# Fluid Project

Proper Orthogonal Decomposition on Schlieren Images

Somil Maheshwari
Nikhil Gangwar
Abhishek Kumar
Aman Bhansali
Rushil Samir Patel

## Introduction

## Introduction to schlieren imaging

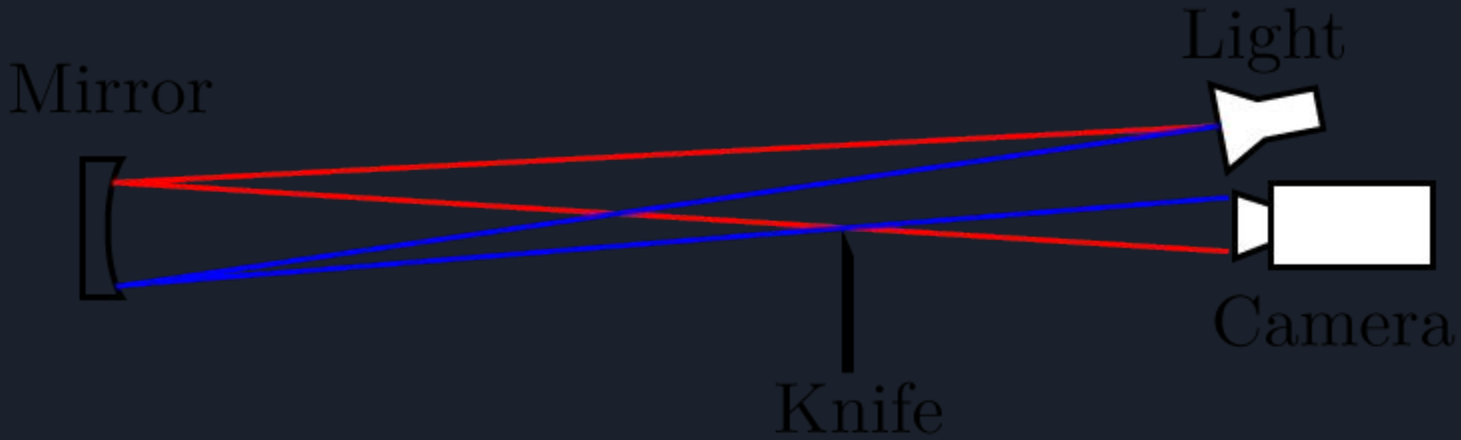
- Schlieren imaging is a process for photographing fluid flow.
- The schlieren imaging technique is a classical optical flow visualization technique based on density changes in the fluid.
- Schlieren optics measure the change in the refractive index of either air or another transparent media, and this enables flow patterns around an object to be imaged
- It is a technique that is used to not only determine the characteristics of the airflow itself but to also evaluate how an object behaves in certain airflow environments.

## Background

Schlieren imaging systems have been used since the early 1800’s to visualize fluctuations in optical density, but between 1859 and 1864, August Toepler rediscovered the technique and named it after the German term for optical inhomogeneities in glass . The term "Schlieren" refers to any optical inhomogeneities in a transparent medium.

## SETUP

The schlieren technique relies on small differences in the index of refraction bending light rays behind an obstruction.



Reference:- jove.com

## PROCEDURE

- ➢ Activate the dryer towers to dehydrate the air
- ➢ Check if the test section is clear of other objects, then close test section.
- ➢ Make sure the main valve for air flow control is closed, then turn on the compressor to pressurize the air storage tank.
- ➢ Turn on the controller for the high-speed valve
- ➢ Turn on the light and cooling fan of the schlieren imaging system.
- ➢ Place a piece of paper on the opposite side of the test section from the light source
- ➢ Align the concave mirror to allow light to pass through the test section.
- ➢ Adjust the knife-edge so that it is at the focal point of  mirror.
- ➢ Position a camera on a tripod directly in front of the knife-edge aperture to record the projected image.
- ➢ Open the air supply to the fast-valve controller, then open the main valve that lets air into the system.
- ➢ Turn off the wind tunnel by closing the valves in reverse order. Then turn off the controller.

## OTHER TECHNIQUES

Before schlieren imaging the Shadowgraph techniques are used to visualise the compressible flow.In shadowgraph the second derivative of density is not constant while the,first and third derivative are constant.

Shadowgraphs are typically used to study shocks as these flow features create non-constant second derivatives in density. Shadowgraphs are also useful for imaging boundary layers and turbulent flows. Shadowgraphs have an advantage over schlieren setups for such flows because shadowgraph visualization is uniformly sensitive in all directions.

The primary difference is that while shadowgraphs are sensitive to changes in the second derivative in density, schlieren systems detect changes to the first derivative in density.

Reference: https://shepherd.caltech.edu

# Application of Schlieren Imaging

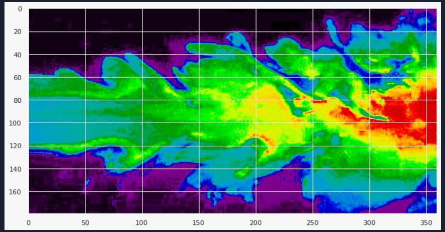
- Schlieren optics can be used to determine how aerodynamic an aircraft is by imaging how the air moves around it.

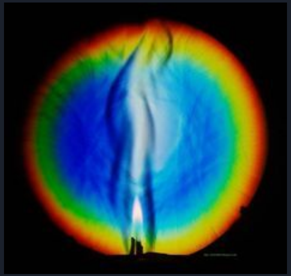
- Schlieren imaging provides valuable information on the spatial distribution of complicated flow structures in compressible, turbulent flow and in test flights.

- It used in air-to-air photography of supersonic aircraft, which involves using the sun and/or moon as a light source and the desert floor as the projecting surface to visualize the shock waves.

- supercomputers and wind tunnel testing are used to predict the formation, propagation, and merging of shock waves on an aircraft. To enhance the quality of these predictions, a database of sonic boom measurements are collected at various speeds and altitudes. This technique permits supersonic flow visualization of a full-scale aircraft, rather than a scaled-down model.

- Scramjets are air breathing engines that rely on the pure speed of an aircraft to compress air into the engine before combustion.



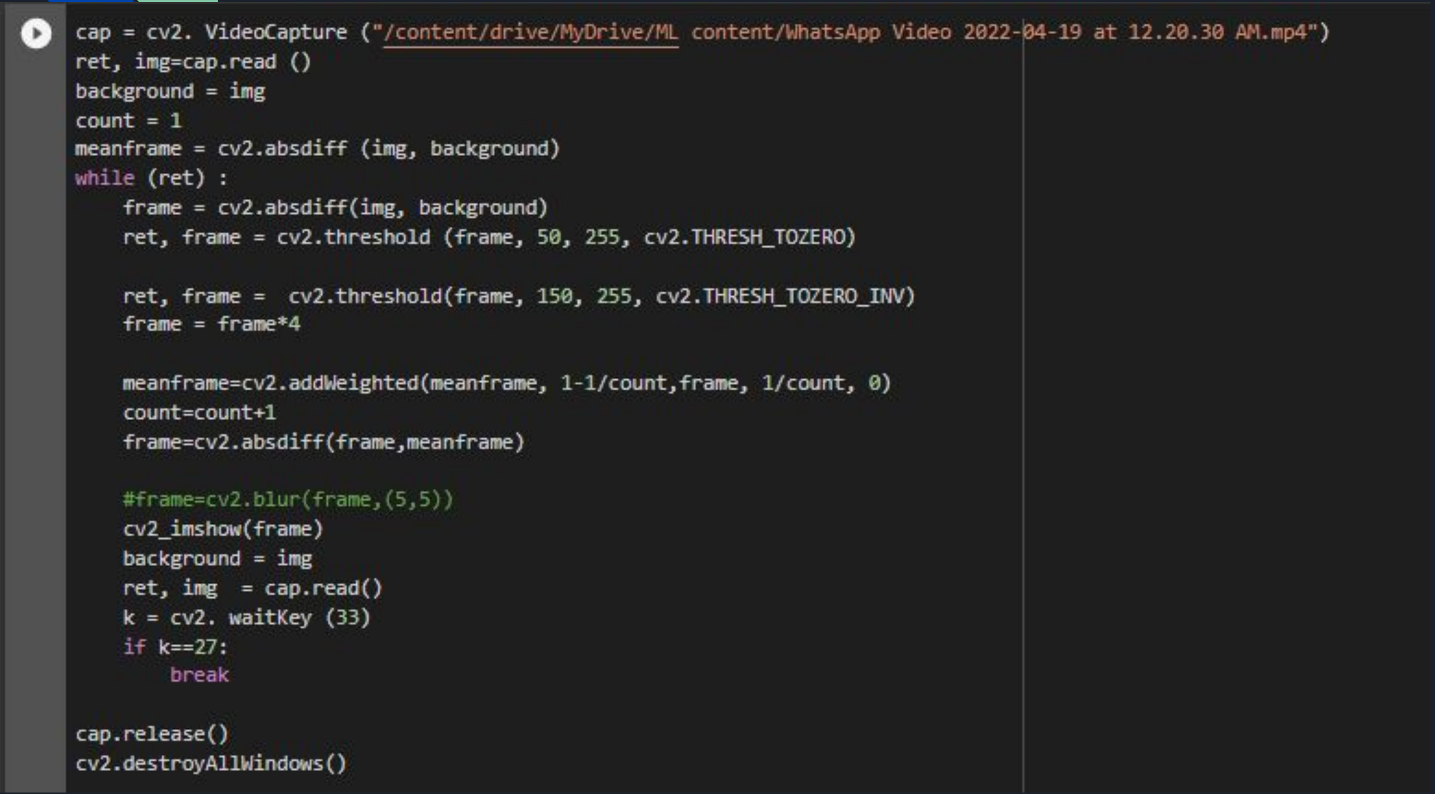


## Limitations

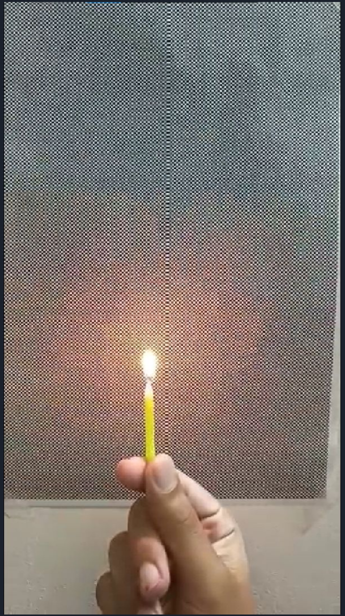
- Size and portability.
- All subjects must be stationary, or small enough to move within the field of view of the system, as the system itself is not significantly flexible or portable.
- Another considerable limitation is the cost of constructing a basic schlieren system.

## POD on Dot Schlieren

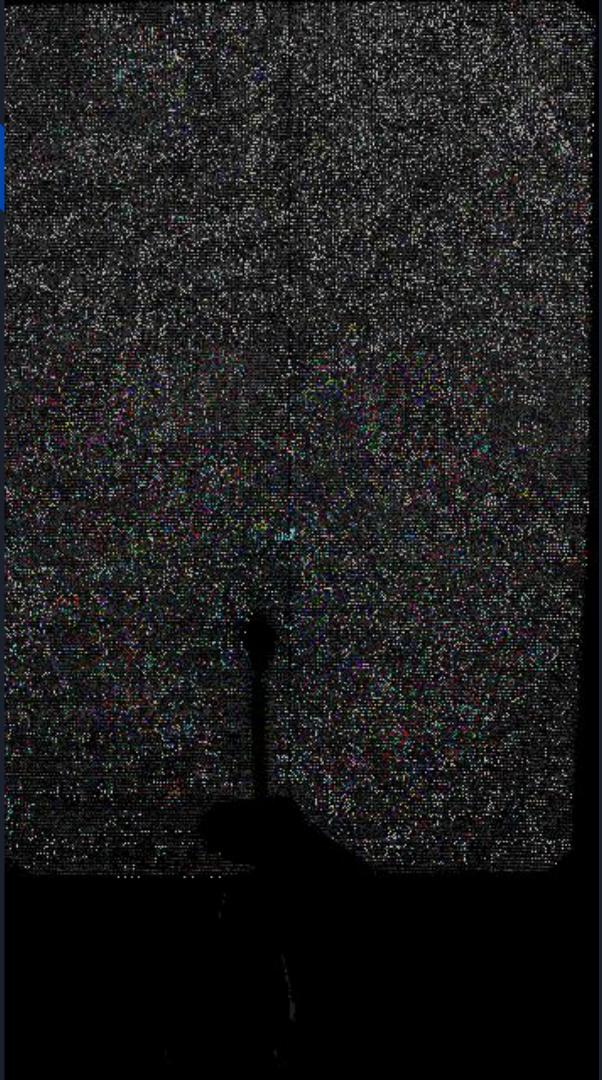
# Implementation of Dot Schlieren Method



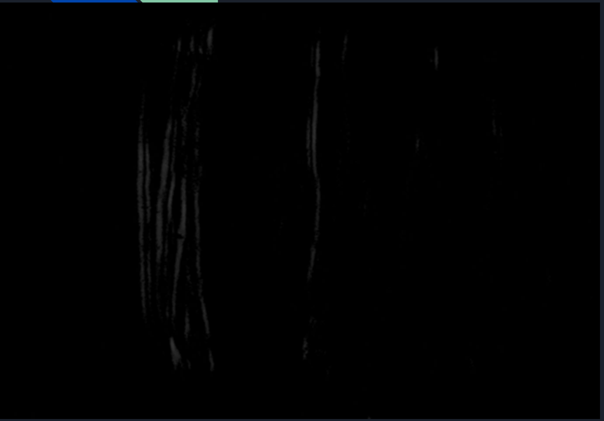
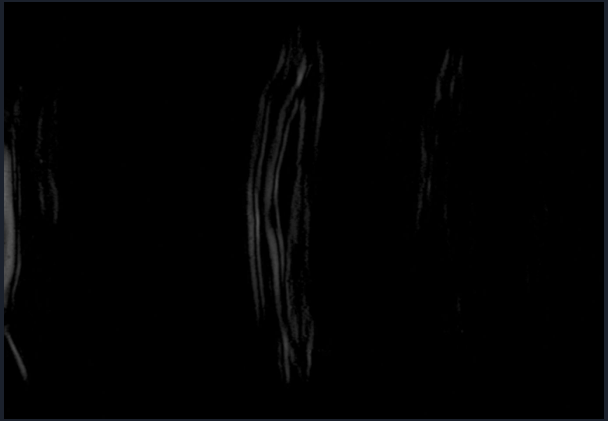
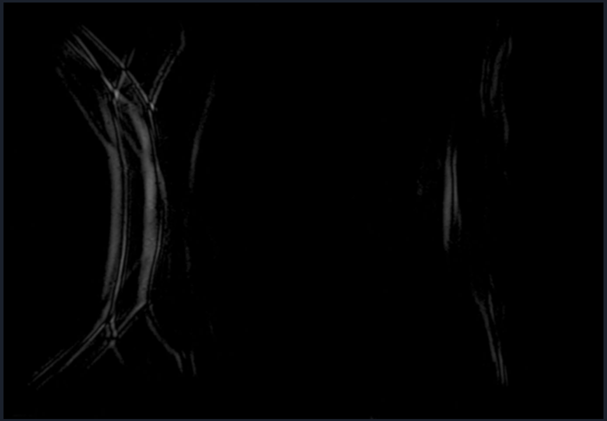
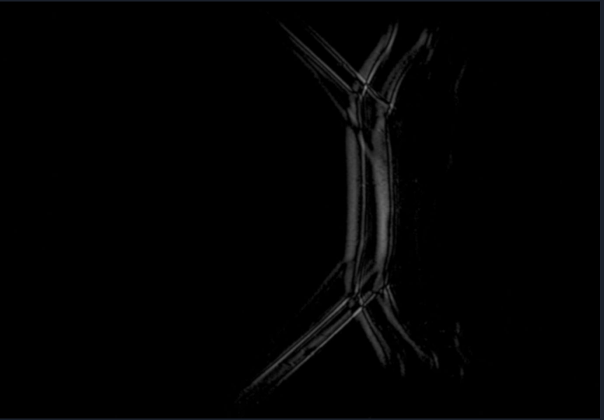
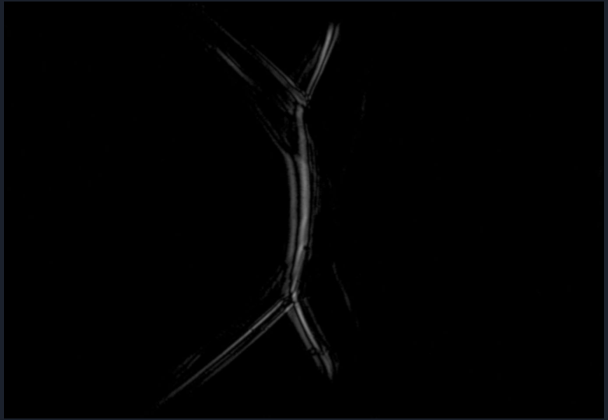
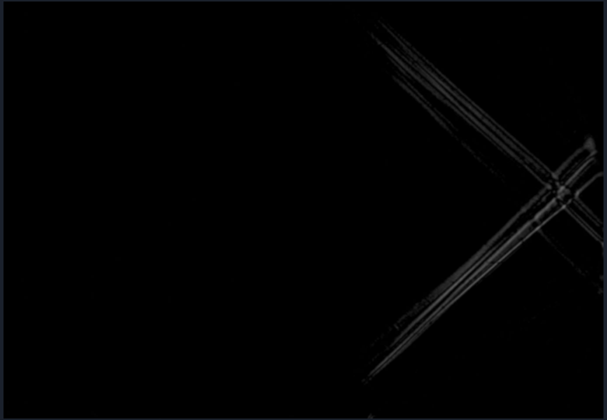
```
cap = cv2. VideoCapture ("/content/drive/MyDrive/ML content/WhatsApp Video 2022-04-19 at 12.20.30 AM.mp4")
ret, img=cap.read ()
background = img
count = 1
meanframe = cv2.absdiff (img, background)
while (ret) :
    frame = cv2.absdiff(img, background)
    ret, frame = cv2.threshold (frame, 50, 255, cv2.THRESH_TOZERO)

    ret, frame =  cv2.threshold(frame, 150, 255, cv2.THRESH_TOZERO_INV)
    frame = frame*4

    meanframe=cv2.addWeighted(meanframe, 1-1/count,frame, 1/count, 0)
    count=count+1
    frame=cv2.absdiff(frame,meanframe)

    #frame=cv2.blur(frame,(5,5))
    cv2_imshow(frame)
    background = img
    ret, img  = cap.read()
    k = cv2. waitKey (33)
    if k==27:
        break

cap.release()
cv2.destroyAllWindows()
```



Reference:
https://youtu.be/J0Ix4Wa3CJk



## Try to visualize Supersonic Flow

## POD

The proper orthogonal decomposition (POD) is one of the most used data analysis and modeling techniques in fluid mechanics and has seen many formulations, variants, and applications over the years.

Suppose we have data that is a function of both space and time.Now,

- How can I use this data to decompose this system into its building blocks?
- How can we start to understand the features present in the data?
- How can we efficiently store/represent/manipulate the data?
- Can we build a reduced-complexity model for the dynamics that we are interested in?

To start to answer these questions, we can start with a simple separation of variables:

$$\boldsymbol{y}(x,t) = \sum_{j=1}^{m} \boldsymbol{u}_j(x)\boldsymbol{a}_j(t)$$

Suppose we collect data at spatial locations X1, X2 ,X3..,Xn

Suppose we collect data at times ti,t2t3,.....,tm

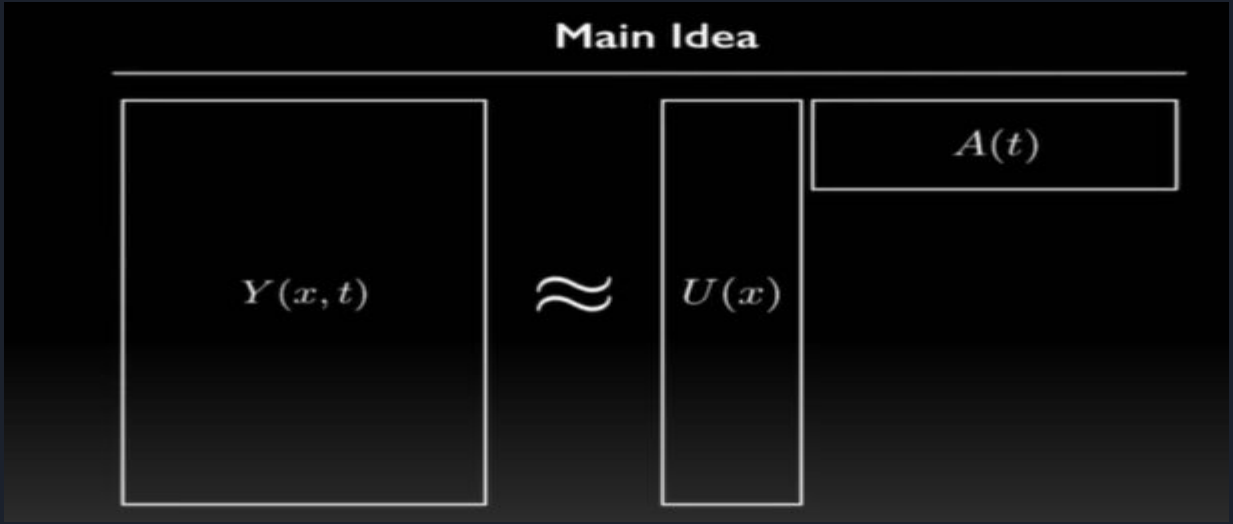
We can start by assembling the data y(x,t) into an m x n matrix

$$Y = \begin{bmatrix} y(x_1,t_1) & y(x_1,t_2) & \cdots & y(x_1,t_m) \\ y(x_2,t_1) & y(x_2,t_2) & \cdots & y(x_2,t_m) \\ \vdots & \vdots & \ddots & \vdots \\ y(x_n,t_1) & y(x_n,t_2) & \cdots & y(x_n,t_m) \end{bmatrix}$$

If we are measuring vector quantities, we can stack these into the data matrix Similarity.For example, if we measure two components of velocity, u and v.

$$Y = \begin{bmatrix} u(x_1,t_1) & u(x_1,t_2) & \cdots & u(x_1,t_m) \\ u(x_2,t_1) & u(x_2,t_2) & \cdots & u(x_2,t_m) \\ \vdots & \vdots & \ddots & \vdots \\ u(x_n,t_1) & u(x_n,t_2) & \cdots & u(x_n,t_m) \\ v(x_1,t_1) & v(x_1,t_2) & \cdots & v(x_1,t_m) \\ v(x_2,t_1) & v(x_2,t_2) & \cdots & v(x_2,t_m) \\ \vdots & \vdots & \ddots & \vdots \\ v(x_n,t_1) & v(x_n,t_2) & \cdots & v(x_n,t_m) \end{bmatrix}$$

Reference: https://youtu.be/TcqBbtWTcIc



This can be achieved using through a singular value decomposition:

$$Y = U\Sigma V^*$$

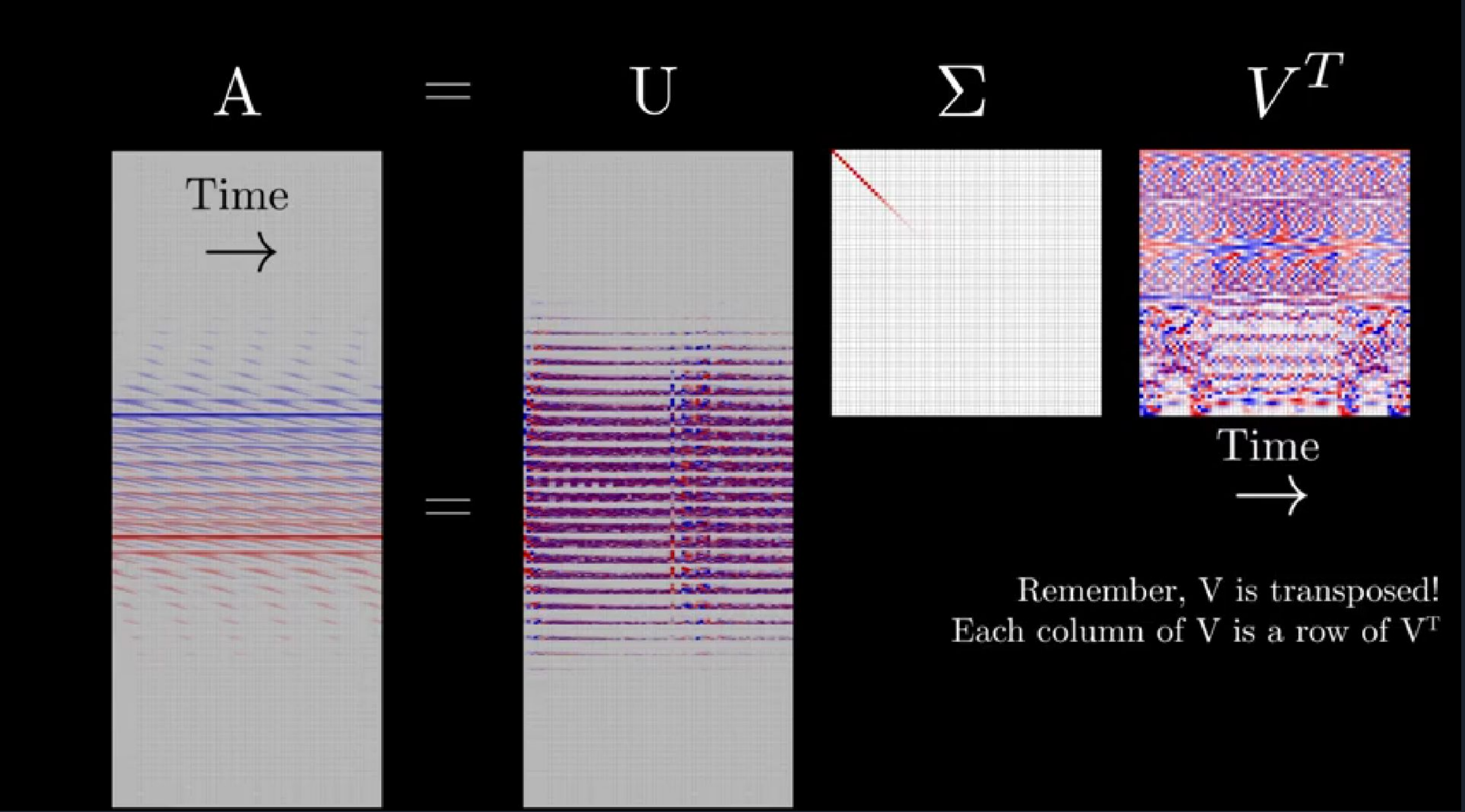
## POD using Full Singular Value Decomposition

## Singular Value Decomposition

It is an important tool used for data processing. It has to do with data reduction, to find the key correlations. It uses linear algebra to find the most influencing characteristic for any set of data given.

In this we decompose a matrix A into three different matrices each carrying their own significance and for the understanding of the modes.

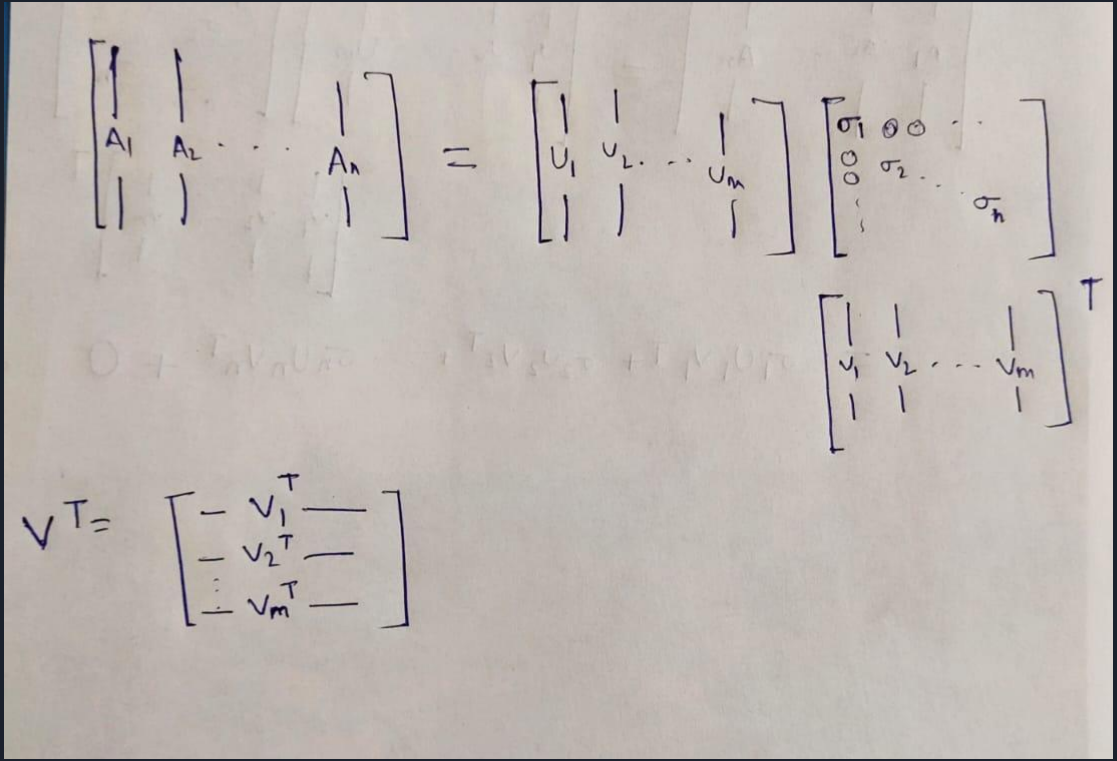
$$M = U \cdot \Sigma \cdot V^*$$



Remember, V is transposed!
Each column of V is a row of $V^T$

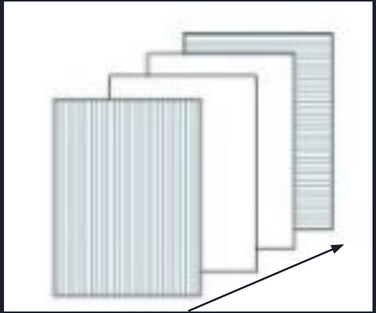
Ref: https://youtu.be/axfUYYNd-4Y

# Understanding and Visualizing

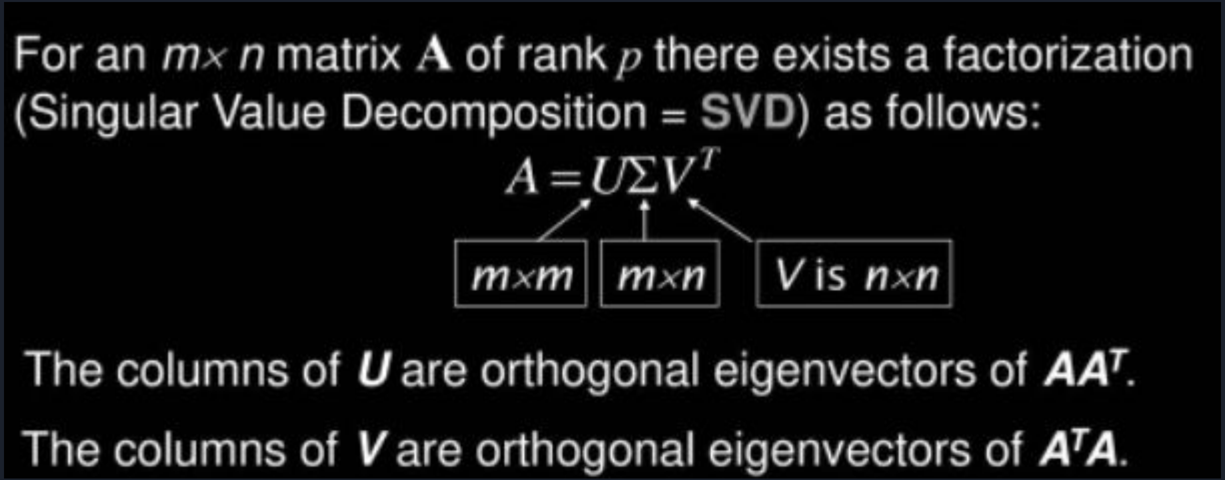


$$\begin{bmatrix} | & | & & | \\ A_1 & A_2 & \cdots & A_n \\ | & | & & | \end{bmatrix} = \begin{bmatrix} | & | & & | \\ U_1 & U_2 & \cdots & U_m \\ | & | & & | \end{bmatrix} \begin{bmatrix} \sigma_1 & 0 & 0 & \cdots \\ 0 & \sigma_2 & \cdots & \\ 0 & & & \\ \vdots & & & \sigma_n \end{bmatrix} \begin{bmatrix} | & | & & | \\ v_1 & v_2 & \cdots & v_m \\ | & | & & | \end{bmatrix}^T$$

$$V^T = \begin{bmatrix} - & v_1^T & - \\ - & v_2^T & - \\ & \vdots & \\ - & v_m^T & - \end{bmatrix}$$



# Interpretation

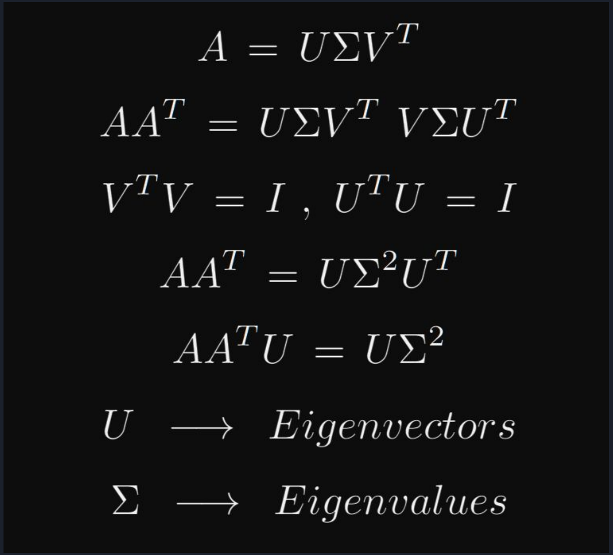


For an $m\times n$ matrix $\mathbf{A}$ of rank $p$ there exists a factorization (Singular Value Decomposition = **SVD**) as follows:

$$A = \underset{\boxed{m\times m}}{U}\,\underset{\boxed{m\times n}}{\Sigma}\,\underset{\boxed{V \text{ is } n\times n}}{V^T}$$

The columns of ***U*** are orthogonal eigenvectors of ***AA***$^T$.

The columns of ***V*** are orthogonal eigenvectors of ***A***$^T$***A***.



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

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

$$V^TV = I\ ,\ U^TU = I$$

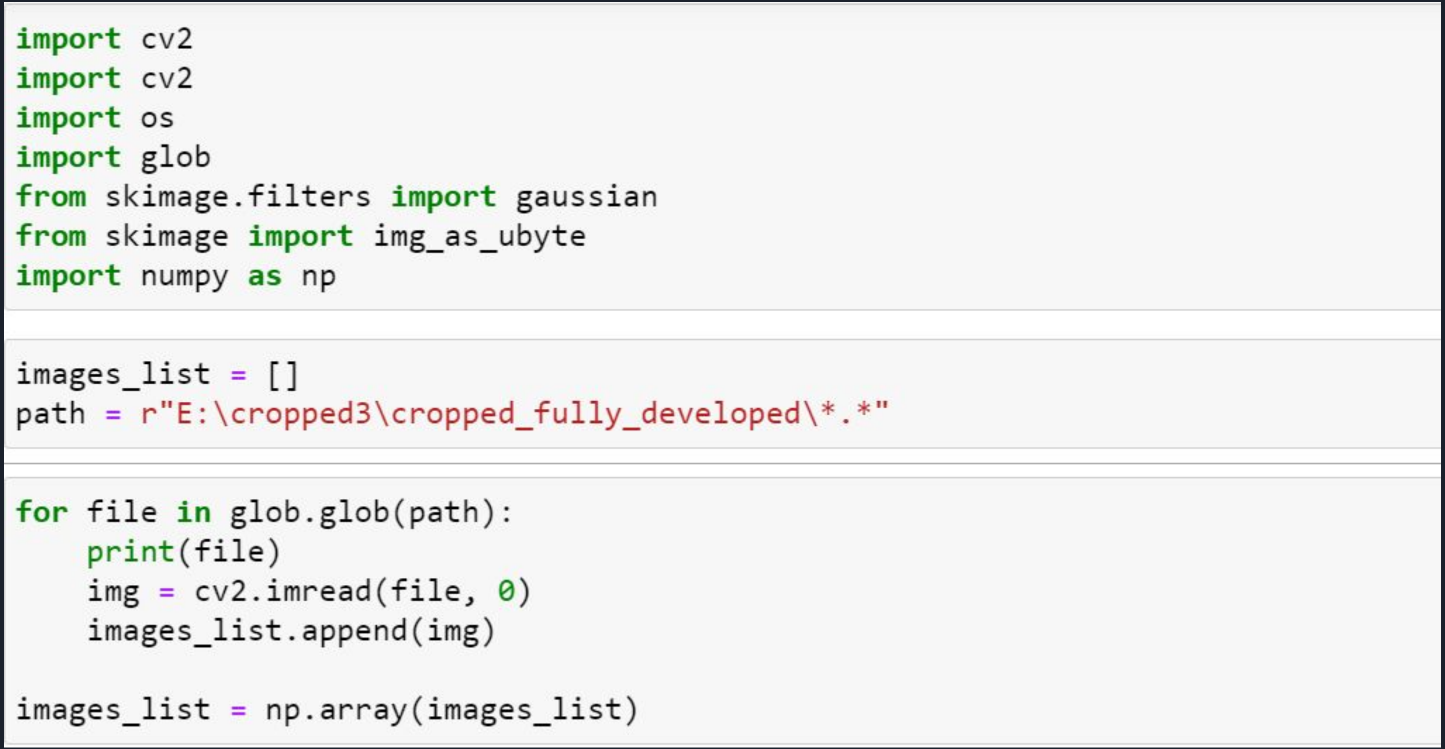
$$AA^T = U\Sigma^2U^T$$

$$AA^TU = U\Sigma^2$$

$$U \longrightarrow Eigenvectors$$

$$\Sigma \longrightarrow Eigenvalues$$

## Code For SVD

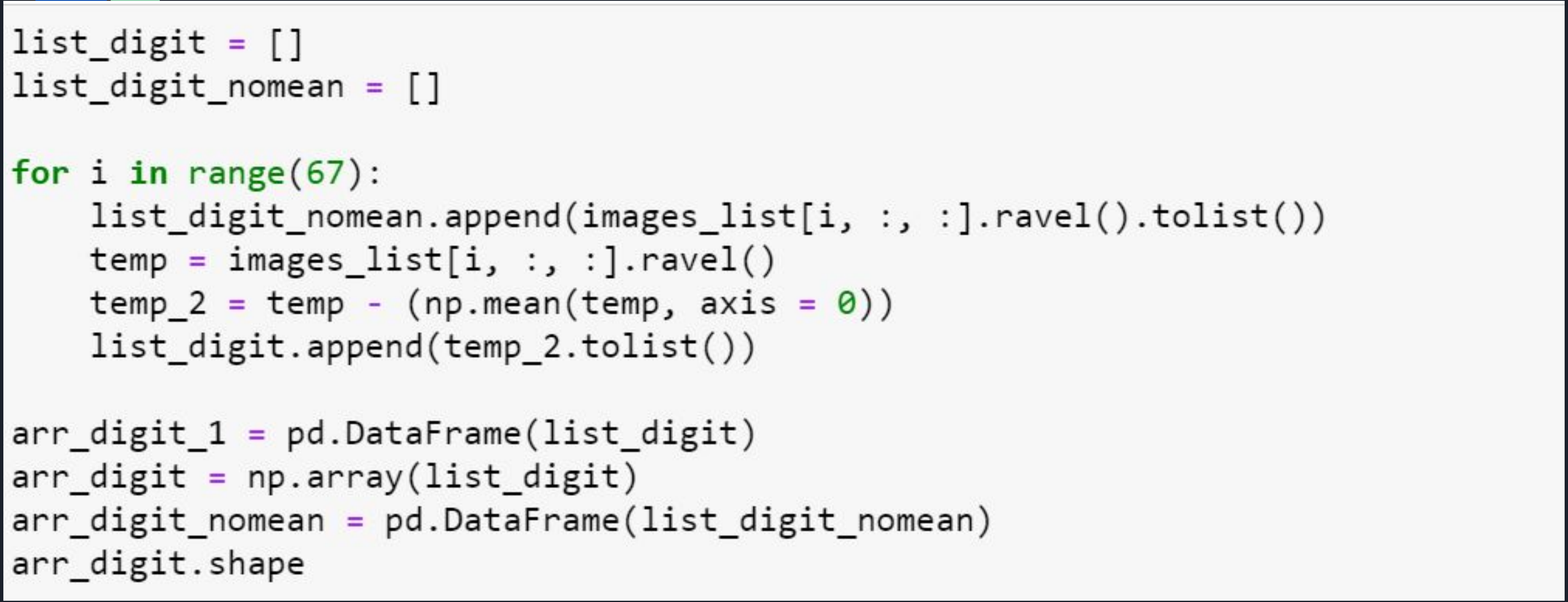


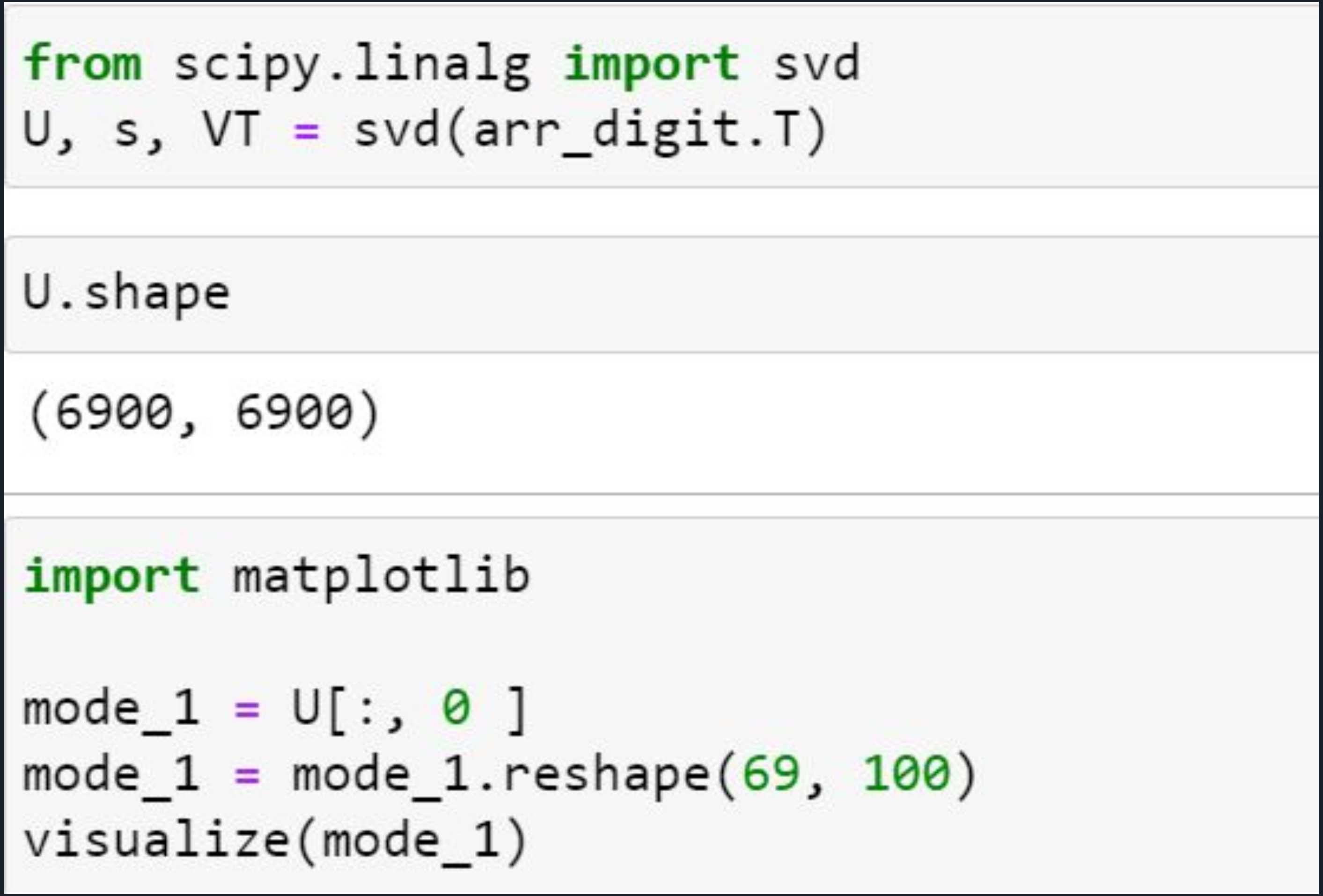
```
import cv2
import cv2
import os
import glob
from skimage.filters import gaussian
from skimage import img_as_ubyte
import numpy as np
```

```
images_list = []
path = r"E:\cropped3\cropped_fully_developed\*.*"
```

```
for file in glob.glob(path):
    print(file)
    img = cv2.imread(file, 0)
    images_list.append(img)

images_list = np.array(images_list)
```

## Preprocessing


```python
list_digit = []
list_digit_nomean = []

for i in range(67):
    list_digit_nomean.append(images_list[i, :, :].ravel().tolist())
    temp = images_list[i, :, :].ravel()
    temp_2 = temp - (np.mean(temp, axis = 0))
    list_digit.append(temp_2.tolist())

arr_digit_1 = pd.DataFrame(list_digit)
arr_digit = np.array(list_digit)
arr_digit_nomean = pd.DataFrame(list_digit_nomean)
arr_digit.shape
```

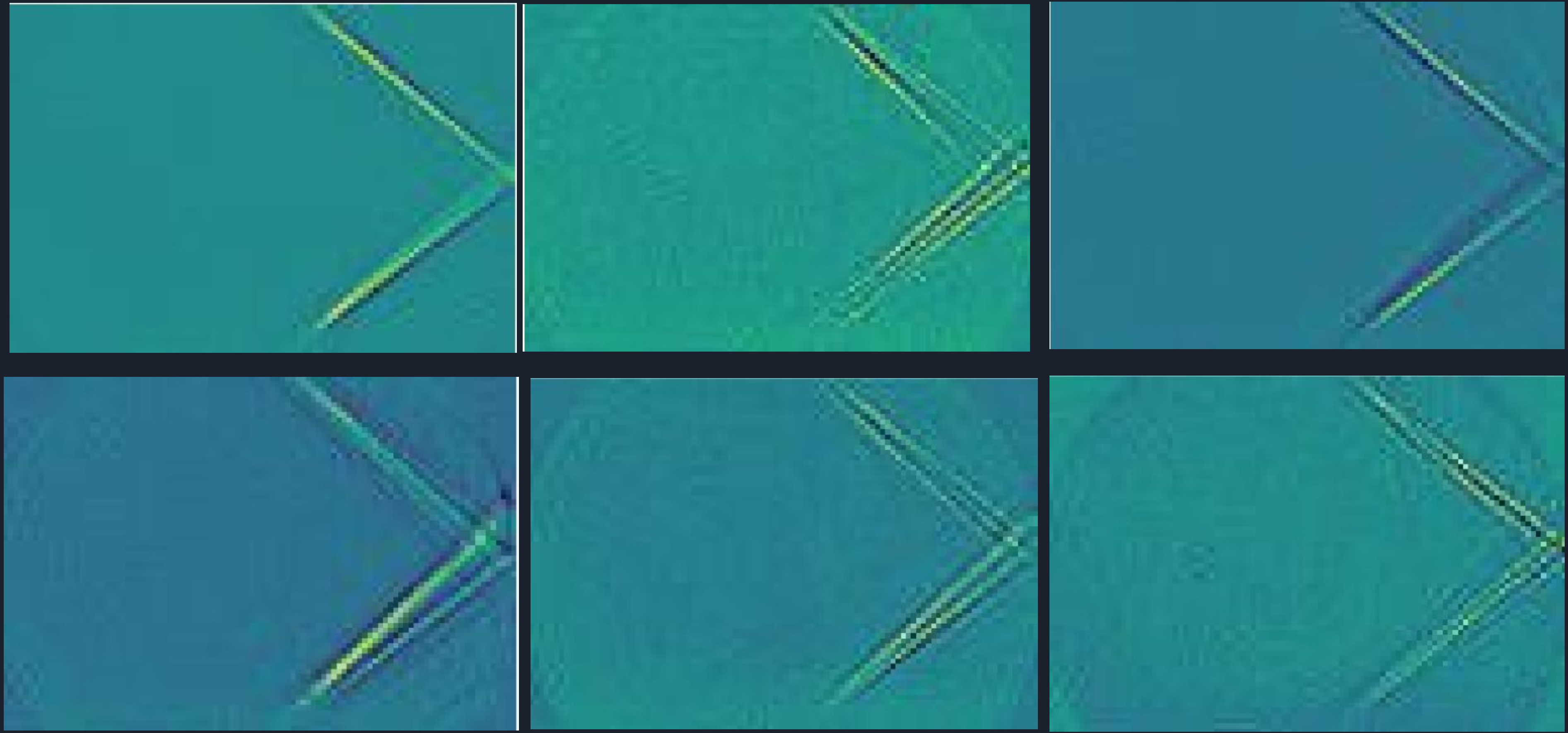


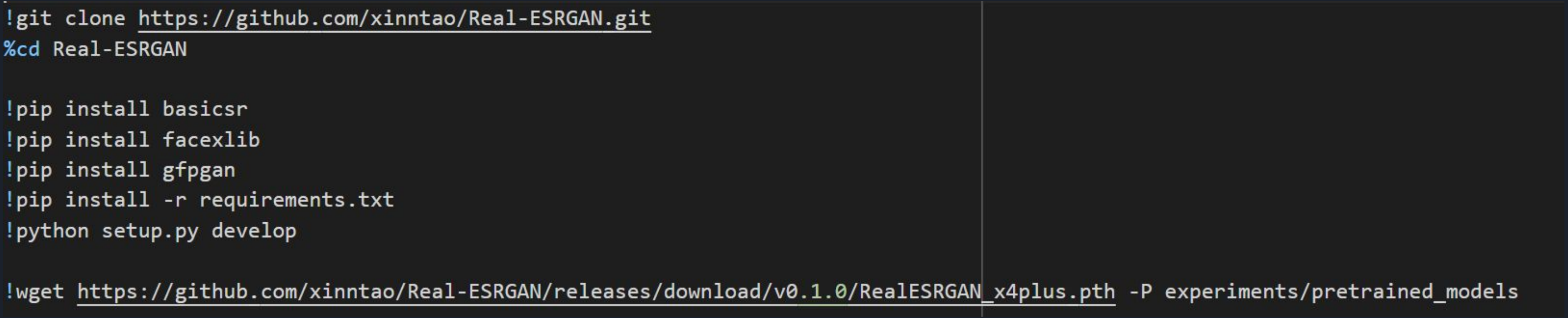
```
from scipy.linalg import svd
U, s, VT = svd(arr_digit.T)
```

```
U.shape
```

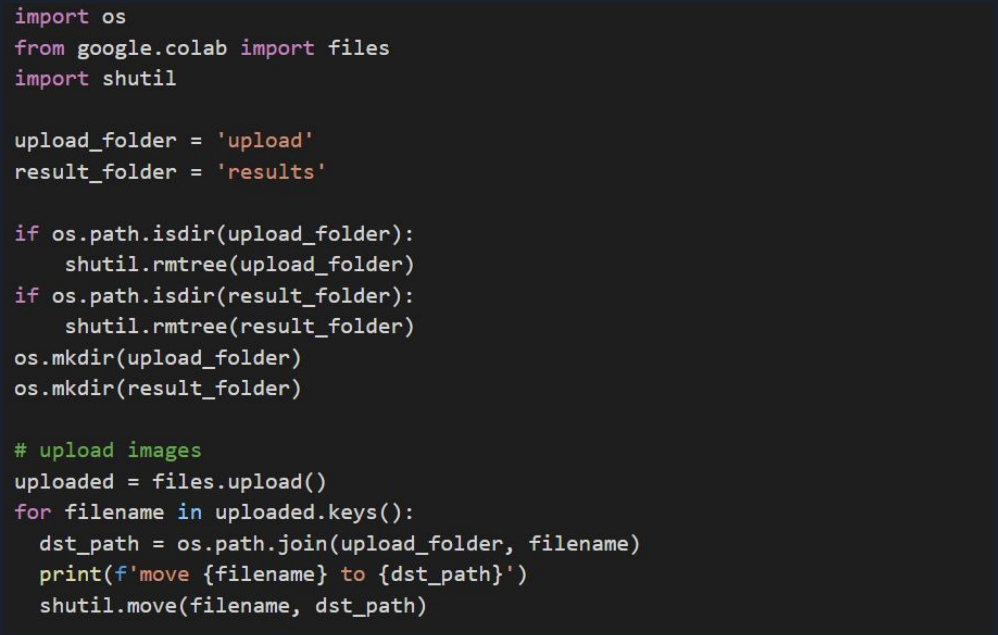
```
(6900, 6900)
```

```
import matplotlib

mode_1 = U[:, 0 ]
mode_1 = mode_1.reshape(69, 100)
visualize(mode_1)
```

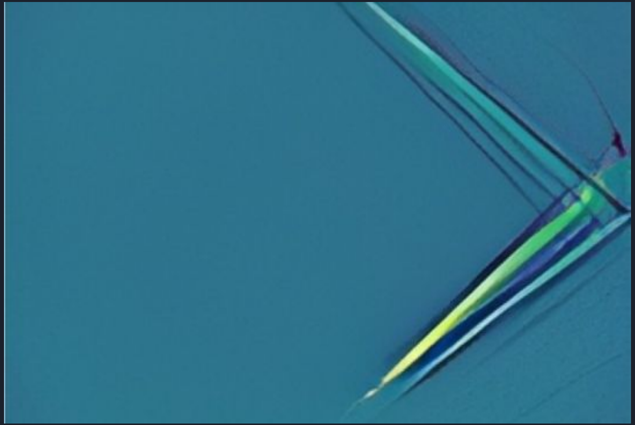
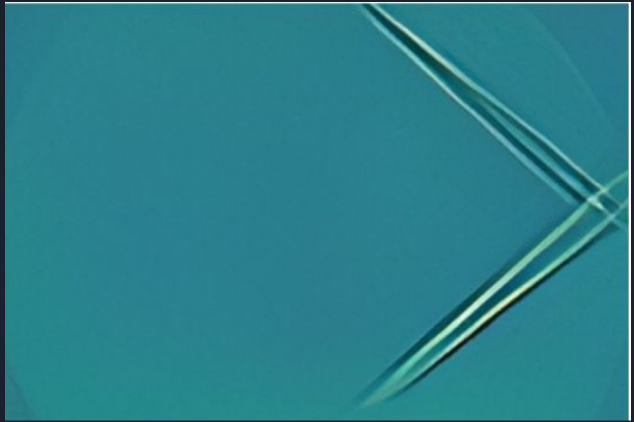
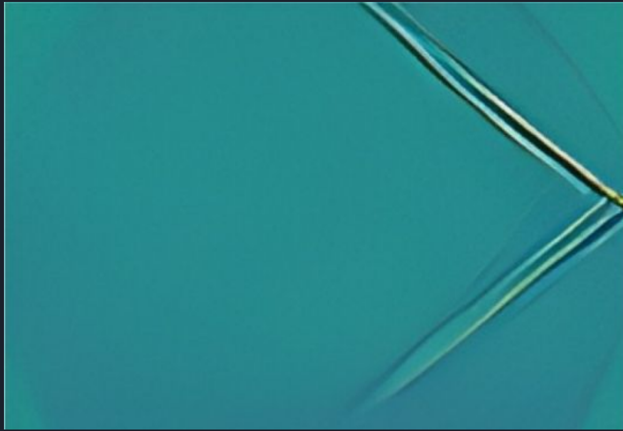




```
!git clone https://github.com/xinntao/Real-ESRGAN.git
%cd Real-ESRGAN

!pip install basicsr
!pip install facexlib
!pip install gfpgan
!pip install -r requirements.txt
!python setup.py develop

!wget https://github.com/xinntao/Real-ESRGAN/releases/download/v0.1.0/RealESRGAN_x4plus.pth -P experiments/pretrained_models
```

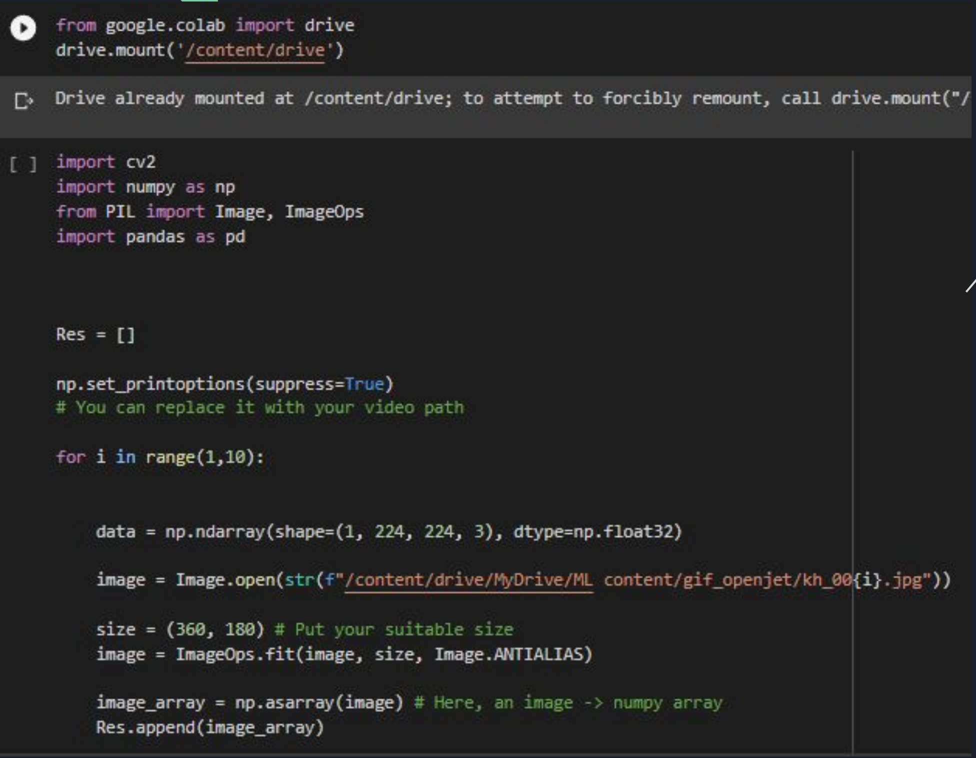


```python
import os
from google.colab import files
import shutil

upload_folder = 'upload'
result_folder = 'results'

if os.path.isdir(upload_folder):
    shutil.rmtree(upload_folder)
if os.path.isdir(result_folder):
    shutil.rmtree(result_folder)
os.mkdir(upload_folder)
os.mkdir(result_folder)

# upload images
uploaded = files.upload()
for filename in uploaded.keys():
  dst_path = os.path.join(upload_folder, filename)
  print(f'move {filename} to {dst_path}')
  shutil.move(filename, dst_path)
```

## POD on Laser Smoke Screen Flow Visualisation

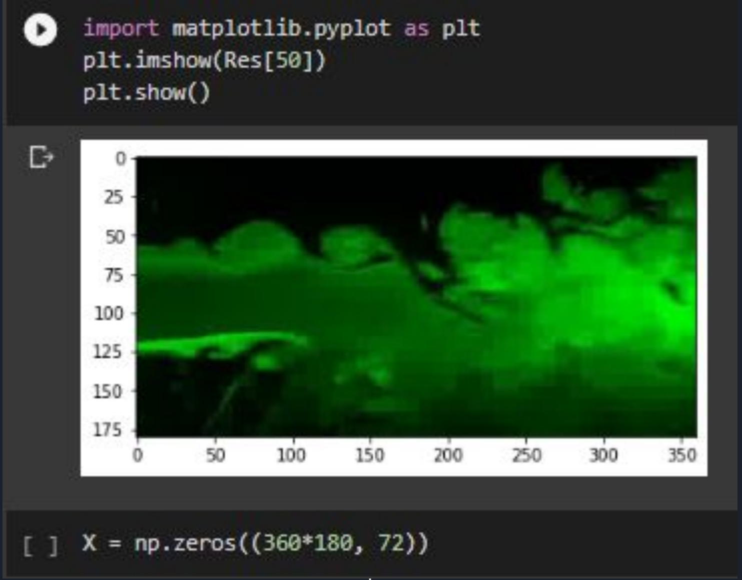
# POD using SVD(reduced) (Laser Smoke Screen method)



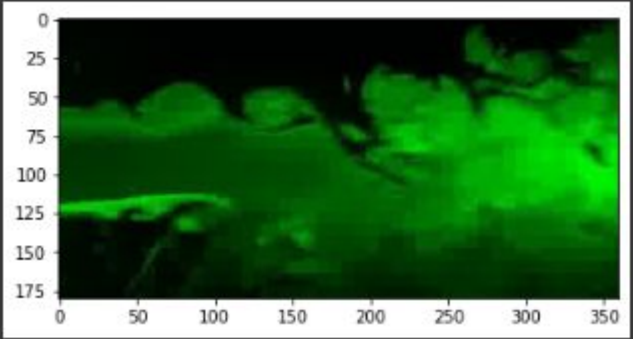
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/
```

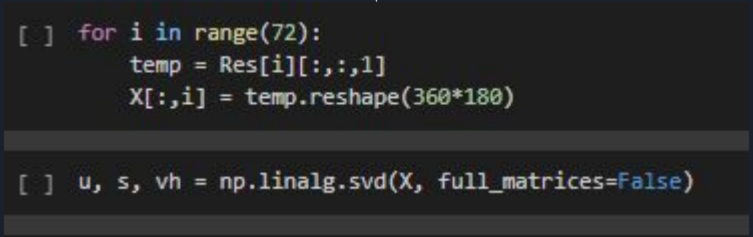
```
[ ] import cv2
    import numpy as np
    from PIL import Image, ImageOps
    import pandas as pd



    Res = []

    np.set_printoptions(suppress=True)
    # You can replace it with your video path

    for i in range(1,10):


        data = np.ndarray(shape=(1, 224, 224, 3), dtype=np.float32)

        image = Image.open(str(f"/content/drive/MyDrive/ML content/gif_openjet/kh_00{i}.jpg"))

        size = (360, 180) # Put your suitable size
        image = ImageOps.fit(image, size, Image.ANTIALIAS)

        image_array = np.asarray(image) # Here, an image -> numpy array
        Res.append(image_array)
```



```
import matplotlib.pyplot as plt
plt.imshow(Res[50])
plt.show()
```



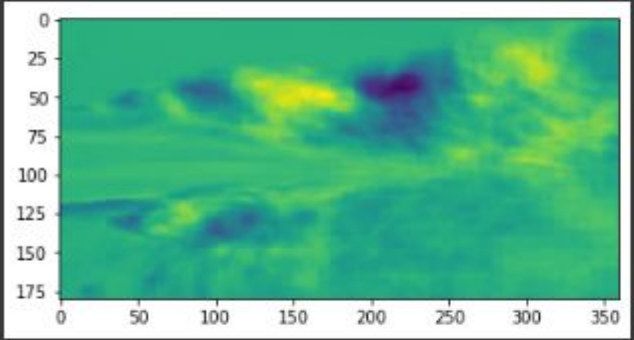
```
[ ] X = np.zeros((360*180, 72))
```



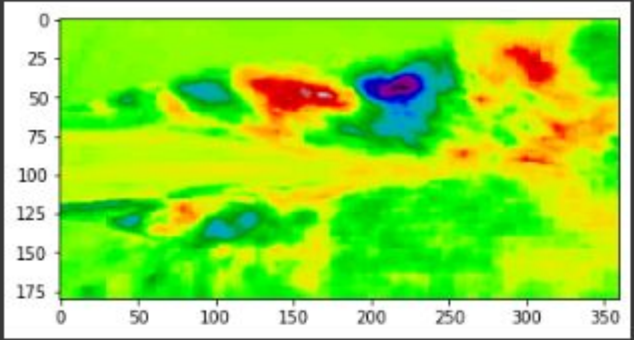
```
[ ] for i in range(72):
        temp = Res[i][:,:,1]
        X[:,i] = temp.reshape(360*180)
```

```
[ ] u, s, vh = np.linalg.svd(X, full_matrices=False)
[ ]  mode_1 = u[:,2].reshape(180,360)
```

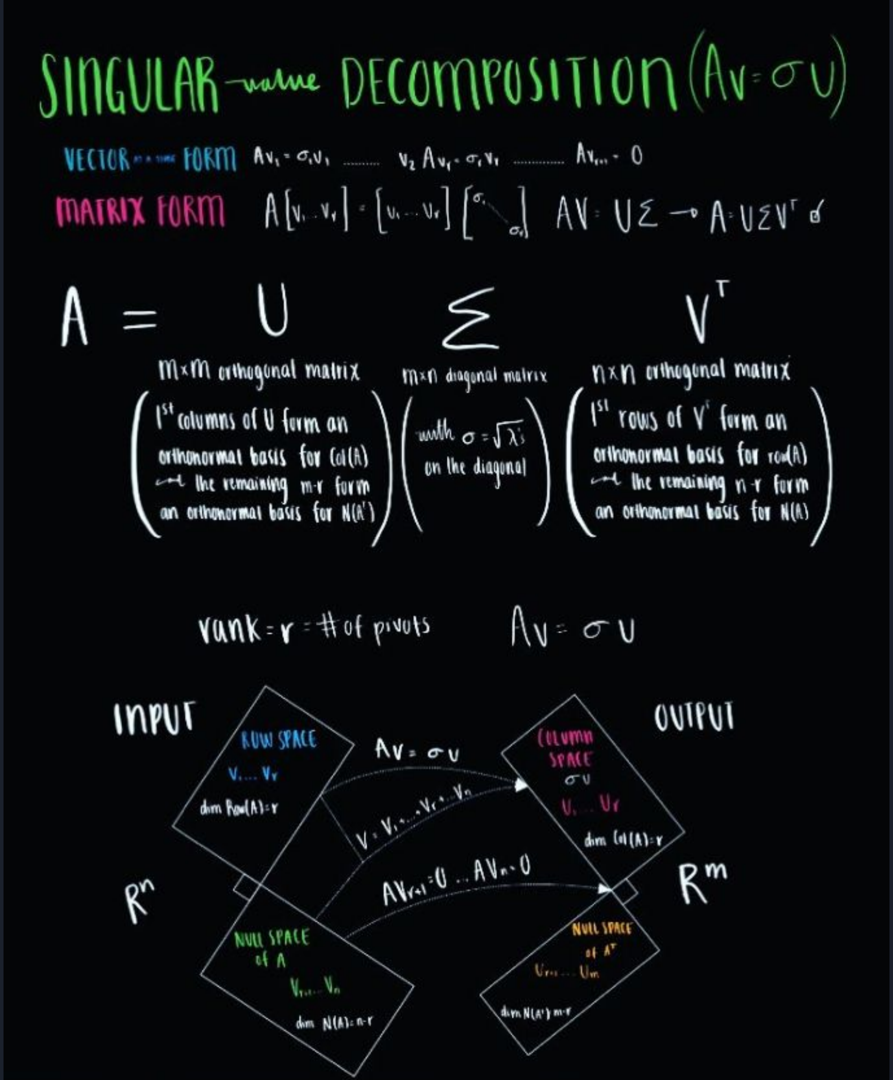
```
[ ]  plt.imshow(mode_1[:,:])
     plt.show()
```



```
[ ]  plt.imshow(mode_1[:,:])

     img = plt.imshow(mode_1[:,:])
     img.set_cmap('nipy_spectral')
     plt.show()
```



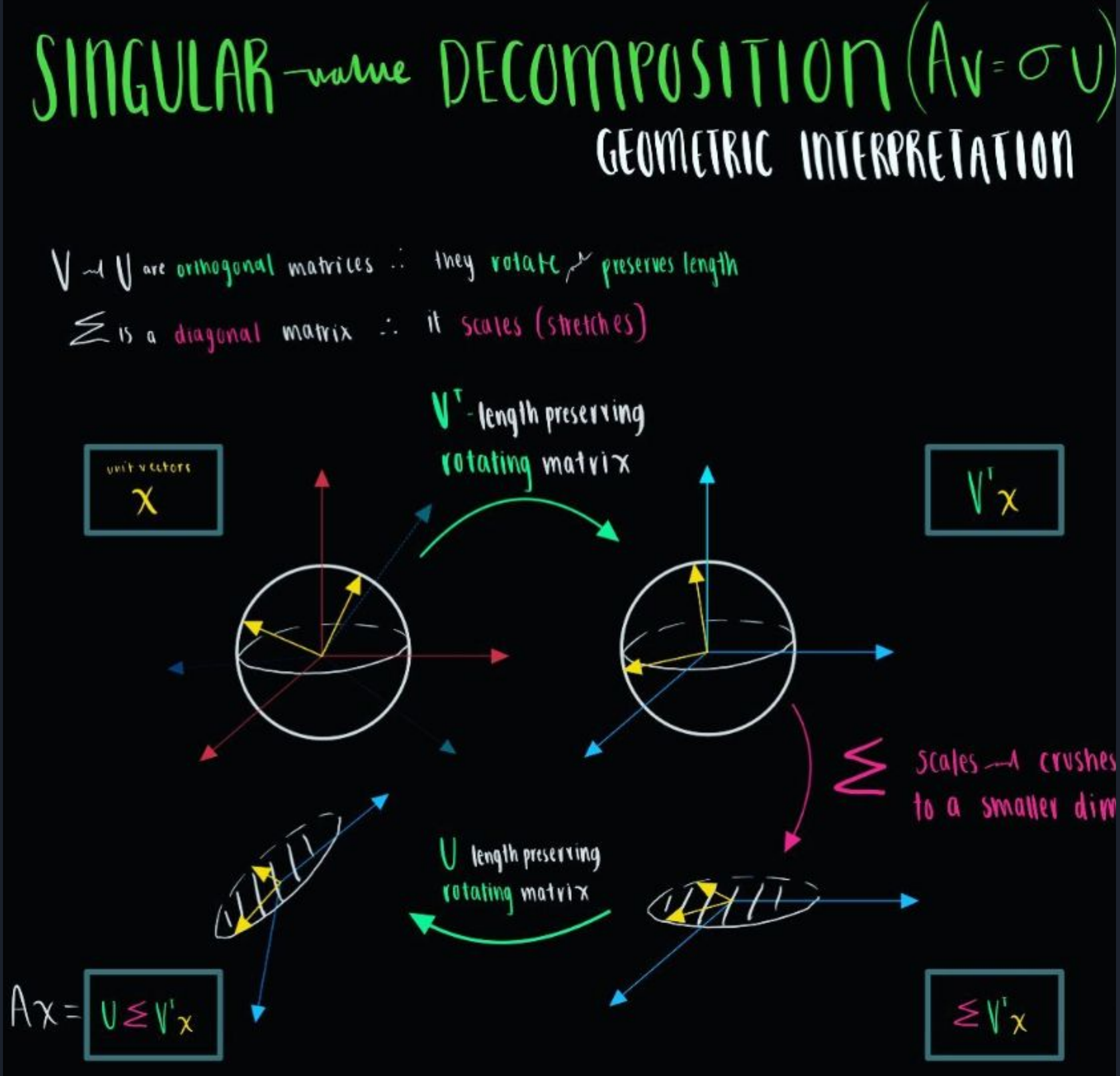
# Method Used here is Singular Value Decomposition (SVD)



SINGULAR-value DECOMPOSITION ($Av = \sigma u$)
VECTOR at a time FORM $Av_1 = \sigma_1 u_1$ ........ $v_2$ $Av_r = \sigma_r u_r$ ........ $Av_{r+1} = 0$
MATRIX FORM $A[v_1 \dots v_r] = [u_1 \dots u_r]\begin{bmatrix}\sigma_1 & & \\ & \ddots & \\ & & \sigma_r\end{bmatrix}$ $AV = U\Sigma \rightarrow A = U\Sigma V^T$
$A = U \Sigma V^T$
$m \times m$ orthogonal matrix
1st columns of U form an orthonormal basis for Col(A) and the remaining m-r form an orthonormal basis for $N(A^T)$
$m \times n$ diagonal matrix
with $\sigma = \sqrt{\lambda_i}$ on the diagonal
$n \times n$ orthogonal matrix
1st rows of $V^T$ form an orthonormal basis for row(A) and the remaining n-r form an orthonormal basis for N(A)
rank = r = # of pivots
$Av = \sigma u$
INPUT
OUTPUT
ROW SPACE
$v_1 \dots v_r$
dim Row(A) = r
$Av = \sigma u$
$v = v_1 \dots v_r \dots v_n$
$Av_{r+1} = 0 \dots Av_n = 0$
COLUMN SPACE
$\sigma u$
$u_1 \dots u_r$
dim Col(A) = r
$R^n$
$R^m$
NULL SPACE of A
$v_{r+1} \dots v_n$
dim N(A) = n-r
NULL SPACE of $A^T$
$u_{r+1} \dots u_m$
dim $N(A^T)$ = m-r

Reference:
Datadiagrams

# Graphical Representation

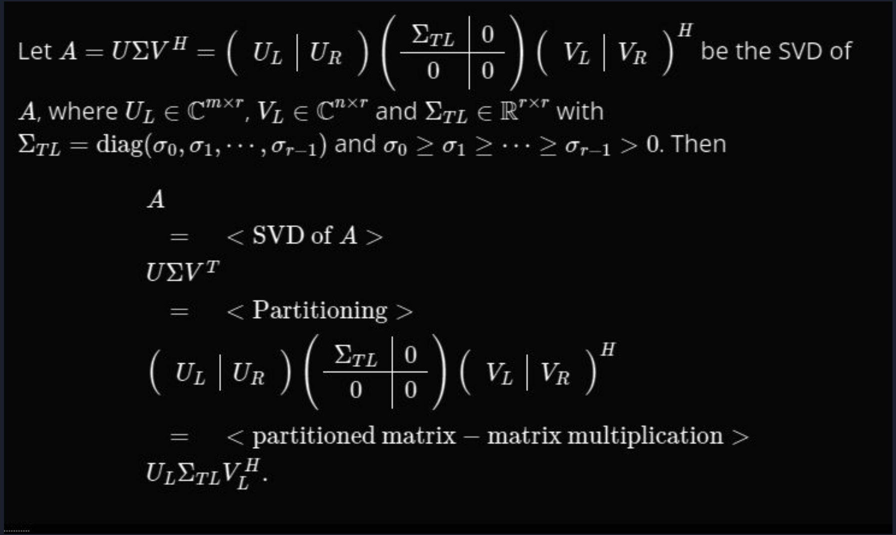

SINGULAR-value DECOMPOSITION $(Av = \sigma u)$
GEOMETRIC INTERPRETATION
V + U are orthogonal matrices ∴ they rotate + preserves length
Σ is a diagonal matrix ∴ it scales (stretches)
$V^T$ - length preserving rotating matrix
unit vectors x
$V^T x$
Σ scales + crushes to a smaller dim
U length preserving rotating matrix
$Ax = U \Sigma V^T x$
$\Sigma V^T x$

Reference:
Datadiagrams

## Reduced SVD

```
u, s, vh = np.linalg.svd(X, full_matrices=False)
```

- the m−r columns that span the left null space are removed from U.
- the padded rows and columns of zeros are removed from Σ.
- the n−r columns that span the null space are removed from V.

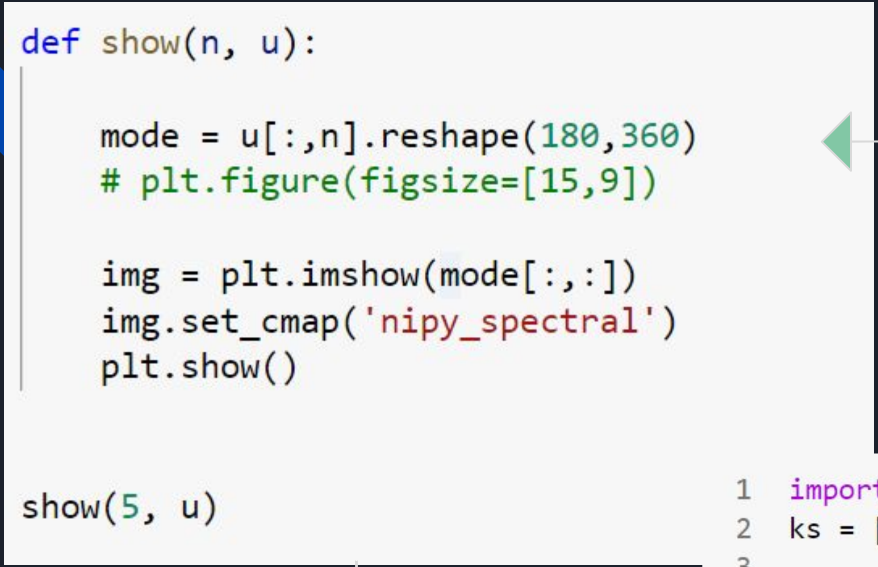


Let $A = U\Sigma V^H = \left( U_L \mid U_R \right) \left( \begin{array}{c|c} \Sigma_{TL} & 0 \\ \hline 0 & 0 \end{array} \right) \left( V_L \mid V_R \right)^H$ be the SVD of $A$, where $U_L \in \mathbb{C}^{m\times r}$, $V_L \in \mathbb{C}^{n\times r}$ and $\Sigma_{TL} \in \mathbb{R}^{r\times r}$ with $\Sigma_{TL} = \mathrm{diag}(\sigma_0, \sigma_1, \cdots, \sigma_{r-1})$ and $\sigma_0 \geq \sigma_1 \geq \cdots \geq \sigma_{r-1} > 0$. Then

$$\begin{array}{l} A \\ \quad = \quad < \text{SVD of } A > \\ U\Sigma V^T \\ \quad = \quad < \text{Partitioning} > \\ \left( U_L \mid U_R \right) \left( \begin{array}{c|c} \Sigma_{TL} & 0 \\ \hline 0 & 0 \end{array} \right) \left( V_L \mid V_R \right)^H \\ \quad = \quad < \text{partitioned matrix} - \text{matrix multiplication} > \\ U_L \Sigma_{TL} V_L^H. \end{array}$$

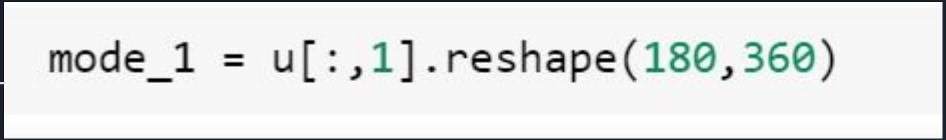
Both reduced SVD and full SVD results in the original A with no information loss.

Source: https://www.cs.utexas.edu/users/flame/laff/alaff/chapter02-the-reduced-svd.html

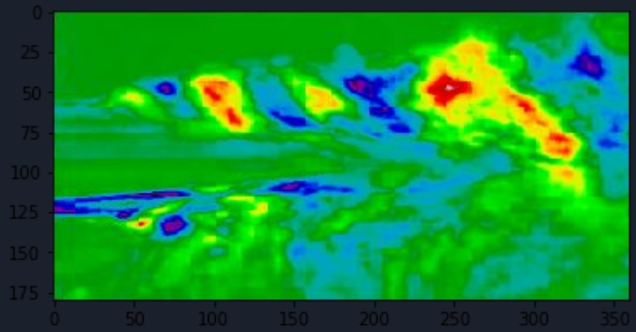


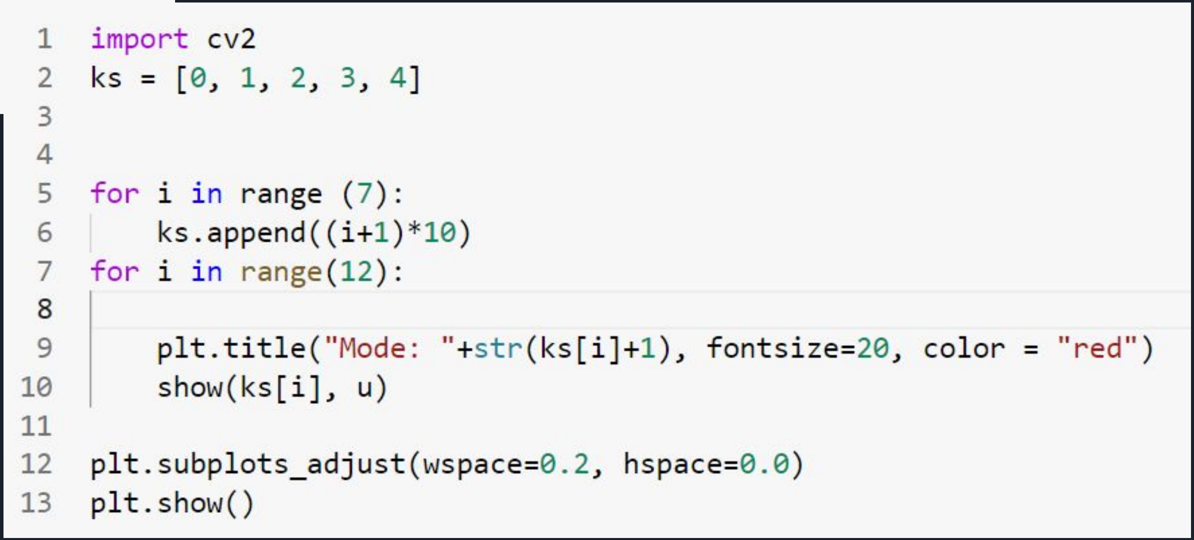
```python
def show(n, u):

    mode = u[:,n].reshape(180,360)
    # plt.figure(figsize=[15,9])

    img = plt.imshow(mode[:,:])
    img.set_cmap('nipy_spectral')
    plt.show()
```

```python
show(5, u)
```

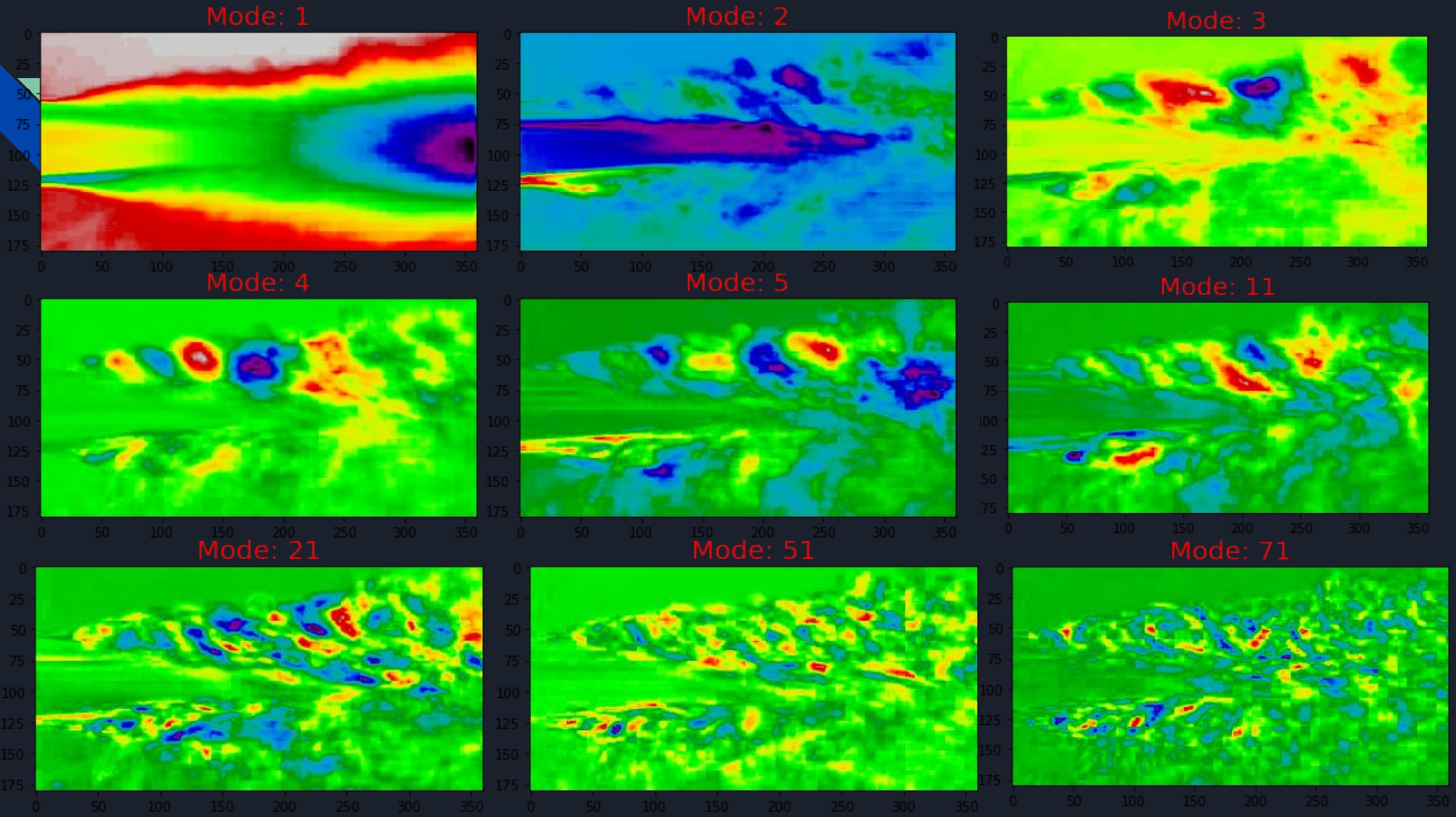


```python
mode_1 = u[:,1].reshape(180,360)
```





```python
import cv2
ks = [0, 1, 2, 3, 4]


for i in range (7):
    ks.append((i+1)*10)
for i in range(12):

    plt.title("Mode: "+str(ks[i]+1), fontsize=20, color = "red")
    show(ks[i], u)

plt.subplots_adjust(wspace=0.2, hspace=0.0)
plt.show()
```


Mode: 1
Mode: 2
Mode: 3
Mode: 4
Mode: 5
Mode: 11
Mode: 21
Mode: 51
Mode: 71
0
25
50
75
100
125
150
175
0
50
100
150
200
250
300
350

## POD using Covariance Matrix

## Proper Orthogonal Decomposition (Covariance Matrix Method)

The method of covariance calculation for the use in proper orthogonal decomposition is mainly used as a statistical method contrary to the SVD method which is mainly used as a tool of linear algebra.

In this method, the first goal is to convert the given images into a linear array. The input images for this case (shock wave images) are of the size 1920 x 1080 pixels. These are first resolved into a smaller dimension of 100 x 69 pixels. This is done so as to reduce the calculative complexity (memory and time) that is faced during the covariance matrix generation and calculation.

The entire input image can be considered to be an array of the dimension 100 x 69. This can be converter into a linear array of the size 1 x 6900 elements. This process is repeated for all the images (124 images of final generated shock) to create a final 2D array of dimensions 124 x 6900. This array represents the entire set of useful images.

$$Image\,1 = \begin{bmatrix} x_{11} & x_{12} & x_{13} & \cdots & x_{1n/d} \\ x_{21} & x_{22} & x_{23} & \cdots & x_{2n/d} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_{d1} & x_{d2} & x_{d3} & \cdots & x_{n} \end{bmatrix}$$

$$\longrightarrow$$

$$Image\,1 = \begin{bmatrix} p_{11} & p_{12} & p_{13} & \cdots & p_{1n} \end{bmatrix}$$

$$Image\,2 = \begin{bmatrix} p_{21} & p_{22} & p_{23} & \cdots & p_{2n} \end{bmatrix}$$

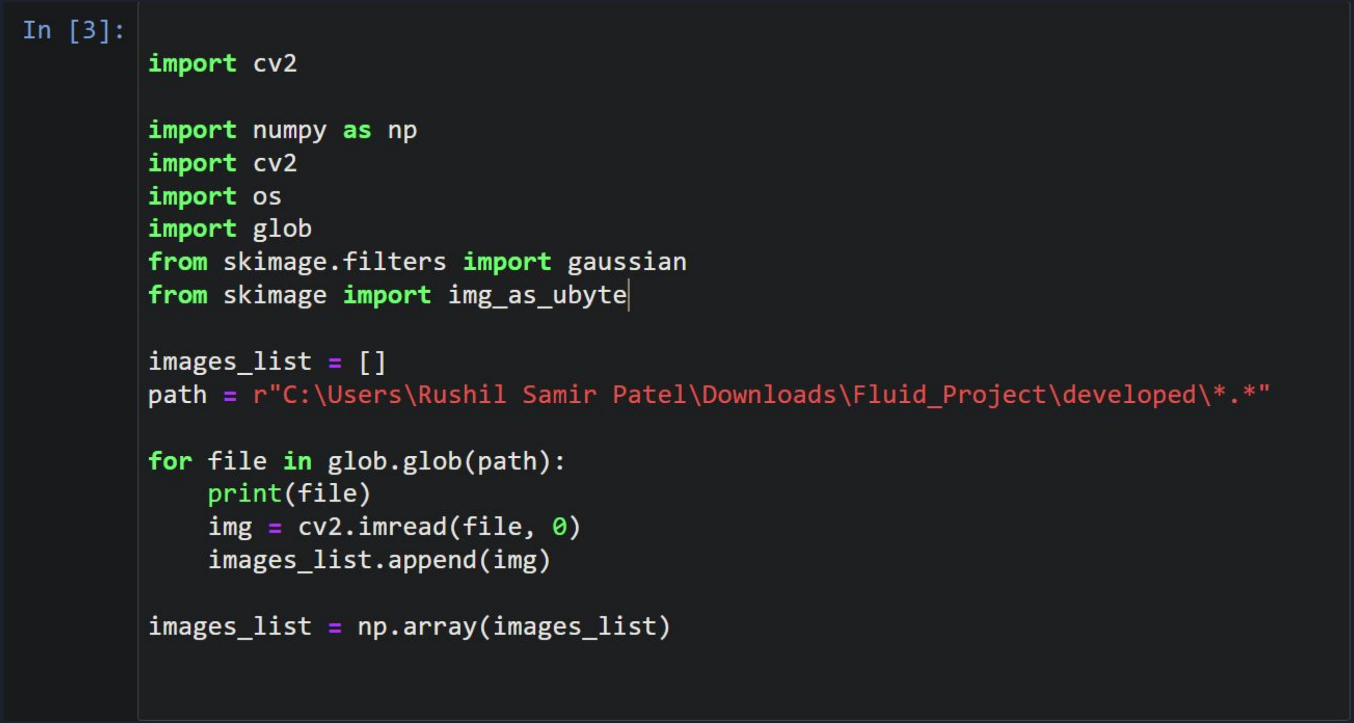
$$Image\,i = \begin{bmatrix} \cdots & \cdots & \cdots & \cdots & \cdots \end{bmatrix}$$

$$Image\,m = \begin{bmatrix} p_{m1} & p_{m2} & p_{m3} & \cdots & p_{mn} \end{bmatrix}$$

$$\longleftarrow$$

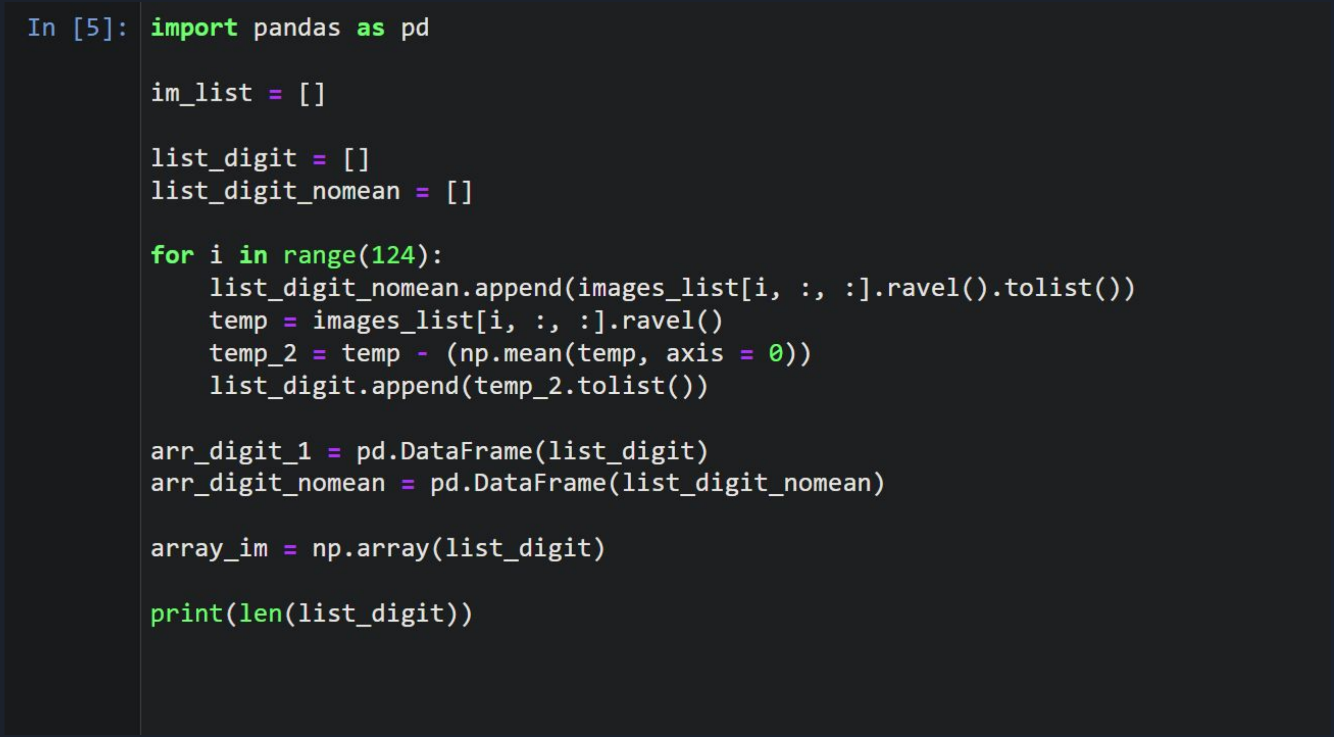
$$Images = \begin{bmatrix} p_{11} & p_{12} & p_{13} & \cdots & p_{1n} \\ p_{21} & p_{22} & p_{23} & \cdots & p_{2n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ p_{m1} & p_{m2} & p_{m3} & \cdots & p_{mn} \end{bmatrix}$$

## Importing of Useful Images



```
In [3]:
import cv2

import numpy as np
import cv2
import os
import glob
from skimage.filters import gaussian
from skimage import img_as_ubyte

images_list = []
path = r"C:\Users\Rushil Samir Patel\Downloads\Fluid_Project\developed\*.*"

for file in glob.glob(path):
    print(file)
    img = cv2.imread(file, 0)
    images_list.append(img)

images_list = np.array(images_list)
```

## Linearisation, Stacking and Mean Subtraction of Image Pixels



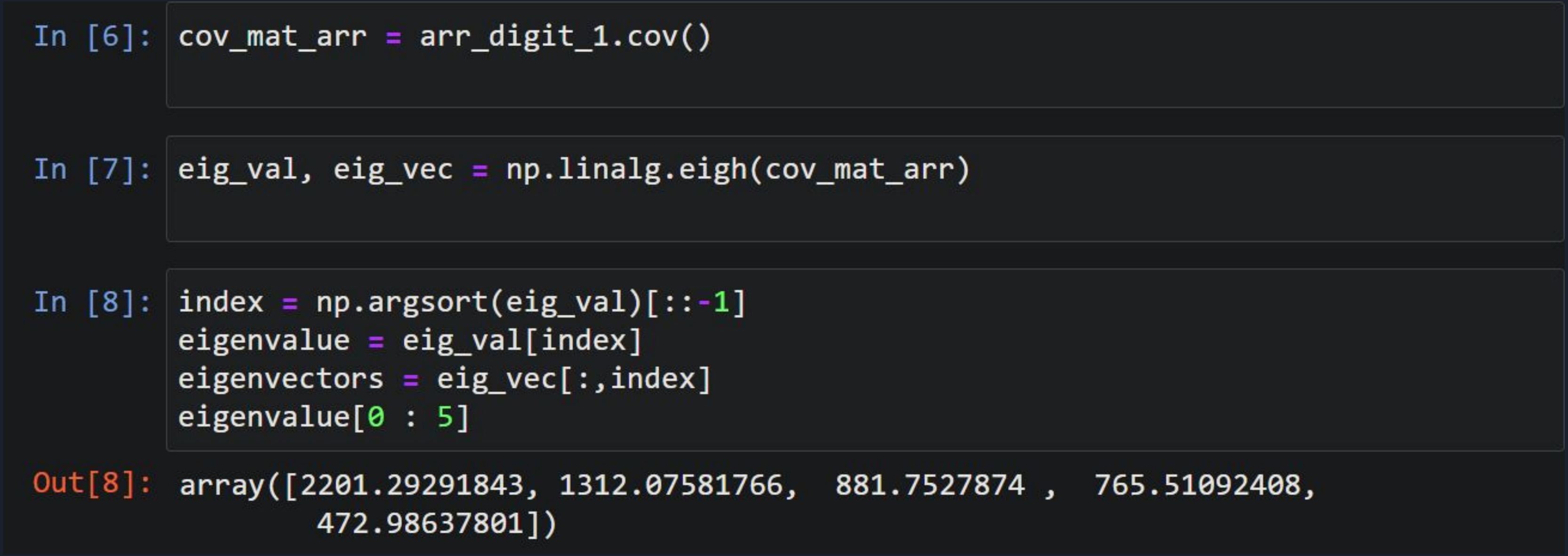
```
In [5]: import pandas as pd

        im_list = []

        list_digit = []
        list_digit_nomean = []

        for i in range(124):
            list_digit_nomean.append(images_list[i, :, :].ravel().tolist())
            temp = images_list[i, :, :].ravel()
            temp_2 = temp - (np.mean(temp, axis = 0))
            list_digit.append(temp_2.tolist())

        arr_digit_1 = pd.DataFrame(list_digit)
        arr_digit_nomean = pd.DataFrame(list_digit_nomean)

        array_im = np.array(list_digit)

        print(len(list_digit))
```

Next, the covariance matrix is calculated. The covariance matrix calculation can be done by:

$$Covariance\ matrix = \frac{1}{n-1}\Sigma\left(x_i - x'\right)\left(x_i - x'\right)^T$$

The covariance matrix in our case has a dimension of 6900 x 6900. Next, we find the eigen vectors and eigen values of the covariance matrix. The eigen vectors arranged in the descending order of the eigen values gives the modes arranged in the most dominating to least dominating order.

## Covariance matrix, Eigenvector and Eigen Value Calculation

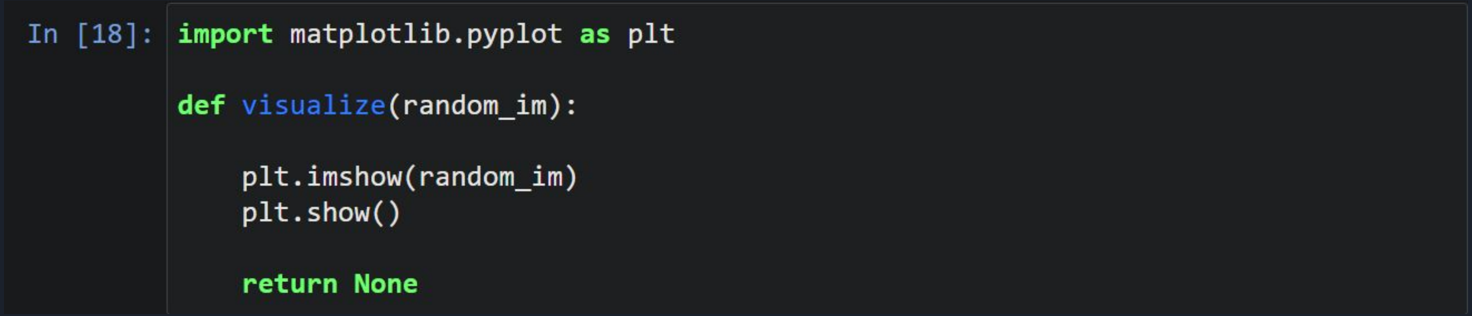


```
In [6]: cov_mat_arr = arr_digit_1.cov()
```

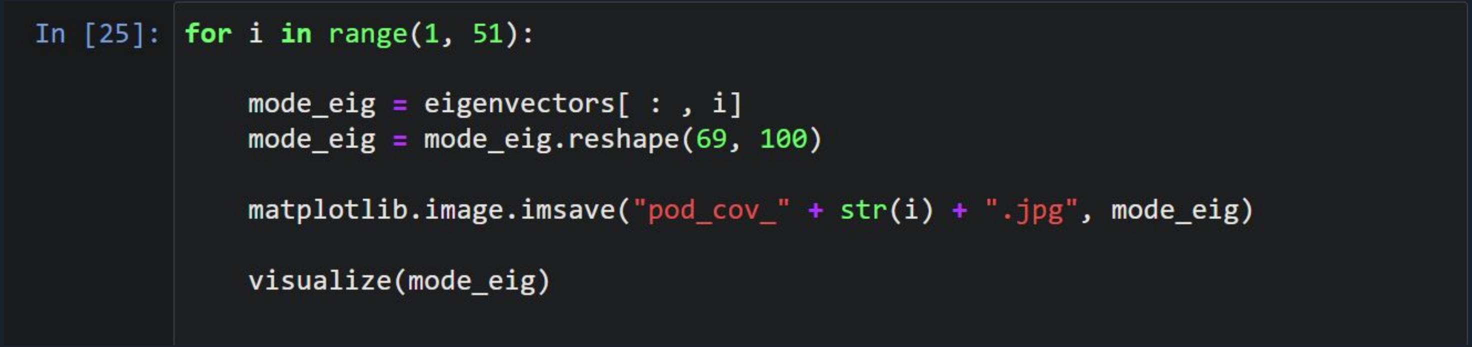
```
In [7]: eig_val, eig_vec = np.linalg.eigh(cov_mat_arr)
```

```
In [8]: index = np.argsort(eig_val)[::-1]
        eigenvalue = eig_val[index]
        eigenvectors = eig_vec[:,index]
        eigenvalue[0 : 5]

Out[8]: array([2201.29291843, 1312.07581766,  881.7527874 ,  765.51092408,
                472.98637801])
```

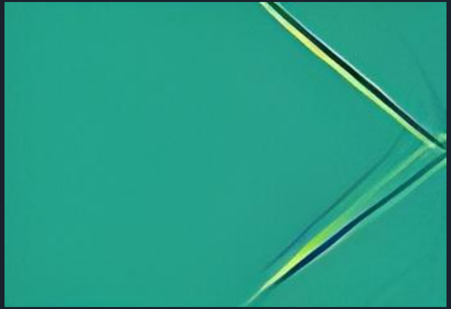
## Visualising and Saving of first 50 Modes

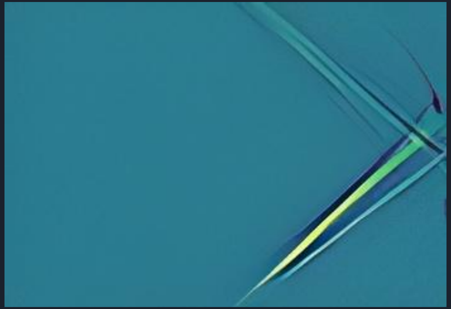


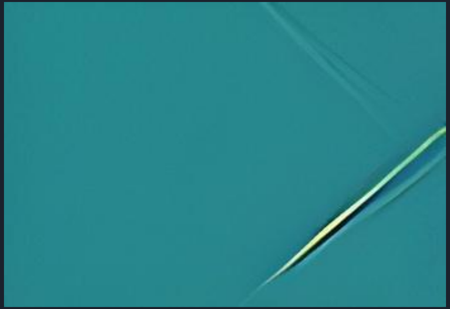
```
In [18]: import matplotlib.pyplot as plt

         def visualize(random_im):

             plt.imshow(random_im)
             plt.show()

             return None
```

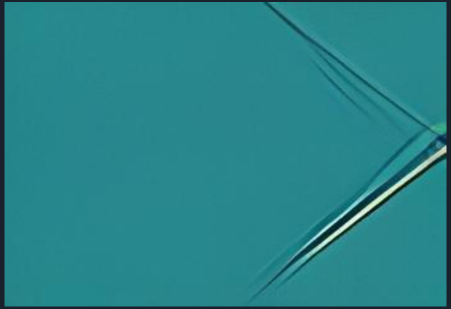


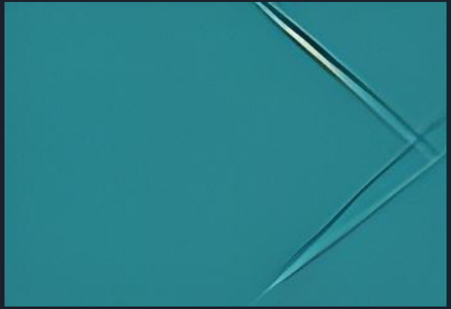
```
In [25]: for i in range(1, 51):

             mode_eig = eigenvectors[ : , i]
             mode_eig = mode_eig.reshape(69, 100)

             matplotlib.image.imsave("pod_cov_" + str(i) + ".jpg", mode_eig)

             visualize(mode_eig)
```

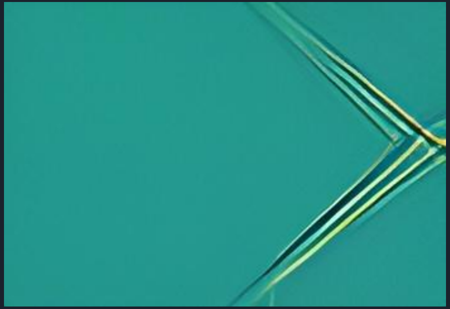

Mode 1


Mode 2


Mode 3


Mode 4


Mode 5


Mode 6

## Issues Encountered During Finding the POD from Covariance Matrix

A few issues encountered during the processing of the images were:

1. Using the original 1920 x 1080 pixel images causes memory overflow in the programming hardware due to the formation of 20,73,600 x 20,73,600 matrix which required an unachievable processing memory of 15 Tb.
2. The next issue encountered was that initially, all of the images were considered for the POD formation. Though this method is completely suitable for PCA, this method is not suitable for POD because of the time dependence of all images. Considering all images resulted in modes which were overlap of modes of all different images including the startup, transition and final shock.
3. The last issue encountered was about the resolution of the final image. These images to a certain extent were incomprehensible because of their lower resolution. This was solved by using the aforementioned resolution enhancing algorithm.

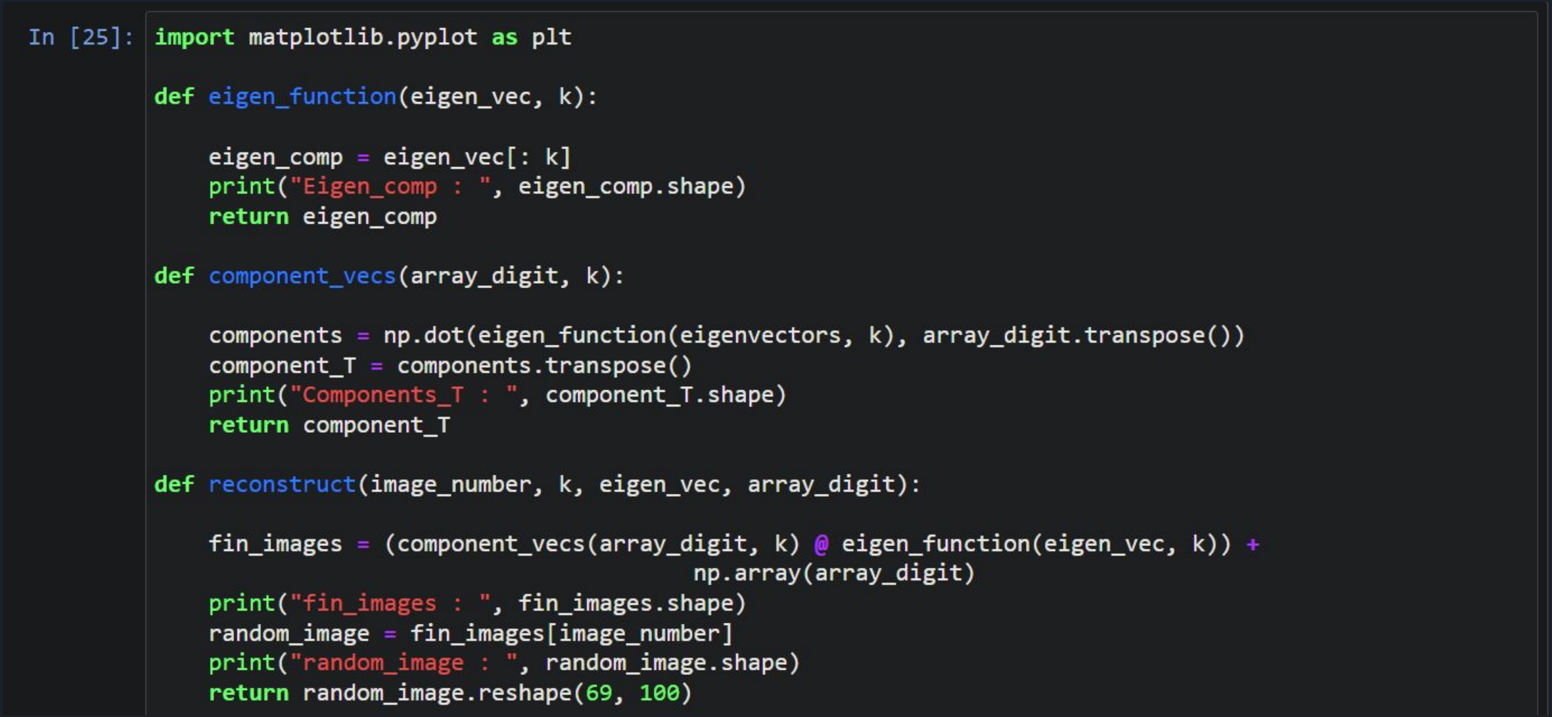
This particular covariance matrix can further be used to do principal component analysis. In this, any top ‘k’ eigen vectors in the sorted fashion are taken, transposed and multiplied with the original image to facilitate dimensionality reduction.

The images thus generated will only contain the information related to the top ‘k’ modes of the images. Here ‘k’ can be any number of dimensions less than or equal to the the initial feature dimensions.

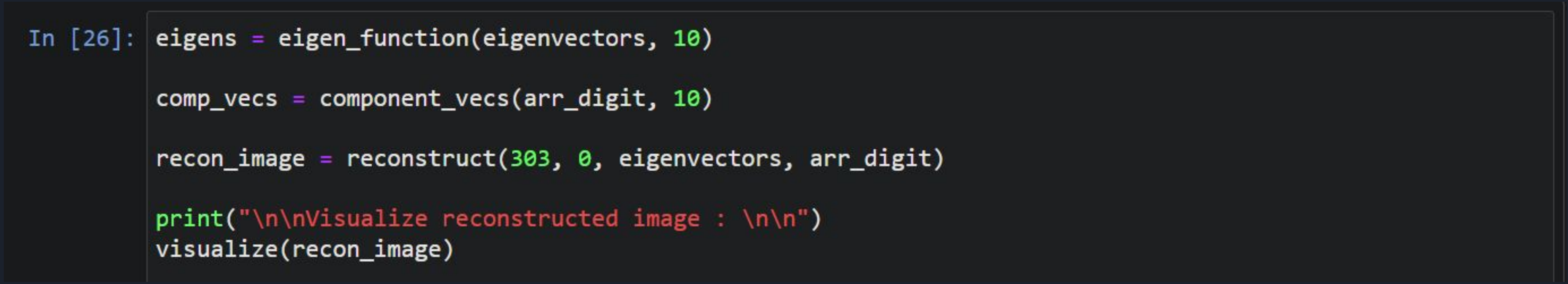
Further, the first mode obtained has been used to generate a saliency mapped image. In this, the aim is to separate the shock wave structure in the image from the background which is not actually necessary to extract any useful information.

For this, an unsupervised learning algorithm known as the k-means clustering is used. In this, regions of similar colour gradients are clubbed together. Next, the image is thresholded to generate a black and white image. By the use of some angle finding algorithms, the angle enclosed between the shock waves is found. This can be used to generate the mach number of the flow computationally.

## Principle Component Analysis



```
In [25]: import matplotlib.pyplot as plt

         def eigen_function(eigen_vec, k):

             eigen_comp = eigen_vec[: k]
             print("Eigen_comp : ", eigen_comp.shape)
             return eigen_comp

         def component_vecs(array_digit, k):

             components = np.dot(eigen_function(eigenvectors, k), array_digit.transpose())
             component_T = components.transpose()
             print("Components_T : ", component_T.shape)
             return component_T

         def reconstruct(image_number, k, eigen_vec, array_digit):

             fin_images = (component_vecs(array_digit, k) @ eigen_function(eigen_vec, k)) +
                                                 np.array(array_digit)
             print("fin_images : ", fin_images.shape)
             random_image = fin_images[image_number]
             print("random_image : ", random_image.shape)
             return random_image.reshape(69, 100)
```



```
In [26]: eigens = eigen_function(eigenvectors, 10)

         comp_vecs = component_vecs(arr_digit, 10)

         recon_image = reconstruct(303, 0, eigenvectors, arr_digit)

         print("\n\nVisualize reconstructed image : \n\n")
         visualize(recon_image)
```

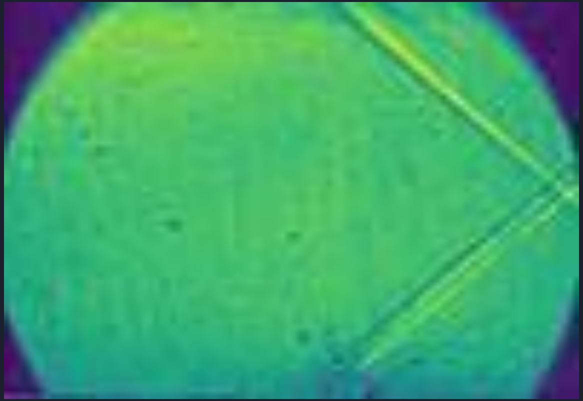
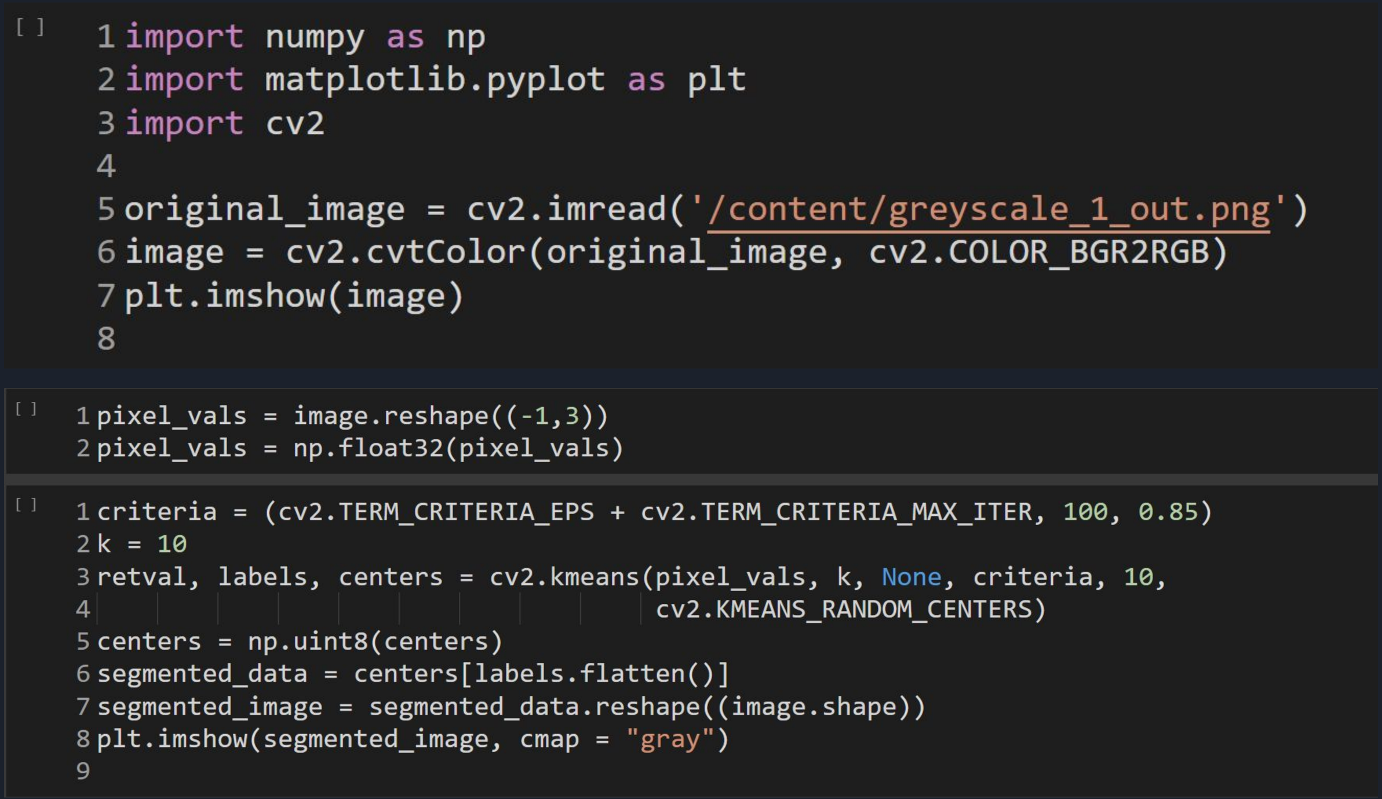


Image with PCA with k  = 10

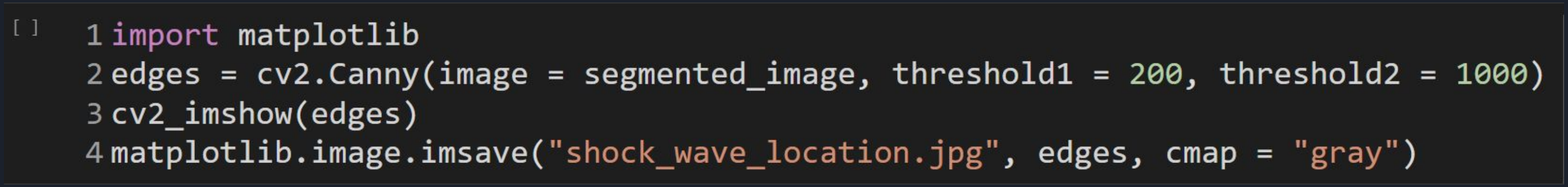
## Generating Clustered Image



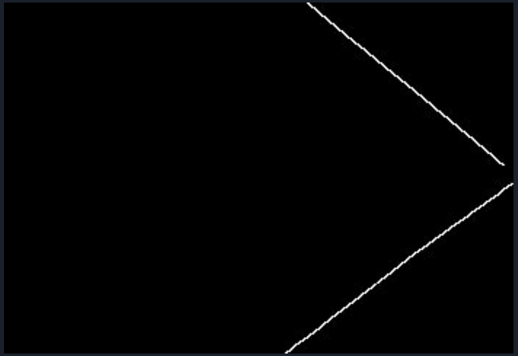
```
[ ]
import numpy as np
import matplotlib.pyplot as plt
import cv2

original_image = cv2.imread('/content/greyscale_1_out.png')
image = cv2.cvtColor(original_image, cv2.COLOR_BGR2RGB)
plt.imshow(image)

```

```
[ ]
pixel_vals = image.reshape((-1,3))
pixel_vals = np.float32(pixel_vals)
```

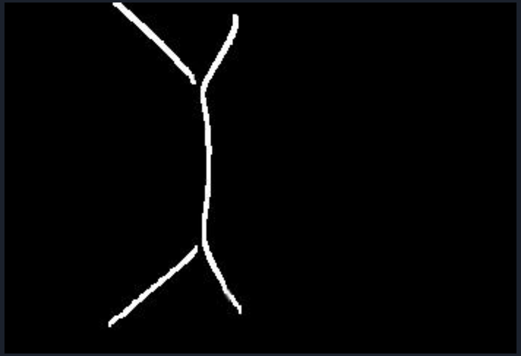
```
[ ]
criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 100, 0.85)
k = 10
retval, labels, centers = cv2.kmeans(pixel_vals, k, None, criteria, 10,
                                      cv2.KMEANS_RANDOM_CENTERS)
centers = np.uint8(centers)
segmented_data = centers[labels.flatten()]
segmented_image = segmented_data.reshape((image.shape))
plt.imshow(segmented_image, cmap = "gray")

```

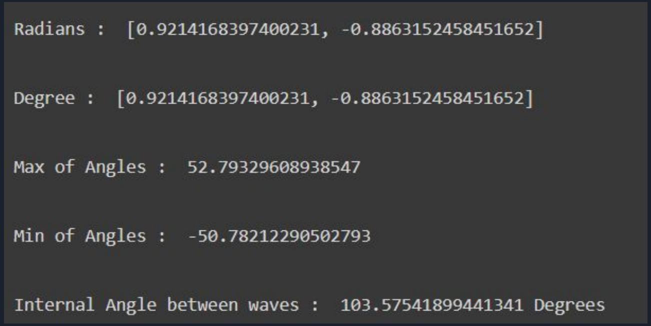


```
import matplotlib
edges = cv2.Canny(image = segmented_image, threshold1 = 200, threshold2 = 1000)
cv2_imshow(edges)
matplotlib.image.imsave("shock_wave_location.jpg", edges, cmap = "gray")
```



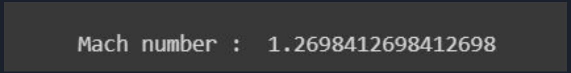
-Results-





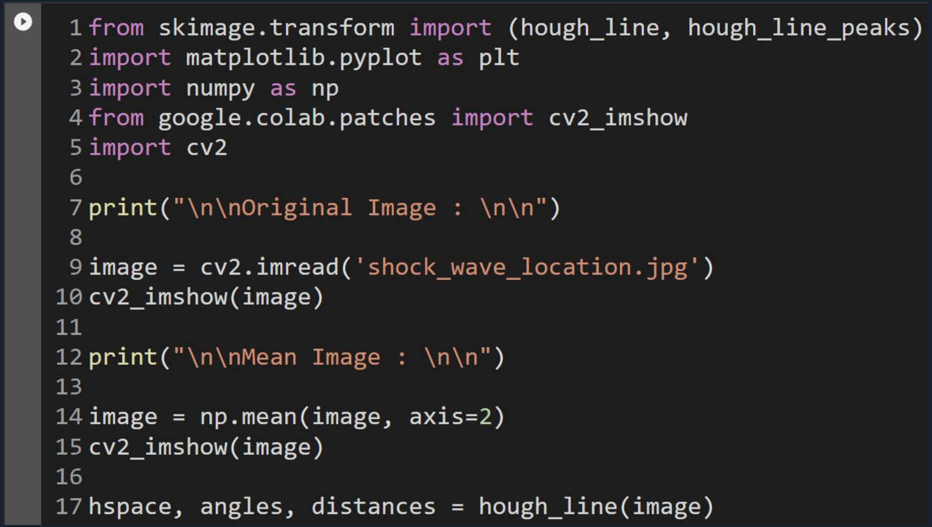
```
Radians :  [0.9214168397400231, -0.8863152458451652]

Degree :  [0.9214168397400231, -0.8863152458451652]

Max of Angles :  52.79329608938547

Min of Angles :  -50.78212290502793

Internal Angle between waves :  103.57541899441341 Degrees
```

$\longrightarrow$

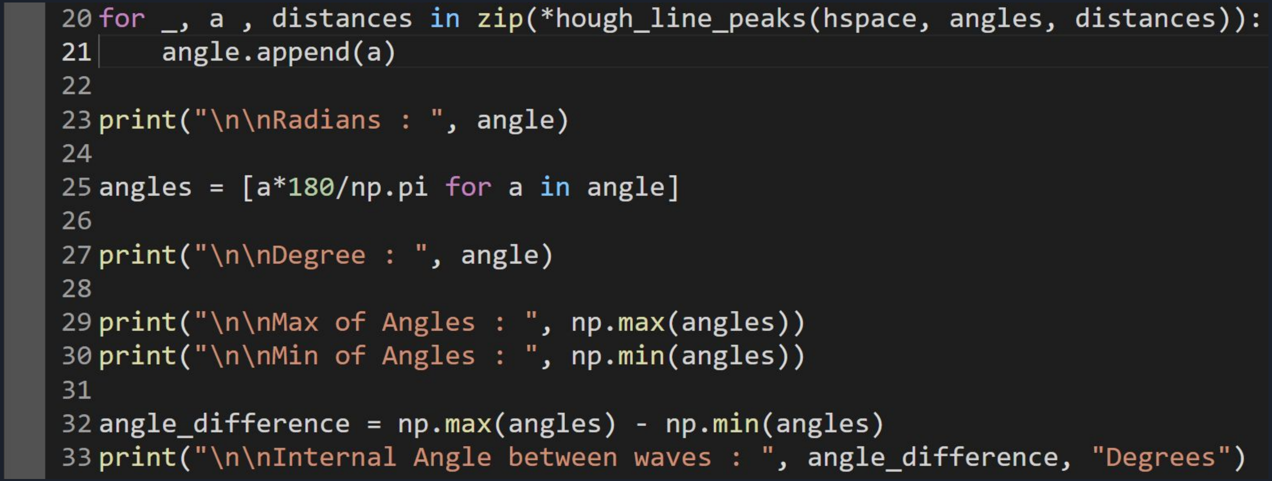


```
Mach number :  1.2698412698412698
```

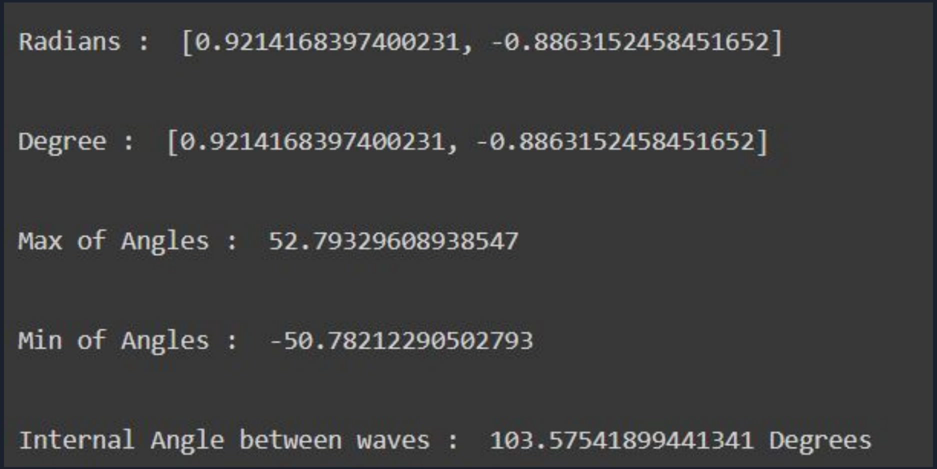
Source : Edge Detection Using OpenCV | LearnOpenCV #



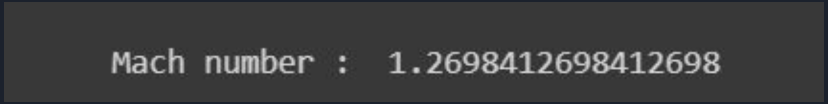
```python
from skimage.transform import (hough_line, hough_line_peaks)
import matplotlib.pyplot as plt
import numpy as np
from google.colab.patches import cv2_imshow
import cv2

print("\n\nOriginal Image : \n\n")

image = cv2.imread('shock_wave_location.jpg')
cv2_imshow(image)

print("\n\nMean Image : \n\n")

image = np.mean(image, axis=2)
cv2_imshow(image)

hspace, angles, distances = hough_line(image)
```



```python
for _, a , distances in zip(*hough_line_peaks(hspace, angles, distances)):
    angle.append(a)

print("\n\nRadians : ", angle)

angles = [a*180/np.pi for a in angle]

print("\n\nDegree : ", angle)

print("\n\nMax of Angles : ", np.max(angles))
print("\n\nMin of Angles : ", np.min(angles))

angle_difference = np.max(angles) - np.min(angles)
print("\n\nInternal Angle between waves : ", angle_difference, "Degrees")
```



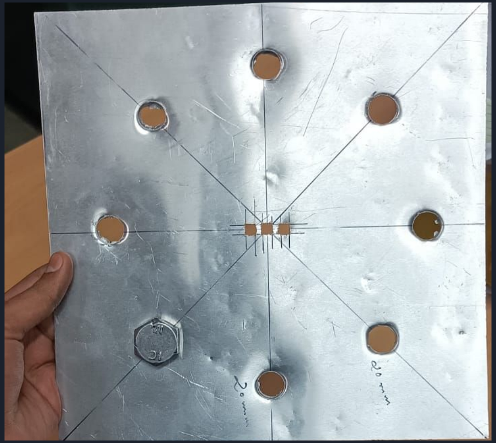
```
Radians :  [0.9214168397400231, -0.8863152458451652]

Degree :  [0.9214168397400231, -0.8863152458451652]

Max of Angles :  52.79329608938547

Min of Angles :  -50.78212290502793

Internal Angle between waves :  103.57541899441341 Degrees
```



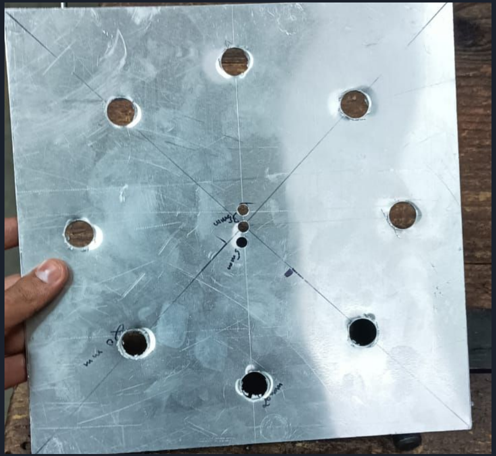
```
Mach number :  1.2698412698412698
```

## Miscellaneous

## Experiment to mimic multiple jet interference



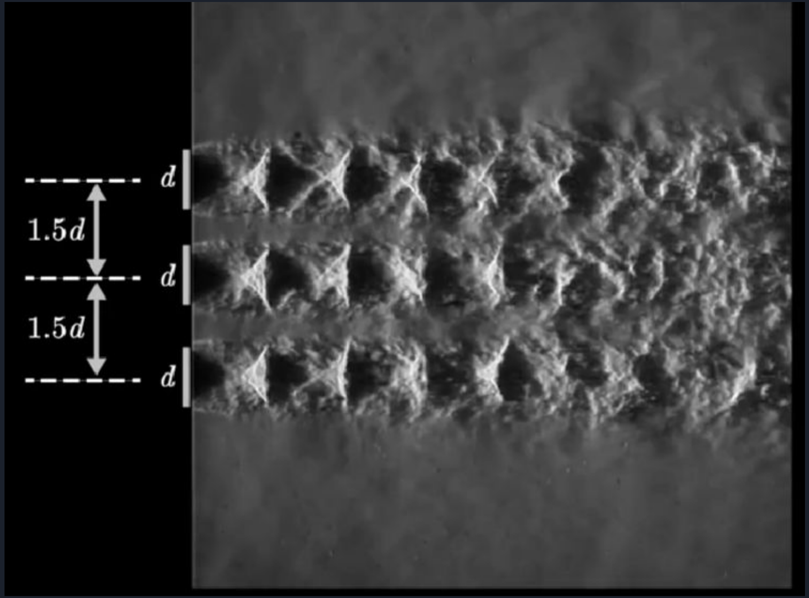
Square Cross Section

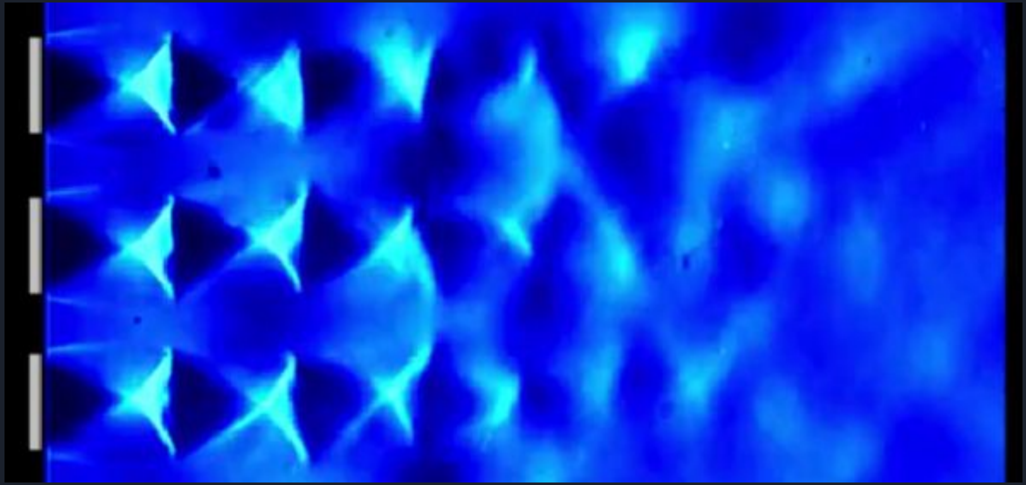


Circular Cross Section

These plates were made to mimic flow of multiple jets placed close to each other but due to material constraint we did not used these plates to perform experiment. The material used (Aluminium) was deemed unsafe for use at high velocity flows.

## Our goal to achieve from the plates manufactured





## Acknowledgements

Supervisor : Prof. Arun Kumar R

And, to postgraduate students who coordinated the supersonic lab for the Schlieren experiments.

Thank you.