

1A: We used Pandas and SciPy to analyze the statistical properties of a dataset from "problem1.csv". First, we loaded the dataset using `pd.read_csv()` and extracted the column "X", which contains the data we want to analyze (This will also be used for the rest of assignment). We then computed key summary statistics: mean (`data.mean()`) to measure the central tendency, variance (`data.var()`) to measure data spread, skewness (`skew(data)`) to check for asymmetry, and kurtosis (`kurtosis(data)`) to determine the shape of the distribution. Finally, we printed these values to summarize the dataset's statistical properties.

1B: For choosing one of the distribution, it depends on whether the distribution has a big population. If the sample size is bigger than 30, I will choose normal distribution, and if the sample size is smaller than 30, I will choose T-distribution. It also depends on whether I know the population standard deviation. If it is known, there is high chance that I choose normal distribution. Still, if the data is known to be skewed or has extreme outliers, I would choose T-distribution over normal distribution since a dataset with more symmetricity percularities is more fit for normal distribution. Normal distribution would be forced to be used somehow under the condition that normality is assumed like linear regression. Under this situation, there is a slight skewness, so I decide to choose T-distribution.

1C: We used SciPy to fit both a T-distribution and a normal distribution to the dataset and performed K-S tests to evaluate how well they match the data. First, we used `t.fit(data)` to estimate the degrees of freedom (df), location (loc), and scale parameters for the T-distribution and printed the fitted values. Then, we used `norm.fit(data)` to estimate the mean ( $\mu$ ) and standard deviation ( $\sigma$ ) for the Normal distribution. After fitting both distributions, we performed K-S tests (`kstest()`) to compare the empirical data with the theoretical distributions, using the fitted parameters as arguments. The K-S test results indicate how well each distribution fits the data.

Based on the test result provided above, both T-distribution and normal distribution has high p-values with 0.99624 for Normal distribution and 0.99645 for T-distribution. For practical application, both could be implemented with confidence. T-distribution has a slightly lower statistic data then normal distribution, so T-distribution is actually a little bit better than normal distribution under this dataset. Therefore, my choice matches with data.

2A: Used Pandas to compute the pairwise covariance matrix of a dataset stored in problem2. The `data2.cov()` function calculates the covariance between all numerical columns in the dataset, measuring how they vary together. A positive covariance indicates that two variables tend to increase or decrease together, while a negative covariance

suggests an inverse relationship. Finally, we printed the covariance matrix to examine the relationships between different variables in the dataset.

2B: we checked if the covariance matrix is positive semi-definite (PSD) by computing its eigenvalues using NumPy. A matrix is PSD if all its eigenvalues are non-negative. First, we used `np.linalg.eigvals(matrix)` to find the eigenvalues of the covariance matrix. Then, we checked whether all values were greater than or equal to zero using `np.all(eigenvalues >= 0)`. Finally, we printed the result to confirm whether the matrix is PSD and displayed the eigenvalues for further analysis. If any eigenvalue is negative, the matrix is not PSD, meaning it may need adjustments before being used in certain financial or statistical models. Under this circumstance, the matrix is not PSD because some of the eigenvalues are negative.

2C: We find the nearest PSD matrix using two different methods: Higham's method and the Rebonato & Jackel near-PSD method. In Higham's method, we iteratively adjusted the matrix by replacing negative eigenvalues with zero while preserving its structure. In Rebonato & Jackel's method, we simply set all negative eigenvalues to a small positive value ( $1e-6$ ) to force the matrix to be PSD. After applying these fixes, we printed the adjusted matrices to compare them with the original.

2D: We calculated the covariance matrix using only overlapping data from the dataset. We used `df.dropna().cov()`, which first removes all rows with missing values (NaN) and then computes the covariance matrix on the remaining data. This ensures that only fully available data points contribute to the covariance calculation. Finally, we printed the resulting matrix to compare it with the original covariance matrix

2E: We compared the covariance matrices from Part C (nearest positive semi-definite (PSD) matrices using Higham's method and Rebonato & Jackel's method) with the covariance matrix from Part D (computed using only overlapping data). To measure how much each matrix differs from the original, we used the Frobenius norm, which calculates the overall difference between two matrices. We applied this norm to compare the original covariance matrix with the Higham PSD matrix, Rebonato & Jackel PSD matrix, and the overlapping data covariance matrix. Finally, we printed the computed differences, helping us understand which method preserves the original structure best.

3A: To fit a multivariate normal distribution to the data, I first loaded the dataset and calculated its mean vector and covariance matrix. The mean vector gives the average values of  $X_1$  and  $X_2$ , which act as the center of our normal distribution. The covariance matrix tells us how these two variables vary together, how strongly they are related and in what direction. By estimating these parameters from the dataset, we essentially define the

multivariate normal distribution that best represents the given data. These values are critical because they allow us to compute conditional probabilities and simulate realistic data points later.

3B: The implemented Python code determines the conditional distribution of  $X_2$  given  $X_1 = 0.6$  by employing two distinct methodologies: the analytical method and the simulation-based approach. The analytical method utilizes fundamental properties of the multivariate normal distribution to derive explicit expressions for the conditional mean and variance. Specifically, it applies the formula for the conditional expectation and variance, which are computed using the estimated mean vector and covariance matrix of the dataset. In contrast, the simulation-based approach generates a large number of samples from the fitted multivariate normal distribution via Cholesky decomposition, a technique that factorizes the covariance matrix into a lower triangular matrix to transform independent standard normal variables into correlated samples. From these simulated samples, observations where  $X_1$  is approximately equal to 0.6 are extracted, and the empirical mean and variance of  $X_2$  are calculated to approximate its conditional distribution. By implementing both approaches, the code provides a rigorous validation of the theoretical results while demonstrating the utility of simulation techniques in statistical inference.

3C: In the code, I used NumPy for numerical computations and Cholesky decomposition to find the conditional mean and variance of  $X_2|X_1=0.6$ . First, I computed the Cholesky root of the covariance matrix using `np.linalg.cholesky()`, which gives a lower triangular matrix  $L$ . Then, I extracted  $L_{11}$ ,  $L_{21}$ ,  $L_{22}$  to express the conditional mean using  $\mu_2 + (L_{21}/L_{11})(x_1 - \mu_1)$ , and the conditional variance as  $L_{22}^2$ . The results show that the conditional mean and variance match those derived using the standard formula, proving that the Cholesky root approach correctly captures the conditional distribution of  $X_2|X_1=0.6$ . This also proves that our simulation approach is correct because both methods lead to the same expected distribution.

4A: For simulating MA(1), MA(2), and MA(3) processes, I first generated random moving average (MA) coefficients between -0.8 and 0.8 to ensure realistic behavior. Using these coefficients, I created MA processes with the `ArmaProcess` function from `statsmodels`, specifying an autoregressive (AR) part of 1 (to keep it purely MA) and the generated MA parameters. I then simulated 200 data points for each process. To visualize the differences, I plotted three things: 1. the time series itself to observe how the process behaves, 2. the ACF to see how past values influence future ones, and 3. PACF to check the direct impact of specific lags. This helped illustrate how increasing the MA order affects the dependency structure in the data.

From the simulations, we can see that for an MA( $q$ ) process, the ACF cuts off after  $q$  lags, meaning it has significant correlations only up to lag  $q$  before dropping off. The PACF, on the other hand, shows a more gradual decay instead of a sharp cutoff. For example, MA(1) has a strong ACF at lag 1 and then drops, while MA(2) and MA(3) extend this pattern to lags 2 and 3, respectively. This confirms the key characteristic of MA processes—ACF is the best way to identify their order, while PACF doesn't provide a clear cutoff like in AR processes.

4B: For simulating AR(1), AR(2), and AR(3) processes, I started by generating random autoregressive (AR) coefficients between -0.8 and 0.8 to make the models behave realistically. Using the ArmaProcess function from statsmodels, I defined each AR process with its respective order while keeping the moving average (MA) part fixed at 1 (so there's no MA component). Then, I simulated 200 data points for each process. To understand how different AR orders affect the time series structure, I plotted three things: (1) the raw time series to see how the data behaves over time, (2) the autocorrelation function (ACF) to check how strongly past values influence future values over different lags, and (3) the partial autocorrelation function (PACF) to pinpoint the specific lags that have a direct impact on the current value. This helped me visualize how increasing the AR order changes the correlation patterns in the data.

I noticed that the ACF of AR processes decays gradually instead of cutting off, which shows that past values keep influencing future ones over many lags. Meanwhile, the PACF cuts off after the order of the AR process (e.g., AR(1) cuts off after lag 1, AR(2) after lag 2), making it useful for identifying the model order. As the AR order increases, the ACF takes longer to decay, meaning more past values are incorporated into the model. This confirms that for AR models, the PACF helps determine the order, while the ACF doesn't have a clear cutoff.

4C: I first plotted its autocorrelation function (ACF) and partial autocorrelation function (PACF) to identify patterns. The ACF showed a gradual decay instead of a sharp cutoff, which is a strong sign that the data follows an autoregressive (AR) process rather than a moving average (MA) process. The PACF, on the other hand, cut off after a few lags, suggesting that the order of the AR process could be determined by where the PACF becomes insignificant. To confirm this, I tested different AR and MA models and compared their AICc (corrected Akaike Information Criterion) values to find the best fit. The results showed that the AR(3) model had the lowest AICc, meaning it provided the best balance between complexity and accuracy. Based on this, I would use an AR(3) process to model the data because it best captures the underlying structure, as indicated by both the PACF cutoff and the AICc comparison.

4D: To determine the best-fitting model, I first took the AR(3) model from part C, since its PACF cutoff and ACF decay suggested an autoregressive structure. To make sure it was truly the best choice, I also fit other AR and MA models, including AR(1), AR(2), AR(3), MA(1), MA(2), MA(3), ARMA(1,1), and ARMA(2,2). For each model, I calculated its AICc (corrected Akaike Information Criterion), which helps compare models by balancing goodness of fit and complexity. After sorting the AICc values, I found that AR(3) had the lowest AICc, meaning it provided the best fit with the least unnecessary complexity. This confirms that AR(3) is the best model for the data, as it effectively captures the underlying structure while avoiding overfitting.

5A: For this part, I wrote a function to calculate an exponentially weighted covariance matrix using a decay factor  $\lambda$ . The idea is that more recent data points should have a higher weight than older ones, which helps capture changing market conditions. To do this, I first generated a series of exponentially decaying weights based on  $\lambda$ , ensuring they sum to 1 for proper normalization. Then, I computed a mean-adjusted return matrix, where each observation is centered around a weighted mean instead of a simple average. Finally, I used matrix multiplication to obtain the weighted covariance matrix. To verify the results, I compared my custom implementation with Pandas' built-in exponentially weighted covariance function `ewm().cov()`. Both methods produced similar results, confirming that the implementation is correct. This approach ensures that recent observations influence the covariance estimate more than past ones, making it ideal for financial applications like risk modeling.

5B: For this part, I tested how different values of  $\lambda$  (0.90, 0.94, and 0.98) affect the exponentially weighted covariance matrix and then analyzed the results using PCA. First, I computed the covariance matrix for each  $\lambda$  using the same exponentially weighted approach. Then, I applied PCA to each matrix to decompose it into eigenvalues and eigenvectors, which helped determine how much variance each principal component explains. To visualize this, I plotted the cumulative variance explained as a function of the number of principal components. The results showed that smaller  $\lambda$  values (like 0.90) concentrate variance into fewer components, meaning fewer factors drive most of the data's movement. On the other hand, larger  $\lambda$  values (like 0.98) spread the variance across more components, indicating that recent data influences the structure more, making the system more dynamic. This confirms that adjusting  $\lambda$  changes how sensitive the covariance matrix is to recent vs. older data.

5C: From the analysis, I found that  $\lambda$  controls how much weight recent data gets when calculating the covariance matrix. A higher  $\lambda$  (e.g., 0.98) puts more emphasis on the most recent observations, making the covariance matrix more reactive to short-term market

changes. This means that variance is spread across more principal components in PCA, capturing more fluctuations. In contrast, a lower  $\lambda$  (e.g., 0.90) gives older data more influence, making the covariance estimate smoother and more stable over time. This results in a stronger first principal component, meaning fewer components explain most of the variance. In financial applications, choosing the right  $\lambda$  depends on whether you want a model that quickly adapts to new market conditions (higher  $\lambda$ ) or one that maintains a more stable, long-term view of risk (lower  $\lambda$ ). This balance is crucial for applications like portfolio risk estimation and volatility modeling.

6B: We simulated 10,000 draws using Principal Component Analysis while ensuring that at least 75% of the variance in the original covariance matrix was retained. First, we computed the eigenvalues and eigenvectors of the covariance matrix and sorted them in descending order. We then calculated the cumulative variance explained by the eigenvalues and selected the smallest number of components that captured at least 75% of the total variance. After that, we generated standard normal samples, projected them onto the selected principal components, and transformed them back into the original space. This method helps reduce dimensionality while still preserving most of the data's structure.

6C: we compared how well the Cholesky-based and PCA-based simulations preserved the original covariance structure. First, we computed the empirical covariance matrices for the Cholesky samples (`cholesky_cov`) and PCA samples (`pca_cov`) using `np.cov()`. Then, we measured the difference between these covariance matrices and the original covariance matrix (`cov_matrix`) using the Frobenius norm, calculated with `np.linalg.norm(matrix1 - matrix2, 'fro')`. The Frobenius norm gives a single value representing how much the simulated covariance matrices deviate from the original. A lower Frobenius norm means the method is more accurate in reconstructing the original covariance matrix in this case.

6D: we analyzed how well the Cholesky and PCA-based simulations preserved the variance structure of the original covariance matrix. First, we computed the eigenvalues of the original covariance matrix (`cov_matrix`), as well as those of the covariance matrices obtained from the Cholesky (`cholesky_cov`) and PCA (`pca_cov`) simulations using `np.linalg.eigh()`. We then sorted these eigenvalues in descending order to prioritize the most significant variance components. Next, we calculated the cumulative variance explained by each set of eigenvalues using `np.cumsum(eigenvalues) / np.sum(eigenvalues)`, which tells us how much of the total variance is captured as we add more components. Finally, we used Matplotlib to plot the cumulative variance curves for

the original, Cholesky, and PCA-based simulations, helping us visualize how well each method preserves the original variance structure.

6E: We measured the execution time for simulating 10,000 draws using both the Cholesky decomposition and PCA-based methods. We used Python's time module to track how long each method takes. First, we recorded the start time with `time.time()`, ran the `simulate_cholesky(cov_matrix)` function, and then calculated the total runtime by subtracting the start time from the new `time.time()`. We repeated the same process for `simulate_pca(cov_matrix)`. Finally, we printed the execution times for both methods, showing which one is faster. In this case, PCA simulation time is faster with 0.03571s when Cholesky Simulation Time is 0.14172s.

6F: Cholesky decomposition fully preserves the original covariance structure, making it more accurate but computationally expensive, especially for large matrices. It also requires the covariance matrix to be positive definite, which sometimes needs adjustments. On the other hand, PCA-based simulation reduces dimensionality by keeping only a subset of the principal components, making it more efficient but at the cost of losing some variance and correlations. PCA is useful when working with high-dimensional data where capturing the most significant variance is more important than full accuracy. Overall, Cholesky is ideal when precision is required, while PCA is a great alternative when computational efficiency is a priority.