```python
# ============================================================
# TEİAŞ DGP 0-1 — FINAL PIPELINE (Multi-State + Hankel + AFFINE DMD/DMDc)
#  FULL MERGED VERSION (RUN ONCE) — EXTENDED (Excel matrix exports + RF/LR comparison)
#
# ☑ Two preprocessing modes:
#    (A) Pipeline 1..9 FULL (Original/Long/6h/Norm/WideByDate)
#    (B) TPYS FAST + (log1p -> MinMax row-wise)
#
# ☑ Affine DMD:   x⁺ = A x + b
# ☑ Affine DMDc:  x⁺ = A x + B u + b
# ☑ Hankel-Affine versions
# ☑ Optional spike hybrid correction (in-sample)
#
# ☑ Exports to Excel (each in its own sheet):
#   - X_real, X1, X2
#   - A, b (+ Hankel Ah, bh)
#   - DMD:  Xaffine_dmd (Xdmd), Xaffine_dmd_hankel (Xhan)
#   - DMDc: Xaffine_dmdc (Xrec), Xaffine_dmdc_hankel (Xh)
#   - Normalized versions (0..1): Xn, Xdmd_n, Xhan_n, Un, Xrec_n, Xh_n
#
# ☑ NEW (YOU REQUESTED):
#   1) Read/keep matrices: X_real, X1, X2, A, B, Xaffine outputs
#      and export them to RESULTS_ALL.xlsx each in its own sheet.
#   2) RF & LR ML comparison (1-year horizon):
#      - Train on earlier part, test on last "year" samples
#      - Plot "Real vs Pred" + "Error curve" for RF and LR
#      - Performance evaluation + graphical analysis
#
# IMPORTANT REQUIREMENT (YOUR NOTE):
#   Hankel outputs MUST NOT have negative values and should be in [0, 1].
#   Therefore: we apply Hankel only on normalized matrices, then clip to [0,1]
#   before saving / returning.
# ============================================================

import os
import itertools
import numpy as np
import pandas as pd

import matplotlib
matplotlib.use("Agg")
import matplotlib.pyplot as plt

from numpy.linalg import svd

# ----------------------------
# 0) PATHS & FLAGS
# ----------------------------
DATA_DIR = r"C:\Users\OMER\new dmd tias kodlarım"   # [P01] Workspace folder path (change if your files moved)
XLSX_NAME = "DGP_0-1_Kodlu_Talimat_Hacimleri_regime6h_NORMALIZED.xlsx"  # [P02] Input Excel filename
XLSX_PATH = os.path.join(DATA_DIR, XLSX_NAME)

TPYS_SHEET = "TPYS"  # [P03] Sheet name containing TPYS data

# ============================================================
# (NEW) Choose ONE preprocessing mode
# ============================================================
USE_PIPELINE_1_9 = True      # [P04] True => Pipeline 1..9 FULL
USE_TPYS_FAST    = False     # [P05] True => TPYS FAST + log1p->MinMax row-wise

if USE_PIPELINE_1_9 == USE_TPYS_FAST:
    raise ValueError("Choose exactly ONE preprocessing mode: set USE_PIPELINE_1_9=True and USE_TPYS_FAST=False (or vice versa).")

DO_CLEAN_NORMALIZE = True     # [P06] True => run preprocessing (clean/normalize). False => only read wide

# Output for pipeline 1..9 workbook
PIPELINE_1_9_OUT_NAME = "PIPELINE_6H_PREPROCESS.xlsx"  # [P07] Output Excel for steps
```

```
    1..9
68  PIPELINE_1_9_OUT_PATH = os.path.join(DATA_DIR, PIPELINE_1_9_OUT_NAME)
69
70  # Output for TPYS fast processed wide
71  PROCESSED_WIDE_NAME = "TPYS_WIDE_PROCESSED.xlsx"  # [P08] Wide output for fast path
72  PROCESSED_WIDE_PATH = os.path.join(DATA_DIR, PROCESSED_WIDE_NAME)
73  PROCESSED_WIDE_SHEET = "WIDE"  # [P09] Sheet name inside the processed wide file
74
75  OUT_XLSX = os.path.join(DATA_DIR, "RESULTS_ALL.xlsx")  # [P10] Final results Excel
76  FIG_ROOT = os.path.join(DATA_DIR, "RESULTS_FIGS")      # [P11] Figures output folder
77
78  CALIBRATION_MODE = "auto"  # [P12] "manual" or "auto" (Grid Search)
79
80  SAVE_PLOTS = True          # [P13] Save plots to disk
81  PLOT_STEP_3D = 4           # [P14] Sampling step for 3D surfaces
82
83  # ------------------------------
84  # (NEW) AFFINE + SPIKES OPTIONS
85  # ------------------------------
86  USE_AFFINE_MODELS = True           # [P15] Enable affine DMD/DMDc
87  USE_HANKEL_AFFINE = True           # [P16] Enable Hankel-affine versions
88  USE_SPIKE_HYBRID  = True           # [P17] Spike correction (in-sample blending)
89  SPIKE_Q           = 0.995          # [P18] Quantile over |diff| to detect spikes
90  SPIKE_BLEND       = 0.85           # [P19] Blend ratio of real at spike indices
91
92  # ------------------------------
93  # 0.1) GRID CONTROL
94  # ------------------------------
95  MAX_TRIALS_DMD  = 80    # [P20] Maximum grid trials for DMD
96  MAX_TRIALS_DMDc = 120   # [P21] Maximum grid trials for DMDc
97
98  PLOTS_DURING_GRID = False  # [P22] If True, grid becomes slower due to plots
99
100 PATIENCE_DMD  = 25   # [P23] Early stop patience for DMD grid
101 PATIENCE_DMDc = 35   # [P24] Early stop patience for DMDc grid
102
103 SMART_GRID = True              # [P25] Enable stage-2 smart expansion
104 STAGE1_ONLY = False            # [P26] If True, stop after stage-1
105 STAGE2_TOP_CANDIDATES = 2      # [P27] Expand around top K candidates
106 STAGE2_MULTIPLIERS_TIK = [0.5, 1.0, 2.0]  # [P28] Tikhonov multipliers around best tik
107 STAGE2_NEIGHBOR_RANK = [-2, 0, +2]        # [P29] Rank neighbors around best
108
109 # ------------------------------
110 # 1) Your column names in TPYS
111 # ------------------------------
112 DATE_COL = "Geçerlilik Tarihi"  # [P30] Date column name in TPYS
113 HOUR_COL = "Saat"               # [P31] Hour column name in TPYS
114
115 TPYS_COLS_REQUIRED = [
116     "Net Talimat",
117     "Yal 0 Miktar",
118     "Yal 1 Miktar",
119     "Yat 0 Miktar",
120     "Yat 1 Miktar",
121     "Yerine Getirilen YAL",
122     "Yerine Getirilen YAT",
123 ]  # [P32] Required TPYS columns (update if your Excel changes)
124
125 # ------------------------------
126 # 2) Multi-State DMDc definition (EXACT)
127 # ------------------------------
128 STATE_ROWS = [
129     "Yerine Getirilen YAL",
130     "Yerine Getirilen YAT",
131     "Net Talimat",
132 ]  # [P33] State rows used in DMDc
133
134 CONTROL_ROWS = [
135     "Yal 0 Miktar",
136     "Yal 1 Miktar",
137     "Yat 0 Miktar",
```

```
    "Yat 1 Miktar",
]  # [P34] Control rows used in DMDc

DMD_ROWS = STATE_ROWS[:]  # [P35] DMD rows for fair comparison (can change for experiments)

# ----------------------------
# 3) MANUAL PARAMS (baseline)
# ----------------------------
MANUAL = dict(
    # ---- DMD ----
    DMD_RANK_MAX=40,           # [P36]
    DMD_TIK=1e-2,              # [P37]
    DMD_STABLE=True,           # [P38]
    DMD_RHO_MAX=0.995,         # [P39]
    DMD_CLIP=None,             # [P40] Clip for rollout (None disables)

    # Hankel-DMD
    DMD_HANKEL_D=16,           # [P41] Hankel window length d
    DMD_HANKEL_RANK_MAX=120,   # [P42]
    DMD_HANKEL_TIK=1e-2,       # [P43]
    DMD_HANKEL_STABLE=True,    # [P44]
    DMD_HANKEL_RHO_MAX=0.995,  # [P45]
    DMD_HANKEL_CLIP=None,      # [P46]

    # ---- DMDc ----
    DMDc_RANK_OMEGA=12,        # [P47]
    DMDc_TIK=1e-2,             # [P48]
    DMDc_STABLE=True,          # [P49]
    DMDc_RHO_MAX=0.995,        # [P50]
    DMDc_CLIP=None,            # [P51]

    # Hankel(DMDc)
    DMDc_HANKEL_D=16,          # [P52]
    DMDc_HANKEL_RANK_MAX=120,  # [P53]
    DMDc_HANKEL_TIK=1e-2,      # [P54]
    DMDc_HANKEL_STABLE=True,   # [P55]
    DMDc_HANKEL_RHO_MAX=0.995,# [P56]
    DMDc_HANKEL_CLIP=None,     # [P57]
)

# ----------------------------
# 4) GRID SEARCH (Stage-1 safe)
# ----------------------------
GRID = dict(
    # --- DMD + Hankel Stage-1 ---
    DMD_RANK_GRID=[20, 40],             # [P58]
    DMD_TIK_GRID=[1e-2],                # [P59]
    DMD_RHO_MAX_GRID=[0.995],           # [P60]
    DMD_CLIP_GRID=[None],               # [P61]

    DMD_HANKEL_D_GRID=[16],             # [P62]
    DMD_HANKEL_RANK_GRID=[80, 120],     # [P63]
    DMD_HANKEL_TIK_GRID=[3e-2, 1e-2],   # [P64]
    DMD_HANKEL_RHO_MAX_GRID=[0.995],    # [P65]
    DMD_HANKEL_CLIP_GRID=[None],        # [P66]

    # --- DMDc + Hankel Stage-1 ---
    DMDc_RANK_OMEGA_GRID=[10, 15],      # [P67]
    DMDc_TIK_GRID=[1e-2, 3e-3],         # [P68]
    DMDc_RHO_MAX_GRID=[0.995],          # [P69]
    DMDc_CLIP_GRID=[None],              # [P70]

    DMDc_HANKEL_D_GRID=[16],            # [P71]
    DMDc_HANKEL_RANK_GRID=[80, 120],    # [P72]
    DMDc_HANKEL_TIK_GRID=[3e-2, 1e-2], # [P73]
    DMDc_HANKEL_RHO_MAX_GRID=[0.995],   # [P74]
    DMDc_HANKEL_CLIP_GRID=[None],       # [P75]

    TOP_K=20  # [P76] How many top grid results to print
)
```

```
208
209 # ============================================================
210 # ☑ NEW (ML comparison controls)
211 # ============================================================
212 ENABLE_ML_COMPARE = True        # [P82] Enable RF / LR comparison
213 YEAR_SAMPLES_6H = 1460          # [P83] ~1 year samples for 6-hour data (4 points/day)
214 ML_TEST_SAMPLES = None          # [P84] None => uses YEAR_SAMPLES_6H or adapts to
    dataset length
215 ML_RANDOM_STATE = 42            # [P85] Random seed
216 RF_N_ESTIMATORS = 400           # [P86] Number of trees in RF
217 RF_MAX_DEPTH = None             # [P87] Tree depth (None = unlimited)
218 RF_MIN_SAMPLES_LEAF = 2         # [P88] Leaf smoothing
219 PLOT_ML_MAX_POINTS = 2500       # [P89] Plot downsampling limit for ML curves
220
221 # ============================================================
222 # Utilities
223 # ============================================================
224
225 def ensure_dir(path: str) -> str:
226     os.makedirs(path, exist_ok=True)
227     return path
228
229 def _canon(s: str) -> str:
230     if s is None:
231         return ""
232     s = str(s).replace("\u00a0", " ")
233     s = " ".join(s.strip().split())
234     return s
235
236 def preview_matrix(M, name="M", max_rows=6, max_cols=10):
237     M = np.asarray(M)
238     print(f"\n[{name}] shape={M.shape}  min={np.nanmin(M):.6g}  max={np.nanmax(M)
    :.6g}")
239     rr = min(max_rows, M.shape[0])
240     cc = min(max_cols, M.shape[1])
241     print(np.array2string(M[:rr, :cc], precision=4, suppress_small=True))
242
243 def df_from_matrix(X, row_names=None, col_names=None):
244     """Helper to create a labeled DataFrame for Excel export."""
245     X = np.asarray(X, dtype=float)
246     if row_names is None:
247         row_names = [f"r{i}" for i in range(X.shape[0])]
248     if col_names is None:
249         col_names = [f"t{k}" for k in range(X.shape[1])]
250     return pd.DataFrame(X, index=row_names, columns=col_names)
251
252 def df_from_vector(v, row_names=None, col_name="value"):
253     v = np.asarray(v, dtype=float).reshape(-1)
254     if row_names is None:
255         row_names = [f"r{i}" for i in range(len(v))]
256     return pd.DataFrame({col_name: v}, index=row_names)
257
258 def df_from_square(M, names=None):
259     M = np.asarray(M, dtype=float)
260     if names is None:
261         names = [f"s{i}" for i in range(M.shape[0])]
262     return pd.DataFrame(M, index=names, columns=names)
263
264 def safe_sheet_name(name: str) -> str:
265     """Excel sheet name: max 31 chars and cannot contain : \ / ? * [ ]"""
266     bad = [":", "\\", "/", "?", "*", "[", "]"]
267     out = str(name)
268     for b in bad:
269         out = out.replace(b, "_")
270     out = out[:31]
271     if not out:
272         out = "Sheet"
273     return out
274
275 # ============================================================
276 # (NEW) Hankelize / DeHankelize
```

```python
# ============================================================

def hankelize(X, d):
    X = np.asarray(X, dtype=float)
    n, T = X.shape
    if d >= T:
        raise ValueError("Hankel d must be < T")
    cols = T - d + 1
    H = np.zeros((n*d, cols), dtype=float)
    for i in range(d):
        H[i*n:(i+1)*n, :] = X[:, i:i+cols]
    return H

def dehankelize(H, n, T, d):
    H = np.asarray(H, dtype=float)
    cols = H.shape[1]
    Xrec = np.zeros((n, T), dtype=float)
    cnt  = np.zeros((n, T), dtype=float)
    for i in range(d):
        blk = H[i*n:(i+1)*n, :]
        Xrec[:, i:i+cols] += blk
        cnt[:,  i:i+cols] += 1.0
    return Xrec / np.maximum(cnt, 1.0)

# ============================================================
# (NEW) log1p -> MinMax (row-wise) + inverse
# ============================================================

def log1p_minmax_rows(X):
    X = np.asarray(X, dtype=float)
    Xp = np.log1p(np.maximum(X, 0.0))  # keep nonnegative before log1p
    mn = np.min(Xp, axis=1, keepdims=True)
    mx = np.max(Xp, axis=1, keepdims=True)
    denom = (mx - mn)
    denom[denom == 0] = 1.0
    Xn = (Xp - mn) / denom
    return Xn, mn, mx

def inv_log1p_minmax_rows(Xn, mn, mx):
    Xn = np.asarray(Xn, dtype=float)
    Xp = Xn * (mx - mn) + mn
    X  = np.expm1(Xp)
    return X

# ============================================================
# (NEW) Spike hybrid correction (in-sample)
# ============================================================

def spike_hybrid_blend(X_real, X_pred, q=0.995, blend=0.85):
    if (not USE_SPIKE_HYBRID) or blend <= 0:
        return X_pred
    X_real = np.asarray(X_real, float)
    X_pred = np.asarray(X_pred, float)
    m, T = X_real.shape
    X_out = X_pred.copy()

    for i in range(m):
        d = np.abs(np.diff(X_real[i], prepend=X_real[i, 0]))
        thr = np.quantile(d, q)
        spike_idx = np.where(d >= thr)[0]
        if spike_idx.size > 0:
            X_out[i, spike_idx] = (1.0 - blend) * X_out[i, spike_idx] + blend * X_real
            [i, spike_idx]
    return X_out

# ============================================================
# Robust time parsing (used by FAST path)
# ============================================================

def _format_hour_cell(h):
    if pd.isna(h):
```

```
        return ""
    hs = str(h).strip()
    if ":" in hs:
        return hs
    try:
        hh = int(float(hs))
        return f"{hh:02d}:00"
    except Exception:
        return hs

def build_df_wide_from_tpys(xlsx_path: str, sheet_name: str) -> pd.DataFrame:
    if not os.path.exists(xlsx_path):
        raise FileNotFoundError(f"Excel not found: {xlsx_path}")

    df = pd.read_excel(xlsx_path, sheet_name=sheet_name)
    df.columns = [_canon(c) for c in df.columns]

    colset = set(df.columns)
    for col in [DATE_COL, HOUR_COL] + TPYS_COLS_REQUIRED:
        if _canon(col) not in colset:
            raise ValueError(f"Missing required column in TPYS: '{col}'")

    date_col = _canon(DATE_COL)
    hour_col = _canon(HOUR_COL)

    date_s = df[date_col].astype(str).str.strip()
    hour_s = df[hour_col].apply(_format_hour_cell).astype(str).str.strip()
    dt_str = date_s + " " + hour_s

    dt_try = pd.to_datetime(dt_str, format="%d.%m.%Y %H:%M", errors="coerce")
    if dt_try.isna().all():
        dt_try = pd.to_datetime(dt_str, format="%Y-%m-%d %H:%M", errors="coerce")
    if dt_try.isna().all():
        dt_try = pd.to_datetime(dt_str, errors="coerce")

    if dt_try.notna().sum() > 0:
        df["_t"] = dt_try
        df = df.sort_values("_t")
        time_index = df["_t"].astype(str).tolist()
    else:
        df = df.sort_values([date_col, hour_col])
        time_index = (df[date_col].astype(str) + " " + df[hour_col].astype(str)).
        tolist()

    Xcols = [_canon(c) for c in TPYS_COLS_REQUIRED]
    X = df[Xcols].apply(pd.to_numeric, errors="coerce")
    X = X.interpolate(limit_direction="both").fillna(0.0)

    df_wide = X.T
    df_wide.columns = time_index
    df_wide.index = Xcols
    df_wide.index = [_canon(i) for i in df_wide.index]
    return df_wide

def pick_rows_from_dfwide(df_wide: pd.DataFrame, wanted_rows):
    rows = []
    names = []
    for r in wanted_rows:
        rc = _canon(r)
        if rc in df_wide.index:
            rows.append(df_wide.loc[rc].values.astype(float))
            names.append(rc)
        else:
            print(f"[WARN] Row missing: {r}")
    if len(rows) == 0:
        raise ValueError("No rows found in wide sheet for requested rows.")
    X = np.vstack(rows)
    return X, names

# ============================================================
# (NEW) Pipeline 1..9 FULL (your exact steps) + log1p before MinMax
```

```python
# ============================================================

def label_regime(hour_int: int) -> str:
    if 0 <= hour_int < 6:
        return "R00_06"
    if 6 <= hour_int < 12:
        return "R06_12"
    if 12 <= hour_int < 18:
        return "R12_18"
    return "R18_24"

def _parse_date_series(s: pd.Series) -> pd.Series:
    dt = pd.to_datetime(s, errors="coerce", dayfirst=True)
    return dt

def preprocess_pipeline_1_9(
    xlsx_path: str,
    sheet_name: str,
    out_path: str,
    date_col_name: str,
    hour_col_name: str,
    value_cols: list,
    round_digits: int = 5   # [P77] Rounding digits used in minmax (pipeline 1..9)
) -> pd.DataFrame:

    if not os.path.exists(xlsx_path):
        raise FileNotFoundError(f"Excel not found: {xlsx_path}")

    df = pd.read_excel(xlsx_path, sheet_name=sheet_name)
    df.columns = [_canon(c) for c in df.columns]

    date_col = _canon(date_col_name)
    hour_col = _canon(hour_col_name)
    val_cols = [_canon(c) for c in value_cols]

    missing = [c for c in [date_col, hour_col] + val_cols if c not in df.columns]
    if missing:
        raise ValueError(f"Missing columns in TPYS: {missing}")

    original_df = df.copy()

    date_dt = _parse_date_series(df[date_col].astype(str).str.strip())
    hour_raw = df[hour_col].apply(_format_hour_cell).astype(str).str.strip()
    hour_num = pd.to_datetime(hour_raw, format="%H:%M", errors="coerce").dt.hour
    hour_num = hour_num.fillna(pd.to_numeric(df[hour_col], errors="coerce"))
    hour_num = hour_num.fillna(0).astype(int).clip(0, 23)

    df["__date__"] = date_dt.dt.date.astype(str)
    df["__hour__"] = hour_num.astype(int)
    df["__datetime__"] = pd.to_datetime(df["__date__"]) + pd.to_timedelta(df[
    "__hour__"], unit="h")
    df = df.sort_values("__datetime__")

    long_df = df.melt(
        id_vars=["__datetime__", "__date__", "__hour__"],
        value_vars=val_cols,
        var_name="variable",
        value_name="value"
    )
    long_df["variable"] = long_df["variable"].apply(_canon)
    long_df["value"] = pd.to_numeric(long_df["value"], errors="coerce")

    long_df["regime"] = long_df["__hour__"].apply(label_regime)
    avg_df = (
        long_df
        .groupby(["__date__", "variable", "regime"], as_index=False)["value"]
        .mean()
    )

    avg_pivot = avg_df.pivot_table(
        index=["__date__", "variable"],
```

```
            columns="regime",
            values="value",
            aggfunc="mean"
        ).reset_index()

        for c in ["R00_06", "R06_12", "R12_18", "R18_24"]:
            if c not in avg_pivot.columns:
                avg_pivot[c] = np.nan

        # log1p before MinMax, per column
        norm_df = avg_pivot.copy()
        info_rows = []
        for c in ["R00_06", "R06_12", "R12_18", "R18_24"]:
            col = pd.to_numeric(norm_df[c], errors="coerce").fillna(0.0).values.astype(
            float)
            col_lp = np.log1p(np.maximum(col, 0.0))
            mn = float(np.min(col_lp))
            mx = float(np.max(col_lp))
            denom = (mx - mn) if (mx - mn) != 0 else 1.0
            norm_df[c] = ((col_lp - mn) / denom).round(round_digits)
            info_rows.append(dict(column=c, min_log1p=mn, max_log1p=mx, denom=denom))

        norm_info = pd.DataFrame(info_rows)

        check_df = norm_df[["__date__", "variable", "R00_06", "R06_12", "R12_18", "R18_24"
        ]].copy()
        check_df["row_min"] = check_df[["R00_06", "R06_12", "R12_18", "R18_24"]].min(axis=
        1)
        check_df["row_max"] = check_df[["R00_06", "R06_12", "R12_18", "R18_24"]].max(axis=
        1)

        rename_map = {"R00_06": "z1", "R06_12": "z2", "R12_18": "z3", "R18_24": "z4"}
        norm_df = norm_df.rename(columns=rename_map)
        norm_df["variable"] = norm_df["variable"].apply(_canon)

        target_rows = TPYS_COLS_REQUIRED[:]  # [P78] You can change target rows for
        pipeline 1..9
        norm_df = norm_df[norm_df["variable"].isin([_canon(x) for x in target_rows])].copy
        ()
        norm_df["variable"] = pd.Categorical(norm_df["variable"], categories=[_canon(x)
        for x in target_rows], ordered=True)
        norm_df = norm_df.sort_values(["variable", "__date__"])

        wide_blocks = []
        dates = sorted(norm_df["__date__"].unique().tolist())
        for d in dates:
            sub = norm_df[norm_df["__date__"] == d].copy()
            sub = sub.set_index("variable")[["z1", "z2", "z3", "z4"]]
            sub.columns = [f"{d}_{c}" for c in sub.columns]
            wide_blocks.append(sub)

        wide_by_date = pd.concat(wide_blocks, axis=1)
        wide_by_date.index = wide_by_date.index.astype(str).map(_canon)

        with pd.ExcelWriter(out_path, engine="openpyxl") as writer:
            original_df.to_excel(writer, sheet_name="Original", index=False)
            long_df.to_excel(writer, sheet_name="Long_Unpivoted", index=False)
            avg_pivot.to_excel(writer, sheet_name="Regime_6h_Averages", index=False)
            norm_df.to_excel(writer, sheet_name="Regime_6h_Normalized", index=False)
            check_df.to_excel(writer, sheet_name="Row_MinMax_Check", index=False)
            norm_info.to_excel(writer, sheet_name="Normalization_Info", index=False)
            wide_by_date.to_excel(writer, sheet_name="Regime_6h_ByDate_Wide")

        return wide_by_date

    # ============================================================
    # Metrics
    # ============================================================

    def metrics_rowwise(X_true, X_pred, row_names):
        X_true = np.asarray(X_true, dtype=float)
```

```
    X_pred = np.asarray(X_pred, dtype=float)
    eps = 1e-12
    out = []
    for i, nm in enumerate(row_names):
        y = X_true[i]
        yh = X_pred[i]
        err = y - yh
        rmse = float(np.sqrt(np.mean(err**2)))
        mae  = float(np.mean(np.abs(err)))
        ss_res = float(np.sum(err**2))
        ss_tot = float(np.sum((y - np.mean(y))**2)) + eps
        r2 = float(1.0 - ss_res/ss_tot)
        out.append(dict(row=nm, RMSE=rmse, MAE=mae, R2=r2))
    df = pd.DataFrame(out)
    score = float(df["RMSE"].mean() + df["MAE"].mean() + (1.0 - df["R2"].mean()))
    return df, score

def rmse_over_time(X_true, X_pred):
    X_true = np.asarray(X_true, float)
    X_pred = np.asarray(X_pred, float)
    err = X_true - X_pred
    return np.sqrt(np.mean(err**2, axis=0))

# ============================================================
# Plotting
# ============================================================

def plot_surface_save(X, title, png_path, step=4):
    X = np.asarray(X, dtype=float)
    m, T = X.shape
    xs = np.arange(0, T, step)
    ys = np.arange(0, m, 1)
    Xs = X[:, xs]

    from mpl_toolkits.mplot3d import Axes3D  # noqa
    fig = plt.figure(figsize=(10, 6))
    ax = fig.add_subplot(111, projection='3d')

    Xgrid, Ygrid = np.meshgrid(xs, ys)
    ax.plot_surface(Xgrid, Ygrid, Xs, linewidth=0, antialiased=True)
    ax.set_title(title)
    ax.set_xlabel("time index")
    ax.set_ylabel("row")
    ax.set_zlabel("value")
    plt.tight_layout()
    fig.savefig(png_path, dpi=150)
    plt.close(fig)

def plot_2d_statewise(X_true, X_pred, row_names, png_path, max_points=2000):  # [P79] max_points controls plot size
    X_true = np.asarray(X_true, dtype=float)
    X_pred = np.asarray(X_pred, dtype=float)
    m, T = X_true.shape
    idx = np.linspace(0, T-1, min(T, max_points)).astype(int)

    fig = plt.figure(figsize=(14, 3*m))
    for i in range(m):
        ax = fig.add_subplot(m, 1, i+1)
        ax.plot(idx, X_true[i, idx], label="Real")
        ax.plot(idx, X_pred[i, idx], label="Pred")
        ax.set_title(f"{row_names[i]} (Real vs Pred)")
        ax.grid(True, alpha=0.3)
        if i == 0:
            ax.legend()
    plt.tight_layout()
    fig.savefig(png_path, dpi=150)
    plt.close(fig)

def plot_heatmap(M, xlabels, ylabels, title, png_path):
    M = np.asarray(M, dtype=float)
    fig = plt.figure(figsize=(10, 6))
```

```
    ax = fig.add_subplot(111)
    im = ax.imshow(M, aspect="auto")
    ax.set_title(title)
    ax.set_xticks(np.arange(len(xlabels)))
    ax.set_yticks(np.arange(len(ylabels)))
    ax.set_xticklabels(xlabels, rotation=45, ha="right")
    ax.set_yticklabels(ylabels)
    fig.colorbar(im, ax=ax, shrink=0.8)
    plt.tight_layout()
    fig.savefig(png_path, dpi=150)
    plt.close(fig)

def plot_surfaces_compare_grid(X_list, titles, png_path, step=4, ncols=3):  # [P80] ncols controls grid width
    from mpl_toolkits.mplot3d import Axes3D  # noqa
    n = len(X_list)
    nrows = int(np.ceil(n / ncols))

    fig = plt.figure(figsize=(6*ncols, 4*nrows))
    for i, (X, ttl) in enumerate(zip(X_list, titles)):
        X = np.asarray(X, dtype=float)
        m, T = X.shape
        xs = np.arange(0, T, step)
        ys = np.arange(0, m, 1)
        Xs = X[:, xs]
        Xgrid, Ygrid = np.meshgrid(xs, ys)

        ax = fig.add_subplot(nrows, ncols, i+1, projection='3d')
        ax.plot_surface(Xgrid, Ygrid, Xs, linewidth=0, antialiased=True)
        ax.set_title(ttl)
        ax.set_xlabel("time")
        ax.set_ylabel("row")
        ax.set_zlabel("val")

    plt.tight_layout()
    fig.savefig(png_path, dpi=150)
    plt.close(fig)

def plot_ml_compare_one_year(time_labels, X_true_year, X_pred_year, row_names, model_name, out_dir):
    """
    Visualization package:
      - Real vs Pred (per state row)
      - Error curve (RMSE over time)
    """
    ensure_dir(out_dir)
    X_true_year = np.asarray(X_true_year, float)
    X_pred_year = np.asarray(X_pred_year, float)

    # Downsample for plotting
    T = X_true_year.shape[1]
    maxp = min(T, PLOT_ML_MAX_POINTS)
    idx = np.linspace(0, T-1, maxp).astype(int)

    # Real vs Pred
    fig = plt.figure(figsize=(14, 3*X_true_year.shape[0]))
    for i, nm in enumerate(row_names):
        ax = fig.add_subplot(X_true_year.shape[0], 1, i+1)
        ax.plot(idx, X_true_year[i, idx], label="Real")
        ax.plot(idx, X_pred_year[i, idx], label=f"{model_name} Pred")
        ax.set_title(f"{model_name} — {nm} (1-year) Real vs Pred")
        ax.grid(True, alpha=0.3)
        if i == 0:
            ax.legend()
    plt.tight_layout()
    fig.savefig(os.path.join(out_dir, f"ML_{model_name}_1Y_RealVsPred.png"), dpi=150)
    plt.close(fig)

    # Error curve (RMSE across rows)
    e = rmse_over_time(X_true_year, X_pred_year)
    fig = plt.figure(figsize=(14, 4))
```

```
    ax = fig.add_subplot(111)
    ax.plot(np.arange(len(e))[idx], e[idx], label="RMSE(t)")
    ax.set_title(f"{model_name} — 1-year Error Curve (RMSE over states)")
    ax.set_xlabel("time index (year window)")
    ax.set_ylabel("RMSE")
    ax.grid(True, alpha=0.3)
    ax.legend()
    plt.tight_layout()
    fig.savefig(os.path.join(out_dir, f"ML_{model_name}_1Y_ErrorCurve.png"), dpi=150)
    plt.close(fig)

# ============================================================
# Stability helpers
# ============================================================

def _project_stable(A, rho_max=0.995):
    eigvals = np.linalg.eigvals(A)
    rad = np.max(np.abs(eigvals))
    if np.isfinite(rad) and rad > rho_max:
        A = A * (rho_max / rad)
    return A

# ============================================================
# (NEW) AFFINE DMD: x⁺ = A x + b
# ============================================================

def affine_dmd_fit_reconstruct(X, rmax=40, tik=1e-2, stable=True, rho_max=0.995,
clip_val=None):
    X = np.asarray(X, dtype=float)
    X1 = X[:, :-1]
    X2 = X[:, 1:]
    n, _ = X1.shape

    Omega = np.vstack([X1, np.ones((1, X1.shape[1]))])  # [X; 1]
    Uo, so, Vho = svd(Omega, full_matrices=False)
    r = min(rmax, len(so))
    Uor = Uo[:, :r]
    Sor = np.diag(so[:r])
    Vor = Vho.conj().T[:, :r]

    Sinv = np.linalg.inv(Sor + tik*np.eye(r))
    AB = X2 @ Vor @ Sinv @ Uor.T  # (n x (n+1))
    A = AB[:, :n]
    b = AB[:, -1]

    if stable:
        A = _project_stable(A, rho_max=rho_max)

    Xrec = np.zeros_like(X)
    Xrec[:, 0] = X[:, 0]
    for k in range(X.shape[1]-1):
        xnext = (A @ Xrec[:, k] + b).astype(float)
        if clip_val is not None:
            xnext = np.clip(xnext, -clip_val, clip_val)
        if not np.all(np.isfinite(xnext)):
            raise FloatingPointError("Affine-DMD rollout exploded.")
        Xrec[:, k+1] = xnext
    return A, b, Xrec

# ============================================================
# (NEW) AFFINE DMDc: x⁺ = A x + B u + b
# ============================================================

def fit_affine_dmdc(X_state, U_ctrl, r_omega=12, tik=1e-2, stable=True, rho_max=0.995
):
    X_state = np.asarray(X_state, dtype=float)
    U_ctrl  = np.asarray(U_ctrl, dtype=float)
    n, T = X_state.shape
    p, Tu = U_ctrl.shape
    if Tu != T:
        raise ValueError("U_ctrl must have same time length as X_state")
```

```
759
760         X1 = X_state[:, :-1]
761         X2 = X_state[:, 1:]
762         U1 = U_ctrl[:, :-1]
763
764         Omega = np.vstack([X1, U1, np.ones((1, T-1))])  # [X; U; 1]
765
766         Uo, so, Vho = svd(Omega, full_matrices=False)
767         r = min(r_omega, len(so))
768         Uor = Uo[:, :r]
769         Sor = np.diag(so[:r])
770         Vor = Vho.conj().T[:, :r]
771
772         Sinv = np.linalg.inv(Sor + tik*np.eye(r))
773         ABb = X2 @ Vor @ Sinv @ Uor.T  # (n x (n+p+1))
774
775         A = ABb[:, :n]
776         B = ABb[:, n:n+p]
777         b = ABb[:, -1]
778
779         if stable:
780             A = _project_stable(A, rho_max=rho_max)
781
782         return A, B, b
783
784     def simulate_affine_dmdc(A, B, b, x0, U_ctrl, clip_val=None):
785         U_ctrl = np.asarray(U_ctrl, dtype=float)
786         n = A.shape[0]
787         T = U_ctrl.shape[1]
788         Xrec = np.zeros((n, T), dtype=float)
789         Xrec[:, 0] = np.asarray(x0, dtype=float).reshape(-1)
790         for k in range(T-1):
791             xnext = (A @ Xrec[:, k] + B @ U_ctrl[:, k] + b).astype(float)
792             if clip_val is not None:
793                 xnext = np.clip(xnext, -clip_val, clip_val)
794             if not np.all(np.isfinite(xnext)):
795                 raise FloatingPointError("Affine-DMDc rollout exploded.")
796             Xrec[:, k+1] = xnext
797         return Xrec
798
799     # ============================================================
800     # (NEW) Hankel-Affine DMD / DMDc  (Normalized-space ONLY)
801     #   - This guarantees outputs can be safely forced to [0,1]
802     # ============================================================
803
804     def hankel_affine_dmd_normalized(Xn, d=16, rmax_h=120, tik=1e-2, stable=True, rho_max=
    0.995, clip_val=None):
805         """
806         Hankel is applied only on normalized Xn.
807         Returns Xhat_n clipped to [0,1].
808         """
809         Xn = np.asarray(Xn, dtype=float)
810         H = hankelize(Xn, d)
811         A, b, Hrec = affine_dmd_fit_reconstruct(
812             H, rmax=rmax_h, tik=tik, stable=stable, rho_max=rho_max, clip_val=clip_val
813         )
814         Xhat_n = dehankelize(Hrec, Xn.shape[0], Xn.shape[1], d)
815         Xhat_n = np.clip(Xhat_n, 0.0, 1.0)  # Requirement: nonnegative + within [0,1]
816         return A, b, Xhat_n, H, Hrec
817
818     def hankel_affine_dmdc_normalized(Xn_state, Un_ctrl, d=16, r_omega=120, tik=1e-2,
    stable=True, rho_max=0.995, clip_val=None):
819         """
820         Hankel is applied on normalized Xn_state and normalized Un_ctrl.
821         Returns Xhat_n clipped to [0,1].
822         """
823         Xn_state = np.asarray(Xn_state, dtype=float)
824         Un_ctrl  = np.asarray(Un_ctrl, dtype=float)
825
826         Hx = hankelize(Xn_state, d)
827         Hu = hankelize(Un_ctrl,  d)
```

```
828
829         A, B, b = fit_affine_dmdc(Hx, Hu, r_omega=r_omega, tik=tik, stable=stable, rho_max
            =rho_max)
830         Hrec = simulate_affine_dmdc(A, B, b, Hx[:, 0], Hu, clip_val=clip_val)
831
832         Xhat_n = dehankelize(Hrec, Xn_state.shape[0], Xn_state.shape[1], d)
833         Xhat_n = np.clip(Xhat_n, 0.0, 1.0)  # Requirement: nonnegative + within [0,1]
834         return A, B, b, Xhat_n, Hx, Hu, Hrec
835
836     # ============================================================
837     # RUNNERS (UPDATED):
838     #   - Compute Xn
839     #   - DMD/DMDc in normalized space
840     #   - Hankel in normalized space (strict [0,1])
841     #   - Inverse back to real space for metrics/plots (optional)
842     # ============================================================
843
844     def run_dmd_and_hankel(X_real, row_names, params, run_dir):
845         ensure_dir(run_dir)
846         allow_plots = bool(params.get("ALLOW_PLOTS", True))
847
848         # log1p -> MinMax (row-wise)
849         Xn, mn, mx = log1p_minmax_rows(X_real)
850         Xn = np.clip(Xn, 0.0, 1.0)  # [P81] extra safety clamp
851
852         # Save splits: X1, X2
853         X1_real = X_real[:, :-1]
854         X2_real = X_real[:, 1:]
855
856         if USE_AFFINE_MODELS:
857             A, b, Xdmd_n = affine_dmd_fit_reconstruct(
858                 Xn,
859                 rmax=params["DMD_RANK_MAX"],
860                 tik=params["DMD_TIK"],
861                 stable=params["DMD_STABLE"],
862                 rho_max=params["DMD_RHO_MAX"],
863                 clip_val=params["DMD_CLIP"]
864             )
865             Xdmd_n = np.clip(Xdmd_n, 0.0, 1.0)
866
867             # Inverse to real space for evaluation/plots
868             Xdmd = inv_log1p_minmax_rows(Xdmd_n, mn, mx)
869             Xdmd = spike_hybrid_blend(X_real, Xdmd, q=SPIKE_Q, blend=SPIKE_BLEND)
870         else:
871             A, b, Xdmd_n = None, None, Xn.copy()
872             Xdmd = X_real.copy()
873
874         met_dmd, score_dmd = metrics_rowwise(X_real, Xdmd, row_names)
875
876         # Hankel layer (NORMALIZED ONLY -> strict [0,1])
877         if USE_HANKEL_AFFINE:
878             Ah, bh, Xhan_n, H_in, H_rec = hankel_affine_dmd_normalized(
879                 Xdmd_n,
880                 d=params["DMD_HANKEL_D"],
881                 rmax_h=params["DMD_HANKEL_RANK_MAX"],
882                 tik=params["DMD_HANKEL_TIK"],
883                 stable=params["DMD_HANKEL_STABLE"],
884                 rho_max=params["DMD_HANKEL_RHO_MAX"],
885                 clip_val=params["DMD_HANKEL_CLIP"]
886             )
887             Xhan = inv_log1p_minmax_rows(Xhan_n, mn, mx)
888             Xhan = spike_hybrid_blend(X_real, Xhan, q=SPIKE_Q, blend=SPIKE_BLEND)
889         else:
890             Ah, bh = None, None
891             Xhan_n = Xdmd_n.copy()
892             H_in, H_rec = None, None
893             Xhan = Xdmd.copy()
894
895         met_h, score_h = metrics_rowwise(X_real, Xhan, row_names)
896
897         if SAVE_PLOTS and allow_plots:
```

```
        plot_surface_save(X_real, "X_real (DMD rows)", os.path.join(run_dir,
        "Xreal_surface.png"), step=PLOT_STEP_3D)
        plot_surface_save(Xdmd,   "Affine-DMD (inv from normalized)", os.path.join(
        run_dir, "Xdmd_surface.png"), step=PLOT_STEP_3D)
        plot_surface_save(Xhan,   "Affine-DMD + Hankel (inv from hankel-normalized)",
        os.path.join(run_dir, "Xdmd_hankel_surface.png"), step=PLOT_STEP_3D)
        plot_2d_statewise(X_real, Xdmd, row_names, os.path.join(run_dir,
        "2D_DMD_statewise.png"))
        plot_2d_statewise(X_real, Xhan, row_names, os.path.join(run_dir,
        "2D_HankelDMD_statewise.png"))
        plot_surfaces_compare_grid(
            [X_real, Xdmd, Xhan],
            ["X_real", "Affine-DMD", "Affine-DMD+Hankel"],
            os.path.join(run_dir, "COMPARE_3D_DMD_triplet.png"),
            step=PLOT_STEP_3D, ncols=3
        )

    return dict(
        A=A, b=b, Ah=Ah, bh=bh,
        X_real=X_real, X1_real=X1_real, X2_real=X2_real,
        Xn=Xn, Xdmd_n=Xdmd_n, Xhan_n=Xhan_n,
        Xdmd=Xdmd, Xhan=Xhan,
        H_in=H_in, H_rec=H_rec,
        met_dmd=met_dmd, met_h=met_h, score_dmd=score_dmd, score_h=score_h
    )

def run_dmdc_and_hankel(X_state, U_ctrl, state_names, ctrl_names, params, run_dir):
    ensure_dir(run_dir)
    allow_plots = bool(params.get("ALLOW_PLOTS", True))

    # log1p -> MinMax for both state and control
    Xn, mnx, mxx = log1p_minmax_rows(X_state)
    Un, mnu, mxu = log1p_minmax_rows(U_ctrl)

    Xn = np.clip(Xn, 0.0, 1.0)
    Un = np.clip(Un, 0.0, 1.0)

    # Save splits requested: X1, X2 for state
    X1_state = X_state[:, :-1]
    X2_state = X_state[:, 1:]

    # Affine DMDc in normalized space
    A, B, b = fit_affine_dmdc(
        Xn, Un,
        r_omega=params["DMDc_RANK_OMEGA"],
        tik=params["DMDc_TIK"],
        stable=params["DMDc_STABLE"],
        rho_max=params["DMDc_RHO_MAX"],
    )
    Xrec_n = simulate_affine_dmdc(A, B, b, Xn[:, 0], Un, clip_val=params["DMDc_CLIP"])
    Xrec_n = np.clip(Xrec_n, 0.0, 1.0)

    # Inverse to real for metrics/plots
    Xrec = inv_log1p_minmax_rows(Xrec_n, mnx, mxx)
    Xrec = spike_hybrid_blend(X_state, Xrec, q=SPIKE_Q, blend=SPIKE_BLEND)

    met_c, score_c = metrics_rowwise(X_state, Xrec, state_names)

    # Hankel Affine DMDc (NORMALIZED ONLY -> strict [0,1])
    if USE_HANKEL_AFFINE:
        Ah, Bh, bh, Xh_n, Hx_in, Hu_in, Hx_rec = hankel_affine_dmdc_normalized(
            Xn, Un,
            d=params["DMDc_HANKEL_D"],
            r_omega=params["DMDc_HANKEL_RANK_MAX"],
            tik=params["DMDc_HANKEL_TIK"],
            stable=params["DMDc_HANKEL_STABLE"],
            rho_max=params["DMDc_HANKEL_RHO_MAX"],
            clip_val=params["DMDc_HANKEL_CLIP"],
        )

        Xh = inv_log1p_minmax_rows(Xh_n, mnx, mxx)
```

```
            Xh = spike_hybrid_blend(X_state, Xh, q=SPIKE_Q, blend=SPIKE_BLEND)
        else:
            Ah, Bh, bh = None, None, None
            Xh_n = Xrec_n.copy()
            Hx_in, Hu_in, Hx_rec = None, None, None
            Xh = Xrec.copy()

        met_h, score_h = metrics_rowwise(X_state, Xh, state_names)

        if SAVE_PLOTS and allow_plots:
            plot_surface_save(X_state, "X_state real", os.path.join(run_dir,
            "Xstate_real_surface.png"), step=PLOT_STEP_3D)
            plot_surface_save(Xrec,    "Affine-DMDc (inv from normalized)", os.path.join(
            run_dir, "XDMDc_surface.png"), step=PLOT_STEP_3D)
            plot_surface_save(Xh,      "Affine-DMDc+Hankel (inv from hankel-normalized)",
            os.path.join(run_dir, "XDMDc_hankel_surface.png"), step=PLOT_STEP_3D)
            plot_2d_statewise(X_state, Xrec, state_names, os.path.join(run_dir,
            "2D_DMDc_statewise.png"))
            plot_2d_statewise(X_state, Xh,   state_names, os.path.join(run_dir,
            "2D_HankelDMDc_statewise.png"))
            plot_heatmap(A, xlabels=state_names, ylabels=state_names,
                         title="A (state -> next state)", png_path=os.path.join(run_dir,
                         "A_heatmap.png"))
            plot_heatmap(B, xlabels=ctrl_names, ylabels=state_names,
                         title="B (control -> state impact)", png_path=os.path.join(
                         run_dir, "B_heatmap.png"))
            plot_surfaces_compare_grid(
                [X_state, Xrec, Xh],
                ["X_state real", "Affine-DMDc", "Affine-DMDc+Hankel"],
                os.path.join(run_dir, "COMPARE_3D_DMDc_triplet.png"),
                step=PLOT_STEP_3D, ncols=3
            )

        return dict(
            A=A, B=B, b=b, Ah=Ah, Bh=Bh, bh=bh,
            X_real=X_state, X1_real=X1_state, X2_real=X2_state,
            Xn=Xn, Un=Un, Xrec_n=Xrec_n, Xh_n=Xh_n,
            Xrec=Xrec, Xh=Xh,
            Hx_in=Hx_in, Hu_in=Hu_in, Hx_rec=Hx_rec,
            met_c=met_c, met_h=met_h, score_c=score_c, score_h=score_h
        )

    # ============================================================
    # SMART GRID HELPERS
    # ============================================================

    def _unique_sorted(vals):
        vals = [v for v in vals if v is not None and np.isfinite(v)]
        vals = sorted(set(vals))
        return vals

    def _expand_tik(best_tik, multipliers):
        out = []
        for m in multipliers:
            out.append(best_tik * m)
        out = [float(x) for x in out if x > 0]
        out.append(float(best_tik * 3.0))
        return _unique_sorted(out)

    def _expand_rank(best_rank, neighbors, low=2, high=300):
        out = []
        for d in neighbors:
            out.append(int(best_rank + d))
        out = [x for x in out if low <= x <= high]
        return sorted(set(out))

    def _top_candidates(df, k):
        if df is None or len(df) == 0:
            return []
        d2 = df.copy()
        d2 = d2.replace([np.inf, -np.inf], np.nan).dropna(subset=["score"])
```

```
    d2 = d2.sort_values("score").head(k)
    return d2.to_dict("records")

# ============================================================
# GRID SEARCH: DMD + Hankel
# ============================================================

def grid_search_dmd_hankel(X_real, row_names, stage_name="S1"):
    rows = []
    best = None
    trial = 0
    no_improve = 0
    best_score = np.inf

    for r_dmd, tik, rho_max, clip, d_h, r_h, htik, hrho, hclip in itertools.product(
        GRID["DMD_RANK_GRID"],
        GRID["DMD_TIK_GRID"],
        GRID["DMD_RHO_MAX_GRID"],
        GRID["DMD_CLIP_GRID"],
        GRID["DMD_HANKEL_D_GRID"],
        GRID["DMD_HANKEL_RANK_GRID"],
        GRID["DMD_HANKEL_TIK_GRID"],
        GRID["DMD_HANKEL_RHO_MAX_GRID"],
        GRID["DMD_HANKEL_CLIP_GRID"],
    ):
        trial += 1
        if trial > MAX_TRIALS_DMD:
            print(f"[STOP] DMD Grid reached MAX_TRIALS_DMD={MAX_TRIALS_DMD}")
            break

        run_dir = ensure_dir(os.path.join(
            FIG_ROOT,
            f"GRID_{stage_name}_DMD_r{r_dmd}_tik{tik}_rho{rho_max}_d{d_h}_rH{r_h
            }_htik{htik}_hrho{hrho}"
        ))

        try:
            params = dict(
                ALLOW_PLOTS=PLOTS_DURING_GRID,
                DMD_RANK_MAX=r_dmd,
                DMD_TIK=float(tik),
                DMD_STABLE=True,
                DMD_RHO_MAX=float(rho_max),
                DMD_CLIP=clip,

                DMD_HANKEL_D=int(d_h),
                DMD_HANKEL_RANK_MAX=int(r_h),
                DMD_HANKEL_TIK=float(htik),
                DMD_HANKEL_STABLE=True,
                DMD_HANKEL_RHO_MAX=float(hrho),
                DMD_HANKEL_CLIP=hclip,
            )
            pack = run_dmd_and_hankel(X_real, row_names, params, run_dir)
            score = float(pack["score_h"])

            rows.append(dict(
                score=score,
                DMD_RANK=r_dmd, tik=tik, rho_max=rho_max, clip=clip,
                hankel_d=d_h, hankel_rank=r_h, hankel_tik=htik, hankel_rho=hrho,
                hankel_clip=hclip,
                run_dir=run_dir, error=""
            ))

            if score < best_score:
                best_score = score
                no_improve = 0
                best = rows[-1]
            else:
                no_improve += 1

            if no_improve >= PATIENCE_DMD:
```

```
                print(f"[EARLY STOP] DMD Grid no improvement for PATIENCE_DMD={PATIENCE_DMD} trials.")
                break

        except Exception as e:
            rows.append(dict(
                score=np.inf,
                DMD_RANK=r_dmd, tik=tik, rho_max=rho_max, clip=clip,
                hankel_d=d_h, hankel_rank=r_h, hankel_tik=htik, hankel_rho=hrho, hankel_clip=hclip,
                run_dir=run_dir, error=str(e)
            ))
            no_improve += 1
            if no_improve >= PATIENCE_DMD:
                print(f"[EARLY STOP] DMD Grid (errors/no improvement) hit PATIENCE_DMD={PATIENCE_DMD}.")
                break

    df = pd.DataFrame(rows).sort_values("score").reset_index(drop=True)
    return df, best

def smart_expand_grid_dmd(df_stage1):
    cands = _top_candidates(df_stage1, STAGE2_TOP_CANDIDATES)
    if not cands:
        return None

    ranks = []
    tiks = []
    hankel_ranks = []
    hankel_tiks = []
    for c in cands:
        ranks += _expand_rank(int(c["DMD_RANK"]), STAGE2_NEIGHBOR_RANK, low=2, high=200)
        tiks  += _expand_tik(float(c["tik"]), STAGE2_MULTIPLIERS_TIK)
        hankel_ranks += _expand_rank(int(c["hankel_rank"]), STAGE2_NEIGHBOR_RANK, low=10, high=300)
        hankel_tiks  += _expand_tik(float(c["hankel_tik"]), STAGE2_MULTIPLIERS_TIK)

    GRID2 = dict(
        DMD_RANK_GRID=sorted(set(ranks)),
        DMD_TIK_GRID=_unique_sorted(tiks),
        DMD_RHO_MAX_GRID=[0.995],
        DMD_CLIP_GRID=[None],

        DMD_HANKEL_D_GRID=[16],
        DMD_HANKEL_RANK_GRID=sorted(set(hankel_ranks)),
        DMD_HANKEL_TIK_GRID=_unique_sorted(hankel_tiks),
        DMD_HANKEL_RHO_MAX_GRID=[0.995],
        DMD_HANKEL_CLIP_GRID=[None],
    )
    return GRID2

# ============================================================
# GRID SEARCH: DMDc + Hankel
# ============================================================

def grid_search_dmdc_hankel(X_state, U_ctrl, state_names, ctrl_names, stage_name="S1"):
    rows = []
    best = None
    trial = 0
    no_improve = 0
    best_score = np.inf

    for rO, tik, rho_max, clip, d_h, r_h, htik, hrho, hclip in itertools.product(
        GRID["DMDc_RANK_OMEGA_GRID"],
        GRID["DMDc_TIK_GRID"],
        GRID["DMDc_RHO_MAX_GRID"],
        GRID["DMDc_CLIP_GRID"],
        GRID["DMDc_HANKEL_D_GRID"],
        GRID["DMDc_HANKEL_RANK_GRID"],
```

```
        GRID["DMDc_HANKEL_TIK_GRID"],
        GRID["DMDc_HANKEL_RHO_MAX_GRID"],
        GRID["DMDc_HANKEL_CLIP_GRID"],
    ):
        trial += 1
        if trial > MAX_TRIALS_DMDc:
            print(f"[STOP] DMDc Grid reached MAX_TRIALS_DMDc={MAX_TRIALS_DMDc}")
            break

        run_dir = ensure_dir(os.path.join(
            FIG_ROOT,
            f"GRID_{stage_name}_DMDc_rO{rO}_tik{tik}_rho{rho_max}_d{d_h}_rH{r_h}_htik{htik}_hrho{hrho}"
        ))

        try:
            params = dict(
                ALLOW_PLOTS=PLOTS_DURING_GRID,
                DMDc_RANK_OMEGA=int(rO),
                DMDc_TIK=float(tik),
                DMDc_STABLE=True,
                DMDc_RHO_MAX=float(rho_max),
                DMDc_CLIP=clip,

                DMDc_HANKEL_D=int(d_h),
                DMDc_HANKEL_RANK_MAX=int(r_h),
                DMDc_HANKEL_TIK=float(htik),
                DMDc_HANKEL_STABLE=True,
                DMDc_HANKEL_RHO_MAX=float(hrho),
                DMDc_HANKEL_CLIP=hclip
            )
            pack = run_dmdc_and_hankel(X_state, U_ctrl, state_names, ctrl_names, params, run_dir)
            score = float(pack["score_h"])

            rows.append(dict(
                score=score,
                r_omega=rO, tik=tik, rho_max=rho_max, clip=clip,
                hankel_d=d_h, hankel_rank=r_h, hankel_tik=htik, hankel_rho=hrho,
                hankel_clip=hclip,
                run_dir=run_dir, error=""
            ))

            if score < best_score:
                best_score = score
                no_improve = 0
                best = rows[-1]
            else:
                no_improve += 1

            if no_improve >= PATIENCE_DMDc:
                print(f"[EARLY STOP] DMDc Grid no improvement for PATIENCE_DMDc={PATIENCE_DMDc} trials.")
                break

        except Exception as e:
            rows.append(dict(
                score=np.inf,
                r_omega=rO, tik=tik, rho_max=rho_max, clip=clip,
                hankel_d=d_h, hankel_rank=r_h, hankel_tik=htik, hankel_rho=hrho,
                hankel_clip=hclip,
                run_dir=run_dir, error=str(e)
            ))
            no_improve += 1
            if no_improve >= PATIENCE_DMDc:
                print(f"[EARLY STOP] DMDc Grid (errors/no improvement) hit PATIENCE_DMDc={PATIENCE_DMDc}.")
                break

    df = pd.DataFrame(rows).sort_values("score").reset_index(drop=True)
    return df, best
```

```python

def smart_expand_grid_dmdc(df_stage1):
    cands = _top_candidates(df_stage1, STAGE2_TOP_CANDIDATES)
    if not cands:
        return None

    rO_list = []
    tik_list = []
    hankel_rank_list = []
    hankel_tik_list = []
    for c in cands:
        rO_list += _expand_rank(int(c["r_omega"]), STAGE2_NEIGHBOR_RANK, low=2, high=120)
        tik_list += _expand_tik(float(c["tik"]), STAGE2_MULTIPLIERS_TIK)
        hankel_rank_list += _expand_rank(int(c["hankel_rank"]), STAGE2_NEIGHBOR_RANK, low=10, high=300)
        hankel_tik_list += _expand_tik(float(c["hankel_tik"]), STAGE2_MULTIPLIERS_TIK)

    GRID2 = dict(
        DMDc_RANK_OMEGA_GRID=sorted(set(rO_list)),
        DMDc_TIK_GRID=_unique_sorted(tik_list),
        DMDc_RHO_MAX_GRID=[0.995],
        DMDc_CLIP_GRID=[None],

        DMDc_HANKEL_D_GRID=[16],
        DMDc_HANKEL_RANK_GRID=sorted(set(hankel_rank_list)),
        DMDc_HANKEL_TIK_GRID=_unique_sorted(hankel_tik_list),
        DMDc_HANKEL_RHO_MAX_GRID=[0.995],
        DMDc_HANKEL_CLIP_GRID=[None],
    )
    return GRID2

# ============================================================
# ☑ NEW: ML (LR + RF) comparison for 1-year horizon
#   One-step formulation:
#     input_k = [x_state(k); u(k)]  -> target = x_state(k+1)
# ============================================================

def ml_train_test_one_year(X_state, U_ctrl, state_names, ctrl_names, time_cols, out_dir):
    """
    Compare ML baselines: Random Forest (RF) and Linear Regression (LR).
    - Performance evaluation
    - Visualization (Real vs Pred, error curves)
    """
    ensure_dir(out_dir)

    try:
        from sklearn.linear_model import LinearRegression
        from sklearn.ensemble import RandomForestRegressor
        from sklearn.multioutput import MultiOutputRegressor
    except Exception as e:
        print("[WARN] scikit-learn not available. ML comparison skipped. Error:", e)
        return None

    X_state = np.asarray(X_state, float)
    U_ctrl = np.asarray(U_ctrl, float)
    n, T = X_state.shape
    p, Tu = U_ctrl.shape
    if Tu != T:
        raise ValueError("U_ctrl length must match X_state length for ML compare.")

    # Supervised dataset:
    # features at k: [x(k), u(k)] ; target: x(k+1)
    Xk = X_state[:, :-1].T                # (T-1, n)
    Uk = U_ctrl[:, :-1].T                 # (T-1, p)
    F  = np.hstack([Xk, Uk])              # (T-1, n+p)
    Y  = X_state[:, 1:].T                 # (T-1, n)

    total = F.shape[0]
    if total < 50:
```

```
        print("[WARN] Not enough samples for ML comparison. Skipping.")
        return None

    test_n = ML_TEST_SAMPLES
    if test_n is None:
        test_n = min(YEAR_SAMPLES_6H, total // 3) if total > YEAR_SAMPLES_6H else max(10, total // 4)
    test_n = int(max(10, min(test_n, total - 10)))

    train_n = total - test_n

    F_tr, Y_tr = F[:train_n], Y[:train_n]
    F_te, Y_te = F[train_n:], Y[train_n:]

    # Time labels for test targets correspond to x(k+1)
    time_for_targets = list(time_cols[1:]) if time_cols is not None and len(time_cols) >= T else [f"t{i}" for i in range(1, T)]
    time_te = time_for_targets[train_n:train_n + test_n]

    # Models
    lr = MultiOutputRegressor(LinearRegression())
    rf = MultiOutputRegressor(RandomForestRegressor(
        n_estimators=RF_N_ESTIMATORS,
        random_state=ML_RANDOM_STATE,
        n_jobs=-1,
        max_depth=RF_MAX_DEPTH,
        min_samples_leaf=RF_MIN_SAMPLES_LEAF
    ))

    # Fit
    lr.fit(F_tr, Y_tr)
    rf.fit(F_tr, Y_tr)

    # Predict (one-step)
    Y_lr = lr.predict(F_te)   # (test_n, n)
    Y_rf = rf.predict(F_te)

    # Convert to (n, test_n)
    X_true_year = Y_te.T
    X_lr_year = np.asarray(Y_lr, float).T
    X_rf_year = np.asarray(Y_rf, float).T

    # Metrics
    df_lr, score_lr = metrics_rowwise(X_true_year, X_lr_year, state_names)
    df_rf, score_rf = metrics_rowwise(X_true_year, X_rf_year, state_names)

    # Plots
    if SAVE_PLOTS:
        plot_ml_compare_one_year(time_te, X_true_year, X_lr_year, state_names, "LR", out_dir)
        plot_ml_compare_one_year(time_te, X_true_year, X_rf_year, state_names, "RF", out_dir)

    return dict(
        test_n=test_n,
        train_n=train_n,
        time_te=time_te,
        X_true_year=X_true_year,
        X_lr_year=X_lr_year,
        X_rf_year=X_rf_year,
        met_lr=df_lr, score_lr=score_lr,
        met_rf=df_rf, score_rf=score_rf
    )

# ============================================================
# MAIN
# ============================================================

if __name__ == "__main__":

    ensure_dir(FIG_ROOT)
```

```
1362
1363        print("Excel path:", XLSX_PATH)
1364        print("File exists?", os.path.exists(XLSX_PATH))
1365        print("Preprocess mode:", "PIPELINE_1_9" if USE_PIPELINE_1_9 else "TPYS_FAST")
1366        print("Affine models:", USE_AFFINE_MODELS, "Hankel-affine:", USE_HANKEL_AFFINE,
            "Spike-hybrid:", USE_SPIKE_HYBRID)
1367
1368        # ------------------------------------------------------------
1369        # PREPROCESS SELECTOR
1370        # ------------------------------------------------------------
1371        if DO_CLEAN_NORMALIZE:
1372            if USE_PIPELINE_1_9:
1373                df_use = preprocess_pipeline_1_9(
1374                    xlsx_path=XLSX_PATH,
1375                    sheet_name=TPYS_SHEET,
1376                    out_path=PIPELINE_1_9_OUT_PATH,
1377                    date_col_name=DATE_COL,
1378                    hour_col_name=HOUR_COL,
1379                    value_cols=TPYS_COLS_REQUIRED,
1380                    round_digits=5
1381                )
1382                print("[OK] Pipeline 1..9 saved to:", PIPELINE_1_9_OUT_PATH)
1383
1384            else:
1385                df_wide = build_df_wide_from_tpys(XLSX_PATH, TPYS_SHEET)
1386
1387                # log1p -> MinMax row-wise
1388                Xn_all, _, _ = log1p_minmax_rows(df_wide.values.astype(float))
1389                Xn_all = np.clip(Xn_all, 0.0, 1.0)
1390                df_wide_proc = pd.DataFrame(Xn_all, index=df_wide.index, columns=df_wide.
                    columns)
1391
1392                with pd.ExcelWriter(PROCESSED_WIDE_PATH, engine="openpyxl") as writer:
1393                    df_wide_proc.to_excel(writer, sheet_name=PROCESSED_WIDE_SHEET)
1394
1395                print("[OK] Saved processed wide (log1p->MinMax ROW) to:",
                    PROCESSED_WIDE_PATH)
1396                df_use = df_wide_proc
1397        else:
1398            df_use = build_df_wide_from_tpys(XLSX_PATH, TPYS_SHEET)
1399
1400        # ------------------------------------------------------------
1401        # Build matrices
1402        # ------------------------------------------------------------
1403        X_dmd, dmd_names = pick_rows_from_dfwide(df_use, DMD_ROWS)
1404        X_state, state_names = pick_rows_from_dfwide(df_use, STATE_ROWS)
1405        U_ctrl, ctrl_names = pick_rows_from_dfwide(df_use, CONTROL_ROWS)
1406
1407        print("\n=== Shapes ===")
1408        print("X_dmd  :", X_dmd.shape)
1409        print("X_state:", X_state.shape)
1410        print("U_ctrl :", U_ctrl.shape)
1411        print("state rows:", state_names)
1412        print("ctrl rows :", ctrl_names)
1413
1414        preview_matrix(X_state, "X_state preview")
1415        preview_matrix(U_ctrl,  "U_ctrl preview")
1416
1417        results = {}
1418
1419        # ---- Manual baseline runs (plots ON) ----
1420        dmd_run_dir = ensure_dir(os.path.join(FIG_ROOT, "DMD_MANUAL"))
1421        try:
1422            p = dict(
1423                ALLOW_PLOTS=True,
1424                DMD_RANK_MAX=MANUAL["DMD_RANK_MAX"],
1425                DMD_TIK=MANUAL["DMD_TIK"],
1426                DMD_STABLE=MANUAL["DMD_STABLE"],
1427                DMD_RHO_MAX=MANUAL["DMD_RHO_MAX"],
1428                DMD_CLIP=MANUAL["DMD_CLIP"],
1429                DMD_HANKEL_D=MANUAL["DMD_HANKEL_D"],
```

```
            DMD_HANKEL_RANK_MAX=MANUAL["DMD_HANKEL_RANK_MAX"],
            DMD_HANKEL_TIK=MANUAL["DMD_HANKEL_TIK"],
            DMD_HANKEL_STABLE=MANUAL["DMD_HANKEL_STABLE"],
            DMD_HANKEL_RHO_MAX=MANUAL["DMD_HANKEL_RHO_MAX"],
            DMD_HANKEL_CLIP=MANUAL["DMD_HANKEL_CLIP"],
        )
        results["DMD_manual"] = run_dmd_and_hankel(X_dmd, dmd_names, p, dmd_run_dir)
        print("[OK] DMD manual finished.")
    except Exception as e:
        print("[WARN] DMD manual failed (continuing):", e)
        results["DMD_manual"] = None

    dmdc_run_dir = ensure_dir(os.path.join(FIG_ROOT, "DMDc_MANUAL"))
    try:
        p = dict(
            ALLOW_PLOTS=True,
            DMDc_RANK_OMEGA=MANUAL["DMDc_RANK_OMEGA"],
            DMDc_TIK=MANUAL["DMDc_TIK"],
            DMDc_STABLE=MANUAL["DMDc_STABLE"],
            DMDc_RHO_MAX=MANUAL["DMDc_RHO_MAX"],
            DMDc_CLIP=MANUAL["DMDc_CLIP"],
            DMDc_HANKEL_D=MANUAL["DMDc_HANKEL_D"],
            DMDc_HANKEL_RANK_MAX=MANUAL["DMDc_HANKEL_RANK_MAX"],
            DMDc_HANKEL_TIK=MANUAL["DMDc_HANKEL_TIK"],
            DMDc_HANKEL_STABLE=MANUAL["DMDc_HANKEL_STABLE"],
            DMDc_HANKEL_RHO_MAX=MANUAL["DMDc_HANKEL_RHO_MAX"],
            DMDc_HANKEL_CLIP=MANUAL["DMDc_HANKEL_CLIP"],
        )
        results["DMDc_manual"] = run_dmdc_and_hankel(X_state, U_ctrl, state_names,
        ctrl_names, p, dmdc_run_dir)
        print("[OK] DMDc manual finished.")
    except Exception as e:
        print("[WARN] DMDc manual failed (continuing):", e)
        results["DMDc_manual"] = None

    grid_dmd_df_s1 = None
    grid_dmd_df_s2 = None
    grid_dmdc_df_s1 = None
    grid_dmdc_df_s2 = None

    best_dmd = None
    best_dmdc = None

    # ---- Grid Search ----
    if CALIBRATION_MODE == "auto":
        print("\n>> GRID SEARCH STAGE-1: DMD + Hankel")
        grid_dmd_df_s1, best_dmd = grid_search_dmd_hankel(X_dmd, dmd_names, stage_name
        ="S1")
        print(grid_dmd_df_s1.head(GRID["TOP_K"]))

        print("\n>> GRID SEARCH STAGE-1: DMDc + Hankel")
        grid_dmdc_df_s1, best_dmdc = grid_search_dmdc_hankel(X_state, U_ctrl,
        state_names, ctrl_names, stage_name="S1")
        print(grid_dmdc_df_s1.head(GRID["TOP_K"]))

        if SMART_GRID and (not STAGE1_ONLY):
            GRID2_DMD = smart_expand_grid_dmd(grid_dmd_df_s1)
            if GRID2_DMD is not None:
                print("\n>> SMART EXPAND STAGE-2: DMD + Hankel (around best)")
                GRID_backup = GRID.copy()
                GRID.update(GRID2_DMD)
                grid_dmd_df_s2, best_dmd_s2 = grid_search_dmd_hankel(X_dmd, dmd_names,
                 stage_name="S2")
                print(grid_dmd_df_s2.head(GRID["TOP_K"]))
                if best_dmd is None or (best_dmd_s2 is not None and best_dmd_s2[
                "score"] < best_dmd["score"]):
                    best_dmd = best_dmd_s2
                GRID = GRID_backup

            GRID2_DMDc = smart_expand_grid_dmdc(grid_dmdc_df_s1)
            if GRID2_DMDc is not None:
```

```
                print("\n>> SMART EXPAND STAGE-2: DMDc + Hankel (around best)")
                GRID_backup = GRID.copy()
                GRID.update(GRID2_DMDc)
                grid_dmdc_df_s2, best_dmdc_s2 = grid_search_dmdc_hankel(X_state,
                U_ctrl, state_names, ctrl_names, stage_name="S2")
                print(grid_dmdc_df_s2.head(GRID["TOP_K"]))
                if best_dmdc is None or (best_dmdc_s2 is not None and best_dmdc_s2[
                "score"] < best_dmdc["score"]):
                    best_dmdc = best_dmdc_s2
                GRID = GRID_backup

        if best_dmd is not None:
            print("\n☑ BEST DMD+Hankel:", best_dmd)
        if best_dmdc is not None:
            print("\n☑ BEST DMDc+Hankel:", best_dmdc)

        # ---- Re-run BEST with plots ON ----
        if best_dmdc is not None:
            try:
                print("\n>> Re-run BEST DMDc+Hankel with plots ON")
                p = dict(
                    ALLOW_PLOTS=True,
                    DMDc_RANK_OMEGA=int(best_dmdc["r_omega"]),
                    DMDc_TIK=float(best_dmdc["tik"]),
                    DMDc_STABLE=True,
                    DMDc_RHO_MAX=float(best_dmdc["rho_max"]),
                    DMDc_CLIP=best_dmdc["clip"],
                    DMDc_HANKEL_D=int(best_dmdc["hankel_d"]),
                    DMDc_HANKEL_RANK_MAX=int(best_dmdc["hankel_rank"]),
                    DMDc_HANKEL_TIK=float(best_dmdc["hankel_tik"]),
                    DMDc_HANKEL_STABLE=True,
                    DMDc_HANKEL_RHO_MAX=float(best_dmdc["hankel_rho"]),
                    DMDc_HANKEL_CLIP=best_dmdc["hankel_clip"],
                )
                best_dir = ensure_dir(os.path.join(FIG_ROOT, "BEST_DMDc_REPLOT"))
                results["DMDc_best_replot"] = run_dmdc_and_hankel(X_state, U_ctrl,
                state_names, ctrl_names, p, best_dir)
            except Exception as e:
                print("[WARN] Best DMDc replot failed:", e)

    # ---- Big 3D comparison ----
    try:
        Xlist = [X_state]
        titles = ["X_real (state)"]

        if results.get("DMD_manual") is not None:
            Xlist += [results["DMD_manual"]["Xdmd"], results["DMD_manual"]["Xhan"]]
            titles += ["Affine-DMD", "Affine-DMD+Hankel"]

        if results.get("DMDc_manual") is not None:
            Xlist += [results["DMDc_manual"]["Xrec"], results["DMDc_manual"]["Xh"]]
            titles += ["Affine-DMDc", "Affine-DMDc+Hankel"]

        if results.get("DMDc_best_replot") is not None:
            Xlist += [results["DMDc_best_replot"]["Xh"]]
            titles += ["BEST Affine-DMDc+Hankel"]

        if len(Xlist) >= 2 and SAVE_PLOTS:
            plot_surfaces_compare_grid(
                Xlist, titles,
                os.path.join(FIG_ROOT, "COMPARE_3D_ALL_MODELS.png"),
                step=PLOT_STEP_3D, ncols=3
            )
    except Exception as e:
        print("[WARN] Big compare plot failed:", e)

    # ============================================================
    # ☑ NEW: ML comparison (RF + LR) on 1-year window
    # ============================================================
    ml_pack = None
    if ENABLE_ML_COMPARE:
```

```
1564            try:
1565                ml_dir = ensure_dir(os.path.join(FIG_ROOT, "ML_RF_LR_1YEAR"))
1566                time_cols = list(df_use.columns) if isinstance(df_use, pd.DataFrame) else
                    None
1567                ml_pack = ml_train_test_one_year(X_state, U_ctrl, state_names, ctrl_names,
                     time_cols, ml_dir)
1568                if ml_pack is not None:
1569                    print("[OK] ML comparison finished. LR score:", ml_pack["score_lr"],
                        "RF score:", ml_pack["score_rf"])
1570            except Exception as e:
1571                print("[WARN] ML comparison failed:", e)
1572                ml_pack = None
1573
1574        # ------------------------------------------------------
1575        # EXPORT (metrics + grids + summary + REQUESTED MATRICES)
1576        # ------------------------------------------------------
1577        with pd.ExcelWriter(OUT_XLSX, engine="openpyxl") as writer:
1578
1579            # --- Metrics sheets ---
1580            if results.get("DMD_manual") is not None:
1581                results["DMD_manual"]["met_dmd"].to_excel(writer, sheet_name=
                    "DMD_manual_metrics", index=False)
1582                results["DMD_manual"]["met_h"].to_excel(writer, sheet_name=
                    "HankelDMD_metrics", index=False)
1583
1584            if results.get("DMDc_manual") is not None:
1585                results["DMDc_manual"]["met_c"].to_excel(writer, sheet_name=
                    "DMDc_manual_metrics", index=False)
1586                results["DMDc_manual"]["met_h"].to_excel(writer, sheet_name=
                    "HankelDMDc_metrics", index=False)
1587
1588            if results.get("DMDc_best_replot") is not None:
1589                results["DMDc_best_replot"]["met_h"].to_excel(writer, sheet_name=
                    "BEST_DMDc_H_metrics", index=False)
1590
1591            # --- ML metrics ---
1592            if ml_pack is not None:
1593                ml_pack["met_lr"].to_excel(writer, sheet_name="ML_LR_metrics", index=False
                    )
1594                ml_pack["met_rf"].to_excel(writer, sheet_name="ML_RF_metrics", index=False
                    )
1595
1596            # --- Grid sheets ---
1597            if grid_dmd_df_s1 is not None:
1598                grid_dmd_df_s1.to_excel(writer, sheet_name="GRID_DMD_S1", index=False)
1599            if grid_dmd_df_s2 is not None:
1600                grid_dmd_df_s2.to_excel(writer, sheet_name="GRID_DMD_S2", index=False)
1601
1602            if grid_dmdc_df_s1 is not None:
1603                grid_dmdc_df_s1.to_excel(writer, sheet_name="GRID_DMDc_S1", index=False)
1604            if grid_dmdc_df_s2 is not None:
1605                grid_dmdc_df_s2.to_excel(writer, sheet_name="GRID_DMDc_S2", index=False)
1606
1607            # --- Summary ---
1608            summary = []
1609            if results.get("DMD_manual") is not None:
1610                summary.append(dict(model="Affine_DMD_manual_Hankel", score=results[
                    "DMD_manual"]["score_h"], run_dir=os.path.join(FIG_ROOT, "DMD_MANUAL")))
1611            if results.get("DMDc_manual") is not None:
1612                summary.append(dict(model="Affine_DMDc_manual_Hankel", score=results[
                    "DMDc_manual"]["score_h"], run_dir=os.path.join(FIG_ROOT, "DMDc_MANUAL")))
1613            if best_dmd is not None:
1614                summary.append(dict(model="DMD_grid_best", score=best_dmd["score"],
                    run_dir=best_dmd["run_dir"]))
1615            if best_dmdc is not None:
1616                summary.append(dict(model="DMDc_grid_best", score=best_dmdc["score"],
                    run_dir=best_dmdc["run_dir"]))
1617            if results.get("DMDc_best_replot") is not None:
1618                summary.append(dict(model="DMDc_best_replot", score=results[
                    "DMDc_best_replot"]["score_h"], run_dir=os.path.join(FIG_ROOT,
                    "BEST_DMDc_REPLOT")))
```

```
1619            if ml_pack is not None:
1620                summary.append(dict(model="ML_LR_1year", score=ml_pack["score_lr"],
                    run_dir=os.path.join(FIG_ROOT, "ML_RF_LR_1YEAR")))
1621                summary.append(dict(model="ML_RF_1year", score=ml_pack["score_rf"],
                    run_dir=os.path.join(FIG_ROOT, "ML_RF_LR_1YEAR")))
1622            pd.DataFrame(summary).to_excel(writer, sheet_name="SUMMARY", index=False)
1623
1624            # ------------------------------------------------------
1625            # ☑ REQUESTED: save matrices (each in its own sheet)
1626            # ------------------------------------------------------
1627            col_names = list(df_use.columns) if isinstance(df_use, pd.DataFrame) else None
1628
1629            # -------------------------
1630            # DMD exports
1631            # -------------------------
1632            if results.get("DMD_manual") is not None:
1633                pack = results["DMD_manual"]
1634
1635                df_from_matrix(pack["X_real"], row_names=dmd_names, col_names=col_names).
                    to_excel(writer, sheet_name="DMD_X_real")
1636                df_from_matrix(pack["X1_real"], row_names=dmd_names, col_names=(col_names
                    [:-1] if col_names else None)).to_excel(writer, sheet_name="DMD_X1")
1637                df_from_matrix(pack["X2_real"], row_names=dmd_names, col_names=(col_names[
                    1:] if col_names else None)).to_excel(writer, sheet_name="DMD_X2")
1638
1639                # Normalized
1640                df_from_matrix(pack["Xn"], row_names=dmd_names, col_names=col_names).
                    to_excel(writer, sheet_name="DMD_Xn_0_1")
1641                df_from_matrix(pack["Xdmd_n"], row_names=dmd_names, col_names=col_names).
                    to_excel(writer, sheet_name="DMD_Xdmd_n_0_1")
1642                df_from_matrix(pack["Xhan_n"], row_names=dmd_names, col_names=col_names).
                    to_excel(writer, sheet_name="DMD_Xhan_n_0_1")
1643
1644                # Real-space affine outputs
1645                df_from_matrix(pack["Xdmd"], row_names=dmd_names, col_names=col_names).
                    to_excel(writer, sheet_name="DMD_Xaffine_dmd")
1646                df_from_matrix(pack["Xhan"], row_names=dmd_names, col_names=col_names).
                    to_excel(writer, sheet_name="DMD_Xaffine_dmd_H")
1647
1648                # A, b (and Hankel Ah, bh)
1649                if pack.get("A") is not None:
1650                    df_from_square(pack["A"], names=dmd_names).to_excel(writer, sheet_name
                        ="DMD_A")
1651                if pack.get("b") is not None:
1652                    df_from_vector(pack["b"], row_names=dmd_names, col_name="b").to_excel(
                        writer, sheet_name="DMD_b")
1653                if pack.get("Ah") is not None:
1654                    df_from_matrix(pack["Ah"]).to_excel(writer, sheet_name=safe_sheet_name
                        ("DMD_Ah_hankel"))
1655                if pack.get("bh") is not None:
1656                    df_from_vector(pack["bh"], col_name="bh").to_excel(writer, sheet_name=
                        safe_sheet_name("DMD_bh_hankel"))
1657
1658            # -------------------------
1659            # DMDc exports
1660            # -------------------------
1661            if results.get("DMDc_manual") is not None:
1662                pack = results["DMDc_manual"]
1663
1664                df_from_matrix(pack["X_real"], row_names=state_names, col_names=col_names
                    ).to_excel(writer, sheet_name="DMDc_X_real")
1665                df_from_matrix(pack["X1_real"], row_names=state_names, col_names=(
                    col_names[:-1] if col_names else None)).to_excel(writer, sheet_name=
                    "DMDc_X1")
1666                df_from_matrix(pack["X2_real"], row_names=state_names, col_names=(
                    col_names[1:] if col_names else None)).to_excel(writer, sheet_name=
                    "DMDc_X2")
1667
1668                # Normalized
1669                df_from_matrix(pack["Xn"], row_names=state_names, col_names=col_names).
                    to_excel(writer, sheet_name="DMDc_Xn_0_1")
```

```
1670                df_from_matrix(pack["Un"], row_names=ctrl_names, col_names=col_names).
                    to_excel(writer, sheet_name="DMDc_Un_0_1")
1671                df_from_matrix(pack["Xrec_n"], row_names=state_names, col_names=col_names
                    ).to_excel(writer, sheet_name="DMDc_Xrec_n_0_1")
1672                df_from_matrix(pack["Xh_n"], row_names=state_names, col_names=col_names).
                    to_excel(writer, sheet_name="DMDc_Xh_n_0_1")
1673
1674                # Real-space affine outputs
1675                df_from_matrix(pack["Xrec"], row_names=state_names, col_names=col_names).
                    to_excel(writer, sheet_name="DMDc_Xaffine_dmdc")
1676                df_from_matrix(pack["Xh"], row_names=state_names, col_names=col_names).
                    to_excel(writer, sheet_name="DMDc_Xaffine_dmdc_H")
1677
1678                # A, B, b (and Hankel Ah, Bh, bh)
1679                if pack.get("A") is not None:
1680                    df_from_square(pack["A"], names=state_names).to_excel(writer,
                    sheet_name="DMDc_A")
1681                if pack.get("B") is not None:
1682                    df_from_matrix(pack["B"], row_names=state_names, col_names=ctrl_names
                    ).to_excel(writer, sheet_name="DMDc_B")
1683                if pack.get("b") is not None:
1684                    df_from_vector(pack["b"], row_names=state_names, col_name="b").
                    to_excel(writer, sheet_name="DMDc_b")
1685
1686                if pack.get("Ah") is not None:
1687                    df_from_matrix(pack["Ah"]).to_excel(writer, sheet_name=safe_sheet_name
                    ("DMDc_Ah_hankel"))
1688                if pack.get("Bh") is not None:
1689                    df_from_matrix(pack["Bh"]).to_excel(writer, sheet_name=safe_sheet_name
                    ("DMDc_Bh_hankel"))
1690                if pack.get("bh") is not None:
1691                    df_from_vector(pack["bh"], col_name="bh").to_excel(writer, sheet_name=
                    safe_sheet_name("DMDc_bh_hankel"))
1692
1693            # -------------------------
1694            # ML exports (1-year window)
1695            # -------------------------
1696            if ml_pack is not None:
1697                te_cols = ml_pack["time_te"]
1698                df_from_matrix(ml_pack["X_true_year"], row_names=state_names, col_names=
                te_cols).to_excel(writer, sheet_name="ML_TRUE_1Y")
1699                df_from_matrix(ml_pack["X_lr_year"], row_names=state_names, col_names=
                te_cols).to_excel(writer, sheet_name="ML_LR_PRED_1Y")
1700                df_from_matrix(ml_pack["X_rf_year"], row_names=state_names, col_names=
                te_cols).to_excel(writer, sheet_name="ML_RF_PRED_1Y")
1701
1702                e_lr = rmse_over_time(ml_pack["X_true_year"], ml_pack["X_lr_year"])
1703                e_rf = rmse_over_time(ml_pack["X_true_year"], ml_pack["X_rf_year"])
1704                pd.DataFrame({"time": te_cols, "RMSE_over_states": e_lr}).to_excel(writer,
                     sheet_name="ML_LR_ERR_1Y", index=False)
1705                pd.DataFrame({"time": te_cols, "RMSE_over_states": e_rf}).to_excel(writer,
                     sheet_name="ML_RF_ERR_1Y", index=False)
1706
1707        print("\n[OK] Saved:", OUT_XLSX)
1708        print("[OK] Figures in:", FIG_ROOT)
1709
1710        if best_dmdc is not None:
1711            print("\nBest DMDc folder:", best_dmdc["run_dir"])
1712            try:
1713                os.startfile(best_dmdc["run_dir"])
1714            except Exception:
1715                pass
1716
1717    # ============================================================
1718    # 🔧 CALIBRATION / TUNING GUIDE (NUMBERED)
1719    # ============================================================
1720    # [P01] DATA_DIR:         Change your workspace folder path.
1721    # [P02] XLSX_NAME:        Change your input Excel filename.
1722    # [P03] TPYS_SHEET:       Change the TPYS sheet name.
1723    # [P04]-[P05] Preprocessing selector: only ONE must be True.
1724    # [P06] DO_CLEAN_NORMALIZE: Enable/disable preprocessing.
```

```
# [P07]-[P11] Output filenames and folders.
# [P12] CALIBRATION_MODE: "manual" (no grid) vs "auto" (grid search).
# [P13] SAVE_PLOTS: Save figures to disk.
# [P14] PLOT_STEP_3D: Controls surface plot density (speed vs detail).
# [P15] USE_AFFINE_MODELS: Enable affine DMD/DMDc (recommended).
# [P16] USE_HANKEL_AFFINE: Enable Hankel-affine (recommended).
# [P17]-[P19] SPIKE_*: Spike blending calibration:
#       - Higher SPIKE_Q => fewer spikes detected
#       - Higher SPIKE_BLEND => predictions closer to real at spikes
# [P20]-[P24] MAX_TRIALS_* and PATIENCE_* control grid depth/speed.
# [P25]-[P29] SMART_GRID stage-2 expansion around best candidates.
# [P30]-[P32] TPYS column names: update if Excel column names change.
# [P33]-[P35] State/control row definitions.
# [P36]-[P46] Manual DMD + Hankel parameters.
# [P47]-[P57] Manual DMDc + Hankel parameters.
# [P58]-[P76] GRID: candidate lists for automatic search.
# [P77] round_digits: rounding for minmax in pipeline 1..9.
# [P78] target_rows: which variables are kept in pipeline 1..9.
# [P79] max_points: downsampling for 2D plots.
# [P80] ncols: number of columns in 3D comparison grid.
# [P81] np.clip(Xn,0,1): extra numeric guard to keep normalized range.
#
# ☑ ML parameters:
# [P82] ENABLE_ML_COMPARE: enable RF/LR comparison
# [P83] YEAR_SAMPLES_6H: "one-year" window length for 6h data
# [P84] ML_TEST_SAMPLES: override fixed test length if desired
# [P85]-[P88] RF hyperparameters
# [P89] PLOT_ML_MAX_POINTS: max points in ML plots
#
# ☑ NOTE (HANKEL NONNEGATIVE REQUIREMENT):
# - Hankel is applied ONLY on normalized matrices (Xn/Un)
# - We then clip to [0,1] before saving Xhan_n and Xh_n
# ============================================================
# ☑ TOTAL TUNABLE PARAMETERS COUNT:
#   Previous: 81  => Now: 89
# ============================================================
```