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.

Pairwise Covariance Matrix:

	x1	x2	x3	x4	x5
x1	1.470484	1.454214	0.877269	1.903226	1.444361
x2	1.454214	1.252078	0.539548	1.621918	1.237877
x3	0.877269	0.539548	1.272425	1.171959	1.091912
x4	1.903226	1.621918	1.171959	1.814469	1.589729
x5	1.444361	1.237877	1.091912	1.589729	1.396186

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.

```
Positive semi-definite? False
Eigenvalues: [ 6.78670573  0.83443367 -0.31024286  0.02797828 -0.13323183]
```

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

Nearest PSD Covariance Matrix:

	0	1	2	3	4
0	1.470484	1.332361	0.884378	1.627602	1.399556
1	1.332361	1.252078	0.619028	1.450604	1.214450
2	0.884378	0.619028	1.272425	1.076847	1.059658
3	1.627602	1.450604	1.076847	1.814469	1.577928
4	1.399556	1.214450	1.059658	1.577928	1.396186

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.

Nearest Positive Semi-Definite Matrix (Rebonato and Jackel):

	0	1	2	3	4
0	1.470484	1.327009	0.842583	1.624464	1.364833
1	1.327009	1.252078	0.555421	1.433109	1.165906
2	0.842583	0.555421	1.272425	1.052789	1.060424
3	1.624464	1.433109	1.052789	1.814469	1.544993
4	1.364833	1.165906	1.060424	1.544993	1.396186

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

Difference between Higham's PSD and Overlapping Data Covariance Matrix:

	x1	x2	x3	x4	x5
x1	3.912592e-08	-1.218535e-01	7.108576e-03	-2.756246e-01	-4.480541e-02
x2	-1.218535e-01	1.916824e-08	7.947982e-02	-1.713142e-01	-2.342662e-02
x3	7.108576e-03	7.947982e-02	3.488501e-09	-9.511247e-02	-3.225366e-02
x4	-2.756246e-01	-1.713142e-01	-9.511247e-02	5.778196e-08	-1.180031e-02
x5	-4.480541e-02	-2.342662e-02	-3.225366e-02	-1.180031e-02	3.811949e-08

Difference between Rebonato and Jackel's PSD and Overlapping Data Covariance Matrix:

	x1	x2	x3	x4	x5
x1	2.220446e-16	-1.272051e-01	-3.468565e-02	-2.787626e-01	-7.952841e-02
x2	-1.272051e-01	2.220446e-16	1.587256e-02	-1.888090e-01	-7.197070e-02
x3	-3.468565e-02	1.587256e-02	-2.220446e-16	-1.191699e-01	-3.148783e-02
x4	-2.787626e-01	-1.888090e-01	-1.191699e-01	2.220446e-16	-4.473607e-02
x5	-7.952841e-02	-7.197070e-02	-3.148783e-02	-4.473607e-02	-2.220446e-16

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.

Under this circumstance, Higham's method and Rebonato and Jackel's method both adjust the original covariance matrix to ensure it is positive semi-definite. The overlapping data covariance matrix is calculated using only complete cases, which may result in a different matrix due to the exclusion of incomplete rows. The differences between the adjusted

matrices and the overlapping data matrix highlight how the methods handle missing data and enforce positive semi-definiteness.

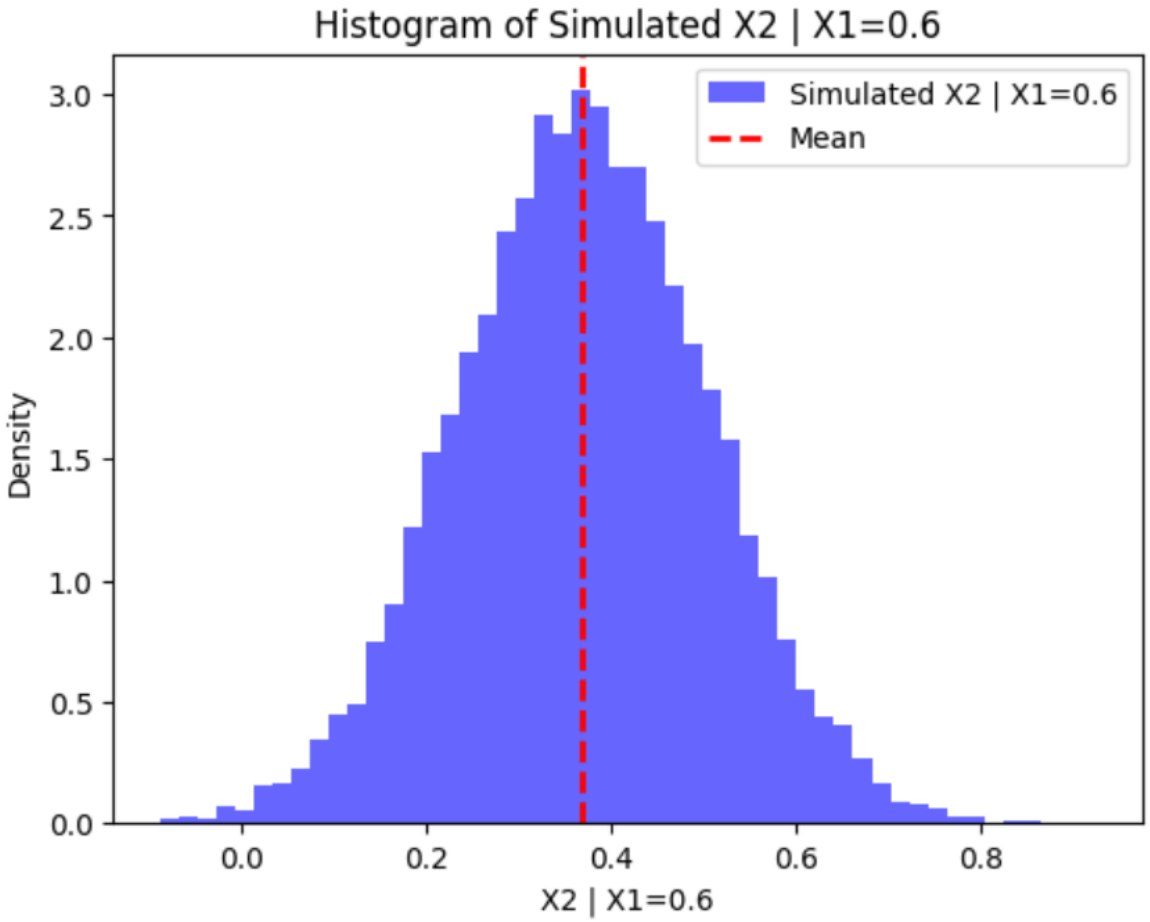
3A: To fit a multivariate normal distribution to the data, I first loaded the dataset and calculated its mean vector and covariance matrix. The mean vector gives the average values of X1 and X2, which act as the center of our normal distribution. The covariance matrix tells us how these two variables 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.

```
Mean vector:
[0.04600157 0.09991502]

Covariance matrix:
[[0.0101622  0.00492354]
 [0.00492354 0.02028441]]
```

3B: The implemented Python code determines the conditional distribution of X2 given X1 =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 X1 is approximately equal to 0.6 are extracted, and the empirical mean and variance of X2 are calculated to approximate its conditional distribution.

Here is the result of conditional mean and conditional variance:

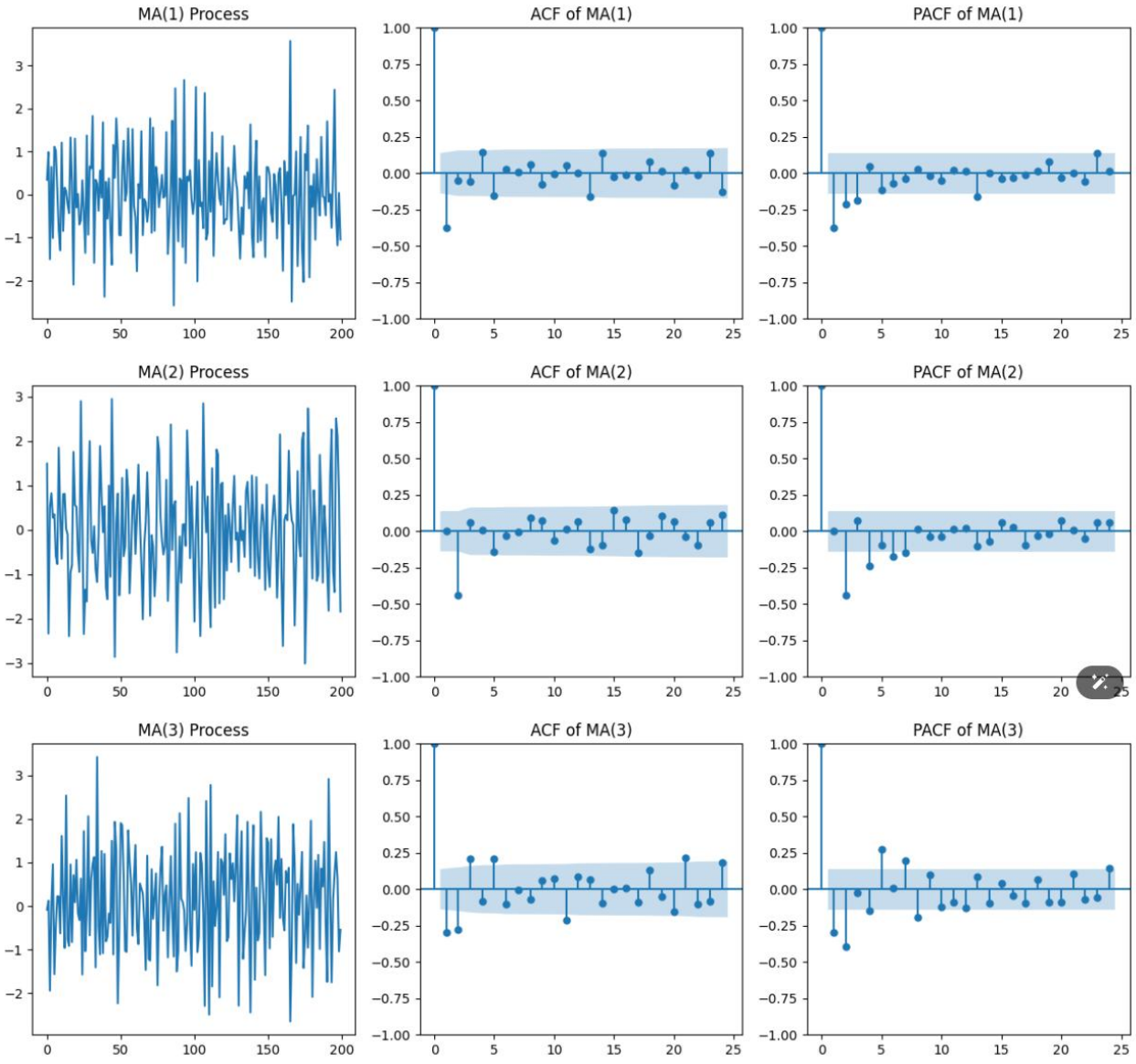
```
Conditional Mean: 0.3683249958609775
Conditional Variance: 0.01789896964508752
```



3C: In the code, I used NumPy for numerical computations and Cholesky decomposition to find the conditional mean and variance of X2|X1=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 L11 ,L21 ,L22 to express the conditional mean using μ2 +(L21/L11)(x1−μ1), and the conditional variance as L22^2 . 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 X2|X1=0.6. This also proves that our simulation approach is correct because both methods lead to the same expected distribution – the very close conditional mean and conditional variance.

```
Conditional Mean using cholesky root: 0.3683249958609775
Conditional Variance using cholesky root: 0.017898969645087522
```

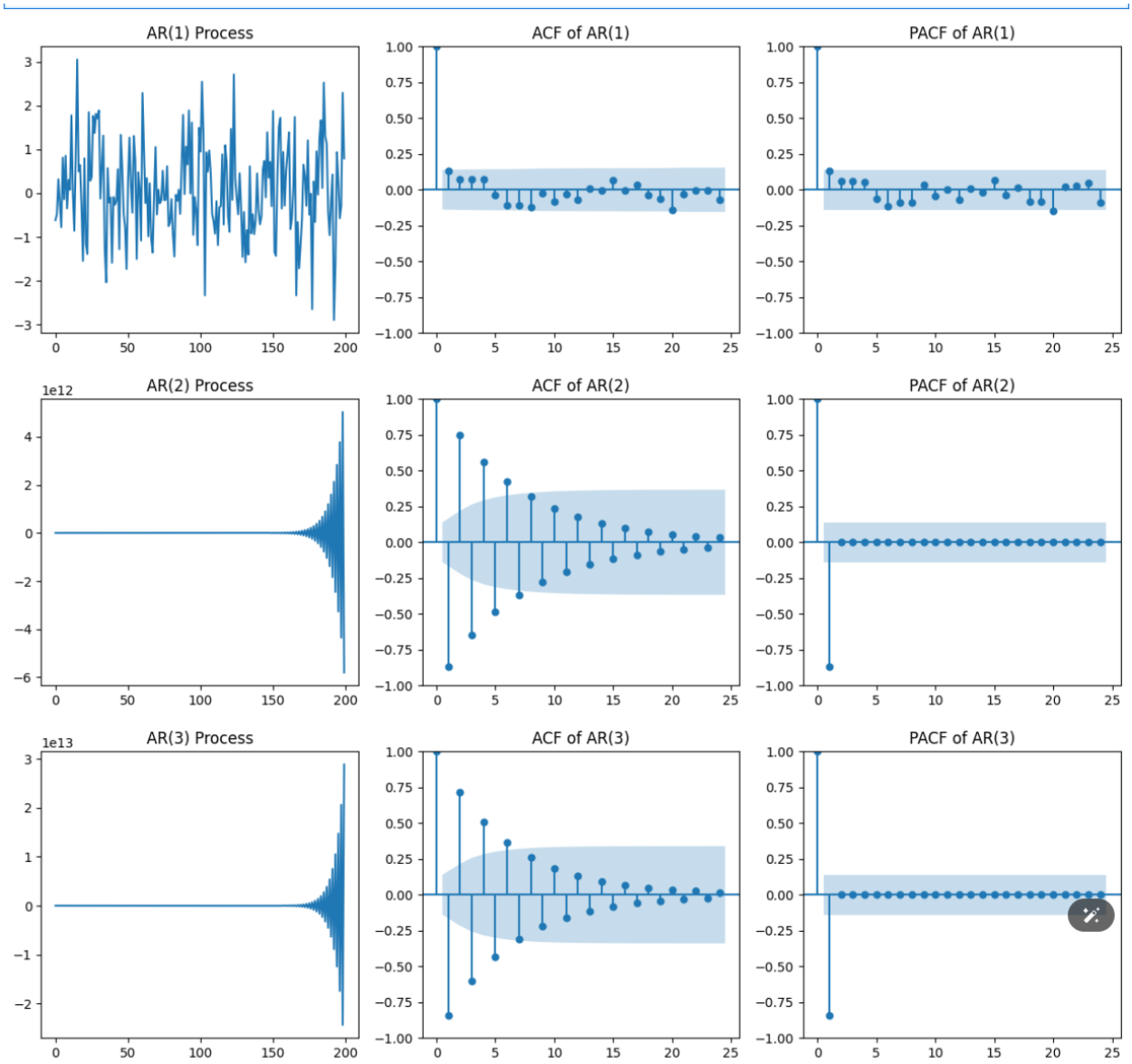
4A: For simulating MA(1), MA(2), and MA(3) processes, I first generated random moving average (MA) coefficients between -0.8 and 0.8 to ensure realistic behavior. Using these coefficients, I created MA processes with the ArmaProcess function from statsmodels, specifying an autoregressive (AR) part of 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. Below are the plots generated.



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.

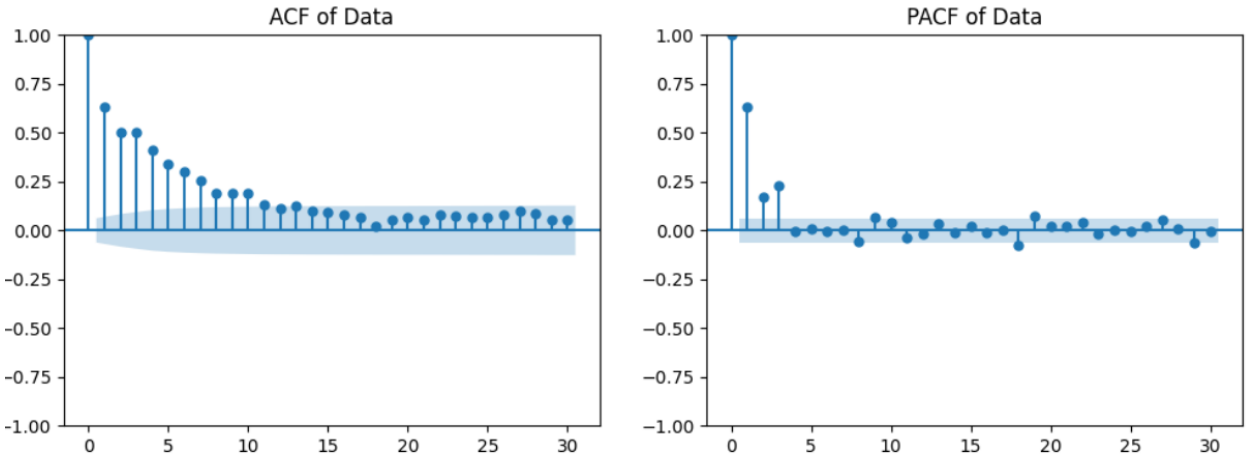
4B: For simulating AR(1), AR(2), and AR(3) processes, I started by generating random autoregressive (AR) coefficients between -0.8 and 0.8 to make the models behave realistically. Using the ArmaProcess function from statsmodels, I defined each AR process with its respective order while keeping the moving 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. Below are the plots generated



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.



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.

	AICc
AR(3)	-1746.221359
ARMA(2,2)	-1739.961771
ARMA(1,1)	-1723.414338
AR(2)	-1696.051484
AR(1)	-1669.065171
MA(3)	-1645.072607
MA(2)	-1559.210731
MA(1)	-1508.902937

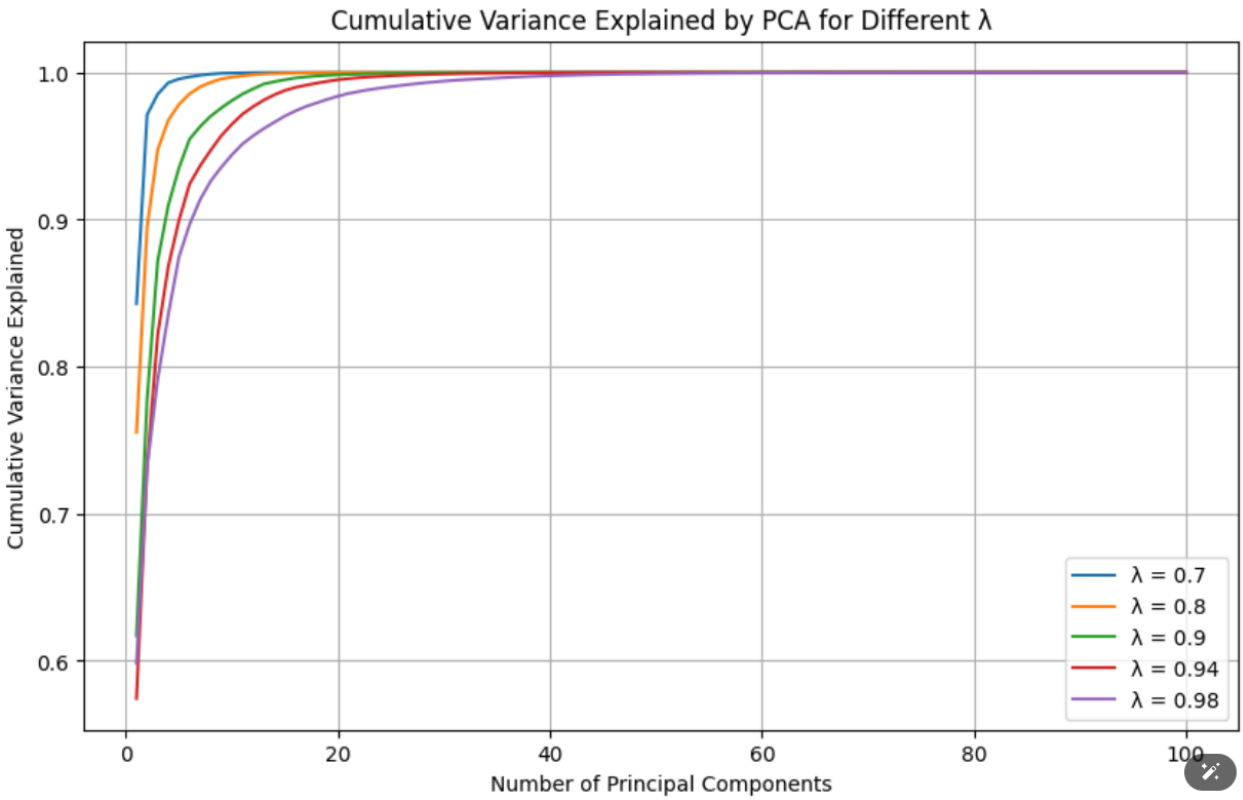
5A: For this part, I wrote a function to calculate an exponentially weighted covariance matrix using a decay factor λ. The idea comes from the class pdf of JP Morgan. To do this, I first generated a series of exponentially decaying weights based on λ, ensuring they sum to

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.

```
Custom Exponentially Weighted Covariance Matrix:
          SPY      AAPL      NVDA      MSFT      AMZN      META     GOOGL  \
SPY   0.000072  0.000054  0.000126  0.000081  0.000112  0.000082  0.000087
AAPL  0.000054  0.000140  0.000042  0.000085  0.000080  0.000058  0.000070
NVDA  0.000126  0.000042  0.000670  0.000137  0.000195  0.000192  0.000143
MSFT  0.000081  0.000085  0.000137  0.000163  0.000173  0.000129  0.000121
AMZN  0.000112  0.000080  0.000195  0.000173  0.000323  0.000187  0.000201
...        ...       ...       ...       ...       ...       ...       ...
KKR   0.000135  0.000042  0.000223  0.000107  0.000188  0.000121  0.000160
MU    0.000150  0.000057  0.000314  0.000157  0.000171  0.000172  0.000164
PLD   0.000060  0.000061  0.000019  0.000064  0.000056  0.000018  0.000024
LRCX  0.000128  0.000085  0.000326  0.000155  0.000185  0.000232  0.000157
EQIX  0.000052  0.000038  0.000048  0.000053  0.000071  0.000073  0.000052

          AVGO      TSLA      GOOG  ...      SBUX       MMC       MDT  \
SPY   0.000174  0.000232  0.000088  ...  0.000041  0.000032  0.000030
AAPL  0.000137  0.000170  0.000070  ...  0.000008  0.000015  0.000009
NVDA  0.000212  0.000224  0.000146  ...  0.000039  0.000016  0.000012
MSFT  0.000189  0.000272  0.000121  ...  0.000018  0.000028  0.000020
AMZN  0.000297  0.000374  0.000199  ...  0.000019  0.000020  0.000012
...        ...       ...       ...  ...       ...       ...       ...
KKR   0.000272  0.000478  0.000163  ...  0.000088  0.000071  0.000060
MU    0.000882  0.000464  0.000178  ...  0.000074  0.000002  0.000031
PLD   0.000056  0.000113  0.000028  ...  0.000084  0.000048  0.000062
LRCX  0.000576  0.000435  0.000161  ...  0.000073  0.000009  0.000020
EQIX  0.000066  0.000156  0.000053  ...  0.000059  0.000044  0.000025

            CB       LMT       KKR        MU       PLD      LRCX      EQIX
SPY   0.000028  0.000027  0.000135  0.000150  0.000060  0.000128  0.000052
AAPL  0.000026  0.000004  0.000042  0.000057  0.000061  0.000085  0.000038
NVDA -0.000006  0.000034  0.000223  0.000314  0.000019  0.000326  0.000048
MSFT  0.000026 -0.000011  0.000107  0.000157  0.000064  0.000155  0.000053
AMZN  0.000015 -0.000009  0.000188  0.000171  0.000056  0.000185  0.000071
...        ...       ...       ...       ...       ...       ...       ...
KKR   0.000072  0.000066  0.000419  0.000256  0.000067  0.000194  0.000098
MU   -0.000001  0.000045  0.000256  0.001488  0.000163  0.000713  0.000084
PLD   0.000035  0.000034  0.000067  0.000163  0.000254  0.000088  0.000087
LRCX  0.000011 -0.000014  0.000194  0.000713  0.000088  0.000742  0.000073
EQIX  0.000029  0.000005  0.000098  0.000084  0.000087  0.000073  0.000153

[100 rows x 100 columns]
```

5B: For this part, I tested how different values of λ (0.7, 0.8, 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 λ 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.



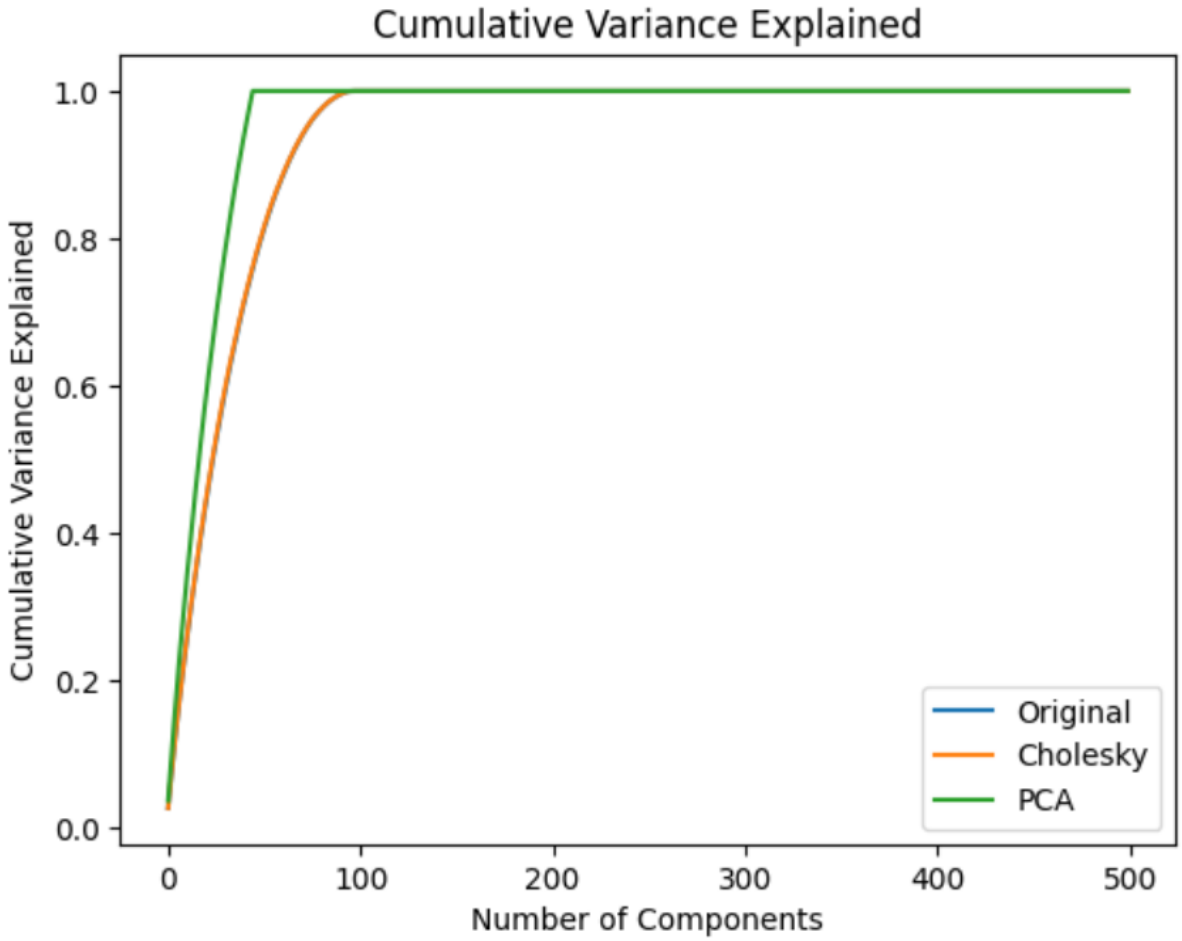
The results showed that smaller λ values (like 0.7) concentrate variance into fewer components, meaning fewer factors drive most of the data's movement. On the other hand, larger λ values (like 0.98) spread the variance across more components, indicating that recent data influences the structure more, making the system more dynamic.

5C: From the analysis, I found that λ controls how much weight recent data gets when calculating the covariance matrix. A higher λ (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 λ (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 λ depends on whether you want a model that quickly adapts to new market conditions (higher λ) or one that maintains a more stable, long-term view of risk (lower λ).

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.



The cumulative variance plot shows that PCA captures variance much faster than both the original and Cholesky-based covariance matrices, meaning fewer components explain most of the variance. Cholesky closely follows the original matrix, indicating it preserves the eigenvalue distribution better. PCA, on the other hand, concentrates variance into fewer dimensions, making it more efficient for dimensionality reduction but less accurate in maintaining the original covariance 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). 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.