PharmaSUG 2026 - Paper DV- 229

Python for Survival Analysis: Kaplan-Meier and Reverse KM Plots Made Easy

Girish Kankipati, Pfizer Inc.

Rajesh Jakka, Pfizer Inc.

ABSTRACT

Survival analysis plays a key role in clinical trials by providing insights into time-to-event outcomes such as overall survival and progression-free survival. Researchers most widely use the Kaplan–Meier (KM) estimator to estimate and visualize survival probabilities over time. This paper presents a practical approach for generating KM plots using Python, a programming language that clinical data analysts increasingly adopt. The paper discusses techniques for plotting KM curves for single and multiple treatment groups, adding confidence intervals, and customizing charts for regulatory reporting. Using Python libraries such as Matplotlib and Plotly, analysts can create both static and interactive visualizations that support reproducible and automated workflows.

The paper also highlights the Reverse Kaplan–Meier (RKM) method, which estimates follow-up time by treating censoring as an event. Clinical trials widely use RKM to report median follow-up, and it complements standard KM analysis. Practical examples demonstrate how to implement RKM plots in Python, calculate median follow-up, and present results in a clear and compliant format. All examples use reproducible Python code and simulate clinical trial data, making them easy to adapt for real-world applications. By leveraging Python for KM and RKM visualization, researchers gain flexibility and automation while producing high-quality outputs that meet industry standards and support better decision-making in clinical research.

INTRODUCTION

Clinical trials frequently analyze endpoints for which the event time may be unknown at the data cutoff, requiring methods that properly accommodate right censoring. Kaplan–Meier (KM) curves provide an intuitive representation of time-to-event distributions and serve as a standard component of clinical study reports and interim analyses. While SAS remains the established tool for survival analysis and regulatory reporting, analysts increasingly leverage Python for exploratory analytics, automation, data integration, and visualization. Python's reproducible figure-generation capabilities help ensure consistent results across evolving data cuts and multiple review cycles.

Clinical trial reports also commonly include median follow-up to assess data maturity. The Reverse Kaplan–Meier method estimates follow-up duration by reversing the event indicator and provides a reliable measure that avoids the biases associated with simple descriptive summaries.

This paper outlines a complete Python-based workflow, from data simulation through KM and Reverse KM estimation to visualization, designed to support transparent, reproducible, and auditable survival analyses in clinical trial settings.

METHODOLOGY

KAPLAN–MEIER ESTIMATION

The Kaplan–Meier (KM) estimator calculates survival probability at each event time by evaluating the number of patients who remain at risk. At each event time, the method updates the survival estimate by multiplying the previous value by the proportion of at-risk patients who do not experience the event. Analysts refer to this approach as the product-limit formula. The estimator handles censored observations

naturally: censored patients contribute to the risk set up to their last known time, after which the method removes them without altering the estimated survival probability.

Analysts quantify uncertainty in the KM estimate using Greenwood's variance. To construct confidence intervals, they apply a log(−log) transformation rather than a simple linear approach. This transformation produces more stable and reliable confidence bounds, particularly when survival probabilities approach 0 or 1.

REVERSE KAPLAN–MEIER (FOLLOW-UP)

The Reverse Kaplan–Meier (RKM) method uses the same computational framework as the standard Kaplan–Meier estimator but addresses a different question: how long investigators followed patients. To apply the method, analysts reverse the event indicator by treating patients who experienced the event as censored and patients who were originally censored as having an event. They then apply the standard Kaplan–Meier estimation procedure to this reversed data set

The resulting curve represents the probability that patients remain under follow-up beyond each time point. Analysts define the median follow-up as the time at which this curve declines to 0.5.

Why RKM Is Preferred

RKM avoids bias from early events, pulling down simple summary measures of followup. It is followed due to the reasons below.

- properly accounts for censoring.
- is consistent with the KM framework used for survival endpoints.
- is recognized by regulatory agencies.
- works for pooled and arm-specific summaries.

Interpreting the Reverse KM Plot

The Reverse Kaplan–Meier (RKM) plot displays the proportion of patients who remain under follow-up over time. As follow-up time increases, the curve decreases in a stepwise manner, and the point at which it crosses 0.5 defines the median follow-up. Analysts calculate confidence intervals using Greenwood's variance, following the same approach applied in standard Kaplan–Meier estimation.

When analysts stratify RKM curves by treatment arm, the plots provide a quick assessment of follow-up balance across groups. Analysts should note substantial differences in follow-up duration between treatment arms, as such imbalances may affect the interpretation of between-arm survival comparisons.

IMPLEMENTATION IN PYTHON

The implementation reuses the same KM estimation function for both standard and reverse KM analysis. Only the event indicator changes, and everything else stays the same. This keeps the code concise and ensures the two analyses are methodologically consistent with each other.

The workflow in five steps:

1. Start from the same time-to-event data set used for standard KM
2. Reverse the event indicator (1 → 0, 0 → 1)
3. Apply the KM estimator to the reversed indicators

4. Plot the RKM curve with confidence intervals
5. Extract median follow-up from the curve

Data is simulated for a two-arm study (Treatment vs. Control) with independent censoring, making all examples fully reproducible and shareable. The modular code structure supports version control, unit testing, and dual-programming validation against SAS when required.

PYTHON LIBRARIES & FUNCTIONS

NumPy (numpy)

- Vectorized numerical operations for survival updates and variance terms.
- Random number generation for synthetic event/censoring times during simulation.

Pandas (pandas)

- Tabular data handling (DataFrame), grouping by treatment arm, and clean input/output columns (time_months, event, arm).
- Convenient for mapping from ADTTE-like data sets to the estimator.

Matplotlib (matplotlib.pyplot)

- KM/RKM step plots and confidence interval bands (publication quality).
- Flexible styling for labels, grid, legends, and export to PNG/PDF (e.g., dpi=300 for submission-quality figures).

Matplotlib Ticker (matplotlib.ticker.MaxNLocator)

- Forces integer ticks on time axes (e.g., months: 0, 6, 12…) to improve readability and alignment with TLF conventions.

Functions:

simulate_survival(n, hazard_event, hazard_censor, arm_label)

- Generates synthetic two-arm data for testing.
- Simulates exponential event and censoring times; sets time = min(event, censor) and event = 1 if event occurs first, else 0.

km_estimator(times, events)

- Computes KM survival function, Greenwood variance, and pointwise 95% CIs via log(-log) transform.

km_by_arm(df, time_col="time_months", event_col="event", arm_col="arm")

- Applies km_estimator to each treatment arm and returns a dictionary of KM result DataFrames.

rkm_by_arm(df, time_col="time_months", event_col="event", arm_col="arm")

- Computes Reverse KM (RKM) by reversing the event indicator (rev_events = 1 - event) and reusing km_estimator.

rkm_median_follow-up(rkm_df)

- Returns the median follow-up as the first time where RKM survival ≤ 0.5.

step_surv_at(km_df, t)

- Finds the KM step value at a censoring time t(for plotting censor ticks on the curve).

```
============================================================================
=
# SURVIVAL ANALYSIS: Kaplan-Meier (KM) and Reverse Kaplan-Meier (RKM) Plots
#
============================================================================
=
# BEFORE RUNNING:
#   Open terminal in VS Code (Ctrl + `) and run:
#   pip install matplotlib numpy pandas
#
============================================================================
=

#
============================================================================
=
# STEP 1: IMPORT LIBRARIES
#
============================================================================
=
# numpy  - numerical calculations (arrays, math)
# pandas - data handling (DataFrames, grouping)
# matplotlib.pyplot - plotting KM and RKM charts
# MaxNLocator - forces integer tick marks on x-axis (0, 6, 12...)

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from matplotlib.ticker import MaxNLocator

print("=" * 60)
print("STEP 1: Libraries imported successfully.")
print(f"  numpy   version : {np.__version__}")
print(f"  pandas  version : {pd.__version__}")
import matplotlib
print(f"  matplotlib version : {matplotlib.__version__}")
print("=" * 60)

#
============================================================================
=
# STEP 2: SET RANDOM SEED#
============================================================================
=
# Fixed seed = same simulated data every run (reproducibility).
```

```
# Change the number to get different random data.

np.random.seed(1234)

print("\nSTEP 2: Random seed set to 1234.")
print("  (Same data will be generated every time you run this script)")

#
==============================================================================
=
# STEP 3: DEFINE DATA SIMULATION FUNCTION#
==============================================================================
=
# Generates synthetic clinical trial data for one treatment arm.
#
# Parameters:
#   n             - number of patients
#   hazard_event  - rate of events (e.g. death) per month
#   hazard_censor - rate of censoring (lost to follow-up) per month
#   arm_label     - arm name ("Treatment" or "Control")
#
# Logic:
#   - Event time and censor time both drawn from exponential distributions
#   - Observed time = min(event time, censor time)
#   - event = 1 if event happened first, else 0 (censored)

def simulate_survival(n, hazard_event, hazard_censor, arm_label):
    t_event = np.random.exponential(scale=1/hazard_event, size=n)   # time
to event
    t_cens  = np.random.exponential(scale=1/hazard_censor, size=n)  # time
to censoring
    time    = np.minimum(t_event, t_cens)                           #
observed time
    event   = (t_event <= t_cens).astype(int)                       #
1=event, 0=censored
    return pd.DataFrame({"time": time, "event": event, "arm": arm_label})

print("\nSTEP 3: simulate_survival() function defined.")

#
==============================================================================
=
# STEP 4: SIMULATE TWO-ARM CLINICAL TRIAL DATA#
==============================================================================
=
# Treatment: lower event hazard (0.08) = better survival
# Control:   higher event hazard (0.12) = worse survival
# Both arms: same censoring rate (0.05)

n_per_arm = 250

haz = {
    "Treatment": {"event": 0.08, "censor": 0.05},
    "Control":   {"event": 0.12, "censor": 0.05},
}

df = pd.concat([
```

```
    simulate_survival(n_per_arm, haz["Treatment"]["event"],
haz["Treatment"]["censor"], "Treatment"),
    simulate_survival(n_per_arm,
haz["Control"]["event"],   haz["Control"]["censor"],   "Control"),
], ignore_index=True)

df["time_months"] = df["time"]   # time is already in months

print("\nSTEP 4: Trial data simulated.")
print(f"  Total patients : {len(df)}")
print(f"  Events         : {df['event'].sum()}")
print(f"  Censored       : {(df['event'] == 0).sum()}")
print(f"  Time range     : {df['time_months'].min():.2f} -
{df['time_months'].max():.2f} months")
print("\n  Per-arm summary:")
print(df.groupby("arm")[["time_months", "event"]].agg(
    Patients=("event", "count"),
    Events=("event", "sum"),
    Censored=("event", lambda x: (x == 0).sum()),
    Median_Time=("time_months", "median")
).to_string())

#
============================================================================
=
# STEP 5: DEFINE KM ESTIMATOR FUNCTION#
============================================================================
=
# Computes KM survival curve for a single group.
# Also calculates:
#   - Greenwood's variance (uncertainty in survival estimate)
#   - 95% CI via log(-log) transform (stable near boundaries 0 and 1)
#
# Returns DataFrame: time, at_risk, events, surv, var, lower, upper

def km_estimator(times, events):
    times  = np.array(times)
    events = np.array(events)

    # Sort by observed time
    ord_idx = np.argsort(times)
    times   = times[ord_idx]
    events  = events[ord_idx]

    # Only process times where actual events occurred
    uniq_times = np.unique(times[events == 1])

    surv    = 1.0   # starts at 100% survival
    var_sum = 0.0
    eps     = 1e-9  # prevents log(0) or division by zero

    at_risk, d, s, var_greenwood = [], [], [], []

    for t in uniq_times:
        n_i = int(np.sum(times >= t))                       # patients still
at risk
```

```
        d_i = int(np.sum((times == t) & (events == 1)))     # events at this
time

        if n_i > 0:
            surv *= (1 - d_i / n_i)   # KM product-limit step

        s.append(surv)
        at_risk.append(n_i)
        d.append(d_i)

        # Greenwood variance
        # max(n_i - d_i, 1) avoids division by zero when all patients have
event
        denom = n_i * max(n_i - d_i, 1)
        var_sum += d_i / (denom + eps)
        var_greenwood.append((surv ** 2) * var_sum)

    km_df = pd.DataFrame({
        "time":    uniq_times,
        "at_risk": at_risk,
        "events":  d,
        "surv":    s,
        "var":     var_greenwood
    })

    # 95% CI via log(-log) transform
    z = 1.96
    km_df["surv_clip"] = km_df["surv"].clip(eps, 1 - eps)
    se     = np.sqrt(np.clip(km_df["var"].values, 0, None))
    lll    = np.log(-np.log(km_df["surv_clip"].values))
    se_lll = se / (km_df["surv_clip"].values *
np.abs(np.log(km_df["surv_clip"].values)))

    km_df["lower"] = np.exp(-np.exp(lll + z * se_lll))   # lower CI bound
    km_df["upper"] = np.exp(-np.exp(lll - z * se_lll))   # upper CI bound

    return km_df

print("\nSTEP 5: km_estimator() function defined.")

#
============================================================================
=
# STEP 6: RUN KM ESTIMATION PER ARM#
============================================================================
=
# km_by_arm() applies km_estimator to each treatment arm separately.
# Returns a dict: { "Treatment": km_df, "Control": km_df }

def km_by_arm(df, time_col="time_months", event_col="event", arm_col="arm"):
    return {
        arm: km_estimator(dfa[time_col], dfa[event_col])
        for arm, dfa in df.groupby(arm_col)
    }

km_results = km_by_arm(df)
```

```
print("\nSTEP 6: KM estimation complete for all arms.")
for arm, km_df in km_results.items():
    median_surv = km_df.loc[km_df["surv"] <= 0.5, "time"].values
    med_str = f"{median_surv[0]:.1f} months" if len(median_surv) else "not
reached"
    print(f"  {arm}: {len(km_df)} event time points | Median survival =
{med_str}")
    print(f"    Survival range: {km_df['surv'].min():.3f} -
{km_df['surv'].max():.3f}")
    print(f"    CI range (lower): {km_df['lower'].min():.3f} -
{km_df['lower'].max():.3f}")
    print(f"    CI range (upper): {km_df['upper'].min():.3f} -
{km_df['upper'].max():.3f}")

#
=============================================================================
=
# STEP 7: DEFINE CENSOR TICK MARK HELPER#
=============================================================================
=
# Censored patients are shown as "|" marks on the KM curve.
# This helper finds the y-position (survival value) at censoring time t
# so the tick mark sits correctly on the step curve.

def step_surv_at(km_df, t):
    mask = km_df["time"] <= t
    return 1.0 if not np.any(mask) else km_df.loc[mask, "surv"].iloc[-1]

print("\nSTEP 7: step_surv_at() helper function defined.")
# Quick check: survival at time=0 should be 1.0
test_val = step_surv_at(km_results["Treatment"], 0)
print(f"  Check: survival at time=0 for Treatment = {test_val} (expected
1.0)")

#
=============================================================================
=
# STEP 8: DEFINE AND RUN REVERSE KM (RKM)#
=============================================================================
=
# RKM estimates FOLLOW-UP TIME distribution (not event time).
#
# Key idea — swap the event indicator:
#   Actual events  (1) → treated as censored (0)
#   Censored       (0) → treated as events   (1)
#
# The RKM curve shows how long patients stayed under observation.
# Median follow-up = first time RKM survival crosses 0.5.

def rkm_by_arm(df, time_col="time_months", event_col="event",
arm_col="arm"):
    rkm = {}
    for arm, dfa in df.groupby(arm_col):
        rev_events = 1 - np.array(dfa[event_col])   # swap: 1->0, 0->1
        rkm[arm]   = km_estimator(dfa[time_col], rev_events)
    return rkm
```

```
# FIX: renamed from rkm_median_follow-up (hyphen is invalid Python syntax)
def rkm_median_follow_up(rkm_df):
    """Returns median follow-up: first time RKM survival drops to <= 0.5"""
    times = rkm_df["time"].values
    survs = rkm_df["surv"].values
    idx   = np.where(survs <= 0.5)[0]
    return times[idx[0]] if len(idx) else np.nan

rkm_results      = rkm_by_arm(df)
median_fu_by_arm = {arm: rkm_median_follow_up(rdf) for arm, rdf in
rkm_results.items()}

print("\nSTEP 8: RKM estimation complete.")
for arm, rdf in rkm_results.items():
    med = median_fu_by_arm[arm]
    print(f"  {arm}: {len(rdf)} time points | Median follow-up = {med:.1f}
months")
    print(f"    RKM survival range: {rdf['surv'].min():.3f} -
{rdf['surv'].max():.3f}")

#
============================================================================
=
# STEP 9: DEFINE RISK TABLE HELPER#
============================================================================
=
# Risk table shows number of patients still being followed at each
# landmark time. Placed below the KM plot for regulatory submissions.

def risk_table_counts(arm_times, landmarks):
    """Count patients still at risk (observed time >= landmark) at each
landmark."""
    return [int(np.sum(arm_times >= lm)) for lm in landmarks]

max_time  = df["time_months"].max()
landmarks = np.arange(0, max_time, max_time / 6).round().astype(int)

print("\nSTEP 9: Risk table helper defined.")
print(f"  Landmark time points: {list(landmarks)}")
for arm in ["Treatment", "Control"]:
    arm_times = df.loc[df["arm"] == arm, "time_months"].values
    counts    = risk_table_counts(arm_times, landmarks)
    print(f"  {arm} at-risk counts: {counts}")

#

============================================================================
=
# STEP 10: PLOT KM CURVES (CI BANDS + CENSOR TICKS + RISK TABLE)#
============================================================================
=
# Two-panel figure:
#   Top    - KM step curves per arm with 95% CI bands and censor tick marks
#   Bottom - Risk table with patient counts at each landmark time

colors = {"Treatment": "#2ca02c", "Control": "#d62728"}
```

```
fig_km, (ax_km, ax_risk) = plt.subplots(
    2, 1,
    figsize=(9, 7),
    gridspec_kw={"height_ratios": [4, 0.5], "hspace": 0.06},  # compact risk
table
    sharex=True                                # shared x-axis
)

# --- KM step curves + CI bands ---
for arm, km_df in km_results.items():
    # Add t=0, S=1 so curve starts from top-left
    times = np.concatenate([[0], km_df["time"].values])
    survs = np.concatenate([[1], km_df["surv"].values])
    lower = np.concatenate([[1], km_df["lower"].values])
    upper = np.concatenate([[1], km_df["upper"].values])
    c = colors.get(arm, "#1f77b4")

    ax_km.step(times, survs, where="post", label=f"{arm}", color=c,
linewidth=2)
    ax_km.fill_between(times, lower, upper, step="post", color=c,
alpha=0.14,
                       label=f"{arm} 95% CI")

# --- Censor tick marks "|" on the KM curve ---
for arm, dfa in df.groupby("arm"):
    c      = colors.get(arm, "#1f77b4")
    cens_t = dfa.loc[dfa["event"] == 0, "time_months"].values
    yvals  = [step_surv_at(km_results[arm], t) for t in cens_t]
    ax_km.plot(cens_t, yvals, linestyle="None", marker="|",
               markersize=9, color=c, alpha=0.85)

# --- Format KM panel ---
ax_km.set_ylabel("Survival probability")
ax_km.set_title("Kaplan-Meier Curves (Synthetic)")
ax_km.set_ylim(0, 1.05)
ax_km.grid(alpha=0.3, linestyle="--")
ax_km.legend(loc="best", frameon=True, fontsize=9)

# --- Risk table panel ---
# Use fixed y positions: 0.65 for top row, 0.25 for bottom row (close
together)
arm_list    = list(colors.keys())   # ["Treatment", "Control"]
row_y       = [0.72, 0.28]          # y positions for each row (tight
spacing)

ax_risk.set_xlim(ax_km.get_xlim())
ax_risk.set_ylim(0, 1)
ax_risk.set_yticks(row_y)
ax_risk.set_yticklabels(arm_list, fontsize=9)
ax_risk.set_xlabel("Time (months)")
ax_risk.text(0, 1.05, "Number at Risk", transform=ax_risk.transAxes,
             fontsize=9, fontweight="bold", va="bottom")
ax_risk.grid(False)
ax_risk.spines[["top", "right", "left", "bottom"]].set_visible(False)
ax_risk.tick_params(left=False, bottom=False)
ax_risk.set_xticklabels([])
```

```
for (arm, c), ry in zip(colors.items(), row_y):
    arm_times = df.loc[df["arm"] == arm, "time_months"].values
    counts    = risk_table_counts(arm_times, landmarks)
    for lm, cnt in zip(landmarks, counts):
        ax_risk.text(lm, ry, str(cnt), ha="center", va="center",
                     fontsize=8, color=c, fontweight="bold")

plt.tight_layout()
plt.savefig("km_simulated.png", dpi=300, bbox_inches="tight")
plt.show()
plt.close(fig_km)

print("\nSTEP 10: KM plot complete.")
print("  File saved : km_simulated.png")
print("  Contents   : KM curves for Treatment and Control arms")
print("  Includes   : 95% CI bands, censor tick marks, risk table")

#
=============================================================================
=
# STEP 11: PLOT REVERSE KM CURVES (CI BANDS + MEDIAN FOLLOW-UP LINES)#
=============================================================================
=
# RKM plot shows follow-up distribution per arm.
# Horizontal dashed line at 0.5 + vertical dotted lines = median follow-up.

fig_rkm, ax_rkm = plt.subplots(figsize=(9, 6))

# --- RKM step curves + CI bands ---
for arm, rkm_df in rkm_results.items():
    times = np.concatenate([[0], rkm_df["time"].values])
    survs = np.concatenate([[1], rkm_df["surv"].values])
    lower = np.concatenate([[1], rkm_df["lower"].values])
    upper = np.concatenate([[1], rkm_df["upper"].values])
    c = colors.get(arm, "#1f77b4")

    ax_rkm.step(times, survs, where="post", label=f"{arm}", color=c,
linewidth=2)
    ax_rkm.fill_between(times, lower, upper, step="post", color=c,
alpha=0.14,
                        label=f"{arm} 95% CI")

# --- Median follow-up reference lines ---
ax_rkm.axhline(0.5, color="#7f7f7f", linestyle="--", linewidth=1.2,
               label="50% threshold")

for arm, med in median_fu_by_arm.items():
    if np.isfinite(med):
        c = colors.get(arm, "#1f77b4")
        ax_rkm.axvline(med, color=c, linestyle=":", linewidth=1.5)
        ax_rkm.text(med, 0.52, f"Median FU ({arm}): {med:.1f} m",
                    rotation=90, va="bottom", ha="center", fontsize=9,
color=c)

# --- Format RKM panel ---
ax_rkm.set_xlabel("Follow-up time (months)")
ax_rkm.set_ylabel("RKM survival (follow-up)")
```

```
ax_rkm.set_title("Reverse Kaplan-Meier (Follow-up) (Synthetic)")
ax_rkm.set_ylim(0, 1.05)
ax_rkm.xaxis.set_major_locator(MaxNLocator(integer=True))
ax_rkm.grid(alpha=0.3, linestyle="--")
ax_rkm.legend(loc="best", frameon=True, fontsize=9)

plt.tight_layout()
plt.savefig("rkm_simulated.png", dpi=300, bbox_inches="tight")
plt.show()
plt.close(fig_rkm)

print("\nSTEP 11: RKM plot complete.")
print("  File saved : rkm_simulated.png")
print("  Contents   : RKM curves for Treatment and Control arms")
print("  Includes   : 95% CI bands, median follow-up annotations")
for arm, med in median_fu_by_arm.items():
    print(f"  Median follow-up ({arm}) : {med:.1f} months")

#
============================================================================
=
# ALL STEPS COMPLETE
#
============================================================================
=
print("\n" + "=" * 60)
print("ALL STEPS COMPLETE!")
print("  km_simulated.png  - KM survival curves")
print("  rkm_simulated.png - Reverse KM follow-up curves")
print("  Both files saved in the same folder as this script.")
print("=" * 60)
```

Program 1 Code for KM and Reverse KM

STEP-BY-STEP DESCRIPTION OF THE CODE:

For the above Program 1 below are the important steps to consider

Importing Required Libraries

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
```

- numpy is used for array and mathematical operations.
- matplotlib.pyplot is used for plotting the graph.
- mpl_toolkits.mplot3d enables the creation of 3D plots.

Create random data

```
np.random.seed(1234)
```

Random data generated when we run the script every time by using the numpy package.

Clinical trial data

```
def simulate_survival(n, hazard_event, hazard_censor, arm_label):
    t_event = np.random.exponential(scale=1/hazard_event, size=n)
 # time to event

    t_cens  = np.random.exponential(scale=1/hazard_censor, size=n)
 # time to censoring
```

The code above simulates clinical trial data with parameters consistent with timetoevent analyses. In the survival simulation, np.random.exponential is used to generate waiting times until an event occurs, representing either the failure time or the censoring time. Both the event times and censoring times are drawn from exponential distributions.

- N= Number of patients
- hazard_event - rate of events (e.g. death) per month
- hazard_censor - rate of censoring (lost to follow-up) per month
- arm_label - arm name ("Treatment" or "Control")

Generating and Merging

```
n_per_arm = 250

haz = {
    "Treatment": {"event": 0.08, "censor": 0.05},
    "Control":   {"event": 0.12, "censor": 0.05},
}

df = pd.concat([
    simulate_survival(n_per_arm, haz["Treatment"]["event"],
haz["Treatment"]["censor"], "Treatment"),
    simulate_survival(n_per_arm,
haz["Control"]["event"],   haz["Control"]["censor"],   "Control"),
], ignore_index=True)
```

The code runs the simulate_survival function twice, once for each treatment arm, and then stacks the results into a single dataframe with 500 total rows (250 per arm).

- **Treatment Group (hazard = 0.08):** A lower hazard indicates a lower event risk, meaning patients in this group are expected to survive longer or maintain their eventfree- status longer.
- **Control Group (hazard = 0.12):** A higher hazard reflects a higher event risk, so this group is expected to experience the event sooner.
- **Censoring Rate (0.05):** Both groups share the same censoring rate, representing loss to followup. Using a common censoring rate helps maintain- fairness in comparing treatment effects.

95% CI and log (-log) transformation (between 0 and 1)

```
#   - 95% CI via log(-log) transform (stable near boundaries 0 and 1)
```

```
# Returns DataFrame: time, at_risk, events, surv, var, lower, upper
def km_estimator(times, events):
    times  = np.array(times)
    events = np.array(events)
    uniq_times = np.unique(times[events == 1])

    # 95% CI via log(-log) transform
    se_lll = se / (km_df["surv_clip"].values *
np.abs(np.log(km_df["surv_clip"].values)))

    km_df["lower"] = np.exp(-np.exp(lll + z * se_lll))   # lower CI bound
    km_df["upper"] = np.exp(-np.exp(lll - z * se_lll))   # upper CI bound
```

The above code uses the following fundamental functions from NumPy to perform the calculation.

np.array: Python code can perform fast calculations across the entire data set at once. Standard Python lists can't handle complex calculations easily.

np.unique: Only process times where actual events occurred.

np.exp(-np.exp(...)): This function is a double exponential operation reverses the log-log transformation to bring the values back to the original probability scale (between 0 and 1).

- The Problem: Standard +/- margins of error can result in a survival probability below 0% or above 100%, which is impossible.
- The Fix: This math transformation warps the confidence intervals, so they are mathematically forced to stay between 0 and 1, making the results realistic for a medical trial.

KM Estimation per Arm:

```
# km_by_arm() applies km_estimator to each treatment arm separately.
# Returns a dict: { "Treatment": km_df, "Control": km_df }
def km_by_arm(df, time_col="time_months", event_col="event", arm_col="arm"):
   return {
      arm: km_estimator(dfa[time_col], dfa[event_col])
      for arm, dfa in df.groupby(arm_col)    }

km_results = km_by_arm(df)

   median_surv = km_df.loc[km_df["surv"] <= 0.5, "time"].values
   med_str = f"{median_surv[0]:.1f} months" if len(median_surv) else "not reached"
```

- The km_by_arm Function takes trial table (df) and splits it by the "arm" column. It runs the km_estimator calculation (from Step 5) on the treatment group and the Control group separately. It stores these results in a dictionary called km_results. km_df.loc[km_df["surv"] <= 0.5, "time"] finds the first time point where survival probability drops to 50% or lower. If the survival curve crosses 50%, it takes the first time point where that happened (median_surv[0]) and formats it to one decimal place. If the curve never drops to 0.5 (meaning more than half the group is still alive at the end of the study), it sets the value to "not reached".

Censor Tick Mark Helper

```
def step_surv_at(km_df, t):
    mask = km_df["time"] <= t
    return 1.0 if not np.any(mask) else km_df.loc[mask, "surv"].iloc[-1]
```

mask = km_df["time"] <= t: This code looks for all recorded time points in your data that happened at or before the time t we are interested in.

if not np.any(mask): If the time t is earlier than your very first data point (like time 0), it assumes survival is still 1.0 (100%).

km_df.loc[mask, "surv"].iloc[-1]: If there is data, it finds the most recent survival probability recorded before or at time t.

For example, if an event happened at month 5 (dropping survival to 80%) and a patient was censored at month 7, this function returns 0.8 so the tick mark sits on that 80% line. The helper finds the y position at censoring time t so the tick mark populates on the step curve.

Reverse KM (RKM)

```
def rkm_by_arm(df, time_col="time_months", event_col="event",
arm_col="arm"):
    rkm = {}
    for arm, dfa in df.groupby(arm_col):
        rev_events = 1 - np.array(dfa[event_col])   # swap: 1->0, 0->1

def rkm_median_follow_up(rkm_df):
    """Returns median follow-up: first time RKM survival drops to <= 0.5"""
    times = rkm_df["time"].values
    survs = rkm_df["surv"].values
    idx   = np.where(survs <= 0.5)[0]
    return times[idx[0]] if len(idx) else np.nan
```

- rkm_by_arm is a custom function used in survival analysis to estimate the follow-up time distribution for different groups (arms) in a data set
- The Swap Logic :1 - np.array(dfa[event_col]: By calculating 1 - event, this is effectively telling the Kaplan-Meier algorithm: Ignore the actual deaths/events to see how long it takes for someone to be censored. Returns median follow-up: first time RKM survival drops to less than equal to 0.5

```
# --- Censor tick marks "|" on the KM curve ---
for arm, dfa in df.groupby("arm"):
    c      = colors.get(arm, "#1f77b4")
    cens_t = dfa.loc[dfa["event"] == 0, "time_months"].values
    yvals  = [step_surv_at(km_results[arm], t) for t in cens_t]
    ax_km.plot(cens_t, yvals, linestyle="None", marker="|",
               markersize=9, color=c, alpha=0.85)

# --- Format KM panel ---
ax_km.set_ylabel("Survival probability")
```

```
ax_km.set_title("Kaplan–Meier Curves (Synthetic)")
ax_km.set_ylim(0, 1.05)
ax_km.grid(alpha=0.3, linestyle="--")
ax_km.legend(loc="best", frameon=True, fontsize=9)

# --- Risk table panel ---
# Use fixed y positions: 0.65 for top row, 0.25 for bottom row (close
together)
arm_list    = list(colors.keys())    # ["Treatment", "Control"]
row_y       = [0.72, 0.28]           # y positions for each row (tight
spacing)

ax_risk.set_xlim(ax_km.get_xlim())
ax_risk.set_ylim(0, 1)
ax_risk.set_yticks(row_y)
ax_risk.set_yticklabels(arm_list, fontsize=9)
ax_risk.set_xlabel("Time (months)")
ax_risk.text(0, 1.05, "Number at Risk", transform=ax_risk.transAxes,
             fontsize=9, fontweight="bold", va="bottom")
ax_risk.grid(False)
ax_risk.spines[["top", "right", "left", "bottom"]].set_visible(False)
ax_risk.tick_params(left=False, bottom=False)
ax_risk.set_xticklabels([])

for (arm, c), ry in zip(colors.items(), row_y):
    arm_times = df.loc[df["arm"] == arm, "time_months"].values
    counts    = risk_table_counts(arm_times, landmarks)
    for lm, cnt in zip(landmarks, counts):
        ax_risk.text(lm, ry, str(cnt), ha="center", va="center",
                     fontsize=8, color=c, fontweight="bold")
```

Censor tick marks on KM curve:

This block processes the data arm-by-arm, pulls each arm’s censored times (event == 0), evaluates the KM estimate at those times

- marker="|": It plots a vertical tick mark at every time point where a patient was censored.
- step_surv_at: It calculates exactly where on the Y-axis (the survival curve) that tick mark should sit.

Formatting the KM Panel:

This section formats the Kaplan–Meier plot by setting the Y-axis label and plot title, constraining the survival probability scale to 0–1 (with slight padding to 1.05), and adding a light dashed grid to improve readability. The legend is placed automatically in the best location (loc="best"), displayed with a frame, and sized for clarity, making group labels easy to interpret without cluttering the figure.

Risk table panel:

Layout & Alignment

- Synchronizing Axes: ax_risk.set_xlim(ax_km.get_xlim()) is the most important line, it ensures the time points in the table line up perfectly with the time points on the curve above it.

- Hardcoded Y-Positions: It uses row_y = [0.72, 0.28] to stack the rows. Since the Y-axis is set from 0 to 1, the "Treatment" row will sit in the top half and "Control" in the bottom half.
- Invisible Graph: It strips away all the standard chart features (spines, grid, ticks, and xticklabels) so that only the text is visible, making it look like a table rather than a plot.

Data Injection (The Loop)

- Filtering: It grabs the data for each group one by one (df.loc[df["arm"] == arm]).
- Calculation: It calls a helper function risk_table_counts (likely defined elsewhere in your script) to calculate how many patients are still "at risk" at each specific landmark (time point).
- Text Placement: It loops through those results and uses ax_risk.text(lm, ry, str(cnt)...) to "stamp" the count onto the plot at the correct time (lm) and row height (ry).

Layout & Structure

```
fig_km, (ax_km, ax_risk) = plt.subplots(
    2, 1,
    figsize=(9, 7),
    gridspec_kw={"height_ratios": [4, 0.5], "hspace": 0.06},  # compact
risk table
    sharex=True                                # shared x-axis
)
```

plt.subplots(2, 1, ...): Creates a figure with 2 rows and 1 column.

gridspec_kw={"height_ratios": [4, 0.5]}: Sets the "Top" plot to be 8 times larger than the "Bottom" table.

sharex=True: Forces both panels to use the exact same horizontal (Time) scale so they line up.

Data Preparation

```
for arm, km_df in km_results.items():
      times = np.concatenate([[0], km_df["time"].values])
    c = colors.get(arm, "#1f77b4")
```

- np.concatenate(): Joins two arrays. Here, it adds 0 to the start of the time data so the graph doesn't start in the middle of the page.
- zip(colors.items(), row_y): Pairs up your labels/colors with their vertical positions to loop through them efficiently.

Cleanup & Export

```
row_y       = [0.72, 0.28]          # y positions for each row (tight
spacing)

ax_risk.set_xlim(ax_km.get_xlim())
ax_risk.set_xlabel("Time (months)")
ax_risk.spines[["top", "right", "left", "bottom"]].set_visible(False)
ax_risk.tick_params(left=False, bottom=False)
```

- set_xlim(ax_km.get_xlim()): This step copies the exact width of the top graph so that the numbers in your table line up perfectly under the correct time points.
- row_y = [0.72, 0.28]: Since this is still technically a graph, it uses these invisible "heights" to place the Treatment numbers on one level and the Control numbers on another.
- spines[...].set_visible(False): This deletes the "box" (top, bottom, left, and right lines). It turns a boxy graph into a clean, floating table.
- tick_params(left=False, bottom=False): This removes the tiny little "ticks" next to the numbers to keep it looking like text, not data points

RESULTS

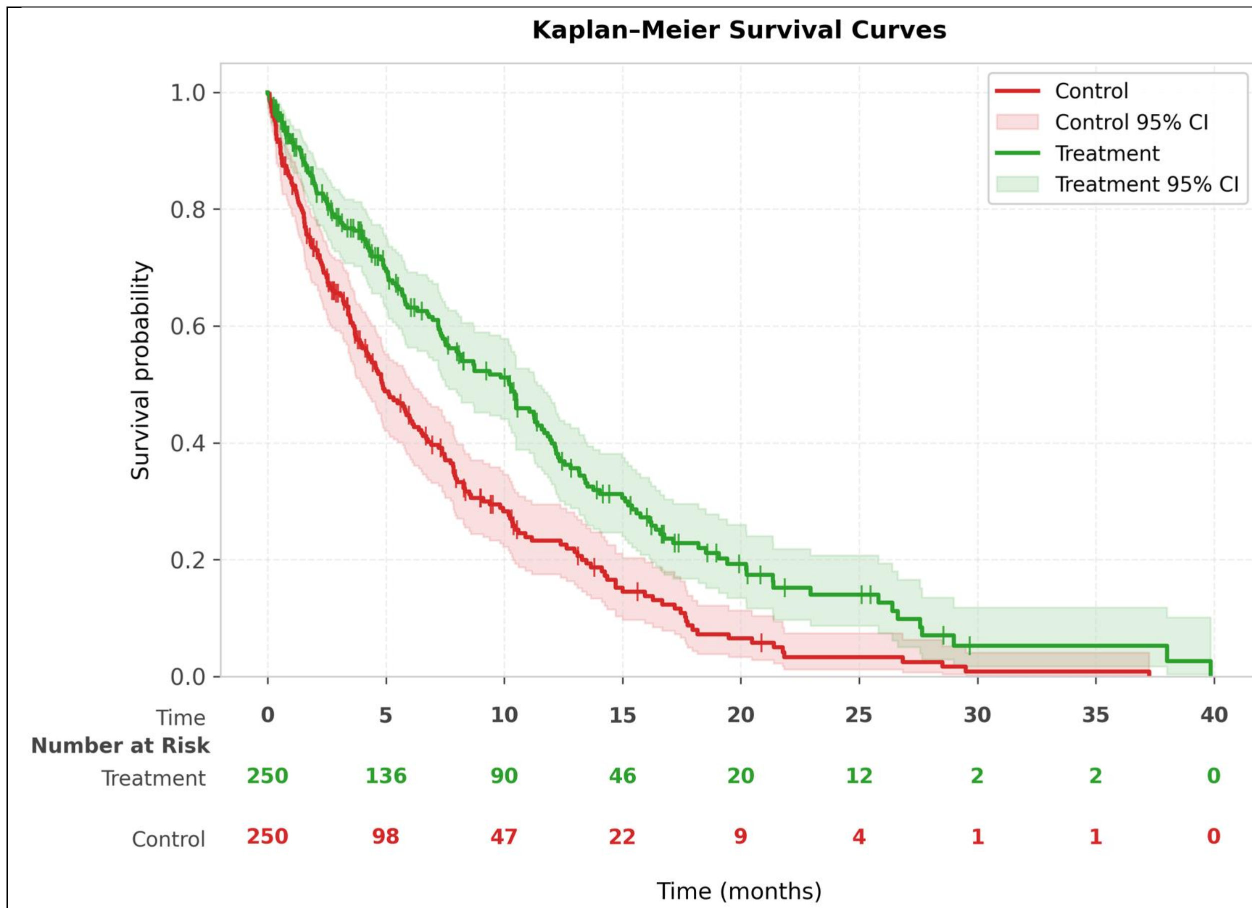

Figure 1. KM SURVIVAL BY ARM (95% CI)

In the above Figure 1 the KM plot displays survival probabilities for both treatment arms over time, with confidence intervals and censor marks. The Treatment arm (lower hazard) shows better survival than the Control arm. The risk table below the curve provides counts of patients still under observation at predefined time points, supporting regulatory reporting standards.

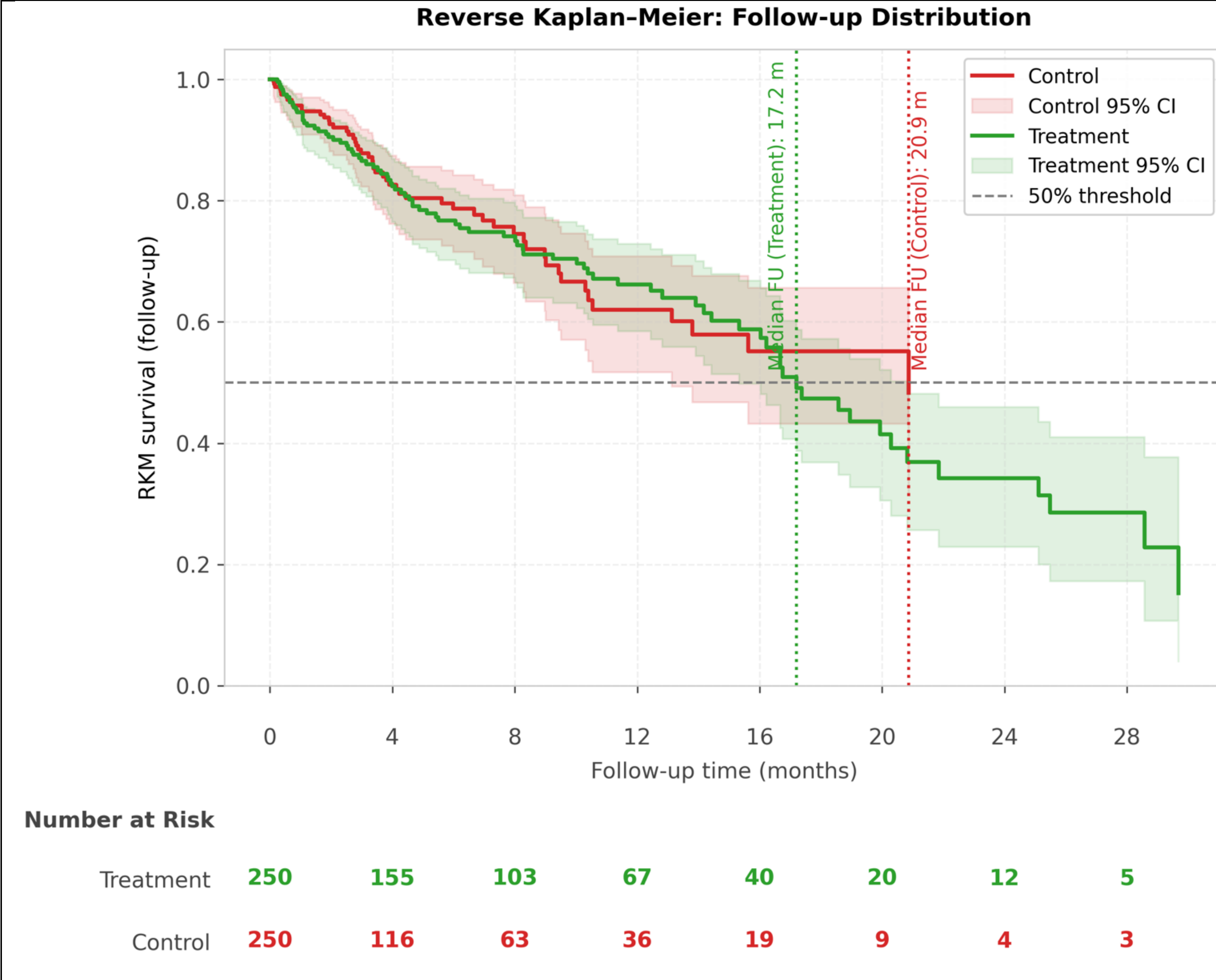

Figure 2. RKM Follow-up Distribution with Median.

From the simulated data set (Treatment hazard 0.08/month; Control 0.12/month; censoring 0.05/month; administrative cutoff 36 months), the workflow produced typical differences in median survival and a pooled RKM estimate of median follow-up.

In the above Figure 2 the RKM plot shows follow-up distribution for all subjects. The median followup is indicated by vertical reference lines for each arm. This curve helps assess whether followup is adequate and whether both arms have comparable observation time, which is important for interpreting survival comparisons.

Clinical Grade Formatting Standards

- Axes and scales: y-axis fixed to [0, 1], x-axis in months.
- Confidence intervals: loglog–based bands using Greenwood's variance.
- Censoring indicators: "+" glyphs at survival level $S(t-)$when censoring occurs.
- Risk table: placed beneath KM plot, with evenly spaced landmarks meaningful to review teams.

- Export: PNG/SVG at ≥300 DPI; consistent filenames (km_plot.png, rkm_plot.png).

Reproducibility and Validation

- Deterministic seeds ensure reproducible results across runs.
- Modular design supports reuse and unit testing.
- QC recommendation: dual program, subset in SAS PROC LIFETEST and verify survival steps, medians, CI behavior, censor marks, and risk counts.
- Documentation: record Python/library versions and capture logs for figure generation.

Python complements traditional tools such as SAS in survival analysis by offering advanced customization, repeatable and automated pipeline development, and interactive visualization capabilities that enhance exploratory data review. This flexibility enables analysts to apply precise color schemes, annotations, and layout configurations that are often more cumbersome to implement in SAS. As a result, Python improves both the interpretability and presentation quality of Kaplan–Meier and Reverse Kaplan–Meier plots in clinical trials.

As implemented here, the Reverse Kaplan–Meier (RKM) method provides a statistically principled assessment of follow-up maturity in clinical trials and effectively avoids the biases associated with simplistic summary statistics, such as understating event-free duration or overlooking censoring patterns. By inverting the traditional Kaplan–Meier estimator, RKM offers deeper insight into data completeness and patient retention, which are critical for informed decision-making in drug development. The proposed code and formatting approach delivers both efficiency and seamless integration into audit-ready clinical reporting workflows, ensuring compliance with regulatory requirements (for example, FDA guidance on graphical representation) while promoting efficiency and reducing manual errors in statistical deliverables.

CONCLUSION

Python reliably generates Kaplan–Meier and Reverse Kaplan–Meier figures that meet regulatory expectations, including confidence intervals, censoring marks, and risk tables. This approach supports reproducible and easily automated workflows and applies directly to real clinical trial data sets. The Reverse Kaplan–Meier method provides an accurate assessment of follow up maturity and avoids the biases associated with simpler summary measures. Overall, Python enables consistent, clear, and audit-friendly survival visualizations within clinical research workflows.

REFERENCES

- Kaplan EL, Meier P. Nonparametric estimation from incomplete observations. *JASA*. 1958;53(282):457–481.
- Schemper M, Smith TL. A note on quantifying follow-up in studies of failure time. *Controlled Clinical Trials*. 1996;17(4):343–346.
- Collett D. *Modelling Survival Data in Medical Research*. 3rd ed. CRC Press; 2015.