# correlated\_monte\_carlo\_fx.py

import numpy as np

import pandas as pd

import yfinance as yf

import matplotlib.pyplot as plt

import datetime as dt

# -------------------------

# User parameters (changeable)

# -------------------------

tickers = ["EURUSD=X", "GBPUSD=X", "USDJPY=X"]

end = dt.datetime.now()

start = end - dt.timedelta(days=365\*3) # 3 years history for better stats

n\_assets = len(tickers)

T = 1 # years to simulate

N = 252 # time steps (trading days)

dt = T / N

n\_sims = 10000 # Monte Carlo paths

plot\_simulations = 20 # number of sample sims to store & plot per asset

weights = np.ones(n\_assets) / n\_assets # equal-weight portfolio

# -------------------------

# 1) Download data and prepare returns

# -------------------------

data = yf.download(tickers, start=start, end=end)["Close"] # DataFrame: datetime x tickers

# Ensure proper shape if single-column returns

if isinstance(data, pd.Series):

data = data.to\_frame()

# Align and drop NaNs (if any)

data = data.dropna(how="any")

# Use last price as S0 for each asset

S0 = data.iloc[-1].values # array length = n\_assets

# Compute log returns (daily)

log\_returns = np.log(data / data.shift(1)).dropna()

# Estimate annualized drift vector and covariance matrix

mu\_daily = log\_returns.mean().values # daily mean for each asset

sigma\_daily = log\_returns.std().values # daily std (not used directly)

mu = mu\_daily \* 252 # annualized drift (mu vector)

cov\_daily = log\_returns.cov().values # daily covariance matrix (n\_assets x n\_assets)

cov = cov\_daily \* 252 # annualized covariance matrix

# Print estimated params

print("S0:", dict(zip(tickers, S0.round(6))))

print("Annualized mu (drift):", dict(zip(tickers, (mu).round(4))))

print("Annualized vol (sqrt diag(cov)):", dict(zip(tickers, np.sqrt(np.diag(cov)).round(4))))

print("Covariance matrix (annualized):\n", pd.DataFrame(cov, index=tickers, columns=tickers))

# -------------------------

# 2) Cholesky decomposition for correlated normals

# -------------------------

# Ensure covariance is positive definite for Cholesky (small numerical fix if needed)

eps = 1e-10

cov\_pd = cov.copy()

# Add small jitter to diagonal if needed

try:

L = np.linalg.cholesky(cov\_pd)

except np.linalg.LinAlgError:

cov\_pd += np.eye(n\_assets) \* eps

L = np.linalg.cholesky(cov\_pd)

# -------------------------

# 3) Memory-friendly Monte Carlo

# - We keep `current\_prices` (n\_assets x n\_sims)

# - We store only `final\_prices` (n\_assets x n\_sims)

# - We store sample paths for first `plot\_simulations` sims for plotting

# -------------------------

# Initialize current prices: each column = a simulation, each row = asset

current\_prices = np.repeat(S0.reshape(-1, 1), n\_sims, axis=1) # shape (n\_assets, n\_sims)

# Storage for a few sample paths to plot

sample\_paths = np.zeros((N+1, n\_assets, plot\_simulations))

sample\_paths[0, :, :] = np.repeat(S0.reshape(-1, 1), plot\_simulations, axis=1)

# Simulate forward

for t in range(1, N+1):

# 1) draw independent normals: shape (n\_assets, n\_sims)

Z\_indep = np.random.normal(size=(n\_assets, n\_sims))

# 2) correlate them: Z\_corr = L @ Z\_indep -> shape (n\_assets, n\_sims)

Z\_corr = L @ Z\_indep

# 3) compute drift and shock terms (use vectorized forms)

drift = (mu.reshape(-1, 1) - 0.5 \* np.diag(cov).reshape(-1, 1)) \* dt # (n\_assets,1)

shock = np.sqrt(dt) \* Z\_corr # (n\_assets, n\_sims) -- scaled independent draws are now correlated (L applied)

# Note: since cov = L L^T, the correct correlated shock term is sigma \* sqrt(dt) \* Z\_correlated.

# We already embedded sigma structure in L, so we use Z\_corr and will multiply by 1 when inside np.exp:

# However we need to ensure units: L had units of covariance sqrt; to keep formula consistent,

# build shock as: (sigma\_element) \* sqrt(dt) \* z\_individual\_correlated.

# A clearer approach: generate standard normals, then multiply by L to produce covariances,

# and use shock = Z\_corr \* 1 with drift computed via mu and cov diag. The exponential will use:

# exp(drift + shock), where shock has correct magnitude through L.

# To make units explicit, normalize Z\_corr rows by sqrt(var)?? Simpler approach below:

# Simpler & correct approach:

# Compute daily-standard-deviation matrix via sqrt of cov diagonal:

sigma\_annual = np.sqrt(np.diag(cov)) # (n\_assets,)

sigma\_daily\_vec = sigma\_annual / np.sqrt(252) # back to daily vol (just for clarity)

# Build shock = (sigma\_daily\_vec \* sqrt(dt)) \* Z\_uncorrelated\_through\_corr

# But we already created Z\_corr with cov structure scaling; to avoid double-scaling, use independent normals

# and apply L\_normalized so that Z\_corr has unit variances. So instead of complexity, we do this:

# Use Cholesky of correlation matrix, not covariance. Let's rebuild using correlation for clarity.

break

# -------------------------

# 3b) Rebuild using correlation (cleaner) — restart memory-friendly simulation

# -------------------------

# Build correlation matrix from cov

std\_annual = np.sqrt(np.diag(cov)) # annual std per asset

corr = cov / np.outer(std\_annual, std\_annual) # correlation matrix

# Cholesky of correlation

L\_corr = np.linalg.cholesky(corr)

# Daily vol vector (annual -> daily)

sigma\_daily = std\_annual / np.sqrt(252) # daily std dev per asset

mu\_daily = mu / 252 # daily drift per asset (annual -> daily)

# Initialize for simulation

current\_prices = np.repeat(S0.reshape(-1, 1), n\_sims, axis=1)

sample\_paths = np.zeros((N+1, n\_assets, plot\_simulations))

sample\_paths[0, :, :] = np.repeat(S0.reshape(-1, 1), plot\_simulations, axis=1)

for t in range(1, N+1):

# independent normals (n\_assets x n\_sims)

Z\_indep = np.random.normal(size=(n\_assets, n\_sims))

# correlate across assets (L\_corr @ Z\_indep) gives correlated standard normals with unit var

Z\_corr\_unit = L\_corr @ Z\_indep # (n\_assets, n\_sims)

# For each asset, compute daily shock = sigma\_daily[asset] \* sqrt(dt) \* Z\_corr\_unit[asset, :]

shock = (sigma\_daily.reshape(-1, 1) \* np.sqrt(dt)) \* Z\_corr\_unit # (n\_assets, n\_sims)

# drift per day: (mu\_daily - 0.5 \* sigma\_daily^2) \* dt

drift = (mu\_daily.reshape(-1, 1) - 0.5 \* (sigma\_daily.reshape(-1, 1) \*\* 2)) \* dt

# Update prices multiplicatively

current\_prices = current\_prices \* np.exp(drift + shock) # (n\_assets, n\_sims)

# Save sample paths

if plot\_simulations > 0:

sample\_paths[t, :, :] = current\_prices[:, :plot\_simulations]

# After loop, current\_prices are final prices per asset (n\_assets x n\_sims)

final\_prices\_per\_asset = current\_prices.copy()

# -------------------------

# 4) Portfolio final values & risk metrics

# -------------------------

# Portfolio final price/value per sim = sum(weights \* final\_asset\_prices)

weights = weights / np.sum(weights) # ensure normalized

# Use starting portfolio value = sum(weights \* S0) to interpret returns

start\_portfolio\_value = np.dot(weights, S0)

final\_portfolio\_values = weights @ final\_prices\_per\_asset # shape (n\_sims,)

# Compute returns

portfolio\_returns = (final\_portfolio\_values / start\_portfolio\_value) - 1.0

# Risk metrics

VaR\_95\_price = np.percentile(final\_portfolio\_values, 5)

VaR\_95\_return = np.percentile(portfolio\_returns, 5)

ES\_95 = final\_portfolio\_values[final\_portfolio\_values <= VaR\_95\_price].mean()

ES\_95\_return = (ES\_95 / start\_portfolio\_value) - 1.0

print("\n--- PORTFOLIO RESULTS ---")

print(f"Start portfolio value: {start\_portfolio\_value:.6f}")

print(f"Expected portfolio return (mean): {np.mean(portfolio\_returns)\*100:.2f}%")

print(f"Annualized portfolio volatility (approx): {np.std(portfolio\_returns)\*100:.2f}%")

print(f"5% VaR (price): {VaR\_95\_price:.6f}")

print(f"5% VaR (return): {VaR\_95\_return\*100:.2f}%")

print(f"5% Expected Shortfall (price): {ES\_95:.6f}")

print(f"5% Expected Shortfall (return): {ES\_95\_return\*100:.2f}%")

print(f"Probability of profit: {np.mean(portfolio\_returns>0)\*100:.2f}%")

# -------------------------

# 5) Plots: sample asset paths and portfolio final distribution

# -------------------------

fig, axes = plt.subplots(n\_assets, 1, figsize=(10, 3\*n\_assets), sharex=True)

if n\_assets == 1:

axes = [axes]

for ai in range(n\_assets):

for s in range(plot\_simulations):

axes[ai].plot(np.linspace(0, T, N+1), sample\_paths[:, ai, s], alpha=0.6)

axes[ai].set\_title(f"{tickers[ai]} sample paths (first {plot\_simulations} sims)")

axes[ai].set\_ylabel("Price")

axes[-1].set\_xlabel("Years")

plt.tight\_layout()

plt.show()

plt.figure(figsize=(8,5))

plt.hist(final\_portfolio\_values, bins=60, alpha=0.7)

plt.title("Histogram of final portfolio values after 1 year")

plt.xlabel("Portfolio value")

plt.ylabel("Frequency")

plt.show()