Comparative Study of ncDMD, fbDMD, and tlsDMD: Balancing Bias Correction and Performance

MATH 5414 Final Project

Kristiana Nakaj

September 7, 2025

This paper compares three noise-aware Dynamic Mode Decomposition methods—noise-corrected DMD, forward-backward DMD, and total least-squares DMD on synthetic systems with known matrices and eigenvalues. By measuring each method's ability to eliminate sensor-noise bias, reduce estimate variance, and maintain computational efficiency, we aim to get a deeper understanding of these techniques and provide clear insights for choosing the optimal DMD variant.

1. Introduction

Dynamic Mode Decomposition (DMD) is a data-driven method that computes a low-rank linear approximation of system dynamics using only measurement data. While Proper Orthogonal Decomposition (POD) finds the most energetic spatial patterns and Fourier Analysis decomposes signals into basic frequencies, DMD links spatial patterns to their specific temporal evolution [1]. The main advantage is that DMD operates without requiring prior knowledge of the system's governing equations. Originally introduced by Schmid [4] for analyzing fluid dynamics, DMD's application has since expanded into diverse fields such as neuroscience, financial engineering, epidemiology, robotics, and video processing. As a result, it has become a general tool for analyzing systems where traditional model derivation is impracticable.

Essentially DMD decomposes time-series data into dynamic **modes**. Each mode represents a spatial structure that evolves in time with a fixed frequency and grows or decays exponentially at a fixed rate. These characteristics are governed by a complex eigenvalue associated with each mode. This modal decomposition reveals the dominant frequencies and stability properties in the data. Moreover, it often serves as a dimensionality reduction technique, by capturing the high-dimensional dynamics with a significantly smaller set of dominant modes.

2. Project Motivation and Objectives

While standard DMD provides a strong foundation for data analysis, its application to real-world problems has several limitations. A frequent limitation is its sensitivity to additive sensor noise. The standard DMD method assumes that all the error in the data lies within the future snapshots, and considers the initial data to be noise-free. This asymmetry introduces bias into the estimated eigenvalues, and therefore affects the accuracy of the growth/decay rates, leading to misinterpretations of the system dynamics [5].

Recognizing this limitation, an area of DMD research focuses on developing noise-robust DMD variants. In particular, Dawson *et al.* [3] introduced three modified DMD algorithm strategies:

1. Directly correcting the bias based on known noise characteristics.
2. Merging the results of DMD performed forward and backward in time.
3. Using total least squares regression, which treats noise in both input and output data matrices more symmetrically.

Although Dawson *et al.* provided insights into the advantages of these algorithms, a quantitative comparison of their performance trade-offs remains useful. This project aims to provide such reference point by comparing them on synthetic systems and evaluating which of the three most effectively removes **sensor-noise bias**, minimizes estimation **variance**, and does so with the lowest **computational cost**. The results aim to (i) verify the theoretical bias predictions, (ii) reveal variance–cost trade-offs, (iii) offer guidance to those employing the method on selecting the appropriate variant based on their data, the noise environment, and available computational resources.

3. Literature Review

3.1. Standard DMD Computation

Given sequential state snapshots $x_k \in \mathbb{R}^n$ sampled at a fixed interval Δt, DMD assumes the dynamics can be approximated by an unknown linear operator A, such that $x_{k+1} \approx Ax_k$. To estimate it, standard DMD stacks $m = T + 1$ sequential snapshots into the matrices:

$$X = [x_0, x_1, \dots, x_{T-1}] \in \mathbb{R}^{n\times m}, \quad Y = [x_1, x_2, \dots, x_T] \in \mathbb{R}^{n\times m}.$$

and computes the best-fit linear operator $A^\star$ minimizing $\|Y - AX\|_F$ in a least-squares sense, given by $A^\star = YX^+$, where X^+ is the Moore–Penrose pseudoinverse of X.

For high-dimensional systems, directly computing and analyzing $A^\star \in \mathbb{R}^{n\times n}$ is computationally hard. Instead, DMD uses the economy-size **singular value decomposition** (SVD) of the input data matrix X:

$$X = U_r \Sigma_r V_r^*, \quad r \ll min(n, m),$$

, where $U_r \in \mathbb{R}^{n\times r}$ contains the first r left singular vectors (capturing the dominant spatial dynamics, known as POD modes), $\Sigma_r \in \mathbb{R}^{r\times r}$ is a diagonal matrix holding the first r largest singular values, and $V_r \in \mathbb{R}^{m\times r}$ contains the first r right singular vectors (representing the dominant time structures). The dynamics are then projected onto the subspace spanned by the columns of U_r, resulting in a a low-dimensional operator:

$$\tilde{A} = U_r^* A^\star U_r = U_r^* Y V_r \Sigma_r^{-1} \in \mathbb{R}^{r\times r}.$$

Diagonalizing this reduced operator, $\tilde{A}w_j = \lambda_j w_j$, returns eigenvalues λ_j (each mode's frequency/growth rate) and eigenvectors w_j. The full-space DMD modes ϕ_j are then reconstructed by projecting w_j through the basis U_r:

$$\phi_j = U_r w_j$$

There exist different variations in how this last step is performed, for example we can calculate 'exact DMD modes' that incorporate Y directly by $\phi_j = \mu_j^{-1} Y V_r \Sigma_r^{-1} w_j$ [2].

3.2. Sensor Noise

Sensor noise is defined as an additive error ($x_m = x + n$) that corrupts the measurements without affecting the system dynamics, and it is typically assumed to be uncorrelated with the true state. Dawson *et al.* used perturbation theory to analyze the expected value of the DMD operator ($\tilde{A}_m$) computed from noisy data matrices ($X_m = X + N_X$, $Y_m = Y + N_Y$) [3]. Although under standard assumptions linear terms average out, quadratic terms have a consistent non-zero error. This error is distinct from the random error (i.e., variance) that comes from finite data sampling. The dominant component of this bias modifies the expected value of the estimated operator (in the POD basis of X_m) via:

$$\mathbb{E}(\tilde{A}_m) \approx \tilde{A}\left(I - \mathbb{E}(\tilde{N}_X \tilde{N}_X^*)(\tilde{X}\tilde{X}^*)^{-1}\right)$$

, where $\tilde{A}$ is the true operator, $\tilde{N}_X$ is the noise in the projected X data, and $\tilde{X}\tilde{X}^*$ is related to the projected clean data covariance [3].

The authors also argued that this bias becomes the dominant source of error when the number of snapshots m is large relative to the state dimension n. Therefore, understanding it is essential for interpreting DMD results and developing noise-aware DMD algorithms.

3.3. Three Noise-Aware DMD Algorithms

Noise-corrected DMD (ncDMD) estimates the true system operator $\tilde{A}$ by applying a direct correction to $\tilde{A}_m$, based on an inverted approximation of the bias. The general form of this correction is given by:

$$\tilde{A} \approx \tilde{A}_m \left(I - \mathbb{E}(\tilde{N}_X \tilde{N}_X^*)(\tilde{X}\tilde{X}^*)^{-1}\right)^{-1} \tag{1}$$

such that $\mathbb{E}(\tilde{N}_X \tilde{N}_X^*)$ is the expected noise covariance in the projected space of X, and $\tilde{X}\tilde{X}^*$ represents the covariance of the projected clean data [3].

Directly applying Equation (1) requires estimates of the noise covariance term and the clean data covariance, which are usually unknown. Hence, the authors introduced two approximations. First, the clean data covariance projected onto the basis derived from the noisy data ($X_m = U_m \Sigma_m V_m^*$)

is approximated using the singular values of the noisy data itself: $\tilde{X}\tilde{X}^* \approx U_m^* X_m X_m^* U_m = \Sigma_m^2$. Second, assuming that the sensor noise affecting each measurement is independent, identically distributed (i.i.d) with zero mean and a known variance σ_N^2, the noise covariance term simplifies to $\mathbb{E}(\tilde{N}_X \tilde{N}_X^*) = \mathbb{E}(U_m^* N_X N_X^* U_m) \approx m\sigma_N^2 I$, with m being the number of snapshot pairs used.

Substituting these two approximations into (1) gives the ncDMD estimate for the system operator:

$$\tilde{A}_{\mathrm{ncDMD}} \approx \tilde{A}_m \left(I - m\sigma_N^2 \Sigma_m^{-2}\right)^{-1}. \tag{2}$$

For a successful application, we must have an accurate estimate for the sensor noise variance σ_N^2. Furthermore, the dependence on the inverse square of the singular values, Σ_m^{-2}, indicates potential numerical sensitivity when applying the correction, especially if some of the singular values are very small. [3].

Forward–Backward DMD (fbDMD) combines the standard forward-in-time DMD with a backward-in-time counterpart. Forward DMD gives $\tilde{A}_m$ mapping X_m to Y_m, while reversing the data gives a backward DMD operator $\tilde{B}_m$ that approximates the inverse dynamics mapping Y_m back to X_m. For an invertible system, $\tilde{B}_m$, computed from noisy data, presents a similar bias as $\tilde{A}_m$, leading to an underestimation of its eigenvalues. Therefore, when $\tilde{B}_m$ is inverted to estimate the forward dynamics— i.e., $\tilde{A}_m^{\mathrm{back}} = \tilde{B}_m^{-1}$, the resulting eigenvalues tend to be overestimated compared to those of the true operator $\tilde{A}$.

The idea of fbDMD is that the inward shift in $\tilde{A}_m$ and the outward shift in $\tilde{A}_m^{\mathrm{back}}$ can partially cancel each other. Dawson *et al.* [3] showed that the product of these operators approximates the square of the true operator:

$$\tilde{A}_m \tilde{A}_m^{\mathrm{back}} \approx \tilde{A}^2. \tag{3}$$

It follows that the unbiased $\tilde{A}$ can be estimated by taking the matrix square root of this product:

$$\tilde{A}_{\mathrm{fbDMD}} \approx \left(\tilde{A}_m(\tilde{B}_m^{-1})\right)^{1/2} \equiv \left(\tilde{A}_m \tilde{A}_m^{\mathrm{back}}\right)^{1/2} \tag{4}$$

A major benefit of fbDMD is that it does not rely on any prior knowledge of the of the sensor noise statistics (e.g., σ_N^2). However, it requires $\tilde{B}_m$ to be invertible. Additionally, the matrix square root is generally not unique, although a way to settle this ambiguity is by choosing the root closest to $\tilde{A}_m$ [3].

We note that we implement the fbDMD following [3], which utilizes operators projected onto POD bases derived from the input data. The forward operator $\tilde{A}_f$ uses the POD basis ($U_{r,f}$) obtained from the SVD of X_m, while the backward operator $\tilde{B}_b$ uses the basis ($U_{r,b}$) from the SVD of Y_m. Since $U_{r,f}$ and $U_{r,b}$ are generally not identical, the combination $\tilde{A}_f(\tilde{B}_b)^{-1}$ involves a basis mismatch. However, this implementation is widely used, suggesting the dominant POD subspaces often sufficiently overlap in practice.

Total Least-Squares DMD (tlsDMD) uses the Total Least Squares approach to account for errors in both data matrices. Hence, the goal of tlsDMD is to find a linear operator A and small corrections E_X, E_Y that satisfy

$$(Y + E_Y) = A(X + E_X)$$

, while minimizing the Frobenius norm of the total error matrix

$$E = \begin{bmatrix} E_X \\ E_Y \end{bmatrix}.$$

To handle high-dimensional snapshots, Dawson *et al.* first project the data onto a POD basis of dimension r (with $r < m/2$), obtaining reduced $\tilde{X}$ and $\tilde{Y}$. They then form the augmented matrix, and compute its SVD:

$$\begin{bmatrix} \tilde{X} \\ \tilde{Y} \end{bmatrix} = U \Sigma V^*,$$

where $U \in \mathbb{C}^{2r \times 2r}$ and $V \in \mathbb{C}^{m \times m}$ are unitary, and $\Sigma \in \mathbb{R}^{2r \times m}$ is diagonal. Next, U is split into four $r \times r$ blocks:

$$U = \begin{bmatrix} U_{11} & U_{12} \\ U_{21} & U_{22} \end{bmatrix}.$$

Using TLS minimization conditions and the Eckart-Young theorem [3], the tlsDMD estimated operator in the POD basis is then given by:

$$\tilde{A}_{\text{tlsDMD}} = U_{21} U_{11}^{-1}. \tag{5}$$

Like fbDMD, tlsDMD corrects for bias without needing a prior estimate of σ_N^2. However, the successful computation via Equation (5) depends on the invertibility of the submatrix U_{11}.

4. Data and Test Cases

We evaluate each DMD variant on synthetic data generated from discrete-time linear systems

$$x_{k+1} = A_{\text{true}}\, x_k,$$

with $x_k \in \mathbb{R}^n$. From an initial condition x_0, we simulate T steps to build clean snapshots

$$X_{\text{clean}} = [\, x_0, \ldots, x_{T-2} \,], \quad Y_{\text{clean}} = [\, x_1, \ldots, x_{T-1} \,].$$

Each state is then corrupted by i.i.d. Gaussian noise $n_k \sim \mathcal{N}(0, \sigma_N^2 I)$ (where σ_N is the noise standard deviation), producing noisy matrices X_{noisy} and Y_{noisy}. We run every algorithm on five systems of differing size and frequency features, repeating trials over several noise levels σ_N. In all cases, the rank r for POD truncation is set to full order ($r = n$).

Case 0 — Illustrative Simple Case: This is the starting point for initial debugging. We construct a low-dimensional 2×2 system, characterized by a pair of stable complex conjugate eigenvalues. This case highlights the fundamental behavior and noise sensitivity of each algorithm over $T = 100$

time steps with a fixed initial state $x_0 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$.

Case 1 — Well-Conditioned Baseline with Well-Separated Dynamics: We construct a 4×4 linear system, whose four eigenvalues are well-seperated—a stable real root, an unstable real root, and a stable complex-conjugate pair— and whose eigenvectors are constructed to be well-conditioned. We use this case to assess how each algorithm handles sensor noise under ideal system conditions. We simulate the response for $T = 150$ time steps from a random initial state x_0. This choice of x_0 ensures the methods are evaluated on a more general range of system behaviors.

Case 2 — Generic Coupled Modes System: Here we build a 4×4 linear system with one stable real eigenvalue, one unstable real eigenvalue, and a pair of stable complex-conjugate eigenvalues. Unlike Case 1, the true system matrix A_{true} is obtained by applying a random similarity transform to a block-diagonal Jordan form, coupling the modes and producing non-orthogonal eigenvectors. As a result, we get a more realistic scenario with non trivial dynamics, which we simulate over $T = 150$ time steps from a random x_0.

Case 3 — Closely Spaced Eigenvalues: This case constructs a 4×4 linear system with two pairs of stable complex-conjugate eigenvalues whose magnitudes and frequencies lie very near each other. A_{true} is formed in a block-diagonal Jordan form to ensure well-conditioned eigenvectors despite their tight spacing. While Dawson *et al.* [3] evaluated algorithm performance on synthetic systems, they did not explore cases with closely spaced eigenvalues under varying noise levels. This case thus aims to provide new insights into how accurately each algorithm can recover nearly overlapping modal frequencies from noise-corrupted measurements—a common difficulty when analyzing complex datasets.

Case 4 — Non-Normal System with Transient Growth: Unlike the examples in the original study, this case compares the performance of each DMD variant on a non-normal system designed to produce transient energy growth despite asymptotic stability. The true system matrix

$$A_{\text{true}} = \begin{pmatrix} \lambda & \alpha \\ 0 & \lambda \end{pmatrix}$$

has $|\lambda| < 1$ to ensure eventual decay, while $\alpha \neq 0$ introduces the non-normal behavior. We simulate the system response over $T = 150$ time steps from a random initial state x_0. We then evaluate how ncDMD, fbDMD, and tlsDMD recover the true eigenvalue λ under the combined effect of non-normal dynamics and sensor noise.

Case 5 — Higher-Dimensional System: We consider a 6×6 linear system with two stable complex-conjugate pairs, one stable and one unstable real eigenvalue. The true system matrix A_{true} is formed by applying a random similarity transform to its Jordan-block form. This case, also simulated for $T = 150$ time steps from a random initial condition x_0, is used to explore the limits of the algorithms in a larger state space.

5. Metrics

We implement all four DMD algorithms according to Dawson *et al.* [3], using the economy SVD throughout. For each test scenario and noise level, we conduct multiple N_{trials} independent simulations. The performance of each variant is then assessed based on the following metrics, computed across all trials:

1. **Bias Correction:** For each trial, we pair the computed eigenvalues $\hat{\lambda}$ with the sorted true eigenvalues λ_{true} using `MATLAB`'s `matchpairs` function or, if unavailable, a greedy minimum-distance approach. The bias for the i-th eigenvalue is then given by $\text{Bias}_i = \mathbb{E}[\hat{\lambda}_i] - \lambda_{\text{true},i}$. We report either the mean absolute bias $\frac{1}{r}\sum_{i=1}^{r} |\text{Bias}_i|$, or the individual biases $|\text{Bias}_i|$.

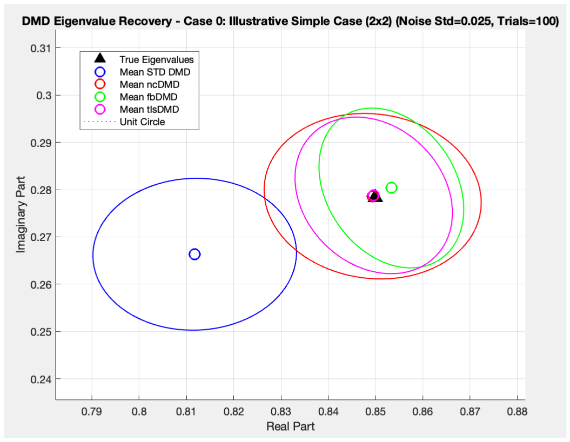
2. **Variance Reduction:** To quantify consistency, we compute for each matched eigenvalue i the total variance $\text{Var}_i = \text{Var}[\text{Re}(\hat{\lambda}_i)] + \text{Var}[\text{Im}(\hat{\lambda}_i)]$. We present the results as either the average of Var_i over all r modes or the individual variances.

3. **Computational Efficiency:** We record the wall-clock time for each algorithm on a single trial using `MATLAB`'s `tic` and `toc` calls. The metric is the average runtime per trial, calculated across all noise levels for a given test case. Memory usage is not tracked.

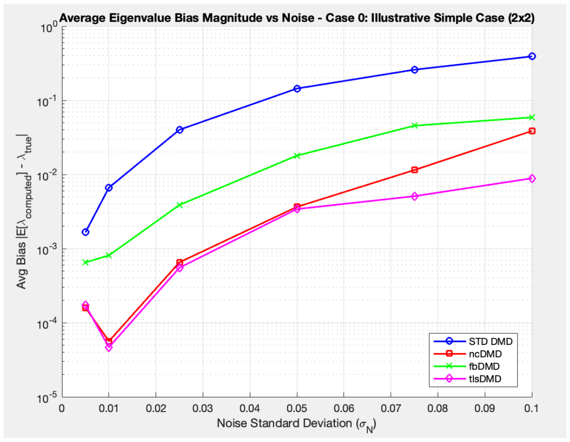
6. Results

Reproducibility Statement: The numerical simulations and analyses presented in this paper were performed using MATLAB R2023b Update 4 on a MacBook Pro with an Apple M3 Pro chip, 18 GB of unified memory, running macOS. Where applicable, functions from the MATLAB Statistics and Machine Learning Toolbox were used. All MATLAB scripts developed for this project are included in Appendix A.

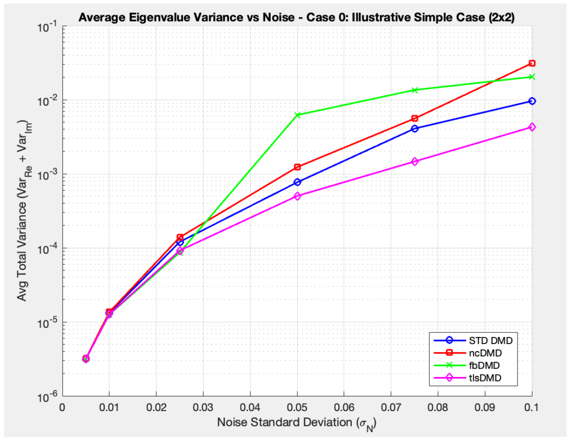
6.1. Case 0 Results



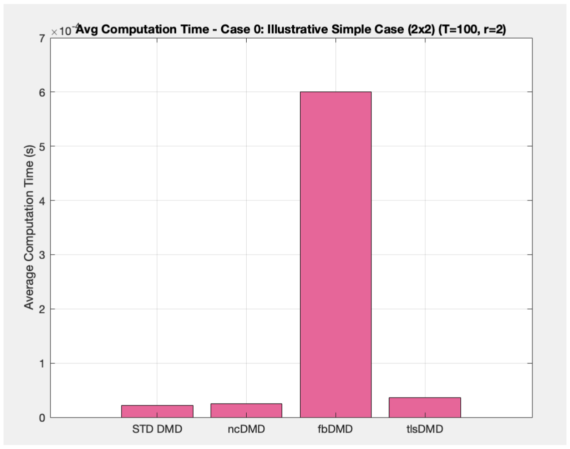
(a) Eigenvalue Recovery



(b) Bias Comparison



(c) Variance Comparison

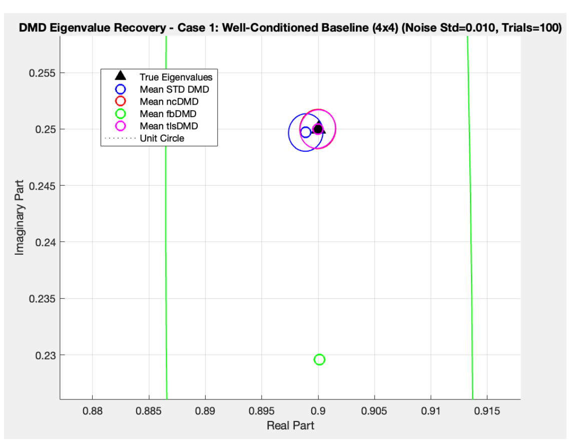


(d) Computation Time

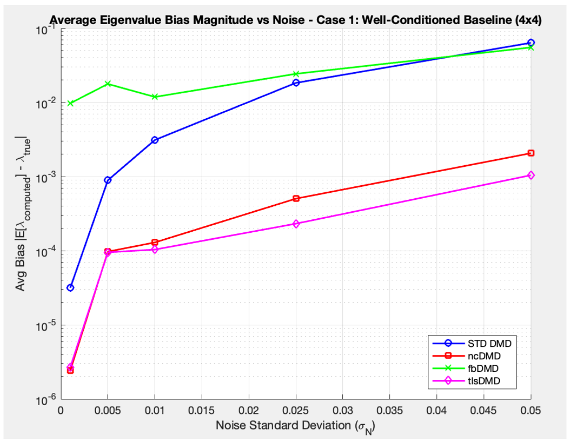
Figure 1: Performance metrics for Case 0

At the tested noise level $\sigma_N = 0.100$, standard DMD showed considerable average eigenvalue bias of roughly 0.391. All three correction methods reduced this bias, with tlsDMD performing best (Fig. 1b). tlsDMD also had the lowest variance (4.28×10^{-3}), outperforming particularly fbDMD, whose increased variance aligns with the matrix-square-root instability warnings we observed in `MATLAB`. In terms of runtime, standard DMD, ncDMD, and tlsDMD had comparable speeds, while fbDMD was significantly slower. Overall, as we can observe from Figure 1 plots, tlsDMD offers the best trade-off between accuracy, consistency, and computational efficiency.

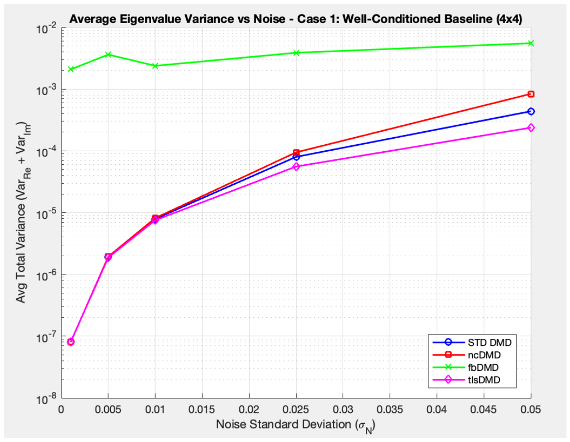
6.2. Case 1 Results



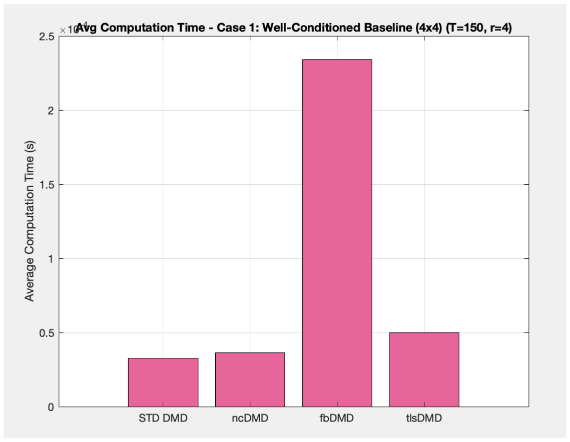
(a) Eigenvalue Recovery



(b) Bias comparison



(c) Variance comparison

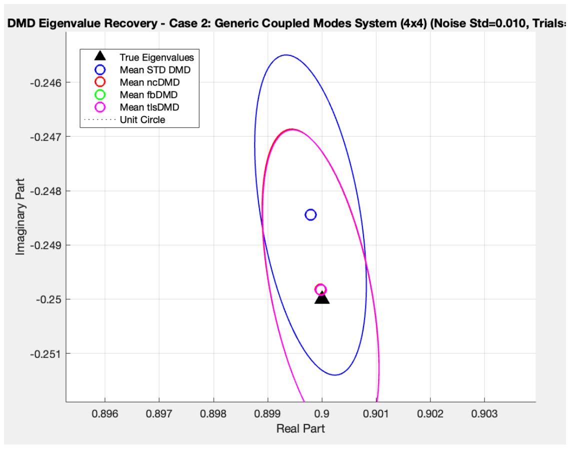


(d) Computation time

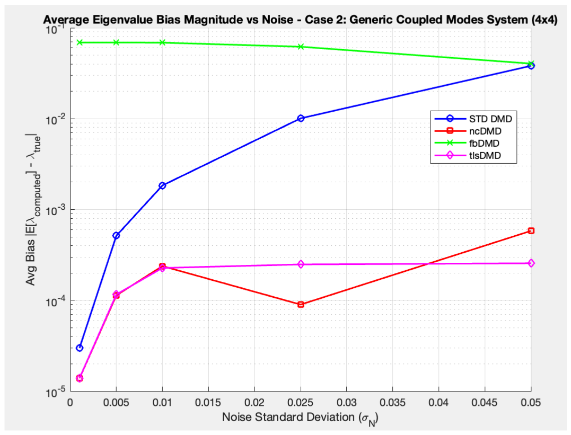
Figure 2: Performance metrics for Case 1

At a noise level of $\sigma_N = 0.050$, standard DMD produced an average eigenvalue bias magnitude of approximately 0.064. As expected, all noise-robust methods cut this bias, where tlsDMD demonstrated the highest accuracy with a mean bias of just 0.0010, followed by ncDMD at 0.0021. tlsDMD also achieved the lowest average total variance (2.37×10^{-4}), while ncDMD and fbDMD resulted in higher spreads of 8.36×10^{-4} and 5.52×10^{-3}, respectively. In terms of computational efficiency, from the noise-aware variants ncDMD was the fastest. Although individual trials may produce different results, tlsDMD consistently delivers the most favorable trade-off across experiments.

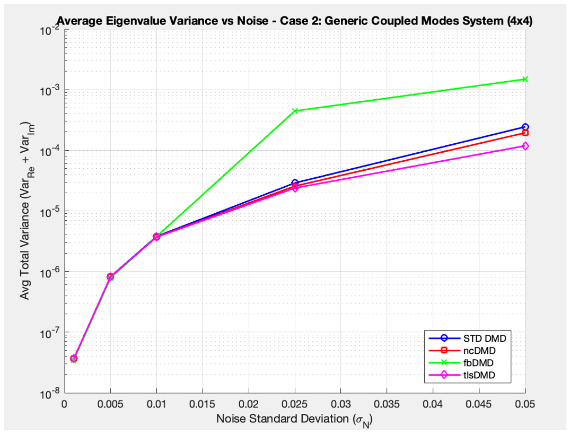
6.3. Case 2 Results



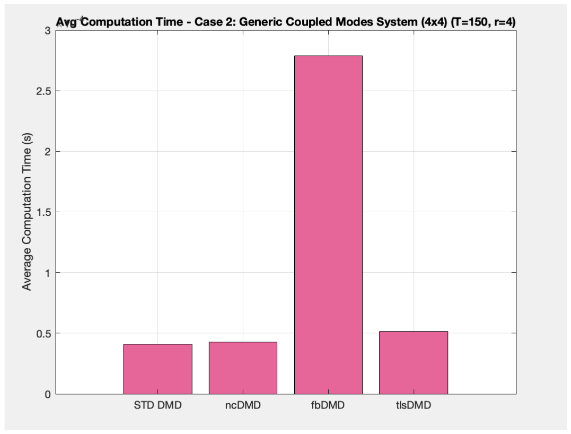
(a) Eigenvalue Recovery



(b) Bias comparison



(c) Variance comparison

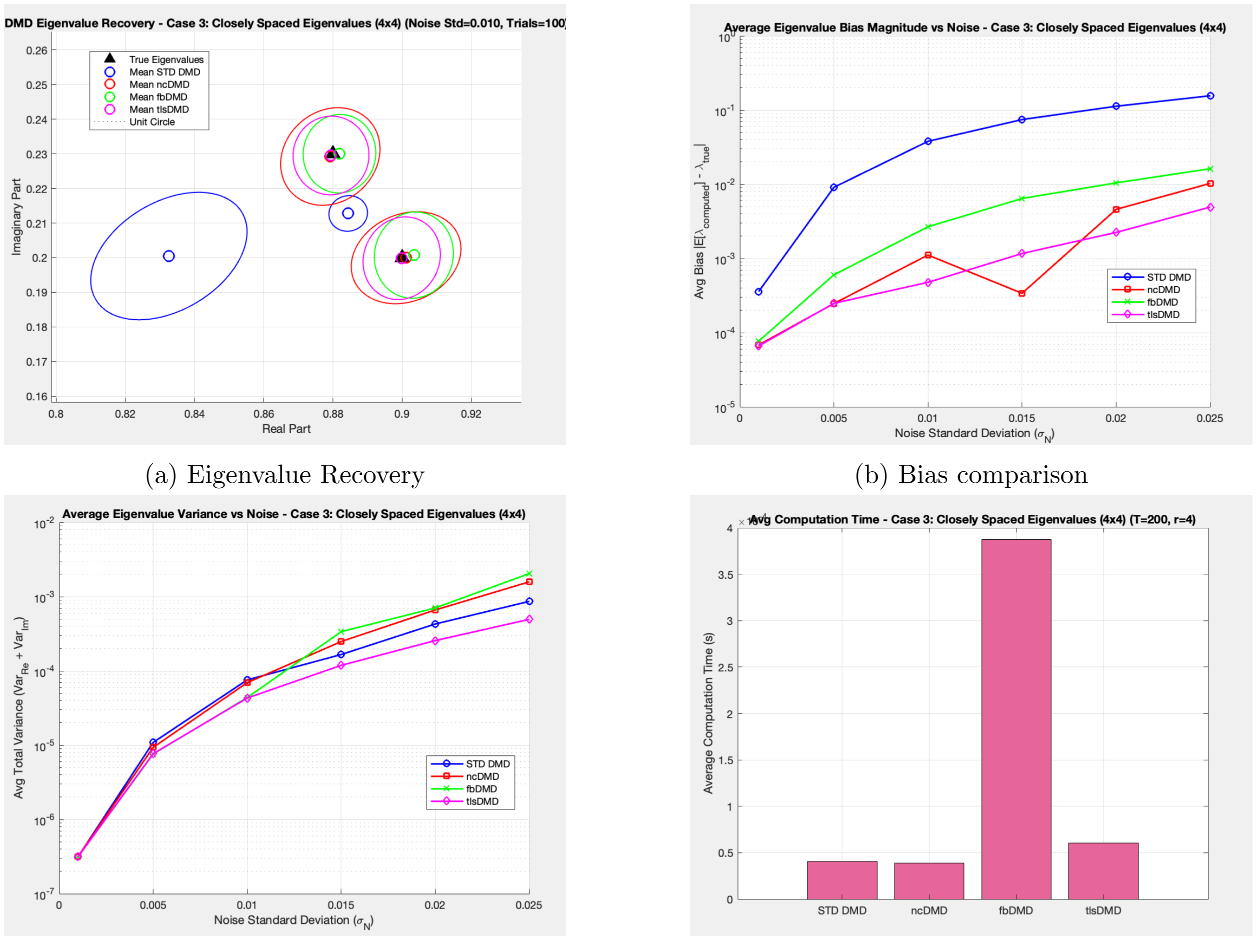


(d) Computation time

Figure 3: Performance metrics for Case 2

At the representative noise level of $\sigma_N = 0.050$, standard DMD produced an average eigenvalue bias of approximately 0.038. Both tlsDMD and ncDMD delivered drastic improvements, achieving mean biases of 0.00026 and 0.00058, respectively. Regarding statistical consistency, tlsDMD again proved most effective, while ncDMD also showed strong stability. By contrast, fbDMD's variance was around 6-12 times higher than that of the other variants. Computational runtimes remained consistent with results from the previous cases. For this system, tlsDMD provides the best performance considering both its excellent bias reduction and lowest variance, with ncDMD a close second.

6.4. Case 3 Results



(a) Eigenvalue Recovery

(b) Bias comparison

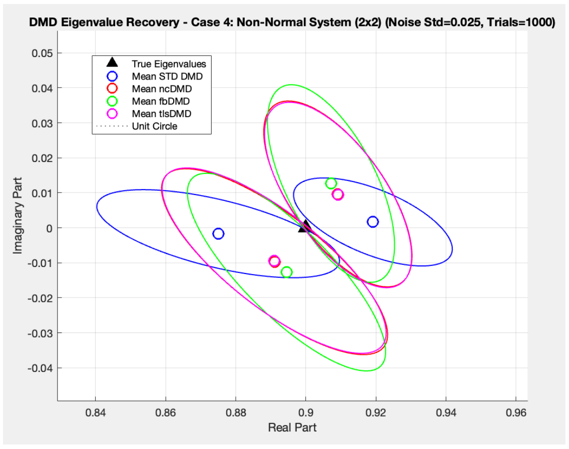
(c) Variance comparison

(d) Computation time

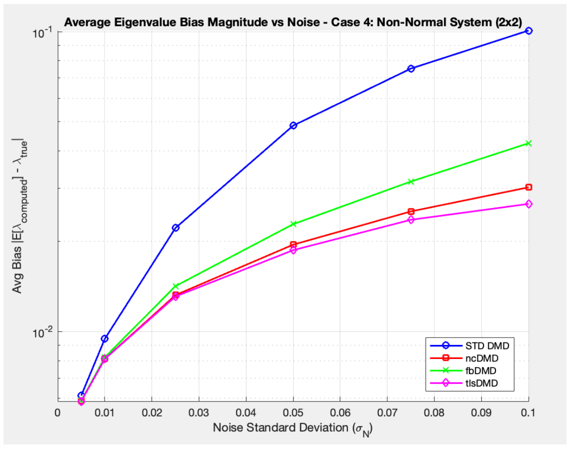
Figure 4: Performance metrics for Case 3

The results at a noise level of $\sigma_N = 0.025$ confirmed that in noisy conditions, accurately distinguishing modes with closely spaced eigenvalues is intrinsically difficult: compared with the well-conditioned system, all methods showed increased average bias and variance. Standard DMD showed a large mean bias of 0.156. Of the noise-robust methods, tlsDMD remains the most accurate with an average bias of 0.0049, followed by ncDMD at 0.010, and then fbDMD at 0.016. In terms of consistency, tlsDMD again achieves the lowest average variance (4.95×10^{-4}), while ncDMD showed markedly higher variance, pointing out to the challenge of being consistent with separating the two close eigenvalue pairs. The computational runtime remained unchanged from previous cases. These results further highlight the robustness of tlsDMD in accuracy and consistency when faced with closely spaced system eigenvalues.

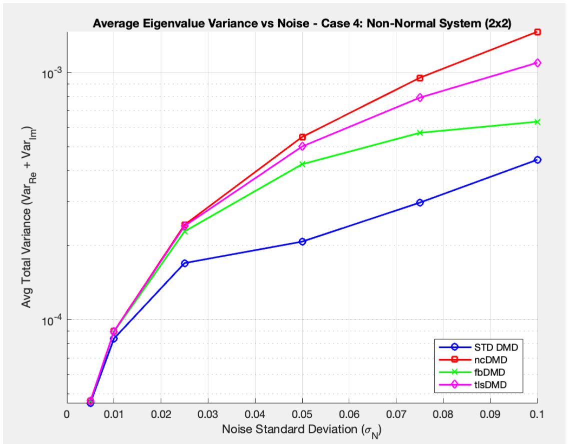
6.5. Case 4 Results



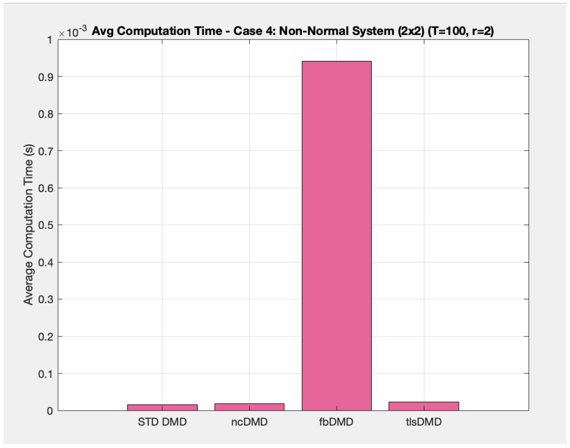
(a) Eigenvalue Recovery



(b) Bias comparison



(c) Variance comparison

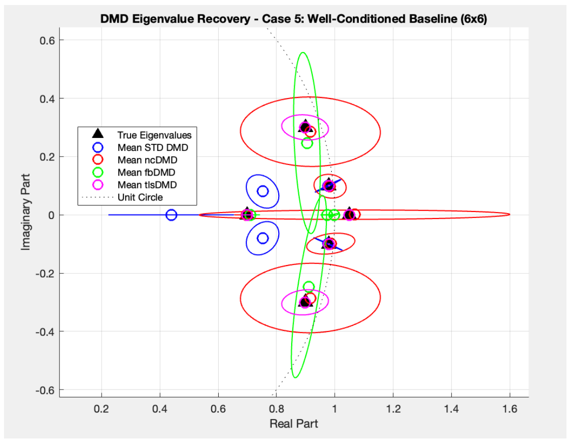


(d) Computation time

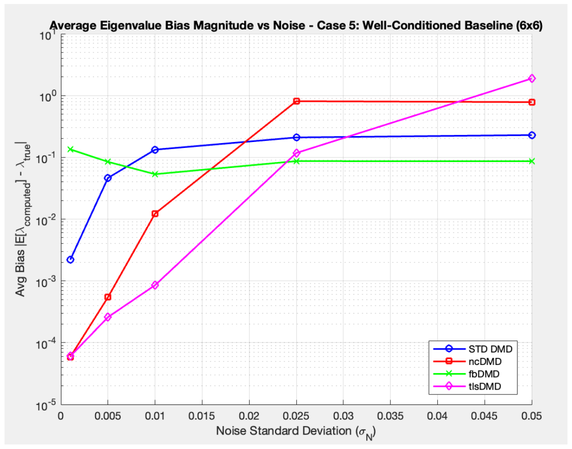
Figure 5: Performance metrics for Case 4

Because the system exhibits strong sensitivity, we ran $N_{\text{trials}} = 1000$ simulations at each noise level to ensure statistically stability. At $\sigma_N = 0.100$, standard DMD showed a large eigenvalue bias (≈ 0.101). All correction methods reduced this bias, with tlsDMD performing best (≈ 0.027), followed by ncDMD and fbDMD. Computational runtimes also remained consistent with earlier cases. However, the variance results revealed a questioning performance hierarchy. Standard DMD exhibited the lowest average total variance (4.44×10^{-4}), followed by fbDMD (6.32×10^{-4}). In contrast, tlsDMD and ncDMD produced eigenvalue estimates with considerably higher variability, suggesting that the mechanisms they use to eliminate bias may amplify noise sensitivity under non-normal, high-noise systems. These results, if implemented correctly, show that simultaneously optimizing for both bias and variance with DMD-based methods when analyzing sensitive systems subject to noise is challenging.

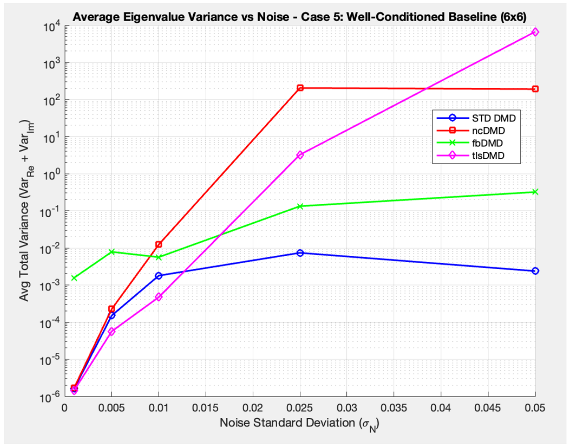
6.6. Case 5 Results



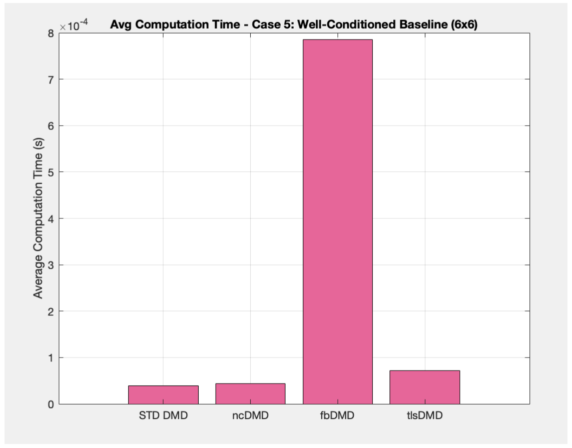
(a) Eigenvalue Recovery



(b) Bias comparison



(c) Variance comparison



(d) Computation time

Figure 6: Performance metrics for Case 5

Initial tests on the 6×6 system at $\sigma_N = 0.050$ yielded severe numerical issues. Standard DMD produced a mean bias of approximately 0.23 with moderate variance ($\approx 2.4 \times 10^{-3}$), and fbDMD reduced the bias to roughly 0.086 at the cost of high variance (≈ 0.32). However, ncDMD and tlsDMD broke down. Contrary to expectations based on previous cases, ncDMD produced an average bias of approximately 0.78 with variance on the order of $\approx 1.9 \times 10^2$. tlsDMD failed even worse with bias around 1.89 and variance greater than 6.6×10^3. These failures likely come from using all $n = 6$ POD modes. Modes associated with tiny singular values, may be dominated by noise and destabilize the correction steps (e.g., Σ_m^{-2} in ncDMD, U_{11}^{-1} in tlsDMD).

7. Discussion

7.1. Interpretation

Our controlled comparison highlights three key findings. First, standard DMD consistently underestimates eigenvalues in the presence of sensor noise—its low computational cost comes at the expense of substantial bias. Second, tlsDMD delivers the most reliable estimates across well-conditioned, coupled-mode, closely spaced, and non-normal systems. Its simultaneous minimization of errors in both X and Y results in low bias and variance without extra runtime cost. Third, ncDMD, which uses exact noise-variance knowledge, matches tlsDMD's bias performance but can amplify variability in small singular directions via its Σ^{-2} correction. fbDMD reduces bias without requiring σ_N^2, but its matrix-square-root and inversion steps introduce numerical fragility and a slowdown ranging from 7 to 20 times in computational speed.

Interestingly, for the non-normal system, the biased but stable estimates of standard DMD were statistically more consistent than any correction method, showing that bias reduction does not guarantee minimized variance under high-noise, sensitive dynamics. Also, in the full-rank 6×6 test, keeping all POD modes caused ncDMD and tlsDMD to fail—stressing the need for rank truncation based on singular-value decay.

7.2. Limitations

Rank Truncation Assumption: To focus on the intrinsic noise-correction behavior of ncDMD, fbDMD, and tlsDMD, we set the truncation rank to the full system dimension in all cases. While this simplifies comparisons, it differs from common practice in high-dimensional settings—where choosing $r < n$ via singular-value decay is essential for computational tractability, noise filtering, and numerical stability. This is likely the reason why in Case 5 ncDMD and tlsDMD failed. Therefore, while our choice helps to understand the algorithms deeper, the results may not fully reflect the performance when employed in real system scenarios.

fbDMD Implementation Nuances: Our fbDMD follows Dawson *et al.*'s projected approach [3]. Although this avoids requiring noise-variance estimates, it introduces numerical fragility related to the matrix inversion and square root steps, which could be worsened by a small mismatch in the basis caused from using separate POD projections for the forward ($\tilde{A}_f$) and backward ($\tilde{B}_b$) operators. fbDMD performed worse than all the other methods, particularly for the systems involving coupled modes. As an alternative, we implemented an unprojected fbDMD by computing full $n \times n$ operators via pseudoinverses, but we did not evaluate it due to time constraints.

ncDMD Noise Variance Requirement: An important factor influencing the application of ncDMD is its dependence on an estimate of the sensor noise variance σ_N^2 required for its correction term. In our synthetic tests, we supplied ncDMD with the true noise variance, which is an ideal scenario. In real-world data σ_N^2 is typically unknown and must be estimated from the available noisy measurements, leading to errors that may lower ncDMD's performance. We did not study how sensitive ncDMD is to misestimated noise levels. Therefore, the performance of the variant in our test cases should be interpreted with the recognition that it is based on this ideal assumption.

Eigenvalue-Only Metrics: Our evaluation primarily relies on eigenvalue bias and variance. While we get information regarding the temporal characteristics of the identified dynamics, they represent only one component of the complete DMD model. We did not quantify mode-accuracy, data reconstruction precision, or prediction error, which could reveal additional trade-offs among the algorithms.

8. Conclusion

We have presented a quantitative comparison of three noise-robust Dynamic Mode Decomposition algorithms—ncDMD, fbDMD, and tlsDMD—against standard DMD on synthetic linear systems with known dynamics. Our test cases ranged from simple 2×2 examples to higher-dimensional systems featuring well-separated, coupled, closely spaced, and non-normal eigenvalue structures, all under varying levels of additive Gaussian noise. Our results show the following:

- **Standard DMD** is fast but suffers from large eigenvalue bias, limiting its use in noisy settings.
- **tlsDMD** consistently offers the best balance of bias reduction, low variance, and computational efficiency for well-behaved to moderately complex systems.
- **ncDMD** competes with tlsDMD when the true noise variance σ_N^2 is known, but can display increased variance due to its Σ^{-2} correction term.
- **fbDMD** requires no noise-variance input yet experiences higher variance and a considerable runtime cost from its matrix-square-root operations.

Several interesting questions arise for future work. Our results show that further investigation into the fbDMD algorithm is urgent. We must explore the impact of specific implementation choices, such as the initial data projection strategy as in [3] versus alternatives from other fbDMD literature, on its accuracy, robustness to varying system complexities, and computational scaling. Besides implementation, it would be valuable to see how ncDMD, fbDMD, and tlsDMD perform when faced with process noise, and to explore their effectiveness on data from truly nonlinear systems. Beyond eigenvalue accuracy, we can gain a deeper understanding by assessing how well each method recovers the spatial modes and how accurately the resulting models can predict future system states. Furthermore, investigating how the algorithms interact with optimal, data-driven strategies for selecting the truncation rank would be a significant step towards their broader practical application.

References

[1] K. K. Chen, J. H. Tu, and C. W. Rowley. Variants of dynamic mode decomposition: boundary condition, Koopman, and Fourier analyses. *Journal of Nonlinear Science*, 22(6):887–915, December 2012.

[2] Matthew J. Colbrook. The multiverse of dynamic mode decomposition algorithms, 2023.

[3] Scott T. M. Dawson, Maziar S. Hemati, Matthew O. Williams, and Clarence W. Rowley. Characterizing and correcting for the effect of sensor noise in the dynamic mode decomposition. *Experiments in Fluids*, 57:42, 2016.

[4] PETER J. SCHMID. Dynamic mode decomposition of numerical and experimental data. *Journal of Fluid Mechanics*, 656:5–28, 2010.

[5] Zhaoxia Wu, Steven L Brunton, and Shai Revzen. Challenges in dynamic mode decomposition. *Journal of the Royal Society Interface*, 18(185):20210686, 2021.

A. MATLAB Code

A.1. Data Generation Function (generate_data.m)

```
function [X_clean, Y_clean, X_noisy, Y_noisy, A_true, t_vals] =
   generate_data(A_true, x0, T, noise_stddev)
% Generates synthetic data for DMD comparison.
%
% Inputs:
%   A_true - The true system matrix (n x n)
%   x0 - The initial condition vector (n x 1)
%   T - The total number of time steps (including x0)
%   noise_stddev - Standard deviation of the noise to add
%
% Outputs:
%   X_clean - Clean data matrix
%   Y_clean - Clean data matrix
%   X_noisy - Noisy data matrix X_clean + noise (n x T-1)
%   Y_noisy - Noisy data matrix Y_clean + noise (n x T-1)
%   A_true - The true system matrix
%   t_vals - Time steps vector
    n = size(A_true, 1); % Dimension of the system
    if size(x0, 1) ~= n || size(x0, 2) ~= 1
        error('Initial condition x0 must be a column vector of size n x
           1.');
    end
    if T < 2
        error('Number of time steps T must be at least 2.');
    end
    %  Generate Clean Data
    all_states_clean = zeros(n, T);
    all_states_clean(:, 1) = x0;
    for k = 1:(T - 1)
        all_states_clean(:, k + 1) = A_true * all_states_clean(:, k);
    end
    % Create X and Y matrices
    X_clean = all_states_clean(:, 1:(T - 1));
    Y_clean = all_states_clean(:, 2:T);
    t_vals = 0:(T-1);
    % Add Noise
    noise_variance = noise_stddev^2;

    all_states_noisy = zeros(n, T);
    for k = 1:T
         noise_vector = sqrt(noise_variance) * randn(n, 1); % Generate
            noise for step k
         all_states_noisy(:,k) = all_states_clean(:,k) + noise_vector;
    end

    X_noisy = all_states_noisy(:, 1:(T - 1));
    Y_noisy = all_states_noisy(:, 2:T);

end
```

A.2. Helper Function (run_dmd_trial.m)

```
function [eig_storage_matched, time_storage] = run_dmd_trial(method_func,
   args, eig_storage_matched, time_storage, trial_idx, r,
   true_eigenvalues_sorted)
% Runs a single trial for a given DMD method,matches computed eigenvalues
   to true eigenvalues, and stores results.
%
% Inputs:
%   method_func - Function handle to the DMD method
%   args - Cell array of arguments for the method_func
%   eig_storage_matched - storing matched eigenvalues across trials
%   time_storage - storing computation times across trials
%   trial_idx - current trial number
%   r - rank used for DMD
%   true_eigenvalues_sorted - Sorted true eigenvalues
% Outputs:
%   eig_storage_matched - Updated matched eigenvalue storage matrix
%   time_storage - Updated time storage vector

    % Initialize with NaN
    eig_for_this_trial = complex(nan(r,1));
    time_taken_this_trial = NaN;

    persistent warned_about_greedy_matching;
    if isempty(warned_about_greedy_matching)
        warned_about_greedy_matching = containers.Map('KeyType','char','
           ValueType','logical');
    end
    method_name_str = func2str(method_func);

    try
        % Execute DMD Method and Time It
        state_sqrtm_singular = warning('off','MATLAB:sqrtm:SingularMatrix
           ');
        state_sqrtm_nonpos = warning('off','MATLAB:sqrtm:NonPosRealEig');

        tic;
```

```
% Assume DMD functions return eigenvalues as their first output
eig_comp_raw = method_func(args{:});
time_taken_this_trial = toc;

warning(state_sqrtm_singular);
warning(state_sqrtm_nonpos);

% Process Computed Eigenvalues
if isempty(eig_comp_raw) || ~isnumeric(eig_comp_raw) || all(isnan(
   eig_comp_raw(:)))
    warning('Method %s trial %d: Returned empty, non-numeric, or
       all NaN eigenvalues. Storing NaNs.', method_name_str,
       trial_idx);
else
    eig_comp_finite = eig_comp_raw(isfinite(eig_comp_raw(:))); %
       Keep only finite values, ensure column

    if length(eig_comp_finite) < r
        warning('Method %s trial %d: Returned %d finite
           eigenvalues, expected %d. Padding with NaNs for
           matching.', ...
                 method_name_str, trial_idx, length(eig_comp_finite
                    ), r);
        eig_comp_padded = complex(nan(r,1));
        eig_comp_padded(1:length(eig_comp_finite)) =
           eig_comp_finite;
        eig_comp = eig_comp_padded;
    elseif length(eig_comp_finite) > r
        % If more than r eigenvalues, select r based on proximity
           to unit circle
        warning('Method %s trial %d: Returned %d finite
           eigenvalues, expected %d. Selecting %d closest to unit
           circle.', ...
                 method_name_str, trial_idx, length(eig_comp_finite
                    ), r, r);
        [~, mag_sort_idx] = sort(abs(abs(eig_comp_finite) - 1));
        eig_comp = eig_comp_finite(mag_sort_idx(1:r));
    else
        eig_comp = eig_comp_finite;
    end

    eig_comp = eig_comp(:);

    % Eigenvalue Matching to True Eigenvalues
    if r > 0 && ~isempty(eig_comp) && ~all(isnan(eig_comp))
        temp_matched_eig_for_trial = complex(nan(r,1));
```

```
if license('test', 'Statistics_Toolbox') && exist('
   matchpairs', 'file')
    % Use matchpairs
    cost_matrix = abs(bsxfun(@minus, eig_comp,
       true_eigenvalues_sorted.')).^2;

    % Determine a reasonable unmatched cost
    max_cost_val = max(cost_matrix(:));
    unmatched_cost_threshold = max(1e-6, max_cost_val *
       1.1);
    if isempty(unmatched_cost_threshold) || isnan(
       unmatched_cost_threshold) || ~isfinite(
       unmatched_cost_threshold)
        unmatched_cost_threshold = 1e8;
    end

    matches = matchpairs(cost_matrix,
       unmatched_cost_threshold);

    for k_match = 1:size(matches,1)
        comp_idx_matched = matches(k_match, 1);
        true_idx_matched = matches(k_match, 2);
        if comp_idx_matched <= length(eig_comp)
            temp_matched_eig_for_trial(true_idx_matched) =
               eig_comp(comp_idx_matched);
        end
    end
    eig_for_this_trial = temp_matched_eig_for_trial;
else
    % Fallback: Greedy Matching
    if ~isKey(warned_about_greedy_matching,
       method_name_str) || ~warned_about_greedy_matching(
       method_name_str)
        warning('run_dmd_trial:matchpairs', 'Using greedy
           eigenvalue matching for %s. Not good for
           closely spaced eigenvalues.', method_name_str);
        warned_about_greedy_matching(method_name_str) =
           true;
    end

    cost_matrix = abs(bsxfun(@minus, eig_comp,
       true_eigenvalues_sorted.')).^2;

    num_comp_eigs_to_match = length(eig_comp);
    available_comp_idx = 1:num_comp_eigs_to_match;
    available_true_idx = 1:r;
```

```
                    for k_match_iter = 1:min(num_comp_eigs_to_match, r)
                        if isempty(available_comp_idx) || isempty(
                            available_true_idx)
                             break;
                        end

                        sub_cost_matrix = cost_matrix(available_comp_idx,
                            available_true_idx);

                        [~, linear_idx] = min(sub_cost_matrix(:));
                        [row_rel_idx, col_rel_idx] = ind2sub(size(
                            sub_cost_matrix), linear_idx);

                        actual_comp_idx_to_match = available_comp_idx(
                            row_rel_idx);
                        actual_true_idx_to_match_to = available_true_idx(
                            col_rel_idx);

                        temp_matched_eig_for_trial(
                            actual_true_idx_to_match_to) = eig_comp(
                            actual_comp_idx_to_match);

                        available_comp_idx(row_rel_idx) = [];
                        available_true_idx(col_rel_idx) = [];
                    end
                    eig_for_this_trial = temp_matched_eig_for_trial;
                end
            else

            end
        end

    catch ME
        warning('Error during DMD method %s trial %d: %s. Storing NaNs.',
            method_name_str, trial_idx, ME.message);
        eig_for_this_trial = complex(nan(r,1)); % Reinitialize to ensure
            correct size on error
        time_taken_this_trial = NaN;
    end

    % Store results for this trial
    eig_storage_matched(:, trial_idx) = eig_for_this_trial;
    time_storage(trial_idx) = time_taken_this_trial;
end
```

A.3. Standard DMD (standard_dmd.m)

```
function [eigenvalues, modes, Atilde, Ur, Sr, Vr] = standard_dmd(X, Y, r)
% Computes the standard/Exact Dynamic Mode Decomposition.
% Based on Algorithm 1 in Dawson et al. (2016) / Exact DMD notes from
   class

% Inputs:
%   X - Data matrix
%   Y - Data matrix
%   r - Rank for SVD truncation
%
% Outputs:
%   eigenvalues - Computed DMD eigenvalues
%   modes       - Computed DMD modes
%   Atilde      - Low-dimensional DMD operator
%   Ur, Sr, Vr  - Truncated SVD components of X

    % Compute SVD of X
    [U, S, V] = svd(X, 'econ');

    % Truncate SVD (USE FULL RANK FOR NOW, IF I HAVE TIME ILL EXPERIMENT
       WITH THIS)
    if nargin < 3 || r == 0 || r > size(U, 2)
        r = size(U, 2);
    end
    Ur = U(:, 1:r);
    Sr = S(1:r, 1:r);
    Vr = V(:, 1:r);

    % Check condition number of Sr
    if cond(Sr) > 1e10
        warning('standard_dmd:illConditioned', 'Singular value matrix is
           ill-conditioned. Results may be inaccurate.');
    end

    % Compute low-dimensional operator
    Atilde = Ur' * Y * Vr / Sr;

    % Compute eigenvalues and eigenvectors
    [eigvec_Atilde, D] = eig(Atilde);
    eigenvalues = diag(D);

    % Exact DMD mode calculation
    modes_exact = zeros(size(X,1), r, 'like', 1i); % Initialize complex
       modes matrix
    YVr_invSr = Y * Vr / Sr;
    for i = 1:r
        if abs(eigenvalues(i)) > 1e-12 % Threshold to avoid division by
           near-zero
             modes_exact(:, i) = YVr_invSr * eigvec_Atilde(:, i) /
                 eigenvalues(i);
        else
             % Handle zero eigenvalues if necessary
             modes_exact(:, i) = zeros(size(X,1),1);
        end
    end
    modes = modes_exact; % Use Exact DMD modes


end
```

A.4. ncDMD (nc_dmd.m)

```

function [eigenvalues, modes, Atilde_nc, Atilde_m] = nc_dmd(X_noisy,
   Y_noisy, r, noise_variance)
% Computes noise-corrected DMD (ncDMD). Based on Algorithm 2 in Dawson et
   al. (2016).
% Inputs:
%   X_noisy - Noisy data matrix X
%   Y_noisy - Noisy data matrix Y
%   r - Rank for SVD truncation
%   noise_variance - Estimate of the variance (sigma_N^2) of sensor noise
   per element

% Outputs:
%   eigenvalues - ncDMD eigenvalues
%   modes-  ncDMD modes
%   Atilde_nc - Corrected low-dimensional operator
%   Atilde_m - Original biased low-dimensional operator

    if nargin < 4
        error('nc_dmd requires noise_variance as an input.');
    end
    if r <= 0
        error('nc_dmd requires r > 0.');
    end

    m = size(X_noisy, 2); % snapshot pairs

    % Compute SVD of X_noisy and truncate
    [U, S, V] = svd(X_noisy, 'econ');
    if r > size(U, 2)
        warning('nc_dmd:rankTooHigh', 'Requested rank higher than
           available rank. Using max rank %d.', size(U,2));
        r = size(U, 2);
    end
    Ur = U(:, 1:r);
    Sr = S(1:r, 1:r);
    Vr = V(:, 1:r);
    Sm = Sr; % sigma matrix

    % Check condition number of Sm
    if cond(Sm) > 1e10
        warning('nc_dmd:illConditioned', 'Sm is ill-conditioned. Results
           may be inaccurate.');
    end

    Atilde_m = Ur' * Y_noisy * Vr / Sm;

    % Compute the corrected operator
    % Atilde_nc = Atilde_m * inv(eye(r) - m * noise_variance * inv(Sm^2));
        % Less stable
    correction_matrix = eye(r) - m * noise_variance * (Sm \ eye(r)) / Sm;
    if cond(correction_matrix) > 1e10
         warning('nc_dmd:correctionUnstable', 'Correction matrix is ill-
            conditioned. Bias correction might be inaccurate.');
    end
    Atilde_nc = Atilde_m / correction_matrix;

    % Compute eigenvalues and eigenvectors
    [eigvec_Atilde_nc, D] = eig(Atilde_nc);
    eigenvalues = diag(D);

    % Compute ncDMD modes
    modes = zeros(size(X_noisy,1), r, 'like', 1i);
    YVr_invSm = Y_noisy * Vr / Sm; % Use original SVD components
    for i = 1:r
        if abs(eigenvalues(i)) > 1e-12
            modes(:, i) = YVr_invSm * eigvec_Atilde_nc(:, i) / eigenvalues
               (i);
        else
            modes(:, i) = zeros(size(X_noisy,1),1); % Or NaN
        end
    end
end
```

A.5. fbDMD (fb_dmd.m)

```
function [eigenvalues, modes, Atilde_fb, Atilde_f, Atilde_b_inv] = fb_dmd(
   X_noisy, Y_noisy, r)
% Computes forward-backward DMD (fbDMD). Based on Algorithm 3 in Dawson et
    al. (2016).
%
% Inputs:
%   X_noisy - Noisy data matrix X
%   Y_noisy - Noisy data matrix Y
%   r - Rank for SVD truncation
%
% Outputs:
%   eigenvalues - fbDMD eigenvalues
%   modes - fbDMD modes
%   Atilde_fb - Combined forward-backward low-dimensional operator
%   Atilde_f - Forward low-dimensional operator
%   Atilde_b_inv - Inverse of the backward low-dimensional operator

    if nargin < 3 || r <= 0
        error('fb_dmd requires a truncation rank r > 0.');
    end
    n = size(X_noisy, 1);

    % Compute forward operator
    [U_f, S_f, V_f] = svd(X_noisy, 'econ');
    if r > size(U_f, 2)
         warning('fb_dmd:rankTooHighFwd', 'Requested rank r is higher than
             forward rank. Using max rank %d.', size(U_f,2));
         r = size(U_f, 2);
    end
    Ur_f = U_f(:, 1:r);
    Sr_f = S_f(1:r, 1:r);
    Vr_f = V_f(:, 1:r);

    if cond(Sr_f) > 1e10
        warning('fb_dmd:illConditionedFwd', 'Sr_f is ill-conditioned.');
    end
    Atilde_f = Ur_f' * Y_noisy * Vr_f / Sr_f;


    % Compute backward operator Btilde_b
    [U_b, S_b, V_b] = svd(Y_noisy, 'econ');
     if r > size(U_b, 2)
         warning('fb_dmd:rankTooHighBwd', 'Requested rank r is higher than
             backward rank. Using max rank %d.', size(U_b,2));
         r = min(r, size(U_b,2));

         Ur_f = U_f(:, 1:r);
         Sr_f = S_f(1:r, 1:r);
         Vr_f = V_f(:, 1:r);
         Atilde_f = Ur_f' * Y_noisy * Vr_f / Sr_f;
    end
    Ur_b = U_b(:, 1:r);
    Sr_b = S_b(1:r, 1:r);
    Vr_b = V_b(:, 1:r);

    if cond(Sr_b) > 1e10
        warning('fb_dmd:illConditionedBwd', ' Sr_b is ill-conditioned.');
    end

    Btilde_b = Ur_b' * X_noisy * Vr_b / Sr_b; % Backward operator


    if cond(Btilde_b) > 1e10
        warning('fb_dmd:illConditionedBwdInv', 'Backward operator Btilde_b
         is ill-conditioned. Inverse may be inaccurate.');
        Atilde_b_inv = pinv(Btilde_b); % Use pseudoinverse if ill-
           conditioned
    else
        Atilde_b_inv = inv(Btilde_b);
    end

    % Compute combined operator using matrix square root
    Atilde_prod = Atilde_f * Atilde_b_inv;
    [Atilde_fb, sqrt_info] = sqrtm(Atilde_prod); % Complex matrix square
       root
    if sqrt_info ~= 0
       warning('fb_dmd:sqrtmFailed', 'Matrix square root calculation
          failed or gave non-primary root. Info: %d', sqrt_info);
    end
    % Ensure the result is the principal square root if possible, or
       handle ambiguity.
    % One way: choose the sqrt closest to Atilde_f or inv(Btilde_b).
    % sqrtm should return the principal root.

    % Compute eigenvalues and eigenvectors
    [eigvec_Atilde_fb, D] = eig(Atilde_fb);
    eigenvalues = diag(D);

    % Compute fbDMD modes
    %modes = Ur_f * eigvec_Atilde_fb;
```

```
    modes_exact = zeros(size(X_noisy,1), r, 'like', 1i);
    YVr_invSr = Y_noisy * Vr_f / Sr_f;
    for i = 1:r
        if abs(eigenvalues(i)) > 1e-12
            modes_exact(:, i) = YVr_invSr * eigvec_Atilde_fb(:, i) /
               eigenvalues(i);
        else
            modes_exact(:, i) = zeros(size(X_noisy,1),1);
        end
    end
    modes = modes_exact; % Use Exact DMD modes based on forward projection

end
```

A.6. fbDMD Unprojected (fb_dmd_unprojected.m)

```
    % NOT USED. I DIDN'T HAVE ENOUGH TIME TO INCORPORATE INTO PAPER.

function [eigenvalues, modes, A_fb, A_f, A_b_inv] = fb_dmd_unprojected(
   X_noisy, Y_noisy)
% Computes fbDMD WITHOUT POD projection.
% Calculates the full n x n operators directly using pseudoinverses.
% Inputs:
%   X_noisy - Noisy data matrix X
%   Y_noisy - Noisy data matrix Y
% Outputs:
%   eigenvalues - fbDMD eigenvalues
%   modes - fbDMD modes
%   A_fb - Combined forward-backward full n x n operator estimate
%   A_f - Forward full n x n operator estimate
%   A_b_inv - Inverse of the backward full n x n operator estimate

    n = size(X_noisy, 1);
    m = size(X_noisy, 2);



    % Initialize output
    eigenvalues = complex(nan(n,1));
    modes = complex(nan(n,n));
    A_fb = complex(nan(n,n));
    A_f = complex(nan(n,n));
    A_b_inv = complex(nan(n,n));

    try
        % Compute Forward Operator
        tic_af = tic;
```

```
pinv_X = pinv(X_noisy); % Moore-Penrose Pseudoinverse
A_f = Y_noisy * pinv_X;
fprintf('  Computed A_f (%.3f sec)\n', toc(tic_af));

% Compute Backward Operator
tic_b = tic;
pinv_Y = pinv(Y_noisy);
B = X_noisy * pinv_Y;
fprintf('  Computed B (%.3f sec)\n', toc(tic_b));

% Compute Inverse of Backward Operator
tic_binv = tic;
cond_B = cond(B);
if cond_B > 1/eps
    warning('fb_dmd_unprojected:illConditionedB', ...
            'Full Backward operator B is ill-conditioned (cond=%.2
               e). Using pseudoinverse.', cond_B);
    A_b_inv = pinv(B);
else
    try
        A_b_inv = inv(B);
    catch ME_invB
        warning('fb_dmd_unprojected:invBError', ...
                'inv(B) failed (cond=%.2e): %s. Using
                   pseudoinverse.', cond_B, ME_invB.message);
        A_b_inv = pinv(B);
    end
end
fprintf('  Computed A_b_inv (using %s) (%.3f sec)\n', ...
    MException.last.identifier == "" && cond_B <= 1/eps, 'inv', '
       pinv', toc(tic_binv)); % Simple logic based on last warning
       /error might be fragile


% Compute Combined Operator using Matrix Square Root
A_prod = A_f * A_b_inv;
tic_sqrtm = tic;
try
    state_sqrtm_singular = warning('off','MATLAB:sqrtm:
       SingularMatrix');
    state_sqrtm_nonpos = warning('off','MATLAB:sqrtm:NonPosRealEig
       ');
    [A_fb, sqrt_info] = sqrtm(A_prod); % Complex matrix square
       root
    warning(state_sqrtm_singular);
    warning(state_sqrtm_nonpos);
    if sqrt_info ~= 0
       warning('fb_dmd_unprojected:sqrtmFailed', ...
               'Matrix square root calculation failed or gave non-
                  primary root. Info: %d. Results might be
                  inaccurate.', sqrt_info);
    end
catch ME_sqrtm
    warning('fb_dmd_unprojected:sqrtmError', 'sqrtm(A_f * A_b_inv)
        failed: %s. Setting A_fb to NaN.', ME_sqrtm.message);
    A_fb = complex(nan(n,n)); % Set A_fb to NaN if sqrtm fails
end
fprintf('  Computed A_fb (%.3f sec)\n', toc(tic_sqrtm));

% Compute Eigenvalues and Modes
% For the full operator, the eigenvectors are the modes
if ~any(isnan(A_fb(:))) % Proceed only if A_fb calculation was
   successful
    tic_eig = tic;
    try
        [eigvecs, D] = eig(A_fb, 'vector');
        eigenvalues = D;

        modes = eigvecs;

        % Sort eigenvalues and modes consistently
        try
            [~, sort_idx] = sortrows([imag(eigenvalues), real(
               eigenvalues)]);
            eigenvalues = eigenvalues(sort_idx);
            modes = modes(:, sort_idx);
        catch ME_sort
             warning('fb_dmd_unprojected:SortError', 'Could not
                sort eigenvalues/modes: %s', ME_sort.message);
        end
        fprintf('  Computed Eigenvalues/Modes of A_fb (%.3f sec)\n
           ', toc(tic_eig));

    catch ME_eig
        warning('fb_dmd_unprojected:EigError', 'Eigendecomposition
            of A_fb failed: %s. Returning NaNs.', ME_eig.message);
        eigenvalues = complex(nan(n,1));
        modes = complex(nan(n,n));
    end
else
     warning('fb_dmd_unprojected:NoEig', 'Skipping
        eigendecomposition because A_fb calculation failed.');
end
```

```
    catch ME_main
        warning('fb_dmd_unprojected:MainError', 'Error during unprojected
           fbDMD: %s. Returning NaNs.', ME_main.message);
        eigenvalues = complex(nan(n,1));
        modes = complex(nan(n,n));
        A_fb = complex(nan(n,n));
        A_f = complex(nan(n,n));
        A_b_inv = complex(nan(n,n));
    end
    fprintf('Unprojected fbDMD finished.\n');
end
```

A.7. tlsDMD (tls_dmd.m)

```
function [eigenvalues, modes, Atilde_tls] = tls_dmd(X_noisy, Y_noisy, r)
% Computes total least-squares DMD (tlsDMD). Based on Algorithm 4 in
   Dawson et al. (2016).
% Inputs:
%   X_noisy - Noisy data matrix X
%   Y_noisy - Noisy data matrix Y
%   r - Rank for projection (<m/2 based on the paper)
% Outputs:
%   eigenvalues - tlsDMD eigenvalues
%   modes - tlsDMD modes
%   Atilde_tls - tlsDMD low-dimensional operator

    n = size(X_noisy, 1);
    m = size(X_noisy, 2);

    if nargin < 3 || r <= 0
        error('tls_dmd requires a truncation rank r > 0.');
    end

    % Project onto r POD modes of X_noisy
    [U_proj, S_proj, V_proj] = svd(X_noisy, 'econ');
    if r > size(U_proj, 2)
        warning('tls_dmd:rankTooHighProj', 'Requested rank r is higher
           than rank of X. Using max rank %d.', size(U_proj,2));
        r = size(U_proj, 2);
    end
    Ur_proj = U_proj(:, 1:r);
    Xtilde = diag(diag(S_proj(1:r, 1:r))) * V_proj(:,1:r)';
    Ytilde = Ur_proj' * Y_noisy;
    % Form augmented matrix and take its SVD
    Ztilde = [Xtilde; Ytilde];
    [Uz, Sz, Vz] = svd(Ztilde, 'econ');

    % Partition Uz
    if size(Uz, 2) < r
        error('tls_dmd:rankTooLowTLS', 'Rank of augmented matrix Ztilde (%
           d) is less than required rank r (%d).', size(Uz,2), r);
    end

    Uz_r = Uz(:, 1:r);
    U11 = Uz_r(1:r, 1:r);
    U21 = Uz_r((r+1):(2*r), 1:r);

    % Compute tlsDMD operator
    if cond(U11) > 1e10
        warning('tls_dmd:illConditionedU11', 'Matrix U11 is ill-
           conditioned. TLS result may be inaccurate.');
        Atilde_tls = U21 * pinv(U11);
    else
        Atilde_tls = U21 / U11;
    end

    % Compute eigenvalues and eigenvectors
    [eigvec_Atilde_tls, D] = eig(Atilde_tls);
    eigenvalues = diag(D);

    % Compute tlsDMD modes
    % Lift eigenvectors using the initial projection basis
    modes = Ur_proj * eigvec_Atilde_tls;

end
```

A.8. Ellipse Function for Eigenvaluen Visualization (calculate_ellipse.m)

```
function [X, Y] = calculate_ellipse(mean_point, covariance_matrix,
   confidence_level)
% Calculates points on a ellipse for 2D data.
%
% Inputs:
%   mean_point - coordinates of the center
%   covariance_matrix - 2x2 covariance matrix of the data
%   confidence_level - Confidence level
%
% Outputs:
```

```
%   X, Y - Coordinates of points on the ellipse boundary

    if nargin < 3
        confidence_level = 0.95;
    end

    [eigvec, eigval_diag] = eig(covariance_matrix);

    % ensure eigenvalues non-negative
    eigval = max(diag(eigval_diag), 0);

    % threshold to scale the ellipse at confidence lvl
    chi2_val = chi2inv(confidence_level, 2);

    a = sqrt(chi2_val * eigval(1));
    b = sqrt(chi2_val * eigval(2));

    theta = linspace(0, 2*pi, 100);
    ellipse_points_std_basis = [a * cos(theta); b * sin(theta)];

    % Rotate points according to eigenvectors
    rotated_points = eigvec * ellipse_points_std_basis;

    % Translate points to center at the mean
    X = mean_point(1) + rotated_points(1, :);
    Y = mean_point(2) + rotated_points(2, :);

end
```

A.9. Example Main Simulation Script (runme_case0.m)

```
    clear; close all; clc;
rng('default'); % For reproducibility (Note: I did have this off when
   analyzing.)

%% Define System (Case 0: Simple Case)

A_true = [0.9, -0.4;
          0.2, 0.8];
n = size(A_true, 1);
x0 = [1; 0];

T = 100;
r = n;  % trancation rank

% Parameters for Noise
```

```
noise_levels_stddev = [0.005, 0.01, 0.025, 0.05, 0.075, 0.1];
num_noise_levels = length(noise_levels_stddev);
num_trials = 100;

% Check tlsDMD condition
if r >= (T-1)/2 && T > 1
     fprintf('Note: Rank r=%d is not strictly less than m/2 = %f .\n', r,
        (T-1)/2);
end

%% Store Results
true_eigenvalues = eig(A_true);
[~, sort_idx_true] = sort(imag(true_eigenvalues));
true_eigenvalues_sorted = true_eigenvalues(sort_idx_true);

% Results for each noise level
all_eig_std = cell(num_noise_levels, 1);
all_eig_nc  = cell(num_noise_levels, 1);
all_eig_fb  = cell(num_noise_levels, 1);
all_eig_tls = cell(num_noise_levels, 1);

% Summary statistics for each noise level
mean_bias_mag_std = zeros(r, num_noise_levels);
mean_bias_mag_nc  = zeros(r, num_noise_levels);
mean_bias_mag_fb  = zeros(r, num_noise_levels);
mean_bias_mag_tls = zeros(r, num_noise_levels);

total_variance_std = zeros(r, num_noise_levels);
total_variance_nc  = zeros(r, num_noise_levels);
total_variance_fb  = zeros(r, num_noise_levels);
total_variance_tls = zeros(r, num_noise_levels);

mean_time_std = zeros(num_noise_levels, 1);
mean_time_nc  = zeros(num_noise_levels, 1);
mean_time_fb  = zeros(num_noise_levels, 1);
mean_time_tls = zeros(num_noise_levels, 1);

% Ellipse data
ellipse_noise_idx = ceil(num_noise_levels / 2);
ellipse_means = struct('std', [], 'nc', [], 'fb', [], 'tls', []);
ellipse_covs = struct('std', [], 'nc', [], 'fb', [], 'tls', []);


%% Run Simulation Loops
fprintf('Running %d trials for each of %d noise levels...\n', num_trials,
   num_noise_levels);
```

```
for nl = 1:num_noise_levels
    current_noise_stddev = noise_levels_stddev(nl);
    current_noise_variance = current_noise_stddev^2;
    fprintf(' Running Noise Level %.3f (%d/%d)\n', current_noise_stddev,
       nl, num_noise_levels);

    eigenvalues_std_nl = complex(zeros(r, num_trials));
    eigenvalues_nc_nl  = complex(zeros(r, num_trials));
    eigenvalues_fb_nl  = complex(zeros(r, num_trials));
    eigenvalues_tls_nl = complex(zeros(r, num_trials));

    time_std_nl = zeros(num_trials, 1);
    time_nc_nl  = zeros(num_trials, 1);
    time_fb_nl  = zeros(num_trials, 1);
    time_tls_nl = zeros(num_trials, 1);

     for trial = 1:num_trials
        % Generate new noisy data for this trial
        [~, ~, X_noisy, Y_noisy] = generate_data(A_true, x0, T,
           current_noise_stddev);

        %Run the methods
        [eigenvalues_std_nl, time_std_nl] = run_dmd_trial(@standard_dmd, {
           X_noisy, Y_noisy, r}, eigenvalues_std_nl, time_std_nl, trial, r
           , true_eigenvalues_sorted);
        if current_noise_variance > eps
             [eigenvalues_nc_nl, time_nc_nl] = run_dmd_trial(@nc_dmd, {
                X_noisy, Y_noisy, r, current_noise_variance},
                eigenvalues_nc_nl, time_nc_nl, trial, r,
                true_eigenvalues_sorted);
        else
             eigenvalues_nc_nl(:, trial) = eigenvalues_std_nl(:,trial);
             time_nc_nl(trial) = time_std_nl(trial);
        end
        [eigenvalues_fb_nl, time_fb_nl]   = run_dmd_trial(@fb_dmd, {
           X_noisy, Y_noisy, r}, eigenvalues_fb_nl, time_fb_nl, trial, r,
           true_eigenvalues_sorted);
        [eigenvalues_tls_nl, time_tls_nl] = run_dmd_trial(@tls_dmd, {
           X_noisy, Y_noisy, r}, eigenvalues_tls_nl, time_tls_nl, trial, r
           , true_eigenvalues_sorted);
     end
     for trial = 1:num_trials
        % Generate new noisy data for this trial
        [~, ~, X_noisy, Y_noisy] = generate_data(A_true, x0, T,
           current_noise_stddev);

        %Run the methods
        [eigenvalues_std_nl, time_std_nl] = run_dmd_trial(@standard_dmd, {
           X_noisy, Y_noisy, r}, eigenvalues_std_nl, time_std_nl, trial, r
           , true_eigenvalues_sorted);
        if current_noise_variance > eps
             [eigenvalues_nc_nl, time_nc_nl] = run_dmd_trial(@nc_dmd, {
                X_noisy, Y_noisy, r, current_noise_variance},
                eigenvalues_nc_nl, time_nc_nl, trial, r,
                true_eigenvalues_sorted);
        else
             eigenvalues_nc_nl(:, trial) = eigenvalues_std_nl(:,trial);
             time_nc_nl(trial) = time_std_nl(trial);
        end
        [eigenvalues_fb_nl, time_fb_nl]   = run_dmd_trial(@fb_dmd, {
           X_noisy, Y_noisy, r}, eigenvalues_fb_nl, time_fb_nl, trial, r,
           true_eigenvalues_sorted);
        [eigenvalues_tls_nl, time_tls_nl] = run_dmd_trial(@tls_dmd, {
           X_noisy, Y_noisy, r}, eigenvalues_tls_nl, time_tls_nl, trial, r
           , true_eigenvalues_sorted);
     end

    % Store all eigenvalues for this noise level
    all_eig_std{nl} = eigenvalues_std_nl;
    all_eig_nc{nl}  = eigenvalues_nc_nl;
    all_eig_fb{nl}  = eigenvalues_fb_nl;
    all_eig_tls{nl} = eigenvalues_tls_nl;

    %  Calculate and Store Summary Statistics
    for i = 1:r

        mean_eig_std = mean(eigenvalues_std_nl(i,:), 2, 'omitnan');
        mean_eig_nc  = mean(eigenvalues_nc_nl(i,:), 2, 'omitnan');
        mean_eig_fb  = mean(eigenvalues_fb_nl(i,:), 2, 'omitnan');
        mean_eig_tls = mean(eigenvalues_tls_nl(i,:), 2, 'omitnan');

        mean_bias_mag_std(i, nl) = abs(mean_eig_std -
           true_eigenvalues_sorted(i));
        mean_bias_mag_nc(i, nl)  = abs(mean_eig_nc  -
           true_eigenvalues_sorted(i));
        mean_bias_mag_fb(i, nl)  = abs(mean_eig_fb  -
           true_eigenvalues_sorted(i));
        mean_bias_mag_tls(i, nl) = abs(mean_eig_tls -
           true_eigenvalues_sorted(i));

        % Calculate Total Variance
        total_variance_std(i, nl) = var(real(eigenvalues_std_nl(i,:)), 0,
           'omitnan') + var(imag(eigenvalues_std_nl(i,:)), 0, 'omitnan');
        total_variance_nc(i, nl)  = var(real(eigenvalues_nc_nl(i,:)), 0, '
           omitnan') + var(imag(eigenvalues_nc_nl(i,:)), 0, 'omitnan');
        total_variance_fb(i, nl)  = var(real(eigenvalues_fb_nl(i,:)), 0, '
           omitnan') + var(imag(eigenvalues_fb_nl(i,:)), 0, 'omitnan');
        total_variance_tls(i, nl) = var(real(eigenvalues_tls_nl(i,:)), 0,
           'omitnan') + var(imag(eigenvalues_tls_nl(i,:)), 0, 'omitnan');

        % Store ellipse data
        if nl == ellipse_noise_idx
             if ~isnan(mean_eig_std)
                 ellipse_means.std = [ellipse_means.std; [real(
                    mean_eig_std), imag(mean_eig_std)]];
                 ellipse_covs.std  = [ellipse_covs.std; {cov(real(
                    eigenvalues_std_nl(i,:))', imag(eigenvalues_std_nl(i
                    ,:))', 'omitrows')}];
             end
             if ~isnan(mean_eig_nc)
                 ellipse_means.nc  = [ellipse_means.nc; [real(mean_eig_nc)
                    , imag(mean_eig_nc)]];
                 ellipse_covs.nc   = [ellipse_covs.nc; {cov(real(
                    eigenvalues_nc_nl(i,:))', imag(eigenvalues_nc_nl(i,:))
                    ', 'omitrows')}];
             end
             if ~isnan(mean_eig_fb)
                 ellipse_means.fb  = [ellipse_means.fb; [real(mean_eig_fb)
                    , imag(mean_eig_fb)]];
                 ellipse_covs.fb   = [ellipse_covs.fb; {cov(real(
                    eigenvalues_fb_nl(i,:))', imag(eigenvalues_fb_nl(i,:))
                    ', 'omitrows')}];
             end
              if ~isnan(mean_eig_tls)
                 ellipse_means.tls = [ellipse_means.tls; [real(
                    mean_eig_tls), imag(mean_eig_tls)]];
                 ellipse_covs.tls  = [ellipse_covs.tls; {cov(real(
                    eigenvalues_tls_nl(i,:))', imag(eigenvalues_tls_nl(i
                    ,:))', 'omitrows')}];
             end
        end
    end

    % Calculate Mean Computation Times
    mean_time_std(nl) = mean(time_std_nl, 'omitnan');
    mean_time_nc(nl)  = mean(time_nc_nl, 'omitnan');
    mean_time_fb(nl)  = mean(time_fb_nl, 'omitnan');
    mean_time_tls(nl) = mean(time_tls_nl, 'omitnan');

end
```

```
fprintf('Simulations complete.\n');



%% Plots
case_name = sprintf('Case 0: Illustrative Simple Case (%dx%d)', n, n);

% Plot 1: Eigenvalue Plot ( NOTE: I SEARCHED ONLINE HOW TO MAKE THIS PLOT)
ellipse_noise_val = noise_levels_stddev(ellipse_noise_idx);
figure('Name', sprintf('Eigenvalue Recovery - %s (Noise Std=%.3f)',
   case_name, ellipse_noise_val));
hold on;
plot(real(true_eigenvalues_sorted), imag(true_eigenvalues_sorted), 'k^', '
   MarkerSize', 12, 'MarkerFaceColor', 'k', 'DisplayName', 'True
   Eigenvalues');
method_colors = {'b', 'r', 'g', 'm'};
method_names = {'std', 'nc', 'fb', 'tls'};
method_display_names = {'STD DMD', 'ncDMD', 'fbDMD', 'tlsDMD'};
for method_idx = 1:length(method_names)
    method = method_names{method_idx};
    means_data = ellipse_means.(method);
    covs_data  = ellipse_covs.(method);

    if ~isempty(means_data)
        plot(means_data(:,1), means_data(:,2), [method_colors{method_idx}
           'o'], 'MarkerSize', 10, 'LineWidth', 1.5, 'DisplayName', ['Mean
            ' method_display_names{method_idx}]);
        for i = 1:size(means_data, 1)
            if i <= length(covs_data) && ~isempty(covs_data{i}) && ~any(
               isnan(covs_data{i}(:))) && all(isfinite(covs_data{i}(:)))
               && ~any(isnan(means_data(i,:))) && isreal(covs_data{i}) &&
               issymmetric(covs_data{i}) && all(size(covs_data{i}) == [2
               2])
                 try
                    [ell_x, ell_y] = calculate_ellipse(means_data(i,:),
                       covs_data{i});
                    plot(ell_x, ell_y, [method_colors{method_idx} '-'], '
                       LineWidth', 1, 'HandleVisibility','off');
                 catch ME_ellipse
                    warning('Could not plot ellipse for %s Eig %d: %s',
                       method_display_names{method_idx}, i, ME_ellipse.
                       message);
                 end
            end
        end
    end
end
```

```
theta = linspace(0, 2*pi, 100);
plot(cos(theta), sin(theta), 'k:', 'DisplayName', 'Unit Circle');
title(sprintf('DMD Eigenvalue Recovery - %s (Noise Std=%.3f, Trials=%d)',
   case_name, ellipse_noise_val, num_trials));
xlabel('Real Part'); ylabel('Imaginary Part');
legend('show', 'Location', 'best'); axis equal; grid on;
hold off;

% Plot 2: Bias Magnitude vs Noise Lvl (Averaged over eigenvalues)
avg_bias_std = mean(mean_bias_mag_std, 1, 'omitnan');
avg_bias_nc  = mean(mean_bias_mag_nc,  1, 'omitnan');
avg_bias_fb  = mean(mean_bias_mag_fb,  1, 'omitnan');
avg_bias_tls = mean(mean_bias_mag_tls, 1, 'omitnan');

figure('Name', ['Avg Bias Magnitude vs Noise - ' case_name]);
hold on;
plot_styles = {'bo-', 'rs-', 'gx-', 'md-'};
plot(noise_levels_stddev, avg_bias_std, plot_styles{1}, 'LineWidth', 1.5,
   'DisplayName', 'STD DMD');
plot(noise_levels_stddev, avg_bias_nc,  plot_styles{2}, 'LineWidth', 1.5,
   'DisplayName', 'ncDMD');
plot(noise_levels_stddev, avg_bias_fb,  plot_styles{3}, 'LineWidth', 1.5,
   'DisplayName', 'fbDMD');
plot(noise_levels_stddev, avg_bias_tls, plot_styles{4}, 'LineWidth', 1.5,
   'DisplayName', 'tlsDMD');
set(gca, 'YScale', 'log');
title(['Average Eigenvalue Bias Magnitude vs Noise - ' case_name]);
xlabel('Noise Standard Deviation (\sigma_N)'); ylabel('Avg Bias |E[\
   lambda_{computed}] - \lambda_{true}|');
legend('show', 'Location', 'best'); grid on;
hold off;

% Plot 3: Variance vs Noise Level (Averaged over eigenvalues)
avg_var_std = mean(total_variance_std, 1, 'omitnan');
avg_var_nc  = mean(total_variance_nc,  1, 'omitnan');
avg_var_fb  = mean(total_variance_fb,  1, 'omitnan');
avg_var_tls = mean(total_variance_tls, 1, 'omitnan');

figure('Name', ['Avg Variance vs Noise - ' case_name]);
hold on;
plot(noise_levels_stddev, avg_var_std, plot_styles{1}, 'LineWidth', 1.5, '
   DisplayName', 'STD DMD');
plot(noise_levels_stddev, avg_var_nc,  plot_styles{2}, 'LineWidth', 1.5, '
   DisplayName', 'ncDMD');
plot(noise_levels_stddev, avg_var_fb,  plot_styles{3}, 'LineWidth', 1.5, '
   DisplayName', 'fbDMD');
```

```
plot(noise_levels_stddev, avg_var_tls, plot_styles{4}, 'LineWidth', 1.5, '
   DisplayName', 'tlsDMD');
set(gca, 'YScale', 'log');
title(['Average Eigenvalue Variance vs Noise - ' case_name]);
xlabel('Noise Standard Deviation (\sigma_N)'); ylabel('Avg Total Variance
   (Var_{Re} + Var_{Im})');
legend('show', 'Location', 'best'); grid on;
hold off;

% Plot 4: Computation Times
overall_mean_time_std = mean(mean_time_std, 'omitnan');
overall_mean_time_nc  = mean(mean_time_nc, 'omitnan');
overall_mean_time_fb  = mean(mean_time_fb, 'omitnan');
overall_mean_time_tls = mean(mean_time_tls, 'omitnan');

figure('Name', ['Computation Time - ' case_name]);
times = [overall_mean_time_std, overall_mean_time_nc, overall_mean_time_fb
   , overall_mean_time_tls];
bar_labels = {'STD DMD', 'ncDMD', 'fbDMD', 'tlsDMD'};

b = bar(times);
b.FaceColor = [0.9, 0.4, 0.6];

set(gca, 'xticklabel', bar_labels);
ylabel('Average Computation Time (s)');
title(sprintf('Avg Computation Time - %s (T=%d, r=%d)',case_name, T, r));
grid on;



%% Summary Statements
fprintf('\n========== Summary: %s ==========\n', case_name);
highest_noise_idx = num_noise_levels;
highest_noise_val = noise_levels_stddev(highest_noise_idx);

fprintf('\n--- Bias Correction Performance (at Noise Std = %.3f, Averaged
   over %d Eigenvalues) ---\n', highest_noise_val, r);
avg_bias_high_noise = [avg_bias_std(highest_noise_idx); avg_bias_nc(
   highest_noise_idx); avg_bias_fb(highest_noise_idx); avg_bias_tls(
   highest_noise_idx)]';
fprintf('Method      | Avg Bias  \n');
fprintf('------------|-----------\n');
for mi = 1:length(method_display_names)
    fprintf('%-11s | %.5f\n', method_display_names{mi},
       avg_bias_high_noise(mi));
end
[min_avg_bias, best_bias_overall_idx] = min(avg_bias_high_noise);
```

```
if ~isnan(min_avg_bias)
    fprintf('Overall Best Avg Bias Correction: %s (Avg Bias: %.5f)\n',
       method_display_names{best_bias_overall_idx}, min_avg_bias);
else
    fprintf('Could not determine overall best bias correction due to NaNs
       .\n');
end


fprintf('\n--- Variance Performance (at Noise Std = %.3f, Averaged over %d
    Eigenvalues) ---\n', highest_noise_val, r);
avg_var_high_noise = [avg_var_std(highest_noise_idx); avg_var_nc(
   highest_noise_idx); avg_var_fb(highest_noise_idx); avg_var_tls(
   highest_noise_idx)]';
fprintf('Method      | Avg Var   \n');
fprintf('------------|-----------\n');
for mi = 1:length(method_display_names)
    fprintf('%-11s | %.5e\n', method_display_names{mi}, avg_var_high_noise
       (mi));
end
[min_avg_var, best_var_overall_idx] = min(avg_var_high_noise);
if ~isnan(min_avg_var)
    fprintf('Overall Lowest Avg Variance: %s (Avg Var: %.5e)\n',
       method_display_names{best_var_overall_idx}, min_avg_var);
else
    fprintf('Could not determine overall lowest variance due to NaNs.\n');
end


fprintf('\n--- Computational Efficiency ---\n');
fprintf('Std DMD Avg Time:  %.6f s\n', overall_mean_time_std);
fprintf('ncDMD Avg Time:  %.6f s\n', overall_mean_time_nc);
fprintf('fbDMD Avg Time:  %.6f s\n', overall_mean_time_fb);
fprintf('tlsDMD Avg Time: %.6f s\n', overall_mean_time_tls);

times_valid_idx = ~isnan(times) & times > 0;
fastest_method_idx = NaN;
if any(times_valid_idx)
    min_valid_time = min(times(times_valid_idx));
    time_ratios = times ./ min_valid_time;
    fprintf('Relative Speeds (approx): STD=%.1fx, nc=%.1fx, fb=%.1fx, tls
       =%.1fx \n', ... %lower is faster
            time_ratios(1), time_ratios(2), time_ratios(3), time_ratios(4)
               );

    temp_times = times; temp_times(~times_valid_idx) = inf;
    [~, fastest_method_idx_val] = min(temp_times);
    if ~isinf(fastest_method_idx_val)
        fastest_method_idx = fastest_method_idx_val;
        fprintf('Fastest method overall: %s\n', method_display_names{
           fastest_method_idx});
    else
        fprintf('Could not determine fastest method.\n');
    end
else
    fprintf('Could not determine fastest method.\n');
end

fprintf('\n--- Overall Balance (for %s at Noise Std = %.3f) ---\n',
   case_name, highest_noise_val);
if ~isnan(best_bias_overall_idx) && ~isnan(best_var_overall_idx) && ~isnan
   (fastest_method_idx)
    best_bias_method = method_display_names{best_bias_overall_idx};
    best_var_method = method_display_names{best_var_overall_idx};
    fastest_method = method_display_names{fastest_method_idx};
    fprintf('- Best Avg Bias: %s\n', best_bias_method);
    fprintf('- Lowest Avg Var: %s\n', best_var_method);
    fprintf('- Fastest: %s\n', fastest_method);
else
     fprintf('- Could not determine overall balance due to missing results
        .\n');
end
```