von Karman Institute for Fluid Dynamics
Chaussée de Waterloo, 72
B - 1640 Rhode Saint Genèse - Belgium

Stagiaire Report

POD AND DMD DECOMPOSITION OF NUMERICAL AND EXPERIMENTAL DATA

Author: K. Zdybal

Supervisor: M. A. Mendez

September 2016

Acknowledgements

I am afraid I will never be able to thank enough for these wonderful two months of my life. But I at least want to give it a try.

I would like to thank my supervisor Miguel for giving me a chance to work on this amazing topic and letting me discover the power of linear algebra.

I would like to thank Claire for her kindness and for all the chances to work in the garden, both of which have made my stay in Saint-Genèse so pleasant.

Abstract

Matrix decomposition is a useful tool for approximating large data matrices which might be obtained from experimental or numerical results. The decomposition methods give a powerful insight into the underlying physical phenomena hidden in the collected data. The present work discusses two decomposition methods: *Proper Orthogonal Decomposition* (POD) and *Dynamic Mode Decomposition* (DMD) (Chapter 1).

Two examples of data decomposition applications are presented in this work: approximating the pulsating velocity profile of the *Poiseuille flow* (Chapter 2) and approximating the flow behind a cylinder (Chapter 3). The results are obtained using *Matlab* software and various codes for performing POD and DMD and for post-processing are shown in the appendices.

The development of the *Graphical User Interface* (GUI) program in *Matlab* for decomposing data with POD and DMD is presented in Chapter 4.

The concepts of data decomposition lie in linear algebra and linear dynamical systems. Additional exercises in the form of ideas and problem-solving are presented in Chapter 5.

Keywords: Linear Algebra, Linear Dynamical Systems, Proper Orthogonal Decomposition, Dynamic Mode Decomposition, Matlab

Contents

Chapter 1

Introduction to Data Decomposition

1.1 Setting the Stage

The present work revolves around large data matrices and the ways to deal with them. They can come from experimental results, such as PIV[1] or wind tunnel tests, or from numerical results such as CFD[2], and hence they are matrices that contain information about the evolution of a certain physical quantity in space and time. Their structure is therefore the following: their rows are linked to a particular coordinate in space and their columns to a particular moment in time. The space coordinate can represent any dimension but typically it will be 1D, 2D or 3D space coordinate. Each entry inside a data matrix $\mathbf{D}$, say $\mathbf{D}_{ij}$, gives a numerical value to a certain quantity at position p_i and at time t_j. This value could for instance be velocity, pressure or any other physical quantity of choice. A single, full column extracted from a data matrix is called a *snapshot*. It represents the full field of a physical quantity at a single instance of time - it is indeed like a picture. Using Matlab notation we can refer to it as $\mathbf{D}(:, m)$ for any time t_m. The graphical representation of a general data matrix is captured in the Figure 1.1.

An important property that describes a data matrix is its *rank*. The rank of a matrix specifies how many linearly independent columns or rows a matrix has. A matrix of size $n_p \times n_t$ is of *full-rank* when its rank is defined as $\min(n_p, n_t)$. When a matrix has a smaller rank than $\min(n_p, n_t)$ some of its rows (or columns) can be expressed in terms of a linear combination of its other rows (or columns).

Usually, and for the data matrices shown later in this work, we assume that the number of spatial coordinates is much larger than the number of moments in time: $n_p > n_t$, hence if a matrix is of full-rank it will have the rank equal to n_t.

An important concept now emerges: imagine we could approximate a data matrix of full-rank well enough with another matrix of a lower rank. Suppose we wanted to share that data matrix with someone else, and instead of sending a matrix of full-rank with all its entries inside, we could send only the linearly independent rows or columns and a rule to figure out the remaining, linearly dependent ones. This decreases the amount of data that we have to store or share, but of course comes at a certain error of approximation. The goal then is to minimize the error while minimizing the rank of a data matrix at the same time.

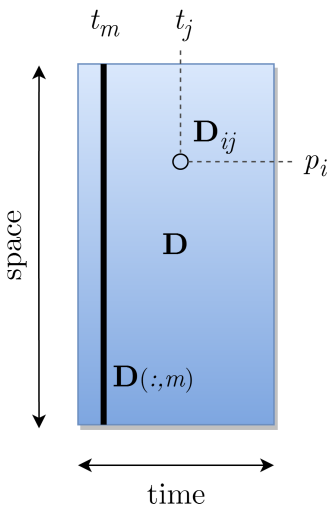


Figure 1.1: Structure of a data matrix $\mathbf{D}$.

[1] Particle Image Velocimetry
[2] Computational Fluid Dynamics

1.2 Data Matrix Decomposition

Two methods of a data matrix decomposition and approximation are discussed in this work.

By matrix decomposition we mean writing the original matrix $\mathbf{D}$ as a product of three matrices:

$$\mathbf{D} = \mathbf{ABC} \tag{1.1}$$

Such product is not unique, meaning that we can find several sets of matrices $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C}$ that will satisfy the equation (1.1). In order to create a unique decomposition, we have to impose some additional constraints on component matrices $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C}$.

The nature of these constraints is what creates a new matrix decomposition method. Two such methods, described below, are the main subject of this work.

In the methods to follow, matrices $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C}$ have a special physical meaning. A product of corresponding columns of the three component matrices is called a *mode*. Matrix $\mathbf{A}$ is a matrix of spatial structures, which gives a "shape" to the mode. Matrix $\mathbf{B}$ is a matrix of amplitudes, which specifies the importance of every mode in the solution. Matrix $\mathbf{C}$ is a matrix of temporal structures and gives dynamics to the mode - it specifies how fast spatial structures evolve from one to another. It should therefore be noticed, that in addition to reducing the rank of a data matrix and approximating it, decomposition gives a better insight into the underlying physical phenomena which are hidden in the data obtained.

1.3 Proper Orthogonal Decomposition (POD)

The concept of the *Proper Orthogonal Decomposition* (POD) on a discrete data set (therefore a matrix) coincides with the *Singular Value Decomposition* (SVD)[3]. More information is given in [2] and [3].

We write therefore:

$$\mathbf{D} = \mathbf{U\Sigma V}^T \tag{1.2}$$

The constraints of SVD are, that the matrices $\mathbf{U}$ and $\mathbf{V}$ are both orthogonal and orthonormal.

The orthogonality of a matrix means that the *inner product*[4] of any two different columns of this matrix is zero and the orthonormality means that the inner product of any of the column with itself is 1. Putting it in other words, the products $\mathbf{VV}^T$ and $\mathbf{UU}^T$ both give an identity matrix. The orthogonality condition also means that the matrices $\mathbf{U}$ and $\mathbf{V}$ are of full-rank and none of its columns can be obtained by a linear combination of their other columns. The SVD also requires that the matrix $\mathbf{\Sigma}$ is diagonal with elements on the diagonal denoted as σ_i.

Once the SVD decomposition is made, we approximate the original matrix of rank d by a matrix $\widetilde{\mathbf{D}}_r$ of a lower rank r, which we write as a sum of "simple" matrices $\mathbf{A}_i$ multiplied by the corresponding amplitude σ_i, extracted from the diagonal of matrix $\mathbf{\Sigma}$.

$$\mathbf{D} \approx \widetilde{\mathbf{D}}_r = \mathbf{A}_1\sigma_1 + \mathbf{A}_2\sigma_2 + \cdots + \mathbf{A}_r\sigma_r \tag{1.3}$$

Each matrix $\mathbf{A}_i$ has rank equal to 1 and the same size as the original matrix. In order to obtain these matrices we multiply the corresponding columns of $\mathbf{U}$ and $\mathbf{V}$ with an *outer product*. Using the Matlab notation, the computation of $\mathbf{A}_i$ is the following: $\mathbf{A}_i = \mathbf{U}(:,i) \cdot \mathbf{V}(:,i)^T$. We are therefore guaranteed that the matrix $\mathbf{A}_i$

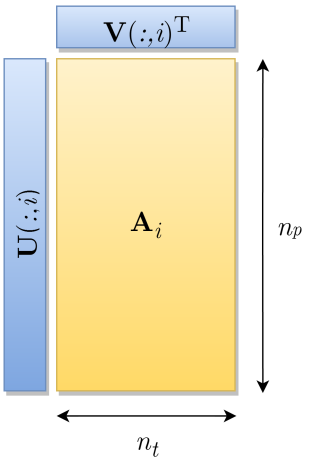


Figure 1.2: Computation of matrix $\mathbf{A}_i$.

[3] Check section 5.3 for additional information on SVD.

[4] Check section 5.1 for more information on the inner products.

is of rank 1: each column of $\mathbf{A}_i$ is the i^{th} column of $\mathbf{U}$ multiplied by a corresponding element inside the vector $\mathbf{V}(:,i)^T$ (and so is just a linear combination of the i^{th} column of $\mathbf{U}$!). We are also guaranteed that any matrices $\mathbf{A}_i$ and $\mathbf{A}_j$ for $i \neq j$ will be orthogonal to each other, since any two columns $\mathbf{U}(:,i)$ and $\mathbf{U}(:,j)$ are, from the definition of SVD, linearly independent. In the Matlab notation we write the equation 1.3 as

$\widetilde{\mathbf{D}}_r \approx \mathbf{U}(:,1:r)\mathbf{\Sigma}(1:r,1:r)\mathbf{V}(:,1:r)^T$.

Hence, by keeping r terms of the sum in (1.3), where $r < d$, we create a matrix of lower rank r that is an approximation to the original matrix $\mathbf{D}$. The error we make by truncating the sum after r elements, computed as the L^2 norm is:

$$||\mathbf{D} - \widetilde{\mathbf{D}}_r||_2 = \sigma_{r+1} \tag{1.4}$$

with $\widetilde{\mathbf{D}}_r = \sum_{i=1}^{r} \mathbf{A}_i \sigma_i$. This is known as the Eckart-Young theorem. The tilde denotes that it is the approximation to matrix $\mathbf{D}$ and the r in the subscript specifies the number of elements in the sum that we decide to keep; σ_{r+1} is an amplitude of the first term left out.

1.4 Dynamic Mode Decomposition (DMD)

The background of the Dynamic Mode Decomposition lies in the linear dynamical systems. On a practical level we might think of the DMD method as fitting a linear system into the data matrix. A more general framework, however, is given in [4], [5], [6] and [7]. This implies that any snapshot taken from the data matrix $\mathbf{D}$ at time $i+1$ is a linear combination of the previous snapshot at time i. In a vector form we write that:

$$\mathbf{x}_{i+1} = \mathbf{A}\mathbf{x}_i \tag{1.5}$$

where $\mathbf{x}_{i+1}$ and $\mathbf{x}_i$ are the $(i+1)^{th}$ and the i^{th} column from the matrix $\mathbf{D}$ respectively. The matrix $\mathbf{A}$ is a linear propagator matrix that tells us how the system will evolve from one snapshot to another, for all columns inside matrix $\mathbf{D}$.

In a general case, it is impossible to fit the linear system into the data matrix that we have, as this data might not be linear in the first place.

We extract two data sets from the matrix $\mathbf{D}$. The data set $\mathbf{X}_1$ is created by all the columns of $\mathbf{D}$ except the last one and the data set $\mathbf{X}_2$ is created by all the columns of $\mathbf{D}$ except the first one. In the Matlab notation we can write them as: $\mathbf{X}_1 = \mathbf{D}(:,1:\text{end}-1)$, $\mathbf{X}_2 = \mathbf{D}(:,2:\text{end})$. Therefore in general:

$$\mathbf{X}_2 = \mathbf{A}\mathbf{X}_1 \tag{1.6}$$

where every i^{th} column of $\mathbf{D}$ is linked by the linear propagator matrix to every $(i+1)^{th}$ column of $\mathbf{D}$. The equation (1.6) is equivalent to the equation (1.5), except it is written for all columns inside $\mathbf{D}$.[5]

We perform the SVD and later the POD approximation of matrix $\mathbf{X}_1$:

$$\mathbf{X}_1 = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T \qquad \mathbf{X}_1 \approx \widetilde{\mathbf{X}}_{1,r} \tag{1.7}$$

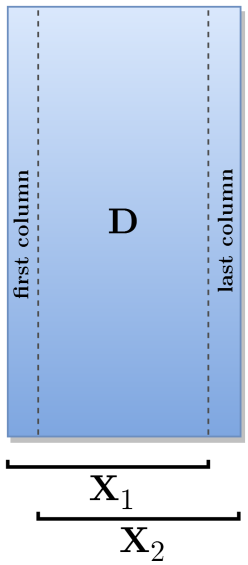


Figure 1.3: Extracted data sets.

Since finding the matrix $\mathbf{A}$ is in general impossible, it is enough to extract certain properties of that matrix, like its eigenvalues. We seek therefore a matrix $\mathbf{S}$ that is similar to matrix $\mathbf{A}$. Similar matrices have the same eigenvalues and share a few other properties. We use the matrix similarity

[5] It is interesting to think when or whether the equation 1.6 could be solved for $\mathbf{A}$ explicitly. This issue is discussed in section 5.4.

condition, which says that if matrices satisfy the equation: $\mathbf{S} = \mathbf{Q}^{-1}\mathbf{A}\mathbf{Q}$ for any matrix $\mathbf{Q}$ that has an inverse, then $\mathbf{S}$ and $\mathbf{A}$ are similar[6].

In order to find the matrix $\mathbf{S}$, we perform algebraic operations on the equation (1.6), substituting the POD approximation from the equation (1.7):

$$\mathbf{X}_2 = \mathbf{A}\mathbf{U}\mathbf{\Sigma}\mathbf{V}^T \quad / \ \mathbf{U}^T\times \tag{1.8a}$$

$$\mathbf{U}^T\mathbf{X}_2 = \mathbf{U}^T\mathbf{A}\mathbf{U}\mathbf{\Sigma}\mathbf{V}^T \quad / \times \mathbf{V} \tag{1.8b}$$

$$\mathbf{U}^T\mathbf{X}_2\mathbf{V} = \mathbf{U}^T\mathbf{A}\mathbf{U}\mathbf{\Sigma}\mathbf{V}^T\mathbf{V} \quad / \times \mathbf{\Sigma}^{-1} \tag{1.8c}$$

$$\mathbf{U}^T\mathbf{X}_2\mathbf{V}\mathbf{\Sigma}^{-1} = \mathbf{U}^T\mathbf{A}\mathbf{U} \tag{1.8d}$$

NOTE 1: in the equation (1.8c) the product $\mathbf{V}^T\mathbf{V}$ becomes a unity matrix, since $\mathbf{V}$ is an orthogonal and orthonormal matrix.

NOTE 2: matrix $\mathbf{\Sigma}$ must be an invertible matrix, which also means that it has to be a square matrix (which is accomplished by performing the POD approximation of matrix $\mathbf{X}_1$).

We now call the product $\mathbf{U}^T\mathbf{X}_2\mathbf{V}\mathbf{\Sigma}^{-1} = \mathbf{S}$ and observe, that in the equation (1.8d) we arrive at the condition of similarity of two matrices:

$$\mathbf{S} = \mathbf{U}^T\mathbf{A}\mathbf{U} \tag{1.9}$$

where the orthogonal matrix $\mathbf{U}$ satisfies the condition $\mathbf{U}^T = \mathbf{U}^{-1}$ and so plays a role of a matrix $\mathbf{Q}$. However, it should be noted here that the matrix $\mathbf{U}$ might not be a square matrix. The matrix $\mathbf{S}$ is size $r \times r$.

We perform the eigenvalue decomposition of $\mathbf{S}$:

$$\mathbf{S} = \mathbf{\Phi}\mathbf{M}\mathbf{\Phi}^{-1} \tag{1.10}$$

where $\mathbf{\Phi}$ is a matrix of eigenvectors of $\mathbf{S}$ and $\mathbf{M}$ is a matrix of eigenvalues of $\mathbf{S}$. In general, $\mathbf{\Phi}$ and $\mathbf{M}$ are complex matrices. Since $\mathbf{S}$ is the size $r \times r$, and typically smaller size than the matrix $\mathbf{A}$, we can only approximate r eigenvalues of $\mathbf{A}$. Notice also, that similar matrices don't share eigenvectors[7].

We extract the elements from the diagonal of matrix $\mathbf{M}$ into a vector $\boldsymbol{\mu}$. Since these elements are in general complex, they can be written as $\mu_i = e^{\omega_i}$, where ω_i is a complex frequency in terms of pulsation.

We find a vector of frequencies $\boldsymbol{\omega}$ by computing the natural logarithm of $\boldsymbol{\mu}$:

$$\boldsymbol{\omega} = \ln(\boldsymbol{\mu}) \tag{1.11}$$

Next, we extract the DMD spatial modes:

$$\boldsymbol{\phi} = \mathbf{U}\mathbf{\Phi} \tag{1.12}$$

The above equation has a meaning of projecting vectors onto the basis made from orthogonal columns of $\mathbf{U}$. The coefficients of the projection are extracted from the eigenvectors matrix $\mathbf{\Phi}$.

We compute the initial DMD amplitudes $\mathbf{b}$ by solving the equation $\mathbf{x}_1 = \boldsymbol{\phi}\mathbf{b}$ with the least-squares approach:

$$\mathbf{b} = (\boldsymbol{\phi}^T\boldsymbol{\phi})^{-1}\boldsymbol{\phi}^T\mathbf{x}_1 \tag{1.13}$$

where $\mathbf{x}_1$ is the full first column of the matrix $\mathbf{X}_1$.

We compute the DMD temporal modes in a form of a matrix $\mathbf{T}_{\text{modes}}$, which will have the size of $r \times n_t$. Each i^{th} column of this matrix is computed using the following relation:

$$\mathbf{t}_i = \mathbf{b}e^{\boldsymbol{\omega}k} = \mathbf{b}e^{\boldsymbol{\omega}t_i/\Delta t} \tag{1.14}$$

[6]More on matrix similarity can be found in section 5.5.

[7]There exists, however, a relationship, see section 5.5 for more details.

with the number $k = \frac{t_i}{\Delta t}$, where t_i is a particular moment in time and Δt is a time step at which snapshots from the data matrix $\mathbf{D}$ have been taken. Notice that the DMD method hence requires that the Δt is a constant sampling interval for the whole matrix $\mathbf{D}$. If the matrix $\mathbf{D}$ represents experimental data, they must have been sampled at equal intervals of time. k is an integer and comes directly from the linear dynamical system character of the DMD method[8].

Notice, that each j^{th} row of matrix $\mathbf{T}_{\text{modes}}$ corresponds to only one j^{th} frequency and only one j^{th} amplitude.

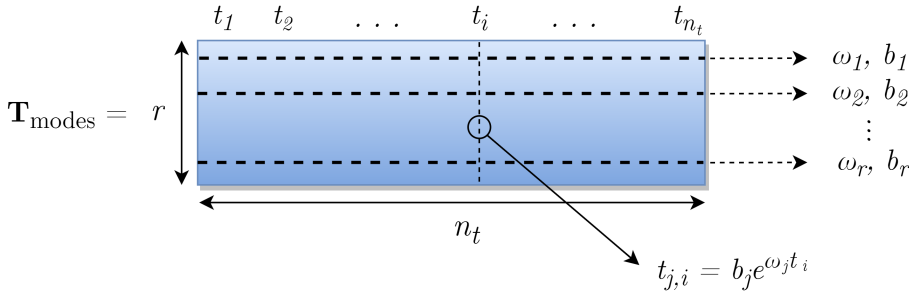


Figure 1.4: Structure of the matrix $\mathbf{T}_{\text{modes}}$.

The DMD approximation to the original data matrix $\mathbf{D}$ is finally computed as a product of spatial and temporal modes:

$$\widetilde{\mathbf{D}}_r = \boldsymbol{\phi} \cdot \mathbf{T}_{\text{modes}} \tag{1.15}$$

Notice that the resemblance with the decomposition equation (1.1) is not evident, since we arrived at the product of only two matrices, but the third matrix of amplitudes $\mathbf{b}$ is hidden in the matrix $\mathbf{T}_{\text{modes}}$. To write the equation (1.15) in the form (1.1), one should write $\mathbf{T}_{\text{modes}}$ as a product of a diagonal matrix and a Vandermonde matrix [4].

1.5 Criteria for the Choice of the Approximation Rank

The rank r of the approximation in the decomposition methods is perhaps the most important and as well the most mysterious parameter that a person performing POD or DMD has to decide upon.

1.5.1 Rank for POD

In the POD method the choice is rather simple. Since the amplitudes σ_i are ordered, we are sure that increasing r will result in a better approximation. We might then decide, that for some value r_i we are satisfied with the approximation. Two situations are possible: that the amplitudes converge to zero and that the amplitudes converge to some constant value c. More information is given in [2].

We can plot the graph of $\sigma_i(r)$, which is composed of discrete lines as the rank r is an integer.

[8] This is further discussed in section 5.6.

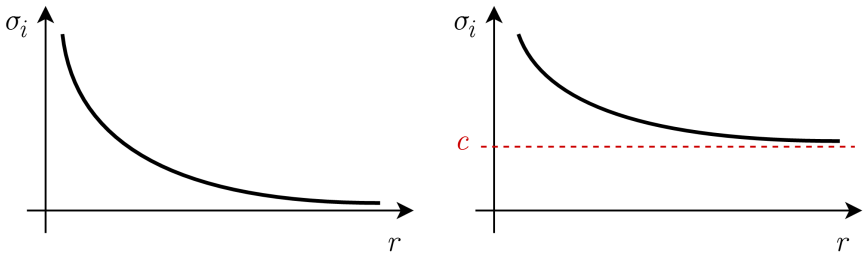


Figure 1.5: Graph of $\sigma_i(r)$.

$\sigma_{\min}$ method

In the case when the amplitudes converge to zero, we can reduce the error of the approximation as much as we want. This error is equal to σ_{i+1} and since the values σ_i converge to zero, the error converges to zero as well. In the $\sigma_{\min}$ method we find such rank r that the error is not greater then a specified value $\sigma_{\min}$. This rank can be found by increasing r from 1 to $r_i + 1$ and each time computing the error of the approximation. Once the error is smaller than the value specified:

$$\sigma_{i+1} \leq \sigma_{\min} \tag{1.16}$$

we can stop the approximation.

Slope method

In the case when the amplitudes converge to a constant value c, it is better to look at the changing slopes of the graph $\sigma_i(r)$. We can never reach the error smaller then c, we can only get as close to c as we like. The slope of the graph is strongly decreasing for the initial values of r and later on it is close to 0 and almost constant. Once the two consecutive slopes of the graph $\sigma_i(r)$ don't change more then some specified value, we can stop the approximation.

1.5.2 Rank for DMD

The criteria for the choice of the rank in the DMD method is still an open question but more complete approach can be found in [8].

In the DMD method the choice of the rank r is made at the level of performing the POD on the extracted data set $\mathbf{X}_1$. One idea is to perform the POD of matrix $\mathbf{X}_1$ with a large number r, proceed with DMD and compute complex eigenvalues of the matrix $\mathbf{S}$. Each eigenvalue is denoted by μ_i and is an element of a vector $\boldsymbol{\mu}$.

Since in general they are complex numbers, we can extract their real λ_r and imaginary λ_i part and write each of them as a sum:

$$\mu_i = \lambda_r + \lambda_i \tag{1.17}$$

Each eigenvalue creates a point on a complex plane (λ_r, λ_i). If we plot these points with respect to a unit circle on a complex plane, we may apply the reasoning from linear dynamical systems. From it we know that the position of the eigenvalue with respect to the unit circle has a special meaning. Points laying exactly on the circle create a dynamically stable solution. Points laying inside the circle represent a system decaying in time and in the DMD approximation have a meaning of noise in the data. They are therefore less relevant to the approximation than points laying on the circle.

The choice on the rank is sometimes not evident. Since the amplitudes b_i are not ordered, we are unsure how many terms we should keep. Some eigenvalues laying inside the circle may have large amplitudes, which means that they eventually decay with time.

Circle based method

Increasing the rank number r we are getting more points on a complex plane. The points laying on the circle or close to the circle boundary are the most important. Usually, the approximation can be stopped once noise start to appear in the data. The criteria then might be, to create a smaller circle of radius $R < 1$ (this radius can be chosen by us). Once the eigenvalues start appearing inside that circle we stop the approximation and retrieve the largest rank r for which all the eigenvalues are outside the small circle.

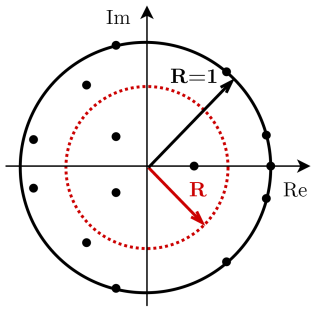


Figure 1.6: Circle based choice of the rank r.

1.6 Comparison of POD and DMD

The POD method imposes the orthogonality and orthonormality restriction on the spatial and temporal structures. In the POD, several frequencies per mode might be present.

The DMD method introduces a single frequency dynamics into the temporal modes. This method is hence especially informative if we know that a particular phenomenon is occuring at a certain frequency.

The rank of the approximation for the purpose of performing the SVD, both in the POD and DMD methods, can be chosen manually at the start of computations and later adjusted (lowered or increased) once an implemented criteria decides, that the approximation has reached the satisfactory level.

Chapter 2

Pulsating Poiseuille Flow

The test case considered in this chapter is the *Poiseuille flow* where the pressure gradient between two sections is changing in time. This definition of pressure results in a velocity profile which deforms in time and differs from the parabolic one obtained for a constant pressure gradient. This flow situation is analyzed in detail in [1], where the analytic solution to the velocity profile function is derived in terms of the *Asymptotic Complex Solution* method and the *Eigenfunction Expansion* method. In the present chapter, three methods are adopted to approximate the analytical solution. The three methods are: the former Eigenfunction Expansion method, the *Proper Orthogonal Decomposition* (POD) and the *Dynamic Mode Decomposition* (DMD). The scope of this chapter is to compare the three methods and analyze the behaviour of their solutions.

2.1 Test Case

The test case for the purpose of this assignemnt is a flow between two parallel plates known as the *Poiseuille flow*. We analyze the situation with the pulsating pressure gradient between two sections. The variation of pressure in time is thus described by:

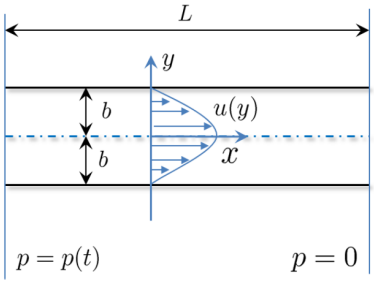


Figure 2.1: Pulsating Poiseuille flow. Figure taken from [1].

$$p(t) = p_M + p_A \cos(\omega t) \tag{2.1}$$

The momentum balance in the Navier-Stokes equation in the stream-wise direction is concerned:

$$\frac{\partial u}{\partial t} = -\frac{1}{\rho}\frac{\partial p}{\partial x} + \nu \frac{\partial^2 u}{\partial y^2} \tag{2.2}$$

This is a partial differential equation which links the pressure gradient with the velocity distribution. In [1], the Navier-Stokes equation from (2.2) is nondimensionalized for simplicty and later on, the Eigenfunction Expansion method and the Asymptotic Complex Solution method are applied to derive the solution. Two important nondimensional constants appear: the Womersley number $\mathcal{W}$ and the dimensionless pressure amplitude $\hat{p}_A$.

In the present chapter we approximate the solution to the equation (2.2) using three methods and compare the results of these approximations.

2.2 Asymptotic Complex Solution

The analytical solution that is used as a base for the approximations to follow is obtained in [1] with the Asymptotic Complex Solution method.

In this method, the solution is a real part of the complex velocity function:

$$\mathbf{U_R}(\hat{y}, \hat{t}) = Re\{\tilde{u}(\hat{y}, \hat{t})\} \tag{2.3}$$

where $\hat{y}$ is a dimensionless space coordinate and $\hat{t}$ is a dimensionless time.

The result is a matrix $\mathbf{U_R}$, which in general might be a full-rank, rectangular matrix. Its rows are linked to the 1D spatial coordinates and its columns are linked to the consecutive time steps. The size of this matrix is therefore $n_y \times n_t$. Depending on how we descretize space and time, we might end up with a matrix of a large size. Typically, we descretize $\hat{y}$ and $\hat{t}$ in such way, that the number of rows n_y is larger than the number of columns n_t.

A vector $\hat{y}$, specifying the discretization of space, contains n_y entries ranging from -1 to 1.

A vector $\hat{t}$, specifying the discretization of time, contains n_t entries ranging from 0 to some t_{end} which in this case is the final time of the simulation that we wish to perform.

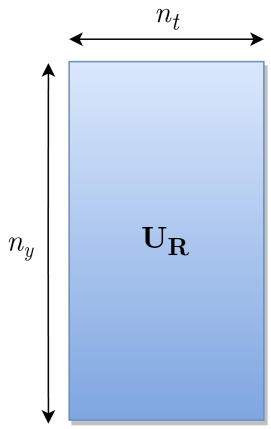


Figure 2.2: Structure of the matrix $\mathbf{U_R}$.

2.3 Eigenfunction Expansion

The Eigenfunction Expansion solution is also obtained in [1]. In the Eigenfunction Expansion method the solution is written in terms of the sum:

$$\mathbf{U_A}(\hat{y}, \hat{t}) = \sum_{i=1}^{n} \phi_i(\hat{y}) a_i \Psi_i(\hat{t}) + \mathbf{U_M} \tag{2.4}$$

where $\phi_i(\hat{y})$ is a spatial structure, a_i is an amplitude and $\Psi_i(\hat{t})$ is a temporal structure. $\mathbf{U_M}$ is the mean flow.

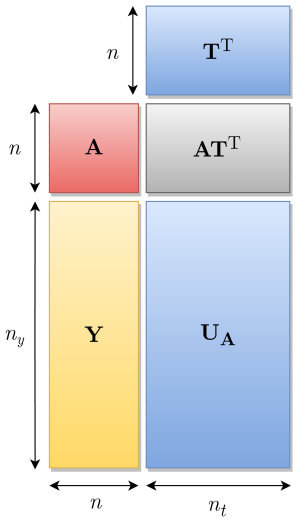


Figure 2.3: Graphical representation of the equation (2.5).

A number n describes the number of modes that we decide to keep in the approximation. When $n = \max(n_y, n_t)$, the solution is exact. The product $\phi_i(\hat{y}) a_i \Psi_i(\hat{t})$, computed for any i is the i^{th} mode of the approximation.

Computing the sum in (2.4) is equivalent to multiplying three matrices:

$$\mathbf{U_A}(\hat{y}, \hat{t}) = \mathbf{YAT}^T \tag{2.5}$$

where $\mathbf{Y}$ is the matrix of spatial structures of size $n_y \times n$, $\mathbf{T}$ is the matrix of temporal structures of size $n_t \times n$. Matrix $\mathbf{A}$ is a diagonal matrix of amplitudes of size $n \times n$.

Each column of the spatial structures matrix $\mathbf{Y}$ is made up from a cosine function of a particular frequency. Spatial structures give a shape to the solution. The entries a_i in the amplitude matrix $\mathbf{A}$ depend on the dimensionless pressure amplitude $\hat{p}_A$ and the Womersley number $\mathcal{W}$. Each amplitude describes how important every mode is in the summation. Each column of the temporal matrix $\mathbf{T}$ (or each row of its transpose) represents the time evolution and can be viewed as giving the dynamics to the system.

2.4 Discrete Proper Orthogonal Decomposition (POD)

In the POD method we start with the data matrix $\mathbf{U_R}$, as defined in 2.2, and approximate it with a lower rank matrix $\mathbf{U_{POD}}$ of the same size.

We perform the SVD on the matrix $\mathbf{U_R}$ and decompose it into a product of three matrices:

$$\mathbf{U_R} = \mathbf{U\Sigma V}^T \tag{2.6}$$

The SVD decomposition is imposing an orthogonality and orthonormality constraint on matrices $\mathbf{U}$ and $\mathbf{V}$. $\mathbf{U}$ is the matrix of spacial structures and $\mathbf{V}$ is the matrix of temporal structures. Unlike in the Eigenfunction Expansion method, the POD may introduce several frequencies per mode.

Matrix $\mathbf{\Sigma}$ is a diagonal matrix. The entries σ_i on its diagonal have the meaning of amplitudes which, similarly to the entries a_i, specify the importance of each POD mode.

Next, the POD approximation is performed by computing the truncated sum:

$$\mathbf{U_R} \approx \mathbf{U_{POD}} = \mathbf{u}_1\mathbf{v}_1^T\sigma_1 + \mathbf{u}_2\mathbf{v}_2^T\sigma_2 + \cdots + \mathbf{u}_r\mathbf{v}_r^T\sigma_r \tag{2.7}$$

where u_i is the i^{th} column of $\mathbf{U}$ and v_i is the i^{th} column of $\mathbf{V}$.

It is important to note that in the SVD the elements σ_i are *ordered* and for any i it always holds that: $\sigma_i > \sigma_{i+1}$. This condition is very important in the POD approximation: on truncating the sum after the r^{th} term, we are sure that every next element of the sum will contribute less then any element from the ones that we have kept.

Since the L^2 norm error of POD is equal to the amplitude of the first term left out, it is important for the accuracy of the approximation that the amplitudes σ_i converge to zero. Otherwise, if they converge to some constant value, the norm will remain almost constant after some time, no matter how many terms we add to the approximation. When the amplitudes converge to zero, the error converges to zero as well. The faster it converges to zero, the more the matrix $\mathbf{U_R}$ is close to be rank deficient.

2.5 Discrete Dynamic Mode Decomposition (DMD)

In the DMD method we start again with the data matrix $\mathbf{U_R}$ from which we extract the data sets $\mathbf{X}_1$ and $\mathbf{X}_2$, as defined in the section 1.4. The rank r is chosen at the level of computing the POD of a matrix $\mathbf{X}_1$.

The analysis that follows is analogous to the one described in section 1.4.

The vector $\boldsymbol{\mu}$ of the eigenvalues of the matrix $\mathbf{S}$ is decomposed into the real and imaginary part:

$$\boldsymbol{\mu} = \mathrm{Re}(\boldsymbol{\mu}) + \mathrm{Im}(\boldsymbol{\mu}) = \lambda_r + \lambda_i \tag{2.8}$$

where $\lambda_r = \mathrm{Re}(\boldsymbol{\mu})$ and $\lambda_i = \mathrm{Im}(\boldsymbol{\mu})$. The vector $\boldsymbol{\mu}$ is length r, and so is the vector of frequencies $\boldsymbol{\omega}$ and the vector of initial amplitudes $\mathbf{b}$.

As a result we obtain the DMD approximation to the original data matrix:

$$\mathbf{U_R} \approx \mathbf{U_{DMD}} = \boldsymbol{\phi} \cdot \mathbf{T}_{\mathrm{modes}} \tag{2.9}$$

An important thing to note is that the amplitudes in the DMD method (unlike in the POD method) are **not** ordered. This means that some $b_{i+1} > b_i$ and therefore often we cannot approximate the matrix well with only first few modes, as some further modes may appear important as well.

2.6 Comparison of the Three Approximations

2.6.1 Initial Parameters

A Matlab code [App.A] is developed to simulate the three approximation methods.

The approximations are performed for the Womersley number $\mathcal{W} = 10$ and for the dimensionless pressure coefficient $\hat{p}_A = 60$ (case C2 from [1]).

The matrix $\mathbf{U_R}$ in the present test case has rank = 3. Hence, only three approximations are considered: for $r = 1$, $r = 2$ and $r = 3$. The POD and DMD methods should give an exact result when $r = 3$. For the purpose of plotting the amplitude decay rate and the eigenvalues circle, the number of modes is later increased to 20.

The time step is $dt = 0.05$. The smaller the time step, the smoother graphs of the temporal structures we obtain.

The space step is $dy = 0.05$. The smaller the space step, the smoother graphs of the spatial structures we obtain.

The results presented below are obtained for the total time of simulation $T = 20$.

2.6.2 Comparison of the Modes

A time evolution of the pulsating velocity profile between two parallel plates is obtained, in a form of a movie, for the analytic solution and three approximation methods. The time of the simulation can be chosen arbitrarily. In Figures 2.4 and 2.5 three first modes of the approximations are drawn with respect to the analytic solution, for two different time moments in the simulation.

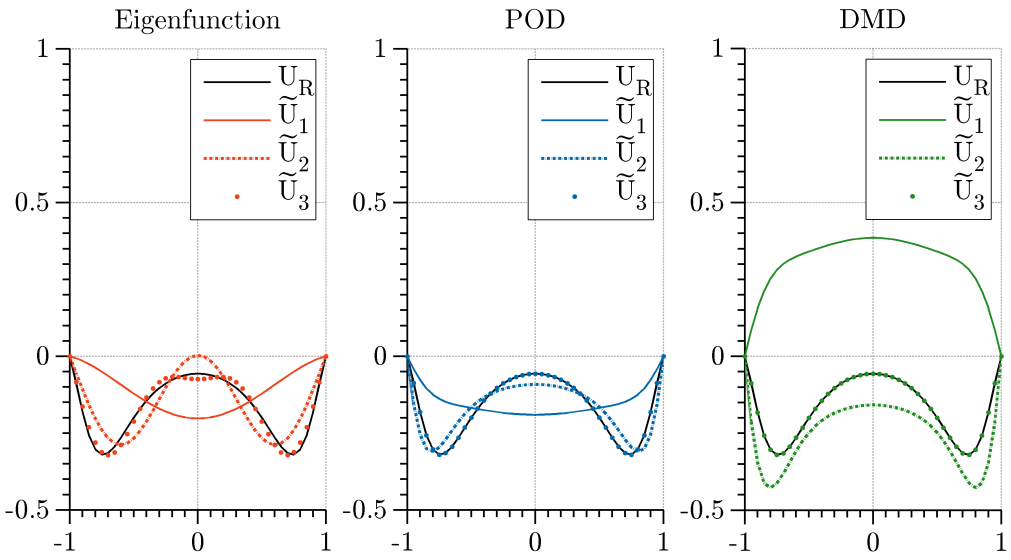


Figure 2.4: Approximation of the asymptotic complex solution with 1, 2 and 3 first modes. Drawn for $t = 2$.

The first mode $\widetilde{\mathbf{U}}_1$ is approximating the mean flow in all three approximation methods.

In the Eigenfunction Expansion and in the POD method, the two first approximations follow in time the original solution. In the DMD method, the first two approximations often find themselves moving in the opposite direction to the original solution.

Approximation with the third mode $\widetilde{\mathbf{U}}_3$ follows exactly the original solution in the POD and in the DMD method. In the Eigenfunction Expansion, the approximation gets better, as we add more modes, but is still not exact for the third mode $\widetilde{\mathbf{U}}_3$.

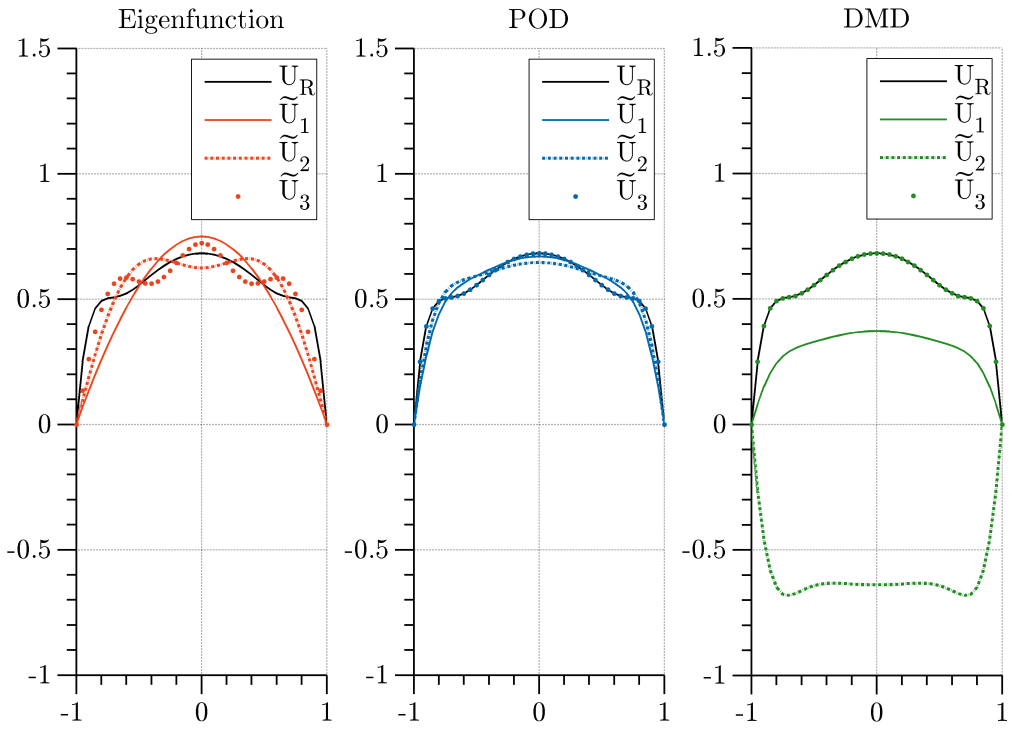


Figure 2.5: Approximation of the asymptotic complex solution with 1, 2 and 3 first modes. Drawn for $t = 3.5$.

2.6.3 Amplitude Decay Rate

Two functions of the amplitude decay rate are obtained by plotting the elements a_i from the diagonal of matrix $\mathbf{A}$ for the eigenfunction expansion and the elements σ_i from the diagonal of matrix $\mathbf{\Sigma}$ for the POD method. In Figure 2.6 they are plotted versus the number of modes taken into account. The amplitudes are normalized so that the largest amplitude has the value 1.

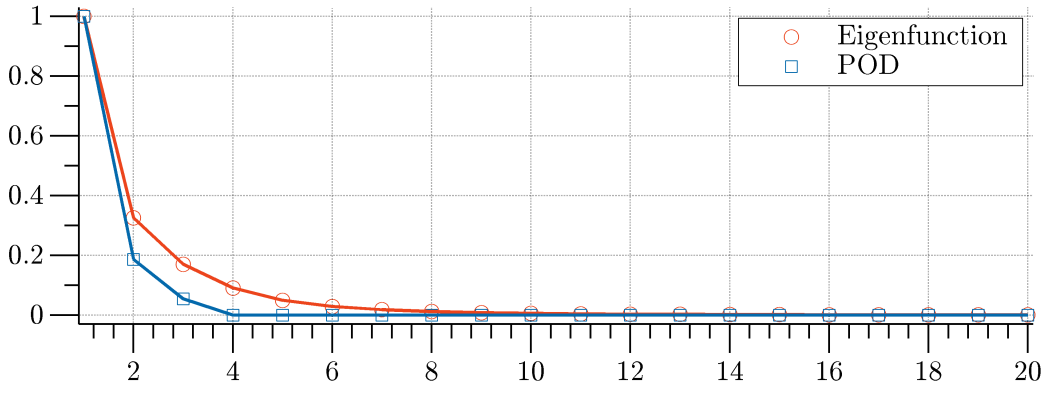


Figure 2.6: The normalized amplitude decay rate.

Some interesting facts can be observed. Firstly, the decay rate is faster for the POD method then the eigenfunction expansion method. The faster the amplitude decay rate, the better we can approximate the solution with a low number of modes.

Secondly, the amplitudes of POD are zero for the number of modes larger than 3. This is due to the rank of matrix $\mathbf{U_R}$ being equal to 3. With 3 modes of POD we are already recovering

the full data set and the L^2 error, equal to the largest term left-out is 0%.

The amplitudes converge to zero for both approximation methods and hence adding more terms in the eigenfunction expansion will keep improving the approximation. With the first three terms of the eigenfunction expansion we read the error to be about 10%.

2.6.4 Eigenvalues Circle

As one of the results of the DMD method we draw the eigenvalues of the matrix **S** on a complex plane versus the corresponding amplitude b_i. Thus, the Figure 2.7 is made up from the points (λ_r, λ_i), positioned at the height equal to the normalized amplitude b_i. 20 first modes of the DMD are drawn. We also include a unit circle to represent the boundary of the linear dynamical system. Position of the eigenvalue with respect to that circle will have a special meaning.

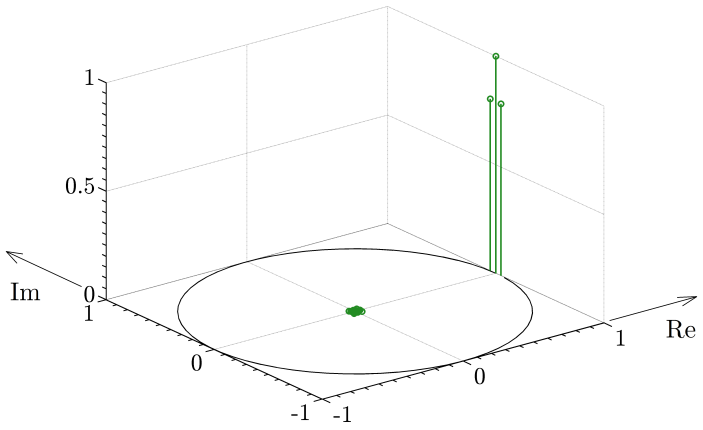


Figure 2.7: The first 20 normalized DMD amplitudes.

The points laying exactly on the circle create a dynamically stable solution. The points laying inside the circle represent the noise in the data: the corresponding eigenvalues are less relevant in the approximation and they eventually die out. Some points however, might be laying inside the circle but close to the circle boundary. They have a meaning of a slowly decaying approximation. Points laying inside the circle, very close to the complex plane origin Any points laying outside the circle represent an exploding dynamical system and would diverge the approximation.

For the Poiseuille flow test case, we have three eigenvalues that lie exactly on the circle and 17 ones very close to the point $(0\,\mathrm{Re}, 0\,\mathrm{Im})$. The first DMD mode is represented by an eigenvalue with only a real part, which approximates the mean flow. The second DMD mode introduces one complex eigenvalue and the third DMD mode introduces its complex conjugate. Notice that every eigenvalue with nonzero imaginary part will have its complex conjugate partner.

2.6.5 The First Three Modes of the POD and DMD Approximations

The first three spatial and temporal structures of POD and DMD approximations are plotted in Figures 2.8-2.13.

The first spatial modes are the same for the POD and the DMD method and they reconstruct the mean flow between the parallel plates.

The POD method introduces several frequencies in the spatial structures. The temporal modes of the POD are shifted in phase.

The first two DMD temporal modes are the same and the third mode is constant.

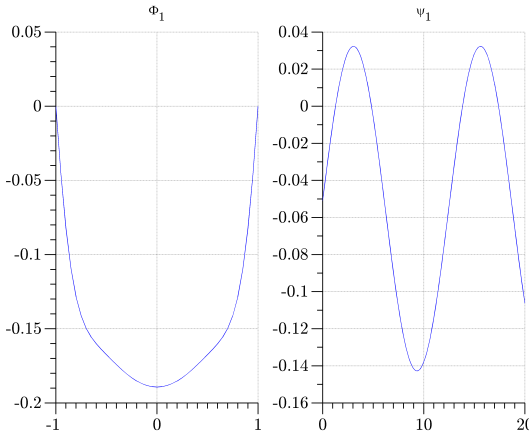


Figure 2.8: POD 1st mode.

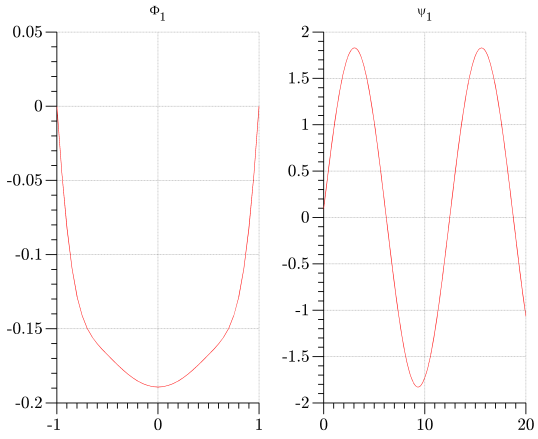


Figure 2.9: DMD 1st mode.

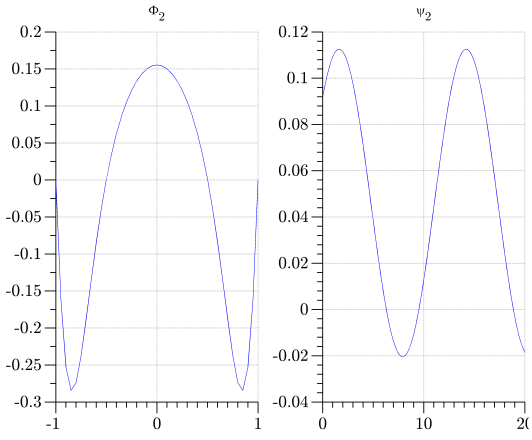


Figure 2.10: POD 2nd mode.

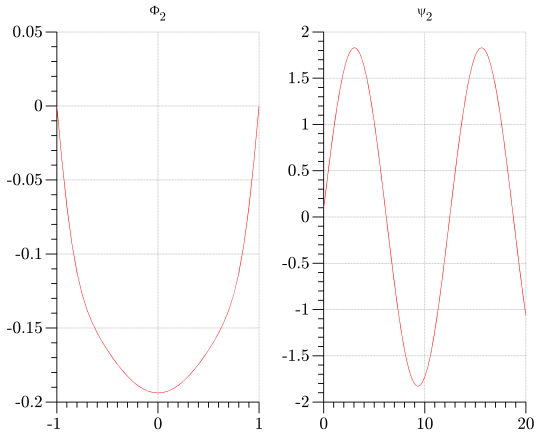


Figure 2.11: DMD 2nd mode.

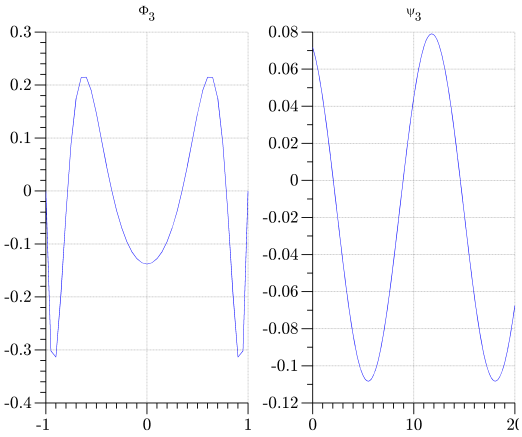


Figure 2.12: POD 3rd mode.

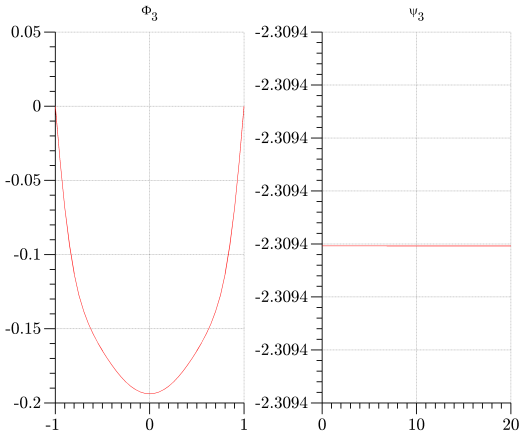


Figure 2.13: DMD 3rd mode.

2.7 Conclusions

Few interesting facts can be observed from the approximations performed.

Approximation methods satisfy the boundary condition, even though this information was not implemented in the procedures.

The mean flow is reconstructed in both the POD and DMD. It appears in the first mode of the approximation. The first spatial structure has the same shape for the POD and DMD method but the temporal structures are different.

In the DMD method, the filtering (the choice of the rank r) does not necessarily have to be performed at the POD level. From the eigenvalue circle we observe that the DMD can filter the noise as well later on. We might therefore include many modes at the POD level, let the DMD method be performed for all of them and then from an implemented criteria correct the approximation for a lower number of r. The purpose of restricting r at the POD level is so that the DMD can start at a smaller set of data. Filtering at the POD level might therefore be useful, as it decreases the sizes of those matrices, whose size is dependent on r. This in turn reduces the use of memory during computations.

Finally, it is worth noticing that in general, columns creating matrices $\mathbf{Y}$ and $\mathbf{T}$ in the Eigenfunction Expansion method, might not be orthogonal to each other nor be orthonormal. In this test case it happened that the matrix $\mathbf{Y}$ was orthogonal, though $\mathbf{T}$ was not. None of these matrices were orthonormal. In the POD method we require the columns of matrices $\mathbf{U}$ and $\mathbf{V}$ to be orthogonal and this is one of the reasons why POD creates a faster approximation.

Chapter 3

Dynamic Mode Decomposition of 2D Data

The flow behind a cylinder was simulated by other student using the OpenFOAM CFD software [10], to produce two large matrices corresponding to two velocity components in the 2D plane. The region of the flow was discretized into a mesh in the x and y axis. In this chapter, the Dynamic Mode Decomposition method and the Proper Orthogonal Decomposition are applied to approximate the simulated flow behind a cylinder, analyze the behaviour of the approximations and compare the two decomposition methods.

3.1 DMD and POD Approximation to the Flow Behind a Cylinder

3.1.1 Initial Data

In this chapter we deal with the 2D flow case, hence the amount of data that we have to process is significantly larger then in the Poiseuille flow case. This time we need two velocity components u and v, corresponding to a point in the xy-plane.

The data obtained from the CFD consists of four matrices:

1. **U** matrix of u-components of velocity `U.mat`
2. **V** matrix of v-components of velocity `V.mat`
3. **X** vector of coordinates on the x-axis `X.mat`
4. **Y** vector of coordinates on the y-axis `Y.mat`

The region of the flow is discretized in the x-axis into **301** points and in the y-axis into **201** points. The total number of spatial points is therefore:

$$n_p = 301 \times 201 = 60501 \tag{3.1}$$

The time of simulation is discretized into **313** timesteps.

Vector **X** contains 301 x-coordinates. Their range is from -0.3287 to -0.0287.

Vector **Y** contains 201 y-coordinates. Their range is from -0.1108 to +0.0892.

Matrix **U** is size 60501×313.

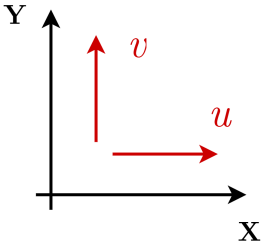


Figure 3.1: Velocity components.

Matrix $\mathbf{V}$ is size 60501×313.

Other parameters of the CFD simulation are presented below:

Free stream velocity	10	$[\frac{m}{s}]$
Cylinder diameter	0.015	$[m]$
Turbulence intensity	5%	$[-]$
Kinematic viscosity	$1.43 \cdot 10^{-5}$	$[\frac{m^2}{s}]$
Strouhal number	0.2	$[-]$
Length of the region in the x-axis	0.3	$[m]$
Length of the region in the y-axis	0.2	$[m]$

The lengths of matrices $\mathbf{U}$ and $\mathbf{V}$ correspond to a 2D space, and hence the rows of these matrices have a special structure. The first 201 rows correspond to each entry inside vector $\mathbf{Y}$ and to the first entry inside vector $\mathbf{X}$. The second 201 rows correspond again to each entry inside vector $\mathbf{Y}$ but now to the second entry inside vector $\mathbf{X}$. In general, it can be viewed as 301 $\mathbf{Y}$ vectors placed one after the other, each of them corresponding to only one x-coordinate. This structure allows to represent 2D data in a single length of the data matrix. It is also graphically represented in the Figure 3.2.

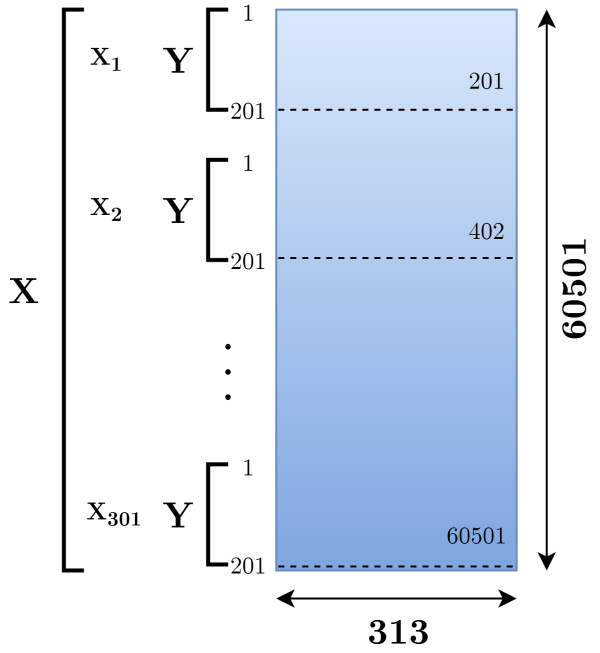


Figure 3.2: Structure of both data matrices $\mathbf{U}$ and $\mathbf{V}$.

The cylinder itself is masked in the data matrices and the velocity components are set to `NaN`, where the cylinder is present.

3.1.2 Dynamic Mode Decomposition

The Dynamic Mode Decomposition is performed on the data matrices **U** and **V** joined together.
The time step chosen for the approximation is 0.01.
The procedure of performing 2D DMD is then the same as described in the section 1.4.
The rank r chosen for the analysis is 4.

DMD Results

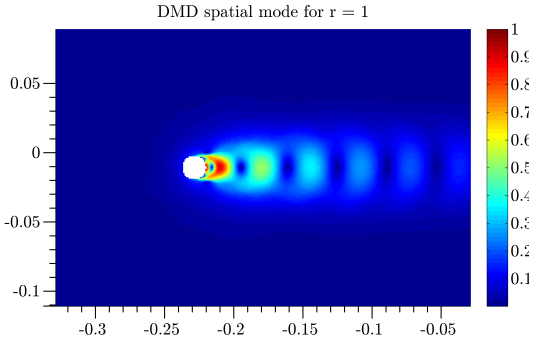


Figure 3.3: DMD spatial mode for r = 1.

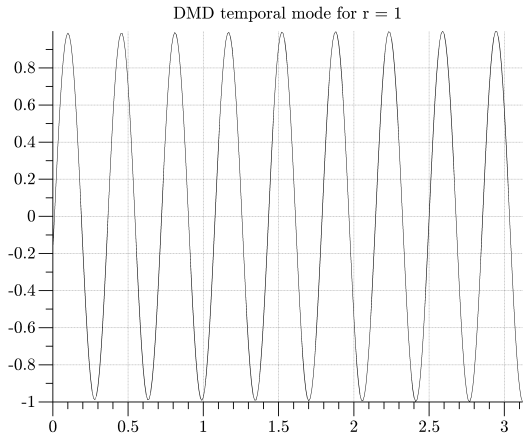


Figure 3.4: DMD temporal mode for r = 1.

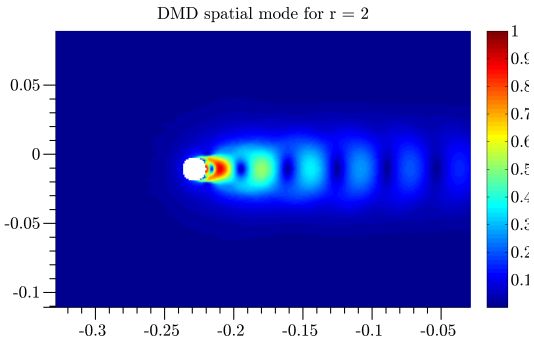


Figure 3.5: DMD spatial mode for r = 2.

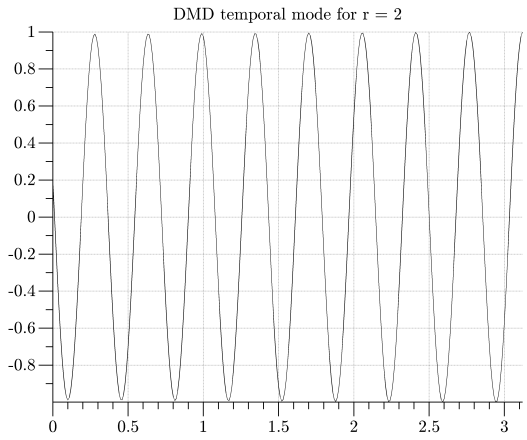


Figure 3.6: DMD temporal mode for r = 2.

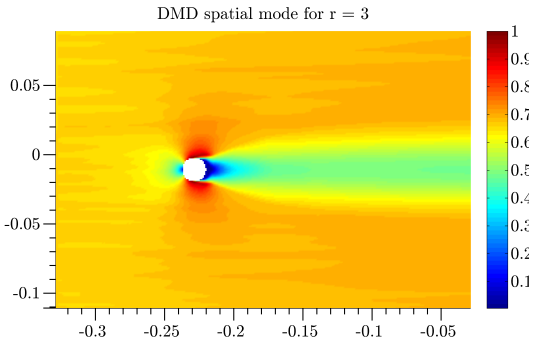


Figure 3.7: DMD spatial mode for r = 3.

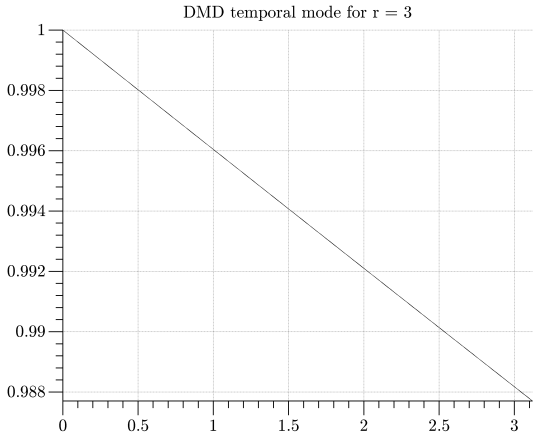


Figure 3.8: DMD temporal mode for r = 3.

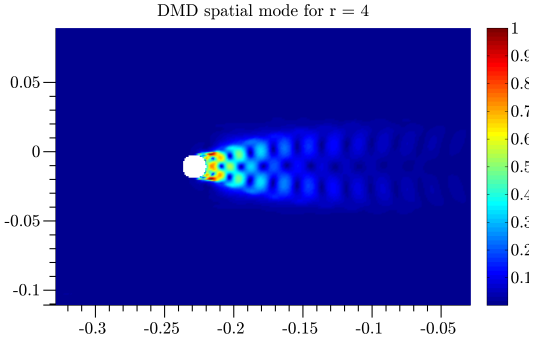


Figure 3.9: DMD spatial mode for r = 4.

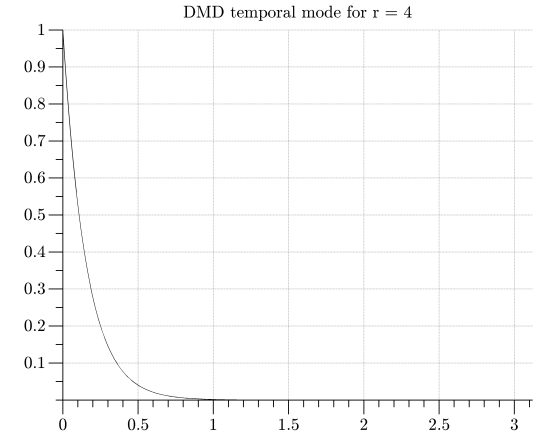


Figure 3.10: DMD temporal mode for r = 4.

3.1.3 Proper Orthogonal Decomposition

The Proper Orthogonal Decomposition is performed on the data matrices **U** and **V** joined together.

The time step chosen for the approximation is 0.01.

The procedure of performing 2D POD is then the same as described in the section 1.3.

The rank r chosen for the analysis is 4.

POD Results

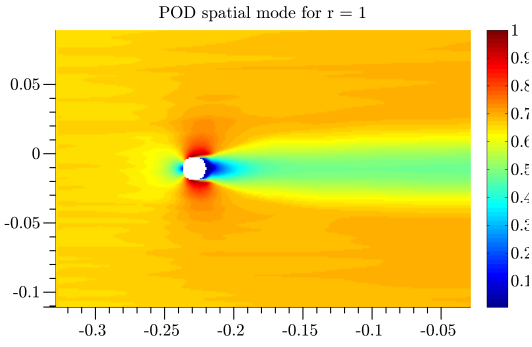


Figure 3.11: POD spatial mode for r = 1.

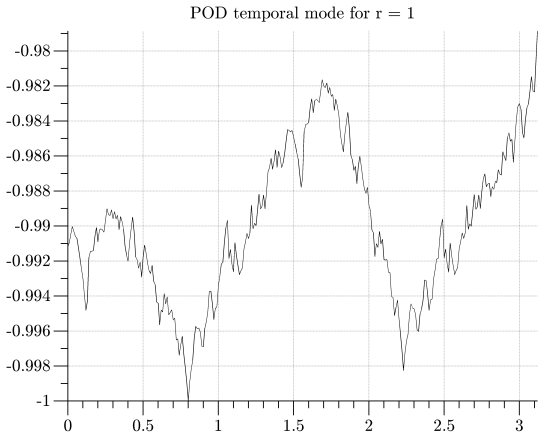


Figure 3.12: POD temporal mode for r = 1.

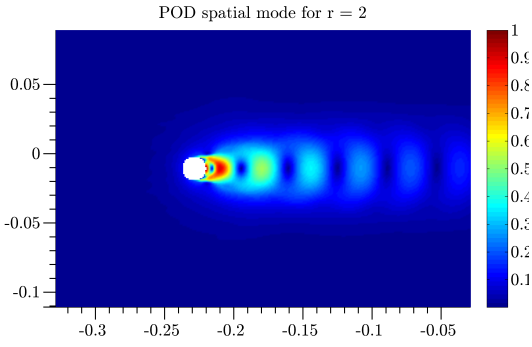


Figure 3.13: POD spatial mode for r = 2.

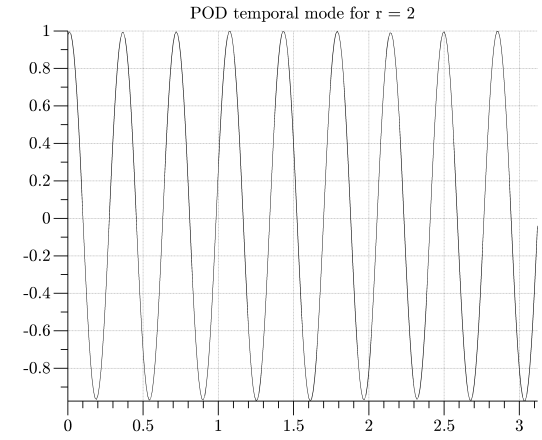


Figure 3.14: POD temporal mode for r = 2.

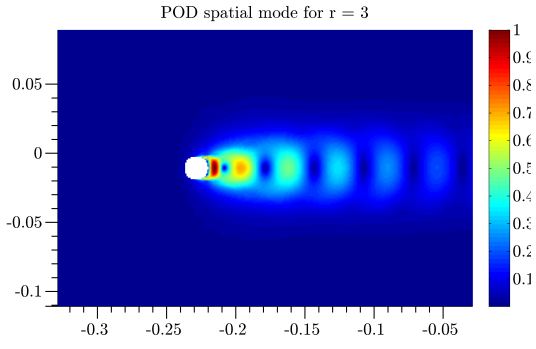


Figure 3.15: POD spatial mode for r = 3.

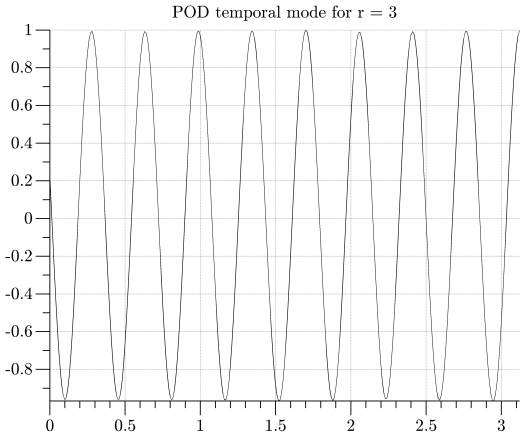


Figure 3.16: POD temporal mode for r = 3.

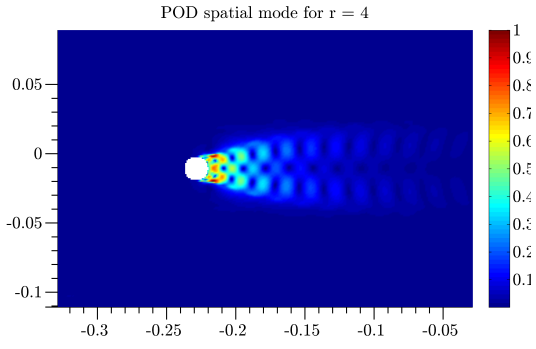


Figure 3.17: POD spatial mode for r = 4.

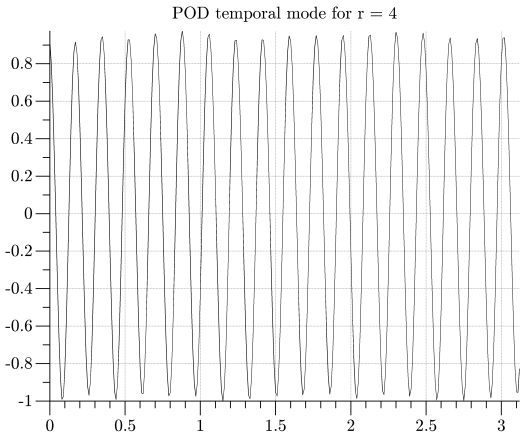


Figure 3.18: POD temporal mode for r = 4.

3.1.4 Conclusions and Comparison of the Two Decomposition Methods

The third spatial structure of DMD and the first spatial structures of POD are approximating the mean flow. The temporal structures associated with the mean flow are almost constant for both approximation methods.

The first two temporal modes of DMD have the same wavelength.

The POD temporal modes (apart from the constant one) are shifted in phase.

Even though in the DMD method the amplitudes of the aproximation are not ordered, with the first two modes we already see the vortex pattern of the flow behind the cylinder.

The time evolution of the approximated flow behind the cylinder is very similar in both methods.

Chapter 4

GUI Beta Version

In this chapter we present a description of a developed beta version GUI program in Matlab to load data, perform POD or DMD, post-process and save the results. This version of GUI is under development and not all the elements are yet implemented or are guaranteed to work without error.

4.1 Scheme of the Program

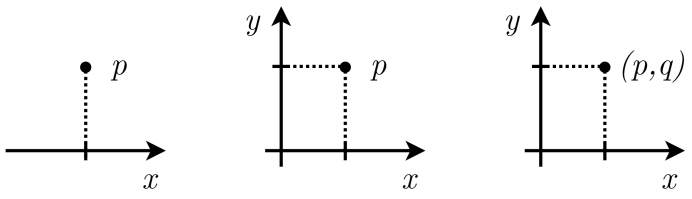
The scheme of the program is presented in Figure 4.2.

The main menu of the GUI is `POD_DMD_beta_1`. This menu needs two string variables specified by the user in the two preceding windows:

`String_An_Type`	which specifies the analysis type
`String_Dec_Type`	which specifies the decomposition type

The user has three choices for the analysis type:

`1DS`	which is 1D data scalar analysis (associates one physical quantity p to one coordinate in space x)
`2DS`	which is 2D data scalar analysis (associates one physical quantity p to two coordinates in space x and y)
`2DV`	which is 2D data vector analysis (associates two physical quantities p and q to two coordinates in space x and y)



1D scalar 2D scalar 2D vector

Figure 4.1: Analysis type: 1D scalar, 2D scalar, 2D vector.

The user has two choices for the decomposition type:

`POD`	which is Proper Orthogonal Decomposition
`DMD`	which is Dynamic Mode Decomposition

The choices made will appear as a reminder in the Matlab command window.

Once in the main menu, four buttons are available:

`Import Data`	opens an `IMPORT` menu
`Decompose`	opens a window to input a variable dt and then opens either `POD_CRITERIA` or `DMD_CRITERIA` menu
`Export Results`	opens `EXPORT_DATA` menu
`Exit`	exits the GUI

The `IMPORT` menu has three buttons where the user can chose the type of data to load into the program:

`Sampled OpenFOAM`	are OpenFOAM datasets
`TxT Dataset`	are files with the `.txt` extension
`Mat Files`	are Matlab files with the `.mat` extension

So far, the `TxT Dataset` is not implemented. Selecting `Sampled OpenFOAM` or `Mat Files` is possible.

When the user choses `Sampled OpenFOAM`, another GUI developed by other student [10] opens, where the user can decide to apply mask to the OpenFOAM dataset or downsample directly without creating a mask.

When the user choses `Mat Files`, a line appears in the Matlab command window to remind what type of data the program is expecting for a given type of analysis. **Notice, that the data files names and the variable names inside of the files must agree with the names requested by the program.** They are:

```
D.mat
y.mat
```

for 1D scalar analysis,

```
U.mat
X.mat
Y.mat
```

for 2D scalar analysis, and:

```
U.mat
V.mat
X.mat
Y.mat
```

for 2D vector analysis.

A pop-up window then appears, where the user can select data to be imported. The function used to import data is called `uipickfiles.m`. Multiple selection is possible.

Once the data is imported, the user should select the `Decompose` button. A pop-up window appears where the user can enter the time step dt of the data. The choice of the time step is many times arbitrary but a note should be made, that the data should have been sampled at equal time steps dt.

Next, according to the decomposition method chosen, either the `POD_CRITERIA` or `DMD_CRITERIA` window appears. These two windows allow the user to select the criteria on the choice of rank r.

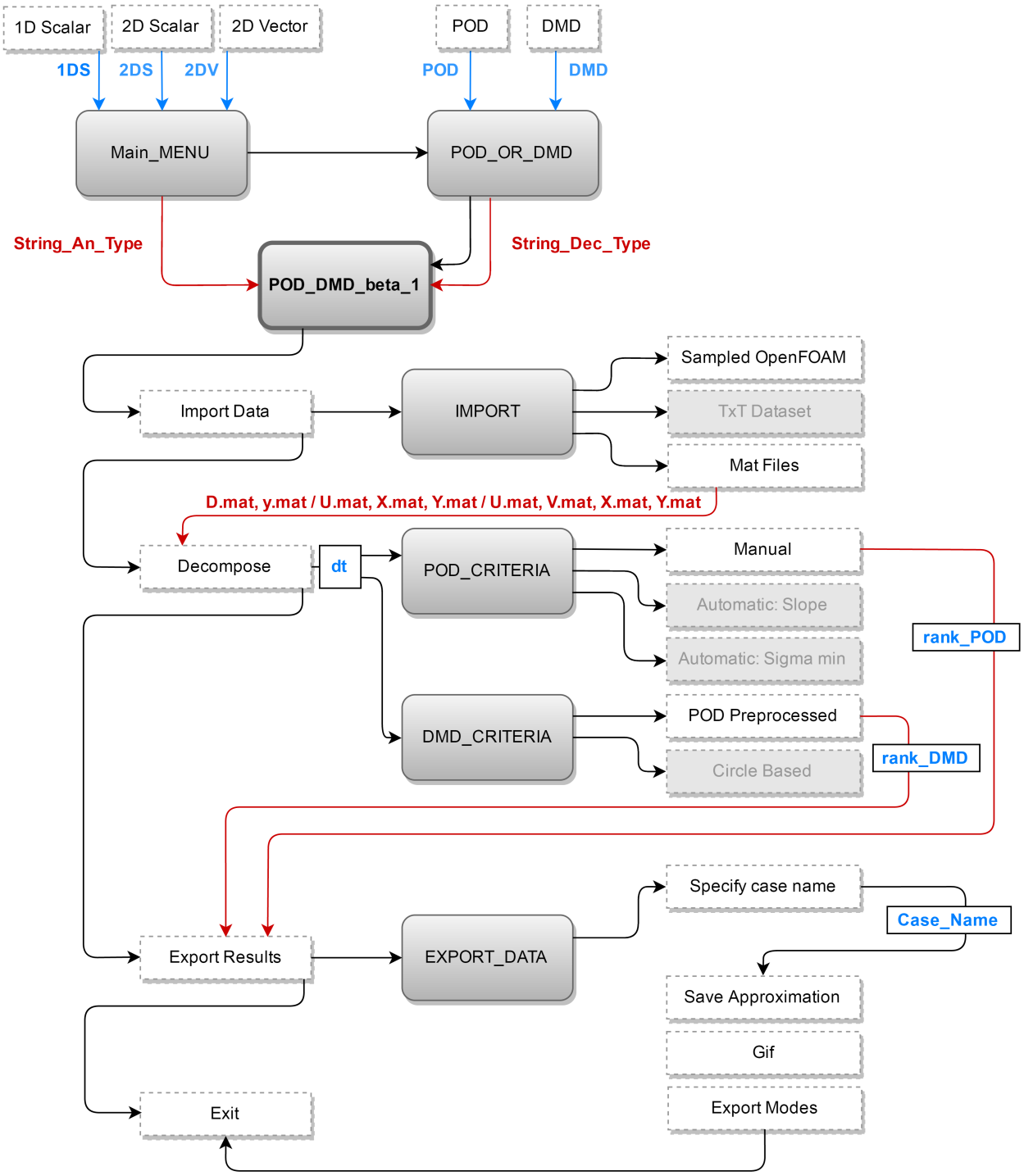


Figure 4.2: Scheme of the POD and DMD GUI.

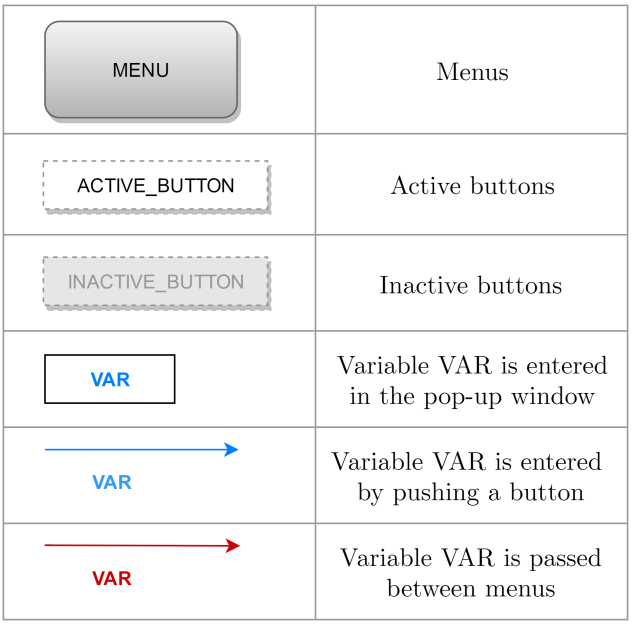


Figure 4.3: Legend of the scheme elements.

In the `POD_CRITERIA` three buttons are available:

`Manual`	where the manual selection of r is made
`Automatic: Slope`	where the automatic slope selection of r is made
`Automatic: Sigma min`	where the automatic $\sigma_{\min}$ selection of r is made

In the `DMD_CRITERIA` two buttons are available:

`POD Preprocessed`	where the manual selection of r is made
`Circle Based`	where the automatic circle based selection of r is made

So far, only the `Manual` and `POD Preprocessed` selections are possible, both of which require the user to perform the choice of the rank r himself.

When the manual selection is chosen, for both decomposition methods, the user receives a graph of the amplitude decay rate of the imported data, which he can zoom in in the representative region and which in turn helps the user to make the initial choice of r. Once the user is ready with his choice, he should press Enter in the Matlab command window. The pop-up window appears, where the user can enter the value of the rank.

Finally, the `Export Results` button opens the `EXPORT_DATA` menu. Four buttons are available, regardless of the decomposition method and analysis type:

`Specify case name`	which allows the user to enter the name of the case
`Save Approximation`	which saves the new approximated matrices as `.mat` files
`Gif`	which saves the `.gif` file of the approximation
`Export Modes`	which saves the `.png` files with graphs of the modes of the approximation

The user must start with the first button for specifying the case name. A pop-up window appears, where a name of the case can be entered. This name will then appear in the name of a created folder, where all the results will be saved. The name of the folder will always have the following pattern:

`[Analysis type]_[Decomposition type]_[User entered case name]`

After the name is entered, the approximated matrices can be saved in the created folder by clicking `Approximation`. Notice that this may take long time if the matrices are of a large size. If not needed, the user may skip saving the approximated matrices.

Then, a `.gif` file with the graphical representation of the results can be viewed and saved by clicking `Gif`.

Finally, to export the modes of the approximation, the user can press the button `Modes`. A pop-up window appears, where the user can select which mode to export. The number that the user enters should be less than or equal to the rank r chosen before.

Notice that the user can decide to save only some of the results, in particular only the Gif and Modes might be chosen.

After saving the results, the user may close the GUI by pressing `Exit`. A note will appear in the Matlab command window, that the GUI is closed.

4.2 Function Executions Inside the Program

There are external Matlab functions, which are utilised by the GUI. They have to be placed in the same directory as the GUI files in order for all the elements to work properly. The following functions are used:

`uigetfiles.m`	function for selecting multiple data files and loading them into Matlab
`POD_2D_S.m`	function that performs 2D scalar POD
`POD_2D_V.m`	function that performs 2D vector POD
`DMD_1D.m`	function that performs 1D DMD
`DMD_2D.m`	function that performs 2D scalar and vector DMD
`POD_1D_PLOT_GIF.m`	function that plots a `.gif` file of the 1D POD approximation
`POD_1D_PLOT_MODES.m`	function that plots a `.png` file of the 1D POD modes
`POD_2D_PLOT_GIF.m`	function that plots a `.gif` file of the 2D POD approximation
`POD_2D_PLOT_MODES.m`	function that plots a `.png` file of the 2D POD modes
`DMD_1D_PLOT_GIF.m`	function that plots a `.gif` file of the 1D DMD approximation
`DMD_1D_PLOT_MODES.m`	function that plots a `.png` file of the 1D DMD modes
`DMD_2D_PLOT_GIF.m`	function that plots a `.gif` file of the 2D DMD approximation
`DMD_2D_PLOT_MODES.m`	function that plots a `.png` file of the 2D DMD modes
`AXIS.m`	function for setting the plot style

The function executions inside menus are presented in the Figure 4.4. It specifies which menus execute a particular function. This figure is useful when any input or output variables are to be changed in any of these functions. They have to be then adjusted in the corresponding menu as well. Note also that the plotting functions executed by the `EXPORT_DATA` menu use outputs from the functions performing 1D and 2D POD and DMD. If the output variables inside these functions are changed, they have to be adapted in the plotting functions. Function `AXIS.m` is used inside every plotting function.

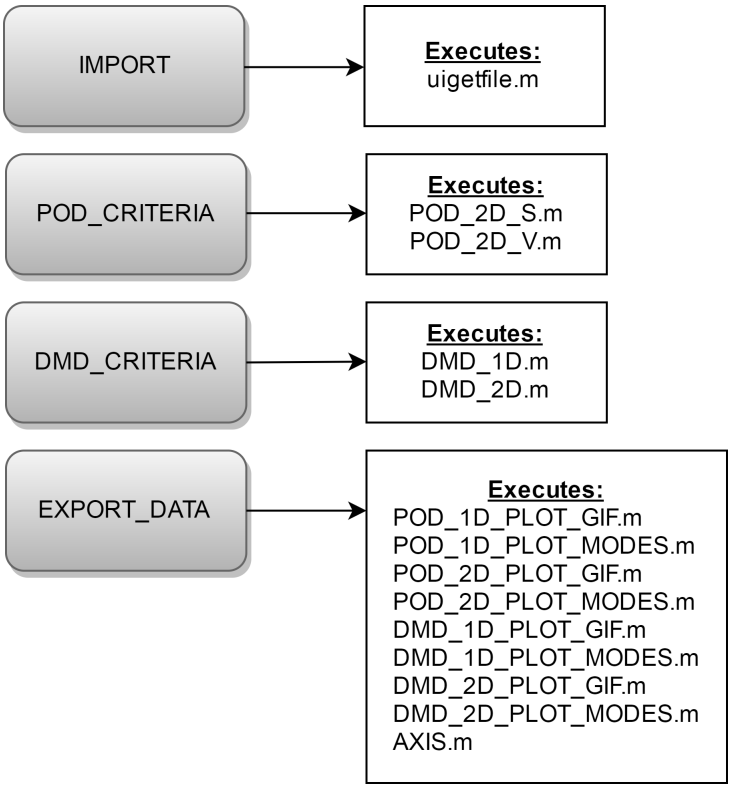


Figure 4.4: Function executions inside GUI.

4.3 Tutorial on Using the GUI

This section is a small tutorial on running the GUI program and performing the POD and DMD on a sample data, coming from the Poiseuille flow (for 1D scalar case) and the flow behind a cylinder (for 2D vector case).

4.3.1 Tutorial Folder

To successfully run this tutorial you need a tutorial folder named `POD_DMD_Vbeta1`. Inside this folder you should see the following data folders:

folder `1D_Data_full_set`:	folder `2D_Data_full_set_S`:	folder `2D_Data_full_set_V`:
`- D.mat`	`- U.mat`	`- U.mat`
`- extracting_1D_data_set.m`	`- X.mat`	`- V.mat`
`- y.mat`	`- Y.mat`	`- X.mat`
		`- Y.mat`

along with the all the Matlab functions that create the GUI.

4.3.2 1D Data

Folder `1D_Data_full_set` contains sample data from the Poiseuille flow described in Chapter 2. This data has been prepared in the form of two matrices: `D.mat` and `y.mat`, for fixed parameters:

`TIME = 10`	total time of the simulation
`W = 10`	Womersley number
`Pa = 60`	dimensionless pressure coefficient
`dt = 0.1`	time step
`dy = 0.01`	space step

In case you want to extract data with different parameters, a code for extracting is included, where you can change things and create new data sets. You can use the code `extracting_1D_data_set.m`.

For now, this definition of parameters results in a matrix $\mathbf{D}$ of size 201×101, which has the same structure as a general data matrix from Figure 1.1 - the rows are related to space coordinates and the columns are related to time coordinates.

The time vector has entries ranging from 0 to TIME, with a time step 0.1. It contains 101 elements.

The space vector has entries ranging from -1 to +1, with a space step 0.01. It contains 201 elements. It represents the position between two parallel plates in the Poiseuille flow.

Open Matlab and change the working directory to the tutorial folder:

```
POD_DMD_Vbeta1
```

Then type `Main_MENU` in the Matlab command window.

NOTE: While proceeding with this tutorial, observe also some feedback information that appears in the Matlab command window when you interact with the GUI.

The first window should appear, where you are asked to select the decomposition mode. Click `1D Scalar`. In the second window you are asked to select the decomposition method. Let's say we will perform the 1D analysis with the POD method, so click `Proper Orthogonal... (POD)`.

Now you are in the main menu and you have to follow the order of the buttons. First, we want to import our data, so click `Import Data`. You are now asked to chose the type of data that you want to import and for now, click `Mat Files`, since our matrices are stored with the extension `.mat`.

Find the folder `1D_Data_full_set` inside the tutorial folder and select both files (by mouse or by holding Shift) `D.mat` and `y.mat`.

Once the data is imported, click the next button `Decompose`. A pop-up window appears, where you are asked to enter the time step in your data. You are in fact free to chose whatever time step you want but if you want to be consistent with how the data was prepared, change the default value to 0.1 (otherwise the total time of simulation will differ from TIME = 10). A window with the choice of criteria on the rank r appears. For the moment, we will chose the rank manually, so click `Manual`. You now receive a graph of the amplitude decay rate of the imported data. This graph helps to make a choice on the rank r. You see that the first mode has the largest amplitude, the second and the third mode has got a lower amplitude and the furhter modes have zero (or almost zero) amplitudes. In general, you can zoom in the graph in the region that it of interest to you to help you make the decision on r. For this imported data it's enough to take three first modes. The program is waiting for your response, and once you're ready to type the rank, press Enter in the Matlab command window.

A pop-up window appears, where you can type 3 and click `OK`. The graph now disappears and the rank is chosen.

The GUI is now ready to process and save your data! Click `Export Results`.

In the `Export Data` menu we have to start with specifying the name of your case, so click `Specify case name`. In the pop-up window you are asked to create a folder name for saving your case. You are free to enter whatever you want. The program is smart though, and always

adds a prefix before your case name. So, even if you create a meaningless name, like `AAAA`, you will still know what kind of analysis was performed, as the folder name will be `1D_POD_AAAA`.

After the name is entered, you have three results that you can save:

1. the approximated matrix `D_POD.mat`
2. the `.gif` file with the movie of a pulsating velocity profile with its approximation
3. the `.png` file with the graph of the spatial and temporal structures of the approximation

In general, you can decide to only save the things that you need, and you can skip eg. saving the `.mat` files (as sometimes this might take a long time). But this time, just to test whether everything is working properly, we will save all the results.

Click `Save Approximation`. Once clicked, the results should be saved as `.mat` files automatically.

Move on to `Gif` button. A pop-up window appears, where you can specify the ranges in the x and y axes. The default values are adjusted to the Poiseuille case, so you can simply click `OK`.

You should now see a nice movie of the pulsating velocity profile of the Poiseuille flow. When the window with the Matlab figure closes, the `.gif` file is saved.

The final thing is to save the modes of the approximation. The maximum number of the firs modes that you can save is equal to the rank number r previously entered. Each time you can chose which mode to export.

You can simply click the button `Export Modes` and in the pop-up window type the number of mode that you want to save (1, 2 or 3). Just to try it out, it's recommended that you save all three, one by one.

You can now click `Exit` to close the GUI.

Go now to the tutorial folder:

```
POD_DMD_Vbeta1
```

and notice that a new folder with your case name was created there. In that folder you should see five files:

`D_POD.mat`	`.mat` file with the approximated matrix
`GIF_1D_POD_r3.gif`	`.gif` file with the movie of a pulsating velocity profile
`MODES_1D_POD_r1.png`	`.png` file with the graph of the 1st POD mode
`MODES_1D_POD_r2.png`	`.png` file with the graph of the 2nd POD mode
`MODES_1D_POD_r3.png`	`.png` file with the graph of the 3rd POD mode

You can double-click the `.gif` file to watch it and you can view the `.png` files with the POD modes.

As an additional exercise, you can run the 1D case with the DMD method, which will be analogous to the POD. You can also move on to testing the 2D data, where we will perform DMD.

4.3.3 2D Data

To test the 2D case inside GUI, we use the sample data from the flow behind a cylinder. We use reduced region data, as the full set might take a long time to process. In any case, please be patient with the GUI this time, as the matrices get larger then they were in the 1D case!

The 2D vector data is prepared in the form of four matrices: `U.mat`, `V.mat`, `X.mat` and `Y.mat`, obtained from the CFD simulation.

Matrices $\mathbf{U}$ and $\mathbf{V}$ have the same structure as presented in the Figure 3.2.

The matrix $\mathbf{U}$ has size 1476×128.

The matrix $\mathbf{V}$ has size 1476×128.

The vector $\mathbf{X}$ has 41 elements.

The vector $\mathbf{Y}$ has 36 elements.

Make sure you are still in the working directory of the tutorial folder:

`POD_DMD_Vbeta1`

Type `Main_MENU` in the Matlab command window.

This time chose `2D Vector` in the first window and `Dynamic Mode... (DMD)` in the second window.

Once in the main menu, import the 2D dataset: click `Import Data`, then chose `Mat Files`. Find the folder `2D_Data_full_set_V` inside the tutorial folder and select four files `U.mat`, `V.mat`, `X.mat` and `Y.mat`.

Next, click `Decompose` and specify the timestep dt. You can enter any value you want, eg. type 0.05.

In the choice of criteria on r menu, click `POD Preprocessed`. This will allow you to manually enter the rank. You will first see the amplitude decay rate of the imported data and it is now recommended that you zoom in the first few amplitudes to see them precisely. The amplitudes diverge to zero and the first 7 amplitudes seem to be the largest. Once you are ready with your choice, click Enter in the Matlab command window. In the pop-up window type 7.

It's time to export our results. Click `Export Results`. Start with specifying the name for your case - you can type anything you want.

Click `Save Approximation` to export the approximated matrices `U_DMD.mat` and `V_DMD.mat`.

Click `Gif` to draw the movie of the flow behind a cylinder. In the pop-up window you can enter the range of your data. You can simply click `OK`.

You should now be seeing a nice colour plot!

Once it is saved, click `Export Modes`. This time we will not save all 7 modes. Suppose, we only want to save the first one, so type 1 in the pop-up window.

You first see the spatial mode and by clicking Enter in the Matlab command window, you move on to the temporal mode.

Click Enter again and you see the graph of the eigenvalues on the complex plane. The red mode is the one that you are exporting at the moment.

Click Enter one more time and the graphs will be closed and saved.

You can exit the GUI by clicking `Exit`.

As the final thing, check if the results are saved. Go to the tutorial folder:

`POD_DMD_Vbeta1`

and find a new folder corresponding to your 2D vector analysis. Inside it, you should see five files:

`U_DMD.mat`	`.mat` file with the approximated matrix
`V_DMD.mat`	`.mat` file with the approximated matrix
`GIF_2D_DMD_r7.gif`	`.gif` file with the movie of an approximated flow behind a cylinder
`Spatial_Modes_2D_DMD_r1.png`	`.png` file with the graph of the 1^{st} DMD spatial mode
`Temp_Modes_2D_DMD_r1.png`	`.png` file with the graph of the 1^{st} DMD temporal mode

Chapter 5

Additional Exercises

5.1 Discrete and Continuous Norms

In the following exercise we seek a relationship between the norm calculated in a discrete way (as defined in the Euclidean space) and in a continuous way (as defined in the Hilbert space).

In Euclidean space we define an operation between two vectors $\mathbf{x}$ and $\mathbf{y}$ called inner product:

$$\langle \mathbf{x}, \mathbf{y} \rangle = \sum_{i=1}^{n} x_i y_i \tag{5.1}$$

which results in a scalar.

We also define a square of the norm of a vector by performing an inner product of that vector with itself:

$$||\mathbf{x}||^2 = \langle \mathbf{x}, \mathbf{x} \rangle \tag{5.2}$$

The concept of an inner product and computing the norm carries over from Euclidean space to the Hilbert space, where it is defined for continuous functions within a certain domain.

We have therefore an inner product between two functions $f(x)$ and $g(x)$:

$$\langle f(x), g(x) \rangle = \int_{-1}^{1} f(x) g(x) dx \tag{5.3}$$

and, once again, a square of the norm of a function $f(x)$ is defined as an inner product of the function with itself:

$$||f(x)||^2 = \int_{-1}^{1} f^2(x) dx \tag{5.4}$$

The functions chosen for this exercise are two Legendre polynomials L_3 and L_4:

$$L_3 = \frac{1}{2}\left(5x^3 - 3x\right) \tag{5.5a}$$

$$L_4 = \frac{1}{8}\left(35x^4 - 30x^2 + 3\right) \tag{5.5b}$$

For the purpose of discrete computing we discretize space (x-axis) into n_x points. Functions L_3 and L_4 become vectors with the number of elements equal to n_x. The square of the norm of each function is calculated in three ways for comparison:

1. using the command `trapz()` which approximates the definite integral by calculating the sum of areas of small trapezoids. It is in fact a discrete calculation of a continuous case.

2. using the command `norm()` which computes the norm of a vector

3. using the command `dot()` which computes a dot product of each function with itself

In the continuous case, the integrals are computed analytically and evaluated in the code for the specified integration boundaries.

NOTE: An inner product of functions L_3 and L_4 is also computed and since the Legendre polynomials form an orthogonal basis, we expect it to be zero.

$$\langle L_3, L_4 \rangle = 0 \quad \text{in the continuous case} \tag{5.6a}$$

$$\langle L_3, L_4 \rangle \approx 0 \quad \text{in the discrete case} \tag{5.6b}$$

Listing 5.1: Matlab code to test the relationship between a discrete and continuous evaluation of the norm of two Legendre polynomials. *disc_cont_exercise_1.m*

```
%% Finding a Relationship Between a Discrete and Continuous ==========
% Evaluation of a Norm of Two Functions
% ====================================================================
% NOTE: This code is actually evaluating a square of the norm
% for simplicity.
% ====================================================================
clc, clear

% Number of points to discretize the integration interval into:
n_x = 500;

%% Discretizing the integration interval: ============================
a = -1; b = 1;           % integration boundaries
step = (b-a)/(n_x - 1); % discretization step on the interval
x = [a:step:b];          % x-axis as a vector

%% Functions to be calculated (two Legendre polynomials): ============
L3 = 1/2 * (5*x.^3 - 3*x);          % L3 as a vector
L4 = 1/8 * (35*x.^4 - 30*x.^2 + 3); % L4 as a vector

%% Approximation of a definite integral of L3 and L4 function: =======
Area_L3 = trapz(x,L3);  % approximation to the area below L3
Area_L4 = trapz(x,L4);  % approximation to the area below L4

%% Approximation of the inner product of L3 and L4: ==================
IP = L3.*L4;            % multiplying the two functions L3 and L4

% Discrete calculation of the inner product:
IP_t = trapz(x, IP);    % discrete using trapz()
IP_d = dot(L3, L4);     % discrete using dot()

%% Approximation of the norm of function L3: =========================
L3_L3 = L3.^2;          % multiplying L3 with itself

% Discrete calculation of the norm:
NL3_t = trapz(x,L3_L3);     % discrete using trapz()
NL3_n = norm(L3)^2/(n_x/2); % discrete using norm()
```

```
NL3_d = dot(L3,L3)/(n_x/2); % discrete using dot()

% Continuous calculation of the norm:
NL3_a = b^3*(25*b^4-42*b^2+21)/28-a^3*(25*a^4-42*a^2+21)/28;

%% Approximation of the norm of function L4: ==========================
L4_L4 = L4.^2;              % multiplying L4 with itself

% Discrete calculation of the norm:
NL4_t = trapz(x,L4_L4);      % discrete using trapz()
NL4_n = norm(L4)^2/(n_x/2); % discrete using norm()
NL4_d = dot(L4,L4)/(n_x/2); % discrete using dot()

% Continuous calculation of the norm:
NL4_a = b*(1225*b^8-2700*b^6+1998*b^4-540*b^2+81)/576-  ...
    a*(1225*a^8-2700*a^6+1998*a^4-540*a^2+81)/576;
```

The numerical results of the code are presented below:

```
IP_t = -5.2042e-018
IP_d = 1.4433e-015

NL3_t = 0.28575
NL3_n = 0.28917
NL3_d = 0.28917
NL3_a = 0.28571

NL4_t = 0.22228
NL4_n = 0.22583
NL4_d = 0.22583
NL4_a = 0.22222
```

It is seen therefore, that in order to match up the norms calculated using the analytic solution with the ones calculated using discrete methods, we had to divide the latter by $n_x/2$.

As a result of this exercise we can therefore conclude that:

$$\int_{-1}^{1} f^2(x)dx \approx \frac{||\mathbf{f}||^2}{\frac{n_x}{2}} \tag{5.7}$$

and hence in reverse:

$$||\mathbf{f}|| \approx \sqrt{\frac{n_x}{2}\int_{-1}^{1} f^2(x)dx} \tag{5.8}$$

where $\mathbf{f}$ is a discrete vector and $f(x)$ is a continuous function. We can also put it in words that:

$$\Big(\text{discrete norm}\Big)^2 \approx \frac{n_x}{2}\Big(\text{continuous norm}\Big) \tag{5.9}$$

The approximation gets better as we increase the number of discretization points n_x.

The following result might be useful for approximating integrals which can be written in the general form as:

$$\int_{-1}^{1} f^2(x)dx$$

5.2 The Phase Shift Ψ Between Two Sine Functions

In this exercise we retrieve a phase shift Ψ between two sine functions by computing the inner product of two discrete vectors $\mathbf{y}_1$ and $\mathbf{y}_2$, defined in the following way:

$$\mathbf{y}_1 = \sin(\mathbf{x}) \tag{5.10a}$$

$$\mathbf{y}_2 = \sin(\mathbf{x} + \Psi) \tag{5.10b}$$

where $\mathbf{x}$ is a vector obtained by discretizing the x-axis and Ψ is the pre-specified phase.

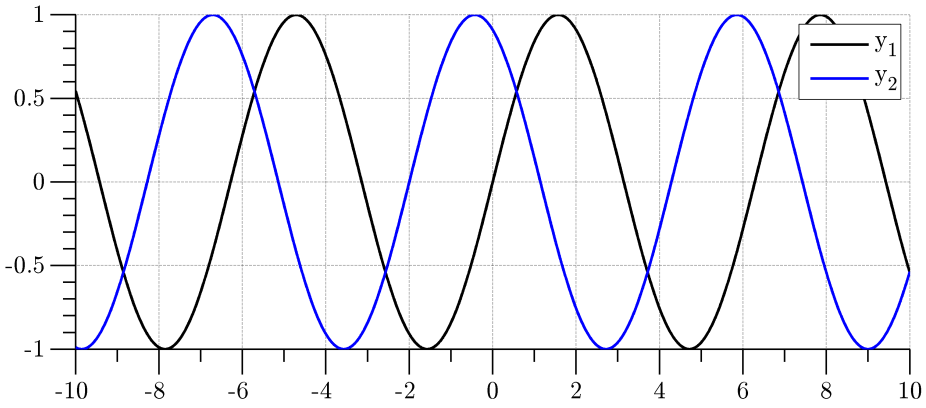


Figure 5.1: Two sine functions shifted in phase by $\Psi = 2$.

The idea is to find the *correlation coefficient* ρ between these two vectors $\mathbf{y}_1$ and $\mathbf{y}_2$. The correlation coefficient is defined as a cosine of the angle Ψ between two vectors and is computed by means of an inner product:

$$\rho = \cos(\Psi) = \frac{\langle \mathbf{y}_1, \mathbf{y}_2 \rangle}{||\mathbf{y}_1|| \cdot ||\mathbf{y}_2||} \tag{5.11}$$

The correlation coefficient is hence a number between -1 and $+1$ and is related to the phase shift. For example, it is zero when the phase is $\pi/2$ or $3\pi/2$, -1 when the phase is π and 1 when the phase is 0 or 2π.

Finally, as we take the $\arccos(\rho)$ we get the phase back, since:

$$\arccos(\rho) = \arccos(\cos(\Psi)) = \Psi \tag{5.12}$$

There is, however, a problem with retrieving the phase shift exactly. This is due to the symmetry of the correlaction coefficient with respect to the phase Ψ. This symmetry is captured in the figure below:

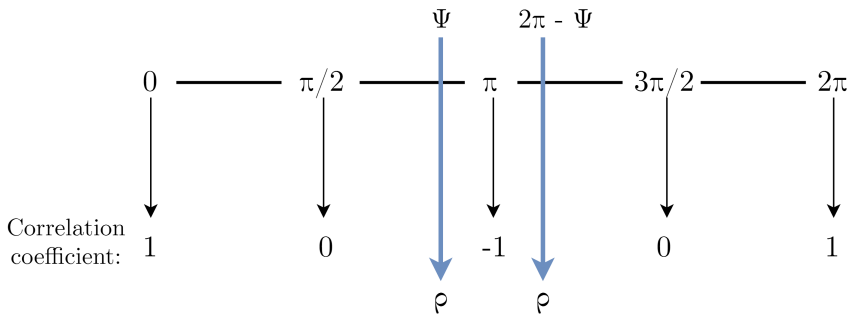


Figure 5.2: Symmetry in the correlation coefficient.

For a certain correlation coefficient ρ we cannot tell whether the phase shift is Ψ or $2\pi - \Psi$. We therefore compute both. In the graphical representation, this means that we cannot tell whether vector $\mathbf{y}_2$ was shifted to the left or to the right of vector $\mathbf{y}_1$.

Listing 5.2: Matlab code for retrieving the phase shift between two sine functions. *phase_shift_of_two_sines.m*

```
%% Phase shift between two sine functions ==========================
% Retrieving the phase shift by computing an inner product.
% ==================================================================
clc
clear

%% USER INPUT: =====================================================
PHASE = 2;              % pre-specified phase
xstart = -10;           % start interval on the x-axis
xend = -xstart;         % end interval on the x-axis
step = 0.01;            % step on the x-axis
% END OF USER INPUT ==============================================

%% Computing the actual phase: =====================================
Actual_phase = rem(PHASE, 2*pi);
disp(['The pre-specified phase is: ', num2str(Actual_phase)])

%% Define the span on the x-axis: ==================================
x = [xstart:step:xend];

%% Define the vectors: =============================================
y1 = sin(x);
y2 = sin(x + PHASE);

%% Find the inner product of two vectors y1 and y2: ================
Inner_prod = dot(y1, y2);
NORMy1 = norm(y1);
NORMy2 = norm(y2);
CORRELATION = Inner_prod/(NORMy1*NORMy2);
disp(['The correlation coefficient is: ', num2str(CORRELATION)]);

%% Retrieving the phase: ===========================================
PHASE_back = acos(CORRELATION);
mirror_phase = pi + (pi-PHASE_back);
disp(['Retrieved phase: ', num2str(PHASE_back)]);
disp(['or: ', num2str(mirror_phase)]);
```

In the Matlab code presented above, we calculate the actual, minimum phase, from the pre-specified one, taking into account that shifting the sine function by any $2k\pi$ can be neglected.

The numerical results of the code are presented below:

```
The pre-specified phase is: 2
The correlation coefficient is: -0.40054
Retrieved phase: 1.9829
or: 4.3003
```

NOTE: As we increase the span on the x-axis, and/or decrease the step in vector $\mathbf{x}$, the approximation to the phase gets better.

5.3 A Note on the Sizes of Component Matrices in the SVD

5.3.1 SVD on a General Matrix D

When the SVD is performed on a general matrix $\mathbf{D}$ of size $n \times m$ the sizes of the resultant matrices are as follows:

The matrix $\mathbf{U}$ is $n \times n$ and is always a square matrix.

The matrix $\mathbf{\Sigma}$ is $n \times m$ and is the same size as the matrix $\mathbf{D}$.

The matrix $\mathbf{V}$ is $m \times m$ and is always a square matrix.

The matrix $\mathbf{D}$ can be written by means of:

$$\mathbf{D} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T \tag{5.13}$$

The matrix multiplication from the equation (5.13) is represented graphically in the figure (5.3).

5.3.2 SVD on the Matrix $\mathbf{X}_1$ from DMD

When the SVD is performed on a general matrix $\mathbf{X}_1$ of size $n_p \times (n_t - 1)$ the sizes of the resultant matrices are as follows:

The matrix $\mathbf{U}$ is $n_p \times n_p$ and is always a square matrix.

The matrix $\mathbf{\Sigma}$ is $n_p \times (n_t - 1)$ and is the same size as the matrix $\mathbf{X}_1$.

The matrix $\mathbf{V}$ is $n_t \times n_t$ and is always a square matrix.

The matrix $\mathbf{X}_1$ can be written by means of:

$$\mathbf{X}_1 = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T \tag{5.14}$$

The matrix multiplication from the equation (5.14) is represented graphically in the figure (5.4).

5.3.3 SVD with POD Approximation on Matrices D and $\mathbf{X}_1$

After approximating the matrices $\mathbf{D}$ and $\mathbf{X}_1$ with the POD method, the sizes of decomposition matrices change. Suppose the rank of the approximation is r.

$$\widetilde{\mathbf{D}}_r \approx \mathbf{U}(:, 1:r)\mathbf{\Sigma}(1:r, 1:r)\mathbf{V}(:, 1:r)^T \tag{5.15}$$

The matrix multiplication from the equation (5.15) is represented graphically in the figure (5.5).

$$\widetilde{\mathbf{X}}_{1,r} \approx \mathbf{U}(:, 1:r)\mathbf{\Sigma}(1:r, 1:r)\mathbf{V}(:, 1:r)^T \tag{5.16}$$

The matrix multiplication from the equation (5.16) is represented graphically in the figure (5.6).

The sizes of $\widetilde{\mathbf{D}}_r$ and $\widetilde{\mathbf{X}}_{1,r}$ are the same as the sizes of $\mathbf{D}$ and $\mathbf{X}_1$.

When the POD approximation is performed on a general matrix $\mathbf{D}$ of size $n_p \times n_t$ the sizes of the resultant matrices are as follows:

The matrix $\mathbf{U}$ is $n_p \times r$ and is in general a rectangular matrix.

The matrix $\mathbf{\Sigma}$ is $r \times r$ and becomes a square matrix.

The matrix $\mathbf{V}$ is $n_t \times r$ and is in general a rectangular matrix.

When the POD approximation is performed on a general matrix $\mathbf{X}_1$ of size $n_p \times (n_t - 1)$ the sizes of the resultant matrices are as follows:

The matrix $\mathbf{U}$ is $n_p \times r$ and is in general a rectangular matrix.

The matrix $\mathbf{\Sigma}$ is $r \times r$ and becomes a square matrix.

The matrix $\mathbf{V}$ is $(n_t - 1) \times r$ and is in general a rectangular matrix.

Notice also, that with the assumption that $n_p > n_t$, the maximum rank of the matrix $\mathbf{D}$ is n_t, so the rank of the approximation $\widetilde{\mathbf{D}}_r$ is at most $r = n_t$. The maximum rank of the matrix

$\mathbf{X}_1$ is $n_t - 1$ and analogously the rank of the approximation $\widetilde{\mathbf{X}}_{1,r}$ is at most $r = n_t - 1$. This assumption restricts what the maximum sizes of the component matrices can be.

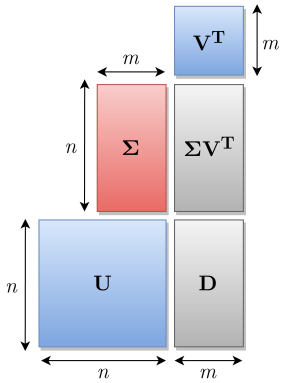


Figure 5.3: Graphical representation of the equation (5.13).

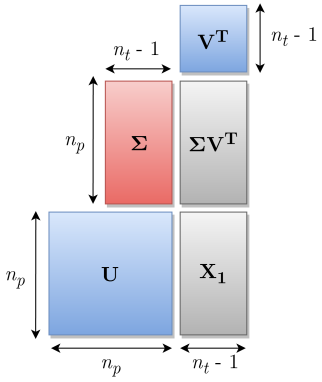


Figure 5.4: Graphical representation of the equation (5.14).

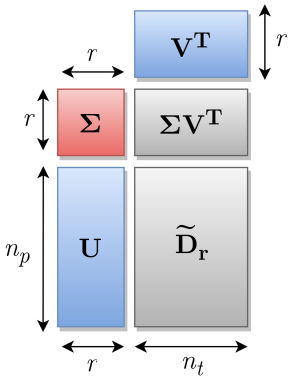


Figure 5.5: Graphical representation of the equation (5.15).

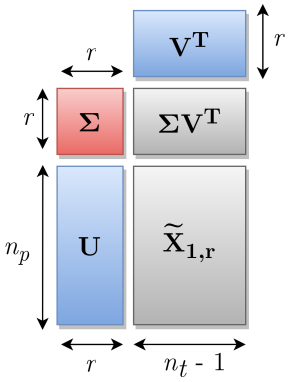


Figure 5.6: Graphical representation of the equation (5.16).

5.4 A Note on the Linear Propagator Matrix

In this section we look deeper into fitting a linear system into the data matrix $\mathbf{D}$, and especially we look at the equation (1.6), which we rewrite below:

$$\mathbf{X}_2 = \mathbf{A}\mathbf{X}_1 \tag{1.6}$$

We investigate whether it is possible to find matrix $\mathbf{A}$ explicitly.

First, let's analyze the sizes of each of the matrices in the above equation. Matrices $\mathbf{X}_1$ and $\mathbf{X}_2$ have size $n_p \times (n_t - 1)$ and since the matrix $\mathbf{A}$ multiplies the matrix $\mathbf{X}_1$ to give a matrix of the same size as $\mathbf{X}_1$, it has to be size $n_p \times n_p$.

We then want to find an equation from which the matrix $\mathbf{A}$ can be solved. Since in general matrix $\mathbf{X}_1$ is not square (unless in a rare case $n_p = n_t - 1$), we cannot compute its inverse directly. We seek a way to find a special kind of inverse, denoted by $\mathbf{X}_1^{-1*}$, so that:

$$\mathbf{X}_2\mathbf{X}_1^{-1*} = \mathbf{A} \tag{5.17}$$

One idea might be to use a Moore-Penrose inverse and compute the matrix $\mathbf{A}$ by means of the least-squares method. We perform a few algebraic operations on the equation (1.6):

$$\mathbf{X}_2 = \mathbf{A}\mathbf{X}_1 \quad / \times \mathbf{X}_1^T \tag{5.18a}$$

$$\mathbf{X}_2\mathbf{X}_1^T = \mathbf{A}\mathbf{X}_1\mathbf{X}_1^T \quad / \times (\mathbf{X}_1\mathbf{X}_1^T)^{-1} \tag{5.18b}$$

$$\mathbf{X}_2\mathbf{X}_1^T(\mathbf{X}_1\mathbf{X}_1^T)^{-1} = \mathbf{A} \tag{5.18c}$$

This will only hold when the product $\mathbf{X}_1\mathbf{X}_1^T$ is invertible.

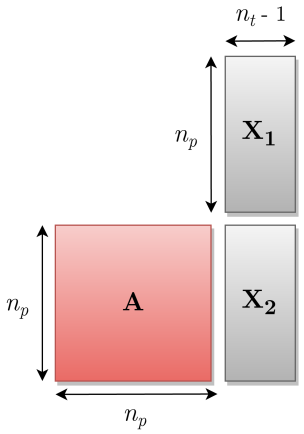


Figure 5.7: Graphical representation of the equation (1.6).

Listing 5.3: Matlab code for attempting to find a linear propagator matrix $\mathbf{A}$. *fitting_linear_systems.m*

```
%% Investigating the linear system of a form X2 = A X1 ==========
% We attempt to find matrix A for a nonlinear set of data and then
% to retrieve matrix A for an artificially created set.
% ================================================================
clc
clear

%% Attempt at fitting a linear propagator matrix A ===============
% into nonlinear data:
% Generation of a data matrix of full-rank:
D = [2 0 0 0 0 ; 0 8 2 0 0 ; 0 0 1 1 0 ; ...
     0 0 0 0 5 ; 0 2 0 0 1 ; 5 0 0 0 1 ; 0 0 0 1 0 ];

% Checking the rank of matrix D:
rank_D = rank(D);

% Extracting data sets:
X1_D = D(:,1:end-1);
X2_D = D(:,2:end);

% Finding a Moore-Penrose inverse:
square_D = X1_D * X1_D';
det(square_D);
inverse_D = inv(square_D);

% Attempt at finding A:
A_D = X2_D * X1_D' * inverse_D;

%% Attempt at retrieving the linear propagator matrix A: =========
% Creating matrix A:
```

```
A = [5 2 1 5 ; 1 2 2 2 ; 3 6 9 5 ; 1 0 2 2];

% Creating data set X1:
X1_A = [1 2 3 ; 5 6 1 ; 0 0 2 ; 2 0 0];

% Computing data set X2:
X2_A = A*X1_A;

% Finding a Moore-Penrose inverse:
square_A = X1_A*X1_A';
det(square_A);
inverse_A = inv(square_A);

% Attempt at getting back A:
A_A = X2_A * X1_A' * inverse_A;
```

In the Matlab code presented above we try to find matrix $\mathbf{A}$ for a data set matrix $\mathbf{D}$ of rank 1. The code produces a warning that the matrix `inverse_D` is singular, which means that the determinant of the product $\mathbf{X}_1\mathbf{X}_1^T$ is equal to or close to 0. Matrix `A_D` is hence not computed by Matlab.

In the second part of the code, we attempt to retrieve the matrix $\mathbf{A}$, specifying it at the begining. We therefore go in reverse, and compute the matrix $\mathbf{X}_2$ that satisfies the equation (1.6). The code produces the same warning about matrix `inverse_A`. The matrix `A_A` gets computed but is in no way retrieving the original matrix `A`. The results are the following:

```
A =

     5     2     1     5
     1     2     2     2
     3     6     9     5
     1     0     2     2

A_A =

   -2.0000    1.5000   -8.0000    4.7500
         0    2.2500         0    2.0000
    6.0000    5.0000   -2.0000    8.5000
    1.2500    0.2500    3.0000    1.8750
```

We might conclude that finding a linear propagator matrix of a linear system is in general impossible without an error.

5.5 A Note on the Similarity of Matrices

In this section we analyze in a closer detail the condition of similarity of two matrices, which is an important concept in the DMD method, where instead of finding a linear propagator matrix $\mathbf{A}$, we find a similar matrix $\mathbf{S}$.

Two matrices are similar if they satifsy the equation:

$$\mathbf{S} = \mathbf{Q}^{-1}\mathbf{A}\mathbf{Q} \tag{5.19}$$

for any matrix $\mathbf{Q}$ that has an inverse.

In the DMD method, we use the orthogonal matrix $\mathbf{U}$ to play a role of a matrix $\mathbf{Q}$. The matrix $\mathbf{U}$ has got an inverse, and it is equal to its transpose:

$$\mathbf{U}^{-1} = \mathbf{U}^T \tag{5.20}$$

We have therefore the equation (1.9) from section 1.4:

$$\mathbf{S} = \mathbf{U}^T\mathbf{A}\mathbf{U} \tag{1.9}$$

We find the size of matrix $\mathbf{S}$, and we compute it from the left hand side of the equation (1.8d), which we recall below:

$$\mathbf{U}^T\mathbf{X}_2\mathbf{V}\mathbf{\Sigma}^{-1} = \mathbf{U}^T\mathbf{A}\mathbf{U} \tag{1.8d}$$

Matrices $\mathbf{U}$, $\mathbf{V}$ and $\mathbf{\Sigma}$ have already been approximated with the POD method, so the dimension from section 5.3.3 apply.

Hence we have the following size multiplication:

$$\text{size}(\mathbf{S}) = (r \times n_p) \cdot (n_p \times (n_t - 1)) \cdot ((n_t - 1) \times r) \cdot (r \times r) \tag{5.21a}$$

$$\text{size}(\mathbf{S}) = (r \times r) \tag{5.21b}$$

We then check whether this size agrees with the equation (1.9) and with the size of the matrix $\mathbf{A}$ obtained in the section 5.4. We have therefore:

$$\text{size}(\mathbf{S}) = (r \times n_p) \cdot (n_p \times n_p) \cdot (n_p \times r) \tag{5.22a}$$

$$\text{size}(\mathbf{S}) = (r \times r) \tag{5.22b}$$

In general, the size of matrix $\mathbf{A}$ is larger then the size of matrix $\mathbf{S}$. The maximum size of matrix $\mathbf{S}$ can be achieved when $r = n_t - 1$ and is in that case equal to $(n_t - 1) \times (n_t - 1)$. We then find the eigenvalues of matrix $\mathbf{S}$, which approximate some of the eigenvalues of $\mathbf{A}$. Producing a matrix $\mathbf{S}$ of size $r \times r$ we can only retrieve r of them.

In a rare case when $n_p = n_t - 1 = r$, the sizes of matrices $\mathbf{S}$ and $\mathbf{A}$ are the same and we can retrieve all eigenvalues of $\mathbf{A}$.

In a Matlab code presented below we investigate the behaviour of the eigenvalues of a similar matrix $\mathbf{S}$, first, when the matrix $\mathbf{S}$ is of the same size as matrix $\mathbf{A}$, and next, when the matrix $\mathbf{S}$ is of a reduced size $r \times r$.

Listing 5.4: Matlab code to investigate the similarity condition. *similar_matrices.m*

```
%% Similar Matrices ========================================================
% In this code we investigate the similarity condition:
% S = U^-1 A U, for two cases:
% - when matrix U is of the same size as matrix A.
% - when matrix U is of reduced size, and approximates only
% r eigenvalues of matrix A.
% =========================================================================
clc
clear

%% USER INPUT: ============================================================
% Choice of rank to decrease the size of matrix S:
r = 7;
% END OF USER INPUT ======================================================

%% Initial data:
% Generation of a dummy data matrix:
D = [2 0 0 0 0 ; 0 8 2 0 0 ; 0 0 1 1 0 ; ...
     0 0 0 0 5 ; 0 2 0 0 1 ; 5 0 0 0 1 ; 0 0 0 1 0 ];

len_D = size(D,1); % length of matrix D
wid_D = size(D,2); % width of matrix D
```

```
23
24 % Generation of a linear propagator matrix A:
25 A = zeros(len_D, len_D);
26
27 for i = 1:1:(len_D*len_D)
28     A(i) = i-1;
29 end
30
31 for i = 1:4:(len_D*len_D)
32     A(i) = i*2;
33 end
34
35 % Finding the eigenvectors and eigenvalues of A:
36 [eigvec_A , eigval_A] = eig(A);
37 E_A = diag(eigval_A)
38
39 %% Full size: ============================================================
40 % Creating an orthogonal matrix U:
41 [U, Sigma, V] = svd(D);
42
43 % Creating a similar matrix S:
44 S = U' * A * U;
45
46 % Finding the eigenvectors and eigenvalues of S:
47 [eigvec_S , eigval_S] = eig(S);
48 E_S = diag(eigval_S)
49
50 %% Reduced size: ==========================================================
51 % Extracting r columns of U:
52 U_app = U(:,1:1:r);
53
54 % Creating a similar matrix S:
55 S_app = U_app' * A * U_app;
56
57 % Finding the eigenvectors and eigenvalues of S:
58 [eigvec_S_app , eigval_S_app] = eig(S_app);
59 E_S_app = diag(eigval_S_app);
60 % Sorting the eigenvalues by absolute values:
61 [~,n] = sort(abs(E_S_app), 'descend');
62 E_S_app = E_S_app(n)
63
64 %% Reconstructing eigenvectors of S:
65 EV_S = U' * eigvec_A
66 eigvec_S
67 ERR = norm(abs(EV_S) - abs(eigvec_S))
```

The eigenvalues of a matrix $\mathbf{A}$ of size 7×7 are:

```
E_A =

  230.1355
   61.1446
   43.4887
   28.9760
   -9.5426
   -2.4386
   -1.7637
```

The eigenvalues of a matrix $\mathbf{S}$ of full size 7×7 are:

```
E_S =

  230.1355
   61.1446
   43.4887
   28.9760
   -9.5426
   -2.4386
   -1.7637
```

Next, we perform the approximations of the eigenvalues of $\mathbf{A}$ by increasing the rank r from 1 to 7.

For $r = 1$:

```
E_S_app =

   32.9295
```

For $r = 2$:

```
E_S_app =

  145.9173
   -1.5572
```

For $r = 3$:

```
E_S_app =

  145.9611
   30.7503
   -3.4328
```

For $r = 4$:

```
E_S_app =

  191.3768
   54.7387
   29.5563
   -3.4428
```

For $r = 5$:

```
E_S_app =

  231.9506
   59.2763
   29.5688
    7.8134
   -3.7689
```

For $r = 6$:

```
E_S_app =

  227.7946
   60.9257
   32.8918
   14.4950
   -9.8219
   -2.3767
```

For $r = 7$:

```
E_S_app =

  230.1355
   61.1446
   43.4887
   28.9760
   -9.5426
   -2.4386
   -1.7637
```

The eigenvectors of matrices $\mathbf{A}$ are different, however, they are linked by the following relationship:

$$\text{eigenvectors}(\mathbf{S}) = U^{-1} \, \text{eigenvectors}(\mathbf{A}) \tag{5.23}$$

This relationship is checked in the Matlab code above and produces the following results.
The eigenvectors of matrix $\mathbf{S}$:

```
EV_S =

    0.4488    0.4735   -0.1361   -0.1386    0.4951    0.6765    0.4071
   -0.6016   -0.5088   -0.0648   -0.3259   -0.0558    0.3550    0.0981
   -0.0148    0.2253   -0.2696   -0.8201    0.1092   -0.2603   -0.0056
    0.5369   -0.6507   -0.3023   -0.0148    0.2808    0.1577   -0.4903
    0.3209   -0.1834    0.3233   -0.2662   -0.4042   -0.2030    0.4880
    0.1544   -0.0895    0.6101   -0.0923    0.7003   -0.5239    0.1950
    0.1459    0.0346    0.5800   -0.3499    0.0849    0.0900   -0.5550
```

The eigenvectors of matrix **S** reconstructed as with the relation (5.23):

```
eigvec_S =

   -0.4488    0.4735   -0.1361    0.1386    0.4951    0.6765   -0.4071
    0.6016   -0.5088   -0.0648    0.3259   -0.0558    0.3550   -0.0981
    0.0148    0.2253   -0.2696    0.8201    0.1092   -0.2603    0.0056
   -0.5369   -0.6507   -0.3023    0.0148    0.2808    0.1577    0.4903
   -0.3209   -0.1834    0.3233    0.2662   -0.4042   -0.2030   -0.4880
   -0.1544   -0.0895    0.6101    0.0923    0.7003   -0.5239   -0.1950
   -0.1459    0.0346    0.5800    0.3499    0.0849    0.0900    0.5550
```

Notice, that they are the same with respect to the absolute value (some of them only differ in sign). The L^2 norm error computed from the absolute values of the above matrices is:

```
ERR =

  1.7422e-014
```

5.6 A Note on the Linear Dynamical Systems

In the linear dynamical systems, any k^{th} column of matrix $\mathbf{D}$ can be represented by the initial column $\mathbf{D}_0$ multiplied by the k^{th} power of the linear propagator matrix $\mathbf{S}$:

$$\mathbf{D}_k = \mathbf{S}^k \mathbf{D}_0 \tag{5.24}$$

Substituting the eigendecomposition of matrix $\mathbf{S}$, we get:

$$\mathbf{D}_k = \Phi \mathbf{M}^k \Phi^{-1} \mathbf{D}_0 \tag{5.25}$$

Writing the above equation as a sum we get:

$$\mathbf{D}_k = \sum_{j=1}^{r} \phi_j \mu_j^k b_j \tag{5.26}$$

Substituting for $\mu_j = e^{\omega_j}$:

$$\widetilde{\mathbf{D}}_k = \sum_{j=1}^{r} \phi_j (e^{\omega_j})^k b_j = \sum_{j=1}^{r} \phi_j e^{\omega_j k} b_j \tag{5.27}$$

Since every column of matrix $\mathbf{D}$ is linked to a particular moment in time, the integer k, can be written in terms of the time it corresponds to, divided by the timestep in our data: $k = t_i/\Delta t$. Substituting this back into (5.27) we get:

$$\widetilde{\mathbf{D}}_r = \sum_{j=1}^{r} \phi_j e^{\omega_j t/\Delta t} b_j \tag{5.28}$$

Appendix A

Pulsating Poiseuille Flow

Listing A.1: Matlab code to approximate the Poiseuille flow with three different methods. *pulsating_poiseuille_approximations.m*

```
%% Pulsating Poiseuille Flow ==============================================
% Approximating the velocity profile with eigenfunction expansion,
% POD and DMD.
% ========================================================================
clc
clear
close all

%% Initial data: ==========================================================
% USER INPUT =============================================================
TIME = 10;      % total time of the animation
W = 10;      % Womersley number
Pa = 60;     % dimensionless pressure coefficient
FONT = 10;  % fontsize for graphs
modes = 5;  % number of modes for all approximations (max 7)
dt = 0.05;  % time step
dy = 0.05;  % space step
% END OF USER INPUT ======================================================
MODES = modes;  % number of modes in eigenfunction approximation
RANK = modes;   % number of modes in POD approximation
r = modes;      % number of modes in DMD approximation
nm = modes;     % number of first modes to draw

% Definition of colours for plotting:
red = [236 68 28] ./ 255;       % used for eigenfunction
blue = [3 105 172] ./ 255;      % used for POD
green = [34 139 34] ./ 255;     % used for DMD

% Initialize error matrices:
ERROR_EIG = zeros(MODES, 1);
ERROR_POD = zeros(RANK, 1);
ERROR_DMD = zeros(r, 1);

%% Analytical result from asymptotic complex solution: ====================
t = [0:dt:TIME]; n_t = length(t);     % time discretization
y = [-1:dy:1]; n_y = length(y);       % space discretization
u_A_r = zeros(n_y, n_t);              % initialize solution
```

```
38
39 % Construct real and imaginary parts:
40 for j = 1:length(t)
41
42     Y = (1 - cosh(W*sqrt(1i).*y)./(cosh(W*sqrt(1i))))*1i*Pa/W.^2;
43     u_A_r(:,j) = real(Y.*exp(1i*t(j)));
44 end
45
46 % Adding the mean flow component:
47 u_Mb = (1 - y.^2)*0.5;                 % mean flow
48 u_M = repmat(u_Mb, length(t), 1);      % repeat solution
49 u_A_R = u_M' + u_A_r;                  % real analytical solution
50
51 %% Eigenfunction approximation: ==========================================
52 % Extended solution matrix (for plotting):
53 u_T_extend = zeros(n_y, MODES*n_t);
54
55 % Calculate the solution matrix for each mode:
56 for j = 1:1:MODES
57
58 n = [1:1:j]; % matrix of modes to include in the summation
59
60 % Initialize matrices:
61 Y = zeros(n_y, j);       % initialize spacial basis
62 A_n = zeros(j,j);        % initialize amplitude matrix
63 T = zeros(n_t, j);       % initialize temporal basis
64 U_A = zeros(n_t, n_y);   % initialize PDE solution
65
66 % Construct spatial basis:
67 for i = 1:1:length(n)
68
69     N = 2*n(i) - 1;      % odd number in the series
70     Y(:,i) = cos(N*pi*y/2);
71 end
72
73 % Construct the amplitudes:
74 for i = 1:1:length(n)
75
76     N = 2*n(i) - 1;      % odd number in the series
77     A_n(i,i) = (16*Pa) / (N*pi*sqrt((2*W)^4 + N^4*pi^4));
78 end
79
80 % Construct the temporal modes:
81 for i = 1:1:length(n)
82
83     N = 2*n(i) - 1;      % odd number in the series
84     T(:,i) = (-1)^(n(i)) * cos(t - atan((4*W^2) / (N^2*pi^2)));
85 end
86
87 % Assembly solution:
88 U_A  = Y * A_n * T';
89
90 % Adding the mean flow component:
91 u_Mb = (1 - y.^2) * 0.5;               % mean flow
```

```
92  u_M = repmat(u_Mb, length(t), 1);    % repeat solution
93  u_T = U_A + u_M';                    % eigenfunction solution
94
95  % Paste solution to the large matrix (for plotting):
96  u_T_extend(:, ((j - 1)*n_t + 1):1:(j*n_t)) = u_T;
97
98  % Compute the error of the current approximation:
99  ERROR_EIG(j) = abs(norm(u_T - u_A_R));
100 end
101
102 % Obtain elements from the diagonal:
103 sigma_A_n = diag(A_n);
104
105 %% POD approximation: ==========================================
106 % SVD of the original solution matrix:
107 [U_POD, S_POD, V_POD] = svd(u_A_R);
108
109 for j = 1:1:RANK
110
111     % Create a POD approximation:
112     U_POD_approx = U_POD(:, 1:1:j) * ...
113         S_POD(1:1:j, 1:1:j) * V_POD(:, 1:1:j)';
114     % Compute the error of the current approximation:
115     ERROR_POD(j) = abs(norm(U_POD_approx - u_A_R));
116 end
117
118 % Obtain elements from the diagonal:
119 sigma_POD = diag(S_POD);
120
121 %% DMD approximation: ==========================================
122 % Extended solution matrix (for plotting):
123 U_DMD_extend = zeros(n_y, r*n_t);
124
125 % Calculate the solution matrix for each mode:
126 for j = 1:1:r
127
128     % Define matrix D:
129     D = u_A_R;
130
131     % Construct data sets X1 and X2:
132     X1 = D(:, 1:end-1); X2 = D(:, 2:end);
133
134     % Compute the POD (SVD) of X1:
135     [U, Sigma, V] = svd(X1, 'econ');
136
137     % Approximate matrix X1 keeping only r elements of the sum:
138     U = U(:, 1:1:j);              % retain only r modes in U
139     Sigma = Sigma(1:j, 1:j);      % retain only r modes in Sigma
140     V = V(:, 1:1:j);              % retain only r modes in V
141
142     % Construct the propagator S:
143     S = U' * X2 * V * inv(Sigma);
144
145     % Compute eigenvalues and eigenvectors of the matrix S:
```

```
146     [PHI, MU] = eig(S);
147
148     % Extract frequencies from the diagonal:
149     mu = diag(MU);
150
151     % Extract real and imaginary parts of frequencies:
152     lambda_r = real(mu); lambda_i = imag(mu);
153
154     % Frequency in terms of pulsation:
155     omega = log(mu)/dt;
156
157     % Compute the DMD spatial modes:
158     Phi = U * PHI;
159
160     % Compute amplitudes with the least-squares method:
161     b2 = inv(Phi' * Phi) * Phi' * X1(:,1);
162
163     % Compute the DMD temporal modes:
164     T_modes = zeros(j, n_t);
165         for i = 1:length(t)
166
167             T_modes(:,i) = b2 .* exp(omega * t(i));
168         end
169
170     % Get the full DMD reconstruction:
171     U_DMD = real(Phi * T_modes);
172
173     % Paste solution to the large matrix (for plotting):
174     U_DMD_extend(:, ((j - 1)*n_t + 1):1:(j*n_t)) = U_DMD;
175
176     % Compute the error of the current approximation:
177     ERROR_DMD(j) = abs(norm(U_DMD - u_A_R));
178
179     %% Plot of the first modes: ==========================================
180     hfig1 = figure(1);
181     set(hfig1, 'units', 'normalized', 'outerposition', [0 0 1 1]);
182
183     % DMD spatial modes:
184     subplot(2, nm ,j);
185     plot(y, real(Phi(:,j))/norm(Phi), 'color', green, ...
186         'LineStyle', ':', 'LineWidth', 1.5);
187     [M] = AXIS(FONT);
188     set(gcf, 'color', 'w');
189     title(['\phi_{', num2str(j), '}']);
190     hold on
191
192     % DMD temporal modes:
193     subplot(2, nm, j+nm);
194     plot(t, real(T_modes(1,:))/norm(T_modes), 'color', ...
195         green, 'LineStyle', '-');
196     [M] = AXIS(FONT);
197     set(gcf, 'color', 'w');
198     title(['\psi_{', num2str(j), '}']);
199     hold on
```

```
end

%% Plot of the first modes: ========================================
hfig1 = figure(1);

% Plot for modes from eigenfunction expansion:
% Spatial:
for j = 1:1:nm

        subplot(2, nm, j);
        plot(y, Y(:,j)/norm(Y), 'color', red, 'LineStyle', '-');
        hold on
        [M] = AXIS(FONT);
        set(gcf, 'color', 'w');
        title(['\phi_{', num2str(j), '}']);
end

% Temporal:
for j = 1:1:nm

        subplot(2, nm, j+nm);
        plot(t, T(:,j)/norm(T), 'color', red, 'LineStyle', '-');
        hold on
        [M] = AXIS(FONT);
        set(gcf, 'color', 'w');
        title(['\psi_{', num2str(j), '}']);
end

% Plot for modes from POD:
% Spatial:
for j = 1:1:nm

    subplot(2, nm, j);
    plot(y, U_POD(:,j)/norm(U_POD), 'color', blue, 'LineStyle', '-')
    [M] = AXIS(FONT);
    set(gcf, 'color', 'w');
end

% Temporal:
for j = 1:1:nm

    subplot(2, nm, j+nm);
    plot(t, V_POD(:,j)/norm(V_POD), 'color', blue, 'LineStyle', '-')
    [M] = AXIS(FONT);
    set(gcf, 'color', 'w');
end

% Save the plot:
print('-dpng', '-r500', ['Modes_T', num2str(TIME), '.png'])

%% Plot for amplitude decay for both approximations: ===============
hfig2 = figure(2);
LABEL = {'Eigenfunction...', 'POD'};
MARKER = {'o', 's'};
```

```
254
255 % Get the first element to normalize the plot:
256 norm_POD = sigma_POD(1);
257 norm_EIG = sigma_A_n(1);
258
259 % Use only MODES number of terms from the diagonal:
260 Line_POD = sigma_POD(1:MODES)/norm_POD;
261 Line_A_n = sigma_A_n(1:MODES)/norm_EIG;
262
263 for j = 1:1:MODES
264
265     % Plot for eigenfunction amplitude decay:
266     plot(j, sigma_A_n(j)/norm_EIG, MARKER{1}, 'color', red);
267     [M] = AXIS(FONT);
268     set(gcf, 'color', 'w');
269     hold on
270
271     % Plot for POD amplitude decay:
272     plot(j, sigma_POD(j)/norm_POD, MARKER{2}, 'color', blue);
273     [M] = AXIS(FONT);
274     set(gcf, 'color', 'w');
275     legend(LABEL);
276     line([1:MODES], [Line_A_n], 'color', red);
277     line([1:MODES], [Line_POD], 'color', blue);
278     title(['Amplitude decay rate (normalized)']);
279     ylim([-0.1 1.1]);
280     xlim([0.9 (modes + 0.1)]);
281 end
282
283 % Save the plot:
284 print('-dpng', '-r500', 'Amplitude_decay.png')
285
286 %% Plot of the eigenvalues circle: ================================
287 hfig3 = figure(3);
288
289 % Definition of the circle in a complex plane:
290 radius = 1; z_Circle = radius * exp((0:0.1:(2*pi)) * sqrt(-1));
291
292 % Eigenvalues and complex circle:
293 stem3(lambda_r, lambda_i, real(abs(b2)), 'color', green);
294 hold on
295 plot(real(z_Circle), imag(z_Circle), 'k-')
296 [M] = AXIS(FONT);
297 set(gcf, 'color', 'w');
298 title(['Eigenvalues circle for DMD with r = ', num2str(r)]);
299
300 % Save the plot:
301 print('-dpng', '-r500', 'Eigenvalues_circle.png')
302
303 %% Movie of a pulsating velocity profile with approximations: ======
304 hfig4 = figure(4);
305 MARKER = {':', '--', '-.', '--', '--', '--', '--'};
306 LABEL = {'Original..', 'U_1', 'U_2', 'U_3'};
307
```

```
for i = 1:1:length(t)

    hold off

    % Plot for eigenfunction approximation:
    subplot(1,3,1)
    ORIGINAL = u_A_R(:,i);
    plot(y, ORIGINAL, 'k-', 'linewidth', 1);
    [M] = AXIS(FONT);
    set(gcf, 'color', 'w');
    hold on
    for j = 1:1:MODES

        EIGEN = u_T_extend(:, (n_t*(j - 1) + i));
        plot(y, EIGEN, MARKER{j}, 'color', red);
        [M] = AXIS(FONT);
        set(gcf, 'color', 'w');
    end
    title(['Eigenfunction']);
    if modes <= 3

        legend(LABEL);
    end
    ylim([-1 1.5]);
    xlim([-1 1]);
    hold off

    % Plot of the POD approximation:
    subplot(1,3,2)
    plot(y, ORIGINAL, 'k-', 'linewidth', 1);
    [M] = AXIS(FONT);
    set(gcf, 'color', 'w');
    hold on
    for j = 1:1:RANK

        U_POD_approx = U_POD(:, 1:1:j) * ...
            S_POD(1:1:j, 1:1:j) * V_POD(:, 1:1:j)';
        POD = U_POD_approx(:,i);
        plot(y, POD, MARKER{j}, 'color', blue);
        [M] = AXIS(FONT);
        set(gcf, 'color', 'w');
    end
    title(['POD']);
    if modes <= 3

        legend(LABEL);
    end
    ylim([-1 1.5]);
    xlim([-1 1]);
    hold off

    % Plot of the DMD approximation:
    subplot(1,3,3)
    plot(y, ORIGINAL, 'k-', 'linewidth', 1);
```

```
362         [M] = AXIS(FONT);
363         set(gcf, 'color', 'w');
364         hold on
365         for j = 1:1:r
366 
367             DMD_APPROX = U_DMD_extend(:, (n_t*(j-1)+i));
368             plot(y, DMD_APPROX, MARKER{j}, 'color', green);
369             [M] = AXIS(FONT);
370             set(gcf, 'color', 'w');
371         end
372         title(['DMD']);
373         if modes <= 3
374 
375             legend(LABEL);
376         end
377         ylim([-1 1.5]);
378         xlim([-1 1]);
379         hold off
380         drawnow
381 
382         % Save the gif:
383         frame = getframe(4);
384         im = frame2im(frame);
385         [imind, cm] = rgb2ind(im, 256);
386         filename = 'Pulsating_Poiseuille_Movie.gif';
387         if i == 1;
388             imwrite(imind, cm, filename, 'gif', 'Loopcount', inf);
389         else
390             imwrite(imind, cm, filename, 'gif', 'WriteMode', ...
391                 'append', 'DelayTime', 0.1);
392         end
393 end
394 
395 %% Plot of the error of each approximation: ==========================
396 hfig5 = figure(5);
397 
398 % Maximum of each error:
399 max_eig = max(ERROR_EIG);
400 max_pod = max(ERROR_POD);
401 max_dmd = max(ERROR_DMD);
402 
403 for j = 1:1:modes
404 
405     LABEL = {'Eigenfunction...', 'POD', 'DMD'};
406     plot(j, ERROR_EIG(j)/max_eig, 'color', red, 'LineStyle', 'o')
407     [M] = AXIS(FONT);
408     set(gcf, 'color', 'w');
409     hold on
410     plot(j, ERROR_POD(j)/max_pod, 'color', blue, 'LineStyle', 's')
411     [M] = AXIS(FONT);
412     set(gcf, 'color', 'w');
413     plot(j, ERROR_DMD(j)/max_dmd, 'color', green, 'LineStyle', '^')
414     [M] = AXIS(FONT);
415     set(gcf, 'color', 'w');
```

```
416         legend(LABEL);
417         line([1:modes], [ERROR_EIG]./max_eig, 'color', red);
418         line([1:modes], [ERROR_POD]./max_pod, 'color', blue);
419         line([1:modes], [ERROR_DMD]./max_dmd, 'color', green);
420         title(['Normalized error of each approximation']);
421         ylim([-0.1 1.1]);
422         xlim([0.9 (modes + 0.1)]);
423     end
424
425     % Save the plot:
426     print('-dpng', '-r500', 'Error.png')
427
428     %% Ending: ================================================================
429     close all
430     clc
```

Appendix B

1D and 2D POD Functions

Listing B.1: Matlab function for performing 1D POD. *POD_1D.m*

```
%% POD_1D function ========================================================
% POD done on a 1D data set created by:
% - matrix D
% - vector y
% ========================================================================

function [D_POD, U_POD, S_POD, V_POD] = POD_1D(D, r)
% SVD of original solution matrix:
[U_POD, S_POD, V_POD] = svd(D);

% POD Approximation:
D_POD = U_POD(:,1:1:r) * S_POD(1:1:r,1:1:r) * V_POD(:,1:1:r)';
end
```

Listing B.2: Matlab function for performing 2D POD. *POD_2D.m*

```
%% POD_2D function ========================================================
% POD done on a 2D data set created by:
% - matrix U
% - matrix V
% - vector X
% - vector Y
% ========================================================================

function [U_POD, V_POD, UU_POD, VU_POD, UV_POD, VV_POD] = ...
        POD_2D(U, V, X, Y, r)

%% Check the sizes of matrices U, V, X and Y: ============================
len_U = size(U,1);
len_V = size(V,1);
wid_U = size(U,2);
wid_V = size(V,2);
len_X = length(X);
len_Y = length(Y);

if len_U == len_V && len_U == len_X * len_Y && wid_U == wid_V
%% Initial definitions: ==================================================
% Changing NaN to zeros for the solid parts:
V(isnan(V)) = 0;
U(isnan(U)) = 0;

SPACE_X = X;                    % extracting space X-coordinates
SPACE_Y = Y;                    % extracting space Y-coordinates
n_x = length(SPACE_X);          % number of X-coordinates
n_y = length(SPACE_Y);          % number of Y-coordinates
n_t = size(V,2);                % number of time steps

%% POD approximation: ====================================================
% SVD of original solution matrix:
[UU_POD, SU_POD, VU_POD] = svd(U, 'econ');
[UV_POD, SV_POD, VV_POD] = svd(V, 'econ');

% POD Approximation:
U_POD = UU_POD(:,1:1:r) * SU_POD(1:1:r,1:1:r) * VU_POD(:,1:1:r)';
V_POD = UV_POD(:,1:1:r) * SV_POD(1:1:r,1:1:r) * VV_POD(:,1:1:r)';

% Change zeros to NaN to re-create solid parts:
U_POD(U_POD == 0) = NaN;
V_POD(V_POD == 0) = NaN;

else
    disp(['Wrong matrix dimensions.'])

    U_POD = NaN; V_POD = NaN; UU_POD = NaN; VU_POD = NaN; ...
        UV_POD = NaN; VV_POD = NaN;
end
```

Appendix C

1D and 2D DMD Functions

Listing C.1: Matlab function for performing 1D DMD. *DMD_1D.m*

```
%% DMD_1D function ========================================================
% DMD done on a 1D data set created by:
% - matrix D
% - vector y
% ========================================================================

function [lambda_r, lambda_i, D_DMD_extend, bD, Phi_extend, ...
          T_modes_extend] = DMD_1D(D, y, dt, r)

%% Check the sizes of matrices D and y: ==================================
len_D = size(D,1);
len_y = length(y);

if len_D == len_y
%% Initial definitions: ==================================================
n_t = size(D,2);          % number of time steps
n_y = size(D,1);     % number of space y-coordinates

%% Create extended output matrices to containt all ranks ================
% approximations up to r:
D_DMD_extend = zeros(n_y, r*n_t); % solution
T_modes_extend = zeros((1+r)*r/2, n_t); % temporal structures
Phi_extend = zeros(n_y, (1+r)*r/2); % spatial structures
start = 1;

%% DMD approximation: ====================================================
% Construct data sets X1 and X2:
X1 = D(:, 1:end - 1); X2 = D(:, 2:end);

for i = 1:1:r
% Compute the POD (SVD) of X1:
[UD, SigmaD, VD] = svd(X1, 'econ');

% Approximate matrix X1 keeping only r elements of the sum:
UD = UD(:, 1:1:i);              % filter: retain only r modes in U
SigmaD = SigmaD(1:1:i, 1:1:i); % filter: retain only r modes in Sigma
VD = VD(:, 1:1:i);              % filter: retain only r modes in V

```

```
% Construct the propagator S:
S = UD' * X2 * VD * inv(SigmaD);

% Compute eigenvalues and eigenvectors of the matrix S:
[PHI, MU] = eig(S); % eigenvalue decomposition of matrix S

% Extract frequencies:
mu = diag(MU);

% Extract real and imaginary parts of frequencies:
lambda_r = real(mu); lambda_i = imag(mu);

% Frequency in terms of pulsation:
omega = log(mu)/dt;      % computing natural log

% Compute DMD spatial modes:
Phi = UD * PHI;

% Compute DMD amplitudes with the least-squares method:
bD = inv(Phi' * Phi) * Phi' * X1(:,1);

% Define temporal behaviour:
TIME = [0:1:n_t - 1] * dt;

% Initialize temporal modes matrix:
T_modes = zeros(i, n_t);

% Compute DMD temporal modes:
for j = 1:length(TIME)
    T_modes(:,j) = bD .* exp(omega * TIME(j));
end

% Get the real part of the DMD reconstruction:
D_DMD = real(Phi * T_modes);

% Paste current solutions into the right place in extended matrices:
D_DMD_extend(:, ((i - 1)*n_t + 1):1:(i*n_t)) = D_DMD;
start = start + (i-1);
T_modes_extend((start:1:start+(i-1)), :) = real(T_modes);
Phi_extend(:, (start:1:start+(i-1))) = real(Phi);
end

else
    disp(['Wrong matrix dimensions.'])

    lambda_r = NaN; lambda_i = NaN; D_DMD = NaN; bD = NaN;
end
```

Listing C.2: Matlab function for performing 2D DMD. *DMD_2D.m*

```
%% DMD_2D function ========================================================
% DMD done on a 2D data set created by:
% - matrix U
% - matrix V
% - vector X
% - vector Y
% ========================================================================

function [lambdaU_r, lambdaU_i, lambdaV_r, lambdaV_i, U_DMD, ...
            V_DMD, bU, bV, Phi_U, Phi_V, T_modesU, T_modesV] ...
            = DMD_2D(U, V, X, Y, dt, r)

%% Check the sizes of matrices U, V, X and Y: ============================
len_U = size(U,1);
len_V = size(V,1);
wid_U = size(U,2);
wid_V = size(V,2);
len_X = length(X);
len_Y = length(Y);

if len_U == len_V && len_U == len_X * len_Y && wid_U == wid_V
%% Initial definitions: ==================================================
% Changing NaN to zeros for the solid parts:
V(isnan(V)) = 0;
U(isnan(U)) = 0;

SPACE_X = X;                    % extracting space X-coordinates
SPACE_Y = Y;                    % extracting space Y-coordinates
n_x = length(SPACE_X);          % number of X-coordinates
n_y = length(SPACE_Y);          % number of Y-coordinates
n_t = size(V,2);                % number of time steps

%% DMD approximation: ====================================================
% Construct data sets X1 and X2:
X1U = U(:, 1:end-1); X2U = U(:, 2:end);
X1V = V(:, 1:end-1); X2V = V(:, 2:end);

% Compute the POD (SVD) of X1:
[UU, SigmaU, VU] = svd(X1U, 'econ');
[UV, SigmaV, VV] = svd(X1V, 'econ');

% Approximate matrix X1 keeping only r elements of the sum:
UUn = UU(:, 1:1:r);             % filter: retain only r modes in U
SigmaUn = SigmaU(1:r, 1:r);     % filter: retain only r modes in Sigma
VUn = VU(:, 1:1:r);             % filter: retain only r modes in V

UVn = UV(:, 1:1:r);             % filter: retain only r modes in U
SigmaVn = SigmaV(1:r, 1:r);     % filter: retain only r modes in Sigma
VVn = VV(:, 1:1:r);             % filter: retain only r modes in V

% Construct the propagator S:
SU = UUn' * X2U * VUn * inv(SigmaUn);
SV = UVn' * X2V * VVn * inv(SigmaVn);
```

```
54
55 % Compute eigenvalues and eigenvectors of the matrix S:
56 [PHIU, MUU] = eig(SU);  % eigenvalue decomposition of matrix S
57 [PHIV, MUV] = eig(SV);  % eigenvalue decomposition of matrix S
58
59 % Extract frequencies:
60 muU = diag(MUU);
61 muV = diag(MUV);
62
63 % Extract real and imaginary parts of frequencies:
64 lambdaU_r = real(muU);     % real
65 lambdaU_i = imag(muU);     % imaginary
66 lambdaV_r = real(muV);     % real
67 lambdaV_i = imag(muV);     % imaginary
68
69 % Frequency in terms of pulsation:
70 omegaU = log(muU)/dt;      % computing natural log
71 omegaV = log(muV)/dt;      % computing natural log
72
73 % Compute DMD spatial modes:
74 Phi_U = UUn * PHIU;
75 Phi_V = UVn * PHIV;
76
77 % Compute DMD amplitudes with the least-squares method:
78 bU = inv(Phi_U' * Phi_U) * Phi_U' * X1U(:,1);
79 bV = inv(Phi_V' * Phi_V) * Phi_V' * X1V(:,1);
80
81 % Define temporal behaviour:
82 TIME = [0:1:n_t - 1] * dt;
83
84 % Initialize temporal modes matrix:
85 T_modesU = zeros(r, n_t);
86 T_modesV = zeros(r, n_t);
87
88 % Compute DMD temporal modes:
89 for i = 1:length(TIME)
90     T_modesU(:,i) = bU .* exp(omegaU * TIME(i));
91     T_modesV(:,i) = bV .* exp(omegaV * TIME(i));
92 end
93
94 % Get the real part of the DMD reconstruction:
95 U_DMD = real(Phi_U * T_modesU); % real part taken
96 V_DMD = real(Phi_V * T_modesV); % real part taken
97
98 % Change zeros to NaN to re-create solid parts:
99 U_DMD(U_DMD == 0) = NaN;
100 V_DMD(V_DMD == 0) = NaN;
101
102 else
103     disp(['Wrong matrix dimensions.'])
104
105     lambdaU_r = NaN; lambdaU_i = NaN; lambdaV_r = NaN; ...
106         lambdaV_i = NaN; U_DMD = NaN; V_DMD = NaN; bU = NaN; ...
107     bV = NaN; Phi_U = NaN; Phi_V = NaN; T_modesU = NaN; ...
```

```
108         T_modesV = NaN;
109     end
```

Appendix D

1D and 2D POD Results Plotting

Listing D.1: Matlab function for plotting the .gif file of the 1D POD approximation. *POD_1D_PLOT_GIF.m*

```
%% POD_1D_PLOT_GIF function ==============================================
% This function plots the .gif file of the approximation
% ========================================================================

function POD_1D_PLOT_GIF(D, rank_POD, U_POD, S_POD, V_POD, ...
        y, dt, Curr_Dir)
hfig1 = figure(1);
MARKER = {':', '--', '-.', '--', '--'};
LABEL = {'Original..', 'U_1', 'U_2', 'U_3', 'U_4', 'U_5'};

% Definition of time span:
n_t = size(D,2);
t = [0:1:n_t-1]*dt;

filename = (['GIF_1D_POD_r', num2str(rank_POD), '.gif']);

%% .gif of a pulsating velocity profile with approximations: ============
for i = 1:1:length(t)
    hold off

    % Plot of the original matrix D:
    plot(y, D(:, i), 'k-');
    [M] = AXIS(12);
    set(gcf, 'color', 'w');

    % Plot of the POD approximation:
    hold on
    for j = 1:1:rank_POD
        POD1D_approx = U_POD(:, 1:1:j) * ...
            S_POD(1:1:j, 1:1:j) * V_POD(:, 1:1:j)';
        plot(y, POD1D_approx(:,i), MARKER{j}, 'color', 'b');
        [M] = AXIS(12);
        set(gcf, 'color', 'w');
    end
    title(['POD approximation']);
    drawnow
    xlim([min(y)*1.1 max(y)*1.1]);
```

```
38      ylim([min(min(D))*1.1 max(max(D))*1.1]);
39
40      % Save the gif:
41      cd(Curr_Dir);
42      frame = getframe(1);
43      im = frame2im(frame);
44      [imind, cm] = rgb2ind(im, 256);
45
46      if i == 1;
47          imwrite(imind, cm, filename, 'gif', 'Loopcount', inf);
48      else
49          imwrite(imind, cm, filename, 'gif', 'WriteMode', ...
50              'append', 'DelayTime', 0.1);
51      end
52      cd ..
53  end
54
55  close(hfig1)
```

Listing D.2: Matlab function for plotting the .gif file of the 2D POD approximation. *POD_2D_PLOT_GIF.m*

```
%% POD_2D_PLOT_GIF function ==============================================
% This function plots the .gif file of the approximation
% ==========================================================================

function POD_2D_PLOT_GIF(U_POD, V_POD, rank_POD, X, Y, dt, Curr_Dir)

% Definition of time span:
n_t = size(U_POD,2);
t = [0:1:n_t-1]*dt;

SPACE_X = X;
SPACE_Y = Y;
n_X = length(SPACE_X);
n_Y = length(SPACE_Y);

filename = (['GIF_2D_POD_r', num2str(rank_POD) '.gif']);

% Limit on the time of simulation:
if length(t) <=30
    sim_t = length(t);
else
    sim_t = 30;
end

% .gif of a pulsating velocity profile with approximations: ==============
for i = 1:1:sim_t

    % Extract i-th column from U_DMD and V_DMD:
    U_extr = U_POD(:,i);
    V_extr = V_POD(:,i);

    % Prepare the velocity components for plotting:
    U_vel = zeros(n_Y, n_X);
    V_vel = zeros(n_Y, n_X);

    % Re-structuring the matrix U_vel for the purpose of plotting:
    for j = 1:1:n_X
        U_vel(:,j) = U_extr(((j-1)*n_Y+1):1:((j)*n_Y), :);
        V_vel(:,j) = V_extr(((j-1)*n_Y+1):1:((j)*n_Y), :);
    end

    VELOCITY = sqrt(U_vel.^2 + V_vel.^2);
    VELOCITY(VELOCITY == 0) = NaN;

    % Plotting the colour plot:
    hfig1 = figure(1);
    pcolor(SPACE_X, SPACE_Y, VELOCITY);
    [M] = AXIS(12);
    set(gcf, 'color', 'w');
    shading interp
    colorbar
    axis equal
```

```
53      grid off
54      daspect([1 1 1])
55      title(['POD approximation with r = ', num2str(rank_POD)]);
56      ylim([min(SPACE_Y) max(SPACE_Y)]);
57      xlim([min(SPACE_X) max(SPACE_X)]);
58      [M] = AXIS(12);
59      set(gcf, 'color', 'w');
60      drawnow
61      hold off
62
63      % Save the gif:
64      cd(Curr_Dir);
65      frame = getframe(1);
66      im = frame2im(frame);
67      [imind, cm] = rgb2ind(im, 256);
68
69      if i == 1;
70          imwrite(imind, cm, filename, 'gif', 'Loopcount', inf);
71      else
72          imwrite(imind, cm, filename, 'gif', 'WriteMode', ...
73              'append', 'DelayTime', 0.1);
74      end
75          cd ..
76  end
77
78  close(hfig1)
```

Listing D.3: Matlab function for plotting the .png file of the modes of the 1D POD approximation. *POD_1D_PLOT_MODES.m*

```
%% POD_1D_PLOT_MODES function ==========================================
% This function plots the .png file of the modes of the approximation
% ====================================================================

function POD_1D_PLOT_MODES(D, rank_POD, U_POD, S_POD, V_POD, ...
         y, dt, Curr_Dir)

% Definition of time span:
n_t = size(D,2);
t = [0:1:n_t-1]*dt;

%% Modes of the approximation: ========================================
hfig2 = figure(2);

for j = 1:1:rank_POD

    subplot(2, rank_POD, j)
    plot(y, U_POD(:,j)/norm(U_POD), 'b-')
    [M] = AXIS(12);
    set(gcf, 'color', 'w');
    title(['\Phi_{', num2str(j), '}']);

    subplot(2, rank_POD, j+rank_POD)
    plot(t, V_POD(:,j)/norm(V_POD), 'b-')
    [M] = AXIS(12);
    set(gcf, 'color', 'w');
    title(['\psi_{', num2str(j), '}']);

    drawnow

end

    cd(Curr_Dir);
    print('-dpng', '-r500', ['MODES_1D_POD_r', ...
        num2str(rank_POD), '.png'])
    cd ..

    close(hfig2)
```

Listing D.4: Matlab function for plotting the .png file of the modes of the 2D POD approximation. *POD_2D_PLOT_MODES.m*

```
%% POD_2D_PLOT_MODES function ============================================
% This function plots the .png file of the modes of the approximation
% =======================================================================

function POD_2D_PLOT_MODES(r, UU_POD, VU_POD, UV_POD, ...
        VV_POD, V, X, Y, dt, Curr_Dir)

% Definition of time span:
n_t = size(V,2);
t = [0:1:n_t-1]*dt;

% Definition of space:
SPACE_X = X;
SPACE_Y = Y;
n_Y = length(SPACE_Y);
n_X = length(SPACE_X);

%% Plotting spatial structures:
hfig1 = figure(1);
% Extract single columns representing a single spatial structure:
U_extr = UU_POD(:,r);
V_extr = UV_POD(:,r);

% Prepare the velocity components for plotting:
U_vel = zeros(n_Y, n_X);
V_vel = zeros(n_Y, n_X);

% Re-structuring the matrix U_vel for the purpose of plotting:
for j = 1:1:n_X
    U_vel(:,j) = U_extr(((j-1)*n_Y+1):1:((j)*n_Y), :);
    V_vel(:,j) = V_extr(((j-1)*n_Y+1):1:((j)*n_Y), :);
end

MODE = sqrt(U_vel.^2 + V_vel.^2);
MODE = MODE/(max(max(abs(MODE))));
MODE(MODE == 0) = NaN;

% Spatial structures:
pcolor(SPACE_X, SPACE_Y, MODE);
[M] = AXIS(12);
set(gcf, 'color', 'w');
shading interp
colorbar
axis equal
grid off
daspect([1 1 1])
title(['POD spatial mode for r = ', num2str(r)]);
ylim([min(SPACE_Y) max(SPACE_Y)]);
xlim([min(SPACE_X) max(SPACE_X)]);
[M] = AXIS(12);
set(gcf, 'color', 'w');

```

```
53 cd(Curr_Dir);
54 print('-dpng', '-r500', ['Spatial_Modes_2D_POD_r', ...
55     num2str(r), '.png'])
56 cd ..
57
58 %% Plotting temporal structures:
59 hfig2 = figure(2);
60 temp_mode = sqrt((VU_POD(:,r)).^2 + (VV_POD(:,r)).^2 );
61 temp_mode = temp_mode/(max(max(abs(temp_mode))));
62
63 plot(t, temp_mode, 'k-')
64 [M] = AXIS(12);
65 set(gcf, 'color', 'w');
66 title(['POD temporal mode for r = ', num2str(r)]);
67 ylim([min(temp_mode) max(temp_mode)]);
68 xlim([min(t) max(t)]);
69
70 cd(Curr_Dir);
71 print('-dpng', '-r500', ['Temp_Modes_2D_POD_r', ...
72     num2str(r), '.png'])
73 cd ..
74
75 close(hfig1)
76 close(hfig2)
```

Appendix E

1D and 2D DMD Results Plotting

Listing E.1: Matlab function for plotting the .gif file of the 1D DMD approximation. *DMD_1D_PLOT_GIF.m*

```
%% DMD_1D_PLOT_GIF function ==============================================
% This function plots the .gif file of the approximation
% ========================================================================

function DMD_1D_PLOT_GIF(D, rank_DMD, D_DMD_extend, y, dt, Curr_Dir)
hfig1 = figure(1);
MARKER = {':', '--', '-.', '--', '--'};
LABEL = {'Original..', 'U_1', 'U_2', 'U_3', 'U_4', 'U_5'};

% Definition of time span:
n_t = size(D,2);
t = [0:1:n_t-1]*dt;

filename = ([ 'GIF_1D_DMD_r', num2str(rank_DMD), '.gif']);

%% .gif of a pulsating velocity profile with approximations: ============
for i = 1:1:length(t)
    hold off

    % Plot of the original matrix D:
    plot(y, D(:, i), 'k-');
    [M] = AXIS(12);
    set(gcf, 'color', 'w');

    % Plot of the DMD approximation:
    hold on
    for j = 1:1:rank_DMD
        DMD_APPROX = D_DMD_extend(:, (n_t*(j-1)+i));
        plot(y, DMD_APPROX, MARKER{j}, 'color', 'r');
        [M] = AXIS(12);
        set(gcf, 'color', 'w');
    end
    title([ 'DMD approximation']);
    drawnow
    xlim([min(y)*1.1 max(y)*1.1]);
    ylim([min(min(D))*1.1 max(max(D))*1.1]);

```

```
38      % Save the gif:
39      cd(Curr_Dir);
40      frame = getframe(1);
41      im = frame2im(frame);
42      [imind, cm] = rgb2ind(im, 256);
43
44      if i == 1;
45          imwrite(imind, cm, filename, 'gif', 'Loopcount', inf);
46      else
47          imwrite(imind, cm, filename, 'gif', 'WriteMode', ...
48              'append', 'DelayTime', 0.1);
49      end
50      cd ..
51  end
52
53  close(hfig1)
```

Listing E.2: Matlab function for plotting the .gif file of the 2D DMD approximation. *DMD_2D_PLOT_GIF.m*

```
%% DMD_2D_PLOT_GIF function ==============================================
% This function plots the .gif file of the approximation
% ==========================================================================

function DMD_2D_PLOT_GIF(U_DMD, V_DMD, rank_DMD, X, Y, dt, Curr_Dir)

% Definition of time span:
n_t = size(U_DMD,2);
t = [0:1:n_t-1]*dt;

SPACE_X = X;
SPACE_Y = Y;
n_X = length(SPACE_X);
n_Y = length(SPACE_Y);

filename = ([ 'GIF_2D_DMD_r', num2str(rank_DMD), '.gif']);

% Limit on the time of simulation:
if length(t) <=30
    sim_t = length(t);
else
    sim_t = 30;
end

% .gif of a pulsating velocity profile with approximations: ==========
for i = 1:1:sim_t

    % Extract i-th column from U_DMD and V_DMD:
    U_extr = U_DMD(:,i);
    V_extr = V_DMD(:,i);

    % Prepare the velocity components for plotting:
    U_vel = zeros(n_Y, n_X);
    V_vel = zeros(n_Y, n_X);

    % Re-structuring the matrix U_vel for the purpose of plotting:
    for j = 1:1:n_X
        U_vel(:,j) = U_extr(((j-1)*n_Y+1):1:((j)*n_Y), :);
        V_vel(:,j) = V_extr(((j-1)*n_Y+1):1:((j)*n_Y), :);
    end

    VELOCITY = sqrt(U_vel.^2 + V_vel.^2);
    VELOCITY(VELOCITY == 0) = NaN;

    % Plotting the colour plot:
    hfig1 = figure(1);
    pcolor(SPACE_X, SPACE_Y, VELOCITY);
    [M] = AXIS(12);
    set(gcf, 'color', 'w');
    shading interp
    colorbar
    axis equal
```

```
53      grid off
54      daspect([1 1 1])
55      title(['DMD approximation with r = ', num2str(rank_DMD)]);
56      ylim([min(SPACE_Y) max(SPACE_Y)]);
57      xlim([min(SPACE_X) max(SPACE_X)]);
58      [M] = AXIS(12);
59      set(gcf, 'color', 'w');
60      drawnow
61      hold off
62
63      % Save the gif:
64      cd(Curr_Dir);
65      frame = getframe(1);
66      im = frame2im(frame);
67      [imind, cm] = rgb2ind(im, 256);
68
69      if i == 1;
70          imwrite(imind, cm, filename, 'gif', 'Loopcount', inf);
71      else
72          imwrite(imind, cm, filename, 'gif', 'WriteMode', ...
73              'append', 'DelayTime', 0.1);
74      end
75          cd ..
76  end
77
78  close(hfig1)
```

Listing E.3: Matlab function for plotting the .png file of the modes of the 1D DMD approximation. *DMD_1D_PLOT_MODES.m*

```
%% DMD_1D_PLOT_MODES function ============================================
% This function plots the .png file of the modes of the approximation
% =======================================================================

function DMD_1D_PLOT_MODES(D, rank_DMD, Phi_extend, ...
        T_modes_extend, y, dt, Curr_Dir)

% Definition of time span:
n_t = size(D,2);
t = [0:1:n_t-1]*dt;

%% Modes of the approximation: ==========================================
hfig2 = figure(2);
start = 1;

for j = 1:1:rank_DMD

    start = start + (j-1);

    subplot(2, rank_DMD, j)
    plot(y, Phi_extend(:, start), 'r-');
    [M] = AXIS(12);
    set(gcf, 'color', 'w');
    title(['\Phi_{', num2str(j), '}']);

    subplot(2, rank_DMD, j+rank_DMD)
    plot(t, T_modes_extend(start, :), 'r-');
    [M] = AXIS(12);
    set(gcf, 'color', 'w');
    title(['\psi_{', num2str(j), '}']);

    drawnow

end

    cd(Curr_Dir);
    print('-dpng', '-r500', ['MODES_1D_DMD_r', ...
        num2str(rank_DMD), '.png'])
    cd ..

    close(hfig2)
```

Listing E.4: Matlab function for plotting the .png file of the modes of the 2D DMD approximation. *DMD_2D_PLOT_MODES.m*

```
%% DMD_2D_PLOT_MODES function ============================================
% This function plots the .png file of the modes of the approximation
% ======================================================================

function DMD_2D_PLOT_MODES(r, Phi_U, Phi_V, T_modesU, ...
          T_modesV, V, X, Y, dt, Curr_Dir)

% Definition of time span:
n_t = size(V,2);
t = [0:1:n_t-1]*dt;

% Definition of space:
SPACE_X = X;
SPACE_Y = Y;
n_Y = length(SPACE_Y);
n_X = length(SPACE_X);

% Taking real values of the modes:
Phi_U = real(Phi_U);
Phi_V = real(Phi_V);
T_modesU = real(T_modesU);
T_modesV = real(T_modesV);

%% Plotting spatial structures:
hfig1 = figure(1);
% Extract single columns representing a single spatial structure:
U_extr = Phi_U(:,r);
V_extr = Phi_V(:,r);

% Prepare the velocity components for plotting:
U_vel = zeros(n_Y, n_X);
V_vel = zeros(n_Y, n_X);

% Re-structuring the matrix U_vel for the purpose of plotting:
for j = 1:1:n_X
    U_vel(:,j) = U_extr(((j-1)*n_Y+1):1:((j)*n_Y), :);
    V_vel(:,j) = V_extr(((j-1)*n_Y+1):1:((j)*n_Y), :);
end

MODE = sqrt(U_vel.^2 + V_vel.^2);
MODE = MODE/(max(max(abs(MODE))));
MODE(MODE == 0) = NaN;

% Spatial structures:
pcolor(SPACE_X, SPACE_Y, MODE);
[M] = AXIS(12);
set(gcf, 'color', 'w');
shading interp
colorbar
axis equal
grid off
daspect([1 1 1])
```

```
53 title([ 'DMD spatial mode for r = ', num2str(r)]);
54 ylim([min(SPACE_Y) max(SPACE_Y)]);
55 xlim([min(SPACE_X) max(SPACE_X)]);
56 [M] = AXIS(12);
57 set(gcf, 'color', 'w');
58
59 cd(Curr_Dir);
60 print('-dpng', '-r500', ['Spatial_Modes_2D_DMD_r', ...
61     num2str(r), '.png'])
62 cd ..
63
64 %% Plotting temporal structures:
65 hfig2 = figure(2);
66 temp_mode = sqrt((T_modesU(r,:)).^2 + (T_modesV(r,:)).^2 );
67 temp_mode = temp_mode/(max(max(abs(temp_mode))));
68
69 plot(t, temp_mode, 'k-')
70 [M] = AXIS(12);
71 set(gcf, 'color', 'w');
72 title([ 'DMD temporal mode for r = ', num2str(r)]);
73 ylim([min(temp_mode) max(temp_mode)]);
74 xlim([min(t) max(t)]);
75
76 cd(Curr_Dir);
77 print('-dpng', '-r500', ['Temp_Modes_2D_DMD_r', ...
78     num2str(r), '.png'])
79 cd ..
80
81 close(hfig1)
82 close(hfig2)
```

Appendix F

List of Useful Matlab Commands

For any matrix **A**:

`norm(A)`	computes a norm of a matrix **A**
`[U, S, V] = svd(A)`	performs a singular value decomposition of matrix **A**
`[U, S, V] = svd(A, 'econ')`	performs a singular value decomposition of matrix **A** and computes economy-sized matrices
`repmat(A, m, n)`	creates an $m \times n$ array of a matrix **A**
`reshape(A, [a, b])`	reshapes elements of matrix **A** into matrices of size $a \times b$, the number of elements in **A** must be equal to $a \cdot b$
`diag(A)`	extracts elements from the diagonal of matrix **A**
`size(A)`	computes the size of matrix **A**
`A'`	computes a transpose of matrix **A**

For a square matrix **A**

`[U, S] = eig(A)`	computes eigenvectors **U** and eigenvalues **S** of a matrix **A**
`inv(A)`	computes an inverse of a matrix **A**

For vectors **x** and **y**:

`norm(x)`	computes a norm of a vector **x**
`dot(x, y)`	computes an inner (dot) product of vectors **x** and **y**
`trapz(x, y)`	computes an approximation for an integral of **y** on an interval **x**
`length(x)`	computes the length of a vector **x**

References to matrix elements:

`A(i, j)`	extracts a single element from i^{th} row and j^{th} column
`A(n:m, i:j)`	extracts rows n to m for columns i to j
`A(:, i)`	extracts full i^{th} column
`A(:, n:end)`	extracts full columns from n^{th} till the last one
`A(:, n:m)`	extracts full columns from n^{th} till m^{th}
`A(i, :)`	extracts full i^{th} row
`A(n:end, :)`	extracts full rows from n^{th} till the last one
`A(n:m, :)`	extracts full rows from n^{th} till m^{th}

Appendix G

Complete List of Codes Produced

pulsating_poiseuille_approximations.m
POD_1D.m
POD_2D.m
DMD_1D.m
DMD_2D.m
POD_1D_PLOT_GIF.m
POD_2D_PLOT_GIF.m
POD_1D_PLOT_MODES.m
POD_2D_PLOT_MODES.m
DMD_1D_PLOT_GIF.m
DMD_2D_PLOT_GIF.m
DMD_1D_PLOT_MODES.m
DMD_2D_PLOT_MODES.m
disc_cont_exercise_1.m
phase_shift_of_two_sines.m
fitting_linear_systems.m
similar_matrices.m

GUI Components:

DMD_CRITERIA.m
EXPORT_DATA.m
IMPORT.m
Main_MENU.m
POD_CRITERIA.m
POD_DMD_beta_1.m
POD_OR_DMD.m

Bibliography

[1] M.A. Mendez, J.-M. Buchlin, "Notes on 2D Pulsatile Poiseuille Flows: An Introduction to Eigenfunction Expansion and Complex Variables using Matlab," VKI Technical Notes: TN 215, February 2016

[2] M.A. Mendez, M. Raiola, A. Masullo, S. Discetti, A. Ianiro, R. Theunissen, J.-M. Buchlin, "POD-based background removal for particle image velocimetry," Experimental Thermal and Fluid Science, January 2017

[3] P.J. Schmid, "Advanced Post-Processing of Experimental and Numerical Data," VKI Lecture Series 2014-01, November 2013

[4] P.J. Schmid, "Dynamic mode decomposition of numerical and experimental data," Journal of Fluid Mechanics, July 2010

[5] B.O. Koopman, "Hamiltonian Systems and Transformations in Space," Mathematics, Vol. 17, 1931

[6] C.W. Rowley, I. Mezic, S. Bagheri, P. Schlatter, D.S. Henningson, "Spectral analysis of nonlinear flows," Journal of Fluid Mechanics, September 2009

[7] I. Mezic, "Spectral properties of dynamical systems, model reduction and decompositions," Nonlinear Dynamics, June 2004

[8] M.R. Jovanovic, P.J. Schmid, J.W. Nichols, "Sparsity-promoting dynamic mode decomposition," Physics of Fluids, February 2014

[9] E.R. Scheinerman, *Invitation to Dynamical Systems*, 1996

[10] D. Dumoulin, "Numerical Characterization of an Impinging Jet Flow," VKI Stagiaire Report, September 2016

[11] http://www.mathworks.com/moler/eigs.pdf