Data Driven Fluid Mechanics

Yash Golani, Aditya Kumar
Mechanical Engineering, IIT Jodhpur

Abstract

This is a report deliverable for our Fluid Mechanics Course project, i.e Data-driven fluid mechanics and Machine learning Techniques to accelerate Fluid Mechanics Simulations. The content of this report is inspired largely from numerous resources, each cited at the end of the report

Repository location
https://colab.research.google.com/drive/13FlKuUHOpRPIAf5JG6YdNyNSF3qkgwP2?usp=sharing

Context This report has been produced in LATEX. This data is produced as apart of Fluid Mechanics Project which is our understanding from numerous research papers, and other sources.

1 Review of Machine Learning in Fluids

By incorporating machine learning (ML) methods into the field of fluid mechanics , a significant change in perspective can be observed. This can be a means of providing innovative approaches to solve the complex issues associated with flow modeling, dynamics analysis, and computational fluid dynamics (CFD). This review of ours provides a brief overview of influential publications written by Brunton et al., Brenner et al., and Nocket et al. and several others, where we have investigated the complex technical aspects of using machine learning techniques in fluid mechanics calculations and dynamics.

Papers:

Data-Driven methods in fluid dynamics: Sparse classification from experimental data
Machine Learning for Fluid Mechanics
Data-Driven Fluid Mechanics.

The papers we did a study on provide a comprehensive analysis of the technical difficulties that arise in fluid dynamics. They highlight the importance of accurately measuring the underlying physical mechanisms and the intricate, complex, highly varying –multi-scale characteristics of fluid flows. This study argues in favor of machine learning models that place emphasis on interpretability, generalizability, and robustness, so as to ensure that new technologies and methods can implemented in this domain to meet the demands of the current industrial needs. It highlights the significance of employing cross-validation techniques to address the issue of overfitting. The study investigates a range of machine learning approaches specifically designed for the field of fluid dynamics. These techniques encompass dimensionality reduction tools such as Proper Orthogonal Decomposition (POD) sing Single Value Decomposition (SVD) and Principal Component Analysis (PCA), which facilitate the extraction of crucial flow characteristics and patterns from the empirical data at hand and extensive simulations, hence enabling the development of simplified models and improving computational effectiveness. The Singular Value Decomposition (SVD) is a factorization method for matrices, which is widely used in various applications, including data compression, dimensionality reduction, and solving linear systems.

Given a matrix $A \in R^{m \times n}$, the SVD factorizes A into three matrices as follows:

$$A = U\Sigma V^T$$

where:

- A is the original matrix of size $m \times n$.
- U is an orthogonal matrix of size $m \times m$ containing the left singular vectors.
- Σ is a diagonal matrix of size $m \times n$ containing the singular values.
- V^T is an orthogonal matrix of size $n \times n$ containing the right singular vectors.

The SVD process can be summarized in the following steps:

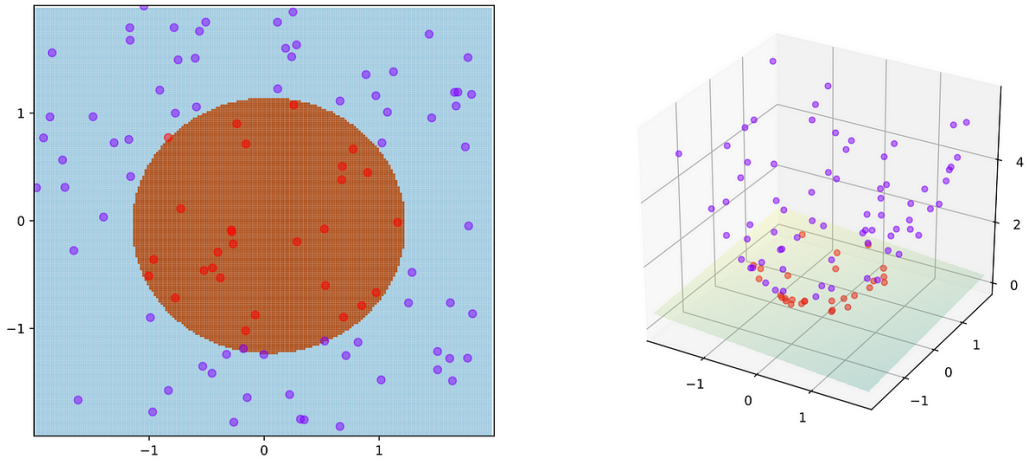
1. Compute the product $A^T A$. This matrix is symmetric and positive semidefinite.
2. Compute the eigenvectors and eigenvalues of $A^T A$. Let the eigenvectors be denoted by v_i and the corresponding eigenvalues be denoted by λ_i.
3. Order the eigenvectors v_i in descending order of eigenvalues λ_i.
4. Compute the singular values $\sigma_i = \sqrt{\lambda_i}$.
5. Compute the left singular vectors U as the eigenvectors of AA^T. Normalize each eigenvector to unit length.
6. Compute the right singular vectors V using the relation $Av_i = \sigma_i u_i$, where u_i are the columns of U.
7. Assemble U, Σ, and V to form the SVD of A.

Furthermore, the papers examine the interdependent association between machine learning (ML) and sparse optimization, emphasizing the manner in which sparse algorithms enhance ML models through expedited computations and effective acquisition of sparse inputs. In addition, the research explores super-resolution techniques that aim to enhance resolution and eliminate noise in fluid flow simulations and experiments. This demonstrates the potential of machine learning (ML) in boosting the quality and accuracy of flow field reconstructions.
In the above studies, the papers explore the utilization of machine learning (ML) methods, namely deep learning, for the examination of flow patterns and dynamics in porous media. This field is characterized by intricate fluid behavior. Major recent research highlights the significance of convolutional neural networks (CNNs) in the process of extracting features and recognizing patterns from flow images acquired through experimental and simulation methods. Brenner et al. demonstrate the utilization of data-driven methodologies to elucidate the complex flow dynamics exhibited by porous materials. The work of these papers provides valuable insights into the characterization and optimization of flow for a wide range of applications.
Recent researchers have investigated the convergence of machine learning and computational fluid dynamics , with a specific emphasis on the advancement of precise turbulence models, which are fundamental to the field of fluid mechanics research. It is a field of view that looks and explores for current advancements through incorporation of machine learning techniques, specifically artificial neural networks (ANNs) and support vector machines (SVMs), into conventional computational fluid dynamics (CFD) so that the accuracy of

these procedures can be improved and relate for turbulence prediction and flow management. The papers we have mentioned herewith showcase the utilization of machine learning techniques to enhance turbulence closure models and optimize the simulation of turbulent flows, consequently contributing to the advancement of our comprehension of intricate flow phenomena.
The use of machine learning approaches in fluid mechanics study signifies a significant advancement. It provided advanced methodologies to comprehend the intricacies of fluid flow dynamics and improve computational effectiveness. We know that researchers seek to tackle persistent issues in flow analysis, prediction, and control by utilizing sophisticated machine learning algorithms for feature extraction, pattern identification, and turbulence modeling. As the subject is more researched upon, the integration of machine learning (ML) with fluid mechanics holds the potential to explore novel domains in controlling fluid dynamics for a wide range of applications.

2 New Ideas

1. Support Vector Machines for De-noising Images: Support Vector Machine (SVM) is primarily used for classification tasks. Typically, it is not employed for removing noises from images. SVMs work by finding the optimal hyperplane that separates data points into different classes by projecting them into a higher dimension. We can train the SVM model to identify between clean and noisy images for a given flow, by splitting into train and test data sets. The higher energy pixels can be filtered out. But we are working on the assumption that it is Gaussian Noise. It can be used to classify image patches as noisy or noise-free. However, it won't be as effective as other denoising techniques like Gaussian Blur.



2. HMM(Hidden Markov Models) for predicting fluid flow:A HMM can be used to predict future fluid flow, which is not directly observable but can be inferred from the data at hand. In HMM, we have probability distributions for various eventualities that could happen, based on past data.An adjacency matrix can be made based on the various probabilities of past flows at same positions, and based on that we can produce new flows. It is kind of generative. In our context of fluid flows, we can assume flow patterns as the hidden models, and each such each state has some probability distribu-

tion. However, the limitation rises that if there is some sudden change in flow, we cannot ascertain it by prediction.

We can employ Linear Regression to estimate the various dimensionless quantities such as the Reynold's no., Strouhal no. and Euler's no. An exhaustive dataset can be created for a specific fluid after doing numerous experiments, and that dataset can be trained upon, and finally numerous applications can be found for this.This offers a reliable means of predicting dimensionless parameters.

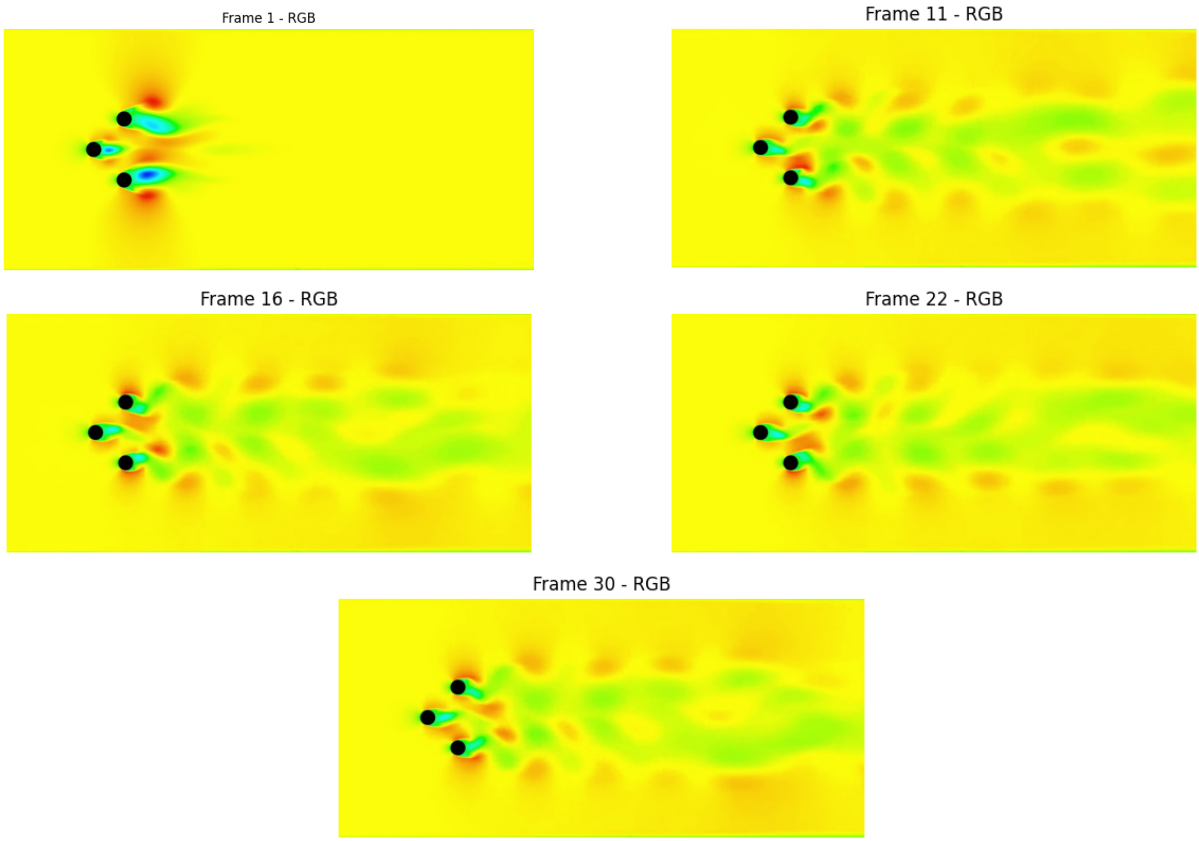
3. Bi-Directional LSTM for Fluid state prediction: Bi - directional LSTMs are mainly used in two layers, the first layer analyzes the data from the past states of the system and the second one from the future states of the system. While analysing Fluid Flows in Computational Fluid Dynamics, LSTM algorithms can be used based on data from previous time intervals, and prediction of future flow patterns can be made. LSTM is generally used for topics such as Sentiment based Stock Price Prediction, Sales Forecasting, etc.

4. Estimation of Dimensionless Nos. obtained in Dimensional analysis of Navier Stokes Equation
We can employ Linear Regression to estimate the various dimensionless quantities such as the Reynold's no., Strouhal no. and Euler's no. An exhaustive dataset can be created for a specific fluid after doing numerous experiments, and that dataset can be trained upon, and finally numerous applications can be found for this.This offers a reliable means of predicting dimensionless parameters.

3 Proper Orthogonal Decomposition

We have performed Proper Orthogonal Decomposition using Reduced Singular Value Decomposition in our Code, along with implementation of a downlampling provision to reduce 3 channel (RGB) images to grayscale, and decrease the size of the images in terms of pixels in appropriate ratios.

3.1 Image Extraction



Next we have performed Image Extraction. For this purpose of study, we have currently chosen the time interval to be 0.5 seconds, which means that from the provided 31 seconds video , we are able to extract 62 images.
First, we imported necessary libraries: os for file operations, cv2 for video processing, and matplotlib.pyplot for displaying images.

Next, we defined a function called frames which takes four parameters: videopath (the path to the video file), folder (the folder where extracted frames will be saved), intervalseconds (the time interval between frame extraction), and colorscale (the color scale for displaying frames, either 'RGB' or 'Grayscale').

def frames(videopath, folder, intervalseconds, colorscale):

Inside the function, checking if the specified folder already exists. If it does, removed it and its contents to start fresh.

if os.path.exists(folder):
shutil.rmtree(folder)
Then, created a VideoCapture object to open the video file specified by videopath.

cap = cv2.VideoCapture(videopath)

Next, created the folder to store the extracted frames, if it doesn't already exist.
os.makedirs(folder, existok=True)

Retrieving the frame rate of the video using cv2.CAP-PROP-FPS and calculating the

frame interval based on the specified interval-seconds.

framerate = cap.get(cv2.CAP-PROP-FPS)
frameinterval = int(framerate * intervalseconds)

Next, Initialized variables to keep track of the total number of frames processed (frame-count) and the number of frames saved (saved-frame-count) and set up the figure size for displaying images using plt.figure.

frame-count = 0
saved-frame-count = 0
plt.figure(figsize=(15, 4))

Then started a loop to read frames from the video using cap.read() Inside the loop, we check if a frame was successfully read. If not, break out of the loop.

while(cap.isOpened())
ret, frame = cap.read()
if ret == False:
break

Incrementing the frame-count by 1 for each frame read.
framecount += 1

Checking if the current frame index is a multiple of the frame-interval. If so, save the frame and update saved=frame-count.

if frame-count frame-interval == 0:
saved-frame-count += 1
frame-filename = os.path.join(folder, f"frame-saved-frame-count:04d.jpg")
cv2.imwrite(frame-filename, frame)

Depending on the specified color-scale, Displaying the frame in either RGB or Grayscale using plt.imshow.
if color-scale == 'RGB':
plt.imshow(cv2.cvtColor(frame, cv2.COLOR-BGR2RGB))
elif color-scale == 'Grayscale':
plt.imshow(cv2.cvtColor(frame, cv2.COLOR-BGR2GRAY), cmap='gray')

plt.title(f"Frame saved-frame-count - color-scale")
plt.show()
Finally, after processing all frames, we released the VideoCapture object and print the total number of frames extracted.

cap.release()
print(f"Total frames extracted: saved-frame-count")
return saved-frame-count

```
Total frames extracted: 62
```

CODE:

```
import os
import cv2
import matplotlib.pyplot as plt

def frames(video_path, folder, interval_seconds,
    color_scale):
     if os.path.exists(folder):
         # If the folder exists, delete it and its contents
         shutil.rmtree(folder)

     cap = cv2.VideoCapture(video_path)
     os.makedirs(folder, exist_ok=True)

     frame_rate = cap.get(cv2.CAP_PROP_FPS)
     frame_interval = int(frame_rate * interval_seconds)
     frame_count = 0
     saved_frame_count = 0
     plt.figure(figsize=(15, 4))

     while(cap.isOpened()):
         ret, frame = cap.read()
         if ret == False:
             break
         frame_count += 1
         if frame_count % frame_interval == 0:
             saved_frame_count += 1
             frame_filename = os.path.join(folder,
                f"frame_{saved_frame_count:04d}.jpg")
             cv2.imwrite(frame_filename, frame)
             if color_scale == 'RGB':
                 plt.imshow(cv2.cvtColor(frame,
                    cv2.COLOR_BGR2RGB))
             elif color_scale == 'Grayscale':
                 plt.imshow(cv2.cvtColor(frame,
                    cv2.COLOR_BGR2GRAY), cmap='gray')
             plt.title(f"Frame {saved_frame_count} -
                {color_scale}")
             plt.axis("off")
             plt.show()

     cap.release()

```

```
    print(f"Total frames extracted: {saved_frame_count}")

    return saved_frame_count
```

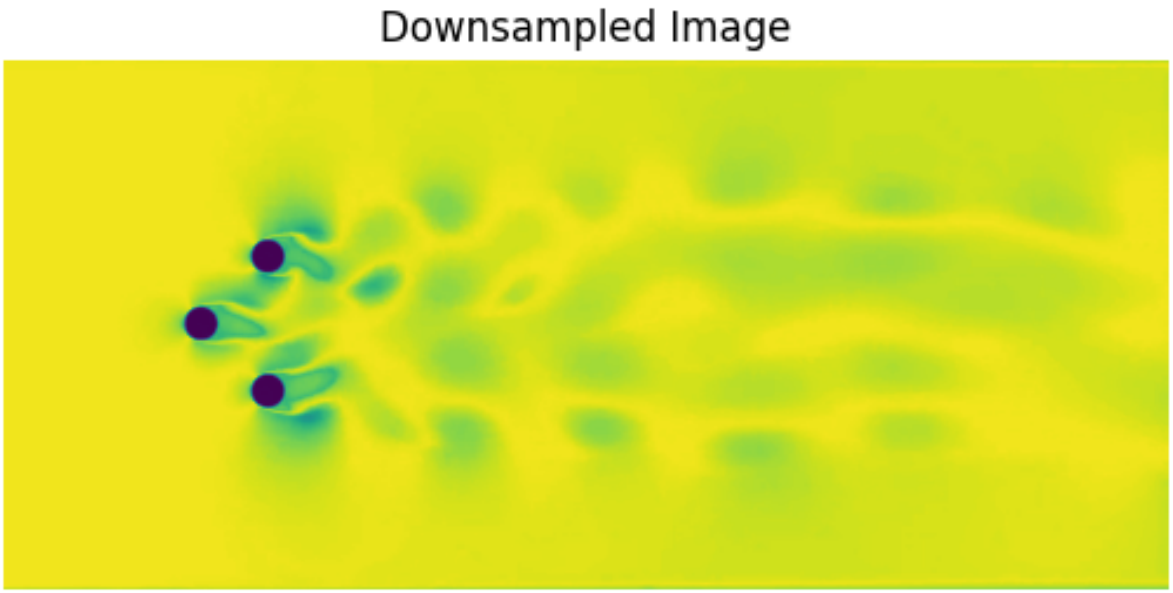
3.2 POD Execution

Downsampling the images: To decrease Computational complexity, we decided to keep provison for downsampling of the pictures from to decrease the size of the pixels in a fixed ratio, so that later on, the time inteval between snapshots can be decreased and a larger number of frames can be considered

Next, the variable that we had defined earlier, "folder-path" is assigned to the directory in which the photos are kept. While processing phase, an empty list which is named as "downsampled-images-gray" was generated to store the downsampled grayscale photos. The new-width was configured to 1558 pixels, while the new-height was set to 708 pixels, which is currently the same as the original pixel size(since we have just kept the provision) . After which a loop runs, which iterates over each file within the designated directory by utilizing the os.listdir(folder-path) function. The complete path to the image was created for each file using the os.path.join(folder-path, filename) function.

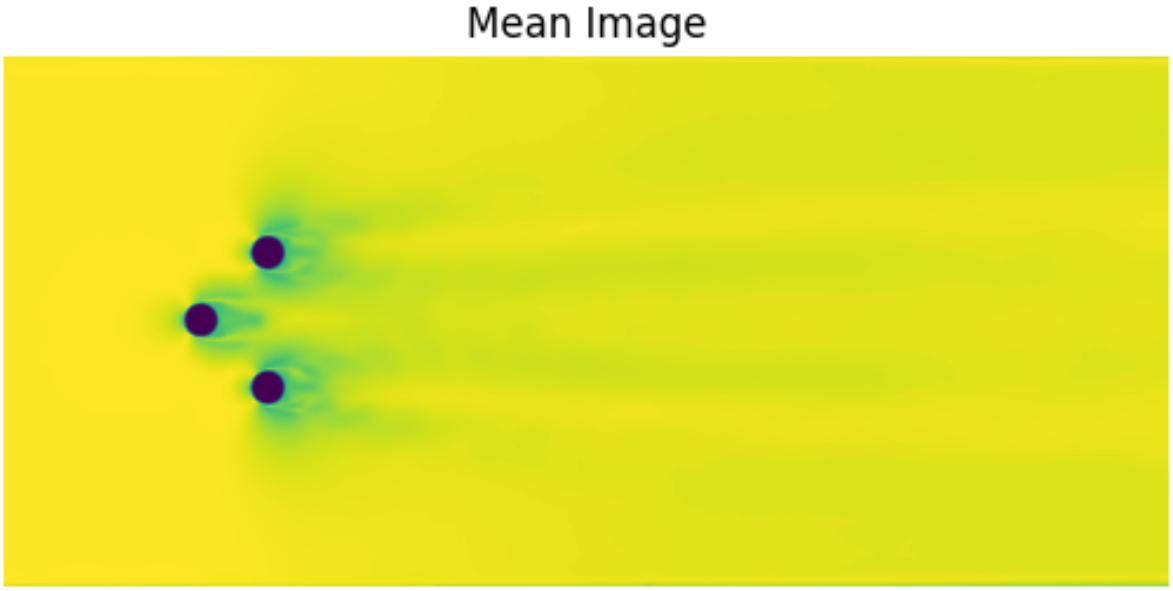
We verified whether the path related to a file by utilizing the os.path function.The function isfile(img-path). If the image is a file, we utilized the cv2.imread(img-path, cv2.IMREAD-GRAYSCALE) function to read the image in grayscale. After the image is read, we have applied the INTER_LINEAR interpolation method to downscale it to the given dimensions using cv2.resize(). The downsampled grayscale image was subsequently added to the list downsampled-images-gray.

Upon completing the image processing, we printed the downsampled photos as can be seen below



Calculating Mean Image Using 'np.mean' from the numpy library we calculated the mean of all image pixels to calculate and display a mean image which will be useful for

us in further computation.



The mean image serves as a crucial point of reference in the field of fluid dynamics research, as it effectively captures the consistent characteristics of fluid flow by calculating the average intensity values across all retrieved frames. It highlights the fundamental flow patterns, eliminating temporary variations. Within the framework of POD, it serves to build the fundamental flow, against which deviations and phenomena of higher energy are quantified and examined, so establishing a fundamental basis for comprehending the prevailing behaviors of the flow.

Flattening and Normalizing the images By transforming the pixel data into vectors and grouping them in a matrix, we created a complete dataset that shows how space changes over time. Utilization of the function "stack-images" does essential job of turning two-dimensional images into one-dimensional rays, or "flattening" them. This first step is very important for turning the complicated image data structure into a numerical grid that can be easily applied to math problems.

Each row of our generated this two-dimensional matrix represents a frame from the fluid flow movie after the arrays have been flattened. Later, the average of all frames, which is the mean picture, is taken away from each individual frame. This step before Ordinary Orthogonal Decomposition (POD) is necessary because it centers the data around the mean, which makes the variations caused by fluid flow dynamics stand out more. The centralization of this matrix is essential for the smooth implementation of POD. Finding the most important modes in a flow using POD requires a center dataset where each row represents a moment in time. Centralization makes sure that the modes obtained later through Singular Value Decomposition (SVD) mostly show the flow's energetic content, without any static background or baseline flow effects. Utilizing SVD on this centered matrix lets us divide the complex flow into its main parts, or modes, arranged by how much energy they contribute to the general flow. Our methodical approach lets us focus on and study the most changing parts of the flow.

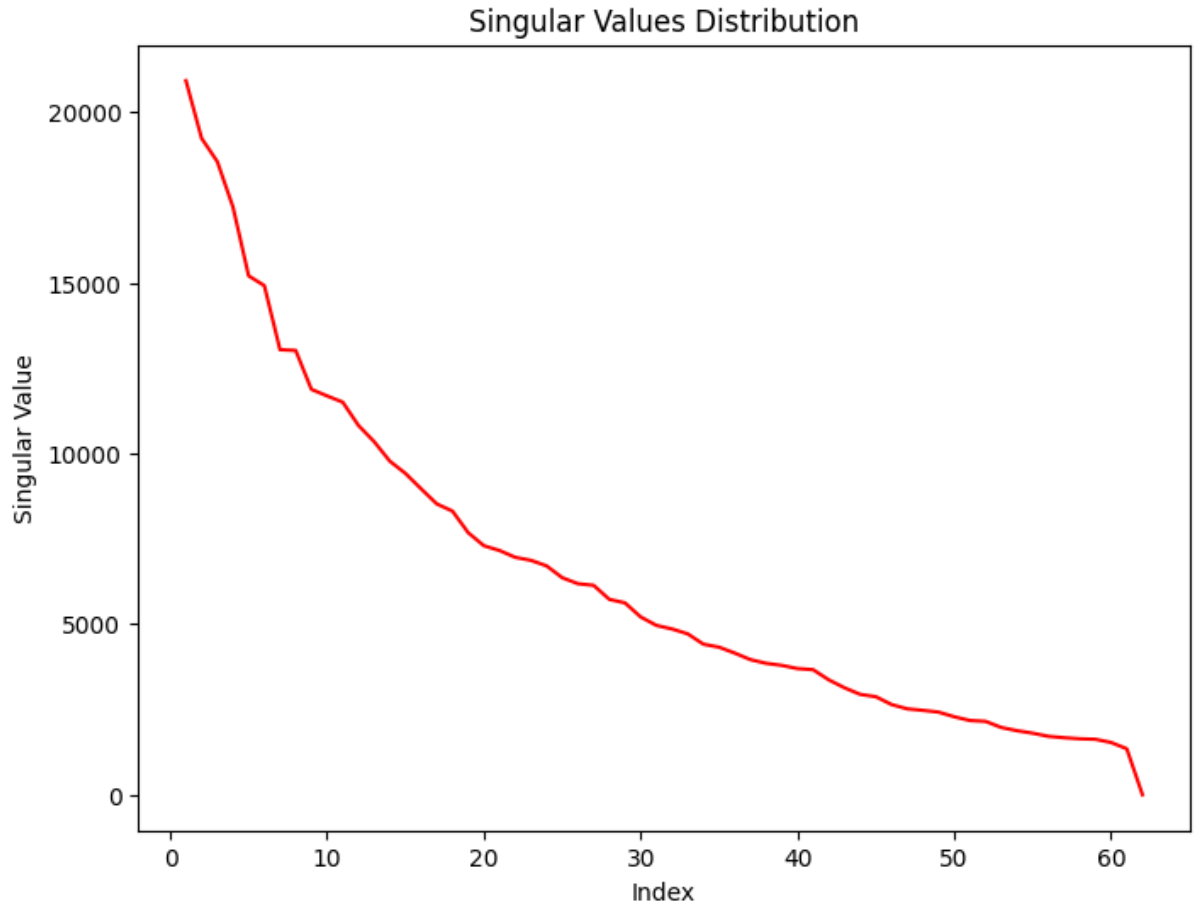
SVD Using Library Function In the next part, we have applied SVD to the centralized image data matrix. The SVD is a mathematical technique that deco mposes the

matrix into three separate matrices which include the spatial modes, singular values, and temporal coefficients of the fluid flow. The `data_matrix` variable, which contains the centralized images, serves as the input for the SVD. This matrix includes the fluctuations of the fluid flow around the mean image. Using mathematical functions from NumPy, the SVD is performed, which decomposes the `data_matrix` into U, S, and VT, corresponding to the spatial modes, singular values, and temporal coefficients, respectively. We have optimized computation using `full_matrices=False`

- We represented the matrix U as the spatial modes of the fluid flow, which is necessary for understanding the spatial structure of the flow .
- The diagonal matrix S consists of singular values only, the larger the singular value, the greater the contribuition of energy.
- The matrix VT contains the temporal coefficients, providing insight into how each mode evolves over time.

Next step was the selection of top 10 modes which are the modes with highest energy value.

Results



Examining the plot, we can clearly observe a sharp decrease in values from the initial indices, indicating that the primary modes include a substantial portion of the energy within the fluid flow. With increasing indices, the singular values decrease, implying a decreasing contribution from these higher-indexed modes to the overall flow structure.

The curve's pattern suggests that a considerable count of modes, particularly those associated with the highest singular values, might represent the majority of the energy or information within the fluid flow. Such a scenario is frequently encountered in fluid dynamics investigations, characterized by the dominance and representation of a few dominant flow structures.

CODE:

```
import numpy as np
import matplotlib.pyplot as plt

# Calculate mean image
mean_image = np.mean(downsampled_images_gray, axis=0)
plt.imshow(mean_image)
plt.title('Mean Image')
plt.axis('off')
plt.show()
print(mean_image.shape)

# Function to flatten and stack images into a matrix
def stack_images(images):
    stacked_images = np.array([image.flatten() for image in
       images])
    return stacked_images

# Stack images into a matrix
stacked_images = stack_images(downsampled_images_gray)
print(stacked_images)
print(stacked_images.shape)

# Compute mean image
mean_image11 = np.mean(stacked_images, axis=0)
print(mean_image11)
print(mean_image11.shape)

# Step 2: Subtract the mean from each image
centered_images = stacked_images - mean_image11
print(centered_images)
print(centered_images.shape)

data_matrix = centered_images

# Perform Singular Value Decomposition (SVD)
U, S, Vt = np.linalg.svd(data_matrix, full_matrices=False)
print(U)
print("spatial modes ",U.shape)
print(S)
print("singular values ",S.shape)
print(Vt)
print("temporal coefficients ",Vt.shape)
```

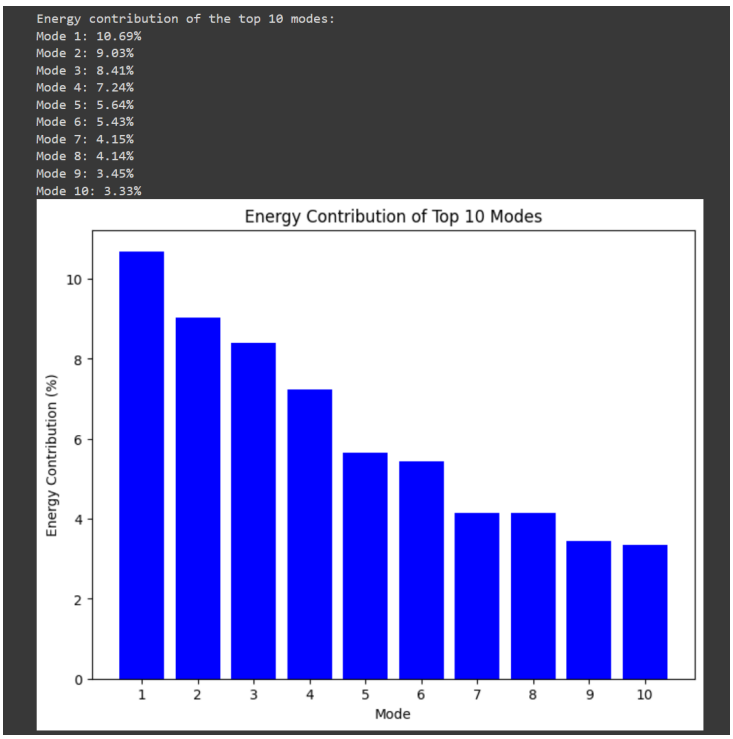
```
42
43 # Select the top 10 modes
44 top_modes = Vt[:10, :]
45 print(top_modes)
46 print(top_modes.shape)
47
48 # Plot the top eigenvalues
49 plt.figure(figsize=(8, 6))
50 plt.plot(np.arange(1, len(S) + 1), S, linestyle='-',
      color="red")
51 plt.title('Singular Values Distribution')
52 plt.xlabel('Index')
53 plt.ylabel('Singular Value')
54 plt.show()
```

3.3 Analysing POD Modes

The subsequent step involves determining the energy attributed to each singular mode, derived from the squared singular values, facilitating an evaluation of their respective contributions to the total energy of the system. Another crucial objective is to ascertain the minimum number of modes necessary to capture a significant portion of the system's energy, specifically 95

To achieve this, we have calculated the system's total energy by summing the squares of the singular values. Subsequently, the energy contribution of each mode is calculated and represented as a percentage of this total energy. Focusing mainly on the top 10 modes, presenting their energy contributions graphically through a bar chart, visualizing the dominance of these principal modes.

Following this, a cumulative variance graph is plotted to discern the number of modes required to retain 95 percent of the system's total variance. This graphical shows the trade-off between the number of modes retained and the captured variance.



Mode 1 represents the highest energy. The energy contribution experiences a significant decline from Mode 1 to Mode 2 and continues to decrease as the mode number increases—a common characteristic observed in fluid dynamics, where a handful of dominant structures hold the maximum variance and information.
Prioritizing these primary modes in reconstructing the flow field would gave us a representation that preserves the important features. This approach ensures a more efficient analysis and interpretation of the flow dynamics, while also decreasing computational complexity.

This analysis shows that only a limited number of modes are essential to justify and explain the majority of the system's energy.

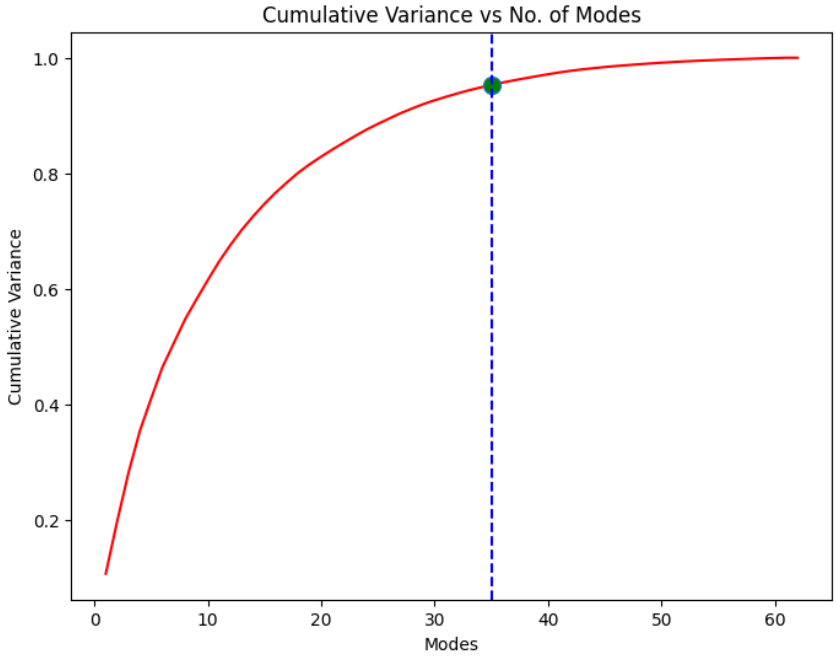


Figure 2: As we move towards the right along the x-axis, which represents the increasing number of modes, the curve begins to plateau, showing that each additional mode contributes less to the explanation of the total variance.

By identifying the number of modes that account for 95% of the total variance, we ensure that the reconstruction of the fluid flow retains most of the significant features while discarding the rest as noise or less important fluctuations. We then proceed to reduce the dimensions of my matrices accordingly, focusing only on the significant modes, which simplifies the complexity of the system and is essential for efficient data processing and analysis.

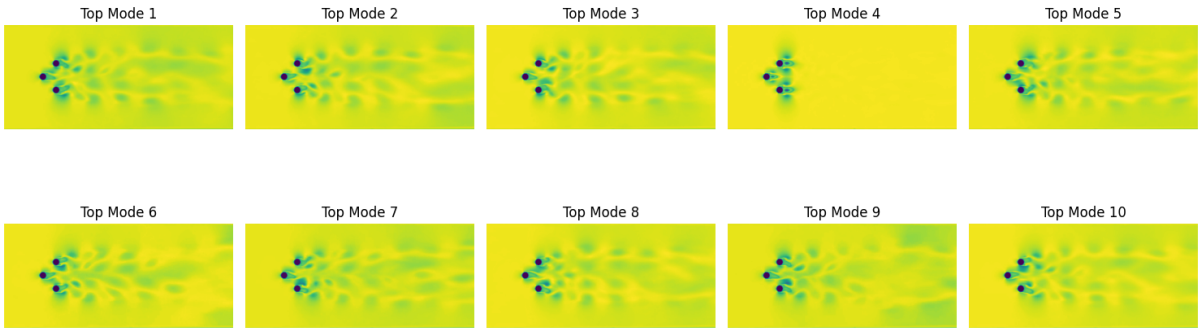


Figure 3: The above are the representations of the top 10 Modes. These are the 10 modes that have the highest energy.

4 Noise

Since we had memory considerations and constraints at play using Google Colab, we had to introduce modularity in our code so that the various sections can be independently run in a new runtime and we can achieve the results without any memory crash or ram exhausted error. We have explored 2 types of Noise, namely:

Gaussian Noise

Salt and Pepper Noise

4.1 Adding Noise

The next objective of this code is to inject artificial noise into the frames of a fluid flow video. Gaussian noise of varying magnitudes is added to simulate real-world disruptions in fluid flow imagery. By this, we seek to create a dataset that tries to copy practical scenarios where data might be corrupted by noise, thus testing the how well and resilient the POD/SVD process is against such imperfections.
We have used The `add_gaussian_noise`function, which generates a Gaussian distribution of noise, and is overlaid onto the original image. The intensity of this noise is adjustable, and the resulting image values are clamped so as to ensure and maintain validity within pixel boundaries. In the `frames_with_noise` function, the video undergoes frame-by-frame processing, with noise of predetermined percentages introduced at specified intervals as we have defined. Next, these modified frames are stored in separate folder corresponding to their respective noise levels and following a described nomenclature, ensuring an organized structure for subsequent analysis.

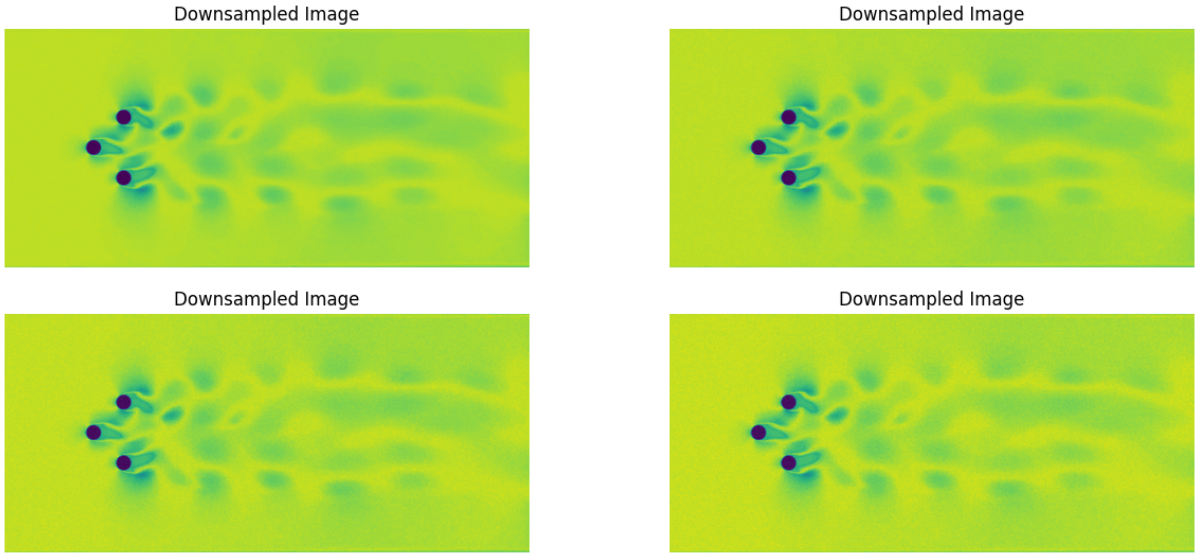
By integrating noise into the analysis, we can assess the resilience of the extracted modes and the accuracy of flow reconstruction post-POD/SVD. The variation in noise levels facilitates a comprehensive examination of fluid flow feature sensitivity to disturbances, crucial for validating the practical applicability of the developed computational model.

The next part of the code is to retrieve images from specified directories, each containing frames with different intensities of Gaussian noise applied. The loading function is a critical component for assembling the datasets required for subsequent noise-impact analysis.

Utilizing the OpenCV library, the `load_images_from_folder` function we have employed iterates through the files in the determined folder as described. It ensures that each im-

age is correctly loaded in grayscale, which simplifies the SVD processes by reducing the complexity of the data. By invoking this function for each noise level directory, I compile separate collections of images (`noisy_images1`, `noisy_images2`, etc.) that correspond to specific noise intensities.

Images after adding noise and down-sampling(which was provided but not utilised) :



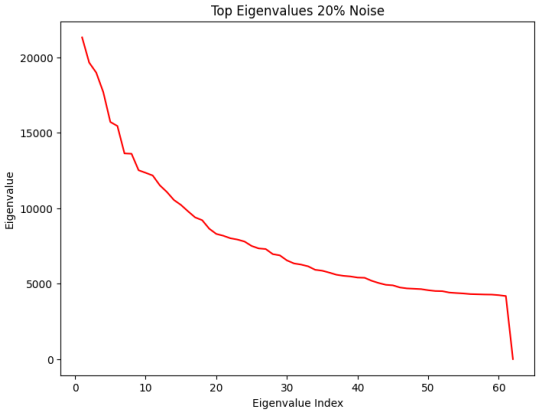
We observed that the introduction of Gaussian noise compromised the clarity of the original images from the fluid flow, which manifested itself as random fluctuations in pixel values across the images. These variations hide the finer details in the fluid flow patterns, and could have introduced apparent fluctuations in the original dataset that were not inherently present. As a result, we observe that images have become greener, probably suggesting higher energies..

In short, what we did was as follows. We first stacked the images into a row, and stacked images on top to form a 2-D matrix. As a result, what we obtained were 1-D stacked images times the no of images. Then what we did was mean centralise the data and pass it to the svd function to observe E, S and Vt values.

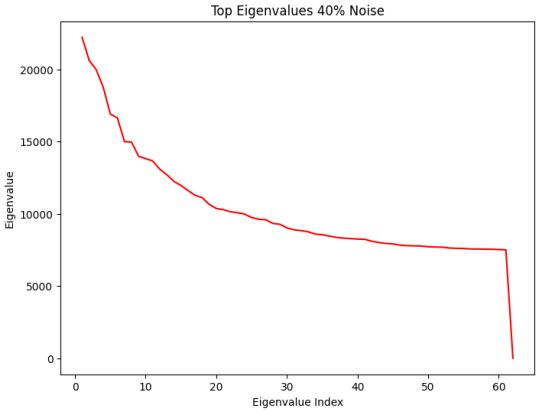
We utilized the `stack_images` function to convert the image sets into matrix form suitable for mathematical manipulation. By subtracting the mean image from each corresponding stacked image matrix, we ensure data centralization, which is pivotal for effective POD and SVD analyses.

Following centralization, SVD is executed on each centralized matrix, yielding spatial modes, singular values, and temporal coefficients for noise levels of 20%, 40%, 60%, and 80%. The dimensions of the resulting matrices are printed, that offered immediate insights into the structure of the decomposed modes.

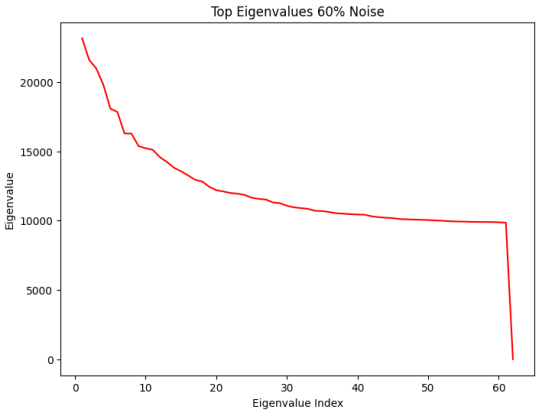
Moreover, we implemented a similar approach as in question 3. In our analysis,bar graphs are generated to ascertain the percentage contributions of the top 10 modes. This analysis helped us in understanding the relative significance of these modes in the flow. The top 10 modes, with their energy contributions, explained variance plots were shown and additionally, a histogram was also shown for visual analysis and appeal.



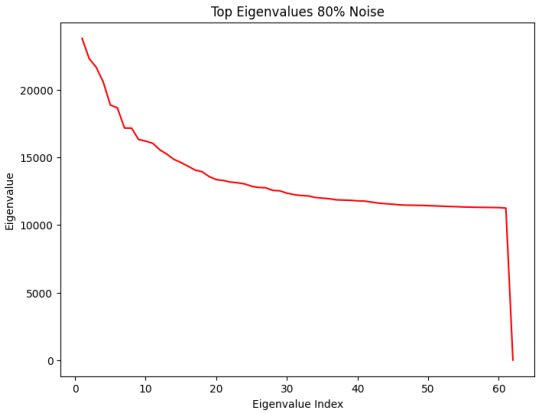
(a) At 20% noise, the eigenvalues exhibit a departure from the original clean data, displaying a less pronounced decline. This suggests that the noise is beginning to disperse the energy across a broader spectrum of modes compared to the noise-free scenario.



(b) With 40% noise, the descent of eigenvalues becomes even more gradual, indicating that the noise is assuming greater significance within the data.



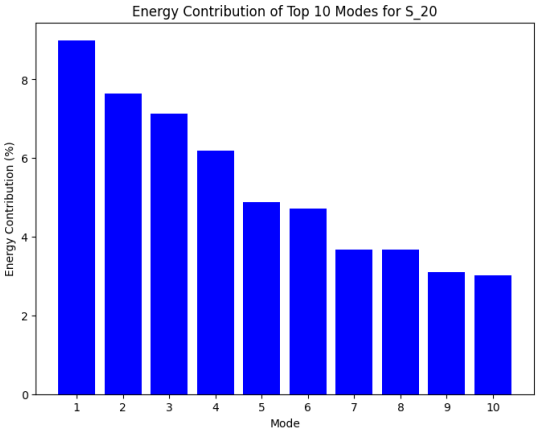
(c) At 60% noise, the presence of noise becomes substantial, as evidenced by the plateauing of eigenvalues. This shows that the noise further obscures the distinction between signal and noise.



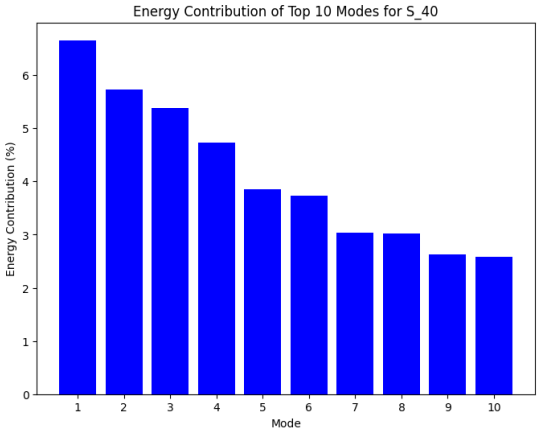
(d) Finally, at 80% noise, the plot illustrates a substantial alteration in eigenvalue distribution, with a flatter decline.

Indeed, when comparing these graphs to the original eigenvalue distribution from the clean data, a clear trend emerges: as the noise level escalates, the capability of the initial modes to encapsulate the majority of the system's energy diminishes. This phenomenon has significant implications for the efficacy of low-dimensional representations of fluid flow.

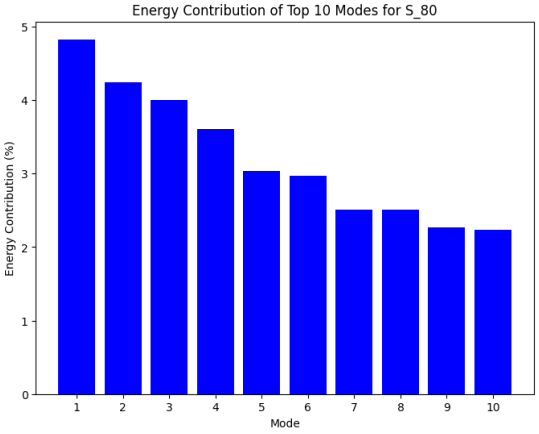
Further, producing bar graphs as question 3, to find out the percentage contributions of the top 10 modes we get:



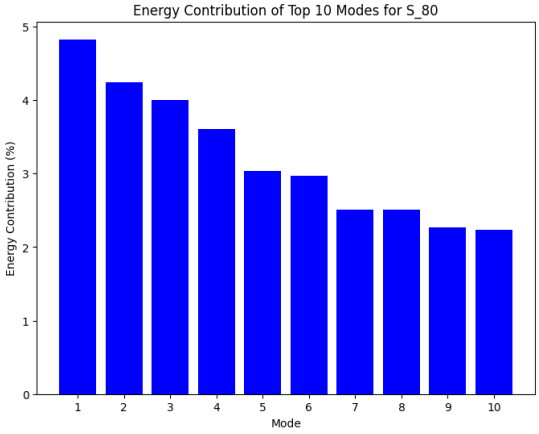
(a) The highest mode still retains a significant amount of energy, similar to the original graph on data without noise, but there is a noticeable distribution of energy to subsequent modes compared to the original dataset which indicates the impact of noise.



(b) The energy becomes more evenly spread across the modes, suggesting that the noise is disrupting the flow patterns.

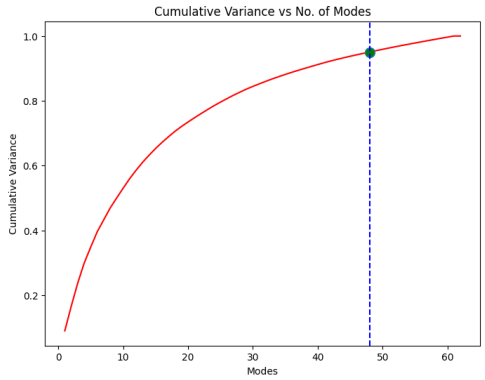


(c) The energy contributions of the modes become even more uniform, indicating that the noise is adding significant complexity to the flow dynamics. The greater amount of noise, the greater it affects the variance of the data and the number of modes required to preserve it

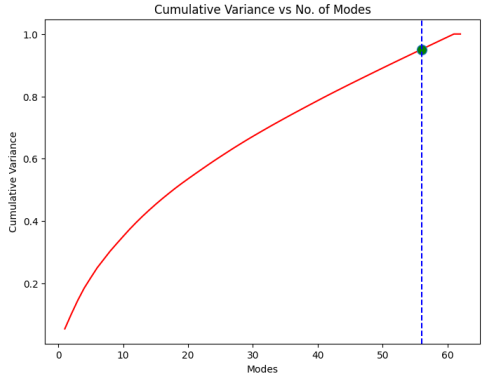


(d) The top modes contribute comparatively less energy, and the energy is spread out over many modes. This indicates that the noise is now dominating the signal, making it increasingly challenging to distinguish between actual flow features and noise.

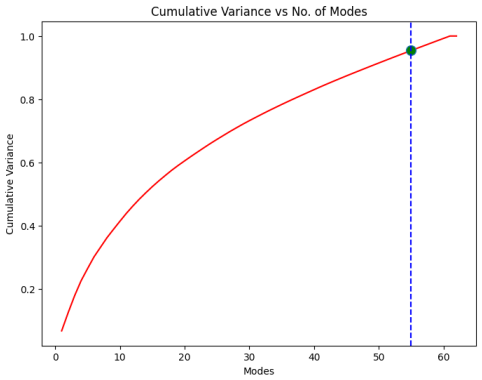
Next, Identifying the number of modes which account for 95 % of the variance we get the following results:



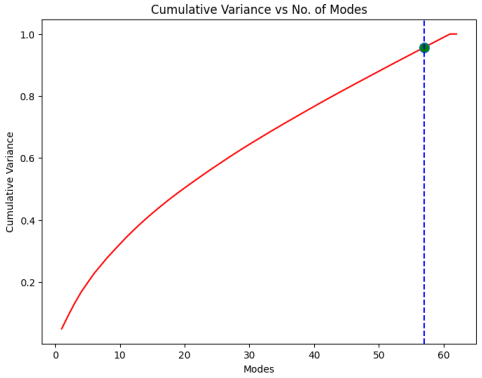
(a) The observed sharp increase followed by a leveling off in the cumulative variance plot near the marker indicates that only a few modes are required to capture the majority of the system's behavior. This trend closely resembles the behavior observed in the original noise-free scenario, implying good data quality despite the presence of noise.



(c) Moving on to 60% Noise, the curve straihgtens out even more, and the dotted line shifts to the right which is a clear indication that as noise increases more number of features are required to explain the cariance



(b) As the noise level increases, the curves become less steep. We can see in the graph, approximately 54 modes explain majority variance in the data.



(d) The presence of the flattest curve among all suggests that the noise has significantly impacted the data, which considerably shifts the dotted line to the right, meaning maximum modes are required to explain 0.95 variance when maximum noise is added

Reducing U, S, Vt to lower dimensions, we get:

```
# Check the shapes
print("Reduced U shape:", U_reduced_20.shape)
print("Reduced S shape:", S_reduced_20.shape)
print("Reduced Vt shape:", Vt_reduced_20.shape)
```

```
Reduced U shape: (62, 48)
Reduced S shape: (48, 48)
Reduced Vt shape: (48, 1103064)
```

```
# Number of modes to retain the desired explained variance ratio
num_modes_40 = num_modes_90_percent_40

# Reduce dimensions of U, S, and Vt
U_reduced_40 = U_40[:, :num_modes_40]
S_reduced_40 = np.diag(S_40[:num_modes_40])
Vt_reduced_40 = Vt_40[:num_modes_40, :]

# Check the shapes
print("Reduced U shape:", U_reduced_40.shape)
print("Reduced S shape:", S_reduced_40.shape)
print("Reduced Vt shape:", Vt_reduced_40.shape)
```

```
Reduced U shape: (62, 55)
Reduced S shape: (55, 55)
Reduced Vt shape: (55, 1103064)
```

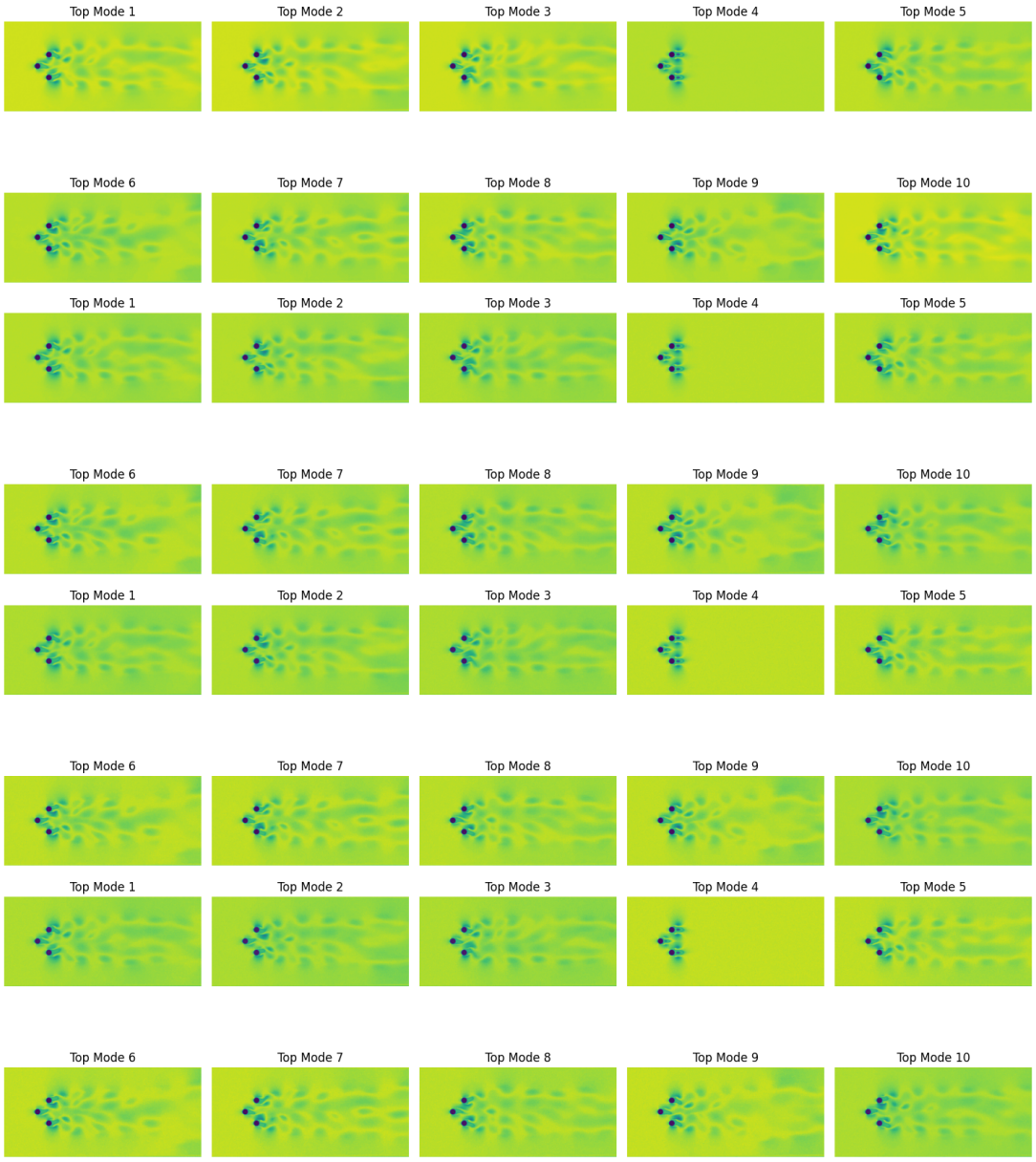
```
[ ] # Number of modes to retain the desired explained variance ratio
num_modes_60 = num_modes_90_percent_60

# Reduce dimensions of U, S, and Vt
U_reduced_60 = U_60[:, :num_modes_60]
S_reduced_60 = np.diag(S_60[:num_modes_60])
Vt_reduced_60 = Vt_60[:num_modes_60, :]

# Check the shapes
print("Reduced U shape:", U_reduced_60.shape)
print("Reduced S shape:", S_reduced_60.shape)
print("Reduced Vt shape:", Vt_reduced_60.shape)
```

```
Reduced U shape: (62, 56)
Reduced S shape: (56, 56)
Reduced Vt shape: (56, 1103064)
```

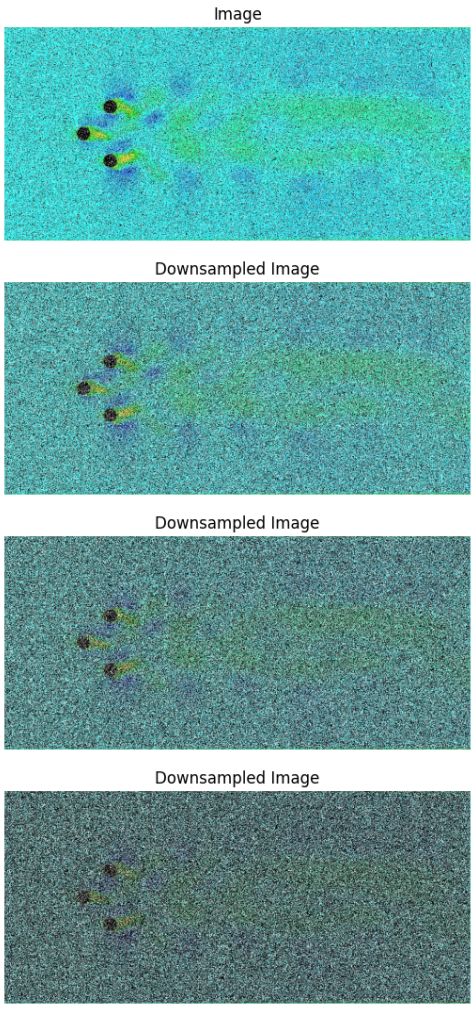
```
[ ] # Number of modes to retain the desired explained variance ratio
num_modes_80 = num_modes_90_percent_80

# Reduce dimensions of U, S, and Vt
U_reduced_80 = U_80[:, :num_modes_80]
S_reduced_80 = np.diag(S_80[:num_modes_80])
Vt_reduced_80 = Vt_80[:num_modes_80, :]

# Check the shapes
print("Reduced U shape:", U_reduced_80.shape)
print("Reduced S shape:", S_reduced_80.shape)
print("Reduced Vt shape:", Vt_reduced_80.shape)
```

```
Reduced U shape: (62, 57)
Reduced S shape: (57, 57)
Reduced Vt shape: (57, 1103064)
```

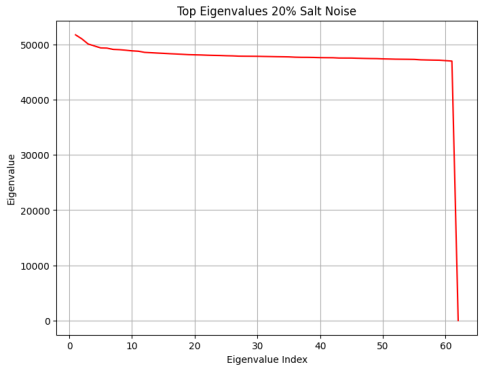
Finally Reconstructing the top 10 modes for each of 20%, 40% , 60%, 80 % noise iterations, we get:



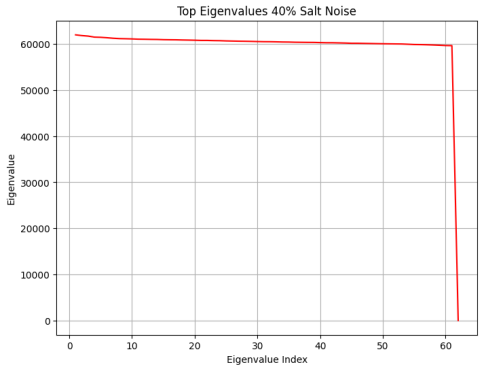
Adding Salt-Pepper Noise: The `add_salt_and_pepper_noise`function are created to simulate random disturbances within image data by adding salt and pepper noise, which type of noise is characterized by abrupt and sharp changes in image intensity.



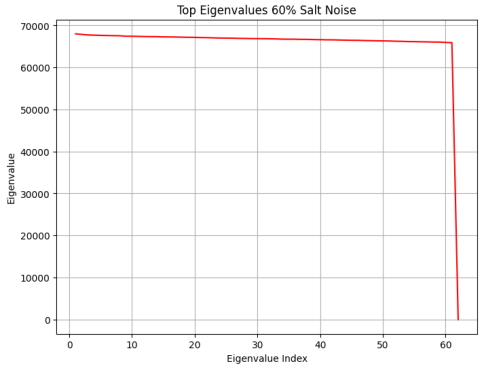
Upon receiving an image, the function which we have created creates a duplicate to preserve the original data's integrity and then calculates the number of 'salt' and 'pepper' pixels based on given percentages to the total number of pixels in the image. Next the function randomly selects coordinates within the image to introduce 'salt' (white pixels) and 'pepper' (black pixels), and modifies them accordingly.
After running similar functions and following the similar process as for Gaussian Noise, for Salt and pepper noise we get the following graphs:



(a) At 20% Salt-and-Pepper Noise: Even at this relatively low level, the impact of such noise can be significant. The eigenvalues are likely to exhibit drastic changes from the original, potentially resulting in an increase in smaller eigenvalues compared to the original clean data due to the binary nature of the noise.



(b) At 40% and 60% Salt-and-Pepper Noise: At these intermediate levels, noise starts becoming the dominant feature, likely causing a flattening effect on the eigenvalue curve. The extreme contrasts in pixel intensity, is a characteristic of salt-and-pepper noise



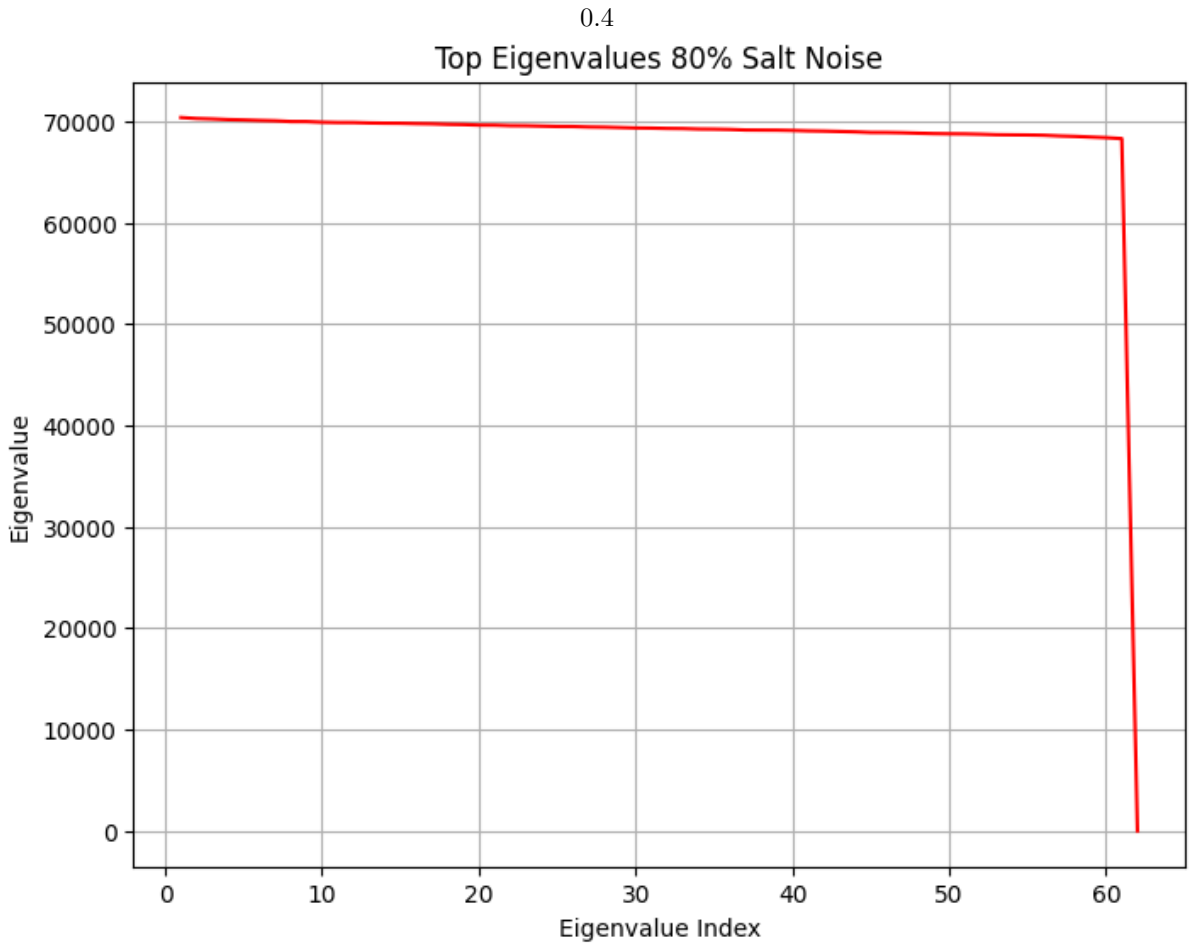
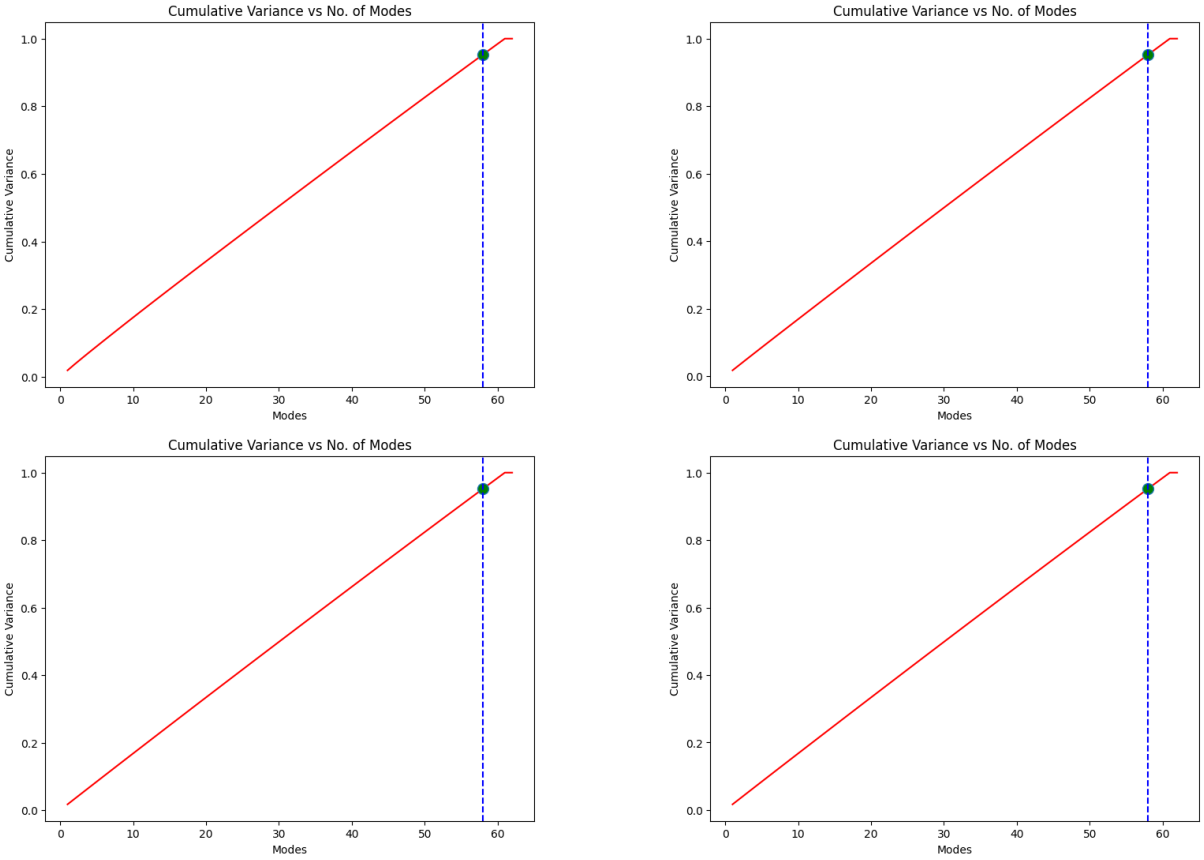
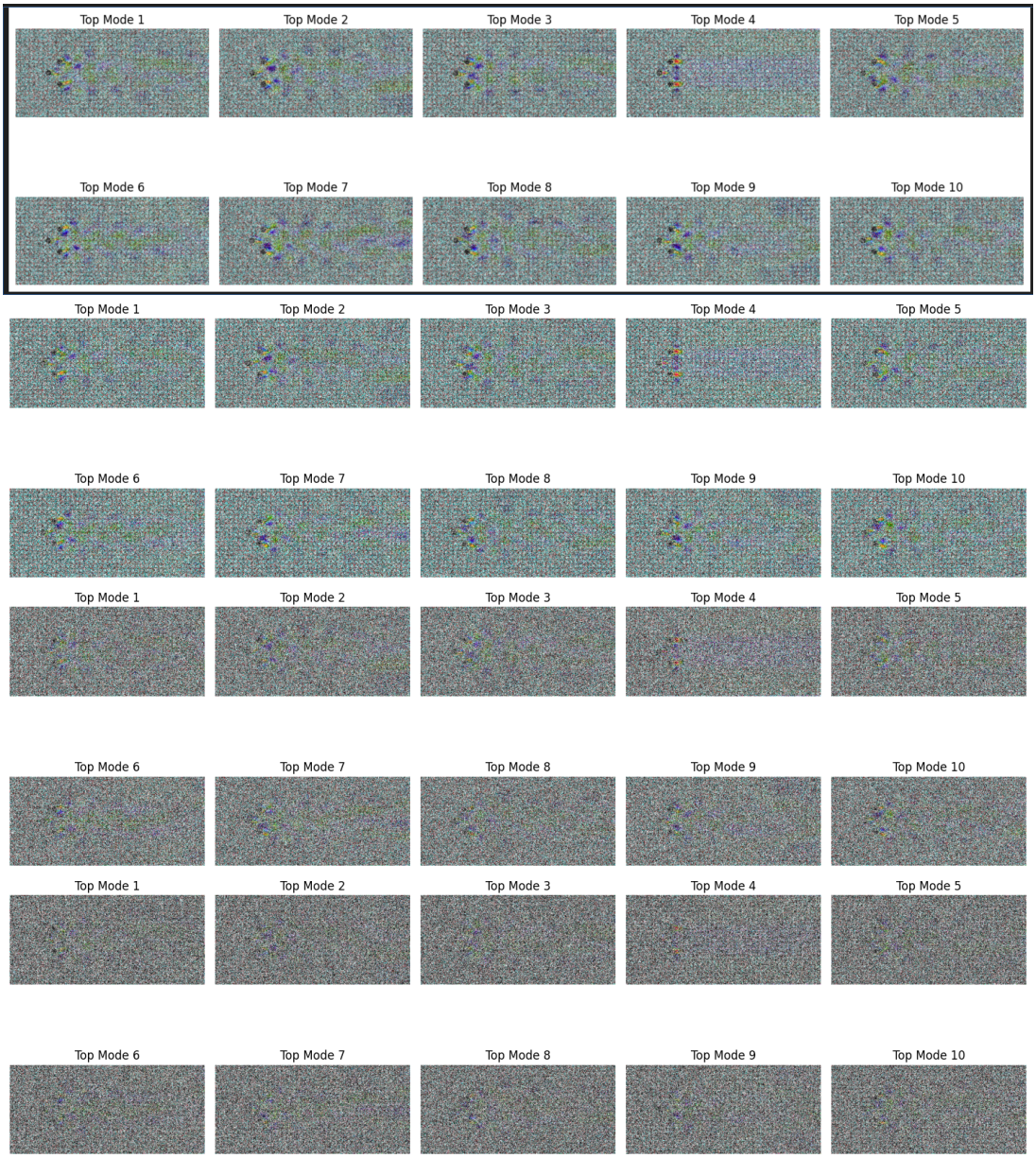


Figure 11: At 80% Salt-and-Pepper Noise: This has a very high level of noise, the eigenvalue distribution may or may not accurately reflect the underlying fluid flow structure. The data now predominantly represents the nouse itself and loses majority variance.

Compared to the original dataset's eigenvalue curve, this suggests that the noise is becomes more dominant as significant flow structures across a broader range of modes, which makes us lose our ability to extract meaningful data from noise.



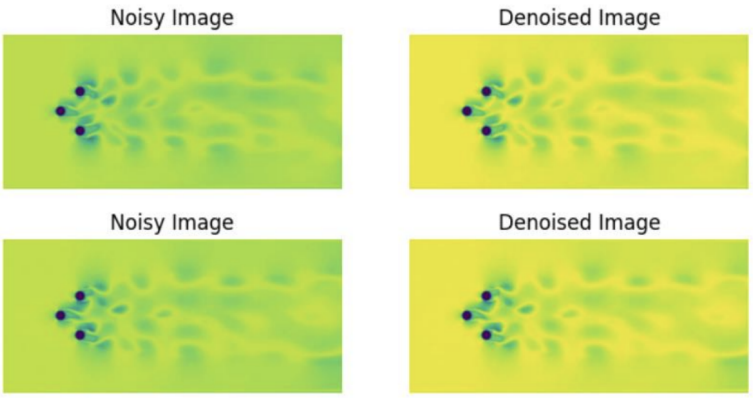
The common trend in the presence of salt-and-pepper noise shows that as the noise level increases, the curve flattens and moves to the right hand side, which shows that more modes are needed to capture the same amount of cumulative variance. This shift occurs because the salt-and-pepper noise, which introduces artificial high frequency components into the image pixels.



These reconstructed modes show that the quality is decreased due to noise and highlight the challenges in extracting meaningful flow dynamics as noise levels increase.
In conclusion, we can say that adding noise and then removing it was a good exercise to consider how well a noise works. To say which noise suits the best is subjective- it depends on how we want it. As observed, the gaussian noise increases as we increase its contribution, it does not reach a very high value of noise saturation like salt and pepper. Hence, we can say that the gaussian noise is better, in terms that we can evaluate our model performance more effectively, since on increasing the salt and pepper noise, the noise rises too high too fast, which is difficult for evaluation purposes. Different noises have different effect on modes, like gaussian raises the no of components for 95% explained cumulative variance slowly as it is added, while the salt and pepper noise raises it to a very high value very soon, and hence the no of components to explain 95% (or some other value) cumulative variance is very high and typically remains whether the noise contribution is 20% or if it is 80%.

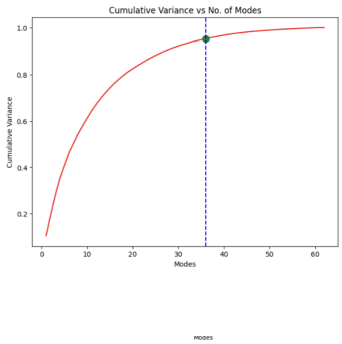
In close accordance with that thought process, we created new noisy images in the same way that we did in Q4. We introduced gaussian noise into the system by using the function add_gaussian_noise, where we passed input images and % noise as magnitude that would be introduced in the images. We also used the frames_with_noise function to extract frames at specified intervals and store it iteratively in separate folders namely Denoise_x, where x denotes the noise% added to the code. We hence have made four different folders where different concentration of Gaussian Noise exists. In our analysis, we have used only two noise concentrations to denoise to highlight the difference extensively of how much the different denoising methods we have employed are successful in their jobs.
Firstly, We employed **Gaussian Blur**
We used the denoise_with_gaussian_blur function to denoise the images, which employed the Gaussian Blur using OpenCV's cv2.GaussianBlur function, with kernel size of 5x5. As a result, we obtained a set of denoised images which helps smooth out the images.

For 20% Noise
Here we have displayed how the function has worked. The denoised images resemble closely with the original flow images, which is evident from the pictures attached.



Now, to look into and analyse how well our gaussian blur function has worked, we will again apply SVD to find the energies explained by the top 10 modes and find how much it varies from clean image. We perform the process as we have done before. We flatten and the stack the noisy images into a matrix, where each row represents an image. We find the mean and then mean-centralise the stacked images by subtracting the mean of the stacked images. We form this data matrix, pass it to the SVD function of linalg library and get the resulting spatial modes(U), Singular values(S) and the temporal coefficients (Vt). We then find the energy contribution of the top 10 modes and make some plots, Firstly, we observe that the Total contribution of the top 10 modes as a percentage of total energy: 61.12870962835086 % In comparison, for clean images, the Total contribution of the top 10 modes as a percentage of total energy: 61.5223969779374 %

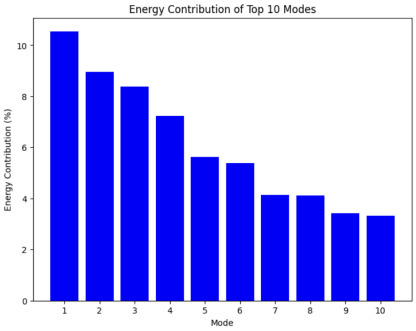
Hence, we observe that for 20% noise, the gaussian blur method effectively reduced the noise while maintaining the structural information of the images. It smoothened out the noise while retaining the important information. As a result, the top modes extracted by applying SVD captured almost similar energies for both clean and noisy image.



In comparison of 35 modes that explained 95% variance for clean images, our gaussian blur achieved it in 36 modes only.

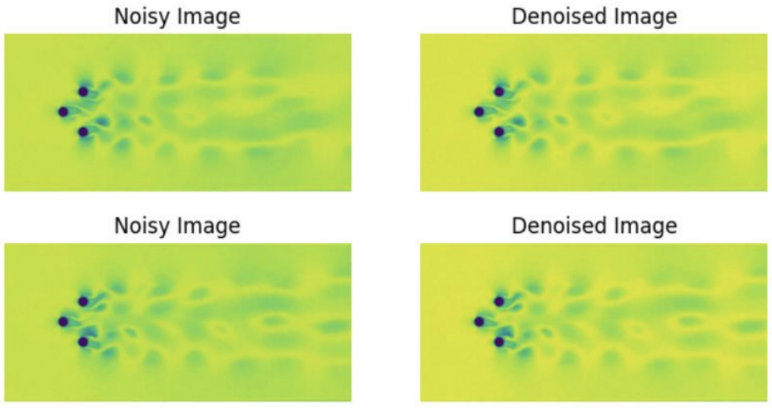
Number of modes to retain 95.0% variance: 36

```
Energy contribution of the top 10
modes:
Mode 1: 10.55%
Mode 2: 8.96%
Mode 3: 8.37%
Mode 4: 7.22%
Mode 5: 5.63%
Mode 6: 5.39%
Mode 7: 4.14%
Mode 8: 4.12%
Mode 9: 3.43%
Mode 10: 3.32%
```



For 80% Noise

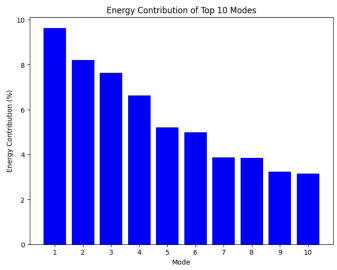
To observe stark difference, we skipped analysis on 40% and 60% noise and directly did analysis on 80% noise. Here we have displayed how the function has worked. The denoised images resemble almost closely with the original flow images, which is evident from the pictures attached.



We implemented the same process again and now we will display the results we observed. Total contribution of the top 10 modes as a percentage of total energy: 56.41481409781544 % In comparison, for clean images, the Total contribution of the top 10 modes as a percentage of total energy: 61.5223969779374 %

Hence, we observe that for 80% noise, the gaussian blur method effectively reduced the noise while maintaining the structural information of the images, however it does it slightly less impactfully than with 20% noise. This is because since the noise itself has become too dominant in the image data and as such it is difficult for denoising techniques to distinguish between what is noise and what is the image. As a result, for higher noise, though the process is a slight effective, it preserves the essential structural information of the images.

```
Energy contribution of the
top 10 modes:


Mode 1: 9.64%
Mode 2: 8.21%
Mode 3: 7.63%
Mode 4: 6.64%
Mode 5: 5.20%
Mode 6: 4.99%
Mode 7: 3.86%
Mode 8: 3.85%
Mode 9: 3.24%
Mode 10: 3.16%
```

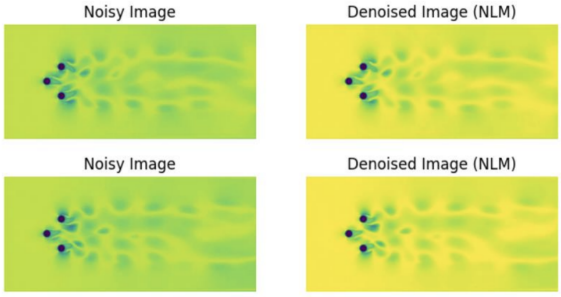


Method 2

We also considered using NLM denoising – Non-local Means Denoising, and implemented it using the function "denoise_with_nlm". This method effectively reduces noise while preserving important image details. We used the cv2.fastNLMeansDenoising function from the OpenCV library to achieve our objectives. This method accounts for similarities between different patches within the image to remove noise, while keeping structural info contact. Overall, it improves the quality of input image.

For 20% noise

Here we have displayed how the function has worked. The denoised images resemble closely with the original flow images, which is evident from the pictures attached



Now, to look into and analyse how well our NLM denoising function has worked, we will again apply SVD to find the energies explained by the top 10 modes and find how much it varies from clean image.

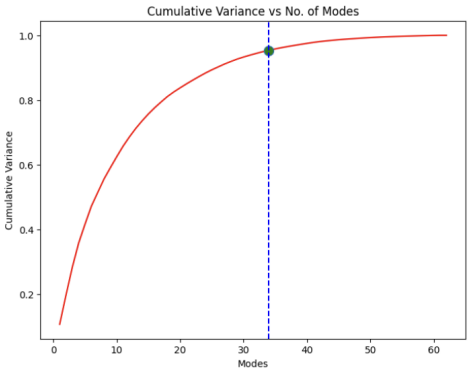
We perform the process as we have done before. We flatten and the stack the noisy images into a matrix, where each row represents an image. We find the mean and then mean-centralise the stacked images by subtracting the mean of the stacked images. We form this data matrix, pass it to the SVD function of linalg library and get the resulting spatial modes(U), Singular values(S) and the temporal coefficients (Vt). We then find the energy contribution of the top 10 modes and make some plots,

We observe that, Total contribution of the top 10 modes as a percentage of total energy: 62.512653008215366 % In comparison, for clean images, the Total contribution of the top 10 modes as a percentage of total energy: 61.5223969779374 %

We observed that the total contribution of the top 10 modes for the denoised images

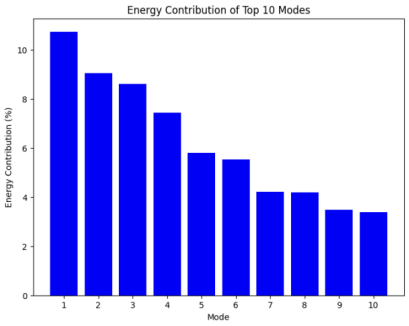
(62.5127%) is even higher than that of the clean images (61.5224%). This suggests that our NLM denoising function has increased the clarity of the image.

However, in a typical scenario, the denoising process tries to remove the noise from the images. This will result in a loss of some image information. However, this was not observed since the expected total contribution of the top modes after denoising was higher than that of the original clean images. This can be attributed that this function was unable to distinguish between noise and real image, or it may indicate that the denoising process has introduced some features that were originally not present in the original clean images. This can be attributed to the algorithm being overly aggressive or unable to distinguish between true and noise image features.



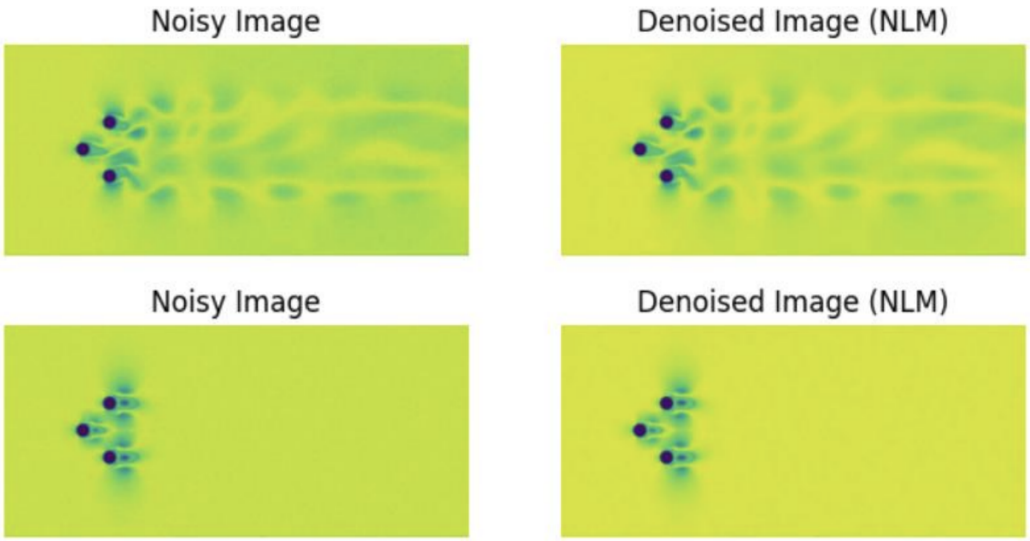
In comparison of 35 modes that explained 95% variance for clean images, our NLM-denoising function achieved it in only 34 modes.

```
Energy contribution of the top 10
modes:
Mode 1: 10.75%
Mode 2: 9.06%
Mode 3: 8.62%
Mode 4: 7.45%
Mode 5: 5.81%
Mode 6: 5.54%
Mode 7: 4.23%
Mode 8: 4.19%
Mode 9: 3.48%
Mode 10: 3.38%
```



For 80% Noise

To observe stark difference, we skipped analysis on 40% and 60% noise and directly did analysis on 80% noise. Here we have displayed how the function has worked. The denoised images resemble almost closely with the original flow images, which is evident from the pictures attached.



We implemented the same process again and now we will display the results we observed.

Total contribution of the top 10 modes as a percentage of total energy: 61.65473232793423 %

In comparison, for clean images, the Total contribution of the top 10 modes as a percentage of total energy: 61.5223969779374 %.

Again, we observed that the NLM produced better results than the actual clean images.

We observed that the total contribution of the top 10 modes for the denoised images (61.6547%) is even higher than that of the clean images (61.5224%). This suggests that our NLM denoising function has increased the clarity of the image. However, in a typical scenario, the denoising process tries to remove the noise from the images. This will result in a loss of some image information.

However, this was not observed since the expected total contribution of the top modes after denoising was higher than that of the original clean images. This can be attributed that this function was unable to distinguish between noise and real image, or it may indicate that the denoising process has introduced some features that were originally not present in the original clean images. This can be attributed to the algorithm being overly aggressive or unable to distinguish between true and noise image features.

Number of modes to retain 95.0% variance: 35

Number of modes to retain 95.0% variance: 35