CHARACTERIZATION OF THE PRIMARY INSTABILITY ON ATOMIZING JETS USING DYNAMIC MODE DECOMPOSITION

by

William Chirstopher Krolick

A thesis submitted in partial fulfillment
of the requirements for the degree

of

Master of Science

in

Mechanical Engineering

MONTANA STATE UNIVERSITY
Bozeman, Montana

April 2018

©COPYRIGHT

by

William Chirstopher Krolick

2018

All Rights Reserved

DEDICATION

I dedicate this work to my parents, Darrell Krolick and Carol Kosinski, who have provided the support I needed to complete this work in every way. My brother Alex Krolick has also been instrumental in giving advice and insight during this process.

ACKNOWLEDGEMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 1511325. Computational efforts were performed on the Hyalite High-Performance Computing System, operated and supported by University Information Technology Research Cyberinfrastructure at Montana State University.

TABLE OF CONTENTS

1. INTRODUCTION 1

2. THEORY 7

2.1 Reduced Order Modeling 7
2.2 Singular Value Decomposition and Proper Orthogonal Decomposition 8
2.2.1 Algorithm 2.2 Discussion 12
2.2.2 "Economy" SVD 13
2.3 Dynamic Mode Decomposition 14
2.3.1 Koopman Theory 16
2.3.2 Krylov Subspace 18
2.3.3 Arnoldi Process 18
2.3.4 Algorithm 2.3 Discussion 20
2.3.5 Numeric Properties of Dynamic Mode Decomposition 23
2.3.6 Predictive Capability of Dynamic Mode Decomposition 24
2.4 Parallelization 25

3. METHODOLOGY 27

3.1 CFD Solver and Platform 27
3.2 Extraction of Jet's Liquid Core Surface 28
3.2.1 Expansion on Key Steps of Core Height Extraction Logic 31
3.3 Parallelization 42
3.4 Applying Exact DMD Algorithm to Extracted Positional Data 44
3.4.1 Expansion of Key Steps in DMD Application Logic 46
3.5 Test Case 47

4. RESULTS 59

4.1 Simulation Parameters 59
4.2 Initial Conditions 60
4.3 Simulation 61
4.4 Dynamic Mode Decomposition of Data Extracted from Simulation 63
4.4.1 Discussion On Optimal Axial Truncation of Data 66

5. CONCLUSION 77

6. FUTURE WORK 79

REFERENCES CITED 82

TABLE OF CONTENTS – CONTINUED

APPENDICES 88

APPENDIX A : Extracted Positional Data Code
Written in Fortran and Integrated Into the NGA Platform 89
APPENDIX B : Application of Exact DMD to Extracted
Positional Data Code, Written in Matlab 106
APPENDIX C : Application of Exact DMD to a Test Case
Code
Written in Matlab 117

LIST OF FIGURES

Figure Page

1.1 DNS Simulation of an Atomizing Jet 2

2.1 The stretching and rotation of a circle by a matrix A 9

2.2 Matrix Size Representation of SVD 10

2.3 Matrix Size Representation of Economy SVD 14

2.4 Dynamic Mode Decomposition 15

2.5 Vectorization from three dimensional data to two dimensional data 19

3.1 Axial Jet Core Data Extraction Progression 29

3.2 Core Height Extraction Logic 30

3.3 Curved Jet Core Data Extraction Progression 31

3.4 Vector Math Visual 34

3.5 Cross sectional view of line marching through the grid 35

3.6 Cross sectional view of line $\boldsymbol{L}$ intersecting the interface (G is gas and L is liquid) 35

3.7 Cross sectional view of line $\boldsymbol{L}$ missing the interface (G is gas and L is liquid) 35

3.8 $\boldsymbol{L}$ propagation through a computational cell 37

3.9 Line/PLIC Intersection 39

3.10 Line/PLIC Intersection Miss 40

3.11 Overlap Solution 42

3.12 Application of DMD to Data Logic 45

3.13 Original ("high rank") data at initial timestep 49

3.14 Original ("high rank") data at initial timestep from top view 49

3.15 Original ("high rank") data at fifth timestep 50

LIST OF FIGURES – CONTINUED

Figure Page

3.16 Original ("high rank") data at fifth timestep from top view 50
3.17 Normalized Singular Values of X_1 51
3.18 Primary dynamic mode and associated time dynamics 52
3.19 Dynamic mode 2 and associated time dynamics 53
3.20 Dynamic mode 4 and associated time dynamics 53
3.21 Superimposed dynamic mode and associated time dynamics 54
3.22 "Low rank" data at fifth timestep 55
3.23 "Low rank" data at fifth timestep from top view 56
3.24 Original ("high rank") data at fifth timestep from top view 56
3.25 Real vs. Imaginary Eigenvalues 56
3.26 Real vs. Imaginary Logarithmic Eigenvalues 57
3.27 Logarithmic Imaginary Eigenvalues vs. Mode Norm 57
4.1 Diametric view of the initial shape of the jet 61
4.2 Side (X/Z plane) view of the initial shape of the jet 62
4.3 Axial (Y/Z plane) view of turbulent inflow conditions 62
4.4 First timestep from the psuedo-steady state the data was collected from 64
4.5 Full data collected from the first psuedo-steady state timestep 64
4.6 65% truncated data collected from the first psuedo-steady state timestep 65
4.7 35% truncated data collected from the first psuedo-steady state timestep 65
4.8 Energy carried by modes of 65% truncated data 69

LIST OF FIGURES – CONTINUED

Figure Page

4.9 65% truncated data's primary dynamic mode and its time dynamics 69

4.10 Full data's primary dynamic mode and its time dynamics 70

4.11 35% truncated data's primary dynamic mode and its time dynamics 70

4.12 65% truncated data's 17th dynamic mode and its time dynamics 71

4.13 Full data's 17th dynamic mode and its time dynamics 71

4.14 35% truncated data's 17th dynamic mode and its time dynamics 72

4.15 65% truncated data's 22nd dynamic mode and its time dynamics 72

4.16 Full data's 22nd dynamic mode and its time dynamics 73

4.17 35% truncated data's 22nd dynamic mode and its time dynamics 73

4.18 65% truncated data's 5th dynamic mode and its time dynamics 74

4.19 65% truncated data's superimposed dynamic mode and its time dynamics 74

4.20 Real vs. Imaginary Eigenvalues 75

4.21 Real vs. Imaginary Logarithmic Eigenvalues 76

4.22 Logarithmic Imaginary Eigenvalues vs. Mode Norm 76

LIST OF ALGORITHMS

Algorithm Page

2.1 Solving for the components of the SVD [22, 23, 31] 11

2.2 Proper Orthogonal Decomposition 12

2.3 Exact DMD Algorithm [29] 20

ABSTRACT

Numerical methods have advanced to the point that many groups can perform detailed numerical simulations of atomizing liquid jets and replicate experimental measurements. However, the simulation results have not lead to a substantial advancement to our understanding of these flows due to the massive amount of data produced. In this work, a tool is developed to extract the physics that destabilize the jets liquid core by leveraging dynamic mode decomposition (DMD). DMD takes ideas from the Arnoldi method as well as the Koopman method to evaluate a non-linear system with a low-rank linear operator. The method reduces the order of the simulation results from all the original data through time to a few key pieces of information. Most important of these are the dynamic modes, their time dynamics, and the DMD spectra. In this case, DMD is applied to the jets liquid core outer radius, which is computed at streamwise and azimuthal locations, i.e., $R(\theta, x)$. With the DMD data, we obtain the dominant spatial and temporal modes of the system and their characteristics. The dominant modes provide a useful way to collapse the large data set produced by the simulation into a length and timescale that can be used to initiate reduced-order models and numerically categorize the instabilities on the jets liquid core.

CHAPTER ONE

INTRODUCTION

The atomization of liquid jets has many engineering applications, fuel injection and cooling sprays being some of the more notable ones. Improvements in atomizing jets initially came mostly through experimentation and observation, however, the emergence of computational fluid dynamics (CFD) has proven to be a useful tool in understanding the process. Because the equations that define fluid systems have no analytic solution, numeric simulations discretize the equations and essentially brute force solutions to these systems. An array of different numeric techniques yields different levels of accuracy, and due to the recent surge in computational power, very accurately resolved results are obtained through a method of computation called direct numeric simulation (DNS). The trade off of accuracy and the time it takes to run these direct numeric simulations is a huge focus in CFD and is as important in atomizing jets as it is in any other CFD application. The problem compounds with the fact that an atomizing jet is a multiphase flow (where both liquid and gas mediums are present). This increases computational demand because of the interface between the two mediums. An example of a DNS of an atomizing jet is provided in Fig. 1.1.

So, although direct numeric simulations have reached the point that they give high fidelity results and compare to experimental data very well, they are still computationally demanding enough to not be practical for commercial use. Many research groups as well as industry still use less computationally demanding methods like Large Eddy Simulations or Reynolds Averaged Navier Stokes simulations. These

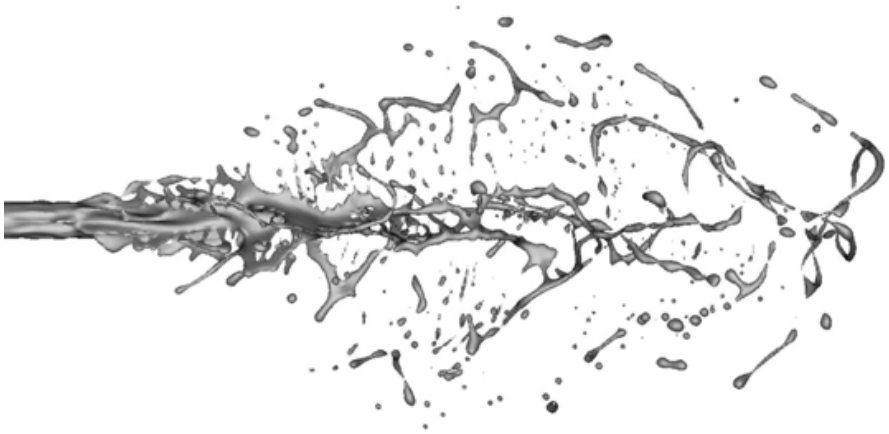


Figure 1.1: DNS Simulation of an Atomizing Jet

methods use representations of droplet breakup on the sub-grid scale that are not directly governed by computationally expensive equations that define direct numeric simulations. They use methods like injecting spheres of different sizes and manipulating them to then represent jet breakup. Blob injection, frequency induction, and sheet breakup [42] are some of the ways this is accomplished. There are many other more advanced methodologies that exist as well. The Eulerian-Lagrangian Spray and Atomization (ELSA) model [54] switches between Eularian and Lagrangian approaches at some downstream location when the spray becomes dilute [40]. The Unified Spray Breakup (USB) model [11] uses experimental observations to identify three distinct atomizing jet sub-processes to be analyzed separately [40]. The Enhanced Taylor analogy breakup (ETAB) [17] model describes the oscillating and distorting droplet as a spring-mass system [40]. These methods are all either approximations of atomization from observations and require tuning to match the physical system or simply do not hold up for every situation and limit accuracy and adaptability.

Another issue with direct numeric simulations is the significant amount of data

produced. For many simulations this number is in the 100s of terabytes. For these reasons, it has become desirable to develop tools that can be used to 1) produce more accurate lower order models and 2) gain a better understanding of the atomization process that would not otherwise be attainable due to the amount of data being visualized. The natural path to follow in the accomplishment of these goals is to find a way to numerically categorize the instabilities in an atomizing jets liquid core. Numerically extracting these instabilities could be very useful in the development of a new type of jet breakup model for lower order simulations.

Reduced order modeling (ROM) is a relatively new subject of research that has been introduced to accomplish a very similar goal to those mentioned above. The large amount of data being produced around the world has introduced the question of how to deal with it all. Several relevant topics have come out of researching ROMs that are applicable to CFD and, more specifically, atomizing jets. Those topics are proper orthogonal decomposition (POD) and dynamic mode decomposition (DMD). POD uses singular value decomposition to extract dominant spatial modes in a system, and DMD furthers this idea to include how those dominant modes act through time. POD and DMD are closely related [29,47] and they have been applied to similar CFD applications [5, 28, 46, 48–50]. There have been other studies as well using different techniques to investigate the atomizing process [8, 18, 26]. The traditional use of the ROM methods is to take a complicated non-linear system (including most fluid systems), and replicate the system using lower rank or lower order mathematics. This produces a significantly less computationally demanding simulation. There is currently much research being done on the subject and how to use these methods for practical purposes. As discussed in Chapter 6, there are many potential uses, but the goal of this work was to develop a tool using the methods already established to directly assist in solving the issues discussed above. Because DMD handles the

analysis of transient data sets much better than POD (Section 2.3), DMD was used in this work. From the several different iterations of DMD available, the specific method called Exact DMD was chosen for reasons discussed in Section 2.3.4.

Before the underlying physics of the system could be extracted, it was first necessary to define the system. An established computational platform was used to perform simulations of atomizing jets. A new tool was added to the code to extract the desirable observable (data set) [32]. DMD capability was also added to the code to perform the DMD on the data obtained with the new tool. Finding the observable that best represents the information required is important. If the wrong observable is chosen the results could be meaningless, or just as difficult to establish useful information about the jet breakup as the original simulation. Too much data to in an observable is a difficult problem in a high resolution three-dimensional simulation. For that reason, along with the focus being on the jets core, it was desirable that only the surface of the jets liquid core be used in the DMD analysis. Discussed in Section 3.2, a methodology was developed to represent the jets liquid core in a form of cylindrical coordinates. As mentioned above, several studies have been done on the DMD of atomizing jets. Where this work differs is the observable data the DMD was applied to. Most other studies have either used two dimensional simulations and experiments, velocity, or vorticity as observable data sets. This work uses a three dimensional simulation and positional data for its observables. The distinction is relevant because if a new sub-grid breakup model were to be implemented using information obtained from DMD, it would be important to establish a relationship between the dynamic modes of multiple variables of interest. Information gathered from the DMD of positional data could also provide valuable information about droplet behavior when implementing a sub-grid breakup model. If distinct oscillatory modes are found that contribute to jet breakup, their wavelengths and frequency could

be used to define droplet size and rate of droplet breakup, respectively. This would create a sub-grid breakup model based on information obtained directly from a DNS.

Before applying DMD to a DNS of an atomizing jet, a test scenario was developed to establish proof of concept and to determine if DMD would produce results that could be used in the ways that were desired. It was important to do this before integrating the decomposition into a large and complex code that could require days to run a simulation, especially if the simulation was only needed for testing and debugging. Once the DMD code was written it could be straightforwardly implemented into the DNS code already written. The test case is discussed in Section 3.5.

Although the surface representation code was not computationally expensive, in order to integrate it into the established CFD code it still had to be parallelized. This was necessary to avoid holding up the rest of the simulation. Parallelization in a DNS is extremely valuable. The ability to break up parts of a simulation into multiple pieces reduces the run time of the simulation by a factor of almost the number of processors being used. A simulation that could take a week to run on one processor could be reduced to a matter of hours or even minutes. As discussed in Section 3.3, the jets liquid core surface representation code was parallelized in all three dimensions. The DMD code was not parallelized, but due to the possibility of matrices of a significant size being created by the DMD it was necessary to use a method to consolidate the matrix size. The method described in Section 2.2.2 was used to do this and complete the DMD and postprocessing on one processor. This could be accomplished even when performing DMD on large DNS data sets.

A DMD analysis on the jets liquid core surface data yielded promising results. Distinct modes were effectively extracted and separated into time-varying and non-time-varying modes, and it could be clearly seen and reasoned which modes

contributed to the breakup of the jet. These results (assuming the continuation of the research) could aid in the creation of a new type of sub-grid jet breakup model for lower order simulations that does not rely on assumptions or tuning, but rather the information directly obtained from DNS simulations.

CHAPTER TWO

THEORY

This section gives background information on the methods used, the math associated with them, and presents the information that went into the decisions that were made in the choosing of those methods. It should be noted that the focus and purpose of the research being presented is not the actual development of reduced order modeling methods and techniques, but the use and adaptation of them to develop tools that will be helpful in the characterization of the destabilization of a jet and improving lower order models.

2.1 Reduced Order Modeling

Through an improved understanding of the physical systems that shape the way things work in the world along with the advancement of data collection schemes comes the increase in the amount of data produced. This has led to the development of methods that interpret trends in large amounts of data. Reduced order modeling has been developed through the idea that in a nonlinear dynamical system (e.g. a fluid system), there exists some low dimensional dynamics that drive the system and can be identified and extracted. There are two ways to accomplish this. One is to reduce the order of the equations being used to model the system. This method is the root of all reduced order modeling, used in methods like the Arnoldi process [[31,47]] and the Koopman theory [29,33,55]. The ideas from these methods along with the desire to perform ROM on experimentally collected data sets led to the second type of ROM, data driven ROM. Using similar ideas to Arnoldi/Koopman and combining them with data driven techniques like singular value decomposition,

methods like dynamic mode decomposition were developed. Data driven ROM can also be used for numeric simulations, as it can be desirable to not change the governing equations driving the original system and get accurate initial data sets to analyze. To extract the data necessary for data driven ROM, there must first be some snapshots of the system captured through time. These are the observables discussed in section 2.3.1. The singular value decomposition of these observables is what most data driven reduce order models are based from. This is because of the ability to identify important spatial correlations in data using the concept of orthogonality and eigenvalue decomposition. The result of this reduced order model is that computational cost is driven down significantly, and the dynamics of the system are still captured efficiently. The discussion in the following sections will start with the basis of data driven ROM, the SVD, and then move on into DMD.

2.2 Singular Value Decomposition and Proper Orthogonal Decomposition

Taking the singular value decomposition of a matrix can be represented as the warping of a circle or sphere. The result of taking a matrix and multiplying it against a data set of a unit circle/sphere would be to elongate the circle in some direction, causing the formation of an ellipse. The two components of the multiplication are the elongation of the circle to an ellipse, and the rotation of the ellipse. Any (m, n) matrix would act upon a circle in this way, and therefore any matrix can be broken down into its corresponding components. This idea is very similar to that of an eigenvalue decomposition, where a matrix is essentially being represented in a different space, one that is easier to describe and see what the matrix properties are. The SVD is capturing very similar information, but with two different spaces. Doing this is useful because it allows a type of eigenvalue matrix (the $\boldsymbol{S}$ matrix in equation 2.2) to be captured from any matrix of any size [31, 34]. The idea

is to do instead of an eigenvalue decomposition of the matrix itself, an eigenvalue decomposition of the transpose of the matrix multiplied by itself for one direction and the same thing but reversed in the other direction. This process is described in Algorithm 2.1. Conceptually, depending on the type of data being analyzed, one of these decompositions could represent the spatial components of the system while the other would represent the temporal components [34]. Both relate to the same $\boldsymbol{S}$ matrix. As described in section 2.3, SVD is used to extrapolate time dynamics, but they are not immediately obtained in just performing SVD on a matrix of vectorized time shifted data [48, 49]. So, performing the SVD on a matrix extracts three unique matrices, one for elongation, one for a primary rotation that represents the elongation information in a more manageable space, and one secondary shift that accounts for the non-typical eigenvalue decomposition that was performed to get the information desired. A visual representation of this is shown in Fig. 2.1.

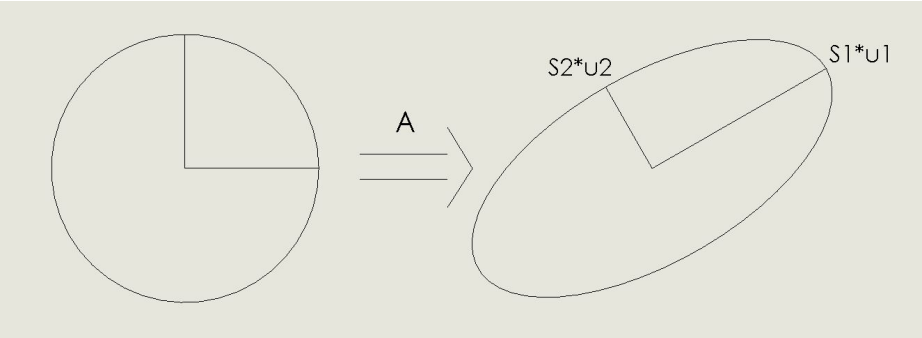


Figure 2.1: The stretching and rotation of a circle by a matrix A

We start with $\boldsymbol{A}$ acting on $\boldsymbol{V}$ such that

$$\boldsymbol{A} * \boldsymbol{V} = \boldsymbol{U} * \boldsymbol{S}. \tag{2.1}$$

Almost in eigenvalue decomposition form, the SVD of matrix $\boldsymbol{A}$ now results in

$$\boldsymbol{A} = \boldsymbol{U} * \boldsymbol{S} * \boldsymbol{V}^*, \tag{2.2}$$

where $\boldsymbol{V}^*$ is the complex conjugate transpose of $\boldsymbol{V}$. Concerning real values, $\boldsymbol{V}^* = \boldsymbol{V}'$, $\boldsymbol{V}'$ being the transpose of $\boldsymbol{V}$.

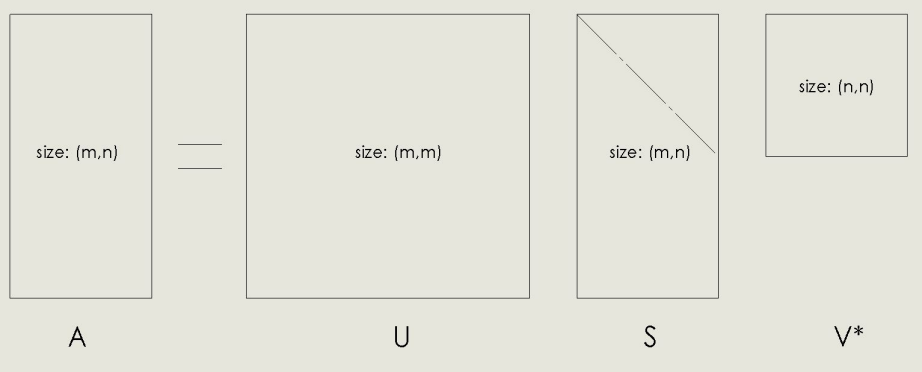


Figure 2.2: Matrix Size Representation of SVD

The columns of the $\boldsymbol{U}$ matrix are known as the left singular vectors, or the principal component vectors. This matrix represents the primary directions of the first shift of basis (or the first rotation) and is a unitary matrix. This means that in real space each column vector in the principal component matrix is orthogonal to the others. Each vector also corresponds to the singular value in the same position in the $\boldsymbol{S}$ matrix. This means that the $\boldsymbol{S}$ matrix represents how much data is projected in each principal direction. The $\boldsymbol{U}$ matrix is represented as u_1 and u_2 in Fig. 2.1, and Fig. 2.2 shows its size of (m, m). The diagonal values of the $\boldsymbol{S}$ matrix are the singular values, these account for the stretch function of the matrix $\boldsymbol{A}$, or the energy in each mode in descending magnitude. In Fig. 2.1 these values are represented as $S1$ and $S2$. As seen in Fig. 2.2, the $\boldsymbol{S}$ matrix is of size (m, n). The $\boldsymbol{V}^*$ matrix, known as the right singular vectors, serves the function of the second shift of basis used to relate

the original shift back to the initial space. It is also unitary. As seen in Fig. 2.2, $\boldsymbol{V}^*$ is of size (n, n). Further reading is available in [23, 31].

Algorithm 2.1: Solving for the components of the SVD [22, 23, 31]

1: $\boldsymbol{A} * \boldsymbol{V} = \boldsymbol{U} * \boldsymbol{S}$, $\boldsymbol{A}$ being the input matrix

2: $\boldsymbol{A} = \boldsymbol{U} * \boldsymbol{S} * \boldsymbol{V}^{-1}$

3: $\boldsymbol{V}$ is a unitary matrix, so $\boldsymbol{V}^{-1} = \boldsymbol{V}^*$, and now $\boldsymbol{A} = \boldsymbol{U} * \boldsymbol{S} * \boldsymbol{V}^*$

a) $\boldsymbol{A} * \boldsymbol{A}^T = \boldsymbol{U} * \boldsymbol{S} * \boldsymbol{V}^* * \boldsymbol{V} * \boldsymbol{S} * \boldsymbol{U}^*$

1) V is unitary so $\boldsymbol{A} * \boldsymbol{A}^T = \boldsymbol{U} * \boldsymbol{S}^2 * \boldsymbol{U}^*$

2) $\boldsymbol{A} * \boldsymbol{A}^T * \boldsymbol{U} = \boldsymbol{U} * \boldsymbol{S}^2 * \boldsymbol{U}^* * \boldsymbol{U} = \boldsymbol{U} * \boldsymbol{S}^2$ ($\boldsymbol{U}$ is unitary so $\boldsymbol{U}^* * \boldsymbol{U} = [\boldsymbol{I}]$)

a) 3-a-2 is an eigenvalue problem of the form $\boldsymbol{M} * \boldsymbol{w} = \boldsymbol{w} * \boldsymbol{\lambda}$ where $\boldsymbol{M} = \boldsymbol{A} * \boldsymbol{A}^T$, $\boldsymbol{w} = \boldsymbol{U}$, and $\boldsymbol{\lambda} = \boldsymbol{S}^2$. So $\boldsymbol{U}$ and $\boldsymbol{S}$ and can be found from the eigenvalue decomposition of $\boldsymbol{A} * \boldsymbol{A}^T$

b) $\boldsymbol{A}^T * \boldsymbol{A} = \boldsymbol{V} * \boldsymbol{S} * \boldsymbol{U}^* * \boldsymbol{U} * \boldsymbol{S} * \boldsymbol{V}^*$

1) U is unitary so $\boldsymbol{A}^T * \boldsymbol{A} = \boldsymbol{V} * \boldsymbol{S}^2 * \boldsymbol{V}^*$

2) $\boldsymbol{A}^T * \boldsymbol{A} * \boldsymbol{V} = \boldsymbol{V} * \boldsymbol{S}^2 * \boldsymbol{V}^* * \boldsymbol{V} = \boldsymbol{V} * \boldsymbol{S}^2$ ($\boldsymbol{V}$ is unitary so $\boldsymbol{V}^* * \boldsymbol{V} = [\boldsymbol{I}]$)

a) 3-b-2 is an eigenvalue problem of the form $\boldsymbol{M} * \boldsymbol{w} = \boldsymbol{w} * \boldsymbol{\lambda}$ where $\boldsymbol{M} = \boldsymbol{A}^T * \boldsymbol{A}$, $\boldsymbol{w} = \boldsymbol{V}$, and $\boldsymbol{\lambda} = \boldsymbol{S}^2$. So $\boldsymbol{V}$ can be found from the eigenvalue decomposition of $\boldsymbol{A}^T * \boldsymbol{A}$ ($\boldsymbol{S}$ is already known)

An important concept to note is that the SVD can also be used to categorize the pseudo-inverse of a matrix [31]. So from $\boldsymbol{A} = \boldsymbol{U} * \boldsymbol{S} * \boldsymbol{V}^*$, we can then also get $\boldsymbol{A}^+ = \boldsymbol{V} * \boldsymbol{S}^{-1} * \boldsymbol{U}^*$, $\boldsymbol{A}^+$ being the pseudo-inverse of $\boldsymbol{A}$. This idea is applied in the DMD algorithm. It should also be noted that the combined matrices resulting from the SVD is a form of a least square fit to the original matrix [23].

Proper Orthogonal Decomposition is not particularly different than SVD, indeed

in some cases there are referred to as synonyms for each other [21]. The difference is in the representation of data. To find the proper orthogonal modes extracted from a POD (or SVD), several extra steps must be performed. These are detailed below in Algorithm 2.2.

Algorithm 2.2: Proper Orthogonal Decomposition

1: Take the SVD of the matrix

2: Truncate the matrices $\boldsymbol{U}$ and $\boldsymbol{S}$ to $\hat{\boldsymbol{U}}$ and $\hat{\boldsymbol{S}}$ where $\hat{\boldsymbol{U}} = \boldsymbol{U}(:, 1:r)$ and $\hat{\boldsymbol{S}} = \boldsymbol{S}(1:r, :)$

3: Create a low order representation of the data using the truncated matrices so that $\hat{\boldsymbol{A}} = \hat{\boldsymbol{U}} * \hat{\boldsymbol{S}} * \boldsymbol{V}^*$ where $\hat{\boldsymbol{A}}$ is an approximation of $\boldsymbol{A}$.

4: Extract individual spatial modes: $\boldsymbol{M}_n = \hat{\boldsymbol{U}}(:, n) * \hat{\boldsymbol{S}}(n, :) * \boldsymbol{V}(:, :)$

2.2.1 Algorithm 2.2 Discussion

It can be seen from this algorithm that the proper orthogonal modes are simply the identified important principal component vectors represented on the same scale as the original data. They are the dominant spatial modes of a static set of data. The truncation mentioned in step 2 is important and should be expanded on.

The method for truncation is as follows: Set a cutoff percentage of the original datas energy that would be acceptable to allow into reduced order model allows. Initially this number is arbitrary, but once results can be seen, as in Figure 3.17, it is easier to identify the modes with the greatest energy, and truncate the rest. Truncation is done by summing a scaled-to-percentages version of the $\boldsymbol{S}$ matrix and keeping track of the total percentage. Once the total percentage is reached the rows of the $\boldsymbol{S}$ matrix after that cutoff is reached

are truncated. Then all corresponding matrices are adjusted accordingly. The rank number of remaining rows in $\boldsymbol{S}$ will be referenced as r. Looking at the math that will be used to create a low order recreation of the data, it can be seen that in addition to truncating $\boldsymbol{S}$ from (m, n) to (m, r), it is also necessary to truncate $\boldsymbol{U}$ from (m, m) to (m, r). $\boldsymbol{V}^*$ can be left unaltered as this will give a recreation of low order data of size (m, n).

Note: In dynamic mode decomposition, the truncation is slightly different for the calculation of $\boldsymbol{A}$ (Algorithm 2.3). It requires $\boldsymbol{U}$ to be truncated from (m, m) to (m, r), $\boldsymbol{S}$ to be truncated from (m, n) to (r, r), and $\boldsymbol{V}^*$ to be truncated from (n, n) to (n, r).

2.2.2 "Economy" SVD

It should be mentioned that SVD has the potential to take up a large amount of memory allocation to store the matrices. As mentioned in section 2.3, part of the DMD algorithm that was used in this research is to vectorize each time step and accumulate it into one large 2D matrix, then perform the SVD of that matrix. If the original data from one timestep was of size (p, n), the $\boldsymbol{U}$ matrix produced by the taking the SVD of the vectorized data would be of size $(p \times n, p \times n)$. In order to represent the liquid core with any amount of accuracy it was entirely possible, even probable, that the $\boldsymbol{U}$ matrix would allocate too much memory for one processor to handle. This situation could be handled by parallelizing the SVD step of the DMD algorithm, but that process is complicated and non-trivial. Shown in Fig. 2.2, the size of the $\boldsymbol{S}$ matrix produced by the SVD is the same size as the original matrix. Given that the type of data set produced by vectorizing the timesteps has many more rows than columns and that the $\boldsymbol{S}$ matrix is a diagonal matrix, the rows below the value of the number of columns

in the $\boldsymbol{S}$ matrix are all 0. This means that we know that most of the columns of the $\boldsymbol{U}$ matrix are going to be 0, and we can limit the initial allocation of the $\boldsymbol{U}$ matrix to a size of $(p \times n, n)$. Shown in Fig. 2.3 (where $m = p \times n$), this allows the entire process to be done on one processor and is what is used in the SVD step of the Exact DMD method provided in Algorithm 2.3 described in Section 2.3.

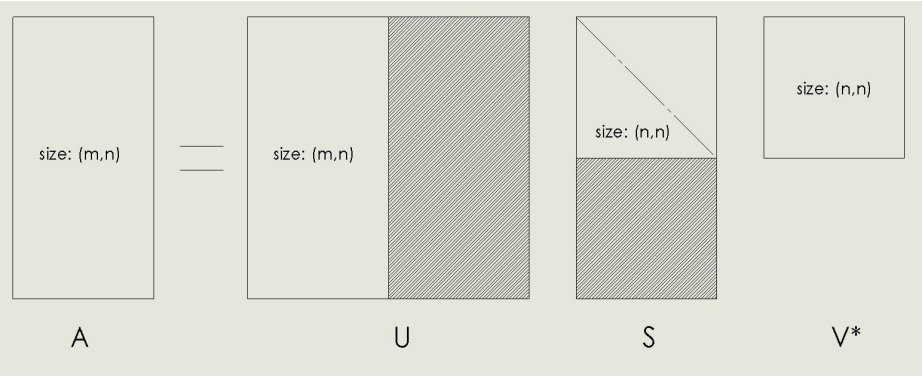


Figure 2.3: Matrix Size Representation of Economy SVD

2.3 Dynamic Mode Decomposition

The goal of DMD is to evaluate a nonlinear system through time with a linear operator. This is achieved through a type of least squares fit. The useful pieces of information are in the eigenvalues and eigenvectors of the linear operator. A manipulation of those eigenvectors and eigenvalues describes the dynamic modes and their associated time dynamics. Each mode is independent of time and is the same size as one timestep in the system (Fig. 2.4). This method breaks up the underlying fluid behavior into a convenient representation of the physics of the system. A low order recreation of the system also becomes available. A clear example of this can be seen in the test case provided in Section 3.5.

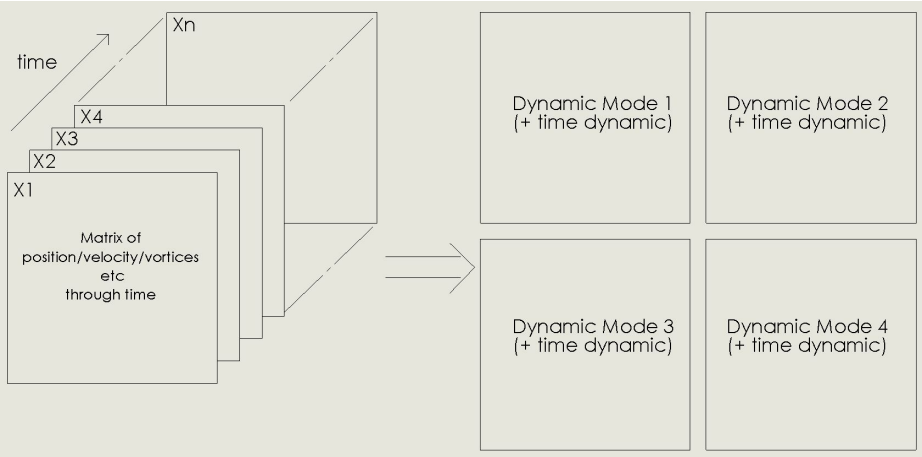


Figure 2.4: Dynamic Mode Decomposition

Although POD does a good job of extracting spatial modes from a single data set, it becomes clear when attempting to apply it to a transient data set that it struggles to identify modes through time. POD is spatially orthogonal, not temporally orthogonal [48, 49]. The extraction of time dynamics is not immediately available, even when each time step is vectorized into one large matrix for each row to be compared to the next. Further development is necessary to get temporal correlation of a data set. Step 7 in algorithm 2.3 shows that DMD clearly obtains more temporal information than just POD. A code could be written to save data at every time step (or however often is decided to be necessary) and then do a POD analysis on each individual timestep. This would be helpful, but still limited since there would still be nothing to quantify what the modes are doing through time. This issue leads to a question of time dynamics, and directly to DMD. Explained in detail below, DMD extracts several things: 1) A low rank recreation of the data with the ability to predict how the data will react past the experiment or simulation run time; 2) a linear operator that evaluates the non-linear system which will be called $\boldsymbol{A}$ (Equation. 2.7); 3) the dynamic modes (scaled eigenvectors of A that are non-time dependent modes

representing what the data is doing); 4) the temporal dynamics of those modes (a manipulation of the eigenvalues of A through time); and 5) the DMD spectra (the eigenvalues of A, representing the stability of the low order model). Below is a description of the Koopman theory, Krylov subspaces, the Arnoldi method, and later how they are used in DMD. It should be noted that the original development of DMD in [47] was done independently of Koopman theory (although the connection was noted). DMD was later expanded in [43] to emphasize the connection to the Koopman operator.

2.3.1 Koopman Theory

The general idea behind the Koopman operator is that there is some representative linear operator $\mathcal{A}$ that acts on a system of scalar observables to move the system from one time step to the next. In the case of a direct numeric simulation of fluid flow, the observables will simply be the data output by the simulation (the location of the surface of the jet's liquid core, in our case specifically). The equation representation of this is

$$\mathcal{A} * \boldsymbol{g}(\boldsymbol{x}) = \boldsymbol{g}(\boldsymbol{N}(\boldsymbol{x})), \tag{2.3}$$

where $\boldsymbol{N}(\boldsymbol{x}) = \frac{\boldsymbol{dx}}{\boldsymbol{dt}}$ and is a finite dimensional nonlinear differential equation, and $\boldsymbol{g}(\boldsymbol{x})$ are the observables. $\frac{\boldsymbol{dx}}{\boldsymbol{dt}}$ could then be represented as an approximation, $\frac{\hat{\boldsymbol{dx}}}{\boldsymbol{dt}}$ with

$$\frac{\hat{\boldsymbol{dx}}}{\boldsymbol{dt}} = \mathcal{A} * \boldsymbol{x}. \tag{2.4}$$

Substituting $\frac{\hat{\boldsymbol{dx}}}{\boldsymbol{dt}}$ for $\boldsymbol{N}(\boldsymbol{x})$ leads to

$$\mathcal{A} * \boldsymbol{g}(\boldsymbol{x}) = \boldsymbol{g}(\mathcal{A} * \boldsymbol{x}). \tag{2.5}$$

This makes the non-linear system $\boldsymbol{N}(\boldsymbol{x})$ into a linear system $\mathcal{A} * \boldsymbol{x}$ (recall that $\mathcal{A}$ is a linear operator). This has an exact solution. One way to solve this is an eigenfunction expansion [41], expressed as

$$\hat{\boldsymbol{x}}(t) = \sum_k \boldsymbol{b}_k * \psi_k * e^{\boldsymbol{w}_k * t} \tag{2.6}$$

where ψ is the eigenvectors of $\mathcal{A}$, $\boldsymbol{w}$ is log(eigenvalues of $\mathcal{A}$), and $\boldsymbol{b}$ is a coefficient representing the initial timestep of the data $\boldsymbol{X}$ in the eigenvector basis [29, 47]. Further reading on the Koopman theory, the math behind the Koopman operator, and its relationship to DMD are available in [7, 27, 29, 32, 47, 50, 55].

The eigenfunction solution of the Koopman linear operator is with respect to the continuous assumed linear dynamics $\mathcal{A}$ (from equation 2.4) of the system [24]. Now, if the system is defined as an analogous discrete time system sampled every dt so that

$$\hat{\boldsymbol{X}}_{n+1} = \boldsymbol{A} * \boldsymbol{X}_n, \tag{2.7}$$

where $\boldsymbol{A} = e^{\mathcal{A}dt}$ [24], the solution to this system is

$$\hat{\boldsymbol{X}}_n = \boldsymbol{\psi} * \boldsymbol{\lambda}^n * \boldsymbol{b}_o. \tag{2.8}$$

$\boldsymbol{\psi}, \boldsymbol{\lambda}$ are the eigenvectors and eigenvalues of $\boldsymbol{A}$, respectively. $\boldsymbol{b}_o$ is again a coefficient representing the initial timestep of the system $\boldsymbol{X}$, $\boldsymbol{X}_o$ so that

$$\boldsymbol{X}_o = \boldsymbol{\psi} * \boldsymbol{b}_o. \tag{2.9}$$

The goal is to find the operator $\boldsymbol{A}$ that will result in $||\boldsymbol{x}(t) – \hat{\boldsymbol{x}}(t)|| << 1$. Algorithm 2.3 shows and describes that arranging the system as in equation 2.7 insures this [24].

2.3.2 Krylov Subspace

A Krylov subspace is an arrangement of terms that can be used to find a representative matrix $\hat{\boldsymbol{X}}$ from the original matrix $\boldsymbol{X}$ using the initial timestep of $\boldsymbol{X}$ ($\boldsymbol{X}_o$) and an operator $\boldsymbol{A}$ that relates one timestep of the (most likely) non-linear system to the next. The method is typically used for iteratively solving linear equations or differential equations [6, 16, 44, 45], but recent adaptations have expanded its use for non-linear PDE's [19]. This means that the operator $\boldsymbol{A}$ seen in equation 2.10 must be linear. It also means that since the system in question is most likely non-linear, the operator $\boldsymbol{A}$ will be an estimate. This comes in a form of a least squares fit. Traditionally a Krylov subspace iteration represents the convergence to the largest eigenvector of the matrix $\boldsymbol{A}$. Because of the assumption of a constant mapping between snapshots, the entire sequence of data may be represented as a Krylov sequence [4,31,47]. This can be written as

$$\hat{\boldsymbol{X}} = [\boldsymbol{X}_o, \boldsymbol{X}_o * \boldsymbol{A}, \boldsymbol{X}_o * \boldsymbol{A}^2, \boldsymbol{X}_o * \boldsymbol{A}^3, ..., \boldsymbol{X}_o * \boldsymbol{A}^n], \tag{2.10}$$

which is derived from the assumption that $\hat{\boldsymbol{X}}_{n+1} = \boldsymbol{A} * \boldsymbol{X}_n$ (equation 2.7). The final solution to the reduced order system shown in equation 2.11 is essentially equation 2.8 expanded into a Krylov subspace form. The calculation of $\boldsymbol{A}$ using properties that ensure the least squares fit of the system will be described in step 7 of Algorithm 2.3.

2.3.3 Arnoldi Process

The Arnoldi process is an iteration process that constructs a Krylov subspace [6, 31, 44]. It can be used to produce the system $\hat{\boldsymbol{X}}$ by performing the QR decomposition (similar in principal to the SVD) of the matrix so that

$\hat{\boldsymbol{X}}_n = \boldsymbol{Q} * \boldsymbol{R}$ where $\boldsymbol{H} = \boldsymbol{R} * \boldsymbol{S} * \boldsymbol{R}^{-1}$ ($\boldsymbol{S}$ is closely related to $\boldsymbol{A}$) and $\boldsymbol{A} * \boldsymbol{Q} = \boldsymbol{Q} * \boldsymbol{H}$. The issue with this method is that to solve for the system $\boldsymbol{A}$ must be known [47]. In this work the problem was tackled from the opposite side, attempting to find $\boldsymbol{A}$ and its associated system properties from data.

Calculating $\boldsymbol{A}$ is the key to dynamic mode decomposition. If the original high order data is known, then from the SVD of that data, certain key properties are obtained, specifically the orthogonal principal components $\boldsymbol{U}$, the linear operator $\boldsymbol{A}$, and its properties. The algorithm for calculating $\boldsymbol{A}$ (and as a result $\boldsymbol{\psi}$ (dynamic modes) and $\boldsymbol{\lambda}$ (eigenvalues of $\boldsymbol{A}$)) is described in Algorithm 2.3, but the resulting equation that is derived and used to calculate individual low rank timesteps $\hat{\boldsymbol{X}}_n$ is simply Equation 2.8 expanded in the form of Equation 2.10.

$$\hat{\boldsymbol{X}} = [\boldsymbol{\psi} * \boldsymbol{\lambda}^0 * \boldsymbol{b}_o, \boldsymbol{\psi} * \boldsymbol{\lambda}^1 * \boldsymbol{b}_o, \boldsymbol{\psi} * \boldsymbol{\lambda}^2 * \boldsymbol{b}_o, ..., \boldsymbol{\psi} * \boldsymbol{\lambda}^n * \boldsymbol{b}_o] \tag{2.11}$$

The method described by Algorithm 2.3 to derive $\boldsymbol{A}$, its properties, and equation 2.11 is called Exact DMD (Kutz et. al [29]) and was chosen for reasons outlined in section 2.3.4.

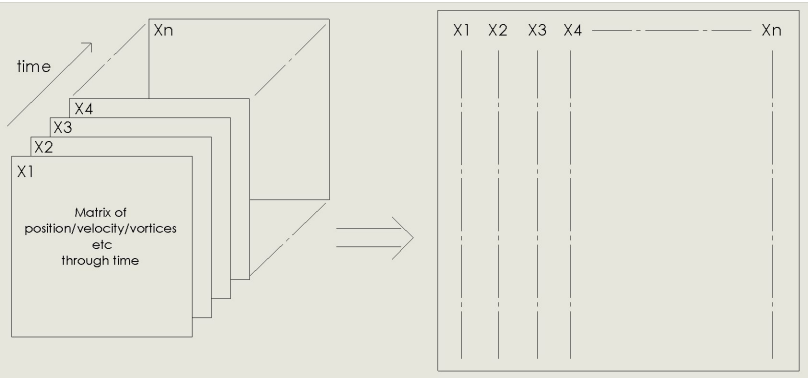


Figure 2.5: Vectorization from three dimensional data to two dimensional data

Algorithm 2.3: Exact DMD Algorithm [29]

1: Vectorize each data set and place into one large matrix $\boldsymbol{X}$, each column containing one timestep

2: Create two new matrices $\boldsymbol{X}_1$, $\boldsymbol{X}_2$ that span $\boldsymbol{X}(1 : end - 1)$ and $\boldsymbol{X}(2 : end)$, respectively

3: Set $\boldsymbol{X}_2 = \boldsymbol{A} * \boldsymbol{X}_1$

4: Take "economy" SVD of $\boldsymbol{X}_1$ so that $\boldsymbol{X}_1 = \boldsymbol{U} * \boldsymbol{S} * \boldsymbol{V}^*$

5: Truncate SVD matrices for desired lower rank representation

6: $\hat{\boldsymbol{A}} = \boldsymbol{U}^* * \boldsymbol{X}_2 * \boldsymbol{V} * \boldsymbol{S}^{-1}$

7: Eigenvalue decomposition of $\hat{\boldsymbol{A}}$: $\hat{\boldsymbol{A}} * \boldsymbol{W} = \boldsymbol{W} * \boldsymbol{\lambda}$

8: Dynamic modes $\boldsymbol{\psi} = \boldsymbol{X}_2 * \boldsymbol{V} * \boldsymbol{S}^{-1} * \boldsymbol{W}$ (time dynamics represented by $\boldsymbol{\lambda}$)

9: Vector coefficients representative of initial conditions: $\boldsymbol{b}_o = \boldsymbol{\psi}^+ * \boldsymbol{X}_o$ where $\boldsymbol{\psi}^+$ is the psuedoinverse of $\boldsymbol{\psi}$

10: Low rank recreation of system at each time step: $\hat{\boldsymbol{X}} = [\boldsymbol{\psi} * \boldsymbol{\lambda}^0 * \boldsymbol{b}_o, \boldsymbol{\psi} * \boldsymbol{\lambda}^1 * \boldsymbol{b}_o, \boldsymbol{\psi} * \boldsymbol{\lambda}^2 * \boldsymbol{b}_o, ..., \boldsymbol{\psi} * \boldsymbol{\lambda}^n * \boldsymbol{b}_o]$

2.3.4 Algorithm 2.3 Discussion

1: Fig. 2.5 shows the vectorization process.

2: $\boldsymbol{X}_1$ and $\boldsymbol{X}_2$ are the "observables."

3: Since $\boldsymbol{X}_2 = \boldsymbol{A} * \boldsymbol{X}_1$, $\boldsymbol{X}_n = \boldsymbol{A} * \boldsymbol{X}_{n-1}$. Carrying this out for $\boldsymbol{n}$ timesteps starting at $\boldsymbol{X}_o$ yields a Krylov subspace of the form shown in equation 2.10. This equation also means that $\boldsymbol{A} = \boldsymbol{X}_2 * \boldsymbol{X}_1^+$, $\boldsymbol{X}_1^+$ being the Moore-Penrose psuedoinverse of the matrix $\boldsymbol{X}_1$. This solution effectively applies a least square fit to the operator $\boldsymbol{A}$ [38].

4: Section 2.2.2.

5: Truncate using method described in section 2.2.1

6: If we start with the equation $\boldsymbol{A} = \boldsymbol{X}_2 * \boldsymbol{X}_1^+$ from step 3, then we can simply define $\boldsymbol{X}_1^+$ as $\boldsymbol{V} * \boldsymbol{S}^{-1} * \boldsymbol{U}^*$, and we have the equation $\boldsymbol{A} = \boldsymbol{X}_2 * \boldsymbol{V} * \boldsymbol{S}^{-1} * \boldsymbol{U}^*$. This is fine, but if we say that $\hat{\boldsymbol{A}} = \boldsymbol{U}^* * \boldsymbol{A} * \boldsymbol{U}$ (the projection of the full matrix $\boldsymbol{A}$ onto the principal components $\boldsymbol{U}$ [24,47]) is what we want, then the equation becomes $\hat{\boldsymbol{A}} = \boldsymbol{U}^* * \boldsymbol{X}_2 * \boldsymbol{V} * \boldsymbol{S}^{-1} * \boldsymbol{U}^* * \boldsymbol{U}$. Because $\boldsymbol{U}$ is unitary, $\boldsymbol{U}^* * \boldsymbol{U}$ is simply an identity matrix, and the equation becomes $\hat{\boldsymbol{A}} = \boldsymbol{U}^* * \boldsymbol{X}_2 * \boldsymbol{V} * \boldsymbol{S}^{-1}$. $\hat{\boldsymbol{A}}$ is also now in a form so that a simple eigenvalue decomposition can be performed on it. This step is the most important conceptually, and two main points will be discussed:

   (a) $\hat{\boldsymbol{A}}$ is the truncated low rank version of $\boldsymbol{A}$, conveniently avoiding the calculation of a full rank $\boldsymbol{A}$ and subsequently its eigenvalue decomposition.

   (b) $\hat{\boldsymbol{A}}$ is the Hessenberg form of the matrix $\boldsymbol{A}$ ($\boldsymbol{U}^* * \boldsymbol{A} * \boldsymbol{U}$). This is significant because it shows that $\hat{\boldsymbol{A}}$ carries more temporal information than just performing the SVD on $\boldsymbol{X}$ and finding $\boldsymbol{U}$. $\boldsymbol{A} * \boldsymbol{U}$ is the representation of $\boldsymbol{U}$ shifting over a timestep (recall $\boldsymbol{A}$ is the operator that relates one timestep to another). So $\boldsymbol{U}^* * \boldsymbol{A} * \boldsymbol{U}$ is the correlation of the principal components to themselves through time [29, 47].

7: $\boldsymbol{W}$ are the eigenvectors and $\boldsymbol{\lambda}$ are the eigenvalues.

8: **Dynamic modes:** The result of the reduction of $\boldsymbol{A}$ to $\hat{\boldsymbol{A}}$ is that when the DMD modes and total low rank data is desired to be extracted, the eigenvectors $\boldsymbol{W}$ must be projected onto the principal components, $\boldsymbol{U}$, in order to satisfy matrix sizes. This is the case in standard DMD, and in fact for standard DMD the equation for the dynamic modes is simply

$\boldsymbol{\psi} = \boldsymbol{U} * \boldsymbol{W}$ [47]. Exact DMD, however, uses the equation presented in step 9 to ensure that $\boldsymbol{\psi}$ does in fact contain the eigenvectors of $\boldsymbol{A}$, not just a projection. This is because the modes calculated using the equation in step 9 are in the image of $\boldsymbol{X}_2$, which is directly used to calculate $\boldsymbol{A}$. In traditional DMD, using the equation $\boldsymbol{\psi} = \boldsymbol{U} * \boldsymbol{W}$ means these modes are in the image of $\boldsymbol{X}_1$ (not directly used to calculate $\boldsymbol{A}$) [29].

Temporal Dynamics: How the eigenvalue associated with each mode is changing through time ($\boldsymbol{\lambda}^n$) is representative of how each mode is changing through time. The imaginary part of the eigenvalue (if there is one) indicates the oscillatory nature the mode, specifically its frequency. The real part of eigenvalues indicate the growth or decay of the mode. The magnitude of an eigenvalue that is greater than one indicates growth, while a magnitude of less than one indicates decay. This is due to the nature of the time dynamics part of the equation given in step 10, the eigenvalues are exponentially increased throughout time. So if the magnitude of the eigenvalue is greater than one, it will increase exponentially (unstable). If the magnitude of the eigenvalue is less than one, it will decay exponentially (stable). Finally, if the real part of the eigenvalue is one without any imaginary part, it will be constant in time. It should also be noted that if an eigenvalue decomposition yields complex eigenvalues, those complex eigenvalues have a tendency to repeat [41].

The time dynamics can be scaled by multiplying by the associated coefficient in $\boldsymbol{b}_o$. This is arbitrary until combinations of modes are desired, and then scaling becomes relevant. The term commonly used for different representations of the eigenvalues of $\boldsymbol{A}$ is called the DMD spectrum. The specific variations and manipulations shall be discussed in

sections 3.4 and 3.5. Several works have established typical convention for the visualization of the DMD spectrum [43, 47–49].

9: $\boldsymbol{\psi} * \boldsymbol{b}_o = \boldsymbol{X}_o$, so $\boldsymbol{b}_o = \boldsymbol{\psi}^+ * \boldsymbol{X}_o$, $\boldsymbol{\psi}^+$ being the psuedoinverse of $\boldsymbol{\psi}$. When plotting/visualizing the dynamic modes and their separate time dynamics, $\boldsymbol{b}_o$ is multiplied by $\boldsymbol{\psi}$ and the time dynamics for scaling purposes.

10: This low rank representation equation is a combination of ideas from the Krylov subspace (Section 2.3.2) and the Koopman process (Section 2.3.1), it is Equation 2.11.

2.3.5 Numeric Properties of Dynamic Mode Decomposition

This section provides some discussion on the numeric properties of the methods used to achieve dynamic mode decomposition. It should be noted that the stability referenced below is not the stability of whatever system the DMD is being performed on, but the stability of the DMD analysis. Convergence of the low order simulation is calculated with respect to the high order data in the original simulation. Discussed in Section 2.3.4, the stability of the system is determined by an inspection of the eigenvalues of the linear operator $\boldsymbol{A}$.

Stability: There are several ways to accomplish DMD. A QR decomposition could be used instead of SVD. It was determined, as [29,47] state, that since the SVD method laid out in [47] is more numerically stable, robust, and universally applicable, it has been generally accepted as the definition of DMD. Exact DMD uses the same properties as DMD with the exception of projected versus actual dynamic modes discussed in Section 2.3.4. The Arnoldi method uses a more numerically stable approach to establish the low order system [31,47], but since the data based ROM is desired instead of a model based one, it is necessary to sacrifice some amount of stability.

Convergence: The convergence of DMD is defined as how close the low order data computed from DMD matches the original simulation. Several studies on DMD have observed the convergence of DMD. In the original paper outlining the use of SVD for DMD [47], Schmid shows that with a sufficient number of snapshots and included POD modes the DMD converges to 10^{-6}. Even if all of the POD modes are used and nothing is truncated, there will still be some residual because the calculation of $\boldsymbol{A}$ uses a form of a least squares fit of a linear operator to a non-linear system. An issue might arise with trying to find the right frequency to sample the data in order to not gather too much data, but still have enough information to effectively capture the dynamics of the system. As described by Kutz et. al [29], DMD has the ability to sample the system non-sequentially and still retrieve sufficient system dynamics using methods from Colonius and Taira [12], Noack et. al [35], and Taira and Colonius [52]. For the purpose of this work, however, this was unnecessary since the sampling was taken at a prescribed frequency after the simulation reached steady state.

2.3.6 Predictive Capability of Dynamic Mode Decomposition

An interesting property of a dynamic mode decomposition is that since the low rank recreation of the data is only a function of $\boldsymbol{A}$ and the initial conditions $\boldsymbol{X}_o$, there is no need to necessarily stop the simulation after the same amount of time steps as the original simulation. It could be run further, representing a low rank recreation of what has already happened and then a low rank creation of what will happen with respect to the linear operator $\boldsymbol{A}$. While possible, prediction is not used this work. However, it has interesting implications with modeling data sets such as disease outbreaks [25, 39] and weather patterns [30].

2.4 Parallelization

The basic idea behind parallel computing is that more processors can do a job faster than one processor. As mentioned in Chapter 1 parallel computing becomes necessary when computational jobs on the scale of DNS multiphase flows are being run. Shared memory and distributed memory are two important concepts to understand when dealing with parallelized computing [53]. Shared memory uses one main memory processor that controls everything and multiple other processors that actually carry out the job. The main memory processor in shared memory controls how the job processors contribute to the main memory. Distributed memory has many processors that each have their own part of the total memory required for the job. Each processor reads the code being run and uses some type of message passing to communicate the information each processor might need from the others. If a job requires no message passing between processors for each processor to complete its job it is considered trivially parallelizable. The processors run their respective parts of the code simultaneously (waiting for a section from a different processor to finish may be required) and, once all processors have completed their respective jobs, they collaborate their information to give the desired result.

Distributed memory is what was used for this research. This method of parallel computing was desirable due to the highly scalable nature of distributed memory. Shared memory becomes more complicated and has a bottle-necking effect due to the main memory processor at scale. The only added cost to scaling up a distributed memory system is the communication per processor. This can be minimized with knowledge of how much memory will be allocated from the meshing of the CFD code. If this is known (in the case of this work it is), a careful calculation of how much computational work to assign to each processor can be made and added

communication cost can be minimized.

The need for parallel computing leads to the question of what kind of a system to run these simulations on. There are several types of systems that have been constructed for the purpose of parallel computing. One of these systems is a computing cluster, which is what was used for this research. A computing cluster is a large network of computers connected to be allocated to do many things at once. A cluster has a number of nodes, which are essentially computers, that each consist of a number of processors. A head node in connection with a few other storage nodes typically connect to the computing nodes. This is done so that the code may be run from the head node, which will allocate the appropriate number of nodes for the job. The storage nodes are so that the computing nodes can output information to keep on the storage node and remain completely available for its real purpose of computing, as opposed to storage. This may seem similar to the shared memory methodology discussed above, but the head node is not the main memory processor, it is simply a way to allocate the number of nodes desired and communicate the code that each node will run in the form of the distributed memory process mentioned above. This work uses the Hyalite luster on the Montana State University campus to run the simulation outlined in Chapter 4. The Hyalite cluster is composed of 65 nodes and a total of 1080 cores [1].

CHAPTER THREE

METHODOLOGY

3.1 CFD Solver and Platform

The proposed work is built into an existing computational platform called NGA [13, 37], which is written in Fortran [51]. The NGA code that was written prior to this work is highly parallelized. As discussed in Section 3.3, the tools built into the NGA code in this work were made equally parallelizable to allow for scalable simulations and fast run times. The NGA code uses Message Passing Interface (MPI) to communicate across processors [51].

The tools created in this work are meant to collaborate with the simulations already available through the NGA code. NGA has multiphase DNS capabilities and uses advanced numeric techniques to accurately represent atomizing jet gas/liquid interfaces. It uses a staggered Cartesian mesh with a semi-implicit second-order Crank-Nicolson formulation similar to the method of Choi and Moin [10] to solve the incompressible Navier Stokes equations at low Mach numbers. For multiphase flow, NGA uses second-order finite difference operators away from the interface [14] and the volume-of-fluid (VOF) method [36, 37] close to the interface. This output a plane in each sub-cell which will be referenced as a PLIC (piecewise linear interface curvature).

To extract the information necessary for the DMD, first a representation of the two-dimensional phase interface is needed (written in Fortran and compiled with the NGA code). To do this, the jet's liquid core was defined as a two-dimensional matrix of radii $R(\theta, x)$ that are a function of the axial direction x and azimuthal angle θ. The jets core was then in a type of cylindrical coordinates, i.e. a two dimensional

matrix. This method is described in detail in section 3.2. With $R(\theta, x)$, the DMD analysis is performed, discussed in section 3.4. The DMD code was written entirely in Matlab.

3.2 Extraction of Jet's Liquid Core Surface

Given that it was decided that DMD would be the best method to achieve the goal of the numeric categorization of an atomizing jets destabilization, it was then decided that the best method to gather the data necessary for DMD would be to find a way to represent the surface of the jets liquid core. The rest of the jet's core inside the surface and the droplets that break off from the jet are excluded. Focusing on the surface of the jet allows for an acceptable representation of the areas of interest in the breakup, which are the fluctuations of the surface topology. Excluding the already separated droplets allows for the focus to remain on the cause of the breakup, the instabilities on the core.

Discussed in Section 3.1, the NGA code represents gas/liquid interfaces as a plane in each cell. This methodology results in possible discontinuities in the representation of the jets interface. A smooth continuous representation of the jets liquid core is preferable. This, along with the fact that DMD must be performed on individual data points related to themselves through time (not equations of planes), means that a continuous function to represent the jets core must be created from the jets PLIC interface representation. Although DMD is capable of using three dimensional Cartesian coordinates as observables (it would simply be a matter of a much taller matrix, with each vectorized timestep having x, y, and z locations stacked on top of each other), the representation of the atomization jet in Cartesian coordinates is not trivial. The most significant issues being that the vectorized matrix size must stay constant and the spatial sampling locations must be constant. Each point must be

related to itself throughout time in order to accurately portray how that same point is moving through time. With the methods considered for Cartesian coordinates, the achievement of this would have been, as mentioned above, non-trivial. Instead, a method was established to represent the jet as a function of its streamwise center-line and the radial distance from that center-line, thus effectively representing the jets liquid core in two dimensions as a continuous function. A visual representation of this can be seen in Fig. 3.1. It should be noted that the function of the center of the axial jet shown in Fig. 3.1 is referenced as $\boldsymbol{X}$. This is because of the axial nature of the jet and equates the axial direction to the center function. This is actually a more specific case of the more general center function of the jet $\boldsymbol{S}$, where the center function does not have to be defined by the axial direction and the jet could have some curvature to it. This is shown in Fig. 3.3. The steps for the calculation of the jets liquid core are shown in Fig. 3.2. Note that the steps given numbers in Fig. 3.2 are for reference during the subsequent discussion of steps, not an exact timeline that was followed. The source code is available in Appendix A.

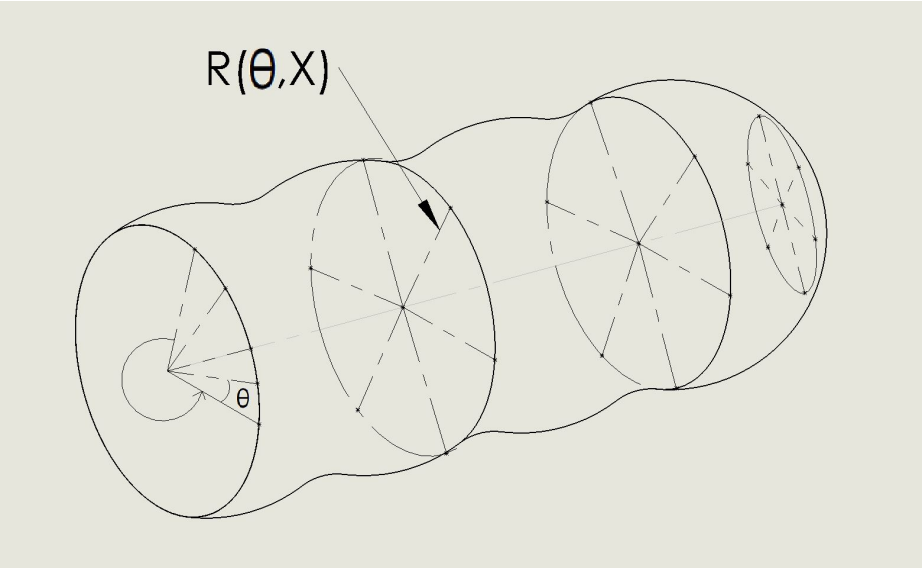


Figure 3.1: Axial Jet Core Data Extraction Progression

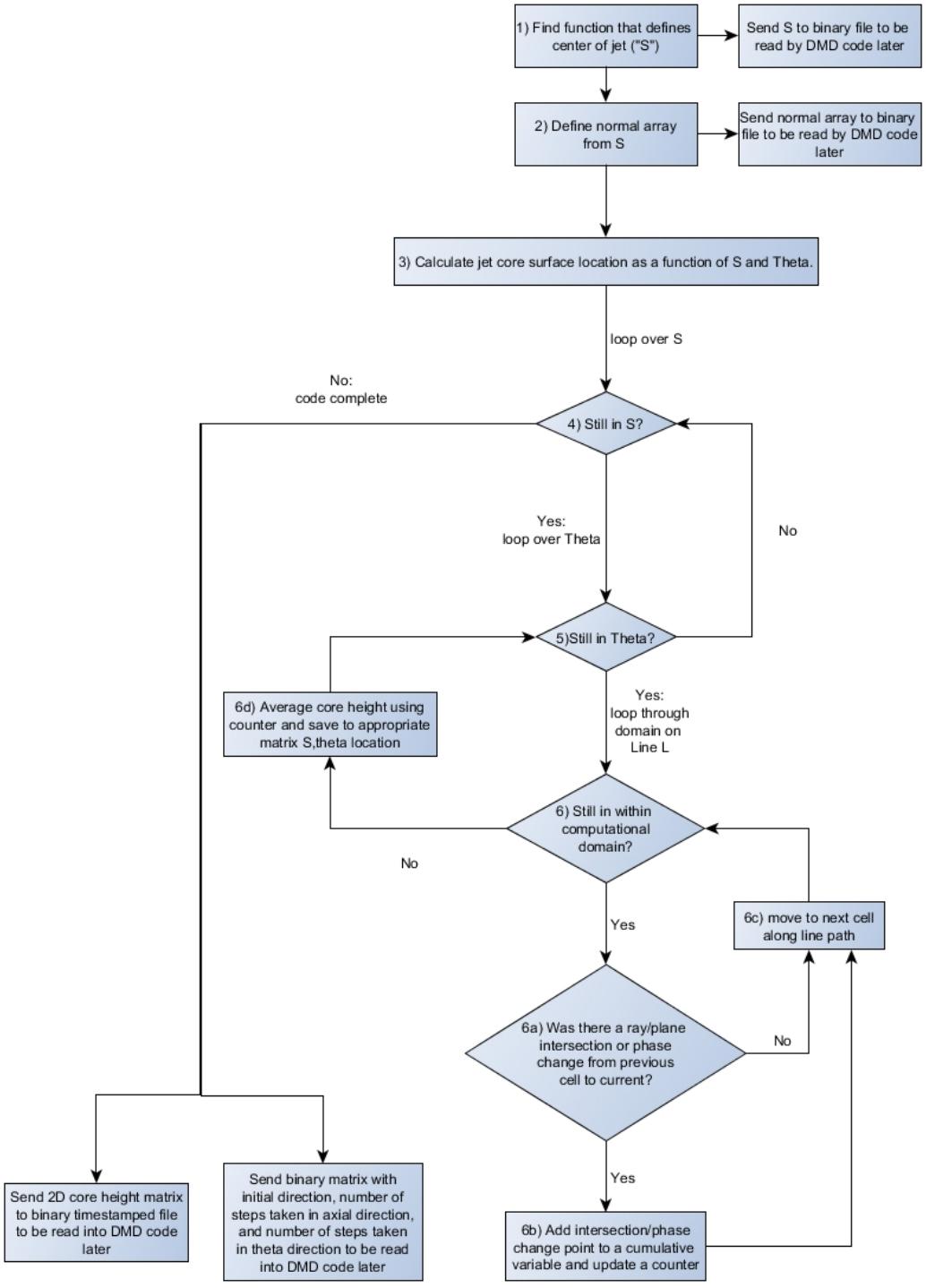


Figure 3.2: Core Height Extraction Logic

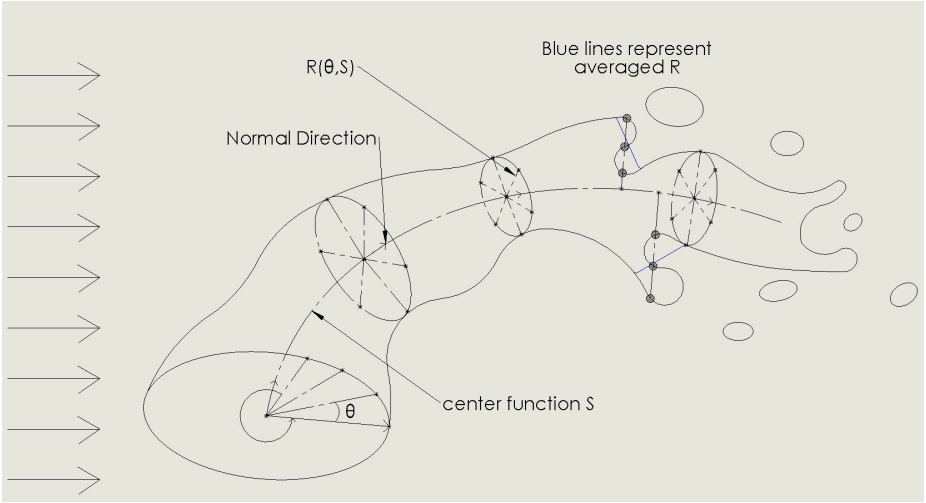


Figure 3.3: Curved Jet Core Data Extraction Progression

3.2.1 Expansion on Key Steps of Core Height Extraction Logic

1: **Find Function That Defines the Center of the Jet (S)**

To establish the function that defines the center of an axial jet, a few things must first be known. Found from the input file and initial conditions, the direction of the jet and the initial shape of the jet can be found. A curved jet requres different information and would require slightly different code structure, described below. Either an axial jet or a curved jet could be analyzed for the DMD. Currently the code is configured for an axial jet, but all of the math in the code and parallelization of the code is valid for a curved jet as well. Note that all of following steps reference an axial jet.

Axial Jet: The axial jet center function definition is very simple. Starting with the initial shape of the jet, found in an initial condition file, the first center point can be defined. In most atomizing jet propagation codes the initial shape of the jet is a half sphere with the face of the shape parallel to the axial direction. The first center point is at the center of

the circle on that face. Next, the propagation direction is defined from the jet direction found in the input file. The center points in the non-axial directions remain the same throughout the function, the only change being in the axial direction by some step size. The step size is determined as an integer/sub-cell. The function runs through the entire domain and remains the same throughout the simulation.

Curved Jet: For application to a curved jet, the main difference is that the entire simulation must be run first to collect the information necessary to find a consistent function. This means that the core height extraction code must be a post processing tool. The simulation would be run, placing and saving a data point at every cell on the jets core. Once the simulation was finished, all the data points would be included into a type of three dimensional least square fit to determine the constant function that defines the jets curved center through time.

2: **Define Normal Array**

For an axial jet the normal array is simply a unit vector in the axial direction at each $\boldsymbol{S}$ step. For a curved jet the normal is defined as the directional unit vector between any "previous" point along $\boldsymbol{S}$ with respect to the "current" point.

3: **Calculate Jet Core Surface as Function of $\boldsymbol{S}$ and θ**

See below: $\boldsymbol{S}$ and θ Loops

4: **$\boldsymbol{S}$ loop**

At each $\boldsymbol{S}$ step, an number of radii (defined by the how many θ iterations are being used) are calculated and saved into a matrix.

5: **θ Loop**

At each θ location (the number of θ iterations is user defined), a radius (or radii) is calculated by marching along a line ($\boldsymbol{L}$) and finding the intersection between the line and the jets liquid core. The line is a function of some radius R and is defined by the normal plane at that $\boldsymbol{S}$ location, two reference vectors, and the θ location.

$$\boldsymbol{L}(R) = (\boldsymbol{t}_1 * \cos(\theta) + \boldsymbol{t}_2 * \sin(\theta) + \boldsymbol{x}_o) * R \tag{3.1}$$

$$\boldsymbol{t}_1 = \boldsymbol{n} \times \boldsymbol{V} \tag{3.2}$$

$$\boldsymbol{t}_2 = \boldsymbol{n} \times \boldsymbol{t}_1 \tag{3.3}$$

Where $\boldsymbol{n}$ is the normal vector, $\boldsymbol{t}_1$ and $\boldsymbol{t}_2$ are the reference vectors, $\boldsymbol{x}_o$ is the Cartesian center location at that step, and $\boldsymbol{V}$ is an arbitrary vector used to construct the two reference vectors. The arbitrary vector is necessary because the cross product of any vector and the normal vector will be on the normal plane and provides a reference for the angle θ to define the line $\boldsymbol{L}$. The only restrictions on the arbitrary vector are that it must always be the same, and it must be different from the normal vector. Fig. 3.4 shows a visual representation of the vectors defined in equations 3.1, 3.2, and 3.3. A visual representation of line/PLIC intersection shown in Figures 3.5 and 3.6. It should be noted that if the number of θ steps is set to something besides 360, then a degree step size variable is calculated and a new counter in the theta loop must be created to account for both true degree location and matrix locations.

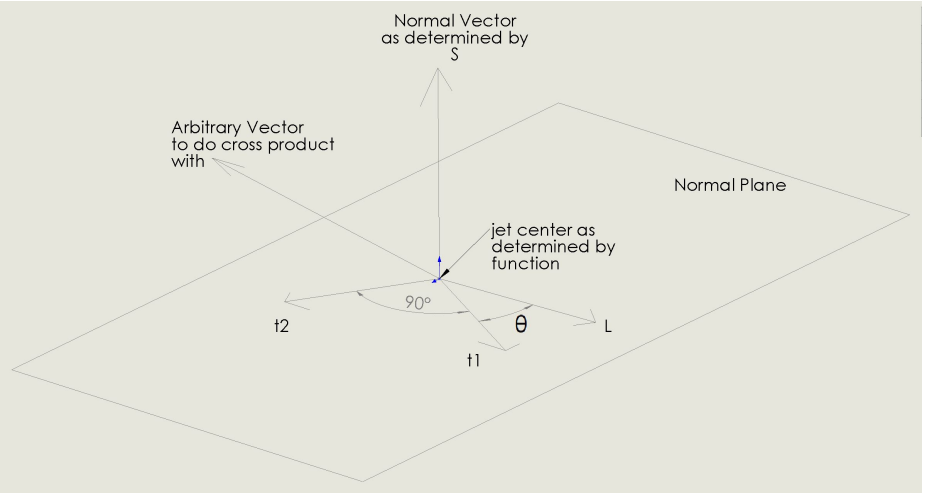


Figure 3.4: Vector Math Visual

6: **Line Loop**

The line loop marches along the line $\boldsymbol{L}$ defined by Equation 3.1 through the computational domain, (stopping when the end of the domain is reached in any direction) checking for line/interface intersections (Fig. 3.6). Fig. 3.5 shows a cross sectional view of the general jet propagation process. However, the actual representation of the jets core looks more like Figures 3.6 and 3.7 (in all of these figures the vertical and horizontal lines represent computational cell boundaries in two dimensions), where the PLIC defined by the NGA code produces a discontinuous function. Because of this, there is a possibility that the line never intersects the interface, but instead goes directly from a liquid to a gas (Fig. 3.7). In this scenario, the intersection of the line and the cell boundary the line crosses from liquid to gas is the intersection that defines the radius. To account for this possibility it is necessary to keep information from both the previous and current cells along the line loop, allowing for calculation of interface intersections as well as phase change locations.

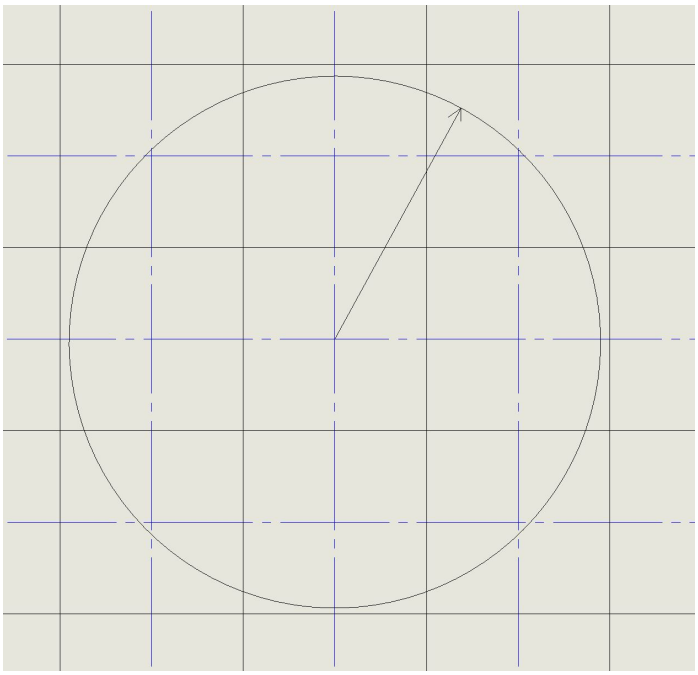


Figure 3.5: Cross sectional view of line marching through the grid

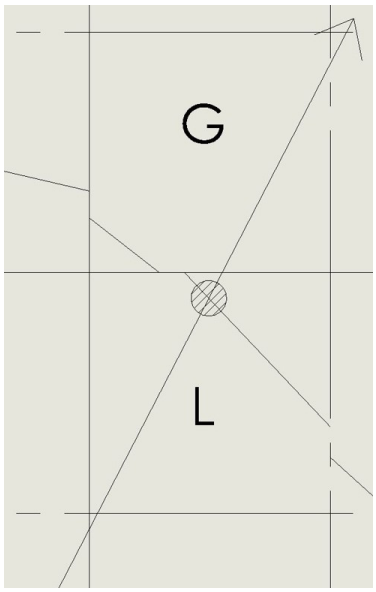


Figure 3.6: Cross sectional view of line $\boldsymbol{L}$ intersecting the interface (G is gas and L is liquid)

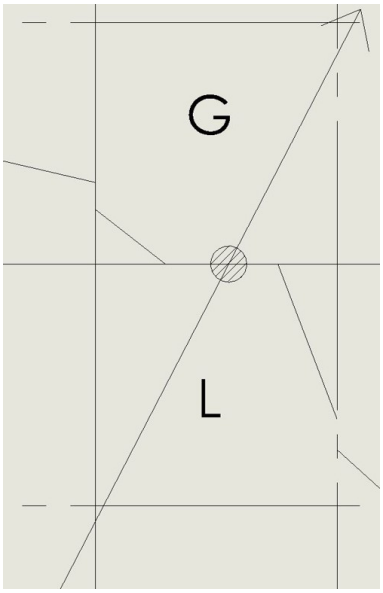


Figure 3.7: Cross sectional view of line $\boldsymbol{L}$ missing the interface (G is gas and L is liquid)

Before interface intersections and phase change locations can be calculated, the method of how to actually propagate along the line $\boldsymbol{L}$ through the computational grid must be introduced. Fig. 3.8 shows $\boldsymbol{L}$ running through a computational cell. Starting in the cell that the center function at that axial location is in, the distance between the starting point of $\boldsymbol{L}$ and each possible cell boundary exit (each boundary is a plane) is calculated. This is done by defining the cell boundaries as

$$A * x + B * y + C * z - D = 0 \tag{3.4}$$

(where all the parameters in Equation 3.4 are known) and the distance as a radius R defined by equating Equations 3.1 and 3.4

$$R = -(A * x_o + B * y_o + C * z_o D)/(A * L_x + B * L_y + C * L_z) \tag{3.5}$$

where L_x, L_y, and L_z are the directional components of $\boldsymbol{L}$. The possible cell boundary exits are determined by the direction $\boldsymbol{L}$ is moving with respect to the way the boundaries of the computational cells are oriented. If $\boldsymbol{L}$ is oriented in the positive X direction, then with respect to that direction it can only exit the cell through one X boundary. Similar reasoning can be made for all three directions. This narrows down the amount of checks necessary. The shortest R value of the possible cell boundary exit intersections defines the exit boundary of the cell. This is because the intersections with the boundaries after the shortest intersection will be outside the cell. Once the exit boundary is known, the cell that shares that boundary is the next cell to move to along the line to. It should be noted that the value R must be positive. If it is negative, the intersection

occurs behind the starting point of $\boldsymbol{L}$. If R is 0, the line is parallel to the plane giving no intersection. These checks are not particularly relevant in this step given that the "non-exit" cells have already been eliminated, but they will be brought up again in subsequent steps when calculating interface intersections. The Cartesian coordinates of the intersection is also calculated using

$$\boldsymbol{X}_i = \boldsymbol{L}(R) \tag{3.6}$$

where $\boldsymbol{X}_i$ is the vector containing the Cartesian coordinates of the intersection point. This was done both for plotting purposes and because the simulation is in Cartesian coordinates.

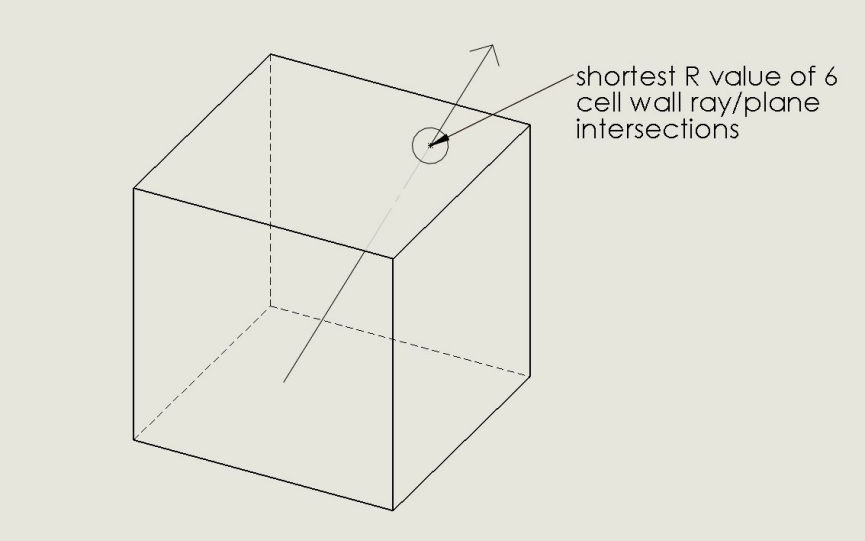


Figure 3.8: $\boldsymbol{L}$ propagation through a computational cell

(a) **Intersection/Phase Change Criteria**

There are two preliminary checks that must be made before the jet's radius can be calculated. The first is structure identification. In the NGA code, each individual liquid structure (the jet's core and every

separated droplet) has its own structure identification. Herrmann [20] established the methodology. An individual liquid structure is defined by a separation of at least one "empty" cell (a cell containing only gas) between the interface of that structure and other structures. This has proven to be a good way to define liquid structures, but it does leave the possibility of long strings of structures that look separate to be defined as one structure. Since only the jet's liquid core is to be accounted for in this calculation, this idea can be used as a preliminary check, saying that if the cell along the line $\boldsymbol{L}$ does not have the same structure identification as the jet's core, it is ignored. If the structure identification in the cell is the same however, we can move on to the second check, the volume-of-fluid check.

Per the NGA output, each cell has a volume-of-fluid assigned to it. If cell has a volume-of-fluid equal to 1, it means that the cell is entirely liquid. If it has a volume-of-fluid equal to 0, it means the cell is entirely gas. If its volume-of-fluid is between 0 and 1, it has a gas/liquid interface defined by a plane in it. The last case mentioned is the case of interest for the radius calculation. The equations used to calculate the line/interface intersection are the same equations used for the cell propagation in the previous step (3.4, 3.5, and 3.6), where the plane is now defined as the interface plane instead of a cell boundary plane. The same R checks mentioned in step 6 are used in this situation. Figures 3.6 and 3.9 show the case of interface intersection within the cell boundaries. If the intersection occurs in the current cell within the cell boundaries, a radius at that $\boldsymbol{S}$ and θ location is saved.

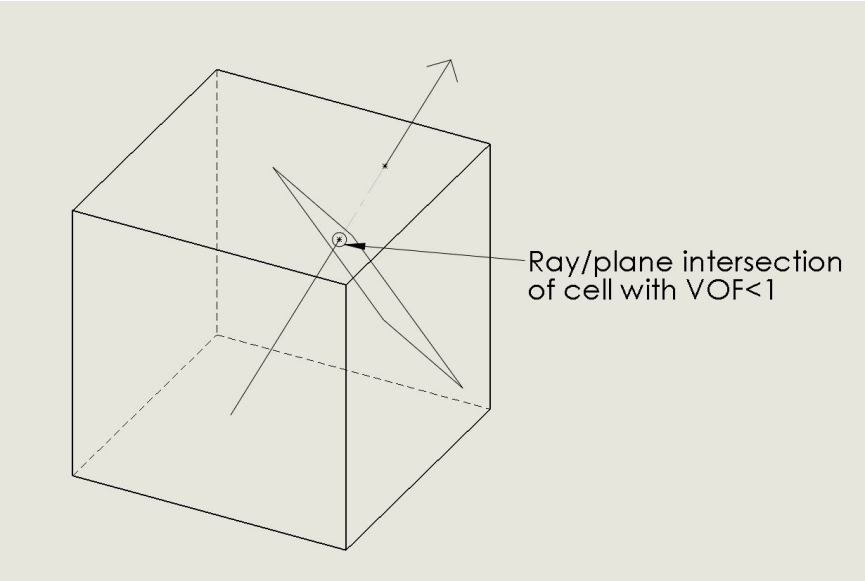


Figure 3.9: Line/PLIC Intersection

If the intersection occurs outside of the cell boundaries (shown in figures Figures 3.7 and 3.10), it must be determined if the line is in liquid or gas. This is done by checking the sign of the plane equation that defines the interface in the cell

$$s = \boldsymbol{n} \cdot \boldsymbol{x} - D \tag{3.7}$$

where s is the sign, $\underline{n}$ is the normal vector of the interface plane in the cell, and $\boldsymbol{x}$ is a vector containing Cartesian coordinates of a location inside the cell and on the line $\boldsymbol{L}$. $\boldsymbol{x}$ is defined by

$$\boldsymbol{x} = (\boldsymbol{x}_{in} - \boldsymbol{x}_{out})/2, \tag{3.8}$$

$\boldsymbol{x}_{in}$ and $\boldsymbol{x}_{out}$ being the entrance and exit points of $\boldsymbol{L}$ through the cell in Cartesian coordinates. If s is positive, the line is in gas, otherwise

the line is in liquid. This is due to the definition of the normal vector of the plane that defines the interface, which points outward from the liquid to the gas.

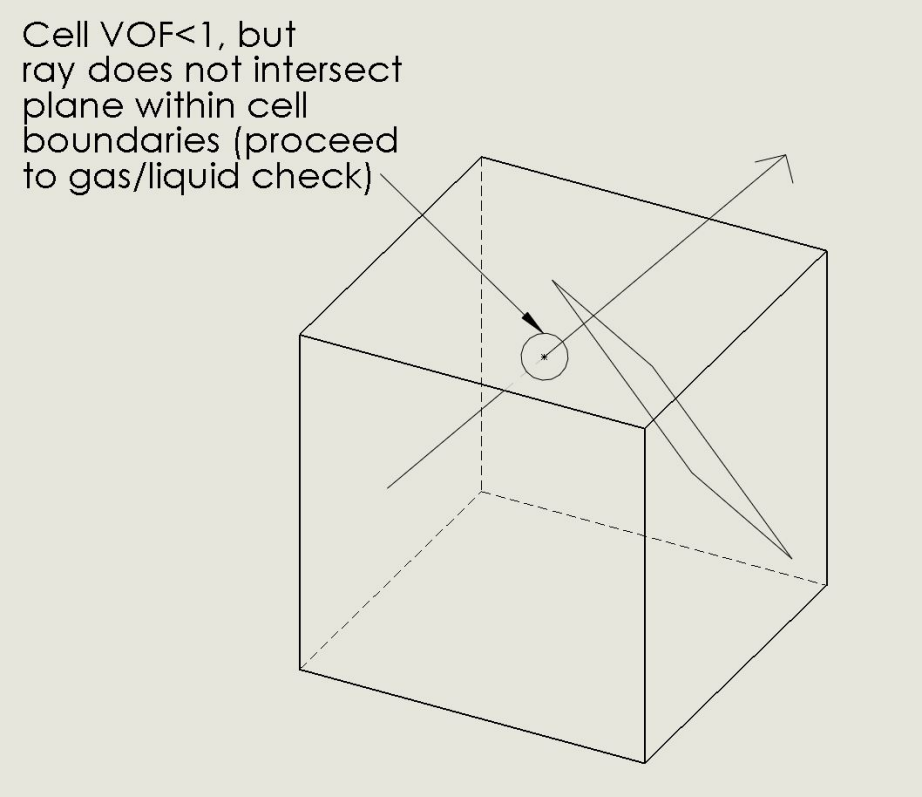


Figure 3.10: Line/PLIC Intersection Miss

If there is a phase change from the current to previous cell, the exit point of the previous cell is saved as a radius at that $\boldsymbol{S}$ and θ location is saved.

(b) **Cumulative Position and Counter Update Criteria**

In a complicated atomizing jet there is a possibility, when calculating a radius, for multiple radii to be captured along the same line. This is shown in Figures 3.3 and 3.11. The overlapping effect shown in these figures is necessary to capture in order to maintain an accurate

description of the jet's core. To do this without sacrificing the two dimensional matrix, the intersection radii are averaged at each θ step. Discussed in Chapter 4, this does still capture the jet's core effectively. So if a radius is defined (using the methodology described in the previous step) at the current cell, the line loop does not exit there. It simply adds the radius to a cumulative radius. Every time this occurs, a counter is also updated.

(c) **Cell Propagation**

Once the current cell has been checked for intersections and (if necessary) the radius and counter cumulative variables have been updated, the next cell along the line (using the methodology described in the line loop discussion) is checked. Discussed above, this process repeats until the end of the domain in any direction is reached.

(d) **Averaging**

When the line loop reaches the end of the domain, the cumulative radius R is averaged by the counter and saved in the appropriate $\boldsymbol{S}$ and θ locations in the radius matrix. The next θ step can then begin.

End of S Function

When the end of the $\boldsymbol{S}$ function has been reached and the core height matrix has been filled, a time-stamped core height matrix and a matrix containing the initial direction, number of steps taken in $\boldsymbol{S}$, and number of steps taken in θ are sent in binary files to be read in by the DMD code discussed in 3.4. The code is called at a user-defined simulation-time interval. It saves a new updated time-stamped binary file each output call for DMD analysis. Sampling frequency is subjective and is discussed in Chapter 4.

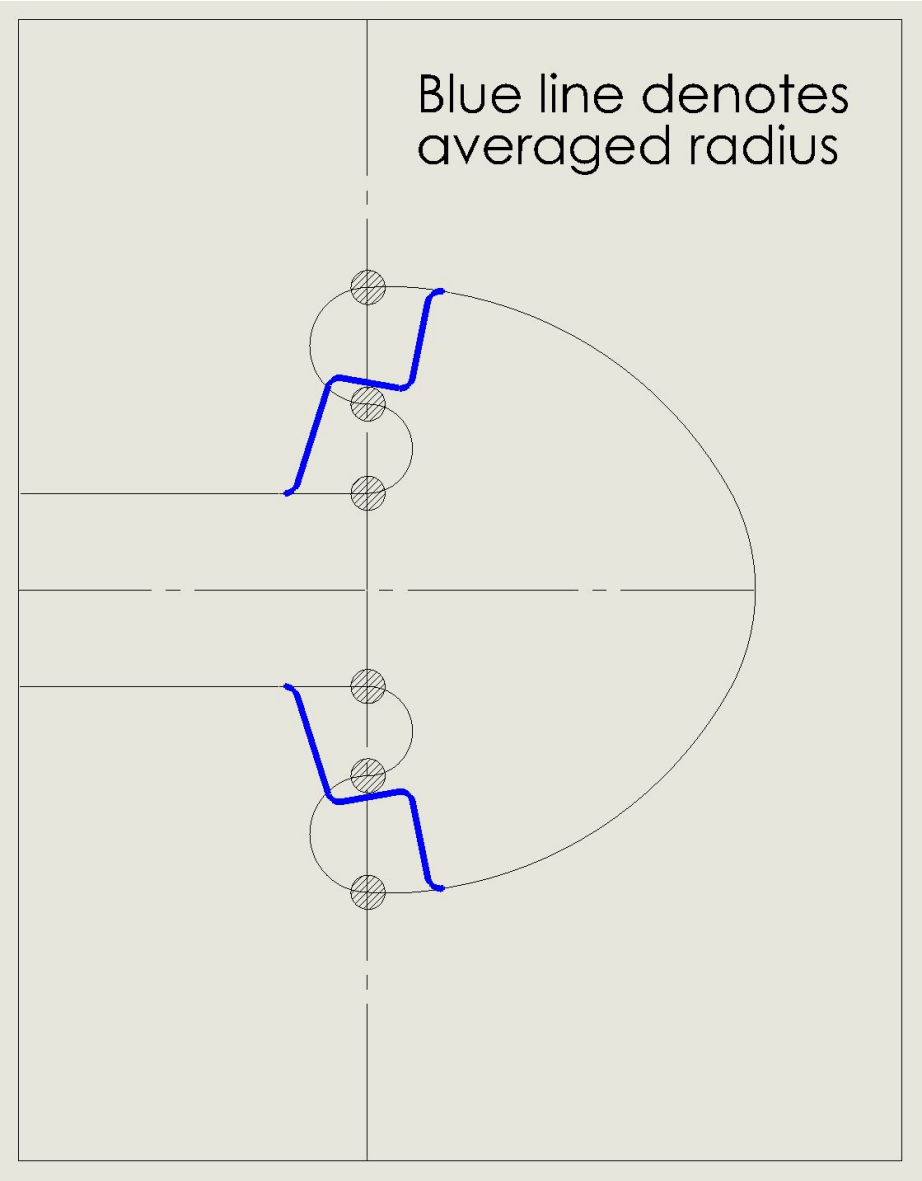


Figure 3.11: Overlap Solution

3.3 Parallelization

The algorithm for the extraction of the surface of the jets liquid core outlined above is not particularly computationally expensive. However, since the Navier-Stokes solver being used is parallelized, it was necessary to parallelize the surface

core algorithm as well to avoid holding up the CFD solver.

For an axial jet where the jet center function is completely straight in the axial direction, the parallelization is trivial in the axial direction. Each processor can do the work allocated to it without any communication from other processors. In a case of 1D axial parallelization, each processor has a slice of the jet to perform the algorithm on. In this situation, there would be no shared calculations in the θ direction, each processor only knows its own local core height matrix. Each processor would then in ascending order contribute its section of the jets liquid core to a global core height matrix. This method is effective to a certain extent and allows the simulation to be run within reason, but still limits the run time.

The method required to increase the dimensionality of the parallelization from one dimensional to three dimensional conveniently applies to a jet in crossflow as well as the axial jet. This allows for an easy adaptation from an axial jet to a jet in crossflow. In the 3D parallelized case, each processor has a matrix the size of the to-be-calculated global core height matrix and simply cycles through the axial ($\boldsymbol{S}$ function) direction line and along the line in the theta direction until a cell that belongs to the processor is reached. The radius calculation and averaging was then carried out for each processor locally (the counter variable was now in the form of the matrix of equal size to the core height matrix). Parallelizing the code in three dimensions creates the possibility of multiple processors having different parts of the line $\boldsymbol{L}$ in their domain. The issue with this is that once a processor loops through the domain and gets to the first cell in its domain, if it is not the first processor along the line $\boldsymbol{L}$, it does not have an accurate current to previous cell relationship to compare its first cell to. To account for this, as each processor looped through other processor domains, it kept track of where it was so that when it did reach the first cell in its own domain it could reach back to the final cell in the previous domain (this is

called a ghost cell, each processor has the information of its immediate surroundings one cell thick in the form ghost cells), and do the same intersection checks. This creates an accurate current to previous cell relationship. Each radii and counter were saved and added to a cumulative value for that location in the global matrix. Once all processors have carried out this process, each processor added its version of the global matrix to all the others. Then once the summation was complete it could be divided elementwise by the matrix of counters for averaging. Now since every processor has the complete core height matrix only one processor was needed to send the binary file for DMD to the post processing tool. This process was still carried out once every time the input file dictated that the dump data files would be called.

3.4 Applying Exact DMD Algorithm to Extracted Positional Data

The application of DMD to the extracted data representing the surface of the liquid core of the jet required creating an external post-processing tool. The tool read in the binary matrices produced by the core height code. The original data was plotted, then DMD algorithm was carried out as described in section 2.3. As mentioned in section 2.2.2, the use of the economy SVD allowed for the entire process to be carried out without the need for parallelization or high-performance computing. The low rank data was plotted along with the dynamic modes and their corresponding time dynamics and the DMD spectra. Several manipulations of the dynamic modes and time dynamics were plotted. The significance of the different visualizations of data will be discussed in section 4. A flow chart and subsequent discussion is provided below in Fig. 3.12 and section 3.4.1. As with the flowchart in Fig. 3.2, the numbers reference details provided in Section 3.4.1. The source code is available in Appendix B.

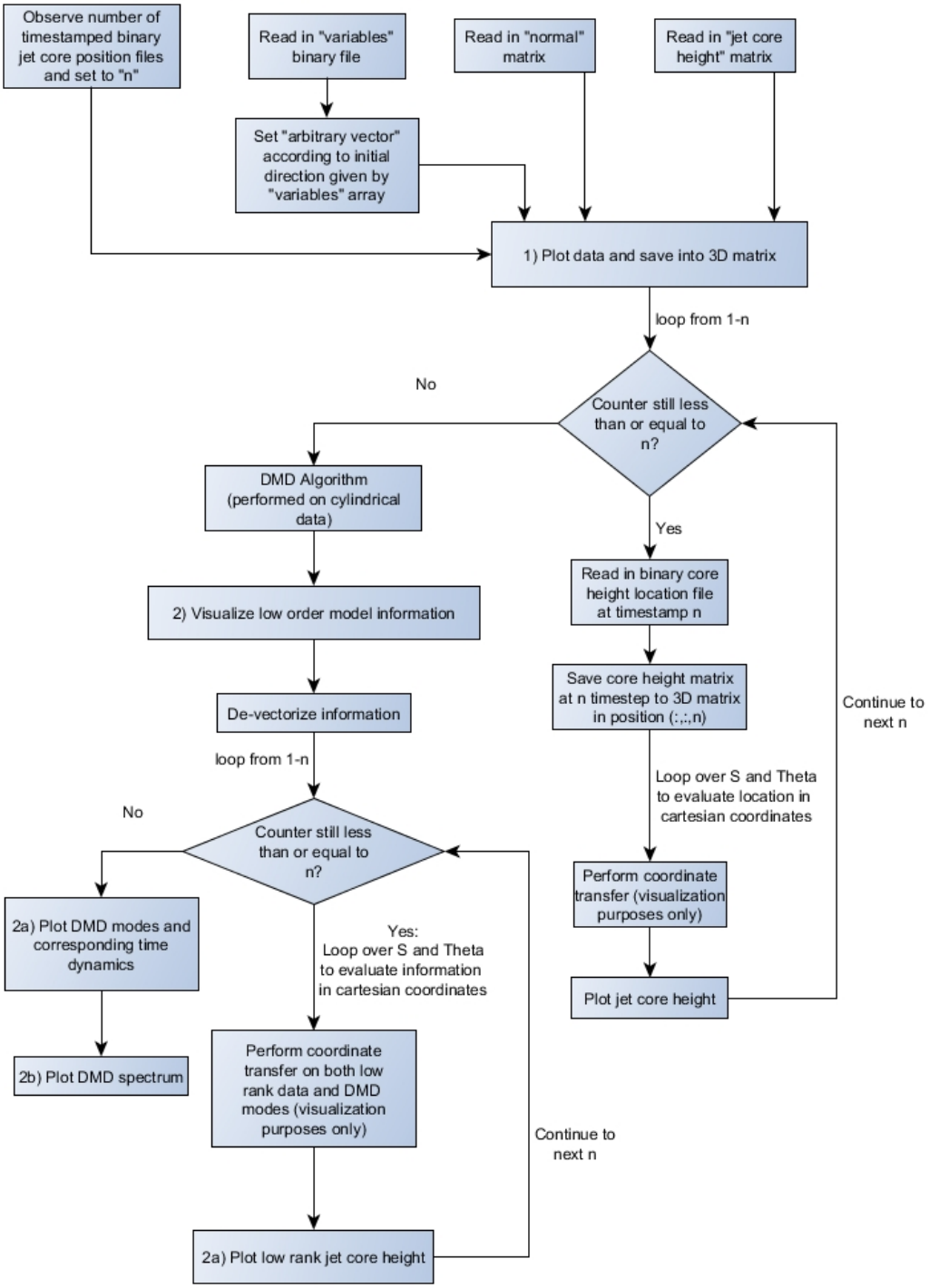


Figure 3.12: Application of DMD to Data Logic

3.4.1 Expansion of Key Steps in DMD Application Logic

1. **Plot Data and Save Into Three Dimensional Matrix**

   The information provided from the "variables matrix was as follows: initial direction; number of steps in the axial direction; number of degrees used in the theta direction at each axial step. From this information along with the jet center location and the normal at each jet center location, the same math (Equations 3.1, 3.2,and 3.3) used in the core height extraction code was performed to calculate the Cartesian coordinates of the jet core height data. X_i, Y_i, and Z_i from Equation 3.6 now are just the X, Y, and Z coordinates of a particular point with respect to a particular radius R at some θ. It should be noted that the coordinate transfer here could have been avoided by sending a different binary file with the core height in Cartesian coordinates from the core height code instead of performing a coordinate transfer here, but the difference was trivial since the coordinate transfer code was already written.

2. **Visualizing data produced by DMD**

   In addition to the matrices produced by the DMD algorithm 2.3 described in section 2.3, several extra matrices were saved for convenience of visualization in this section of the code.

   (a) **Plotting DMD Modes and Low Rank Jet Core Height**

   The DMD modes matrix and the low rank data matrix were both de-vectorized and saved into a separate three dimensional matrix. This made the plotting loop for the low rank core height simpler. The coordinate transfer in this loop was identical to the first coordinate transfer loop described above with the addition of the DMD mode

coordinate transfer. The DMD mode coordinate transfer used the same math (Equation 3.6), was performed immediately after the low rank core height coordinate transfer, and saved the data to a three dimensional matrix.

(b) **Plotting DMD Spectrum**

The best method of visualization of DMD information (in particular the DMD spectrum) is up for debate, but there are some standard methods for the representation of the information that will be described here. All other case specific manipulations of data will be discussed in Chapter 4. Each dynamic mode was scaled by $\boldsymbol{b}_o$ (step 9 of Algorithm 2.3) and plotted separately next to the corresponding time dynamics. Three different visualizations of the DMD spectrum were plotted. $\boldsymbol{\lambda}_r$ vs. $\boldsymbol{\lambda}_i$, $\log(\boldsymbol{\lambda}_r)$ vs. $\log(\boldsymbol{\lambda}_i)$, and the magnitude of the largest singular value of each dynamic mode $||\boldsymbol{\psi}||$ vs. $\log(\boldsymbol{\lambda}_i)$.

At this point, the theory and methodology for applying DMD to an atomizing jet simulation has been established. To validate both the theory and methodology, a test case was developed and will be discussed in the following section.

3.5 Test Case

As mentioned in Chapter 2, a standalone dynamic mode decomposition test case scenario was developed for the purpose of proof of concept, development of the algorithm, and validation of the algorithm. It was desirable to develop this test case scenario without using data from the direct numeric simulation because the large amount of data produced by the DNS would have greatly slowed down the debugging process. Once the algorithm was developed, the code could be applied

to the DNS data with only a few minor changes like importing binary data and coordinate transfers. The source code is provided in Appendix C.

The test case scenario used a combination of parabolic and sine wave functions

```
x = linspace(0,10,100);
y = linspace(0,10,200);
t = 0;
t_end = 20;
for k = 1:t_end
    for i = 1:length(x)
        for j = 1:length(y)
            if y(j) <= 6 + .25*sin(.25*x(i)^2 + k) && y(j)
               >= 4 + .25*sin(.25*x(i)^2 + k)
               C(i,j,k) = 1 - .5*(y(j)-5)^2 + .2*sin(x(i)
                  ^2 + k);
            end
        end
    end
end
```

to represent a statistically steady "jet that oscillated through time. It should be noted that this data was created in Cartesian coordinates, not cylindrical coordinates like the data collected from the DNS. The simulation was run on a 100×200 grid. Figures 3.13 and 3.14 show the first time step of the data from a diametric view and from a top view. Figures 3.15 and 3.16 show the system several timesteps later for a representation of the propagation of the system.

Following the DMD Algorithm 2.3 presented in Section 2.3 the data was then vectorized and the SVD of the resulting X_1 matrix was taken. An array of the normalized diagonal singular value, or $\boldsymbol{S}$, matrix was then created to be plotted. This plot, shown in Fig. 3.17 allows for a good reference for the decision of where to truncate the SVD matrices, described in Section 2.2.1.

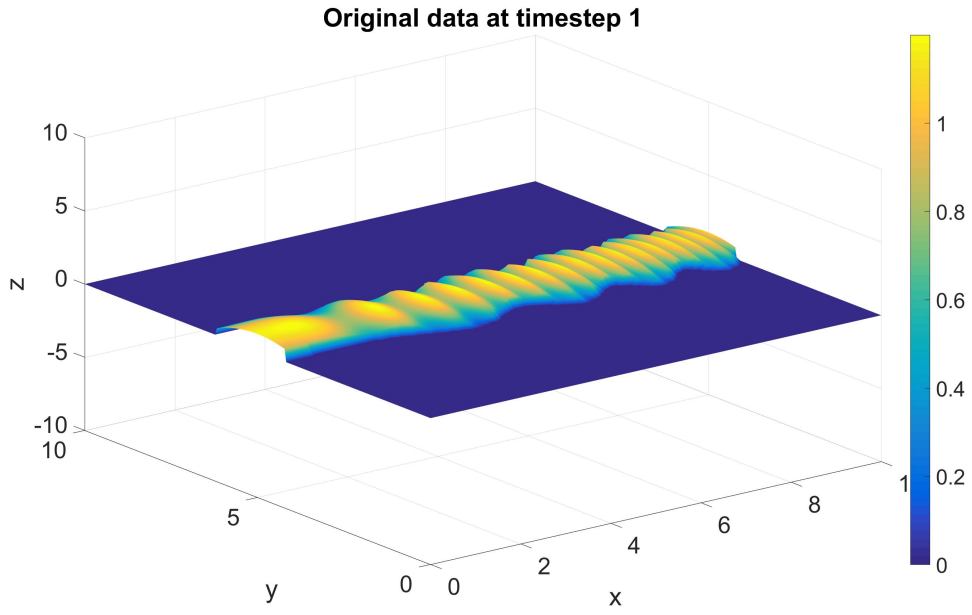


Figure 3.13: Original ("high rank") data at initial timestep

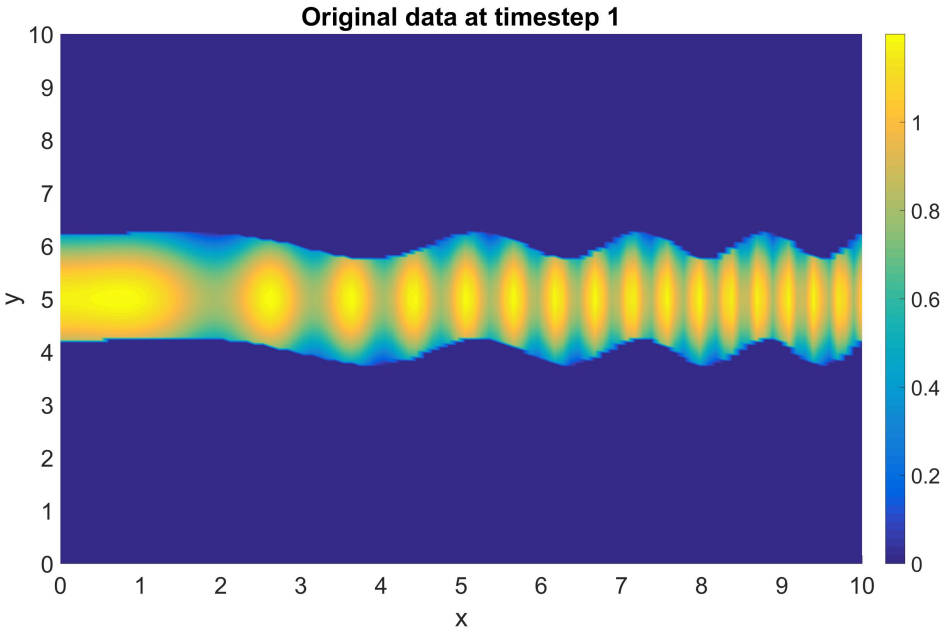


Figure 3.14: Original ("high rank") data at initial timestep from top view

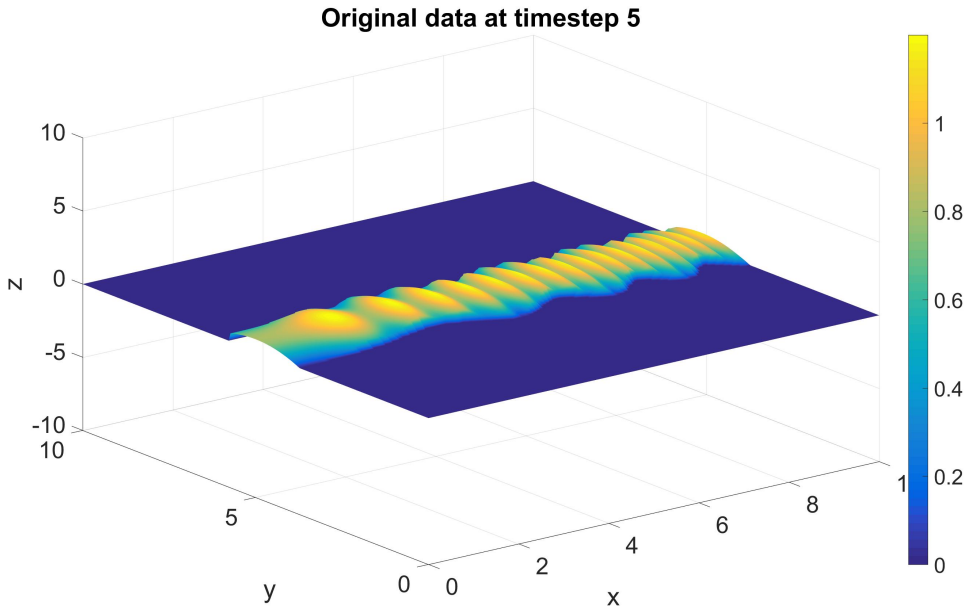


Figure 3.15: Original ("high rank") data at fifth timestep

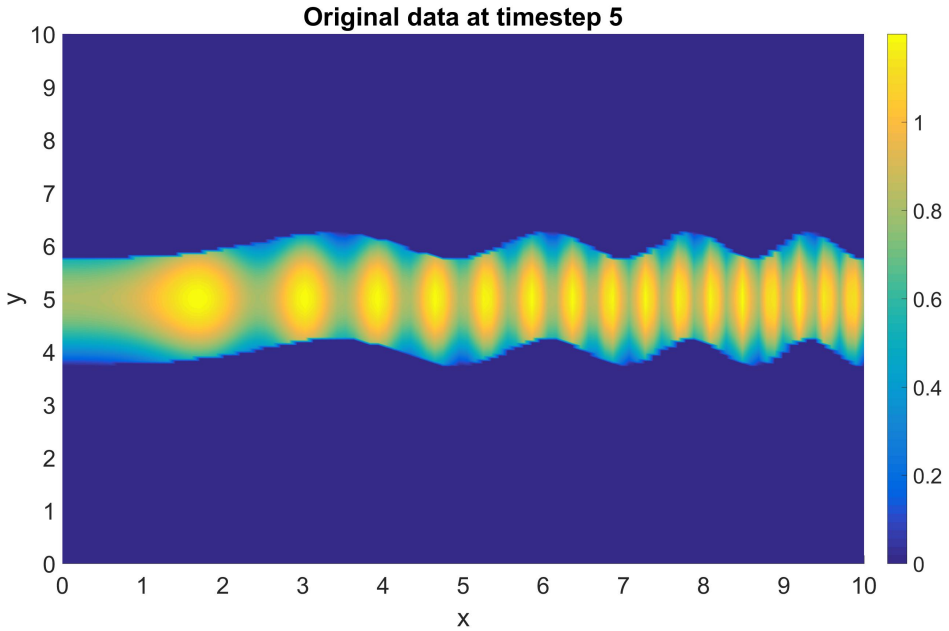


Figure 3.16: Original ("high rank") data at fifth timestep from top view

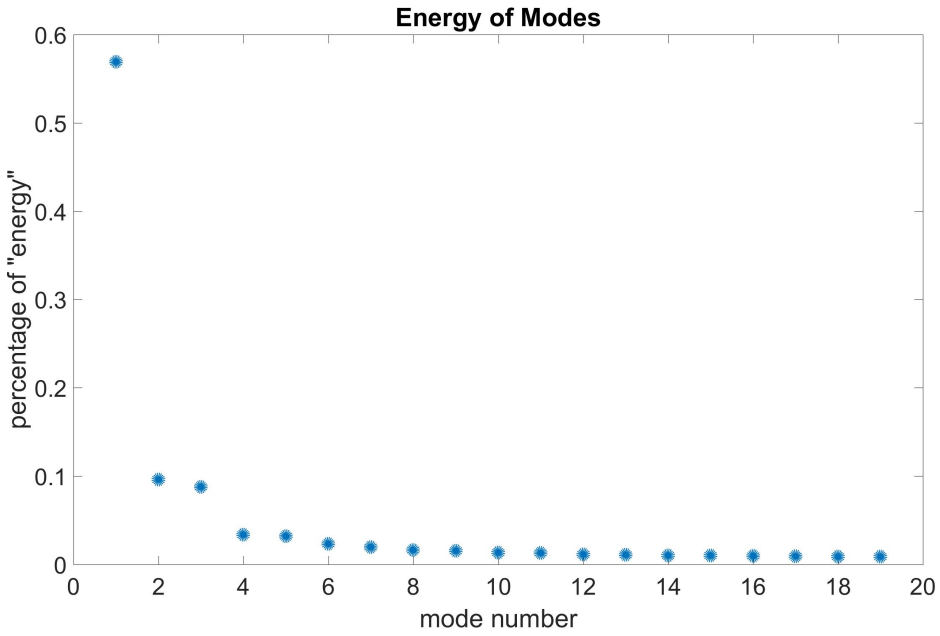


Figure 3.17: Normalized Singular Values of X_1

It can be seen in Fig. 3.17 that there is one primary mode, and a few more dominant modes of interest. The truncation percentage, discussed in Section 2.2.1, was set at 80%, which truncated this data at five modes.

Following the DMD Algorithm 2.3, the linear operator $\boldsymbol{A}$ and what its properties represent were then extracted in the form of dynamic modes and their associated time dynamics. Figures 3.18, 3.19, and 3.20 show the modes with their time dynamics next to them. The DMD tends to extract duplicate modes, particularly in a steady state oscillatory situation. This stems from the repetitive nature of complex eigenvalues discussed in Section 2.3.4. The only difference between the modes is the eigenvalue is opposite in sign. This is why only modes 1, 2, and 4 are shown, modes 3 and 5 are duplicates of 2 and 4. The superposition of all the dynamic modes (and their associated time dynamics) was also plotted in Fig. 3.21. Recall from step 9 of Algorithm 2.3 that $\boldsymbol{X}_o = \boldsymbol{\psi} * \boldsymbol{b}_o$. This means that a superposition of (scaled) modes

is simply a low rank recreation of the initial timestep of data, and the corresponding superimposed time dynamics are representative of the time dynamics of the entire low rank recreated system.

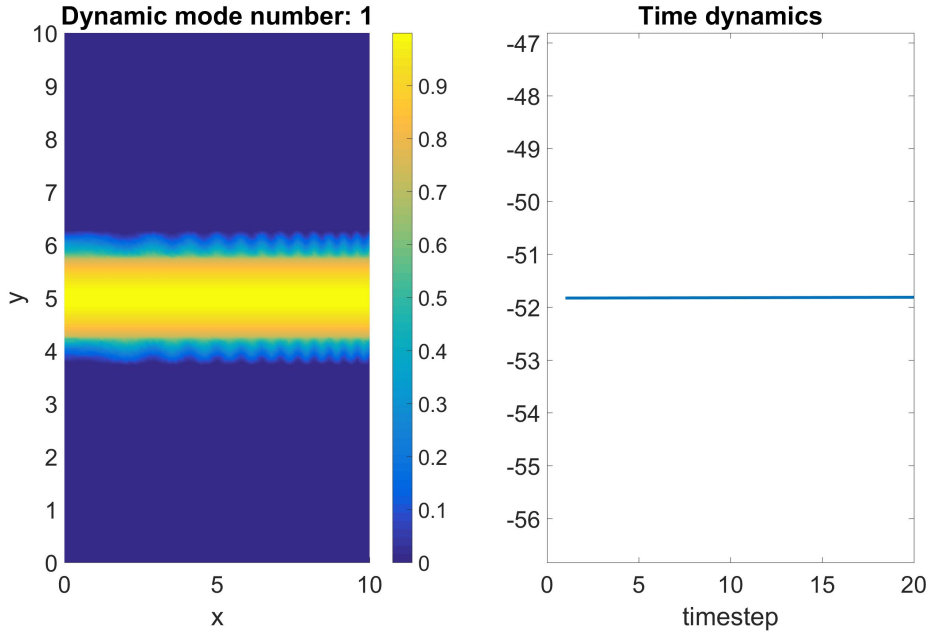


Figure 3.18: Primary dynamic mode and associated time dynamics

The method shows distinct driving modes of the system and accurate representation of the system using low order dynamics (5 modes out of the total 20 were used). A comparison of the superimposed dynamic modes to the initial timestep of original data shows an accurate recreation of the system. The results of the dynamic mode extraction intuitively makes sense as well. It would seem like the primary modes that drive the system would be a constant beam of data in the center and some combination of fluctuations around the edges making up the rest (and ultimately contributing to the jet break up). Clearly the primary mode (mode 1) holds most of the energy (Fig. 3.17), followed by a secondary mode that holds information on the fluctuation of both the top of the jet and sides. The fluctuation is represented

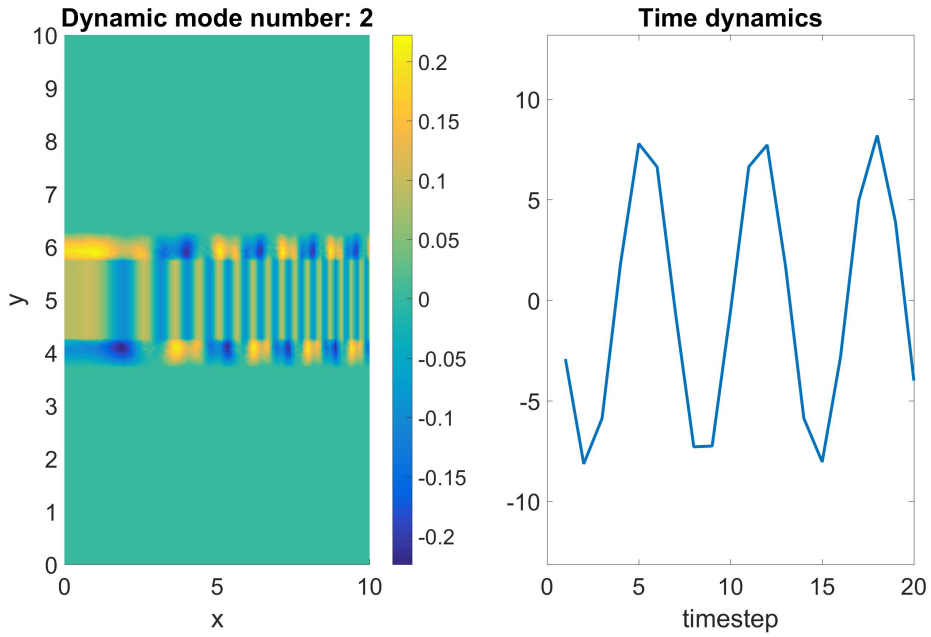


Figure 3.19: Dynamic mode 2 and associated time dynamics

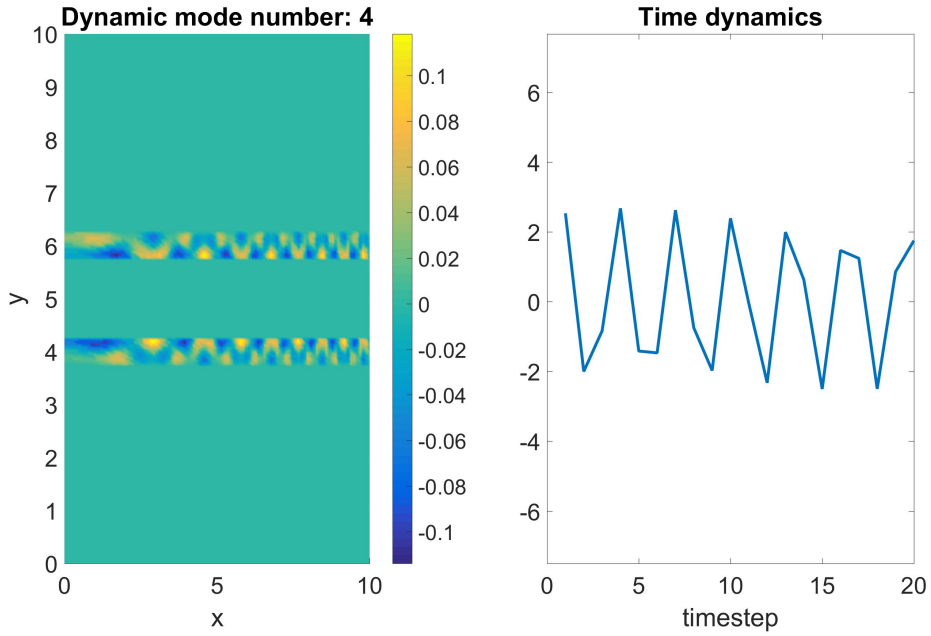


Figure 3.20: Dynamic mode 4 and associated time dynamics

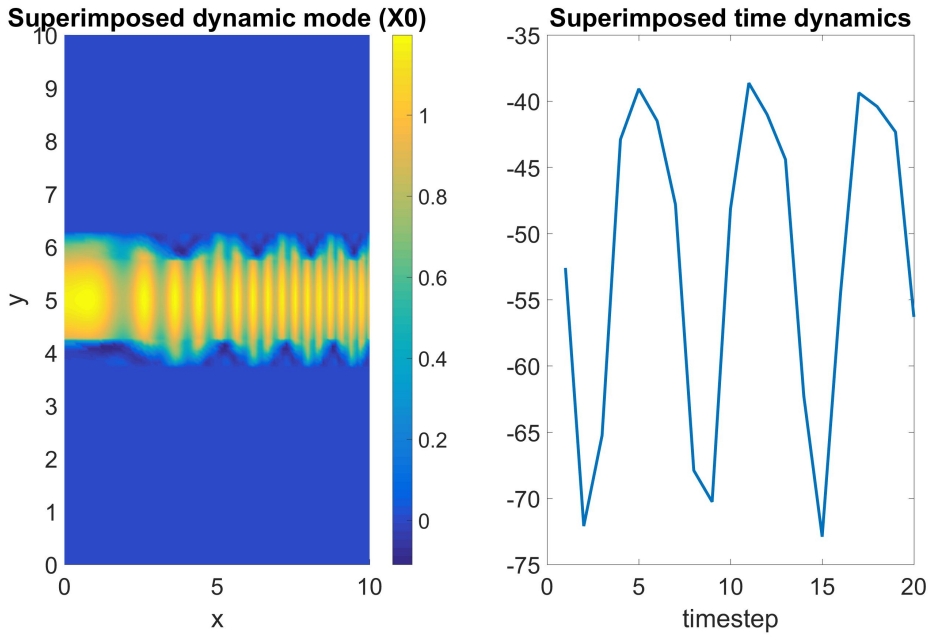


Figure 3.21: Superimposed dynamic mode and associated time dynamics

in the time dynamics, as they are oscillatory. This makes sense because although the side and top of the jet are defined by different parts of the equation driving the system, the data driven model sees fluctuations happening around the edge of the jet and combines them to one fluctuation mode. The subsequent minor modes contribute almost insignificantly, and most are cut off. This is also what is expected to happen in a real jet simulation, with the main stream of data being a cylinder constant in time and the fluctuations causing breakup to be some combination of minor oscillatory modes. It is also possible that the DMD did not directly extract all of individual modes, and that a summation of many low energy modes could be hiding a more significant dynamic of the system. The superimposed dynamic mode provides a proof of concept for more complicated or specific superimpositions. This, along with a convenient comparison to the original data and the extraction of the temporal dynamics of the whole system, was why the superposition of all the test

case non-truncated modes was performed.

The full low order recreation of the system was calculated and visualized. Figures 3.22 and 3.23 show the fifth time step from the diametric and top views. For reference, Fig. 3.23 is shown next to the original data in the same view at the fifth time step (Fig. 3.24). It can again be seen that the low order recreation of the system accurately represents the original high order system. This was expected, and verifies the DMD implementation. Since the original data was a linear system it would make sense that the linear operator would recreate the data with good accuracy. The deviation came from the truncation of modes and the least square mathematics in the SVD and $\boldsymbol{A}$ calculations.

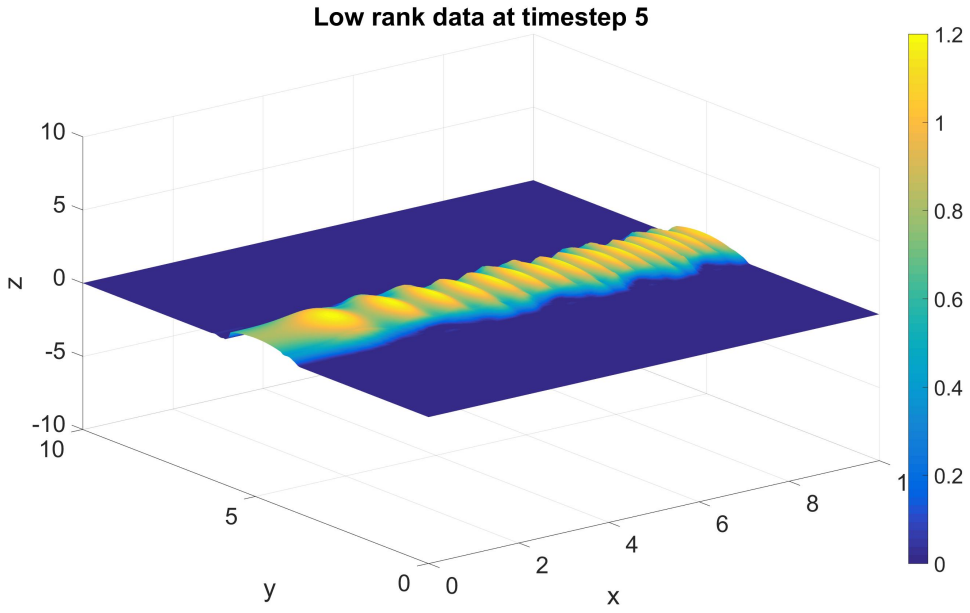


Figure 3.22: "Low rank" data at fifth timestep

The last step was to plot the DMD spectrum. As mentioned in Section 2.3, the DMD spectra is simply a different representation of the eigenvalues of the linear operator $\boldsymbol{A}$ to visualize the stability of the system. Fig. 3.25 shows the real

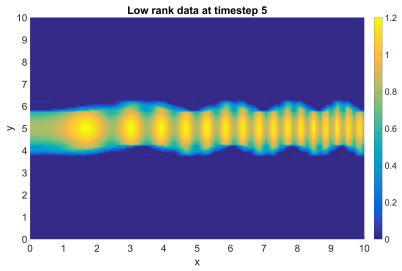


Figure 3.23: "Low rank" data at fifth timestep from top view

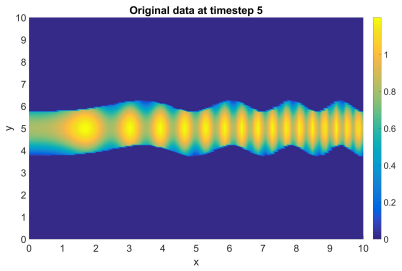


Figure 3.24: Original ("high rank") data at fifth timestep from top view

and imaginary parts of the eigenvalues plotted against each other, Fig. 3.26 shows the logarithmic mapping of the same real vs. imaginary parts of the eigenvalues, and Fig. 3.27 shows the normalized magnitude of the dynamic modes against the imaginary parts of the eigenvalues.

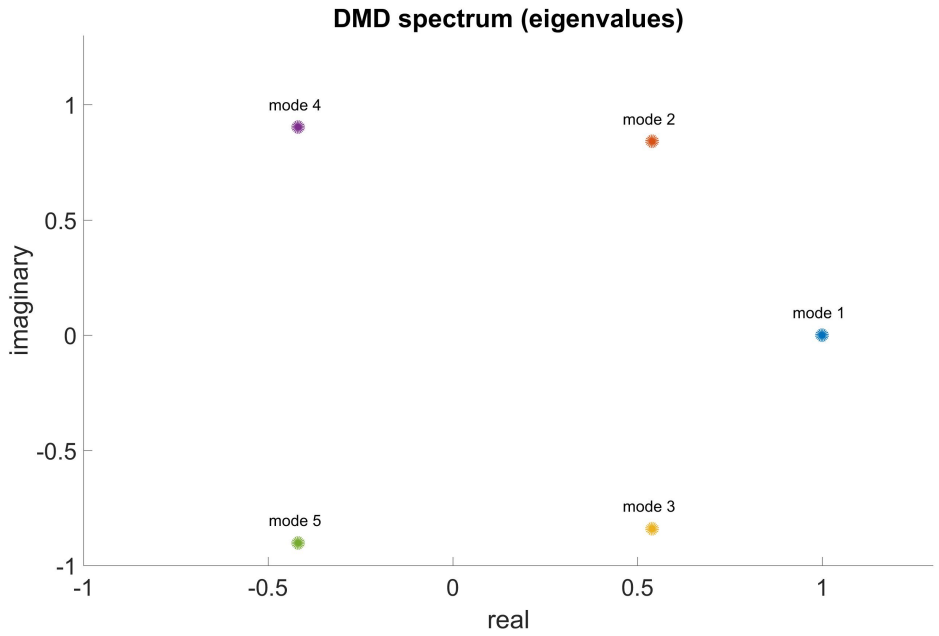


Figure 3.25: Real vs. Imaginary Eigenvalues

As discussed in section 2.3.4, the imaginary parts of the eigenvalues of the

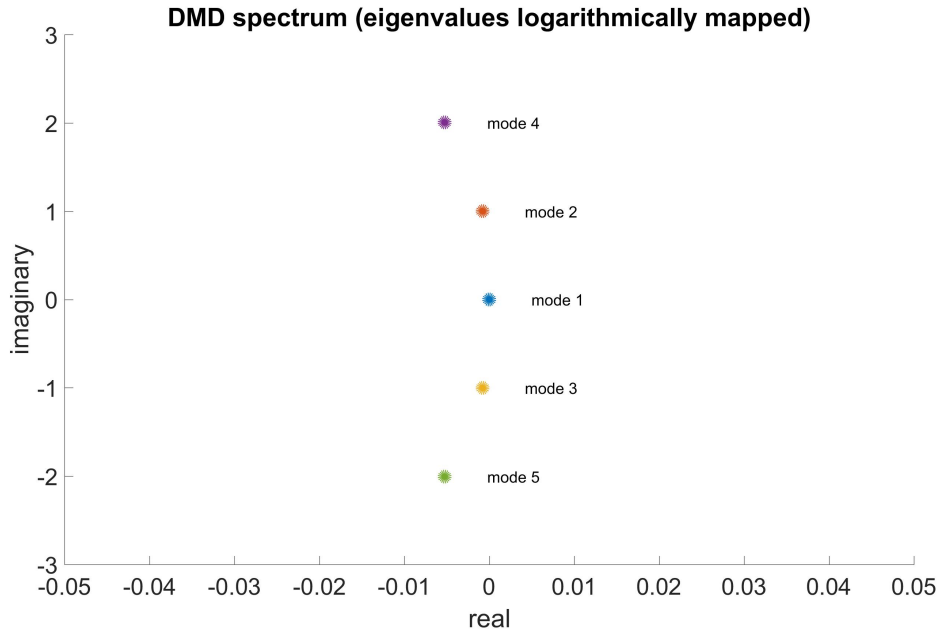


Figure 3.26: Real vs. Imaginary Logarithmic Eigenvalues

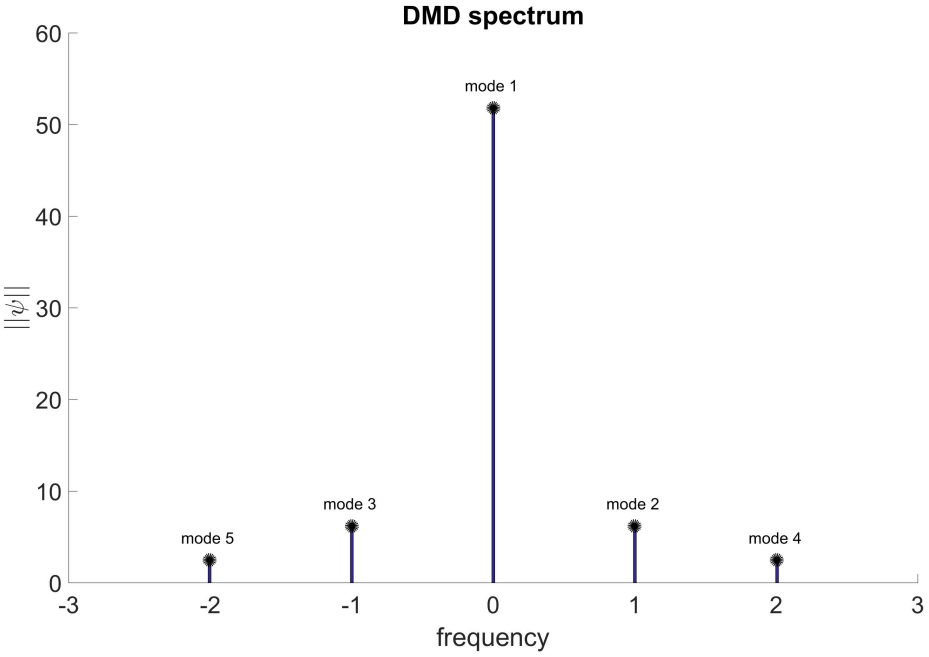


Figure 3.27: Logarithmic Imaginary Eigenvalues vs. Mode Norm

operator A are representative of the frequency of the associated mode in time, and the real parts are representative of the growing or decaying nature of the mode. In a real system, an exactly constant (no growth or decay) time dynamic is very rare. Most modes will have at least some minor growth or decay. The DMD spectrum is another way to see what the time dynamics are already showing. The DMD spectrum here shows stable oscillatory nature of the lower energy modes, and stable non-oscillatory nature of the primary mode.

The test case demonstrated DMD is able to extract dynamic modes that represent coherent structures in the system. While the actual data was sure to be less obvious and more difficult to analyze, the proof of concept was shown in the test case. The test data was not a jet breaking apart as in the atomizing jet DNS, so the physics of the breakup of the jet could not be captured here. But the methodology of combining certain modes to see what affect they had on the system was established and could easily be transferred over to the real simulation to identify the modes causing breakup.

There were two significant differences in the methodology used in the real data compared to the test data that hindered a smooth transition of the DMD algorithm. The first was that the entire process was done without the aid of high performance computing. All the data was generated internal to the DMD algorithm code, so there were no binary files being read in. The second difference in methodology was the matter of a coordinate transfer. The actual data collected would be a function in terms of $\boldsymbol{X}$ (or $\boldsymbol{S}$) and θ, while the test case data was already in Cartesian coordinates and had no need for the coordinate transformation.

CHAPTER FOUR

RESULTS

This section provides the simulation parameters for the case analyzed, results of applying the data extraction tool to the simulation, the results of applying exact DMD to the data, and a discussion of the relevant information extracted.

4.1 Simulation Parameters

The simulation was a three dimensional axial diesel type atomizing jet with a Reynolds number and Weber number of 25000. The Reynolds and Weber numbers were set to encourage turbulence and breakup to test the methodology to extract the primary instability on th liquid core. The axial directions were the X direction.

The non-dimensional domain length was set with respect to an initial jet diameter (Section 4.2) of $D = 5e^{-4}$. Domain lengths of $3D$ were set in the spanwise directions and $12D$ in the axial direction. Mesh sizing in the spanwise directions were set using a 64×64 mesh. An axial mesh size of 256 was used. Note that with this high Reynolds and Weber number on the relatively coarse mesh, the results are not a DNS, but allow the DMD analysis to be tested. Note that even within the "optimal" range of operation the simulation still took well over a full day to deliver its 37 output matrices discussed in section 4.3, and in simulation time the difference was only $7.997e^{-4}$ seconds. This provides good perspective on the need for reduced order modeling.

The time step was set at $1e^{-6}$, although the NGA code uses a variable time stepping technique, so the timestep was not always the same. In fact it was almost never the same.

In regards to the data extraction code from section 3.2, the axial function $\boldsymbol{S}$ was set up with a step size through the entire axial domain to be one step per sub-cell (there are 8 sub-cells per cell, with a length of two sub-cells in the axial direction per cell), and one hundred and eighty steps in the θ direction. The code was called every $2e^{-5}$ seconds of local simulation time. This was determined to be an acceptable frequency from observation of several test and trial simulations. It is important to have an appropriate sampling interval both in the spatial and temporal domains to capture the jet perturbations. Temporally, methods like the Nyquist sampling method have been used [9, 48]. It can be seen that with the sampling frequency mentioned above the propagation of perturbations along the core of the jet are captured effectively.

4.2 Initial Conditions

Shown in Figures 4.1 and 4.2, the interface is initialized as a half sphere in the axial direction with a non-dimensional diameter of $5e^{-4}$. Fig. 4.3 shows that the inflow conditions within the diameter of the semicircle was set as turbulent. These conditions were obtained from a previous simulation of turbulent pipe flow. This was done to simulate flow coming out of a nozzle. Instead of modeling and simulating the complex geometry, the initial conditions described create a similar result. The exact replication of a nozzle is also less important when the focus is to identify properties of liquid given certain turbulent conditions and Reynolds/Weber numbers. It should be noted that since DMD is in fact a form of averaging, it would not yield the desirable results to start collecting data to be analyzed at the initial timestep. To collect the necessary results, the simulation was run at the parameters and initial conditions described until a statistically steady state was reached and the jet extended through the axial domain. The simulation was then stopped and restarted with the surface

extraction tool turned on using the described output frequency.

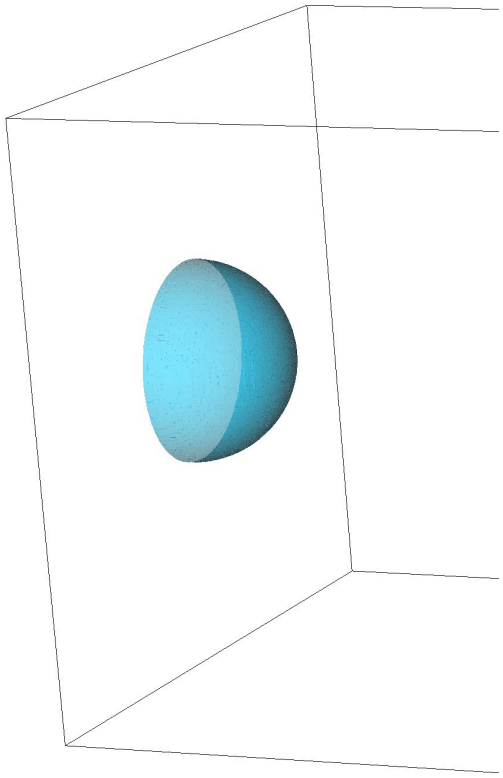


Figure 4.1: Diametric view of the initial shape of the jet

4.3 Simulation

Fig. 4.4 shows the initial timestep from the simulation after a statistically steady state was reached. From this point, 37 snapshots were collected from the simulation. The time between snapshots was small enough to capture the fluctuations propagating along the jets surface. Even though the local simulation run time was very small, the 37 snapshots collected were enough to capture about ten periods of oscillation and safely assume the simulation was in a pseudo-steady state of atomization. The jet running through the domain was acceptable because the goal was to capture the

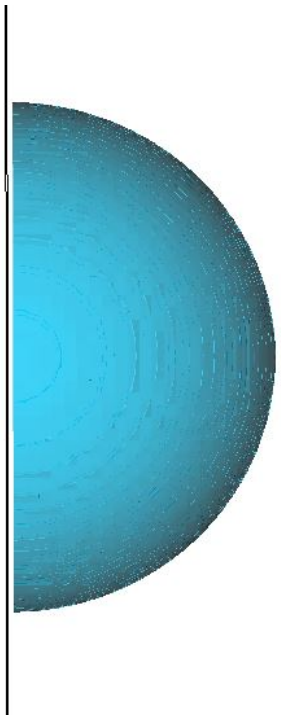


Figure 4.2: Side (X/Z plane) view of the initial shape of the jet

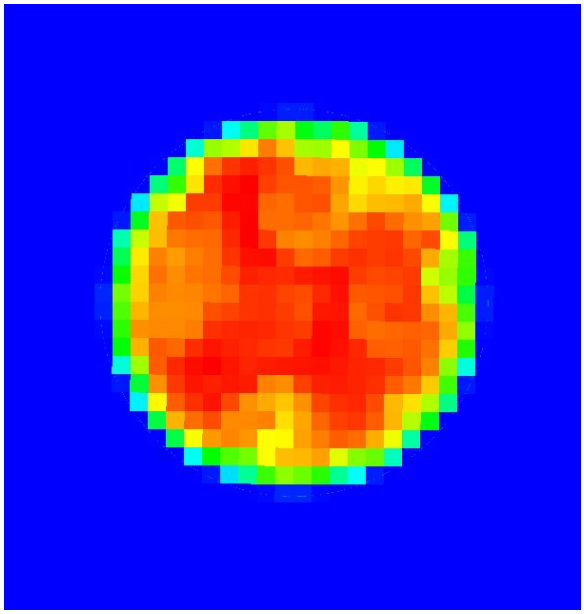


Figure 4.3: Axial (Y/Z plane) view of turbulent inflow conditions

dynamics along the jets core only and make a statement about how they affect the jet breakup. From observation, capturing the dynamics in the initial part of the jet was what is really necessary. It can be seen in Fig. 4.4 that the jet starts to atomize within the domain and Fig. 4.5 shows the data extracted from Fig. 4.4. Towards the end of the domain the jets core starts to become less defined. This is due to the nature of the code that defines the jet structures [20] interacting with the data extraction code, causing a more loose definition of the jets core. With this in mind, the computational domain was still a good measure of the jets core, showing some atomization as well.

4.4 Dynamic Mode Decomposition of Data Extracted from Simulation

The DMD analysis was performed on a slightly truncated version of Fig. 4.5, shown in Fig. 4.6. This was done to capture the jets core only more effectively, and more specifically to only capture the dynamics that are accurately represented by the DMD process. Since the DMD process is using a linear operator to approximate a non-linear system, once the system starts to become defined more dominantly by non-linear or random dynamics (the atomization towards the end of the jet) the DMD less accurately models the system. The results in Fig. 4.5 were truncated at about 65% of the original axial domain. The decision made to truncate the data at 65% is discussed in Section 4.4.1, but comes generally from observation of inaccurate full rank data in Fig. 4.5 and incomplete overly truncated data in Fig. 4.7. A low rank recreation of the data was not necessary as it did not add anything to the investigation of results. The superposition of modes was performed and is all that is necessary to see the dynamics of the system as a whole. Because of this the level of energy allowed in the truncation step of Algorithm 2.3 was set to 100%. By doing this all of the dynamic modes could be observed, no matter how insignificant.

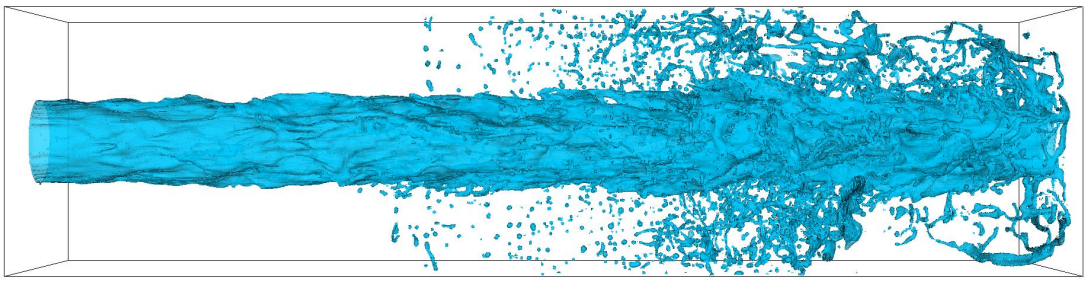


Figure 4.4: First timestep from the psuedo-steady state the data was collected from

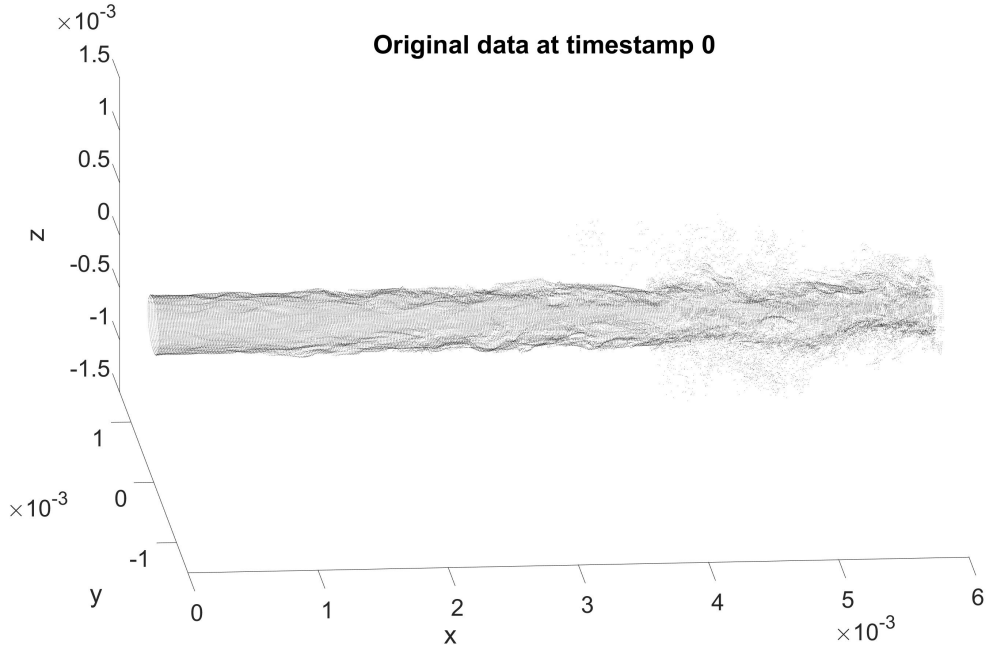


Figure 4.5: Full data collected from the first psuedo-steady state timestep

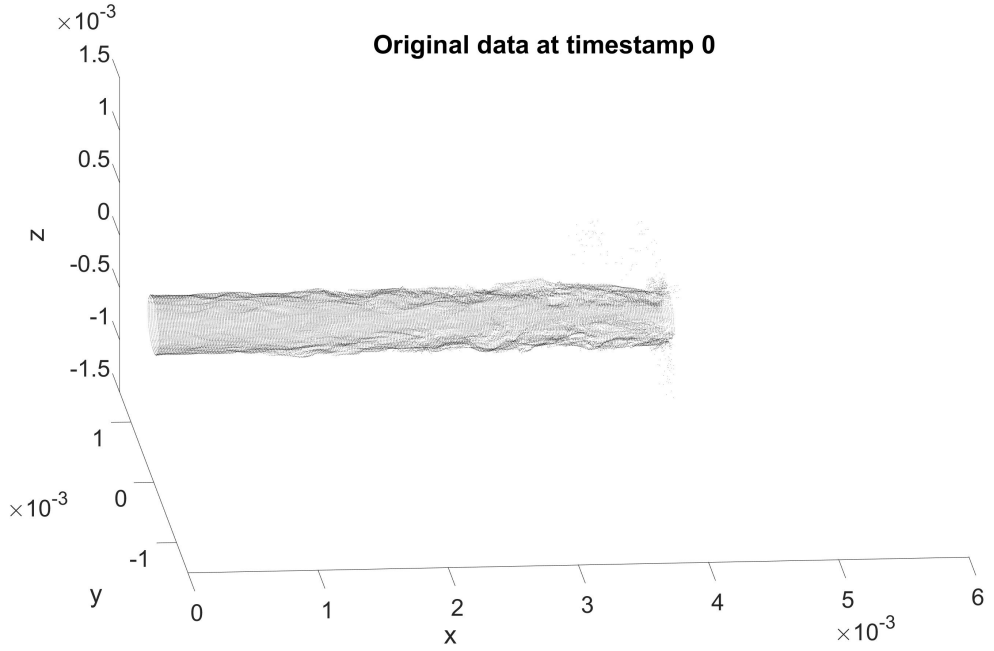


Figure 4.6: 65% truncated data collected from the first psuedo-steady state timestep

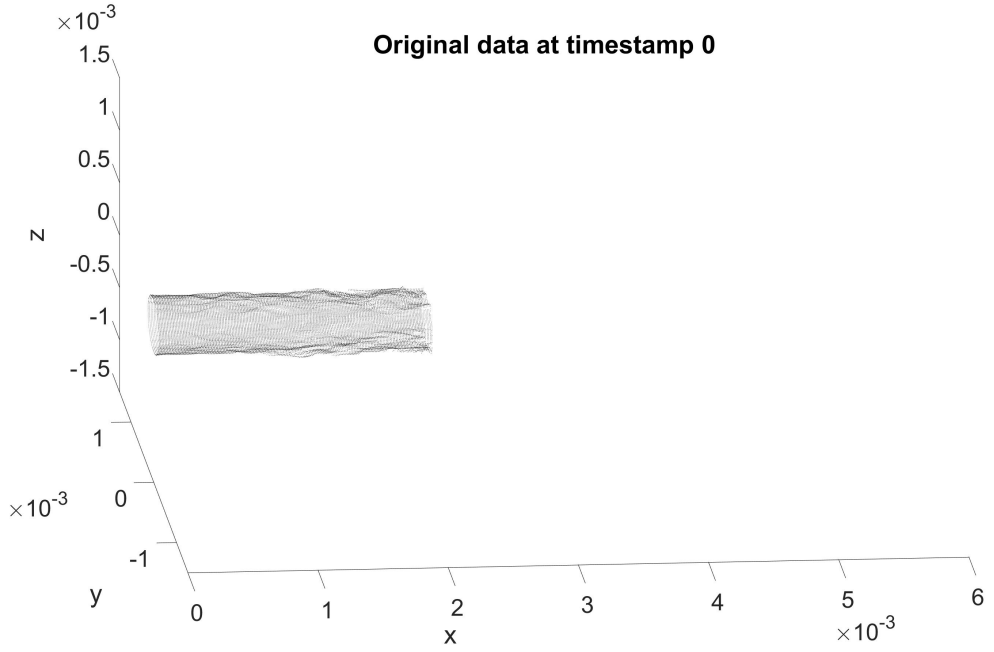


Figure 4.7: 35% truncated data collected from the first psuedo-steady state timestep

Performing the DMD on the 65% truncated results lead to one primary dynamic mode (shown in Fig. 4.9) that carries most of the energy (around 57%) of the system and lots of smaller in magnitude but distinct and repetitive dynamic modes. The categorization of the energy carried by each mode is shown in Fig. 4.8. Two lower energy dynamic modes 17 and 22 are shown in figures 4.12 and 4.15. These modes were chosen due to their distinct structure. There were many other modes of similar structure, and several of less definite structure. Fig. 4.18 shows one such low energy mode. The primary mode is nearly steady in time, while the lower magnitude modes are oscillatory and grow in amplitude along the axial domain. Fig. 4.19 shows the superimposed dynamic modes (representative of the initial timestep of original data) and its associated superimposed time dynamics (representative of the overall time dynamics of the system).

4.4.1 Discussion On Optimal Axial Truncation of Data

The lower energy modes of the same number on the full set of data can be seen in figures 4.13 and 4.16. They can be compared to the modes shown in Figures 4.12 and 4.15. From this comparison it can be seen that the modes are much more distinct and well defined through about 65% of the domain, where after that the amplitude of the modes level off and more random dots appear with no distinct structure to them. This is very obvious when looking at the comparison of the 17th dynamic modes. The less structured region is representative of the DMD providing a less accurate description of the nature of the system towards the end of the domain. The inability of the DMD analysis to capture the full set of data's dynamics can also be seen in the primary mode of the non-truncated data in Fig. 4.10. The primary mode of the jets core should, by intuition and through experience with the test case in section 3.5, be almost

a perfect cylinder that is nearly steady in time. Fig. 4.10 shows the primary mode starting to pick up more perturbations towards the end of the domain. Since the DMD is a type of average/least square fit that is evaluating a non-linear system with a linear operator, the operator does a worse job evaluating the system where non-linear dynamics are dominant. This occurs towards the end of the jet where turbulence and atomization are the driving forces. The data should be truncated before turbulence and atomization begin to dominate the system.

A comparison of the 65% truncated data was also made to a further truncated data set of about 35%. Fig. 4.7 shows the truncation to this level of the original data. The modes of this data set can be seen in figures 4.11, 4.14, and 4.17. The primary mode is similarly shaped and behaves the same through time. This is to be expected by truncating the data further, since the dynamics are now even more linear with less growth in amplitude. The lower energy modes also look and behave similarly to their extended (65%) counterparts, although they do appear to be less dense. This is due to the fact that they are truncated before they reach their maximum amplitude. There is simply less going on in that part of the domain. The dynamics of that part of the domain are captured well, but it are incomplete with respect to the information we are looking for. The desired result is to include as much data as possible from the jets core that can be captured well with DMD. To do this the truncation should occur just before non-linear or random dynamics start driving the system. This occurs at around 65% of the axial domain in this system.

The dynamic modes show what was expected within the system. It might be intuitive to assume that the destabilization modes would be unstable, but since the the whole system is in a pseudo-steady state, the modes will not necessarily be blowing

up. Recall that the DMD does not capture the actual atomization process well due to is highly non-linear or even random nature. Taking the start of atomization out of the analysis leaves us with a more linear system that is more accurately categorized by DMD. So even though the modes are not unstable, they can still account for the initiation of jet breakup further down the jet in the axial direction. As a result of the stability of the modes, the oscillations in time dynamics decay slightly with time, while the magnitude of fluctuations grow over the axial domain. With the knowledge that jet breakup is occurring during the simulation, the non-steady time dynamics and growing sinusoidal amplitude towards the end of the axial domain indicate that the well defined lower magnitude oscillatory modes, or some combinations of multiple repeating lower magnitude modes, are the driving force behind the jet breakup. The same conclusion can not be made for the less distinct low energy modes like mode five in Fig 4.18 because there is no sinusoidal nature to them. When looking at jet breakup, the wavelength and frequency of the breakup modes are of interest. Breakup mode wavelength is indicative of separation droplet size while frequency is indicative of the rate at which the droplets are separated from the core. Comparing the 17th and 22nd modes, mode 17 has a larger wavelength and lower frequency than mode 22. This indicates that the 17th mode is contributing to larger droplets exiting the jets core than the 22nd mode, but mode 22 contributes to more frequent droplet separation.

Plotting the DMD spectrum is another way to see what can be observed from the time dynamics plots. The spectrum visualization was performed with respect to the 65% data. The DMD spectrum shown in Fig. 4.20 includes all of the modes' associated eigenvalues. The reason for plotting every eigenvalue instead of just the ones from the modes shown in figures 4.12 and 4.15 is to observe the nature of the whole system. Fig. 4.20 provides an obvious clue to what the system is doing. It can be seen that

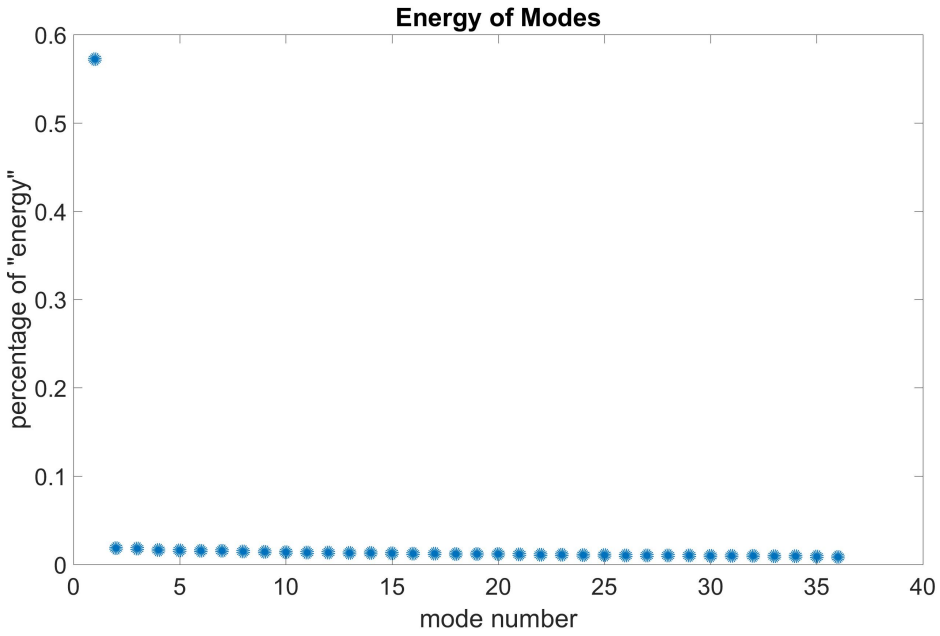


Figure 4.8: Energy carried by modes of 65% truncated data

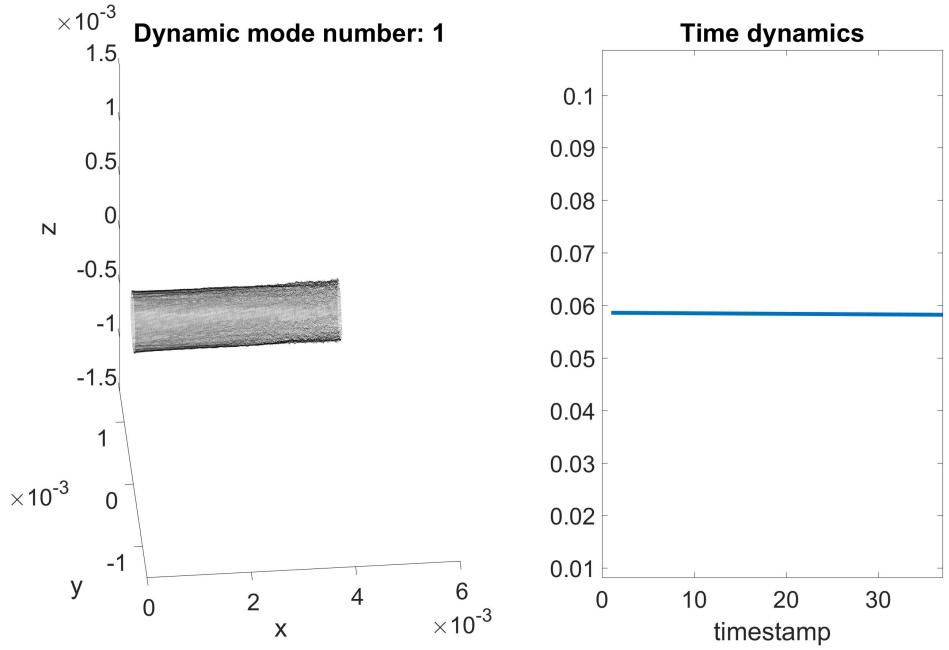


Figure 4.9: 65% truncated data's primary dynamic mode and its time dynamics

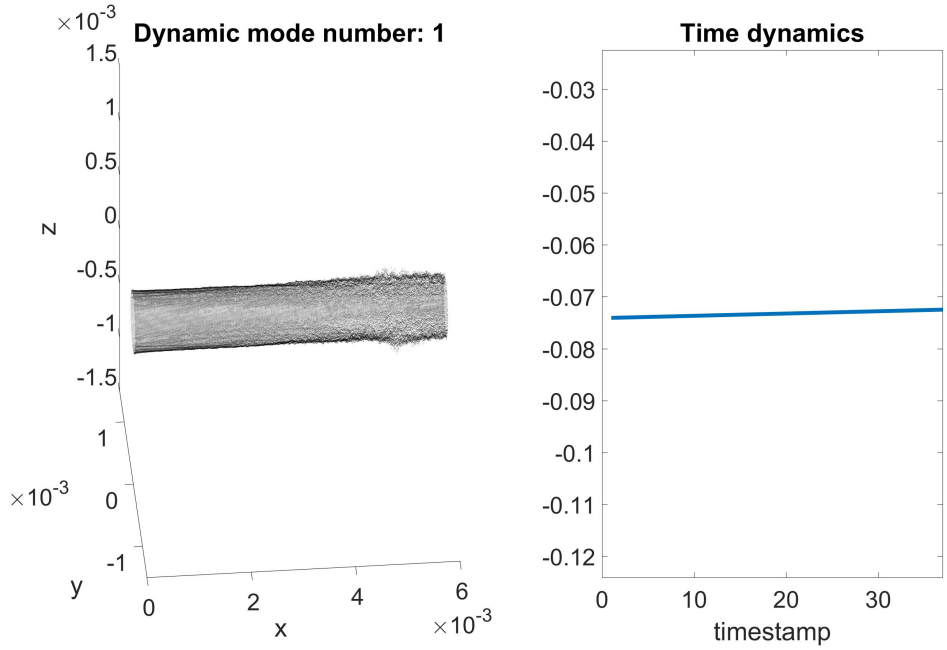


Figure 4.10: Full data's primary dynamic mode and its time dynamics

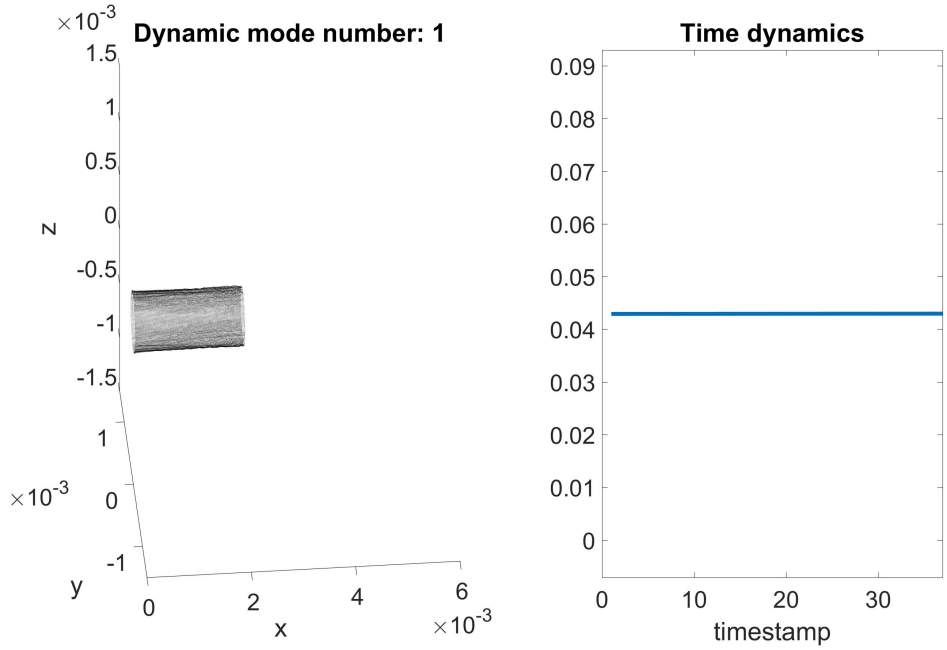


Figure 4.11: 35% truncated data's primary dynamic mode and its time dynamics

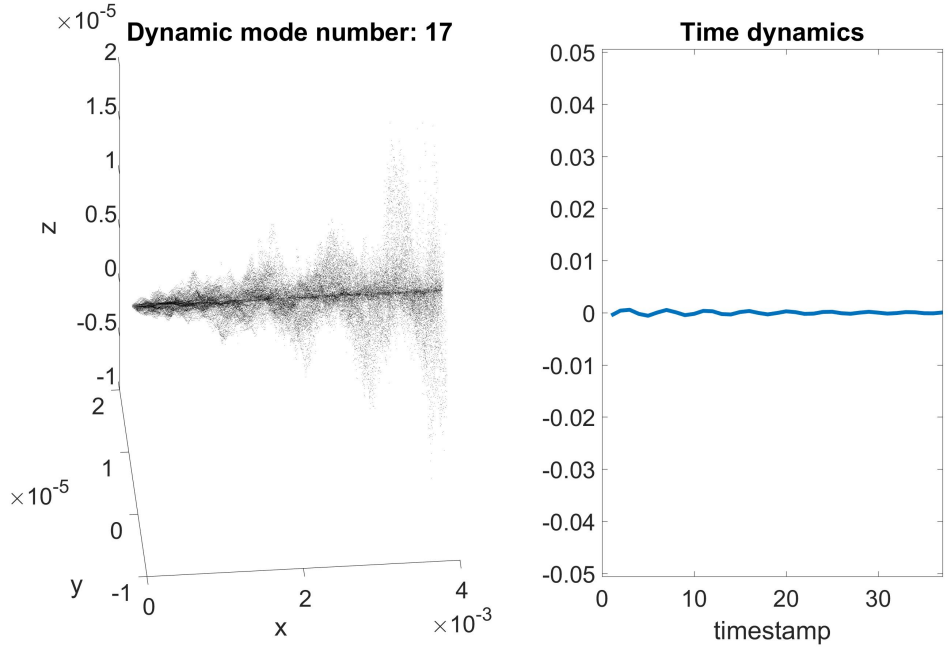


Figure 4.12: 65% truncated data's 17th dynamic mode and its time dynamics

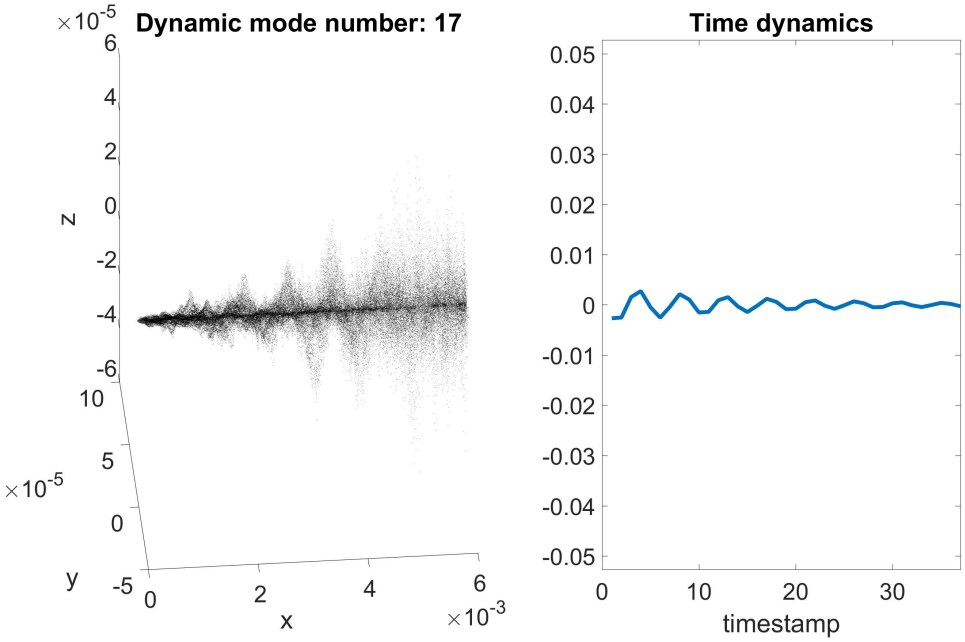


Figure 4.13: Full data's 17th dynamic mode and its time dynamics

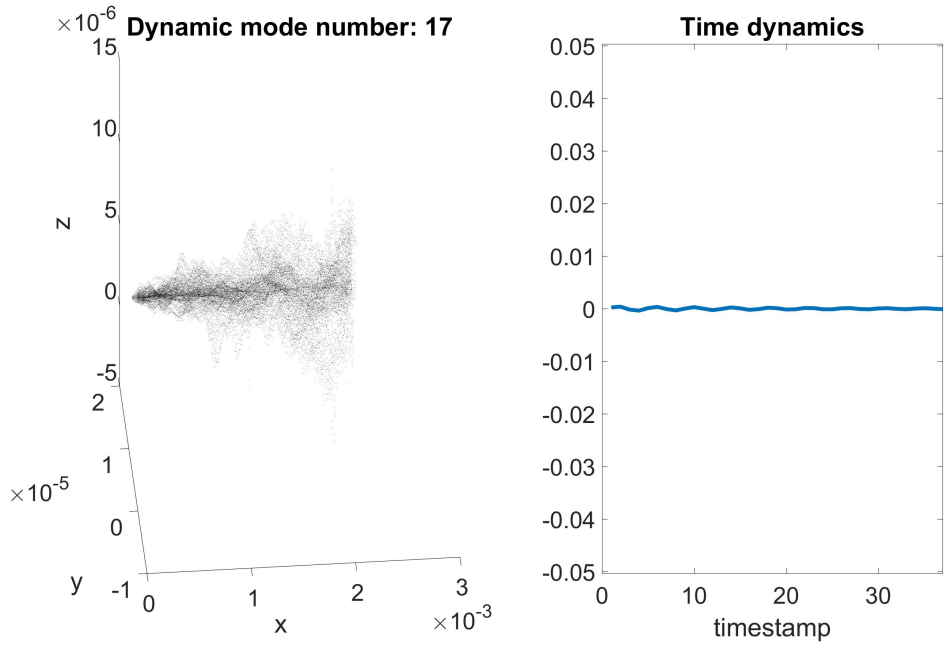


Figure 4.14: 35% truncated data's 17th dynamic mode and its time dynamics

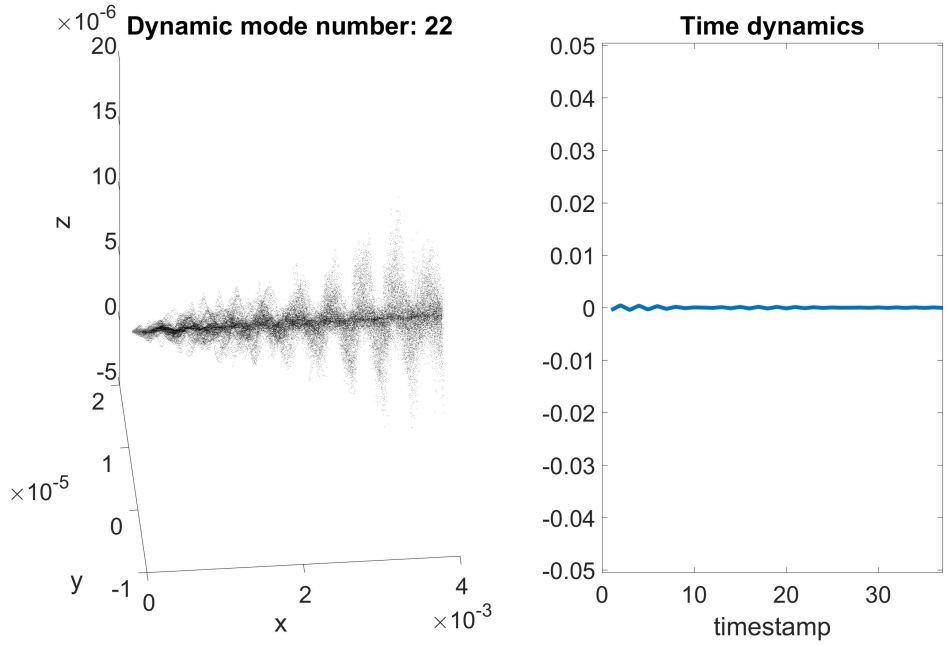


Figure 4.15: 65% truncated data's 22nd dynamic mode and its time dynamics

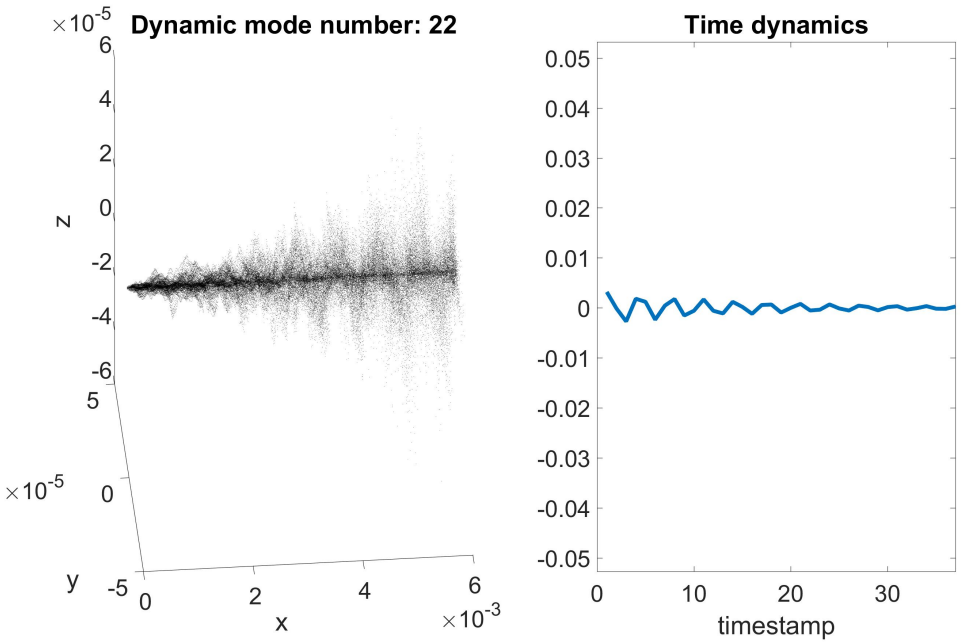


Figure 4.16: Full data's 22nd dynamic mode and its time dynamics

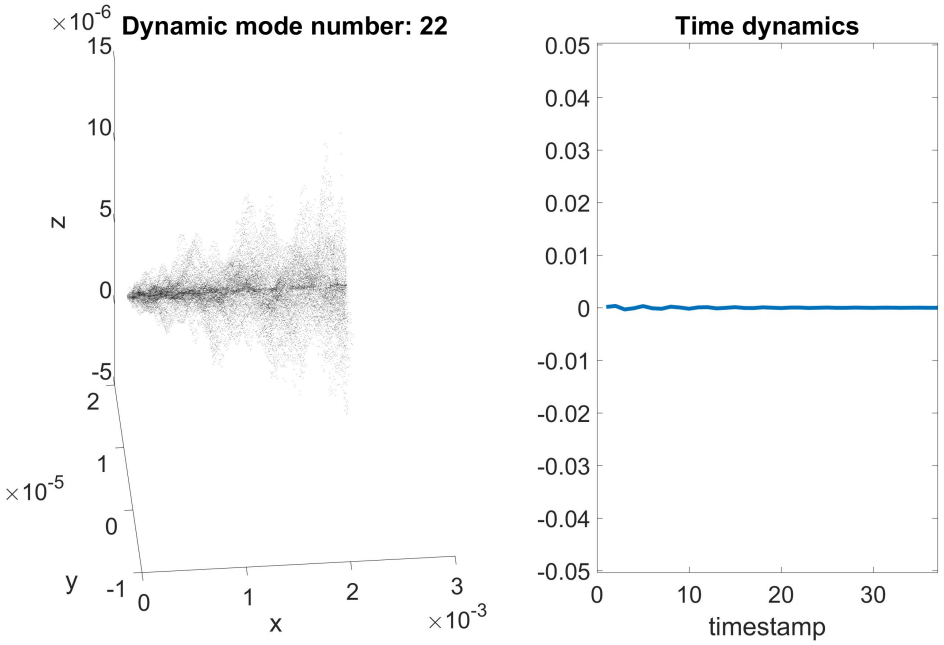


Figure 4.17: 35% truncated data's 22nd dynamic mode and its time dynamics

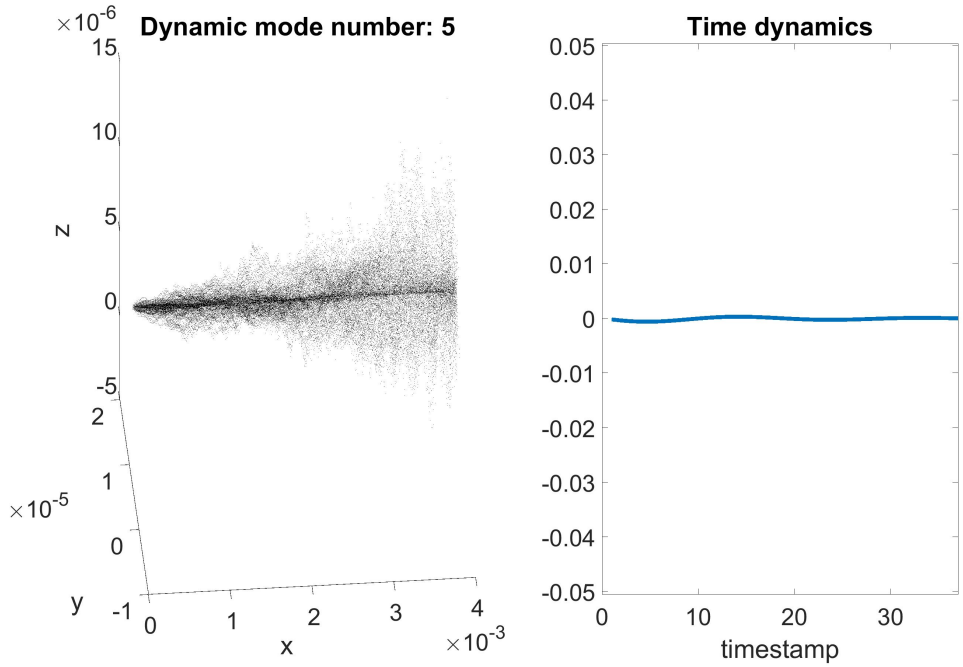


Figure 4.18: 65% truncated data's 5th dynamic mode and its time dynamics

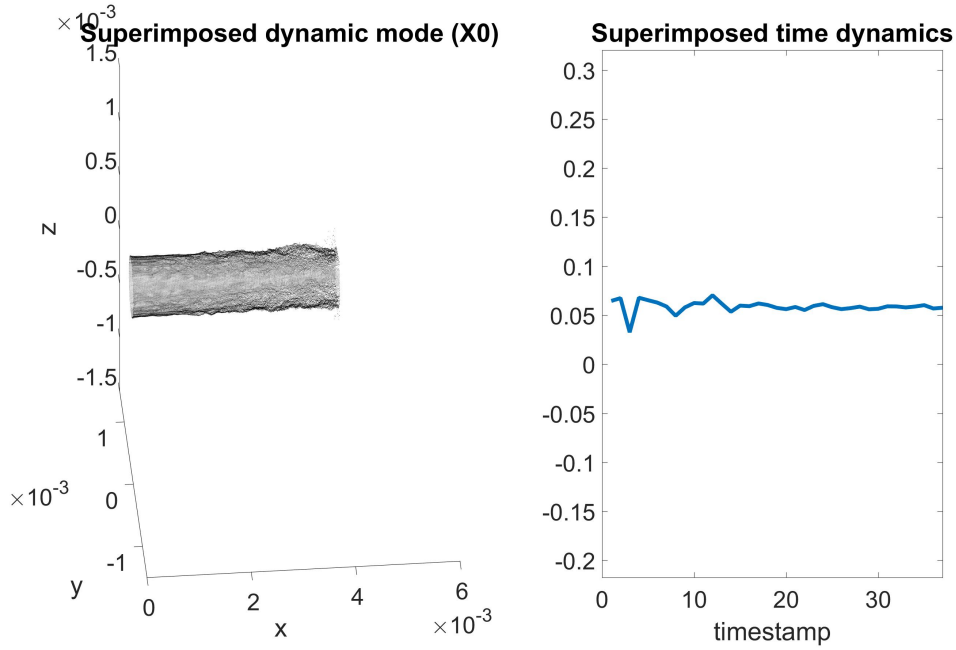


Figure 4.19: 65% truncated data's superimposed dynamic mode and its time dynamics

every mode has a eigenvalue magnitude close to or less than one. The primary mode has no imaginary part and a magnitude of nearly one. This is a confirmation of the observation made about the non-time-varying nature of the primary mode. It very nearly does not grow or decay, and the imaginary part of its eigenvalue is zero, indicating no oscillatory nature. All other modes have some combination of decay and oscillations. This is again a confirmation of the observations of the lower magnitude modes in figures 4.12 and 4.15. The repetitive nature of complex eigenvalues can again be seen as the mapping in Fig. 4.20 is reflected about the 0 imaginary line. The logarithmic mapping of the spectrum in Fig. 4.21 and the norm vs. frequency spectrum in Fig. 4.22 do not account for much new information, but are shown for conventions sake.

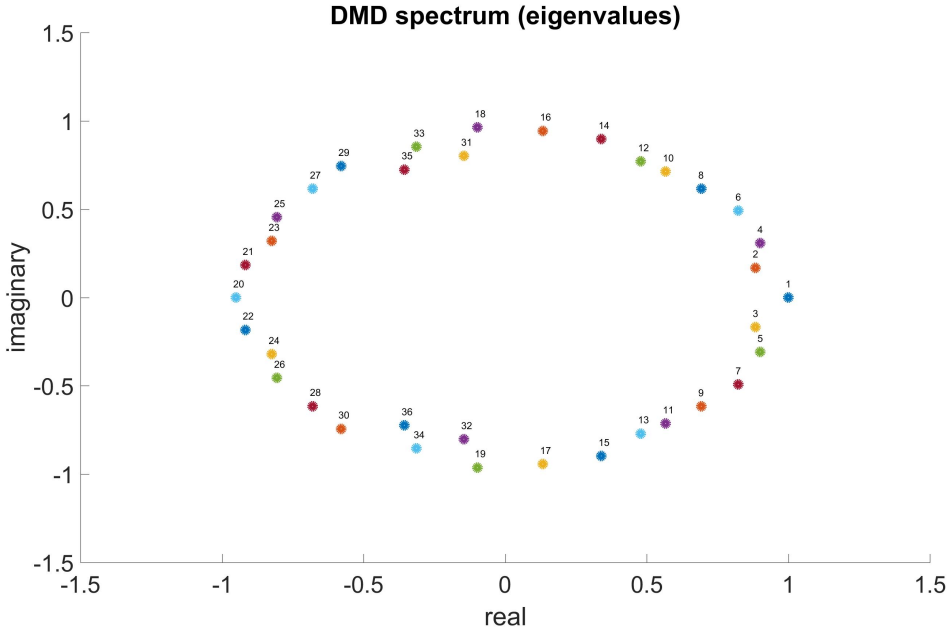


Figure 4.20: Real vs. Imaginary Eigenvalues

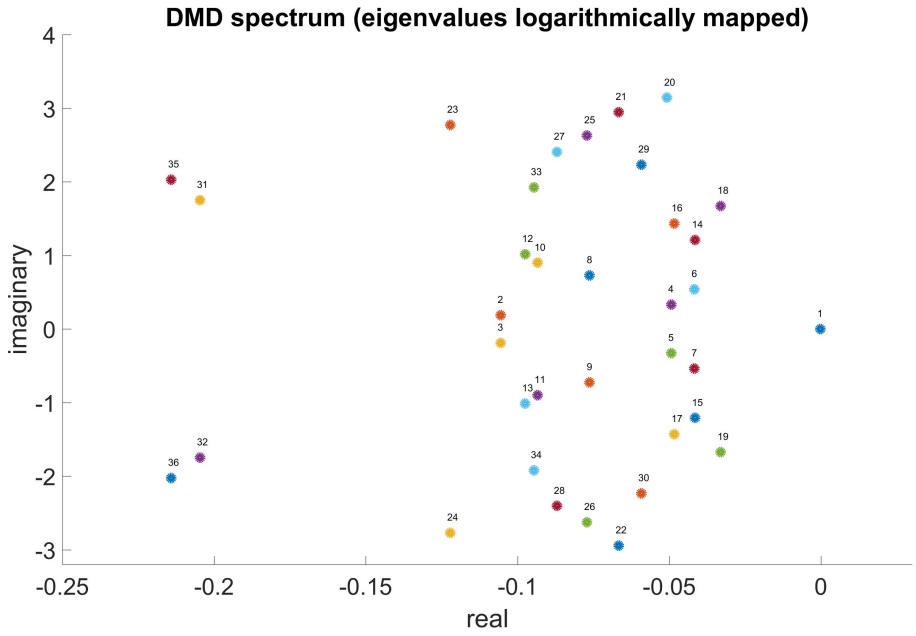


Figure 4.21: Real vs. Imaginary Logarithmic Eigenvalues

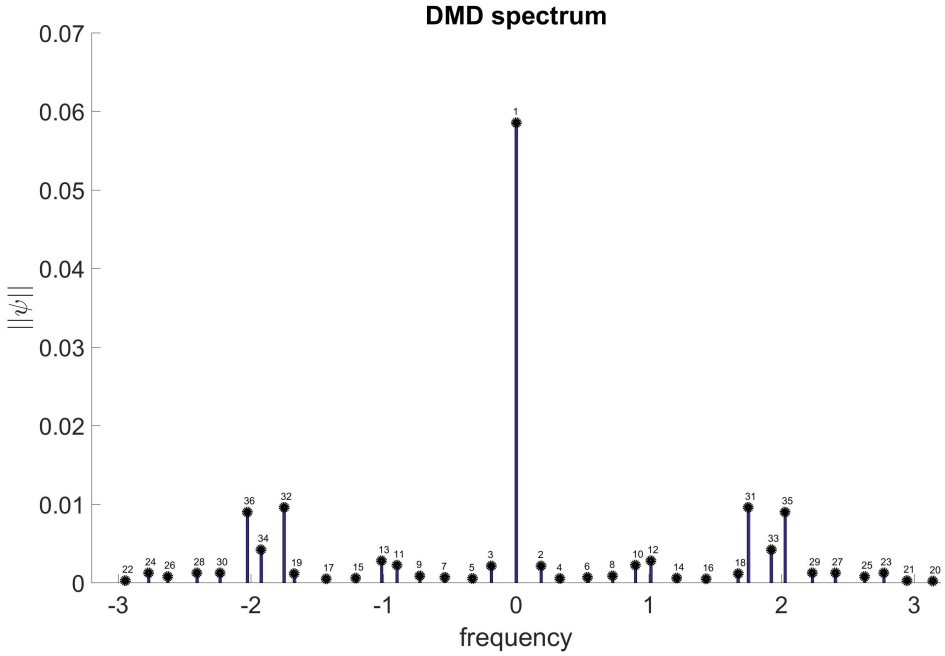


Figure 4.22: Logarithmic Imaginary Eigenvalues vs. Mode Norm

CHAPTER FIVE

CONCLUSION

In this work tools were developed with the goal of better understanding the physics that contribute to an atomizing jets breakup. A code was written and integrated into the established NGA platform that represents an atomizing jets liquid core in two dimensional cylindrical coordinates. The code calculates the jets core as radii that are a function of the jets center $\boldsymbol{S}$ and some azimuthal angle θ from ray intersections with gas/liquid interface planes. This function represents the jets liquid core well and provides a convenient data set for dynamic mode decomposition. Collecting data using this code from a coarse DNS simulation of an axial diesel atomizing jet and performing dynamic mode decomposition on it lead to some important characterizations of the jets driving dynamics. It was necessary to truncate the original data before the DMD was performed in order to capture the dynamics of the jets core more effectively, but once that data was analyzed multiple distinct modes were categorized. The first (primary) mode was expectantly dominant in system energy and non-varying in time. Other modes were less significant in magnitude and energy, but were oscillatory in time and their distinct sinusoidal spatial structure grew in amplitude through the axial domain. These modes were identified as the modes contributing to the initiation of jet breakup. The work presented furthers a rapidly growing field of applying reduced order modeling to DNS simulations. Applying DMD to numeric three dimensional positional data (rather than velocity or vorticity) allows for a direct investigation into the implementation of sub-grid breakup models for lower order simulations by taking the wavelength and frequency information from the identified breakup modes and relating them to droplet size and

rate of droplets exiting the core, respectively. It also opens the door for investigation of relationships between the dynamic mode behavior of interacting variables of interest. Breaking up different variables in an atomizing jet into modes with the knowledge of how those modes directly affect the system and other variables could prove to be significant in the implementation of a new type of jet breakup model for lower order simulations. Discussed in chapter 6, there are several other potential uses for this research that could further the understanding of the jet breakup process and aid in the implementation of more accurate lower order models.

CHAPTER SIX

FUTURE WORK

In directly continuing the work that was provided in this paper several improvements and capabilities could be added:

1: The application to a jet in crossflow would be useful for real world scenarios. As mentioned in section 3.4, the math and code are already written to support this, but there would have to be some rearranging of modules within the computational platform in order to obtain the desired jet center location function.

2: Further integration of the DMD algorithm into the NGA platform could be achieved by writing the DMD code completely in Fortran and output compatible visualization files so DMD could automatically be called through the input file.

3: To improve the uniqueness of each mode obtained from DMD a different DMD method called multi-resolution DMD (MrDMD) could be investigated for implementation. MrDMD implements a hierarchy of frequency categorization to better identify modes with distinct frequencies. Further reading on the subject is available in [30].

Some other ideas for the continuation of work non-directly:

1: A possible relationship that holds true across different types of atomization jets between some characteristic of the destabilization modes (frequency, perhaps) and something like Reynolds/Weber number ratio could be investigated. This could be very useful in the implementation of this breakup information into

lower order simulations' break up models, particularly if a relationship was discovered that was not scenario specific. This would also lead to some important categorizations about jet breakup in the research field of destabilization. The non-existence of such a relationship would also be a meaningful categorization of the destabilization process.

2: Implementation of feedback control. If the recreated (by DMD) low order simulation could be made to automatically detect modes of destabilization based on some pre-defined categorization of perfect breakup, then it may also be possible to have the recreated (by DMD) low order simulation optimize the input parameters for the perfect breakup. This would be much faster than running an optimization (even if it was automated) for the high order DNS. To do this, capability would have to be added to account for the initial stages of jet propagation. This is because with the current method of DMD being used, the low rank representation is a kind of average, and would include jet propagation in that average. This is not an issue if the sampling is being taken at steady state, but if the input parameters and inlet conditions are to be altered based off of the low order simulation, the new low order simulation would also have to include accurate low order initial conditions. The biggest issue with the idea of low order feedback control is that the low order simulation is not a responsive system. If something is changed in the simulation it will not respond accordingly, it will still act with the dynamics that defined the system prior to adjustment. There is some work being done in the category of implementing a type of control or feedback control to low order models by [3, 39].

3: One other possible solution to create responsive reduced order models based on DMD (or even stand-alone data driven reduced order models) is through

machine learning. Through the collection of data from enough low order simulations of varying parameters, it could be possible to create a machine learning algorithm to feed off of the data. This could create a responsive system that does not need an initial high order simulation to collect data from or the governing Navier-Stokes equations to run. The optimization could then be done directly. Work is being done in this field at the University of Washington [2,15].

REFERENCES CITED

[1] Hyalite Cluster - Research Cyberinfrastructure | Montana State University. http://www.montana.edu/uit/rci/hyalite/.

[2] Machine Learning | Kutz Research Group. http://faculty.washington.edu/kutz/page1/page24/, author = J. Nathan Kutz,.

[3] L. Alvergue, H. Babaee, G. Gu, and S. Acharya. Feedback Stabilization of a Reduced-Order Model of a Jet in Crossflow. *AIAA Journal*, 53(9):2472–2481, Sept. 2015.

[4] Anne Greenbaum. *Iterative Methods for Solving Linear Systems*. SIAM, 1997.

[5] M. Arienti and M. C. Soteriou. Time-resolved proper orthogonal decomposition of liquid jet dynamics. *Physics of Fluids*, 21(11):112104, Nov. 2009.

[6] W. E. ARNOLDI. THE PRINCIPLE OF MINIMIZED ITERATIONS IN THE SOLUTION OF THE MATRIX EIGENVALUE PROBLEM. page 13.

[7] B. O. Koopman. HAMILTONIAN SYSTEMS AND TRANSFORMATIONS IN HILBERT SPACE. *Proceedings of the National Academy of Sciences*, 17(5):315–318, May 1931.

[8] S. Bagheri, P. Schlatter, P. J. Schmid, and D. S. Henningson. Global stability of a jet in crossflow. *Journal of Fluid Mechanics*, 624:33, Apr. 2009.

[9] S. L. Brunton, J. L. Proctor, and J. N. Kutz. Compressive sampling and dynamic mode decomposition. page 12.

[10] H. Choi and P. Moin. Effects of the Computational Time Step on Numerical Solutions of Turbulent Flow. *Journal of Computational Physics*, 113(1):1–4, 1994.

[11] C. Chryssakis and D. N. Assanis. A UNIFIED FUEL SPRAY BREAKUP MODEL FOR INTERNAL COMBUSTION ENGINE APPLICATIONS. *Atomization and Sprays*, 18(5):375–426, 2008.

[12] T. Colonius and K. Taira. A fast immersed boundary method using a nullspace approach and multi-domain far-field boundary conditions. *Computer Methods in Applied Mechanics and Engineering*, 197(25-28):2131–2146, Apr. 2008.

[13] O. Desjardins, G. Blanquart, G. Balarac, and H. Pitsch. High order conservative finite difference scheme for variable density low Mach number turbulent flows. *Journal of Computational Physics*, 227(15):7125–7159, July 2008.

[14] O. Desjardins, V. Moureau, and H. Pitsch. An accurate conservative level set/ghost fluid method for simulating turbulent atomization. *Journal of Computational Physics*, 227(18):8395–8416, Sept. 2008.

[15] T. Duriez, S. Brunton, and B. R. Noack. *Machine Learning Control Taming Nonlinear Dynamics and Turbulence*. Fluid Mechanics and Its Applications. Springer International Publishing, 2017.

[16] W. Edwards, L. Tuckerman, R. Friesner, and D. Sorensen. Krylov Methods for the Incompressible Navier-Stokes Equations. *Journal of Computational Physics*, 110(1):82–102, Jan. 1994.

[17] F. X. Tanner. Liquid Jet Atomization and Droplet Breakup Modeling of Non-Evaporating Diesel Fuel Sprays. *SAE Technical Papers*, 1997.

[18] T. F. Fric and A. Roshko. Vortical structure in the wake of a transverse jet. *Journal of Fluid Mechanics*, 279(-1):1, Nov. 1994.

[19] R. A. Friesner, L. S. Tuckerman, B. C. Dornblaser, and T. V. Russo. A method for exponential propagation of large systems of stiff nonlinear differential equations. *Journal of Scientific Computing*, 4(4):327–354, Dec. 1989.

[20] M. Herrmann. A parallel Eulerian interface tracking/Lagrangian point particle multi-scale coupling procedure. *Journal of Computational Physics*, 229(3):745–759, Feb. 2010.

[21] J. Nathan Kutz. Lecture: Principal Componenet Analysis (PCA). `https://www.youtube.com/channel/UCEirPnFv_2QbvzrM67SnKPA`.

[22] J. Nathan Kutz. Lecture: The Singular Value Decomposition (SVD). `https://www.youtube.com/channel/UCEirPnFv_2QbvzrM67SnKPA`.

[23] J. Nathan Kutz. *Data-Driven Modeling & Scientific Computation: Methods for Complex Systems & Big Data*. Oxford University Press, Sept. 2013.

[24] J. Nathan Kutz, Steven L. Brunton, Bingi W. Brunton, and Joshua L. Proctor. *Dynamic Mode Decomposition: Data-Driven Modeling of Complex Systems*. Number 149 in Other titles in applied mathematics. SIAM, Philadelphia, 2016.

[25] Joshua L. Proctor. Dynamic Mode Decomposition with control. `https://www.bing.com/videos/search?q=dynamic+mode+decomposition&&view=detail&mid=498AE148C58C45C3BE3D498AE148C58C45C3BE3D&rvsmid=DCD9C65F848CE076F8CADCD9C65F848CE076F8CA&fsscr=0&FORM=VDFSRV`.

[26] V. Kirsch, M. A. Reddemann, J. Palmer, and R. Kneer. MICROSCOPIC ANALYSIS OF TRANSVERSAL AND AZIMUTHAL SURFACE DISTURBANCES OF HIGH-PRESSURE SPRAYS. *Atomization and Sprays*, 27(9):791–805, 2017.

[27] B. O. Koopman and J. v. Neumann. Dynamical Systems of Continuous Spectra. *Proceedings of the National Academy of Sciences*, 18(3):255–263, Mar. 1932.

[28] S. Kostka, A. C. Lynch, B. C. Huelskamp, B. V. Kiel, J. R. Gord, and S. Roy. Characterization of flame-shedding behavior behind a bluff-body using proper orthogonal decomposition. *Combustion and Flame*, 159(9):2872–2882, Sept. 2012.

[29] J. N. Kutz, S. L. Brunton, D. M. Luchtenburg, C. W. Rowley, and J. H. Tu. On dynamic mode decomposition: Theory and applications. *Journal of Computational Dynamics*, 1(2):391–421, Dec. 2014.

[30] J. N. Kutz, X. Fu, and S. L. Brunton. Multiresolution Dynamic Mode Decomposition. *SIAM Journal on Applied Dynamical Systems*, 15(2):713–735, Jan. 2016.

[31] Lloyd N. Trefethen and David Bau, III. *Numerical Linear Algebra*. SIAM, Philadelphia, 1997.

[32] I. Mezi. Analysis of Fluid Flows via Spectral Properties of the Koopman Operator. *Annual Review of Fluid Mechanics*, 45(1):357–378, Jan. 2013.

[33] Nathan Kutz. PDE Koopman. `https://www.youtube.com/watch?v=jHr84z5lI6E`.

[34] Nathan Kutz. ROM Introduction. `https://www.youtube.com/channel/UCoUOaSVYkTV6W4uLvxvgiFA`.

[35] B. R. Noack, G. Tadmor, and M. Morzynski. Actuation models and dissipative control in empirical Galerkin models of fluid flows. In *Proceedings of the 2004 American Control Conference*, volume 6, pages 5722–5727, 2004.

[36] M. Owkes and O. Desjardins. A computational framework for conservative, three-dimensional, unsplit, geometric transport with application to the volume-of-fluid (VOF) method. *Journal of Computational Physics*, 270:587–612, Aug. 2014.

[37] M. Owkes and O. Desjardins. A mass and momentum conserving unsplit semi-Lagrangian framework for simulating multiphase flows. *Journal of Computational Physics*, 332:21–46, Mar. 2017.

[38] R. Penrose and J. A. Todd. On best approximate solutions of linear matrix equations. *Mathematical Proceedings of the Cambridge Philosophical Society*, 52(01):17, Jan. 1956.

[39] J. L. Proctor, S. L. Brunton, and J. N. Kutz. Dynamic mode decomposition with control. page 10.

[40] W. Qi, W. Zhang, P. Ming, M. Jia, and Y. Peng. NUMERICAL SIMULATION OF HIGH-PRESSURE FUEL SPRAY BY USING A NEW HYBRID BREAKUP MODEL. *Atomization and Sprays*, 27(12):999–1023, 2017.

[41] Richard Haberman. *Applied Partial Differential Equations with Fourier Series and Boundary Value Problems*. Pearson Prentice Hall, New Jersey, 4th edition, 2004.

[42] Rolf D. Reitz. Atomization Models. `https://www.erc.wisc.edu/documents/Short_course_3.pdf`, 2018.

[43] C. W. Rowley, I. Mezi, S. Bagheri, P. Schlatter, and D. S. Henningson. Spectral analysis of nonlinear flows. *Journal of Fluid Mechanics*, 641:115, Dec. 2009.

[44] A. Ruhe. Rational Krylov sequence methods for eigenvalue computation. *Linear Algebra and its Applications*, 58:391–405, Apr. 1984.

[45] Y. Saad. Krylov subspace methods for solving large unsymmetric linear systems. *Mathematics of Computation*, 37(155):105–105, Sept. 1981.

[46] P. Schlatter, S. Bagheri, and D. S. Henningson. Self-sustained global oscillations in a jet in crossflow. *Theoretical and Computational Fluid Dynamics*, 25(1-4):129–146, June 2011.

[47] P. J. Schmid. Dynamic mode decomposition of numerical and experimental data. *Journal of Fluid Mechanics*, 656:5–28, Aug. 2010.

[48] P. J. Schmid. Application of the dynamic mode decomposition to experimental data. *Experiments in Fluids*, 50(4):1123–1130, Apr. 2011.

[49] P. J. Schmid, L. Li, M. P. Juniper, and O. Pust. Applications of the dynamic mode decomposition. *Theoretical and Computational Fluid Dynamics*, 25(1-4):249–259, June 2011.

[50] P. J. Schmid, D. Violato, and F. Scarano. Decomposition of time-resolved tomographic PIV. *Experiments in Fluids*, 52(6):1567–1579, June 2012.

[51] Stephen J. Chapman. *Fortran 90/95 for Scientists and Engineers*. Tom Casson, 1st edition, 1998.

[52] K. Taira and T. Colonius. The immersed boundary method: A projection approach. *Journal of Computational Physics*, 225(2):2118–2137, Aug. 2007.

[53] W. Richard Stevens. *UNIX Network Programming: Interprocess Communications*, volume 2. Prentice Hall, 2 edition, Sept. 1998.

[54] Y. Wang, W. Geun Lee, R. Reitz, and R. Diwakar. Numerical Simulation of Diesel Sprays Using an Eulerian-Lagrangian Spray and Atomization (ELSA) Model Coupled with Nozzle Flow. *SAE Technical Papers*, 2011.

[55] M. O. Williams, I. G. Kevrekidis, and C. W. Rowley. A DataDriven Approximation of the Koopman Operator: Extending Dynamic Mode Decomposition. *Journal of Nonlinear Science*, 25(6):1307–1346, Dec. 2015.

APPENDICES

APPENDIX A

EXTRACTED POSITIONAL DATA CODE

WRITTEN IN FORTRAN AND INTEGRATED INTO THE NGA PLATFORM

```
! ================================================ !
! Compute height of jet core for DMD postprocessing !
! ================================================ !
module dump_core_height
  use precision
  use partition
  use math
  use string
  use multiphase
  use multiphase_tools
  use dump_struct
  implicit none

  integer :: filenum,sc_iter,n_Rad,nDeg,init_dir,s_num,
     deg_spacing
  real(WP) :: xc_jet,yc_jet,zc_jet,Lx,Ly,Lz,s_grid
  real(WP), dimension(:,:), allocatable :: normal,
     jet_curve_loc,cHeight,cHeight_global,ct3,ct3_global
  real(WP), dimension(3) :: AV
  real(WP), dimension(:,:,:), allocatable :: xyz_loc,
     xyz_loc_global

end module

! ============== !
! Initialization !
! ============== !
subroutine dump_core_height_init
  use dump_core_height
  implicit none

  character(50) :: filename2,filename3
  integer :: i4
  real(WP), dimension(:), allocatable :: JCL_init_array
  real(WP), dimension(3) :: variables

  ! Read number of degrees
  call parser_read('Core␣height␣degrees',nDeg,360)
  !number of subcell iterations
  call parser_read('subcell␣iterations',sc_iter,2)
  ! find initial direction
  call parser_read('initial␣direction',init_dir,1)
  ! find x/y/zlength
```

```
!call parser_Read('Lx',Lx,5.0_WP)
Lx = x(imax) + (x(imin_+1) - x(imin_))
!call parser_Read('Ly',Ly,5.0_WP)
Ly = y(jmax) + (y(jmin_+1) - y(jmin_))
!call parser_Read('Lz',Lz,5.0_WP)
Lz = z(kmax) + (z(kmin_+1) - z(kmin_))
deg_spacing = 360/nDeg

! define number of steps to take in axial direction (
   axial jet only)
! define local number of steps for allocation of cHeight
   &xyz_loc (axial jet only)
! define s direction step size (axial jet only)
select case (init_dir)
  case (1)
    s_num = Nx*2*sc_iter
    s_grid = (Lx/(real(Nx,WP)*2))/(real(sc_iter,WP))
  case (2)
    s_num = Ny*2*sc_iter
    s_grid = (Ly/(real(Ny,WP)*2))/(real(sc_iter,WP))
  case (3)
    s_num = Nz*2*sc_iter
    s_grid = (Lz/(real(Nz,WP)*2))/(real(sc_iter,WP))
end select

n_Rad = s_num*nDeg

allocate(ct3(nDeg,s_num))
allocate(ct3_global(nDeg,s_num))
allocate(cHeight(nDeg,s_num))
allocate(cHeight_global(nDeg,s_num))
allocate(xyz_loc(nDeg,3,s_num))
allocate(xyz_loc_global(nDeg,3,s_num))
allocate(jet_curve_loc(s_num,3))
allocate(normal(s_num,3))
allocate(JCL_init_array(s_num))
do i4 = 1,s_num
  JCL_init_array(i4) = i4-1
end do

  variables = [init_dir,s_num,nDeg]

  normal(:,:) = 0.0_WP
```

```
      !define center of jet (try to avoid exactly 0.0_WP)
      xc_jet = 0.0_WP + ((x(imin+1) - x(imin))/(4.0_WP*
         real(sc_iter,WP)))!2.2e-017_WP
      yc_jet = 0.0_WP + ((y(jmin+1) - y(jmin))/(4.0_WP*
         real(sc_iter,WP)))!2.2e-017_WP
      zc_jet = 0.0_WP + ((z(kmin+1) - z(kmin))/(4.0_WP*
         real(sc_iter,WP)))!2.2e-017_WP
      !initialize jet center function
      jet_curve_loc(:,1) = xc_jet
      jet_curve_loc(:,2) = yc_jet
      jet_curve_loc(:,3) = zc_jet
      !calculate jet center function (axial jet only)
      !initial starting direction along with arbitrary
         vector to make sure AV!=normal at first (axial
         jet only)
      select case (init_dir)
        case (1)
          jet_curve_loc(:,1) = (JCL_init_array(:)+xc_jet)*
             s_grid
          normal(:,1) = 1.0_WP !normal(1,:) = [1.0_WP,0.0
             _WP,0.0_WP]
          AV = [0.0_WP,1.0_WP,0.0_WP]
        case (2)
          jet_curve_loc(:,2) = (JCL_init_array(:)+yc_jet)*
             s_grid
          normal(:,2) = 1.0_WP !normal(1,:) = [0.0_WP,1.0
             _WP,0.0_WP]
          AV = [1.0_WP,0.0_WP,0.0_WP]
        case (3)
          jet_curve_loc(:,3) = (JCL_init_array(:)+zc_jet)*
             s_grid
          normal(:,3) = 1.0_WP !normal(1,:) = [0.0_WP,0.0
             _WP,1.0_WP]
          AV = [1.0_WP,0.0_WP,0.0_WP]
      end select
!cleanup, boundaries (fortran vairiable epselon(0.0_WP))
    ??
    filename2='normal'
    filename3='jet_center_loc'

!write all of normal and JCL to file once !call
   binary_file_open/write?
if (irank.eq.1) then
```

```
      open (unit=22, file=filename2, status='replace',
         action = 'write', form = 'unformatted', access='
         stream')
      do i4 = 1, 3
            write(22) normal(:,i4)
      enddo
      close (unit=22)
      open (unit=33, file=filename3, status='replace',
         action = 'write', form = 'unformatted', access='
         stream')
        do i4 = 1, 3
            write(33) jet_curve_loc(:,i4)
        enddo
      close (unit=33)
      open (unit=44, file='variables', status='replace',
         action = 'write', form = 'unformatted', access='
         stream')
      write(44) variables
      close (unit=44)
  end if

  ! Initialize file counter
  filenum=0

end subroutine dump_core_height_init

! =============== !
! data extraction !
! =============== !
subroutine dump_core_height_data
  use dump_core_height
  implicit none
  integer :: n,n1,i,ip,ic,j,jp,jc,k,kp,kc,s,s_p,s_c,i2,i4,
     case_num,update
  integer :: id_largest
  real(WP) x_i,y_i,z_i,R_2,Xd,Yd,Zd,R_save,x_save,y_save,
     z_save,weight
  real(WP) :: vol_largest
  real(WP), dimension(2,2) :: R_plot !1st row is R values
     at each cell, 2nd is line weights
  real(WP), dimension(3) :: t1,t2
  character(50) :: filename
  real(WP), dimension(3,2) :: xyz_save
```

```
!add function found from fit of all data

!initialize cHeight, x/y/z locations
ct3(:,:) = 0.0_WP
cHeight(:,:) = 0.0_WP
xyz_loc(:,:,:) = 0.0_WP

! Get id number of largest liquid structure identified
   by dump_struct
vol_largest=0.0_WP
do n=1,n_meta_struct
  if (meta_structures_list(n)%vol.gt.vol_largest) then
    vol_largest=meta_structures_list(n)%vol
    id_largest=n
  end if
end do
id_largest=meta_structures_list(id_largest)%id

! Compute the height of core looping over function s and
    theta
sloop:  do i2 = 1,s_num
  !find center
  ic = find_i(jet_curve_loc(i2,1)) ! x dir cell number
     of current cell (starting according to center
     location)
  jc = find_j(jet_curve_loc(i2,2)) ! y dir cell number
     of current cell (starting according to center
     location)
  kc = find_k(jet_curve_loc(i2,3)) ! z dir cell number
     of currenct cell (starting according to center
     location)
  s_c =  find_s(ic,jc,kc,jet_curve_loc(i2,1),
     jet_curve_loc(i2,2),jet_curve_loc(i2,3)) !find
     starting subcell

  ! Check if this proc has this cell (axial jet only?)
  select case (init_dir)
    case(1)
      if (imin_.gt.ic .or. ic.gt.imax_) cycle
    case(2)
      if (jmin_.gt.jc .or. jc.gt.jmax_) cycle
    case(3)
```

```
    if (kmin_.gt.kc .or. kc.gt.kmax_) cycle
end select

! !set ray: line = t1*cos(theta) + t2*sin(theta) +
   starting_point(i,j,k) *ray = (Xd*R + Xo)i + (Yd*R +
    Yo)j + (Zd*R + Zo)k
! !                t1 = [normal]X[Arbitrary Vector]  *
   arbirtary vector=> anything cross with normal that
   is not the normal itself will be on desired plane
! !                t2 = [normal]X[t1]
! Xd = t1i*cos(theta) + t2i*sin(theta)
! Yd = t1j*cos(theta) + t2j*sin(theta)
! Zd = t1k*cos(theta) + t2k*sin(theta)

t1(1) = normal(i2,2)*AV(3) - normal(i2,3)*AV(2)
t1(2) = -(normal(i2,1)*AV(3) - normal(i2,3)*AV(1))
t1(3) = normal(i2,1)*AV(2) - normal(i2,2)*AV(1)

t2(1) = normal(i2,2)*t1(3) - normal(i2,3)*t1(2)
t2(2) = -(normal(i2,1)*t1(3) - normal(i2,1)*t1(2))
t2(3) = normal(i2,1)*t1(2) - normal(i2,2)*t1(1)

    n = 0
    do n1=deg_spacing,360,deg_spacing
    n = n + 1
      ! Start at jet center
      i = ic
      j = jc
      k = kc
      s = s_c

      !calculate Xd/Yd/Zd from degree, t1, and t2
      Xd = t1(1)*cos(real(n1,WP)*pi/180.0_WP) + t2(1)*
         sin(real(n1,WP)*pi/180.0_WP)
      Yd = t1(2)*cos(real(n1,WP)*pi/180.0_WP) + t2(2)*
         sin(real(n1,WP)*pi/180.0_WP)
      Zd = t1(3)*cos(real(n1,WP)*pi/180.0_WP) + t2(3)*
         sin(real(n1,WP)*pi/180.0_WP)

      !inititalize line_loop
      R_plot(:,:) = 0.0_WP
      R_plot(1,1) = 0.0_WP
      R_plot(2,1) = 1.0_WP
```

```
xyz_save(1,1) = jet_curve_loc(i2,1)
xyz_save(2,1) = jet_curve_loc(i2,2)
xyz_save(3,1) = jet_curve_loc(i2,3)
cHeight(n,i2) = 0.0_WP
ct3(n,i2) = 0.0_WP

line_loop: do while (i.le.imax .and. i.ge.imin .
   and. j.le.jmax .and. j.ge.jmin .and. k.le.
   kmax .and. k.ge.kmin)
  !check processesors again, update cell and
     cycle until a cell within proc is reached.
  if (imin_.gt.i .or. i.gt.imax_ .or. jmin_.gt.j
      .or. j.gt.jmax_ .or. kmin_.gt.k .or. k.gt.
     kmax_) then
    !save the cell info as previous cell info
    ip = i
    jp = j
    kp = k
    s_p = s
    call find_current_exit_cell_change(Lx,Ly,Lz,
       jet_curve_loc(i2,:),s,i,j,k,R_2,x_i,y_i,
       z_i,Xd,Yd,Zd,case_num)
    call case_num_cell_change(case_num,s,i,j,k)
    if (i.le.imax_ .and. i.ge.imin_ .and. j.le.
       jmax_ .and. j.ge.jmin_ .and. k.le.kmax_ .
       and. k.ge.kmin_) then
      R_plot(1,1) = R_2
      xyz_save(1,1) = x_i
      xyz_save(2,1) = y_i
      xyz_save(3,1) = z_i
      call line_checks(s_p,ip,jp,kp,i2,Xd,Yd,Zd,
         x_i,y_i,z_i,R_save,x_save,y_save,z_save
         ,weight,update)
      R_plot(2,1) = weight
      if (update.eq.1) then
        R_plot(1,1) = R_save
        xyz_save(1,1) = x_save
        xyz_save(2,1) = y_save
        xyz_save(3,1) = z_save
      end if
    end if
    cycle line_loop
  end if
```

```
!current cell info
call find_current_exit_cell_change(Lx,Ly,Lz,
   jet_curve_loc(i2,:),s,i,j,k,R_2,x_i,y_i,z_i
   ,Xd,Yd,Zd,case_num)
R_plot(1,2) = R_2
xyz_save(1,2) = x_i
xyz_save(2,2) = y_i
xyz_save(3,2) = z_i
!check for interfaces
call line_checks(s,i,j,k,i2,Xd,Yd,Zd,x_i,y_i,
   z_i,R_save,x_save,y_save,z_save,weight,
   update)
R_plot(2,2) = weight
if (update.eq.1) then
  R_plot(1,2) = R_save
  xyz_save(1,2) = x_save
  xyz_save(2,2) = y_save
  xyz_save(3,2) = z_save
end if

!find if you want to keep the R then add it
if (R_plot(2,2).eq.0.5_WP) then ! PLIC
   intersection
  cHeight(n,i2) = cHeight(n,i2) + R_plot(1,2)
  xyz_loc(n,1,i2) = xyz_loc(n,1,i2) + xyz_save
     (1,2)
  xyz_loc(n,2,i2) = xyz_loc(n,2,i2) + xyz_save
     (2,2)
  xyz_loc(n,3,i2) = xyz_loc(n,3,i2) + xyz_save
     (3,2)
  ct3(n,i2) = ct3(n,i2) + 1.0_WP
else if (abs(R_plot(2,1)-R_plot(2,2)).gt.0.51
   _WP) then ! Interface at Cell Boundary
  cHeight(n,i2) = cHeight(n,i2) + R_plot(1,1)
  xyz_loc(n,1,i2) = xyz_loc(n,1,i2) + xyz_save
     (1,1)
  xyz_loc(n,2,i2) = xyz_loc(n,2,i2) + xyz_save
     (2,1)
  xyz_loc(n,3,i2) = xyz_loc(n,3,i2) + xyz_save
     (3,1)
  ct3(n,i2) = ct3(n,i2) + 1.0_WP
end if
```

```
            R_plot(1,1) = R_plot(1,2)
            R_plot(2,1) = R_plot(2,2)
            xyz_save(1,1) = xyz_save(1,2)
            xyz_save(2,1) = xyz_save(2,2)
            xyz_save(3,1) = xyz_save(3,2)
            !update cell
            call case_num_cell_change(case_num,s,i,j,k)
          enddo line_loop
        end do
  end do sloop

!parallel_sum here: sum all cHeight's,xyz_loc's, and ct3's
   . Then average all at once.
call parallel_sum_real_2d(cHeight,cHeight_global)
call parallel_sum_real_3d(xyz_loc,xyz_loc_global)
call parallel_sum_real_2d(ct3,ct3_global)
cHeight_global(:,:) =  cHeight_global(:,:)/ct3_global(:,:)
do i4 = 1,3
  xyz_loc_global(:,i4,:) = xyz_loc_global(:,i4,:)/
     ct3_global(:,:)
end do

!write filenames according to filenum (normal and JCL in
   init?)
if (filenum.lt.10) then
  write(filename,'(A,i1)')'core_height_',filenum
elseif (filenum.gt.99) then
  write(filename,'(A,i3)')'core_height_',filenum
else
  write(filename,'(A,i2)')'core_height_',filenum
endif
filenum=filenum+1

if (irank.eq.1) then
! export cHeight to individual binary files at each time
   step
      open (unit=11, file=trim(filename), status='replace'
         , action = 'write', form = 'unformatted', access=
         'stream')
      !Write cHeight to file
      do i4 = 1, s_num
          write(11) cHeight_global(:,i4)
```

```
      enddo
      !Close file
      close (unit=11)
end if

contains

  function find_j(jet_curve_loc2) result(jc)
    implicit none
    real(WP), intent(in) :: jet_curve_loc2
    integer :: jc !intent(out)
    integer :: j

    do j=jmin,jmax
       if (y(j)>jet_curve_loc2) then
          jc=j-1
          EXIT
      end if
    end do
    return
  end function find_j

  function find_k(jet_curve_loc3) result(kc)
  implicit none
    real(WP), intent(in) :: jet_curve_loc3
    integer :: kc !intent(out)
    integer :: k

    do k=kmin,kmax
       if (z(k)>jet_curve_loc3) then
          kc=k-1
          EXIT
      end if
    end do
    return
  end function find_k

  function find_i(jet_curve_loc1) result(ic)
  implicit none
    real(WP), intent(in) :: jet_curve_loc1
    integer :: ic !intent(out)
    integer :: i
```

```
  do i=imin,imax
     if (x(i)>jet_curve_loc1) then
        ic=i-1
        EXIT
    end if
  end do
  return
end function find_i

function find_s(ic,jc,kc,jet_curve_loc1,jet_curve_loc2,
   jet_curve_loc3) result(s_c)
  use dump_core_height
    implicit none
    integer, intent(in) :: ic,jc,kc
    real(WP), intent(in) :: jet_curve_loc1,
       jet_curve_loc2,jet_curve_loc3
    integer :: s_c !intent(out)
    integer :: s

    do s = 1,8
      if (jet_curve_loc3.le.zs2(s,kc) .and.
         jet_curve_loc3.ge.zs1(s,kc) .and.
         jet_curve_loc2.le.ys2(s,jc) .and.&
         &jet_curve_loc2.ge.ys1(s,jc) .and.
            jet_curve_loc1.le.xs2(s,ic) .and.
            jet_curve_loc1.ge.xs1(s,ic)) then
          s_c = s
          EXIT
      end if
    end do

    return
end function find_s

subroutine find_current_exit_cell_change(Lx,Ly,Lz,
   jet_curve_loc,s,i,j,k,R_2,x_i,y_i,z_i,Xd,Yd,Zd,
   case_num)
  implicit none
  integer, intent(in) :: s,i,j,k
  real(WP), intent(in) :: Lx,Ly,Lz,Xd,Yd,Zd
  real(WP), dimension(3), intent(in) :: jet_curve_loc
  real(WP) :: R1,Vd,x_if,y_if,z_if
  real(WP), dimension(4) :: plane
```

```
integer :: i4
real(WP), intent(out) :: R_2,z_i,y_i,x_i
integer, intent(out) :: case_num

!ray-plane intersection
R_2 = Lx + Ly + Lz
do i4 = 1,3
  select case (i4) !set plane according to case
  case (1);
    if (Xd.gt.0) then
      plane = [1.0_WP,0.0_WP,0.0_WP,-xs2(s,i)] !second
          x wall
    else
      plane = [1.0_WP,0.0_WP,0.0_WP,-xs1(s,i)] !first
         x wall
    endif
  case (2);
    if (Yd.gt.0) then
      plane = [0.0_WP,1.0_WP,0.0_WP,-ys2(s,j)] !second
          y wall
    else
      plane = [0.0_WP,1.0_WP,0.0_WP,-ys1(s,j)] !first
         y wall
    end if
  case (3);
    if (Zd.gt.0) then
      plane = [0.0_WP,0.0_WP,1.0_WP,-zs2(s,k)] !second
          z wall
    else
      plane = [0.0_WP,0.0_WP,1.0_WP,-zs1(s,k)] !first
         z wall
    end if
  end select

  Vd = plane(1)*Xd + plane(2)*Yd  + plane(3)*Zd

  if (Vd.ne.0.0_WP) then !Vd checks if ray || plane
    R1 = -(plane(1)*jet_curve_loc(1) + plane(2)*
       jet_curve_loc(2) + plane(3)*jet_curve_loc(3) +
       plane(4))/Vd
    x_if = R1*Xd + jet_curve_loc(1)
    y_if = R1*Yd + jet_curve_loc(2)
    z_if = R1*Zd + jet_curve_loc(3)
```

```
        if (R1.ge.0.0_WP .and. R1.lt.R_2) then
          R_2 = R1
          x_i = x_if
          y_i = y_if
          z_i = z_if
          case_num = i4
          if (case_num.eq.1 .and. Xd.gt.0) then
            case_num = 2
          else if (case_num.eq.1 .and. Xd.lt.0) then
           case_num = 1
          else if (case_num.eq.2 .and. Yd.gt.0) then
            case_num = 4
          else if (case_num.eq.2 .and. Yd.lt.0) then
           case_num = 3
          else if (case_num.eq.3 .and. Zd.gt.0) then
            case_num = 6
          else if (case_num.eq.3 .and. Zd.lt.0) then
           case_num = 5
          end if
        endif
      endif
    end do

    return
  end subroutine find_current_exit_cell_change

  !update cell
  subroutine case_num_cell_change(case_num,s,i,j,k)
      use dump_core_height
      implicit none
      integer, intent(in) :: case_num
      integer, intent(inout) :: s,i,j,k
      i = i
      j = j
      k = k
      select case(case_num)
      case(1)
          i = i + sc_im(s)
          s = sc_sx(s)
      case(2)
          i = i + sc_ip(s)
          s = sc_sx(s)
      case(3)
```

```
            j = j + sc_jm(s)
            s = sc_sy(s)
        case(4)
            j = j + sc_jp(s)
            s = sc_sy(s)
        case(5)
            k = k + sc_km(s)
            s = sc_sz(s)
        case(6)
            k = k + sc_kp(s)
            s = sc_sz(s)
          end select

end subroutine case_num_cell_change

subroutine line_checks(s,i,j,k,i2,Xd,Yd,Zd,x_i,y_i,z_i,
   R_save,x_save,y_save,z_save,weight,update)
    use dump_core_height
    implicit none
    real(WP), intent(in) :: Xd,Yd,Zd,x_i,y_i,z_i
    integer, intent(in) :: s,i,j,k,i2
    real(WP) :: Vd,x_ip,y_ip,z_ip,R_1
    real(WP), intent(out) :: weight,R_save,x_save,y_save
       ,z_save
    integer, intent(out) :: update

    update = 0
    R_save = 0.0_WP
    x_save = 0.0_WP
    y_save = 0.0_WP
    z_save = 0.0_WP

    if (id(i,j,k).ne.id_largest) then !check correct
       structure
      weight = 0.0_WP
    else !find ray/plane intersection point and Radius
      if (VOF(s,i,j,k).lt.VOFhi) then !look for ray/
         plane intersection if cell contains volume
         fraction
        if (VOF(s,i,j,k).le.VOFlo) then !if this cell
           VOF = 0, just set line weight to 0
          weight = 0.0_WP
        else
```

```
      Vd = normx(s,i,j,k)*Xd + normy(s,i,j,k)*Yd +
         normz(s,i,j,k)*Zd
      if (Vd.ne.0.0_WP) then !check if ray || plane
        R_1 = -(normx(s,i,j,k)*jet_curve_loc(i2,1) +
            normy(s,i,j,k)*jet_curve_loc(i2,2) +
           normz(s,i,j,k)*jet_curve_loc(i2,3) - dist
           (s,i,j,k))/Vd
        x_ip = R_1*Xd + jet_curve_loc(i2,1)
        y_ip = R_1*Yd + jet_curve_loc(i2,2)
        z_ip = R_1*Zd + jet_curve_loc(i2,3)
      else ! if ray || plane, go straight to gas/
         liquid check
        x_ip = xs2(s,i) + 1.0_WP
        y_ip = ys2(s,j) + 1.0_WP
        z_ip = zs2(s,k) + 1.0_WP
      endif
      !set interesection and weight if intersection
         is within cell boundaries. Determine if in
         gas or liquid otherwise
      if (z_ip.le.zs2(s,k) .and. z_ip.ge.zs1(s,k) .
         and. y_ip.le.ys2(s,j) .and. y_ip.ge.ys1(s,j
         ) .and. x_ip.le.xs2(s,i) .and. x_ip.ge.xs1(
         s,i) .and. R_1.gt.0.0_WP) then
        !reset R value from cell intersection to
           plane intersection
        update = 1
        R_save = R_1
        x_save = x_ip
        y_save = y_ip
        z_save = z_ip
        weight = 0.5_WP
      elseif ((normx(s,i,j,k)*x_i+normy(s,i,j,k)*y_i
         +normz(s,i,j,k)*z_i-dist(s,i,j,k)).lt.0.0
         _WP) then
        weight = 1.0_WP
      elseif ((normx(s,i,j,k)*x_i+normy(s,i,j,k)*y_i
         +normz(s,i,j,k)*z_i-dist(s,i,j,k)).gt.0.0
         _WP) then
        weight = 0.0_WP
      endif
    endif
  else
    weight = 1.0_WP
```

```
        endif
      endif
  end subroutine line_checks

end subroutine dump_core_height_data
```

APPENDIX B

APPLICATION OF EXACT DMD TO EXTRACTED POSITIONAL DATA

CODE, WRITTEN IN MATLAB

```
clear;
clc;

%initial numbers
fileID = fopen('variables');
variables = fread(fileID,'double');%[1,5],'integer');
init_dir = variables(1);
s_num = variables(2);
deg = variables(3);

deg_spacing = 360/deg;

n = 37; %number of snapshots

%get normal
fileID = fopen('normal');
normal = fread(fileID,[s_num,3],'double');
%get JCL
fileID = fopen('jet_center_loc');
jet_curve_loc = fread(fileID,[s_num,3],'double');
%AV
switch init_dir
    case 1
        AV = [0,1,0];
    case 2
        AV = [1,0,0];
    case 3
        AV = [1,0,0];
end
%%
s_num = 338;%180 %truncate data here
%Data
%p = 0;
C = zeros(deg,s_num,n);
cHeight_o = zeros(deg,3,s_num,n);
t1 = zeros(3);
t2 = zeros(3);
for k = 1:n
    fileID = fopen(['core_height_',num2str(k-1)]);
    C_k = fread(fileID,[deg,s_num],'double');
    C(:,:,k) = C_k;

    figure(1)
```

```
clf(1)
xlabel('x')
ylabel('y')
zlabel('z')
title(['Original␣data␣at␣timestamp␣', num2str(k-1)],'
   fontsize',24)
set(gca,'Fontsize',24)
set(gcf,'units','normalized','outerposition',[0 0 1
   1])
%view([70 25])
view([355 30])
axis([0 .006 -.0015 .0015 -.0015 .0015])
for j = 1:s_num

    t1(1) = normal(j,2)*AV(3) - normal(j,3)*AV(2);
    t1(2) = -(normal(j,1)*AV(3) - normal(j,3)*AV(1));
    t1(3) = normal(j,1)*AV(2) - normal(j,2)*AV(1);

    t2(1) = normal(j,2)*t1(3) - normal(j,3)*t1(2);
    t2(2) = -(normal(j,1)*t1(3) - normal(j,1)*t1(2));
    t2(3) = normal(j,1)*t1(2) - normal(j,2)*t1(1);

    i1 = 0;
    for i = deg_spacing:deg_spacing:360
        i1 = i1 + 1;

        Xd = t1(1)*cosd(i) + t2(1)*sind(i);
        Yd = t1(2)*cosd(i) + t2(2)*sind(i);
        Zd = t1(3)*cosd(i) + t2(3)*sind(i);

        cHeight_o(i1,1,j,k) = C(i1,j,k)*Xd +
           jet_curve_loc(j,1);
        cHeight_o(i1,2,j,k) = C(i1,j,k)*Yd +
           jet_curve_loc(j,2);
        cHeight_o(i1,3,j,k) = C(i1,j,k)*Zd +
           jet_curve_loc(j,3);

    end
    hold on
    plot3(cHeight_o(:,1,j,k),cHeight_o(:,2,j,k),
       cHeight_o(:,3,j,k),'.','MarkerSize',.1,'color'
       ,[0 0 0])
    hold off
```

```
    end
%     if k == ... %set video snapshots here
%      p = p+1;
%      F(p) = getframe(gcf);
%     end
    pause(.01)
end

% video = VideoWriter('OG_data_2.avi','Uncompressed AVI');
% video.FrameRate = 1;
% open(video)
% writeVideo(video,F)
% close(video)
%%
%vectorize C
X = zeros(size(C,1)*size(C,2),n);
for i = 1:size(C,3)
    for j = 0:size(C,2)-1
        X((1+(j*size(C,1))):(size(C,1)+(j*size(C,1))),i) =
            C(:,j+1,i);
    end
end
X1 = X(:,1:end-1); %(1:t-1)
X2 = X(:,2:end); %(2:t)

%POD
[u,s,v] = svd(X1,'econ');
percentages = diag(s)/sum(diag(s));

figure(2)
clf(2)
plot(percentages,'o','LineWidth',8)
ylabel('percentage␣of␣"energy"')
xlabel('mode␣number')
title('Energy␣of␣Modes')
set(gca,'Fontsize',24)
set(gcf,'units','normalized','outerposition',[0 0 1 1])
drawnow

%find number of modes to use
percent_ok_with = 0;
mode_num = 0;
while percent_ok_with <= .8
```

```
    mode_num = mode_num + 1;
    percent_ok_with = percent_ok_with + percentages(
       mode_num);
end
mode_num = 36; %hard set to be able to observe all modes

%cumpute truncated A (koopman operator)
%A = u'*X2*v*(s^(-1));
A = u(:,1:mode_num)'*X2*v(:,1:mode_num)*(s(1:mode_num,1:
   mode_num)^(-1));

%compute eigenvectors W and eigenvalues lambda of A
[W,lambda] = eig(A);
%
% %compute phi (dynamic modes)
%phi = X2*v*(s^(-1))*W;
phi = X2*v(:,1:mode_num)*(s(1:mode_num,1:mode_num)^(-1))*W
   ;
% phi = u(:,1:mode_num)*W; %(standard DMD)

%compute b0 (vector of coefficients)
b0 = phi\X(:,1);

%de-vectorize phi into 3D matrix phi1 to plot later
phi1 = zeros(deg,s_num,mode_num);
for i = 1:mode_num
    for j = 0:s_num-1
        phi1(:,j+1,i) = phi((1+(j*deg)):(deg+(j*deg)),i)*
           b0(i);
        % *b0(i) results in low rank x0 as superposition
           is applied
    end
end

%compute temporal dynamics of dynamic modes (including
   future predictions)
%w = log(lambda)/dt; TD = exp(w*t);
time_dynamics = zeros(mode_num,n*2);
for i = 0:n*2-1
    time_dynamics(:,i+1) = (lambda^i)*b0;
end

%Superimpose for overall time dynamics (including future
```

```
   predictions)
time_dynamics1 = zeros(1,n*2);
for i = 1:mode_num
    time_dynamics1 = time_dynamics1 + time_dynamics(i,:);
end

%new x matrix (including future predictions)
Xp = zeros(size(C,1)*size(C,2),n*2);
for i = 0:n*2-1
    Xp(:,i+1) = phi*(lambda^(i))*b0;
end

%de-vectorize Xp into C1
C1 = zeros(deg,s_num,n*2);
for i = 1:n*2
    for j = 0:s_num-1
        C1(:,j+1,i) = Xp((1+(j*deg)):deg+(j*deg),i);
    end
end
%%
%C1 => cHeight_xyz and phi1 => phi1_xyz and plot
   cHeight_xyz
% p = 0;
cHeight_xyz = zeros(deg,3,s_num,n*2);
phi1_xyz = zeros(deg,3,s_num,mode_num);
for k = 1:n%*2

    figure(3)
    clf(3)
    xlabel('x')
    ylabel('y')
    zlabel('z')
    view([355 30])
    axis([0 .006 -.0015 .0015 -.0015 .0015])
    if k <= n
        title(['Low␣rank␣data␣at␣timestamp␣', num2str(k-1)
           ])
    else
        t = title(['*Predicted*␣Low␣rank␣data␣at␣timestamp
           ␣', num2str(k-1)]);
        t.Color = 'red';
    end
    set(gca,'Fontsize',24)
```

```
set(gcf,'units','normalized','outerposition',[0 0 1
   1])
for j = 1:s_num

    t1(1) = normal(j,2)*AV(3) - normal(j,3)*AV(2);
    t1(2) = -(normal(j,1)*AV(3) - normal(j,3)*AV(1));
    t1(3) = normal(j,1)*AV(2) - normal(j,2)*AV(1);

    t2(1) = normal(j,2)*t1(3) - normal(j,3)*t1(2);
    t2(2) = -(normal(j,1)*t1(3) - normal(j,1)*t1(2));
    t2(3) = normal(j,1)*t1(2) - normal(j,2)*t1(1);

    i1 = 0;
    for i = deg_spacing:deg_spacing:360
        i1 = i1 + 1;

        Xd = t1(1)*cosd(i) + t2(1)*sind(i);
        Yd = t1(2)*cosd(i) + t2(2)*sind(i);
        Zd = t1(3)*cosd(i) + t2(3)*sind(i);

        cHeight_xyz(i1,1,j,k) = C1(i1,j,k)*Xd +
           jet_curve_loc(j,1);
        cHeight_xyz(i1,2,j,k) = C1(i1,j,k)*Yd +
           jet_curve_loc(j,2);
        cHeight_xyz(i1,3,j,k) = C1(i1,j,k)*Zd +
           jet_curve_loc(j,3);

        if k<=mode_num
            phi1_xyz(i1,1,j,k) = phi1(i1,j,k)*Xd +
               jet_curve_loc(j,1);
            phi1_xyz(i1,2,j,k) = phi1(i1,j,k)*Yd +
               jet_curve_loc(j,2);
            phi1_xyz(i1,3,j,k) = phi1(i1,j,k)*Zd +
               jet_curve_loc(j,3);
        end

    end
    hold on
    plot3(real(cHeight_xyz(:,1,j,k)),real(cHeight_xyz
       (:,2,j,k)),real(cHeight_xyz(:,3,j,k)),'.','
       MarkerSize',.2,'color',[0 0 0])
    hold off
end
```

```
%      if k == ... %set video snapshots here
%       p = p+1;
%       F(p) = getframe(gcf);
%      end
    pause(.01)
end
% video = VideoWriter('low_rank_data_2.avi','Uncompressed
   AVI');
% video.FrameRate = 2;
% open(video)
% writeVideo(video,F)
% close(video)

%superimposed dynamic modes in cartesian coordinates
phi2 = zeros(deg,3,s_num);
phi2(:,1,:) = phi1_xyz(:,1,:,1);
for i = 1:mode_num
    phi2(:,2,:) = phi2(:,2,:) + phi1_xyz(:,2,:,i);
    phi2(:,3,:) = phi2(:,3,:) + phi1_xyz(:,3,:,i);
end
%%
% plot dynamic modes and time dynamics
k = 3;
for i = 1:mode_num
    k = k + 1;
    figure(k)
    clf(k)
    subplot(1,2,1)
    xlabel('x')
    ylabel('y')
    zlabel('z')
    title(['Dynamic␣mode␣number:␣', num2str(i)])
    view([355 30])
    %axis([0 .006 -.0015 .0015 -.0015 .0015])
    set(gca,'Fontsize',24)
    for j = 1:s_num
        hold on
        plot3(real(phi1_xyz(:,1,j,i)),real(phi1_xyz(:,2,j,
           i)),real(phi1_xyz(:,3,j,i)),'.','MarkerSize'
           ,.2,'color',[0 0 0])
        hold off
    end
    subplot(1,2,2)
```

```
    plot(1:n,real(time_dynamics(i,1:n)),'LineWidth',4)
    axis([0 n (min(real(time_dynamics(i,1:n)))-.05) (max(
       real(time_dynamics(i,1:n)))+.05)])
%     axis([0 size(time_dynamics,2) (min(real(
   time_dynamics(i,1:n)))-.05) (max(real(time_dynamics(i
   ,1:n)))+.05)])
    xlabel('timestamp')
    title('Time dynamics')
    set(gca,'Fontsize',24)
    set(gcf,'units','normalized','outerposition',[0 0 1
       1])
%     if size(time_dynamics,2) > n
%         hold on
%         plot(n:size(time_dynamics,2),real(time_dynamics(
   i,n:size(time_dynamics,2))),'-.')%,'LineWidth',4)
%         hold off
%         legend('existing temporal behavior','predicted
   temporal behavior')
%     end
    drawnow
    pause(.1)
end

figure(k+1)
clf(k+1)
subplot(1,2,1)
xlabel('x')
ylabel('y')
zlabel('z')
title('Superimposed dynamic mode (X0)')
view([355 30])
axis([0 .006 -.0015 .0015 -.0015 .0015])
set(gca,'Fontsize',24)
for j = 1:s_num
    hold on
    plot3(real(phi2(:,1,j)),real(phi2(:,2,j)),real(phi2
       (:,3,j)),'.','MarkerSize',.2,'color',[0 0 0])
    hold off
end

subplot(1,2,2)
plot(1:n,real(time_dynamics1(1:n)),'LineWidth',4)
xlabel('timestamp')
```

```
title('Superimposed time dynamics')
axis([0 n (min(real(time_dynamics1(1:n)))-.25) (max(real(
   time_dynamics1(1:n)))+.25)])
set(gca,'Fontsize',24)
set(gcf,'units','normalized','outerposition',[0 0 1 1])
%axis([0 size(time_dynamics,2) (min(real(time_dynamics1(1:
   n)))-.25) (max(real(time_dynamics1(1:n)))+.25)])
% if size(time_dynamics1,2) > n
%     hold on
%     plot(n:size(time_dynamics1,2),real(time_dynamics1(n:
   size(time_dynamics1,2))),'-.')%,'LineWidth',4)
%     hold off
%     legend('existing superimposed temporal behavior','
   predicted superimposed temporal behavior')
% end
%%
%plot DMD spectra
figure(k+2)
clf(k+2)
for i = 1:mode_num
    hold on
    plot(real(lambda(i,i)),imag(lambda(i,i)),'o','
       LineWidth',5);
    text(real(lambda(i,i))-.01,imag(lambda(i,i)+.08i),
       num2str(i))
    axis([-1.5 1.5 -1.5 1.5])
    xlabel('real')
    ylabel('imaginary')
    title('DMD spectrum (eigenvalues)')
    set(gca,'Fontsize',24)
    set(gcf,'units','normalized','outerposition',[0 0 1
       1])
    hold off
end
drawnow
%%
figure(k+3)
clf(k+3)
w = log(lambda);
for i = 1:mode_num
    hold on
    plot(real(w(i,i)),imag(w(i,i)),'o','LineWidth',5);
    axis([-.25 .03 -3.2 4])
```

```
     text(real(w(i,i))-.001,imag(w(i,i)+.22i),num2str(i))
    xlabel('real')
    ylabel('imaginary')
    title('DMD spectrum (eigenvalues logarithmically
       mapped)')
    set(gca,'Fontsize',24)
    set(gcf,'units','normalized','outerposition',[0 0 1
       1])
    hold off
end
drawnow
%%
figure(k+4)
clf(k+4)
for i = 1:mode_num
    hold on
    bar(imag(w(i,i)),norm(phi1(:,:,i)),.02)
    plot(imag(w(i,i)),norm(phi1(:,:,i)),'o','LineWidth',5,
       'Color',[0 0 0])
     text(imag(w(i,i)-.025i),norm(phi1(:,:,i))+.0015,
        num2str(i))
     axis([-3.2 3.2 0 .07])
    xlabel('frequency')
    ylabel('$$||\psi||$$','Interpreter','latex')
    title('DMD spectrum')
    set(gca,'Fontsize',24)
    set(gcf,'units','normalized','outerposition',[0 0 1
       1])
    hold off
end
drawnow
```

APPENDIX C

APPLICATION OF EXACT DMD TO A TEST CASE CODE WRITTEN IN MATLAB

```
clear;
clc;

x = linspace(0,10,100);
y = linspace(0,10,200);
t = 0;
t_end = 20;

%generate data changing in time
C = zeros(length(x),length(y),t_end);
for k = 1:t_end
    for i = 1:length(x)
        for j = 1:length(y)
            if y(j) <= 6 + .25*sin(.25*x(i)^2 + k) && y(j)
                >= 4 + .25*sin(.25*x(i)^2 + k)
                C(i,j,k) = 1 - .5*(y(j)-5)^2 + .2*sin(x(i)
                   ^2 + k);
            end
        end
    end
    figure(1)
    clf(1)
    surf(x,y,C(:,:,k)')
    colorbar
    xlabel('x');
    ylabel('y');
    zlabel('z');
    title(['Original data at timestep ', num2str(k)]);
    axis([0 10 0 10 -10 10])
    shading interp
    set(gca,'Fontsize',24)
    set(gcf,'units','normalized','outerposition',[0 0 1
       1])
%     F(k) = getframe(gcf);
    pause(.1)
end
% video = VideoWriter('OG_data.avi','Uncompressed AVI');
% video.FrameRate = 10;
% open(video)
% writeVideo(video,F)
% close(video)
%%
%vectorize C
```

```
X = zeros(size(C,1)*size(C,2),t_end);
for i = 1:size(C,3)
    for j = 0:size(C,2)-1
        X((1+(j*size(C,1))):(size(C,1)+(j*size(C,1))),i) =
            C(:,j+1,i);
    end
end

X1 = X(:,1:end-1); %(1:t-1)
X2 = X(:,2:end); %(2:t)

%POD
[u,s,v] = svd(X1);%,'econ');
percentages = diag(s)/sum(diag(s));
%%
figure(2)
clf(2)
plot(percentages,'*','LineWidth',2)
title('Energy␣of␣Modes')
ylabel('percentage␣of␣"energy"')
xlabel('mode␣number')
set(gca,'Fontsize',24)
set(gcf,'units','normalized','outerposition',[0 0 1 1])
drawnow
%%
%find number of modes to use
percent_ok_with = 0;
mode_num = 0;
while percent_ok_with <= .8
    mode_num = mode_num + 1;
    percent_ok_with = percent_ok_with + percentages(
       mode_num);
end

%cumpute truncated A (koopman operator)
%A = u(:,1:mode_num)'*X2*v(:,1:mode_num)*(s(1:mode_num,1:
   mode_num)^(-1));
A = u(:,1:mode_num)'*X2*v(:,1:mode_num)/s(1:mode_num,1:
   mode_num);

%compute eigenvectors W and eigenvalues lambda of A
[W,lambda] = eig(A);
```

```
%compute phi (dynamic modes)
%phi = X2*v(:,1:mode_num)*(s(1:mode_num,1:mode_num)^(-1))*
   W;
phi = X2*v(:,1:mode_num)/s(1:mode_num,1:mode_num)*W;
% phi = u(:,1:mode_num)*W; %(standard DMD)

%compute b0 (vector of coefficients)
b0 = phi\X(:,1);

%de-vectorize phi into 3D matrix phi1 to plot later
phi1 = zeros(length(x),length(y),mode_num);
for i = 1:mode_num
    for j = 0:length(y)-1
        phi1(:,j+1,i) = phi((1+(j*length(x))):(length(x)+(
           j*length(x))),i)*b0(i);
        % *b0(i) results in low rank x0 as superposition
           is applied
    end
end

%superimpose to get one total dynamic mode (this is just a
    low rank X(:,1))
phi2 = zeros(length(x),length(y));
for i = 1:mode_num
    phi2 = phi2 + phi1(:,:,i);
end

%compute temporal dynamics of dynamic modes (including
   future predictions)
%w = log(lambda)/dt; TD = exp(w*t);
time_dynamics = zeros(mode_num,t_end*2);
for i = 0:t_end*2-1
    time_dynamics(:,i+1) = (lambda^i)*b0;
end

%Superimpose for overal time dynamics (including future
   predictions)
time_dynamics1 = zeros(1,t_end*2);
for i = 1:mode_num
    time_dynamics1 = time_dynamics1 + time_dynamics(i,:);
end
% time_dynamics1 = time_dynamics1./mode_num;
```

```
%new x matrix (including future predictions)
Xp = zeros(size(C,1)*size(C,2),t_end*2);
for i = 0:t_end*2-1
    Xp(:,i+1) = phi*(lambda^(i))*b0;
end
%%
%de-vectorize Xp into C1 and plot
C1 = zeros(length(x),length(y),t_end*2);
for i = 1:size(C1,3)
    for j = 0:size(C1,2)-1
        C1(:,j+1,i) = Xp((1+(j*size(C1,1))):(size(C1,1)+(j
           *size(C1,1))),i);
    end
    figure(3)
    clf(3)
    surf(x,y,(real(C1(:,:,i)))')
    colorbar
    caxis([0, 1.2]);
    xlabel('x')
    ylabel('y')
    zlabel('z')
    if i <= t_end
        title(['Low␣rank␣data␣at␣timestep␣', num2str(i)])
    else
        t = title(['*Predicted*␣Low␣rank␣data␣at␣timestep␣
           ', num2str(i)]);
        t.Color = 'red';
    end
    axis([0 10 0 10 -10 10])
    shading interp
    set(gca,'Fontsize',24)
    set(gcf,'units','normalized','outerposition',[0 0 1
       1])
%     F(i) = getframe(gcf);
    pause(.1)
end
% video = VideoWriter('low_rank_data.avi','Uncompressed
   AVI');
% video.FrameRate = 10;
% open(video)
% writeVideo(video,F)
% close(video)
%%
```

```
% plot dynamic modes and time dynamics
k = 3;
for i = 1:mode_num
    k = k + 1;
    figure(k)
    clf(k)
    subplot(1,2,1)
    surf(x,y,real(phi1(:,:,i))')
    colorbar
    xlabel('x')
    ylabel('y')
    title(['Dynamic mode number: ', num2str(i)])
    axis([0 10 0 10])% -10 10])
    shading interp
    set(gca,'Fontsize',24)
    subplot(1,2,2)
    plot(1:t_end,real(time_dynamics(i,1:t_end)),'LineWidth
       ',3)
    %axis([0 size(time_dynamics,2) (min(real(time_dynamics
       (i,1:t_end)))-5) (max(real(time_dynamics(i,1:t_end)
       ))+5)])
    axis([0 size(time_dynamics,2)/2 (min(real(
       time_dynamics(i,1:t_end)))-5) (max(real(
       time_dynamics(i,1:t_end)))+5)])
    xlabel('timestep')
    title('Time dynamics')
    set(gca,'Fontsize',24)
    set(gcf,'units','normalized','outerposition',[0 0 1
       1])
%     if size(time_dynamics,2) > t_end
%     hold on
%     plot(t_end:size(time_dynamics,2),real(time_dynamics(
   i,t_end:size(time_dynamics,2))),'-.','LineWidth',4)
%     hold off
%     legend('existing temporal behavior','predicted
   temporal behavior')
%     end
    pause(.1)
end

figure(k+1)
clf(k+1)
subplot(1,2,1)
```

```
surf(x,y,real(phi2)')
colorbar
xlabel('x')
ylabel('y')
title('Superimposed dynamic mode (X0)')
axis([0 10 0 10])% -10 10])
shading interp
set(gca,'Fontsize',24)
subplot(1,2,2)
plot(1:t_end,real(time_dynamics1(1:t_end)),'LineWidth',3)
xlabel('timestep')
title('Superimposed time dynamics')
set(gca,'Fontsize',24)
set(gcf,'units','normalized','outerposition',[0 0 1 1])
% if size(time_dynamics1,2) > t_end
% hold on
% plot(t_end:size(time_dynamics1,2),real(time_dynamics1(
   t_end:size(time_dynamics1,2))),'-.','LineWidth',4)
% hold off
% legend('existing superimposed temporal behavior','
   predicted superimposed temporal behavior')
% end

%%
figure(k+2)
clf(k+2)
for i = 1:mode_num
    hold on
    plot(real(lambda(i,i)),imag(lambda(i,i)),'*','
       LineWidth',2);
    axis([-1 1.3 -1 1.3])
    text(real(lambda(i,i))-.08,imag(lambda(i,i)+.1i),['
       mode ', num2str(i)],'Fontsize',16)
    xlabel('real')
    ylabel('imaginary')
    title('DMD spectrum (eigenvalues)')
    set(gca,'Fontsize',24)
    set(gcf,'units','normalized','outerposition',[0 0 1
       1])
    hold off
end
%%
figure(k+3)
```

```
clf(k+3)
w = log(lambda);
for i = 1:mode_num
    hold on
    plot(real(w(i,i)),imag(w(i,i)),'*','LineWidth',2);
    axis([-.05 .05 -3 3])
    text(real(w(i,i))+.005,imag(w(i,i)),['mode ', num2str(
       i)],'Fontsize',16)
    xlabel('real')
    ylabel('imaginary')
    title('DMD spectrum (eigenvalues logarithmically
       mapped)')
    set(gca,'Fontsize',24)
    set(gcf,'units','normalized','outerposition',[0 0 1
       1])
    hold off
end
%%
figure(k+4)
clf(k+4)
for i = 1:mode_num
    hold on
    bar(imag(w(i,i)),norm(phi1(:,:,i)),.02)
    plot(imag(w(i,i)),norm(phi1(:,:,i)),'*','LineWidth',2,
       'Color',[0 0 0])
    text(imag(w(i,i))-.2,norm(phi1(:,:,i))+2.5,['mode ',
       num2str(i)],'Fontsize',16)
    xlabel('frequency')
    ylabel('$$||\psi||$$','Interpreter','latex')
    title('DMD spectrum')
    set(gca,'Fontsize',24)
    set(gcf,'units','normalized','outerposition',[0 0 1
       1])
    hold off
end
```