs
191 |     GSigma0 = (vflat * 1e3)**2 / (hR * 3.086e19) # m/s^2
192 |     eta_obs = gc_obs / np.sqrt(a0 * GSigma0)
193 |
194 |     # beta_predicted = -(1 - f_gas_deep)
195 |     beta_pred = -(1 - f_gas_deep)
196 |
197 |     results.append({
198 |         'name': gname,
199 |         'gc_obs': gc_obs,
200 |         'gc_deep': gc_deep,
201 |         'vflat': vflat,
202 |         'hR': hR,
203 |         'Yd': Yd,
204 |         'f_gas_deep': f_gas_deep,
205 |         'f_gas_deep_w': f_gas_deep_w,
206 |         'f_gas_median': np.median(f_gas_pts),
207 |         'beta_pred': beta_pred,
208 |         'eta_obs': eta_obs,
209 |         'n_deep': int(np.sum(deep_mask)),
210 |         'disk_frac_deep': 1 - f_gas_deep,
211 |     })
212 |
213 |     N = len(results)
214 |     print(f"====: {N}\n")
215 |
216 |     f_gas = np.array([r['f_gas_deep'] for r in results])
217 |     f_gas_w = np.array([r['f_gas_deep_w'] for r in results])
218 |     f_gas_med = np.array([r['f_gas_median'] for r in results])
219 |     beta_pred = np.array([r['beta_pred'] for r in results])
220 |     eta_obs = np.array([r['eta_obs'] for r in results])
221 |     Yd = np.array([r['Yd'] for r in results])
222 |     vflat = np.array([r['vflat'] for r in results])
223 |     hR = np.array([r['hR'] for r in results])
224 |
225 |     # =====
226 |     # (1) f_gas_deep =====
227 |     # =====
228 |     print("-" * 70)
229 |     print("(1) f_gas_deep =====")
230 |     print("-" * 70)
231 |     print(f" mean = {np.mean(f_gas):.4f}")
232 |     print(f" median = {np.median(f_gas):.4f}")
233 |     print(f" std = {np.std(f_gas):.4f}")
234 |     print(f" IQR = [{np.percentile(f_gas,25):.4f}, {np.percentile(f_gas,75):.4f}]")
235 |     print(f" =====: median = {np.median(f_gas_w):.4f}")
236 |     print(f" ===== = 0.59")
237 |     print(f" = {abs(np.median(f_gas) - 0.59):.4f} ({abs(np.median(f_gas)-0.59)/0.59*100:.1f}%)")
238 |
239 |     # =====

```

```

240 | # (2) beta_predicted vs eta_obs
241 | # =====
242 | print("\n" + "-" * 70)
243 | print("(2) beta_obs")
244 | print("-" * 70)
245 |
246 | # beta
247 | log_eta = np.log10(eta_obs)
248 | log_Yd = np.log10(Yd)
249 |
250 | # beta (OLS)
251 | from numpy.linalg import lstsq
252 | X = np.column_stack([log_Yd, np.ones(N)])
253 | sol, _, _, _ = lstsq(X, log_eta, rcond=None)
254 | beta_obs_global = sol[0]
255 |
256 | print(f" beta_obs (global OLS) = {beta_obs_global:.4f}")
257 | print(f" beta_obs (expected) = -0.41 (from eta~Yd^-0.41)")
258 | print(f" beta_pred (median) = {np.median(beta_pred):.4f}")
259 | print(f" beta_pred = -(1-f_gas) = {-1+np.median(f_gas):.4f}")
260 |
261 | # =====
262 | # (3) eta_predicted vs eta_obs
263 | # =====
264 | print("\n" + "-" * 70)
265 | print("(3) eta_obs")
266 | print("-" * 70)
267 |
268 | # Model A: eta = C x Yd^(-0.41) [empirical, fixed exponent]
269 | log_eta_pred_A = -0.41 * log_Yd
270 | C_A = 10**(np.median(log_eta - log_eta_pred_A))
271 | pred_A = np.log10(C_A) + log_eta_pred_A
272 | R2_A = 1 - np.sum((log_eta - pred_A)**2) / np.sum((log_eta - np.mean(log_eta))**2)
273 |
274 | # Model B: eta = C x Yd^(-(1-f_gas_i)) [per-galaxy gas fraction]
275 | log_eta_pred_B = beta_pred * log_Yd
276 | C_B = 10**(np.median(log_eta - log_eta_pred_B))
277 | pred_B = np.log10(C_B) + log_eta_pred_B
278 | R2_B = 1 - np.sum((log_eta - pred_B)**2) / np.sum((log_eta - np.mean(log_eta))**2)
279 |
280 | # Model C: eta = C x Yd^beta_global_obs [global fit]
281 | pred_C = sol[1] + sol[0] * log_Yd
282 | R2_C = 1 - np.sum((log_eta - pred_C)**2) / np.sum((log_eta - np.mean(log_eta))**2)
283 |
284 | # Model D: eta = C (constant, no Yd dependence)
285 | R2_D = 0.0
286 |
287 | print(f" Model A (Yd^-0.41 fixed): R2 = {R2_A:.4f}")
288 | print(f" Model B (Yd^-(1-f_gas_i)): R2 = {R2_B:.4f}")
289 | print(f" Model C (Yd^beta_fit): R2 = {R2_C:.4f}, beta={sol[0]:.4f}")
290 | print(f" Model D (eta=const): R2 = {R2_D:.4f}")
291 |
292 | if R2_B > R2_A:
293 |     print(f"\n Per-galaxy f_gas (B) fixed exponent (A) !")
294 |     print(f" - f_gas_deep eta Yd")
295 | else:
296 |     print(f"\n Per-galaxy f_gas (B) fixed (A)")
297 |     print(f" - f_gas_deep")
298 |
299 | # =====
300 | # (4) f_gas_deep
301 | # =====
302 | print("\n" + "-" * 70)
303 | print("(4) f_gas_deep")
304 | print("-" * 70)
305 |
306 | params_test = {
307 |     'log(Yd)': log_Yd,
308 |     'log(vflat)': np.log10(vflat),
309 |     'log(hR)': np.log10(hR),
310 |     'log(vflat^2/hR)': np.log10(vflat**2 / hR),
311 | }
312 |
313 | for pname, pvals in params_test.items():
314 |     rho, p = spearmanr(f_gas, pvals)
315 |     print(f" rho(f_gas_deep, {pname:20s}) = {rho:+.3f} (p={p:.2e})")
316 |
317 | # =====
318 | # (5) Yd
319 | # =====
320 | print("\n" + "-" * 70)
321 | print("(5) Yd")
322 | print("-" * 70)
323 | print(f" Yd +/-10% ln(gc_deep)/d ln(Yd)")
324 |
325 | beta_numerical = []
326 | for r in results:
327 |     rc = load_rotcurve(r['name'])
328 |     if rc is None: continue
329 |     Yd_base = r['Yd']
330 |
331 | gc_lo = compute_gc_deep(rc, Yd_base * 0.9)

```

```

332 |         gc_hi = compute_gc_deep(rc, Yd_base * 1.1)
333 |
334 |         if gc_lo is None or gc_hi is None or gc_lo &lt;= 0 or gc_hi &lt;= 0:
335 |             continue
336 |
337 |         dlngc_dlnYd = (np.log(gc_hi) - np.log(gc_lo)) / (np.log(1.1) - np.log(0.9))
338 |         beta_numerical.append(dlngc_dlnYd)
339 |
340 |         beta_num = np.array(beta_numerical)
341 |         print(f"   d ln(gc)/d ln(Yd):")
342 |         print(f"   mean = {np.mean(beta_num):.4f}")
343 |         print(f"   median = {np.median(beta_num):.4f}")
344 |         print(f"   std = {np.std(beta_num):.4f}")
345 |         print(f"   -(1-f_gas_deep): median = {np.median(beta_pred):.4f}")
346 |         print(f"   eta exponent:           = -0.41")
347 |
348 |         # VS
349 |         if len(beta_num) == len(beta_pred):
350 |             rho_bn, p_bn = pearsonr(beta_num, beta_pred)
351 |             print(f"\n   r(numerical, analytical) = {rho_bn:.4f} (p={p_bn:.2e})")
352 |
353 |         # =====
354 |         # (6)
355 |         # =====
356 |         print("\n" + "-" * 70)
357 |         print("(6) ")
358 |         print("-" * 70)
359 |
360 |         # eta_theory = eta0 * Yd^(-(1-f_gas))
361 |         # where f_gas = 0.59 (if prediction holds)
362 |         # vs
363 |         # eta_theory = eta0 * Yd^(-(1-f_gas_i)) (per-galaxy)
364 |
365 |         # Residual after theory prediction
366 |         resid_B = log_eta - pred_B
367 |         resid_A = log_eta - pred_A
368 |
369 |         print(f"   Model A (Yd^-0.41):   resid std = {np.std(resid_A):.4f} dex")
370 |         print(f"   Model B (per-galaxy f_gas): resid std = {np.std(resid_B):.4f} dex")
371 |         print(f"   eta total scatter:         std = {np.std(log_eta):.4f} dex")
372 |         print(f"   Yd explained fraction: A={R2_A*100:.1f}%, B={R2_B*100:.1f}%")
373 |
374 |         # =====
375 |         # (7)
376 |         # =====
377 |         print("\n" + "-" * 70)
378 |         print(" ")
379 |         print("-" * 70)
380 |
381 |         print(f" ")
382 |         # eta ~ Yd^beta, beta = -(1 - f_gas_deep)
383 |         f_gas_deep = MOND(+bulge)
384 |
385 |         # f_gas_deep median = {np.median(f_gas):.4f} (0.59)
386 |         # beta_predicted = {np.median(beta_pred):.4f} (-0.41)
387 |         # beta = {np.median(beta_num):.4f}
388 |
389 |         # f_gas_deep ≈ 0.59 → eta ∝ Yd^(-0.41)
390 |
391 |         # (a) f_gas_deep median ∈ [0.50, 0.68] (0.59 ± 15%)
392 |         # (b) beta_numerical ≈ beta_analytical (r > 0.5)
393 |         # (c) Model B R2 >= Model A R2
394 |
395 |         # A-1 (eta) ∝ A B
396 |         # eta ∝ eta
397 |         # 1
398 |         """)
399 |
400 |         # =====
401 |         #
402 |         # =====
403 |         try:
404 |             import matplotlib
405 |             matplotlib.use('Agg')
406 |             import matplotlib.pyplot as plt
407 |
408 |             fig, axes = plt.subplots(2, 2, figsize=(12, 10))
409 |             fig.suptitle('eta first-principles derivation: gas fraction hypothesis', fontsize=14)
410 |
411 |             # (a) f_gas_deep distribution
412 |             ax = axes[0, 0]
413 |             ax.hist(f_gas, bins=30, alpha=0.7, color='steelblue', edgecolor='white')
414 |             ax.axvline(0.59, color='red', lw=2, ls='--', label='predicted (0.59)')
415 |             ax.axvline(np.median(f_gas), color='orange', lw=2, label=f'median ({np.median(f_gas):.3f})')
416 |             ax.set_xlabel('f_gas_deep')

```

```

424 |     ax.set_ylabel('count')
425 |     ax.set_title('Gas fraction at deep-MOND radii')
426 |     ax.legend()
427 |
428 |     # (b) eta_obs vs Yd with theory curves
429 |     ax = axes[0, 1]
430 |     ax.scatter(log_Yd, log_eta, s=10, alpha=0.5, c='steelblue')
431 |     Yd_range = np.linspace(log_Yd.min(), log_Yd.max(), 50)
432 |     ax.plot(Yd_range, np.log10(C_A) - 0.41*Yd_range, 'r-', lw=2, label='Yd^(-0.41) empirical')
433 |     ax.plot(Yd_range, np.log10(C_B) + np.median(beta_pred)*Yd_range, 'g--', lw=2,
434 |             label=f'Yd^{(np.median(beta_pred):.2f)} predicted')
435 |     ax.set_xlabel('log10(Yd)')
436 |     ax.set_ylabel('log10(eta)')
437 |     ax.set_title('eta vs Yd')
438 |     ax.legend()
439 |
440 |     # (c) beta_numerical vs beta_analytical
441 |     ax = axes[1, 0]
442 |     if len(beta_num) == len(beta_pred):
443 |         ax.scatter(beta_pred, beta_num, s=10, alpha=0.5, c='steelblue')
444 |         lim = [min(beta_pred.min(), beta_num.min())-0.1,
445 |               max(beta_pred.max(), beta_num.max())+0.1]
446 |         ax.plot(lim, lim, 'k--', lw=1)
447 |         ax.set_xlabel('beta_analytical = -(1-f_gas)')
448 |         ax.set_ylabel('beta_numerical (Yd +/-10%)')
449 |         ax.set_title(f'Analytical vs numerical beta (r={rho_bn:.3f})')
450 |
451 |     # (d) f_gas vs Yd
452 |     ax = axes[1, 1]
453 |     ax.scatter(log_Yd, f_gas, s=10, alpha=0.5, c='steelblue')
454 |     ax.axhline(0.59, color='red', ls='--', lw=1, label='prediction 0.59')
455 |     ax.set_xlabel('log10(Yd)')
456 |     ax.set_ylabel('f_gas_deep')
457 |     ax.set_title('Gas fraction vs Yd')
458 |     ax.legend()
459 |
460 |     plt.tight_layout()
461 |     figpath = os.path.join(BASE, 'eta_derivation_results.png')
462 |     plt.savefig(figpath, dpi=150)
463 |     print(f"\nFigure saved: {figpath}")
464 |     except Exception as e:
465 |         print(f"\nFigure error: {e}")
466 |
467 |     print("\n====")
468 |
469 | if __name__ == '__main__':
470 |     main()

```