Problem 4: Time Prediction of Circular Cylinder wake in Reduced Dimensional Space

Part 1: Train till $t = 0.5s$ and predict results till $t = 1s$

Python file: “forecast_till_t_eq_1.ipynb”

POD and Reconstruction using 3 modes

We first provide our input parameters and check number of samples

```
dt = 1/3000 #dt
t_Final= 0.5 # given in question
numSamples = int(t_Final/dt) # training samples
print(f'Total number of training samples: {numSamples}')
```

We have 1500 timesteps corresponding to t= 0 to 0.5 sec with $dt = \frac{1}{3000}$.

Then we load 1 sample timestep to check if we are loading the files correctly.

```
filename = "./data/Res00001.dat"
dummy_u = np.genfromtxt(filename, usecols=0, skip_header=1)
dummy_v = np.genfromtxt(filename, usecols=1, skip_header=1)
# check if files are read correctly
print(dummy_u)
print(dummy_v)
#load x and y grid data
filename = "./data/MESH.dat"
x_grid = np.genfromtxt(filename, usecols=0, skip_header=1)
y_grid = np.genfromtxt(filename, usecols=1, skip_header=1)
X,Y=np.meshgrid(x_grid,y_grid)
grid_size= x_grid.size * 2 # *2 because we need to stack u and
v in a column
print(grid_size)
print(x_grid)
print(y_grid)
```

We can verify from `Res00001.dat` file and the grid files that our data is read correctly. Next we turn this into a function so that we can generate columns of POD matrix given the timesteps. I am going to arrange my velocity fields in the order $[u_1, v_1, u_2, v_2, \ldots, u_N, v_N]^T$ and each column is correspond to a given timestep in training data. The size of this matrix `S` hence would be 4260×1500 corresponding to total grid points and timesteps in the training data. The function `uv_col` takes both `u` and `v` velocity field and returns a column with velocity field arranged as mentioned. `separate_uv` performs the inverse operation, that is for a given column(=timestep), returns both velocity vector.

```
def uv_col(filename):
    """return u and v columns stacked (u1,v1,u2,v2,...) given
a filename"""
    dummy_u = np.genfromtxt(filename, usecols=0,
skip_header=1)
    dummy_v = np.genfromtxt(filename, usecols=1,
skip_header=1)
    uv = [element for pair in zip(dummy_u, dummy_v) for
element in pair]
    return uv

def separate_uv(vel_time_block,column):
    """return u and v seperrated for column given an array
whole columns correspond to timesteps and rows are u,v
stacked"""
    return
vel_time_block[::2,column],vel_time_block[1::2,column]
```

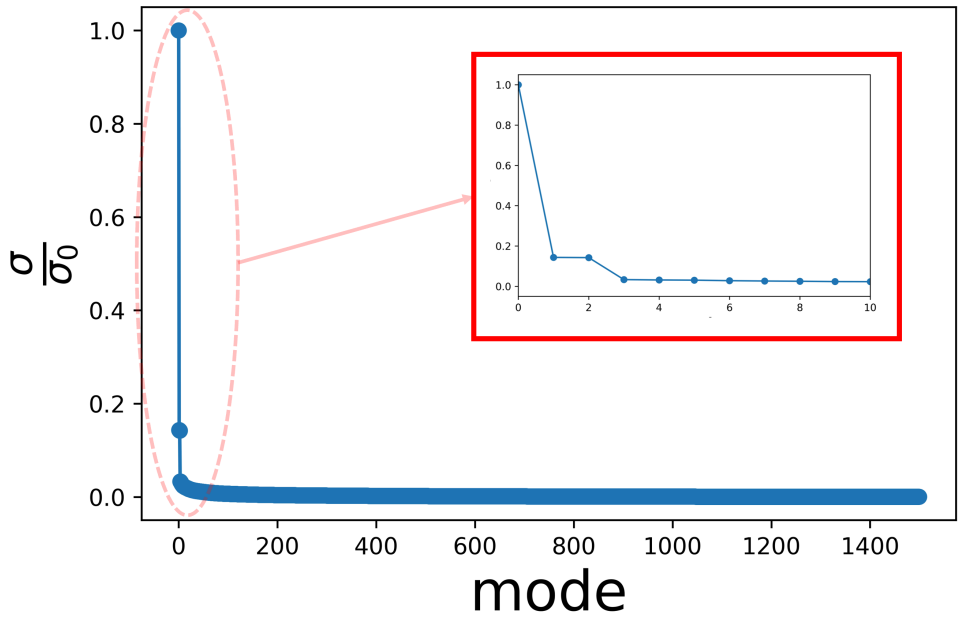
We initialize our matrix `S` with 0s and read the data into it. Then we verity the shape and take a look at the data to make sure everything is correct. The POD is decomposed as $S = U\Sigma V^T$ where U and V are the spatial and temporal eigenvectors respectively and Σ are the eigenvalues.

Generally I have seen POD applied after mean is subtracted, but it was not mentioned in question and I think by 3 modes, it is implied that mode 1 is equivalent of mean mode (if mean was subtracted) and modes 2 and 3 are the complex conjugate modes that capture the vortex shedding. Hence I did not substract the mean mode.

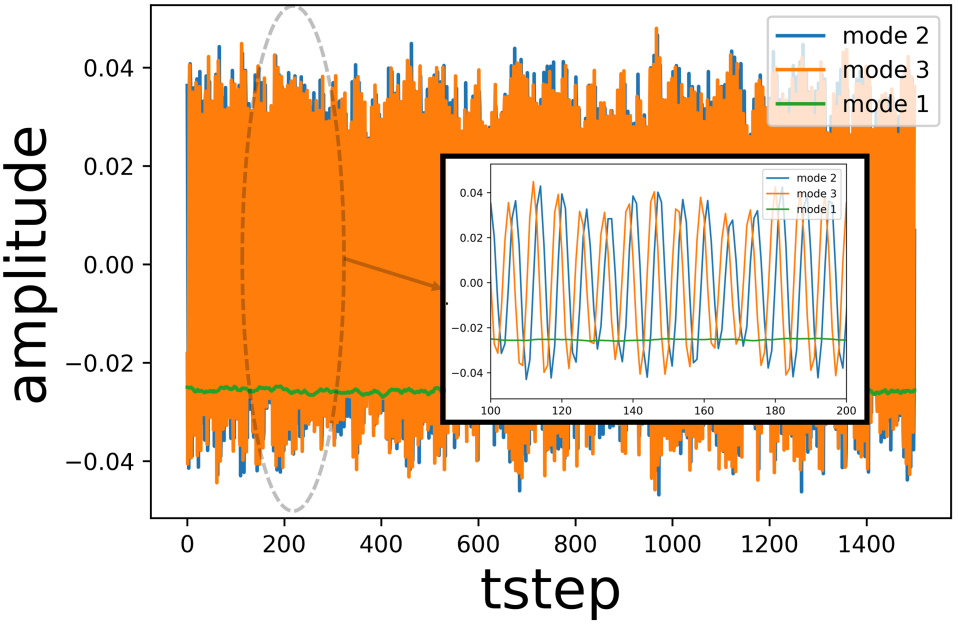
```
S = np.zeros((grid_size, numSamples))

#to construct SVD matrix S
#rows are arranged as u1,v1,u2,v2,...,uN,vN
#columns correspond to t1,t2,t3,...

for i in range(numSamples):
    filename = "./data/Res%05d.dat" % (i+1)
    S[:,i] = uv_col(filename)

```

```
#verify that matrix is correct
print(S)
print(S.shape)

U, Sig, VT = np.linalg.svd(S, full_matrices=False)
```

Now we plot the modes (normalized by the mean mode) and show it below:



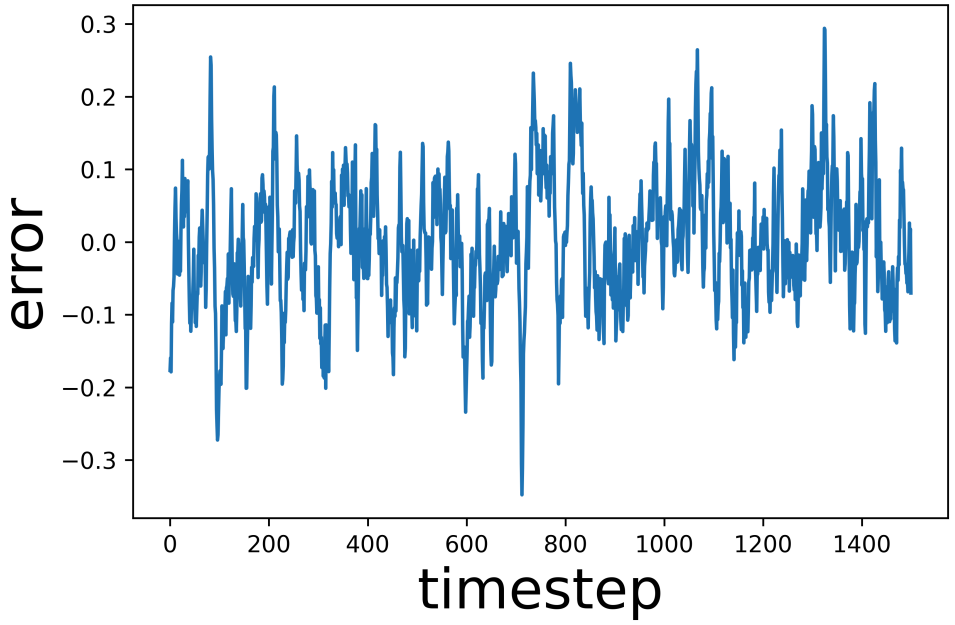
We see that POD has arranged the modes in decreasing order of energy content and since we did not subtract mean, mode 1 alone has the largest value. Modes 2 and 3 are complex conjugate and hence have similar values but their phase should be offset. To check this, let us plot the temporal component of the POD (V^T) for the first 3 modes. In the figure below, focusing on the zoomed version, we see modes 2 and 3 look similar but with a phase gap as we expect from conjugate modes. Mode 1 is almost constant and very similar to mean of the velocity field.



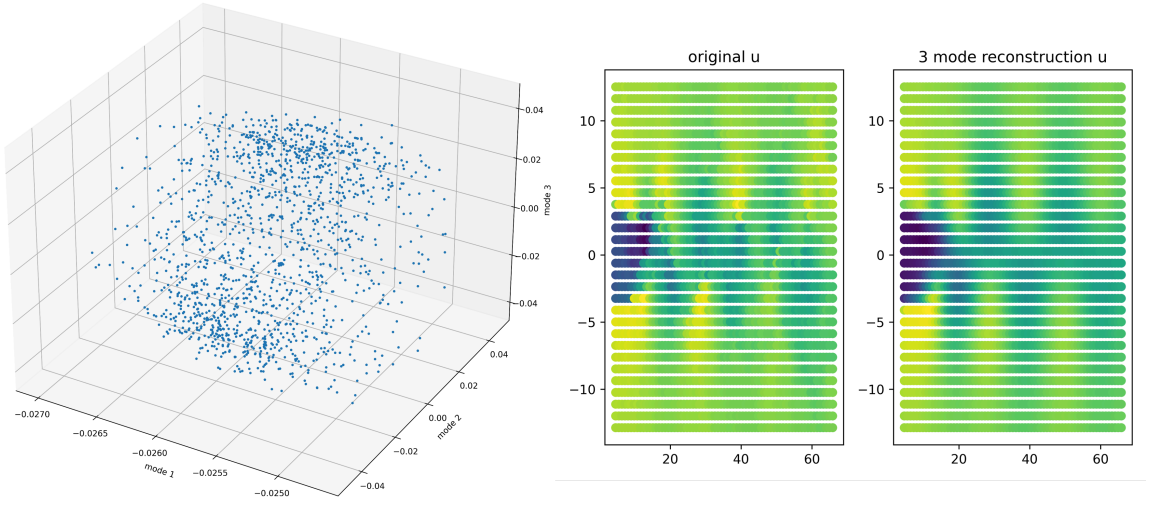
Next, we check the error between actual flow field and the reconstructed flow field with 3 modes. The reconstructed flow field can be written as $S_{Rec} = U\Sigma_R V^T$ where Σ_R has only first 3 elements filled and rest are 0. We could also use $S_{Rec} = U_R\Sigma_R V_R^T$ where U_R has first 3 columns of U and V_R^T is first 3 rows of V^T.

```
Sig_reduced = np.zeros(3)
Sig_reduced[:3] = Sig[:3]
VT_reduced = VT[:3, :] #need this later also for RNN
U_reduced = U[:, :3]
print(Sig_reduced)

reconstructed_S_3_modes =
U_reduced@np.diag(Sig_reduced)@VT_reduced # or U[:,:3] @
np.diag(Sig[:3]) @ VT[:3,:]
print(np.isclose(U[:,:3] @ np.diag(Sig[:3]) @
VT[:3,:],reconstructed_S_3_modes,atol=1e-10).all())
```

For the plot below, I have subtracted the S_{Rec} from the actual matrix S and the resulting columns correspond to timestep and each entry of the given column is the error at different grid points (`x_grid` and `y_grid`). If we average along the columns, then we get overall error at each timesteps, and this is shown below:



To see the overall error accross timestep and all grid, we can use `error_percent = np.linalg.norm(S - reconstructed_S_3_modes)/np.linalg.norm(S)` to get the total error and this comes out to be $\sim$ **15.8**%. This is similar to what we expect from the workshop lectures. We also check reconstruction for a sample timestep (=20 here) and use scatter plot for plotting. To check the manifold using first 3 modes of V^T, we plot each of them on individual axis.



The flow field reconstruction looks reasonable but the flow evolution in the manifold does not look very coherent or in other words does not display any large scale patterns here, possibly because of the smaller dataset.

Training an RNN on Temporal Eigenvectors in reduced dimensional space

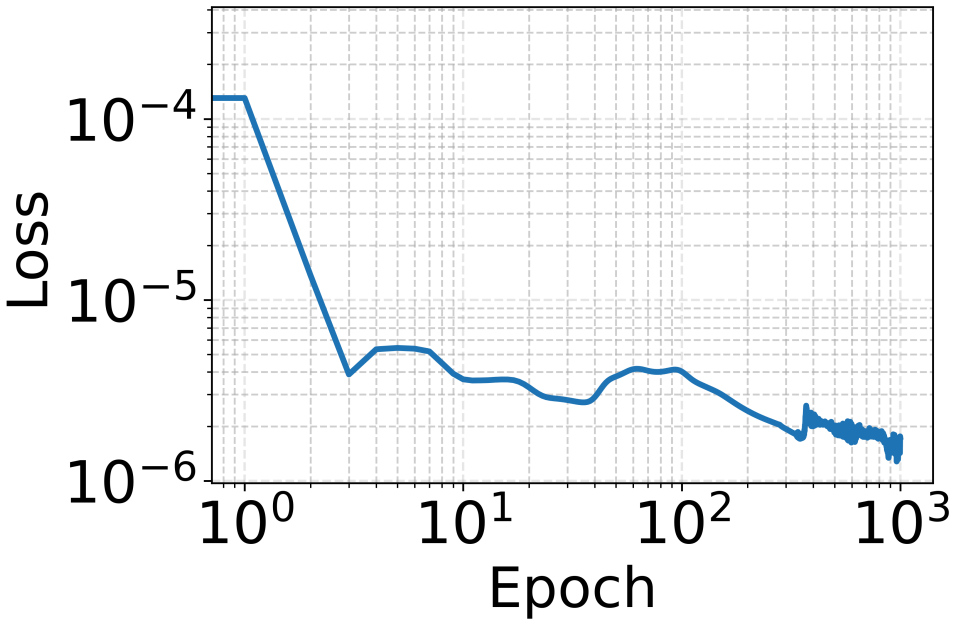
The goal is to now use the reduced dimensional temporal eigenvectors (V_R^T) to train a RNN. This V_R^T contains data for the first 3 modes and naturally contains the temporal component of the flow-field. So if we use this V_R^T data to train the RNN and then predict the next 1500 timesteps (problem asks to predict next 0.5 sec), we can use this new V_{pred}^T and generate future timestep snapshot using $S_{pred} = U_R \Sigma_R V_{pred}^T$. The spatial eigenvectors does not change and we have 'advected' our temporal eigenvectors. `U` and `V` velocity field data for any given future time can be extracted by taking the appropriate column of S_{pred} matrix and using `separate_uv` function described earlier.

```
import torch
from torch.utils.data import DataLoader, TensorDataset

# Prepare data for RNN
def prep_train_target_sequence(data, seq_length):
    sequences = []
    targets = []
    for i in range(len(data) - seq_length):
        sequences.append(data[i:i + seq_length]) # NOTE:
i+seq_length is excluded
        targets.append(data[i + seq_length])
        # targets.append(data[i + seq_length:i + seq_length +
1])
    return np.array(sequences), np.array(targets)

seq_length = 500  #how long the seq length should be?
train, target = prep_train_target_sequence(VT_reduced.T,
seq_length)

# Use torch dataloader for efficient loading
batch_size = 32

train_tensor = torch.tensor(train, dtype=torch.float32)
target_tensor = torch.tensor(target, dtype=torch.float32)
dataset = TensorDataset(train_tensor, target_tensor)
dataloader = DataLoader(dataset, batch_size=batch_size,
shuffle=False) # shuffle=False since we want to keep the
sequence

class RNNModel(torch.nn.Module):
    """
        Some notes below to avoid matrix errors

https://pytorch.org/docs/stable/generated/torch.nn.RNN.html
        inputs: input, h0
```

```
        input:(N,L,H_in) or  batch_size, seq_len, input_size
        h0:(D(=1)*n_layers,N,H_out) or num_layers*num_directions, batch_size, hidden_size
        hidden:(D(=1)*n_layers,N,H_out) or num_layers*num_directions, batch_size, hidden_size
        Outputs: output, h_n
        output:(N,L,H_out*1) or batch_size, seq_len, hidden_size*num_directions
        h_n:(D(=1)*n_layers,N,H_out) or num_layers*num_directions, batch_size, hidden_size
    """
    def __init__(self, input_dim, hidden_dim, output_dim, num_layers=1):
        super(RNNModel, self).__init__()
        self.hidden_dim = hidden_dim
        self.num_layers = num_layers
        self.rnn = torch.nn.RNN(input_dim, hidden_dim, num_layers, nonlinearity='tanh', batch_first=True)
        self.fc = torch.nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        h0 = torch.zeros(self.num_layers, x.size(0), self.hidden_dim).to(x.device)
        out, _ = self.rnn(x, h0)
        out = self.fc(out[:, -1, :])
        return out

input_dim = 3
hidden_dim = 64
output_dim = 3

device = 'cuda' if torch.cuda.is_available() else 'cpu'
model = RNNModel(input_dim, hidden_dim, output_dim).to(device)

loss_function = torch.nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

#>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>> Training loop starts here
num_epochs = 1000
loss_history = []

for epoch in range(num_epochs):
    for input_batch, target_batch in dataloader:

        input_batch, target_batch = input_batch.to(device), target_batch.to(device)

        output = model(input_batch)

        loss = loss_function(output, target_batch)

```

```
        optimizer.zero_grad()

        loss.backward()

        optimizer.step()

    loss_history.append(loss.item())
    if (epoch) % 10 == 0:
        print(f'Epoch [{epoch}/{num_epochs}], Loss:
{loss.item()}')

# test loss -- possibly one could keep aside data of last few
timestep from the training set and use that for test loss


```

Hidden dimension of 64 was selected as we only have 3 dominant modes and flow past a cylinder only has vortex shedding in the wake that we need to capture. I guess a higher value would have been appropriate if we had complex flow features at several scales that we need to predict. The sequence length of 500 (1/3 of data) seems to work fine and RNN does not seem to suffer from vanishing and exploding gradient problems for this sequence length. The learning rate (kept constant) of 0.001 also performs well without diverging. For non linearity, `tanh` was selected. I also tried `ReLu` but `tanh` gave higher prediction accuracy ($\sim$ 10% lower error for the current case). The loss as function of epochs is plotted below:



Here the loss is shown for the train dataset itself. We could also have kept aside data of last few timestep from the training set and use that to create test loss vs epoch. As we see, the loss has almost plateaued near the end with some wiggles. Now we need to test how good our predictions were.

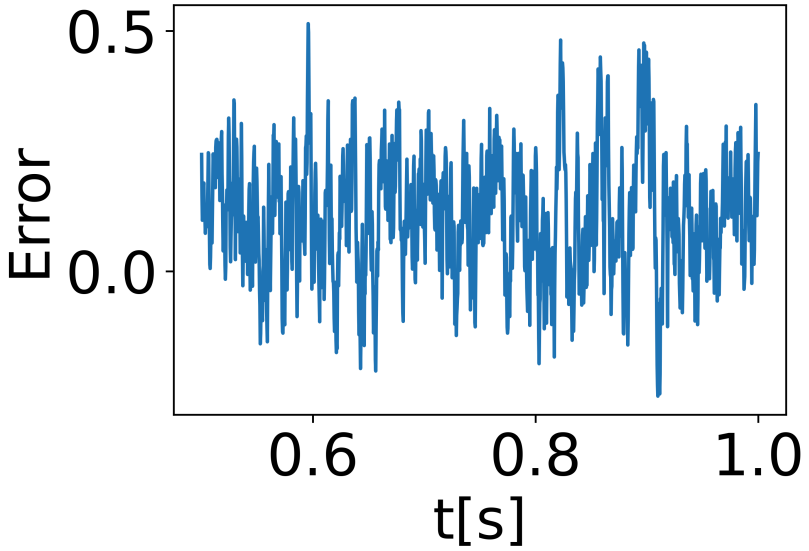
Predictions

Here, the idea is to take all the 1500 timesteps and predict the V_{pred}^T for 1501th timestep. Then ingest the 1501st timestep result to have series from timestep 2 to 1501 and predict the result for 1502 nd timestep and repeat in this manner 1500 times to get results for next 1500 timesteps. The temporal eigenvalues, for each timestep contain value of first 3 modes. So the resulting prediction will also contain temporal results for first 3 modes at the future predicted timesteps. So get the full flow field, we simply take the U_R and Σ_R we had and use $S_{pred} = U_R \Sigma_R V_{pred}^T$ to get S_{pred} and the columns of this matrix contains data at next 1500 timesteps. We can pick any timestep and split `u` and `v` velocity fields for that timestep. To compare how good these predictions are, we simply load the future timestep true results that we have.

```
#  make N new predictions based on the previous prediction
# new prediction are added to the input sequence and used for next set of predictions
def generate_predictions(model, initial_input, num_predictions):
    model.eval() # We dont need backprop calc here
    V_pred = [] # store data to automatically form the temporal eig vectors
    current_input = initial_input.unsqueeze(0)  # Add batch dim

    with torch.no_grad():
        for _ in range(num_predictions):
            current_input = current_input.to(device)
            prediction = model(current_input)
            V_pred.append(prediction.cpu().numpy())
            current_input = torch.cat((current_input[:, 1:, :], prediction.unsqueeze(1)), dim=1)  # ingesting the prediction as input for next prediction
    return np.array(V_pred)

initial_input = train_tensor[-1]  #take  last sequence from dataset as initial input
num_predictions = 1500  #num future predictions
V_pred = generate_predictions(model, initial_input, num_predictions)

V_pred = V_pred.squeeze() # Remove the batch dimension

print("Generated predictions in reduced space:")
```

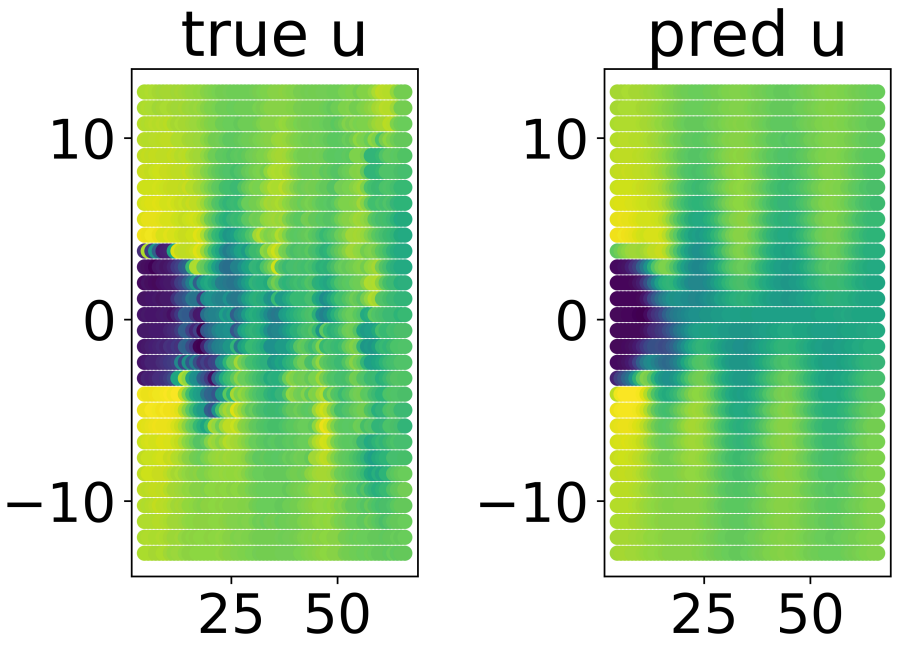
```
# verify the shape of predictions is correct
print(V_pred.shape)
#new VT
VT_pred=V_pred.T
# now we will reconstruct the flow field using this predicted VT

pred_S= U_reduced @np.diag(Sig_reduced)@ VT_pred # our predicted flowfield
true_S = np.zeros((grid_size, numSamples)) #actual data files that we have for future timesteps
for i in range(numSamples):
    filename = "./data/Res%05d.dat" % (i+1501) #since we need results starting from 1501 timestep to compare our predictions
    true_S[:,i] = uv_col(filename)
```

Similar to how we calculated error for original flow field matrix (S) and its reconstruction with 3 modes (S_{Rec}), we now compare the matrix the true unseen results we have: S_{true} and the predicted future results S_{pred}. The mean error between these two matrix corresponding to each timestep is shown below.



To get overall error in flow field prediction, we use `np.linalg.norm(pred_S-true_S)/np.linalg.norm(true_S)` and the error is **24.80** %. This seems pretty good considering without prediction, the error between true and reduced order flow field was $\sim$ 16%. So 'advecting' this reduced temporal component and comparing with true flow-field only resulted in $\sim$ 8% extra error. A sample predicted flow field and true flow field at a sample timestep (=1520) is shown below. The major flow features (or

smooth-out version of velocity field) seems to be present. Calculating the vorticity and comparing them would have been better could be tried later.



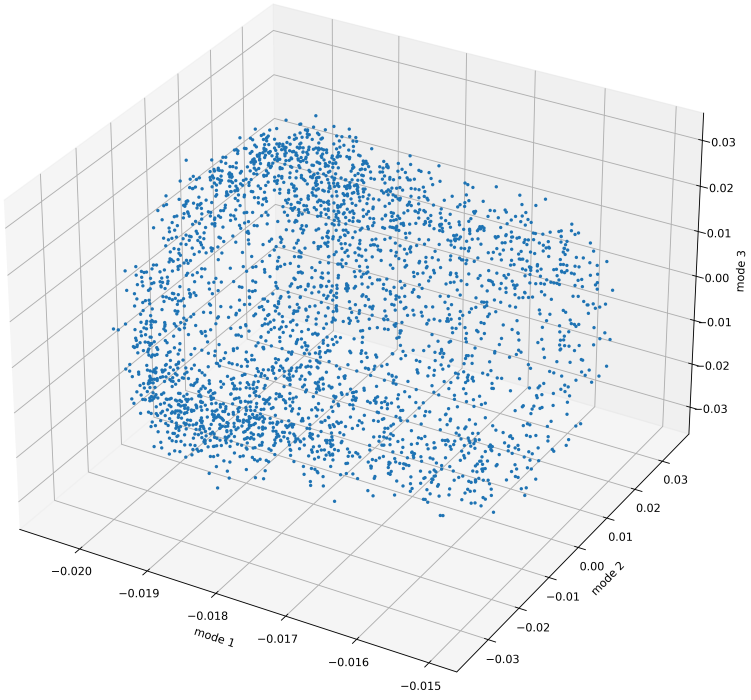
Part 2: Train wake data in $t \in [1s,\ 2s]$ and predict results till $t = 3s$

Python file: “forecast_till_t_eq_3.ipynb”

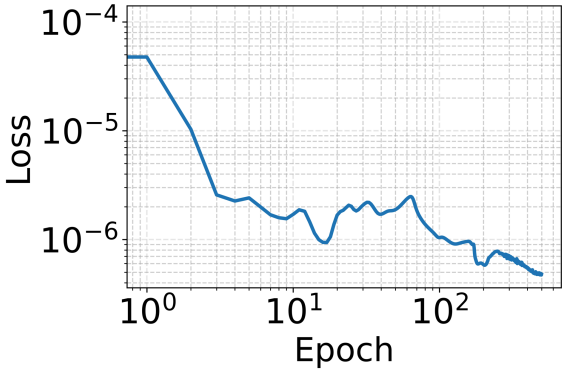
I duplicated the `Prob_4a.ipynb` file and changed the matrix start and end time to account the new time range. Also, now the prediction will be for next 3000 timesteps to predict results till $t = 3s$. I have not repeated all the plots but they can be viewed in jupyter notebook file `Prob_4b.ipynb` file. The eigenvalues are similar and the reconstruction error is $\sim$ 15.94% for the current larger timeseries. With default hyper parameters that was used for previous case, the loss diverges, so hyperparameters were changed to get stabilize the training. I think using lower values of batch size (=16 in this case) helped with convergence. I reduced number of training epochs to 500 since the training was slower now (update: I later noticed high epoch like 1000 also lead to wiggles and higher errors in prediction later on). The sequence length was changed to 1500 as the timeseries for the current case is also twice the size (BTW I noticed that lower sequence values also led to divergence). The hidden layer dimension was also reduced to 32. With all these modifications, the loss stopped diverging. One possible reason for divergence could be that loss became very small

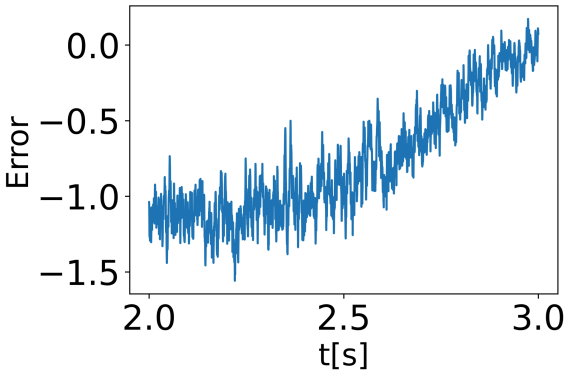
with less parameters (which would have been easier to fit for the NN), however not sure if this is actually the case. I know RNN could be harder to train as with larger network size, the loss becomes small and that leads to vanishing gradient problem. Changing torch data type to 64 bit using `torch.set_default_dtype(torch.float64)` made training extremely slow and I had to cancel the job in between. Adding Batch Norm layers for RNN may help to stabilize the gradients or maybe choosing an LSTM network could also have better performance and could be an exercise for another day.

The manifold in lower dimensional space now looks more coherent than before, possibly due to larger training data. The shape now looks more cylindrical with diameter increasing from one end to the other.

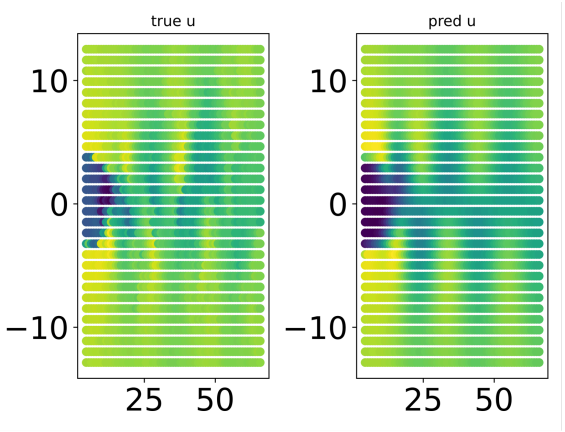


The training loss and the mean error at new timesteps (all steps are same as previous section) is shown below. The overall error in flow field prediction is 30.69% which is slightly higher.

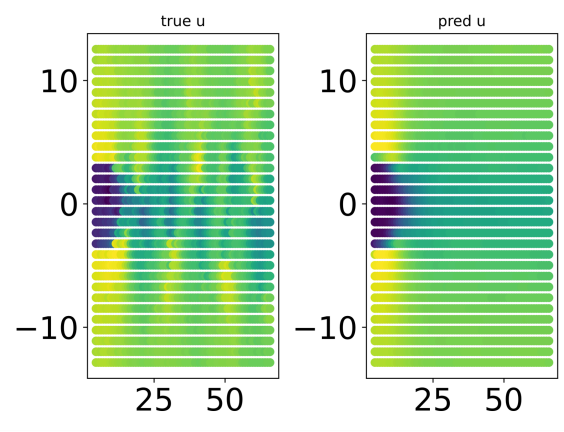




The training loss has reduced to acceptable value (similar to the case we have in previous section) but the error now keeps increasing as we make predictions in longer time range beyond the training zone. A sample contour at 2 sample future timestep (Tstep= 6020 and 8500 in this case) showing true and predicted flow field and as we expect from error plot, the flow field prediction gets worse over time. The wake vortex shedding for time step 8500 case seems to be smudged.



Timestep=6020



Timestep=8500