Minimax Lower Bounds for Uniform Estimation of Covariate-Dependent Copula Parameters

Mathias Nthiani Muia[1*] Olivia Atutey[1†] Chathurika Srimali Abeykoon[2‡]

[1]Department of Mathematics and Statistics, University of South Alabama, Mobile, AL 36688

[2]Department of Mathematics, Rhodes College, Memphis, TN 38112

1 Code for the Simulation Study

This document provides the full Python code used to generate the simulation results reported in Section 5 of the manuscript. The code implements the kernel-weighted local likelihood estimator for covariate-dependent copula parameters, together with leave-one-out likelihood cross-validation for bandwidth selection and theoretical benchmark scaling motivated by the minimax lower-bound analysis.

The implementation follows the methodological specifications of the paper exactly: local linear fitting ($p = 1$), Epanechnikov kernel weighting, unconstrained optimization with inverse-link parameterization, and conditional simulation from the Gumbel copula using analytic inversion and bisection. All numerical results in the simulation study can be reproduced by running the script below.

Software requirements. The code was executed using Python 3.10 and relies on standard scientific libraries (`NumPy`, `SciPy`, `pandas`, `matplotlib`).

*Corresponding author: `mmuia@southalabama.edu`.

†`oatutey@southalabama.edu`.

‡`abeykoonc@rhodes.edu`.

1.1 Gumbel copula

This code implements the Gumbel-only simulation study used in the paper. It estimates the covariate-dependent copula parameter `theta(y)` using a kernel-weighted local likelihood fit on an evaluation grid {`y0`}. The local model uses the unscaled basis `t = Y - y0` with linear predictor `eta = gamma0 + gamma1*t + ... + gammap*tp`, and Epanechnikov weights of the form `w = K(t/h)/h`. The parameter is mapped to the admissible range through the inverse link `theta = 1 + 4/(1+exp(-eta))`, with additional clipping inside the copula density for numerical stability. Optimization at each grid point is carried out using BFGS (`gtol=1e-4`, `maxiter=80`), without warm-starting across grid points. Data generation uses conditional inversion for the Gumbel copula: `U1` is sampled uniformly on `(0,1)` and `U2` is obtained by solving `Cu(u,v;theta)=z` for `v` via bisection. Bandwidth selection is performed by leave-one-out likelihood cross-validation (LSCV), maximizing the sum of leave-one-out log copula densities over a candidate set constructed as multiples of `h_base = range(Y)*n^(-1/5)`. Plots use `N` on the horizontal axis (no log scaling), while error and rate comparisons are shown on a log scale on the vertical axis.

1.1.1 Code

```
# ============================================================
# GUMBEL-ONLY runner
#============================================================

!pip -q install scipy pandas

import numpy as np
import time
import pandas as pd
import matplotlib.pyplot as plt

from dataclasses import dataclass
from typing import Callable, Tuple, Dict, List, Optional
from scipy.optimize import minimize
from scipy.stats import norm
from numpy.random import default_rng
```

```
17
18 # ================================================================
19 # 0. RNGs
20 # ================================================================
21 rng_global = default_rng(2025)
22
23 # ================================================================
24 # 1. Truncated Normal TN(0,4; [-2,2]) - sampler and density
25 # ================================================================
26 def rtruncnorm(n, a=-2.0, b=2.0, mean=0.0, sd=2.0) -> np.ndarray:
27 alpha = (a - mean) / sd
28 beta  = (b - mean) / sd
29 Fa, Fb = norm.cdf(alpha), norm.cdf(beta)
30 u = rng_global.uniform(Fa, Fb, size=n)
31 return mean + sd * norm.ppf(u)
32
33 def dtruncnorm(x, a=-2.0, b=2.0, mean=0.0, sd=2.0):
34 x = np.asarray(x, float)
35 alpha = (a - mean) / sd
36 beta  = (b - mean) / sd
37 Z = norm.cdf(beta) - norm.cdf(alpha)
38 dens = norm.pdf((x - mean) / sd) / (sd * Z)
39 dens = np.where((x >= a) & (x <= b), dens, 0.0)
40 return float(dens) if dens.size == 1 else dens
41
42 # ================================================================
43 # 1B. Minimax benchmark rate r_N (Theorem 1) -- RATE ONLY
44 # ================================================================
45 def minimax_rate_rN(N: int, beta: int, s: int = 1) -> float:
46 N = int(N)
47 return float((np.log(N) / N) ** (beta / (2 * beta + s)))
48
49 # ================================================================
50 # 2. Utilities
51 # ================================================================
52 def clip01(u: np.ndarray, eps: float = 1e-12) -> np.ndarray:
```

```
return np.clip(u, eps, 1 - eps)

def K_epan(u: np.ndarray) -> np.ndarray:
u = np.asarray(u, float)
out = np.zeros_like(u)
m = np.abs(u) <= 1.0
out[m] = 0.75 * (1.0 - u[m]**2)
return out

# ================================================================
# 3. Inverse link psi^{-1}(nu) : R -> (1,5)
#    psi^{-1}(nu) = 1 + 4/(1+exp(-nu))
# ================================================================
def inv_link(nu: np.ndarray) -> np.ndarray:
nu = np.asarray(nu, float)
nu = np.clip(nu, -30.0, 30.0)
return 1.0 + 4.0 / (1.0 + np.exp(-nu))

# ============================================================
# 4. TRUE parameter function theta(y) on ORIGINAL Y-SCALE [-2,2]
# ============================================================
def theta_true_sine_gumbel(y):
y = np.asarray(y, float)
t = (y + 2.0) / 4.0
return 2.0 + 0.8*np.sin(2*np.pi*t)   # range [1.5, 4.8]

# ============================================================
# 5. Gumbel-Hougaard copula density
#    with theta clipped away from boundaries
# ============================================================
def gumbel_pdf(u1, u2, th):
u1 = np.asarray(u1, float)
u2 = np.asarray(u2, float)
th = np.asarray(th, float)

eps_u = 1e-12
```

```
u1 = np.clip(u1, eps_u, 1.0 - eps_u)
u2 = np.clip(u2, eps_u, 1.0 - eps_u)

th = np.clip(th, 1.0 + 5e-3, 5.0 - 5e-3)

x = -np.log(u1)
y = -np.log(u2)
A = x**th + y**th
B = A**(1.0/th)

dens = (x**(th - 1.0) * y**(th - 1.0) *
A**((1.0 - 2.0*th)/th) *
(th + B - 1.0) *
np.exp(-B) / (u1 * u2))

return np.maximum(dens, 1e-12)

def gumbel_log_density(u: np.ndarray, v: np.ndarray, theta: np.ndarray) -> np.ndarray:
dens = gumbel_pdf(u, v, theta)
return np.log(np.maximum(dens, 1e-300))

# ============================================================
# 6. Conditional sampling from Gumbel
#    U1 ~ U(theta,1), then solve Cu(u,v;theta)=z by bisection
# ============================================================
def Cu_gumbel(u, v, theta):
u = float(np.clip(u, 1e-12, 1.0 - 1e-12))
v = float(np.clip(v, 1e-12, 1.0 - 1e-12))
theta = float(theta)

x = -np.log(u)
y = -np.log(v)
A = x**theta + y**theta
B = A**(1.0/theta)

return np.exp(-B) * (B**(1.0 - theta)) * (x**(theta - 1.0)) / u
```

```
125
126 def rGumbel(n: int, theta: float, rng: np.random.Generator) -> Tuple[np.ndarray, np.ndarray]:
127 theta = float(theta)
128 eps = 1e-9
129 U1 = rng.uniform(eps, 1.0 - eps, size=n)
130 U2 = np.empty(n)
131
132 for i in range(n):
133 u = float(U1[i])
134 z = float(rng.uniform())
135
136 lo, hi = eps, 1.0 - eps
137 for _ in range(60):
138 mid = 0.5 * (lo + hi)
139 val = Cu_gumbel(u, mid, theta)
140 if val < z:
141 lo = mid
142 else:
143 hi = mid
144 U2[i] = 0.5 * (lo + hi)
145
146 return U1, U2
147
148 # ============================================================
149 # 7. Copula spec (Gumbel only) + invlink
150 # ============================================================
151 @dataclass
152 class CopulaSpec:
153 name: str
154 log_density: Callable[[np.ndarray, np.ndarray, np.ndarray], np.ndarray]
155 sampler: Callable[[int, float, np.random.Generator], Tuple[np.ndarray, np.ndarray]]
156 invlink: Callable[[np.ndarray], np.ndarray]
157
158 gumbel = CopulaSpec(
159 name="Gumbel",
160 log_density=gumbel_log_density,
```

```
sampler=lambda n, th, rng: rGumbel(n, th, rng),
invlink=inv_link
)

# ============================================================
# 8. Score derivative and \sigma^2(\theta)
# ============================================================
def logdens_and_dlogdens_dtheta_gumbel(u1: float, u2: float, theta: float) -> Tuple[float, float]:
eps_theta = 5e-3

th = float(theta)
th_p = float(np.clip(th + eps_theta, 1.0 + 5e-3, 5.0 - 5e-3))
th_m = float(np.clip(th - eps_theta, 1.0 + 5e-3, 5.0 - 5e-3))
th   = float(np.clip(th,           1.0 + 5e-3, 5.0 - 5e-3))

d0 = float(gumbel_pdf(u1, u2, th))
dp = float(gumbel_pdf(u1, u2, th_p))
dm = float(gumbel_pdf(u1, u2, th_m))

d0 = max(d0, 1e-12); dp = max(dp, 1e-12); dm = max(dm, 1e-12)
logc = np.log(d0)
dldth = (np.log(dp) - np.log(dm)) / (th_p - th_m)
return logc, dldth

def sigma2_theta_gumbel(theta: float, M: int = 1000, seed: int = 24680) -> float:
rng_sig = np.random.default_rng(seed)
theta = float(np.clip(theta, 1.0 + 5e-3, 5.0 - 5e-3))
U1, U2 = rGumbel(M, theta, rng_sig)
dld2 = np.zeros(M)
for k in range(M):
_, dld = logdens_and_dlogdens_dtheta_gumbel(float(U1[k]), float(U2[k]), theta)
dld2[k] = dld**2
return float(np.mean(dld2))

def asymptotic_variance_theta_gumbel(y0: float, theta_hat_y0: float, h: float, n: int) -> float:
C = 3.0 / 5.0
```

```
fy = float(dtruncnorm(y0))
if (fy <= 0) or (not np.isfinite(fy)):
return np.nan

theta_hat_y0 = float(np.clip(theta_hat_y0, 1.0 + 5e-3, 5.0 - 5e-3))
sig2 = sigma2_theta_gumbel(theta_hat_y0)

denom = n * float(h) * sig2 * fy
if (denom <= 0) or (not np.isfinite(denom)):
return np.nan
return float(C / denom)

# ============================================================
# 9. Local likelihood
# ============================================================
def local_loglik_mathias(gamma: np.ndarray,
y0: float,
Y: np.ndarray, U1: np.ndarray, U2: np.ndarray,
h: float,
p: int,
copula: CopulaSpec) -> float:
t = Y - float(y0)
w = K_epan(t / float(h)) / float(h)
idx = w > 0
if not np.any(idx):
return 0.0

t  = t[idx]
w  = w[idx]
u1 = U1[idx]
u2 = U2[idx]

X = np.vstack([t**k for k in range(p + 1)]).T
eta = X @ gamma
theta = copula.invlink(eta)

```

```
dens = gumbel_pdf(u1, u2, theta)
logc = np.log(np.maximum(dens, 1e-300))
return -float(np.sum(w * logc))

def estimate_theta_at_point_mathias(y0: float,
Y: np.ndarray, U1: np.ndarray, U2: np.ndarray,
h: float,
p: int,
copula: CopulaSpec,
init_gamma: Optional[np.ndarray] = None) -> float:
if init_gamma is None:
init_gamma = np.zeros(p + 1, dtype=float)

res = minimize(
local_loglik_mathias,
x0=init_gamma,
args=(y0, Y, U1, U2, h, p, copula),
method="BFGS",
options={"maxiter": 80, "gtol": 1e-4}
)
gamma_hat = res.x
return float(copula.invlink(np.array([gamma_hat[0]]))[0])

def estimate_theta_grid_mathias(Y: np.ndarray, U1: np.ndarray, U2: np.ndarray,
grid: np.ndarray,
h: float,
p: int,
copula: CopulaSpec) -> np.ndarray:
theta_hat = np.zeros_like(grid, dtype=float)
init_gamma = np.zeros(p + 1, dtype=float)
for j, y0 in enumerate(grid):
theta_hat[j] = estimate_theta_at_point_mathias(
y0=float(y0), Y=Y, U1=U1, U2=U2,
h=float(h), p=int(p), copula=copula,
init_gamma=init_gamma
)
```

```
return theta_hat

# ============================================================
# 10. TRUE LSCV bandwidth: maximize LOO loglik
# ============================================================
def select_bandwidth_lscv_mathias(Y: np.ndarray, U1: np.ndarray, U2: np.ndarray,
p: int,
copula: CopulaSpec,
n_h: int = 9,
h_mult_range: Tuple[float, float] = (0.15, 1.0)) -> Tuple[float, Dict[float, float]]:
n = len(Y)
rngY = float(np.max(Y) - np.min(Y))
h_base = rngY * (n ** (-1.0 / 5.0))
hs = np.linspace(h_mult_range[0], h_mult_range[1], int(n_h)) * h_base

scores: Dict[float, float] = {}
for h in hs:
s_val = 0.0
ok = True
for i in range(n):
idx_tr = np.concatenate((np.arange(0, i), np.arange(i + 1, n)))
th_i = estimate_theta_at_point_mathias(
y0=float(Y[i]),
Y=Y[idx_tr], U1=U1[idx_tr], U2=U2[idx_tr],
h=float(h), p=int(p), copula=copula,
init_gamma=np.zeros(p + 1, dtype=float)
)
dens_i = float(gumbel_pdf(float(U1[i]), float(U2[i]), float(th_i)))
if not np.isfinite(dens_i) or dens_i <= 0:
ok = False
break
s_val += float(np.log(max(dens_i, 1e-12)))

scores[float(h)] = (-np.inf if not ok else float(s_val))

best_h = max(scores, key=scores.get)
```

```
return float(best_h), scores

# ============================================================
# 11. Bandwidth h_local
# ============================================================
def h_local(N: int, beta: int, s: int = 1, c: float = 1.0) -> float:
return float(c * ((np.log(N) / N) ** (1.0 / (2 * beta + s))))

# ============================================================
# 12. One replication
# ============================================================
def run_one_replication(N: int,
copula: CopulaSpec,
p: int,
grid: np.ndarray,
rng: np.random.Generator,
c_theory: float,
cv_n_h: int = 9,
cv_mult_range: Tuple[float, float] = (0.15, 1.0)) -> Dict:

Y = rtruncnorm(N, a=-2.0, b=2.0, mean=0.0, sd=2.0)
theta_Y = theta_true_sine_gumbel(Y)

U1 = np.zeros(N, float)
U2 = np.zeros(N, float)
for i in range(N):
u1i, u2i = copula.sampler(1, float(theta_Y[i]), rng)
U1[i] = float(u1i[0]); U2[i] = float(u2i[0])

beta = p + 1
h_th = h_local(N, beta=beta, s=1, c=c_theory)

h_cv, cv_scores = select_bandwidth_lscv_mathias(
Y, U1, U2, p=p, copula=copula, n_h=cv_n_h, h_mult_range=cv_mult_range
)

```

```
theta_true_grid = theta_true_sine_gumbel(grid)

theta_hat_th = estimate_theta_grid_mathias(Y, U1, U2, grid, h=h_th, p=p, copula=copula)
theta_hat_cv = estimate_theta_grid_mathias(Y, U1, U2, grid, h=h_cv, p=p, copula=copula)

asyvar_cv = np.zeros_like(grid, float)
for j, y0 in enumerate(grid):
asyvar_cv[j] = asymptotic_variance_theta_gumbel(float(y0), float(theta_hat_cv[j]), float(h_cv),
    int(N))

sup_th = float(np.max(np.abs(theta_hat_th - theta_true_grid)))
l2_th  = float(np.sqrt(np.mean((theta_hat_th - theta_true_grid) ** 2)))

sup_cv = float(np.max(np.abs(theta_hat_cv - theta_true_grid)))
l2_cv  = float(np.sqrt(np.mean((theta_hat_cv - theta_true_grid) ** 2)))

return {
  "h_local": h_th,
  "h_cv": h_cv,
  "sup_theory": sup_th,
  "l2_theory": l2_th,
  "sup_cv": sup_cv,
  "l2_cv": l2_cv,
  "theta_hat_th": theta_hat_th,
  "theta_hat_cv": theta_hat_cv,
  "theta_true_grid": theta_true_grid,
  "asyvar_cv": asyvar_cv,
  "cv_scores": cv_scores,
  "n_h_candidates": int(len(cv_scores)),
}

# ============================================================
# 13. Experiment runner
# ============================================================
def summarize(rows: List[Dict[str, float]]) -> Dict[str, float]:
out = {}
```

```
for key in ["h_local", "h_cv", "sup_theory", "sup_cv", "l2_theory", "l2_cv"]:
vals = np.array([r[key] for r in rows], dtype=float)
out[key + "_mean"] = float(vals.mean())
out[key + "_sd"]   = float(vals.std(ddof=1)) if len(vals) > 1 else 0.0
return out

def run_experiment(Ns: List[int],
R: int,
copula: CopulaSpec,
p: int,
grid_m: int,
c_theory: float,
seed: int = 777,
delta: float = 0.20,
cv_n_h: int = 9,
cv_mult_range: Tuple[float, float] = (0.15, 1.0)) -> Tuple[pd.DataFrame, Dict[int, Dict[str,
    np.ndarray]]]:

rng = np.random.default_rng(seed)
grid = np.linspace(-2.0 + delta, 2.0 - delta, grid_m)

recs = []
curves: Dict[int, Dict[str, np.ndarray]] = {}

for N in Ns:
t0 = time.time()
rows = []

th_curves_th = []
th_curves_cv = []
asy_curves_cv = []
true_grid = None
nh_list = []

for _ in range(R):
out = run_one_replication(
```

```
N=N, copula=copula, p=p, grid=grid, rng=rng, c_theory=c_theory,
cv_n_h=cv_n_h, cv_mult_range=cv_mult_range
)
rows.append(out)
th_curves_th.append(out["theta_hat_th"])
th_curves_cv.append(out["theta_hat_cv"])
asy_curves_cv.append(out["asyvar_cv"])
true_grid = out["theta_true_grid"]
nh_list.append(out["n_h_candidates"])

th_curves_th  = np.vstack(th_curves_th)
th_curves_cv  = np.vstack(th_curves_cv)
asy_curves_cv = np.vstack(asy_curves_cv)

curves[N] = {
  "grid": grid,
  "theta_true": true_grid,
  "theta_hat_th_mean": th_curves_th.mean(axis=0),
  "theta_hat_cv_mean": th_curves_cv.mean(axis=0),
  "asyvar_cv_mean": np.nanmean(asy_curves_cv, axis=0),
}

s = summarize(rows)
s["N"] = int(N)
s["p"] = int(p)
s["beta"] = int(p + 1)
s["copula"] = copula.name
s["time_sec"] = float(time.time() - t0)
s["n_h_candidates"] = float(np.mean(nh_list))

rN = minimax_rate_rN(N=int(N), beta=int(p + 1), s=1)
s["minimax_rN"] = float(rN)
s["sup_theory_over_rN"] = float(s["sup_theory_mean"] / rN)
s["sup_cv_over_rN"]     = float(s["sup_cv_mean"] / rN)
s["l2_theory_over_rN"]  = float(s["l2_theory_mean"] / rN)
s["l2_cv_over_rN"]      = float(s["l2_cv_mean"] / rN)
```

```
447
448 recs.append(s)
449 print(f"[Gumbel Mathias-style p={p}] N={N} done in {s['time_sec']:.1f}s")
450
451 df = pd.DataFrame(recs).sort_values("N").reset_index(drop=True)
452 return df, curves
453
454 # ============================================================
455 # 14. Plot helpers
456 # ============================================================
457 def plot_bandwidths(df: pd.DataFrame, title: str):
458 Ns = df["N"].to_numpy(float)
459 plt.figure(figsize=(7,4))
460 plt.plot(Ns, df["h_local_mean"].to_numpy(float), marker="o", label="h_local")
461 plt.plot(Ns, df["h_cv_mean"].to_numpy(float), marker="s", label="h_cv")
462 plt.xticks(Ns.astype(int))
463 plt.xlabel("N")
464 plt.ylabel("bandwidth")
465 plt.title(title)
466 plt.grid(True, alpha=0.3)
467 plt.legend()
468 plt.tight_layout()
469 plt.show()
470
471 def plot_errors(df: pd.DataFrame, title: str, which="sup"):
472 Ns = df["N"].to_numpy(float)
473 plt.figure(figsize=(7,4))
474 plt.plot(Ns, df[f"{which}_theory_mean"].to_numpy(float), marker="o", label=f"{which} error
    (h_local)")
475 plt.plot(Ns, df[f"{which}_cv_mean"].to_numpy(float), marker="s", label=f"{which} error (h_cv)")
476 plt.yscale("log")
477 plt.xticks(Ns.astype(int))
478 plt.xlabel("N")
479 plt.ylabel(f"{which} error (log scale)")
480 plt.title(title)
481 plt.grid(True, alpha=0.3)
```

```
plt.legend()
plt.tight_layout()
plt.show()

def plot_mean_theta_with_cv_CI(curves: Dict[int, Dict[str, np.ndarray]], N: int, title: str):
d = curves[N]
g = d["grid"]
true = d["theta_true"]
th_mean = d["theta_hat_th_mean"]
cv_mean = d["theta_hat_cv_mean"]

se = np.sqrt(np.maximum(d["asyvar_cv_mean"], 0.0))
lo = cv_mean - 1.96 * se
hi = cv_mean + 1.96 * se

plt.figure(figsize=(7,4))
plt.plot(g, true, label="true theta(y)")
plt.plot(g, th_mean, label="mean theta_hat(y) (h_local)")
plt.plot(g, cv_mean, label="mean theta_hat(y) (h_cv)")
plt.fill_between(g, lo, hi, alpha=.25, label="95% CI (h_cv only)")
plt.xlabel("y")  # <-- add this
plt.title(title)
plt.grid(True, alpha=0.3)
plt.legend()
plt.tight_layout()
plt.show()

def plot_sup_error_vs_minimax(df: pd.DataFrame, title: str, which: str = "cv"):
sub = df.sort_values("N")
Ns  = sub["N"].to_numpy(float)
sup = sub[f"sup_{which}_mean"].to_numpy(float)
rN  = sub["minimax_rN"].to_numpy(float)

plt.figure(figsize=(7,4))
plt.plot(Ns, sup, marker="o", label=f"Mean sup error ({which})")
plt.plot(Ns, rN,  marker="s", label=r"Benchmark $r_N=(\log N/N)^{\beta/(2\beta+s)}$")
```

```
plt.yscale("log")
plt.xticks(Ns.astype(int))
plt.xlabel("N")
plt.ylabel("log scale")
plt.title(title)
plt.grid(True, alpha=0.3)
plt.legend()
plt.tight_layout()
plt.show()

def plot_scaled_sup_error(df: pd.DataFrame, title: str, which: str = "cv"):
sub = df.sort_values("N")
Ns = sub["N"].to_numpy(float)
scaled = sub[f"sup_{which}_over_rN"].to_numpy(float)

plt.figure(figsize=(7,4))
plt.plot(Ns, scaled, marker="o", label=rf"Mean scaled sup error (sup_{which}/r_N)")
plt.xticks(Ns.astype(int))
plt.xlabel("N")
plt.ylabel(r"sup error / $r_N$")
plt.title(title)
plt.grid(True, alpha=0.3)
plt.legend()
plt.tight_layout()
plt.show()

# ============================================================
# 15. SETTINGS + RUN
# ============================================================
Ns = [50, 100, 250, 500, 750]
R = 100
grid_m = 60
seed = 777
delta = 0.20

P = 1
```

```
554
555 def c_theory_for_p(p: int) -> float:
556 return 1.0
557
558 df, curves = run_experiment(
559 Ns=Ns, R=R, copula=gumbel, p=P, grid_m=grid_m,
560 c_theory=c_theory_for_p(P),
561 seed=seed,
562 delta=delta,
563 cv_n_h=9,
564 cv_mult_range=(0.15, 1.0)
565 )
566
567 # ============================================================
568 # TABLE + SAVE
569 # ============================================================
570 pd.set_option("display.precision", 4)
571
572 cols = [
573 "copula","p","beta","N",
574 "h_local_mean","h_local_sd",
575 "h_cv_mean","h_cv_sd",
576 "minimax_rN",
577 "sup_theory_mean","sup_theory_sd",
578 "sup_cv_mean","sup_cv_sd",
579 "sup_theory_over_rN",
580 "sup_cv_over_rN",
581 "l2_theory_mean","l2_theory_sd",
582 "l2_cv_mean","l2_cv_sd",
583 "l2_theory_over_rN",
584 "l2_cv_over_rN",
585 "n_h_candidates","time_sec"
586 ]
587
588 print(f"\n=== Gumbel-only Summary, p={P} ===")
589
```

```
# ---- Colab-safe full table print (no truncation, no hidden columns)
with pd.option_context(
"display.max_columns", None,
"display.width", 3000,
"display.max_colwidth", None,
"display.expand_frame_repr", False,
"display.precision", 4
):
print(df[cols].sort_values(["p","N"]).to_string(index=False))

out_csv = f"gumbel_only_theta_estimator_p{P}.csv"
df.to_csv(out_csv, index=False)
print(f"\nSaved: {out_csv}")

# ============================================================
# PLOTS
# ============================================================
sub = df.sort_values("N")
plot_bandwidths(sub, f"Gumbel bandwidth comparison, p={P}")
plot_errors(sub,     f"Gumbel sup-norm errors, p={P}", which="sup")
plot_errors(sub,     f"Gumbel L2 errors, p={P}", which="l2")
plot_sup_error_vs_minimax(sub, f"Gumbel: sup error vs minimax benchmark, p={P} (h_cv)", which="cv")
plot_scaled_sup_error(sub,     f"Gumbel: scaled sup error sup_cv/r_N vs N, p={P}", which="cv")

Nmax = int(sub["N"].max())
plot_mean_theta_with_cv_CI(
curves,
Nmax,
f"Gumbel mean theta_hat(y) over R reps, p={P}, N={Nmax} (CI for h_cv only)"
)

```

1.2 Clayton copula

This code implements the Clayton-only simulation study using the same local likelihood framework as in the Gumbel case, with adjustments specific to the Clayton copula family. The covariate-

dependent parameter `theta(y)` is estimated by a kernel-weighted local polynomial likelihood fit on an interior evaluation grid {`y0`}. The local design matrix is built from the unscaled basis `t = y - y0` up to order `p`, and Epanechnikov weights of the form `w = K(t/h)/h` are used throughout. The Clayton copula parameter is mapped to a strictly positive and bounded parameter space via a bounded sigmoid inverse link, ensuring numerical stability in both estimation and likelihood evaluation.

Conditional on each covariate value, pseudo-observations are generated from the Clayton copula using a gamma–exponential mixture representation. Bandwidth selection is carried out using leave-one-out likelihood cross-validation (LOO-CVL), which maximizes the sum of leave-one-out log copula densities evaluated at the locally estimated parameter. Candidate bandwidths are constructed as fixed multiples of the theoretical benchmark `h_local`, and the same candidate set is used across replications for a given sample size and polynomial order. Local likelihood optimization is performed using the L-BFGS-B algorithm, with iteration limits adjusted according to the polynomial order.

For each replication and sample size, the estimator is evaluated on a fixed interior grid, and performance is summarized using sup-norm and discrete L2 losses. Asymptotic variance estimates are computed pointwise and used to construct confidence bands for the cross-validated estimator. In all plots, the sample size `N` is shown on the horizontal axis without logarithmic scaling, while error and rate comparisons are displayed on a logarithmic vertical scale to facilitate comparison with the minimax benchmark rate.

1.2.1 Code

```
# ============================================================
# Clayton-code
# ============================================================

!pip -q install scipy pandas

import numpy as np
import time
import pandas as pd
```

```
import matplotlib.pyplot as plt

from dataclasses import dataclass
from typing import Callable, Tuple, Dict, List, Optional
from scipy.optimize import minimize
from scipy.stats import norm
from numpy.random import default_rng

# ================================================================
# 0. Global RNG for truncated normal
# ================================================================
rng_global = default_rng(2025)

# ================================================================
# 1. Truncated Normal TN(0,4; [-2,2]) - sampler and density
# ================================================================
def rtruncnorm(n, a=-2, b=2, mean=0.0, sd=2.0) -> np.ndarray:
alpha = (a - mean) / sd
beta  = (b - mean) / sd
Fa, Fb = norm.cdf(alpha), norm.cdf(beta)
u = rng_global.uniform(Fa, Fb, size=n)
return mean + sd * norm.ppf(u)

def dtruncnorm(x, a=-2, b=2, mean=0.0, sd=2.0):
x = np.asarray(x, float)
alpha = (a - mean) / sd
beta  = (b - mean) / sd
Z = norm.cdf(beta) - norm.cdf(alpha)
dens = norm.pdf((x - mean) / sd) / (sd * Z)
dens = np.where((x >= a) & (x <= b), dens, 0.0)
return float(dens) if dens.size == 1 else dens

# ================================================================
# 1B. Minimax benchmark rate r_N (Theorem 1) -- RATE ONLY
# ================================================================
def minimax_rate_rN(N: int, beta: int, s: int = 1) -> float:
```

```
N = int(N)
return float((np.log(N) / N) ** (beta / (2 * beta + s)))

# ----------------------------
# Utilities
# ----------------------------
def clip01(u: np.ndarray, eps: float = 1e-12) -> np.ndarray:
return np.clip(u, eps, 1 - eps)

def epanechnikov(t: np.ndarray) -> np.ndarray:
t = np.asarray(t, dtype=float)
out = np.zeros_like(t, dtype=float)
m = np.abs(t) <= 1.0
out[m] = 0.75 * (1.0 - t[m]**2)
return out

def design_matrix_local_poly(y: np.ndarray, y0: float, p: int) -> np.ndarray:
y = np.asarray(y, dtype=float)
z = (y - float(y0))
return np.vstack([z**k for k in range(p+1)]).T

def sigmoid(z: np.ndarray) -> np.ndarray:
z = np.asarray(z, dtype=float)
z = np.clip(z, -50, 50)
return 1.0 / (1.0 + np.exp(-z))

def invlink_bounded_sigmoid(nu: np.ndarray, theta_min: float, theta_max: float) -> np.ndarray:
nu = np.asarray(nu, dtype=float)
return theta_min + (theta_max - theta_min) * sigmoid(nu)

# ----------------------------
# Clayton log-density
# ----------------------------
def clayton_log_density(u: np.ndarray, v: np.ndarray, theta: np.ndarray) -> np.ndarray:
"""
Clayton copula log-density (theta>0):
```

```
c(u,v)=(1+theta(uv)^(-1-theta)(u^{-theta}+v^{-theta}-1)^(-2-1/theta)
"""
u = clip01(np.asarray(u))
v = clip01(np.asarray(v))
th = np.maximum(np.asarray(theta, dtype=float), 1e-8)

logu = np.log(u)
logv = np.log(v)

a = -th * logu
b = -th * logv
m = np.maximum(a, b)
log_sum = m + np.log(np.exp(a - m) + np.exp(b - m))

term = np.maximum(np.exp(log_sum) - 1.0, 1e-300)
log_term = np.log(term)

return np.log1p(th) + (-1.0 - th) * (logu + logv) + (-2.0 - 1.0/th) * log_term


# ----------------------------
# Clayton sampler
# ----------------------------
def sample_clayton(n: int, theta: float, rng: np.random.Generator) -> Tuple[np.ndarray,
    np.ndarray]:
th = max(float(theta), 1e-8)
W = rng.gamma(shape=1.0/th, scale=1.0, size=n)
E1 = rng.exponential(scale=1.0, size=n)
E2 = rng.exponential(scale=1.0, size=n)
U = (1.0 + E1 / W) ** (-1.0 / th)
V = (1.0 + E2 / W) ** (-1.0 / th)
return U, V


# ----------------------------
# Copula spec (Clayton only)
# ----------------------------
@dataclass
```

```
class CopulaSpec:
name: str
log_density: Callable[[np.ndarray, np.ndarray, np.ndarray], np.ndarray]
sampler: Callable[[int, float, np.random.Generator], Tuple[np.ndarray, np.ndarray]]
invlink: Callable[[np.ndarray], np.ndarray]

def clayton_invlink(nu: np.ndarray) -> np.ndarray:
return invlink_bounded_sigmoid(nu, theta_min=0.05, theta_max=8.0)

clayton = CopulaSpec(
name="Clayton",
log_density=clayton_log_density,
sampler=sample_clayton,
invlink=clayton_invlink
)

# ----------------------------
# True parameter function theta(y) on ORIGINAL Y-SCALE [-2,2]
# ----------------------------
def theta_true_sine_clayton(y: np.ndarray) -> np.ndarray:
y = np.asarray(y, dtype=float)
t = (y + 2.0) / 4.0
return 1.5 + 0.9*np.sin(2*np.pi*t)

# ================================================================
# Score derivative and sigma^2(theta) for Clayton
# ================================================================
def logdens_and_dlogdens_dtheta_clayton(u1: float, u2: float, theta: float) -> Tuple[float,
    float]:
eps_theta = 1e-3
th = float(max(theta, 1e-8))

th_p = max(th + eps_theta, 1e-8)
th_m = max(th - eps_theta, 1e-8)

log0 = float(clayton_log_density(np.array([u1]), np.array([u2]), np.array([th]))[0])
```

```
    logp = float(clayton_log_density(np.array([u1]), np.array([u2]), np.array([th_p]))[0])
    logm = float(clayton_log_density(np.array([u1]), np.array([u2]), np.array([th_m]))[0])

    dldth = (logp - logm) / (th_p - th_m)
    return log0, dldth

def sigma2_theta_clayton(theta: float, M: int = 1500, seed: int = 24680) -> float:
    """
    NOTE: Lowered default M from 2000 -> 1500 to reduce cost.
    """
    rng_sig = np.random.default_rng(seed)
    U, V = sample_clayton(M, float(theta), rng_sig)
    out = np.zeros(M, dtype=float)
    for k in range(M):
        _, dld = logdens_and_dlogdens_dtheta_clayton(float(U[k]), float(V[k]), float(theta))
        out[k] = dld**2
    return float(np.mean(out))

def asymptotic_variance_theta_clayton(y0: float, theta_hat_y0: float, h: float, n: int) -> float:
    C = 3/5
    fy = float(dtruncnorm(y0))
    if fy <= 0 or (not np.isfinite(fy)):
        return np.nan

    sig2 = sigma2_theta_clayton(float(theta_hat_y0))
    denom = n * float(h) * sig2 * fy
    if denom <= 0 or (not np.isfinite(denom)):
        return np.nan
    return float(C / denom)

# ----------------------------
# Local likelihood estimation (Clayton: L-BFGS-B)
# ----------------------------
def local_loglik_generic(gamma: np.ndarray,
                         y0: float,
                         y: np.ndarray, u: np.ndarray, v: np.ndarray,
```

```
    h: float, p: int,
    copula: CopulaSpec,
    train_mask: Optional[np.ndarray] = None) -> float:
    if train_mask is None:
        yy, uu, vv = y, u, v
    else:
        yy, uu, vv = y[train_mask], u[train_mask], v[train_mask]

    X = design_matrix_local_poly(yy, y0, p)
    nu = X @ gamma
    theta_vec = copula.invlink(nu)

    w = epanechnikov((yy - y0)/h) / h
    idx = w > 0
    if not np.any(idx):
        return 0.0

    ll = copula.log_density(uu[idx], vv[idx], theta_vec[idx])
    return float(np.sum(w[idx] * ll))

def fit_theta_at_y0(y0: float,
    y: np.ndarray, u: np.ndarray, v: np.ndarray,
    h: float, p: int,
    copula: CopulaSpec,
    init_gamma: Optional[np.ndarray] = None,
    train_mask: Optional[np.ndarray] = None,
    maxiter: int = 80) -> Tuple[float, float]:
    if init_gamma is None:
        init_gamma = np.zeros(p+1, dtype=float)

    def obj(g):
        return -local_loglik_generic(g, y0, y, u, v, h, p, copula, train_mask=train_mask)

    res = minimize(obj, x0=init_gamma, method="L-BFGS-B", options={"maxiter": maxiter, "ftol": 1e-8})

    nu_hat = float(res.x[0])
```

```
    theta_hat = float(copula.invlink(np.array([nu_hat]))[0])
    return nu_hat, theta_hat

def fit_theta_on_grid(y: np.ndarray, u: np.ndarray, v: np.ndarray,
grid: np.ndarray, h: float, p: int,
copula: CopulaSpec,
warm_start: bool = True) -> Tuple[np.ndarray, np.ndarray]:
    nu_hat = np.zeros_like(grid, dtype=float)
    th_hat = np.zeros_like(grid, dtype=float)

    gamma0 = np.zeros(p+1, dtype=float)
    for j, y0 in enumerate(grid):
        nu_j, th_j = fit_theta_at_y0(float(y0), y, u, v, h, p, copula,
        init_gamma=gamma0 if warm_start else None,
        maxiter=100 if p >= 3 else 80)
        nu_hat[j] = nu_j
        th_hat[j] = th_j

        if warm_start:
            gamma0 = np.zeros(p+1, dtype=float)
            gamma0[0] = nu_j

    return nu_hat, th_hat

# ----------------------------
# Bandwidths
# ----------------------------
def h_local(N: int, beta: int, s: int = 1, c: float = 1.0) -> float:
    return float(c * ((np.log(N)/N) ** (1.0/(2*beta + s))))

def B_LOO(h: float,
y: np.ndarray, u: np.ndarray, v: np.ndarray,
p: int, copula: CopulaSpec) -> float:
    """
    LOO-CVL objective. This is the heavy part.
    If you still find it slow, reduce:
```

```
- R
- Ns
- len(h_candidates)
- maxiter in fit_theta_at_y0
"""
N = len(y)
total = 0.0
init_gamma = np.zeros(p+1, dtype=float)

for i in range(N):
train_mask = np.ones(N, dtype=bool)
train_mask[i] = False

_, theta_i_hat = fit_theta_at_y0(float(y[i]), y, u, v, h, p, copula,
init_gamma=init_gamma,
train_mask=train_mask,
maxiter=100 if p >= 3 else 80)

total += float(copula.log_density(np.array([u[i]]),
np.array([v[i]]),
np.array([theta_i_hat]))[0])
return float(total)

def select_h_cv_LOO(h_candidates: List[float],
y: np.ndarray, u: np.ndarray, v: np.ndarray,
p: int, copula: CopulaSpec) -> Tuple[float, Dict[float, float]]:
scores = {}
for hh in h_candidates:
scores[float(hh)] = B_LOO(float(hh), y, u, v, p, copula)
best_h = max(scores, key=scores.get)
return float(best_h), scores

# ----------------------------
# One replication
# ----------------------------
def run_one_replication(N: int,
```

```
copula: CopulaSpec,
p: int,
grid: np.ndarray,
rng: np.random.Generator,
c_theory: float,
h_candidates: List[float]) -> Dict:

y = rtruncnorm(N, a=-2, b=2, mean=0.0, sd=2.0)
theta_vals = theta_true_sine_clayton(y)

u = np.zeros(N, dtype=float)
v = np.zeros(N, dtype=float)
for i in range(N):
ui, vi = copula.sampler(1, float(theta_vals[i]), rng)
u[i] = ui[0]; v[i] = vi[0]

beta = p + 1
h_th = h_local(N, beta=beta, s=1, c=c_theory)

h_cv, _ = select_h_cv_LOO(h_candidates, y, u, v, p, copula)

theta_true_grid = theta_true_sine_clayton(grid)

_, theta_hat_th = fit_theta_on_grid(y, u, v, grid, h_th, p, copula, warm_start=True)
_, theta_hat_cv = fit_theta_on_grid(y, u, v, grid, h_cv, p, copula, warm_start=True)

asyvar_cv = np.zeros_like(grid, dtype=float)
for j, y0 in enumerate(grid):
asyvar_cv[j] = asymptotic_variance_theta_clayton(float(y0), float(theta_hat_cv[j]), float(h_cv),
   int(N))

sup_th = float(np.max(np.abs(theta_hat_th - theta_true_grid)))
l2_th  = float(np.sqrt(np.mean((theta_hat_th - theta_true_grid)**2)))

sup_cv = float(np.max(np.abs(theta_hat_cv - theta_true_grid)))
l2_cv  = float(np.sqrt(np.mean((theta_hat_cv - theta_true_grid)**2)))
```

```
331
332 return {
333  "h_local": h_th,
334  "h_cv": h_cv,
335  "sup_theory": sup_th,
336  "l2_theory": l2_th,
337  "sup_cv": sup_cv,
338  "l2_cv": l2_cv,
339  "theta_hat_th": theta_hat_th,
340  "theta_hat_cv": theta_hat_cv,
341  "theta_true_grid": theta_true_grid,
342  "asyvar_cv": asyvar_cv,
343 }
344
345 # ----------------------------
346 # Experiment runner
347 # ----------------------------
348 def summarize(rows: List[Dict[str, float]]) -> Dict[str, float]:
349 out = {}
350 for key in ["h_local","h_cv","sup_theory","sup_cv","l2_theory","l2_cv"]:
351 vals = np.array([r[key] for r in rows], dtype=float)
352 out[key+"_mean"] = float(vals.mean())
353 out[key+"_sd"]   = float(vals.std(ddof=1)) if len(vals) > 1 else 0.0
354 return out
355
356 def run_experiment(Ns: List[int],
357 R: int,
358 copula: CopulaSpec,
359 p: int,
360 grid_m: int,
361 c_theory: float,
362 h_grid_rule: Callable[[int, int, float], List[float]],
363 seed: int = 777,
364 delta: float = 0.20) -> Tuple[pd.DataFrame, Dict[int, Dict[str, np.ndarray]]]:
365
366 rng = np.random.default_rng(seed)
```

```
grid = np.linspace(-2.0 + delta, 2.0 - delta, grid_m)

recs = []
curve_store: Dict[int, Dict[str, np.ndarray]] = {}

for N in Ns:
t0 = time.time()
rows = []

h_candidates = h_grid_rule(N, p, c_theory)

th_curves_th = []
th_curves_cv = []
asy_curves_cv = []
th_true_grid = None

for _ in range(R):
out = run_one_replication(N, copula, p, grid, rng, c_theory, h_candidates)
rows.append(out)
th_curves_th.append(out["theta_hat_th"])
th_curves_cv.append(out["theta_hat_cv"])
asy_curves_cv.append(out["asyvar_cv"])
th_true_grid = out["theta_true_grid"]

th_curves_th  = np.vstack(th_curves_th)
th_curves_cv  = np.vstack(th_curves_cv)
asy_curves_cv = np.vstack(asy_curves_cv)

curve_store[N] = {
 "grid": grid,
 "theta_true": th_true_grid,
 "theta_hat_th_mean": th_curves_th.mean(axis=0),
 "theta_hat_cv_mean": th_curves_cv.mean(axis=0),
 "asyvar_cv_mean": np.nanmean(asy_curves_cv, axis=0),
}
```

```
s = summarize(rows)
s["N"] = int(N)
s["p"] = int(p)
s["beta"] = int(p + 1)
s["copula"] = copula.name
s["time_sec"] = float(time.time() - t0)
s["n_h_candidates"] = int(len(h_candidates))

rN = minimax_rate_rN(N=int(N), beta=int(p+1), s=1)
s["minimax_rN"] = float(rN)
s["sup_theory_over_rN"] = float(s["sup_theory_mean"] / rN)
s["sup_cv_over_rN"]     = float(s["sup_cv_mean"] / rN)
s["l2_theory_over_rN"]  = float(s["l2_theory_mean"] / rN)
s["l2_cv_over_rN"]      = float(s["l2_cv_mean"] / rN)

recs.append(s)
print(f"[Clayton p={p}] N={N} done in {s['time_sec']:.1f}s (LOO-CVL)")

return pd.DataFrame(recs).sort_values("N").reset_index(drop=True), curve_store

# ----------------------------
# Plot helpers  (NO log x-scale anywhere)
# ----------------------------
def plot_bandwidths(df: pd.DataFrame, title: str):
Ns = df["N"].to_numpy(float)
plt.figure(figsize=(7,4))
plt.plot(Ns, df["h_local_mean"].to_numpy(float), marker="o", label="h_local (paper)")
plt.plot(Ns, df["h_cv_mean"].to_numpy(float), marker="s", label="h_cv (LOO-CVL)")
plt.xlabel("N")
plt.ylabel("bandwidth")
plt.title(title)
plt.grid(True, alpha=0.3)
plt.xticks(Ns.astype(int))
plt.legend()
plt.tight_layout()
plt.show()
```

```
439
440 def plot_errors(df: pd.DataFrame, title: str, which="sup"):
441 Ns = df["N"].to_numpy(float)
442 plt.figure(figsize=(7,4))
443 plt.plot(Ns, df[f"{which}_theory_mean"].to_numpy(float), marker="o", label=f"{which} error
    (h_local)")
444 plt.plot(Ns, df[f"{which}_cv_mean"].to_numpy(float), marker="s", label=f"{which} error (h_cv)")
445 plt.yscale("log")
446 plt.xlabel("N")
447 plt.ylabel(f"{which} error (log scale)")
448 plt.title(title)
449 plt.grid(True, alpha=0.3)
450 plt.xticks(Ns.astype(int))
451 plt.legend()
452 plt.tight_layout()
453 plt.show()
454
455 def plot_mean_theta_with_cv_CI(curves: Dict[int, Dict[str, np.ndarray]], N: int, title: str):
456 d = curves[N]
457 g = d["grid"]
458 true = d["theta_true"]
459 th_mean = d["theta_hat_th_mean"]
460 cv_mean = d["theta_hat_cv_mean"]
461
462 asyvar_mean = d["asyvar_cv_mean"]
463 se = np.sqrt(np.maximum(asyvar_mean, 0.0))
464 lo = cv_mean - 1.96 * se
465 hi = cv_mean + 1.96 * se
466
467 plt.figure(figsize=(7,4))
468 plt.plot(g, true, label="true theta(y)")
469 plt.plot(g, th_mean, label="mean theta_hat(y) (h_local)")
470 plt.plot(g, cv_mean, label="mean theta_hat(y) (h_cv)")
471 plt.fill_between(g, lo, hi, alpha=.25, label="95% CI (h_cv only)")
472 plt.xlabel("y")  # <-- add this
473 plt.title(title)
```

```
plt.grid(True, alpha=0.3)
plt.legend()
plt.tight_layout()
plt.show()

def plot_sup_error_vs_minimax(df: pd.DataFrame, title: str, which: str = "cv"):
sub = df.sort_values("N")
Ns  = sub["N"].to_numpy(float)
sup = sub[f"sup_{which}_mean"].to_numpy(float)
rN  = sub["minimax_rN"].to_numpy(float)

plt.figure(figsize=(7,4))
plt.plot(Ns, sup, marker="o", label=f"Mean sup error ({which})")
plt.plot(Ns, rN,  marker="s", label=r"Benchmark $r_N=(\log N/N)^{\beta/(2\beta+s)}$")
plt.yscale("log")
plt.xlabel("N")
plt.ylabel("log scale")
plt.title(title)
plt.grid(True, alpha=0.3)
plt.xticks(Ns.astype(int))
plt.legend()
plt.tight_layout()
plt.show()

def plot_scaled_sup_error(df: pd.DataFrame, title: str, which: str = "cv"):
sub = df.sort_values("N")
Ns = sub["N"].to_numpy(float)
scaled = sub[f"sup_{which}_over_rN"].to_numpy(float)

plt.figure(figsize=(7,4))
plt.plot(Ns, scaled, marker="o")
plt.xlabel("N")
plt.ylabel(r"$\mathrm{sup\ error}/r_N$")
plt.title(title)
plt.grid(True, alpha=0.3)
plt.xticks(Ns.astype(int))
```

```
plt.tight_layout()
plt.show()

# ============================================================
# SETTINGS (single p ONLY)
#   Run once with P=1.
#   After it finishes, change P=3 and re-run ONLY the RUN cell if you would like see the
   behavious of the estimator for different p.
# ============================================================
Ns = [50, 100, 250, 500, 750]
R = 100
grid_m = 60
seed = 777
delta = 0.20

P = 1   # <<< RUN THIS FIRST. Then set P = 3 and re-run the RUN cell below.

def c_theory_for_p(p: int) -> float:
return 1.0

def h_grid_rule(N: int, p: int, c_theory: float) -> List[float]:
beta = p + 1
hth = h_local(N, beta=beta, s=1, c=c_theory)
multipliers = [0.6, 0.75, 0.9, 1.0, 1.1, 1.25, 1.45]
hs = [float(hth*m) for m in multipliers]
hs = [h for h in hs if 0.03 <= h <= 1.80]
return sorted(list(set(hs)))

# ============================================================
# RUN (single p)
#   When finished, change P above and re-run THIS cell only.
# ============================================================
df, curves = run_experiment(
Ns=Ns, R=R, copula=clayton, p=P, grid_m=grid_m,
c_theory=c_theory_for_p(P),
h_grid_rule=h_grid_rule,
```

```
seed=seed,
delta=delta
)

# ============================================================
# TABLE + SAVE (single p)
# ============================================================
pd.set_option("display.precision", 4)

cols = [
"copula","p","beta","N",
"h_local_mean","h_local_sd",
"h_cv_mean","h_cv_sd",
"minimax_rN",
"sup_theory_mean","sup_theory_sd",
"sup_cv_mean","sup_cv_sd",
"sup_theory_over_rN",
"sup_cv_over_rN",
"l2_theory_mean","l2_theory_sd",
"l2_cv_mean","l2_cv_sd",
"l2_theory_over_rN",
"l2_cv_over_rN",
"n_h_candidates","time_sec"
]

print(f"\n=== Clayton-only Summary (p={P}) ===")

# ---- Colab-safe full table print (no truncation, no hidden columns)
with pd.option_context(
"display.max_columns", None,
"display.width", 3000,
"display.max_colwidth", None,
"display.expand_frame_repr", False,
"display.precision", 4
):
print(df[cols].sort_values(["p","N"]).to_string(index=False))
```

```
581
582 out_csv = f"clayton_only_LOOcv_p{P}_with_CI_and_minimax_benchmark.csv"
583 df.to_csv(out_csv, index=False)
584 print(f"\nSaved: {out_csv}")
585
586 # ============================================================
587 # PLOTS (single p)
588 # ============================================================
589 sub = df.sort_values("N")
590 plot_bandwidths(sub, f"Clayton bandwidth comparison (LOO-CVL), p={P}")
591 plot_errors(sub,     f"Clayton sup-norm errors, p={P}", which="sup")
592 plot_errors(sub,     f"Clayton L2 errors, p={P}", which="l2")
593 plot_sup_error_vs_minimax(sub, f"Clayton: sup error vs minimax benchmark, p={P} (h_cv)",
       which="cv")
594 plot_scaled_sup_error(sub,     f"Clayton: scaled sup error sup_cv/r_N vs N, p={P}", which="cv")
595
596 Nmax = int(sub["N"].max())
597 plot_mean_theta_with_cv_CI(
598 curves,
599 Nmax,
600 f"Clayton mean theta_hat(y) over R reps, p={P}, N={Nmax} (CI for h_cv only)"
601 )
```