Institute of Parallel and Distributed Systems

Department of Simulation of Large Systems

University of Stuttgart
Universitätsstraße 38
D-70569 Stuttgart

Bachelor-Thesis

Dynamic Mode Decomposition for the Monodomain Equation in Neuromuscular System

Moritz Widmayer

Course of Study:	Informatik
Examiner:	Prof. Dr. rer. nat. habil. Miriam Mehl
Supervisor:	Dr.-Ing. Nehzat Emamy

Commenced:	October 15, 2018
Completed:	April 15, 2019

Acknowledgement

I'd like to thank Prof. Dr. Miriam Mehl for giving me the chance to carry out and create my bachelor's thesis at her department of the IPVS. Also, I'd like to thank my supervisor Dr. Nehzat Emamy for the time and effort she put into supporting me throughout the entire procedure of the thesis. I want to thank my family and friends for pushing me to reach further, but also comforting me in bad times. Lastly, I wish to thank Tabea Schoch for giving me more motivation than I ever had and a reason to progress.

Abstract

In this thesis, the dynamic mode decomposition (DMD) was implemented and applied to the resulting snapshots matrix of the monodomain equation in the neuromuscular system. The Hodgkin-Huxley model was used to obtain the snapshots v_l and the DMD was implemented in C++ into the DiHu framework.

The goal of this work was to reduce the number of dimensions of the initial snapshot data. For this, DMD calculates the Koopman operator R that should lead to the next snapshots matrix,

$$V_2^k \approx RV_1^{k-1}, \quad \text{where} \quad V_1^{k-1} = \begin{bmatrix} v_1 & \cdots & v_{k-1} \end{bmatrix} \quad \text{and} \quad V_2^k = \begin{bmatrix} v_2 & \cdots & v_k \end{bmatrix}$$

From the eigenvalues of R, the growth rates and frequencies can be obtained, while the DMD modes can be found in the eigenvectors.

For the used method of DMD, singular value decomposition (SVD) was required. This was implemented with the subroutines by LAPACK. All other matrix operations were also implemented with subroutines by either LAPACK or BLAS.

The results are promising as long as the snapshots contain higher numbers of parameters and when there are more snapshots to apply DMD on. When there aren't very many parameters and snapshots, the error can rise too high to be useful data. We also found that the variables ϵ_1 and ϵ_0, which play a significant role for the dimension reduction, have to be chosen carefully.

Contents

1 Introduction

The connection between the central nervous system, which is located in the brain [1], and the peripheral nervous system, which is composed of all other muscle related cells, is called neuromuscular system [2]. Muscle cells are similar to neurons, as they also propagate electrical action potentials on their cell membranes. They're divided into skeletal, cardiac and smooth muscle, where the former is the considered type for this thesis. It consists of all muscle fiber, that bend and extend bones at joints. [3]

Since the structure of muscle is very complex, modelling the system, by describing it as a composition of electrical elements, is raising a difficult problem. Here, the two dominant models for describing the propagation of electrical action potentials can be used: The bidomain model, which considers the muscle tissue as two continuous domains [4], and the monodomain model, which acts as a less complex variant of the bidomain model and only uses one continuous domain [5].

The Hodgkin-Huxley model, described by Hodgkin and Huxley in 1952, represents the elements of a neuron as a composition of electrical parts. With a total of only five parameters, its degree of freedom is comparatively small. [6]

Another model, developed by Shorten et al. in 2007, describes fatigue in skeletal muscles, while distinguishing between fast and slow twitch muscle fibre types, since fatigue occurs differently. Its number of parameters amounts to 44, so that there is a much greater degree of freedom compared to the Hodgkin-Huxley model. [7]

Now, it may be desirable to apply model order reduction to these models, in order to be able to simulate parts of the neuromuscular system efficiently. For this objective dynamic dode decomposition (DMD) was implemented into the digital human (DiHu) framework.

The DiHu project aims to develop realistic models of the human neuromuscular system. For this, high-performance computing is being used to upgrade existing models in order to realistically simulate skeletal muscles. [8]

2 Theory

2.1 Dynamic mode decomposition

The general idea of dynamic mode decomposition (DMD) is to reduce the dimensions of spatial-temporal data by finding coherent structures [9]. For this purpose, each snapshot v_l at time t_l will be decomposed into its amplitudes α, DMD modes u, frequencies ω and growth rates δ with i denoting the imaginary unit [10].

$$v_l = \sum_{m=1}^{k} \alpha_m u_m e^{(\delta_m + i\omega_m)(l-1)\Delta t} \tag{2.1}$$

First each snapshot of data v_l gets stored column-wise in a snapshots matrix V.

$$V = \begin{bmatrix} v_1 & \cdots & v_k \end{bmatrix} \tag{2.2}$$

For the first step, the snapshots matrix V is needed once without its last snapshot v_k and once without its first snapshot v_1, which are denoted by V_1^{k-1} and V_2^k, respectively.

$$V_1^{k-1} = \begin{bmatrix} v_1 & \cdots & v_{k-1} \end{bmatrix} \qquad V_2^k = \begin{bmatrix} v_2 & \cdots & v_k \end{bmatrix} \tag{2.3}$$

The Koopman operator R is considered such that when R is multiplied from the left onto V_1^{k-1} it results in V_2^k. This means that R advances the snapshots matrix V_1^{k-1} by one time step.

$$V_2^k \approx R V_1^{k-1} \tag{2.4}$$

Since there is no solution of the system when $k-1$ is greater than the number of rows, an approximation might only be reached. On the other hand when $k-1$ is less than the number of rows, there can exist multiple solutions for R.

Now eigendecomposition is applied to R yielding the eigenvalues λ in the diagonal of Λ and eigenvectors q in the columns of Q,

$$RQ = Q\Lambda. \tag{2.5}$$

As the final step of the DMD method, the oscillations and growth rates will be found from the eigenvalues while the DMD modes and amplitudes will be computed from the eigenvectors.

2.2 Eigendecomposition

Evidently, computing the eigenvalues and eigenvectors is very significant for the DMD. The eigendecomposition is used to factorize a given matrix A into its eigenvalues λ and eigenvectors q, which are stored in the diagonal of Λ and column-wise in Q, respectively [11],

$$\begin{aligned} AQ &= Q\Lambda \\ \Leftrightarrow \quad A &= Q\Lambda Q^{-1}. \end{aligned} \tag{2.6}$$

2.3 Singular value decomposition

One very important tool used in the DMD approach is the singular value decomposition (SVD). Its objective is to decompose a given rectangular matrix $V_{j\times k}$ into its left-singular vectors $U_{j\times j}$, singular values $\Sigma_{n\times n}$ and right-singular vectors $\left(T^{-1}\right)_{j\times k}$, where $n = \min(j,k)$.

$$V \overset{\text{SVD}}{=} U\Sigma T^{-1}$$

U and T^{-1} are unitary matrices. Therefore to compute their inverse they merely have to be conjugate transposed, denoted by *,

$$U^{-1} = U^* \qquad\qquad \left(T^{-1}\right)^* = (T^*)^* = T. \tag{2.7}$$

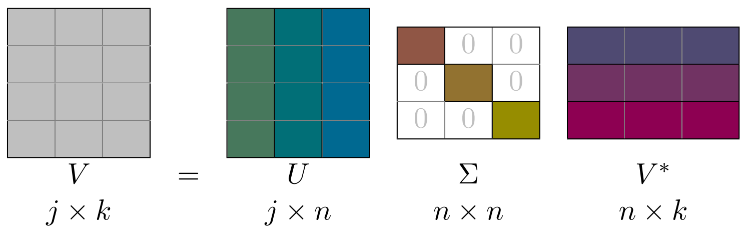
Further, Σ is a diagonal matrix. So the inverse is easily calculated by taking the reciprocal of each singular value.

$$\Sigma^{-1} = \begin{bmatrix} \sigma_1 & 0 & \cdots & \cdots & 0 \\ 0 & \sigma_2 & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & \ddots & \sigma_{n-1} & 0 \\ 0 & \cdots & \cdots & 0 & \sigma_n \end{bmatrix}^{-1} = \begin{bmatrix} \frac{1}{\sigma_1} & 0 & \cdots & \cdots & 0 \\ 0 & \frac{1}{\sigma_2} & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & \ddots & \frac{1}{\sigma_{n-1}} & 0 \\ 0 & \cdots & \cdots & 0 & \frac{1}{\sigma_n} \end{bmatrix} \tag{2.8}$$

SVD will be used for a first reduction step in the applied DMD method as well as to compute the pseudoinverses, which will be required to invert non-square matrices in multiple steps. The calculation of the pseudoinverse V^+ of V works by inverting every matrix that results from SVD and reversing the order of the multiplication [12],

$$V^+ = V\Sigma^{-1}U^*. \tag{2.9}$$

The truncated SVD, which only calculates the first n column vectors of U, diagonal entries of Σ and row vectors of V^*, will be used.



(2.10)

3 Method

Once the shapshots matrix $V_{j\times k}$ is obtained, the DMD method can be applied. The algorithm described in [10] is used.

3.1 Dynamic mode decomposition

The first step of the applied DMD method in [10] aims to reduce the snapshots matrix V by means of the spatial complexity k_{spat} to a reduced snapshots matrix $\hat{T}$.

3.1.1 Step 1: Reduction of the snapshots matrix

The idea of this reduction step is to find the dominant structures via SVD and continue with only the most dominant ones. Therefore SVD will be initially performed on V, which yields the left-singular vectors $U_{j\times n}$, the singular values $\Sigma_{n\times n}$ and the right-singular vectors $(T^{-1})_{n\times k}$, where $n = \min(j,k)$.

$$V \overset{\text{SVD}}{=} U\Sigma T^{-1} \tag{3.1}$$

Now the spatial complexity k_{spat} will be computed which will determine the number of dimensions that will be used in the following steps.

$$\begin{aligned} k_{\text{spat}} &= n - \max\left\{ m \,\middle|\, \frac{\|[\sigma_1 \cdots \sigma_{n-m}]^{\mathsf{T}}\|_2}{\|[\sigma_1 \cdots \sigma_n]^{\mathsf{T}}\|_2} > \epsilon_1 \right\} \\ &= n - \max\left\{ m \,\middle|\, \frac{\sqrt{\sigma_1^2 + \cdots + \sigma_{n-m}^2}}{\sqrt{\sigma_1^2 + \cdots + \sigma_n^2}} > \epsilon_1 \right\} \end{aligned} \tag{3.2}$$

This simply evaluates the number of singular values that can be omitted while the length of their vector is still at least ϵ_1 of the length of the vector of all singular values. Consequently only the first k_{spat} columns of U, diagonal entries of Σ and rows of T^{-1} will continued to be used whereby the reduced snapshots matrix $\hat{T}_{k_{\text{spat}}\times k}$ can now be created.

$$\hat{T} = \Sigma T^{*} \tag{3.3}$$

3.1.2 Step 2: Modified Koopman operator

Now that the least significant spatial dimensions have been omitted, the modified Koopman operator $\hat{R}_{k_{\text{spat}} \times k_{\text{spat}}}$ can be computed. For this the reduced snapshots matrix $\hat{T}$ will be used twice, once without the last reduced snapshot $\hat{t}_k$ and once without the first reduced snapshot $\hat{t}_1$, i.e. $\hat{T}_1^{k-1}$ and $\hat{T}_2^k$, respectively. This leads to the following system of linear equations that has to be solved.

$$\begin{aligned} \hat{R}\hat{T}_1^{k-1} &= \hat{T}_2^k \\ \Leftrightarrow \quad \hat{R} &= \hat{T}_2^k \left(\hat{T}_1^{k-1}\right)^{-1} \end{aligned} \tag{3.4}$$

Since $\hat{T}_1^{k-1}$ isn't necessarily square the pseudoinverse has to be used. To compute this, SVD is applied to $\hat{T}_1^{k-1}$ which can then be used to solve for the Koopman operator $\hat{R}$.

$$\begin{aligned} \hat{T}_1^{k-1} &\overset{\text{SVD}}{=} \hat{U}_1\hat{\Sigma}\hat{U}_2^* \\ \Leftrightarrow \quad \hat{T}_2^k\left(\hat{U}_1\hat{\Sigma}\hat{U}_2^*\right)^+ &= \hat{R} \\ \Leftrightarrow \quad \hat{T}_2^k\hat{U}_2\hat{\Sigma}^{-1}\hat{U}_1^* &= \hat{R} \end{aligned} \tag{3.5}$$

3.1.3 Step 3: Eigendecomposition of the modified Koopman operator

To produce the frequencies, growth rates, DMD modes and amplitudes the eigenvalues and eigenvectors of the modified Koopman operator $\hat{R}$ are required. Thereby eigendecomposition has to be applied and yields the eigenvalues λ_m in the diagonal of $\Lambda_{k_{\text{spat}} \times k_{\text{spat}}}$ and eigenvectors q_m as column vectors in $Q_{k_{\text{spat}} \times k_{\text{spat}}}$.

$$\hat{R} = Q\Lambda Q^{-1} \tag{3.6}$$

DMD modes of the reduced snapshots

Since this is the eigendecomposition of the modified Koopman operator, the eigenvectors q_m are equal to the DMD modes of the reduced snapshots matrix $\hat{T}$, denoted as

$$\hat{u}_m = q_m. \tag{3.7}$$

Growth rates and frequencies

The eigenvalues now contain the frequencies ω_m and the growth rates δ_m. Since they will be raised later to be in the exponent with base e, simply the natural logarithm has to be calculated of each eigenvalue λ_m.

$$\begin{aligned} \delta_m + i\omega_m &= \tfrac{\ln \lambda_m}{\Delta t} \\ \Rightarrow \begin{cases} \delta_m & = \operatorname{Re}\left(\tfrac{\ln \lambda_m}{\Delta t}\right) \\ \quad\omega_m & = \operatorname{Im}\left(\tfrac{\ln \lambda_m}{\Delta t}\right) \end{cases} \end{aligned} \tag{3.8}$$

Amplitudes of the reduced snapshots

Now the amplitudes of the reduced snapshots matrix $\hat{T}$, denoted as a, can be computed by first rewriting the original sum formula as follows.

$$\hat{t}_j = \sum_{m=1}^{k} a_m q_m \lambda_m^{j-1} \tag{3.9}$$

To continue the calculations the matrix form of the previous equation will be used where the reduced snapshots $\hat{t}_j$ will be stacked on top of another in b_K and the amplitudes a_m will be contained in a_K, where $K = k_{\text{spat}} \cdot k$.

$$b = \begin{bmatrix} \hat{t}_1 \\ \hat{t}_2 \\ \vdots \\ \hat{t}_{k-1} \\ \hat{t}_k \end{bmatrix} \qquad \text{and} \qquad a = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_{k_{\text{spat}}-1} \\ a_{k_{\text{spat}}} \end{bmatrix} \qquad \Rightarrow \qquad b = aQ\Lambda^{j-1} \tag{3.10}$$

Furthermore $Q\Lambda^{j-1}$ will be put together in $M_{K \times k_{\text{spat}}}$.

$$M = \begin{bmatrix} Q\Lambda^0 \\ Q\Lambda^1 \\ \vdots \\ Q\Lambda^{j-2} \\ Q\Lambda^{j-1} \end{bmatrix} \Rightarrow Ma = \hat{t}_j \tag{3.11}$$

In order to solve this system of linear equations, SVD will be used once again to compute the pseudoinverse of M.

$$\begin{aligned} M^{+} &\overset{\text{SVD}}{=} (U_r \Sigma_r V_r^*)^{+} \\ &= V_r \Sigma_r^{-1} U_r^* \\ \Rightarrow \quad a &= V_r \Sigma_r^{-1} U_r^* \hat{t}_j \end{aligned} \tag{3.12}$$

3.1.4 Step 4: DMD modes and amplitudes of the snapshots matrix

DMD modes

Now the amplitudes a will be used to scale the eigenvectors in Q to the appropriate length so that they will yield the DMD modes $u_{k_{\text{spat}} \times k}$.

$$u_m = a_m q_m \tag{3.13}$$

Amplitudes

Finally to compute the amplitudes $\alpha_{k_{\text{spat}}}$ the left-singular vectors U, the DMD modes u from the previous step and the number of rows j of the initial snapshots Matrix V will be used as follows.

$$\alpha_m = \frac{\|U u_m\|_2}{\sqrt{j}} \tag{3.14}$$

3.1.5 Step 5: Spectral complexity

Now, to further reduce the number of DMD modes and their belonging growth rates, frequencies and amplitudes, the spectral complexity will be calculated which is dependant on ϵ_0 that can be set manually. For this calculation, the DMD modes have to be ordered by the values of their amplitudes in decreasing order, i.e. $\alpha_l > \alpha_{l+1}$. Next, simply the greatest value for m will be chosen as spectral complexity k_{spec} so that the ratio between α_m and α_1 is greater than the selected ϵ_0.

$$k_{\text{spec}} = \max \left\{ m \,\middle|\, \frac{\alpha_m}{\alpha_1} > \epsilon_0 \right\} \tag{3.15}$$

For the reduction, only the k_{spec} greatest amplitudes and their corresponding DMD modes, growth rates and frequencies will be continued to be used while the remaining ones will be omitted.

3.1.6 Step 6: Reconstruction and error computation

To produce the reconstructed snapshots matrix V_{reconst}, the reconstructed reduced snapshots matrix $\hat{T}_{\text{reconst}}$ has to be obtained first.

Reconstructed reduced snapshots matrix

For this the formula will be used to compute the reduced snapshots $\hat{t}_{\text{reconst}}$ so that the matrix will be assembled column-wise.

$$\left(\hat{t}_{\text{reconst}}\right)_m = ue^{(\delta+i\omega)(t_m-t_0)} \tag{3.16}$$

Reconstructed snapshots matrix

Finally the reconstructed snapshots matrix V_{reconst} will be computed by reversing the very first step of the DMD (3.1), where SVD was applied to decompose the snapshots matrix V into its left-singular vectors U and the reduced snapshots matrix $\hat{T}$, composed of the singular values Σ and right-singular vectors T.

$$V_{\text{reconst}} = U\hat{T}_{\text{reconst}} \tag{3.17}$$

3.2 Implementation

The implementation is a part of the opendihu [13] C++ code. For the operations involving matrices and vectors LAPACK (Linear Algebra Package) and BLAS (Basic Linear Algebra Subroutines) were used. It's important to note that LAPACK works with matrices as single dimensional arrays, `double[]`, in case of real entries and `double _Complex[]` for complex ones.

3.2.1 Singular value decomposition

LAPACK subroutines `dgesvd()` and `zgesvd()`

The subroutines `dgesvd()` for real and `zgesvd()` for complex double precision matrices by LAPACK were used for the implementation of SVD. The very first parameter `matrix_layout` takes the matrix order which is `LAPACK_COL_MAJOR` in this case. By setting both the `jobu` and `jobvt` parameters to `'s'`, the truncated SVD is used. The two next parameters `m` and `n` take the number of rows and columns of the input matrix `a`, which is the next parameter. `lda` is the leading dimension of `a`, for this implementation this equals `m`. `s` and `u` are the matrices for the singular values and the left-singular vectors, respectively. Since LAPACK only returns the singular values in `s`, it only has a dimension of `(min(m,n))`, while `u` is `(m,min(m,n))`

```
int LAPACKE_dgesvd( int matrix_layout, char jobu, char
    jobvt, int m, int n, double a[], int lda, double s[],
    double u[], int ldu, double vt[], int ldvt, double
    superb[] )

int LAPACKE_zgesvd( int matrix_layout, char jobu, char
    jobvt,int m, int n, double _Complex a[], int lda, double
    _Complex s[], double _Complex u[], int ldu, double
    _Complex vt[], int ldvt, double superb[] )
```

because of the `'s'` in the `jobu` parameter and therefore `ldu` is equal to `m`. Lastly `vt` will contain the right-singular vectors stored row-wise and has `(min(m,n),n)` dimensions from which follows that `ldvt` has to be equal to `(min(m,n))`. [14] [15]

Function `getSVD()` in `svd_utility` of opendihu

```
void getSVD( double input[], int rows, int cols, double
    leftSingVec[], double sigma[], double rightSingVecT[] )
```

Our final implementation of the SVD, which wraps the LAPACK SVD, now only takes six parameters: the `input`, dimensions `rows` and `cols`, and resulting matrices of the SVD `leftSingVec`, `sigma` and `rightSingVecT`. After calling the subroutine of LAPACK, `sigma` still has to be filled with the singular values on its diagonal which is performed by two nested `for` loops. [16]

3.2.2 Eigendecomposition

LAPACK subroutine `dgeev()`

```
int LAPACKE_dgeev( int matrix_layout, char jobvl, char
    jobvr, int n, double a[], int lda, double wr[], double
    wi[], double vl[], int ldvl, double vr[], int ldvr )
```

For the eigendecomposition, our third step of the DMD method, only the subroutine `dgeev()`, which takes real double precision arrays has to used, since there are no complex matrices up to this step. First the parameter `matrix_layout` has to be set to `LAPACK_COL_MAJOR`. The two next parameters `jobvl` and `jobvr` are to determine whether the left and/or right eigenvectors are to be computed, which, since only the right eigenvectors are required, have to be set to `'N'` and `'V'` respectively. The following parameter `n` stands for the order of the array `a` which is the next parameter and the input matrix that is to be eigendecomposed. The leading dimension of `a`, `lda`, will simply be the same as its order `n`. `wr` and `wi` are the eigenvalues as `double[]` split into their respective real and imaginary parts. The next two parameters `vl` and `ldvl` can be set to anything valid because they won't be referenced since the left eigenvectors are not required. Lastly `vr` will contain the right eigenvectors as `double[]` with the leading dimension `ldvr` which, again, will be equal to `n`, the order of the initial matrix.

The eigenvalues of a real matrix will be either real or complex conjugate pairs. Further, for each real eigenvalue there will be a real eigenvector and for each complex conjugate pair of eigenvalues there will be a complex conjugate pair of eigenvectors. This property is used by LAPACK. The subroutine computes for each eigenvalue its real and imaginary part, hence the two `double[]` `wr` and `wi` are required. But for each complex conjugate pair of eigenvalues it will only calculate the eigenvector of the first eigenvalue. This is so that `vl` only requires the dimensions `(order,order)` by storing the real parts of the eigenvector in the column of the first eigenvalue and the imaginary parts in the following column of the second eigenvalue of the complex conjugate pair. [17]

Function `getEigen()` in `dmd_utility` of opendihu

```
void getEigen( double input[], int order, double _Complex
   eigenvalues[], double _Complex eigenvectors[] )
```

This implementation takes the real matrix as `input` and its `order` and stores its `eigenvalues` and `eigenvectors` as `double _Complex[]`. To achieve this, the eigenvalues and eigenvectors have to be correctly extracted after the call of `dgeev()`. [16]

3.2.3 Matrix multiplication

BLAS subroutine dgemm()

```
void cblas_dgemm( enum CBLAS_ORDER order, enum
   CBLAS_TRANSPOSE transA, enum CBLAS_TRANSPOSE transB, int
   m, int n, int k, double alpha, double a[], int lda,
   double b[], int ldb, double beta, double c[], int ldc )
```

For matrix multiplication of matrices with only real entries the subroutine dgemm() by BLAS is used. The first three parameters order, transA and transB are data types specifically used by BLAS for setting the order of the used matrices, as well as for specifying transpose options. For the implementation CblasColMajor and CblasNoTrans have to be set here, respectively. The integers m, n, and k are the dimensions of the multiplied matrices, where m is the number rows of a, n is the number of columns of b, and k is the number of columns of a and therefore also the number of rows of b. alpha is a scalar for the multiplication, so setting it to 1 renders it inactive. The array a is the left matrix of the multiplication with lda leading dimensions, meaning it has to be set equal to m. Correspondingly b is the right matrix and has ldb leading dimensions, hence it's k. beta is a scalar for an additional constant, which is not intended and set to 0. The result of the matrix multiplication will be stored in c, which has leading dimensions ldc, which again should be equal to m. [18]

Here it should be added that an analogous subroutine zgemm() for matrices with complex entries does in fact exist [19]. Unfortunately though using it was unsuccessful, because the implementation of BLAS in opendihu does not, unlike its implementation of LAPACK, use the double _Complex data type for complex numbers. Instead it uses, just like dgemm(), standard double numbers.

Function getMatrixMult() in dmd_utility of opendihu

For the final implementation of the matrix multiplication with real matrices, only the subroutine dgemm() was used as described above. Three other versions are also used, where none uses any BLAS subroutine but were only realized by for loops. One for when all three involved matrices, i.e. both inputs and the output, are complex, one for when the left matrix is real while the right and the product matrices are complex, and the last for when the two input matrices are both double _Complex arrays with negligibly small complex parts, so that only the real parts should be used, hence the resulting matrix is real. [16]

```
void getMatrixMult( double inputA[], double inputB[],
   double output[], int rowsA, int colsA_rowsB, int colsB )

void getMatrixMult( double _Complex inputA[], double
   _Complex inputB[], double _Complex output[], int rowsA,
   int colsA_rowsB, int colsB )

void getMatrixMult( double inputA[], double _Complex
   inputB[], double _Complex output[], int rowsA, int
   colsA_rowsB, int colsB )

void getMatrixMult(double _Complex inputA[], double
   _Complex inputB[], double output[], int rowsA, int
   colsA_rowsB, int colsB)
```

3.2.4 Frobenius and max norm

LAPACK subroutines `dlange()` and `zlange()`

```
double LAPACKE_dlange( int matrix_layout, char norm, int m,
   int n, double a[], int lda )

double LAPACKE_zlange( int matrix_layout, char norm, int m,
   int n, double _Complex a[], int lda )
```

For the norm of a matrix, LAPACK provides the subroutines `dlange()` and `zlange()`, again for real and complex matrices, respectively. Once again, `matrix_layout` has to be set to `LAPACK_COL_MAJOR`. The character `norm` specifies the used norm, where `'f'` corresponds to the Frobenius norm, while `'i'` sets the max norm. `m` and `n` are the dimensions of the input matrix `a`, which has `lda` leading dimensions, meaning equal to `m`. [20] [21]

Functions `getNorm()`, `getRangedNorm()` and `getInfNorm()` in `dmd_utility` of opendihu

```
double getNorm( double input[], int rows, int cols )

double getNorm( double _Complex input[], int order )

double getRangedNorm( double input[], int order, int range )

double getInfNorm( double input[], int rows, int cols )
```

The two functions `getNorm()` compute the Frobenius norm and merely implement the subroutines `dlange()` and `zlange()`, respectively, where `norm` is set to `'f'`. Next, `getRangedNorm()` also computes the Frobenius norm but only of the first `order` entries of a square diagonal matrix with `order` dimensions. Lastly, `getInfNorm()` implements `dlange()` with `char` set to `'i'` and therefore computes the max norm of `input`. [16]

3.2.5 Matrix inversion

LAPACK subroutine `dgetri()`

```
int LAPACKE_dgetri( int matrix_layout, int n, double a[],
   int lda, int ipiv[] )
```

For matrix inversion the LAPACK subroutine `dgetri()` is used. Its first parameter `matrix_layout` sets the order of the matrix that is to be inverted, whereby it's set to `LAPACK_COL_MAJOR`. The subroutine inverts the square real matrix `a` of size `n` with `lda` leading dimensions. The integer array `ipiv` of size `n` is for the pivot indices, where `ipiv[i] = i + 1`. [22]

Function `getMatrixInverse()` in `dmd_utility` of opendihu

```
void getMatrixInverse( double a[], int order )
```

In the implementation of `dgetri()` the subroutine is called with `a` as input matrix and `order` as `n` and `lda`. [16]

3.2.6 Matrix left division

LAPACK subroutine `zgesv()`

```
int LAPACKE_zgesv( int matrix_layout, int n, int nrhs,
   double _Complex a[], int lda, int ipiv[], double
   _Complex b[], int ldb )
```

The subroutine `zgesv()` performs the matrix left division $\texttt{a}\backslash\texttt{b} = X$ with complex matrices, where `b` equals X after the computation. `matrix_layout` is to be set to `LAPACK_COL_MAJOR`. The integers `n`, `lda` and `ldb` all have to equal the number of rows or columns of `a` which is also the number of rows of `b`. `nrhs` is equal to the number of columns of `b`. The pivot indices `ipiv` simply have to be an integer array of size `n` where `ipiv[i] = i + 1`. [23]

Function `getMatrixLeftDivision()` in `dmd_utility` of opendihu

```
void getMatrixLeftDivision( double inputA[], double
   _Complex inputB_output[], int n, int nrhs )
```

The function `getMatrixLeftDivision()` takes the real input matrix `inputA` as `double` array and the complex input matrix `inputB` as `double _Complex` array. It then solves the system of linear equations $\texttt{inputA} \cdot X = \texttt{inputB\_output}$ and stores X in `inputB_output`. [16]

3.2.7 Function `getSpatComp()` in `dmd_utility` of opendihu

```
int getSpatComp( double input[], int rows, int cols, double
   leftSingVec[], double sigma[], double rightSingVec[],
   double epsilon )
```

This function first computes the SVD of the real matrix `input`, so that `input` $\overset{\text{SVD}}{=}$ `leftSingVec` $\cdot$ `singVal` $\cdot$ `rightSingVec`, utilizing the function `getSVD()` in `svd_utility`. Next, the spatial complexity `spatComp` will be calculated by using the functions `getNorm` and `getRangedNorm()` with `singVal` and `epsilon`. Lastly the `integer spatComp` gets returned. [16]

3.2.8 Function `getReducedSnapshotsMatrix()` in `dmd_utility` of opendihu

```
void getReducedSnapshotsMatrix( double sigma[], double
   rightSingVec[], double hatT[], int min, int cols, int
   spatComp )
```

To compute the reduced snapshots matrix, this function first cuts off the rows and columns of `sigma` and the rows of `rightSingVec` that lie outside of the previously calculated spatial complexity `spatComp`. This is done by utilizing the function `resizeMatrix()` and yields `sigmaResized` and `rightSingVecResized`, respectively. To produce the reduced snapshots matrix `hatT`, the two reduced matrices will be multiplied, using `getMatrixMult()`. [16]

3.2.9 Function `getKoopmanOperator()` in `dmd_utility` of opendihu

```
void getKoopmanOperator( double input[], double output[],
   int rows, int cols )
```

For the Koopman operator, a series of functions is used. First, the real matrix `input` gets its last column cut off, using `resizeMatrix()` with `cols` $-\,2$ as parameter value of `lastCol`. This yields the array `withoutLast`, on which now SVD is performed in order to compute its pseudoinverse, utilizing `getSVD()`

from `svd_utility` and resulting in `leftSingVec`, `singVal` and `rightSingVec`. Next, the array `withoutFirst`, which will contain `input` without its first column, will be produced, again with the help of `resizeMatrix()`, with `1` as the value of its parameter `firstCol`. The next steps will compute the Koopman operator

$$\texttt{output} = \texttt{withoutFirst} \cdot \texttt{rightSingVec}^{-1} \cdot \texttt{singVal}^{-1} \cdot \texttt{leftSingVec}^{-1}. \quad (3.18)$$

For the inverse of `rightSingVec` and `leftSingVec`, the transpose is used, as they are unitary matrices, meaning that `transposeMatrix()` is the function utilized here. For the inverse of `singVal` the function `getMatrixInverse()` is used. The remaining multiplications are realized by using `getMatrixMult()`. [16]

3.2.10 Function `getDeltaOmega()` in `dmd_utility` of opendihu

```
void getDeltaOmega( double _Complex eigenvalues[], double
   growthRates[], double frequencies[], int size, double
   deltat )
```

This function computes the growth rates and frequencies of the complex matrix `eigenvalues`. The implementation simply iterates over all entries of `eigenvalues`, computing the natural logarithm with `clog()` and storing the real and imaginary part divided by `deltat` in `growthRates` and `frequencies`, respectively. [16]

3.2.11 Function `getAmplitudes()` in `dmd_utility` of opendihu

```
void getAmplitudes( double snapshots[], double _Complex
   eigenvalues[], double _Complex eigenvectors[], int rows,
   int cols, double _Complex amplitudes[] )
```

This function first creates the `doube _Complex[]` $\texttt{mm}_{\texttt{rows}\times\texttt{rows}}$, sets all entries equal to zero, by using `setZero()`, and stores the entries of `eigenvalues` in its diagonal. Furthermore, four additional `double _Complex[]` are declared: $\texttt{mmm}_{(\texttt{rows}\cdot\texttt{cols})\times\texttt{rows}}$, $\texttt{bb}_{\texttt{rows}\cdot\texttt{cols}}$, $\texttt{mmPower}_{\texttt{rows}\times\texttt{rows}}$ and $\texttt{mmmMxM}_{\texttt{rows}\times\texttt{rows}}$. Next, a `for` loop is used to iterate over the number of columns. In each iteration `k`, the matrix `mm` will be raised to the `k`-th power, utilizing `getMatrixPower()`, and the result gets stored in mmPower. Then, the eigenvectors will be multiplied with mmPower by using getMatrixMult(), resulting in mmmMxM. The entries of mmmMxM will now be transferred to mmm, so that for each iteration, mmm gets filled with a rows $\times$ rows sized matrix, starting at the top. The vector bb will also be filled by transferring the columns of snapshots.

Now, getSVD() from svd_utility is applied to mmm, yielding $\mathtt{ur}_{(\mathtt{rows}\cdot\mathtt{cols})\times\mathtt{rows}}$, $\mathtt{sigmar}_{\mathtt{rows}\times\mathtt{rows}}$, and $\mathtt{vrTransposed}_{\mathtt{rows}\times\mathtt{rows}}$. The amplitudes a of the reduced snapshots matrix can now be calculated,

$$\mathtt{a} = \mathtt{vr} \cdot \mathtt{sigmar} \backslash (\mathtt{ur}^{-1} \cdot \mathtt{bb}). \tag{3.19}$$

Here, transposeMatrix() is used on ur and vrTransposed to produce their respective inverse, getMatrixMult() for the multiplications, and getMatrixLeftDivision() for the backslash-division. [16]

3.2.12 Function getDmdModes() in dmd_utility of opendihu

```
void getDmdModes( double _Complex dmdModes[], double
   _Complex a[], double _Complex eigenvectors[], int rows,
   int cols, double leftSingVec[], double
   leftSingVecReduced[], double amplitudes[] )
```

This function first computes the $\mathtt{dmdModes}_{\mathtt{cols}\times\mathtt{cols}}$ by scaling the eigenvectors by the value of the respective amplitude of the reduced snapshots matrix in a. Since the left singular vectors of the very first SVD weren't reduced to the spatial complexity, yet, this has to be done next, in order to calculate the final amplitudes, by using resizeMatrix(), yielding leftSingVecReduced. Now, a for loop iterates over all columns of the dmdModes. In each iteration the corresponding column is extracted into uColm by using resizeMatrix() and then multiplied by leftSingVecReduced, using getMatrixMult() to yield aca. The corresponding amplitude will be computed by taking the Frobenius norm of aca, utilizing getNorm(), and dividing it by the square root of rows. [16]

3.2.13 Function `getSpecComp()` in `dmd_utility` of opendihu

```
int getSpecComp( double _Complex dmdModes[], double
   growthRates[], double frequencies[], double
   amplitudes[], int order, double epsilon0 )
```

The first step of `getSpecComp()` is to sort the `dmdModes` by the value of their `amplitudes`. To do this, they first have to be transposed, using `transposeMatrix()` and yielding `dmdModesT`, in order to have the modes row-wise instead of column-wise. Next the `growthRates`, `frequencies`, and `amplitudes` are concatenated to `dmdModesT`, utilizing the function `concatenateVector()` and resulting in $\texttt{uu}_{\texttt{order}\times(\texttt{order}+3)}$. Now the sorting will be processed by the function `sortMatrix()`, where the result gets stored in `uu1`. This matrix gets transposed with `transposeMatrix()`, so that the now sorted `dmdModes` can be extracted, by using `resizeMatrix()`. Furthermore, the corresponding `growthRates`, `frequencies` and `amplitudes` get extracted in a similar way but by using a simple `for` loop to iterate over their entries. Lastly, the spectral complexity `specComp` will be computed, by iterating over the entries of `amplitudes` and using `epsilon0`, and then returned. [16]

3.2.14 Function `getDmdModesGrowthRatesFrequencies()` in `dmd_utility` of opendihu

```
void getDmdModesGrowthRatesFrequencies( double _Complex
   dmdModes[], double growthRates[], double frequencies[],
   double _Complex dmdModesReduced[], double
   growthRatesReduced[], double frequenciesReduced[], int
   rows, int cols, int compSpec )
```

After computing the spectral complexity, the `dmdModes`, `growthRates`, and `frequencies` have to be reduced to match `specComp`. For the `dmdModes`, this is realized by using `resizeMatrix()`, while the `growthRates` and `frequencies` get reduced by iterating over their first `specComp` entries and only storing these in their respective reduced variant. [16]

3.2.15 Function `reconstructSnapshots()` in `dmd_utility` of opendihu

```
void reconstructSnapshots( double _Complex dmdModes[],
   double growthRates[], double frequencies[], double
   leftSingVec[], double snapshotsReconst[], int rows, int
   cols, int specComp, int spatComp, double deltat )
```

To reconstruct the snapshots, the reduced snapshots have to be reconstructed first. This will be done column by column, so that `hatTreconst` will be composed of the current `hatTreconstCol` for each iteration. The function `contReconst()` is used here `cols` times, to compute each column of the reconstructed reduced snapshots matrix `hatTreconst`. Finally, `getMatrixMult()` is used one last time to reverse the first SVD and compute the reconstructed snapshots matrix `snapshotsReconst`. [16]

3.2.16 Function `resizeMatrix()` in `dmd_utility` of opendihu

```
void resizeMatrix( double input[], double output[], int
   oldRows, int newRows, int firstCol, int lastCol )

void resizeMatrix( double _Complex input[], double _Complex
   output[], int oldRows, int newRows, int firstCol, int
   lastCol )
```

These functions take real or complex matrix as `input` and store the resized version in `output`. The original number of rows `oldRows` has to be specified, as well as the desired number of rows `newRows`. Every row after the first `newRows` will be omitted. For the columns, the first and last column are selected, starting the index at `0`. [16]

3.2.17 Function `printMatrix()` in `dmd_utility` of opendihu

```
void printMatrix( std::string name, double input[], int
   rows, int cols )

void printMatrix( std::string name, double _Complex
   input[], int rows, int cols )
```

These were mainly used for debugging purposes, as they simply print out the entries of the matrix $\texttt{input}_{\texttt{rows}\times\texttt{cols}}$, as well as its `name`. [16]

3.2.18 Function `transposeMatrix()` in `dmd_utility` of opendihu

```
void transposeMatrix( double input[], double output[], int
   rows, int cols )

void transposeMatrix( double _Complex input[], double
   _Complex output[], int rows, int cols )
```

These functions take a real or complex matrix $\texttt{input}_{\texttt{rows}\times\texttt{cols}}$ and store its transpose or conjugate transpose in $\texttt{output}_{\texttt{cols}\times\texttt{rows}}$. [16]

3.2.19 Function `setZero()` in `dmd_utility` of opendihu

```
void setZero( double input[], int rows, int cols )

void setZero( double _Complex input[], int rows, int cols )
```

These iterate over the entire real or complex matrix $\texttt{input}_{\texttt{rows}\times\texttt{cols}}$ and set every entry equal to zero. [16]

3.2.20 Function `getMatrixPower()` in `dmd_utility` of opendihu

```
getMatrixPower( double _Complex input[], double _Complex
   output[], int order, int exponent )
```

This function takes a complex square diagonal matrix $\texttt{input}_{\texttt{order}\times\texttt{order}}$ and raises that to the specified power `exponent`. The result is then stored in `output`. [16]

3.2.21 Function `concatenateVector()` in `dmd_utility` of opendihu

```
concatenateVector( double _Complex inputA[], double
   inputB[], double _Complex output[], int rows, int cols )
```

To concatenate a real vector $\texttt{inputB}_{\texttt{rows}}$ onto a complex matrix $\texttt{inputA}_{\texttt{rows}\times\texttt{cols}}$, this function simply first transfers `inputA` into the first `cols` columns of the complex matrix $\texttt{output}_{\texttt{rows}\times\texttt{cols}+1}$ and then appends the vector `inputB` into the last column. [16]

3.2.22 Function `sortMatrix()` in `dmd_utility` of opendihu

```
sortMatrix( double _Complex input[], double _Complex
   output[], int rows, int cols )
```

This function first makes a copy of the complex matrix $\texttt{input}_{\texttt{rows}\times\texttt{cols}}$, `inputCopy`, so that the original matrix won't be overwritten. It then proceeds to sort the matrix by the real part of the values, where all values should be greater or equal to 0, of its last column by performing selection sort. This is achieved by first iterating over all entries of said column and finding the `greatestValue`, so that its row's index can be stored in `greatestRow`. Next, it iterates over every entry of this row and transfers its contents into the next vacant row of `output`. After this, the value of the last entry of that row will be set equal to `-1`, so that it is now among the lowest values. [16]

3.2.23 Function `contReconst()` in `dmd_utility` of opendihu

```
void contReconst( double t, double t0, double _Complex
   dmdModes[], double growthRates[], double frequencies[],
   int rows, int cols, double output[] )
```

This function is used to reconstruct the single columns of the reduced snapshots matrix and stores the result in $\texttt{output}_{\texttt{rows}}$. It first declares the complex vector $\texttt{vv}_{\texttt{cols}}$ and sets every entry equal to 0, by using `setZero()`. Next, it computes each entry `m` of `vv`, by calculating

$$\texttt{vv}_{\texttt{m}} = e^{(\texttt{growthRates}_{\texttt{m}} + i\cdot\texttt{frequencies}_{\texttt{m}})(\texttt{t}-\texttt{t0})}, \tag{3.20}$$

where the natural exponential function is realized by utilizing `cexp()`. Afterwards, the final values of `output` get calculated by performing matrix multiplication on the $\texttt{dmdModes}_{\texttt{rows}\times\texttt{cols}}$ and `vv`, using `getMatrixMult()`. [16]

4 Analysis

4.1 Example snapshots matrix

Following is the detailed applied DMD method on a comparatively small snapshots matrix,

$$V = \begin{bmatrix} 8.79 & 9.93 & 9.83 & 5.45 & 3.16 \\ 6.11 & 6.91 & 5.04 & -0.27 & 7.98 \\ -9.15 & -7.93 & 4.86 & 4.85 & 3.01 \\ 9.57 & 1.64 & 8.83 & 0.74 & 5.80 \\ -3.49 & 4.02 & 9.80 & 10.00 & 4.27 \\ 9.84 & 0.15 & -8.99 & -6.02 & -5.31 \end{bmatrix}, \tag{4.1}$$

with $j = 6$ rows and $k = 5$ columns, which are the parameters and snapshots, respectively.

The first step computes the singular value decomposition of the snapshots matrix V, which yields the left-singular vectors U, the singular values Σ and the right-singular vectors T^{-1}, as shown in equation 3.1,

$$\begin{aligned} U &= \begin{bmatrix} -0.591142 & 0.263168 & 0.3554300 & 0.314264 & 0.229938 \\ -0.397567 & 0.243799 & -0.2223900 & -0.753466 & -0.363590 \\ -0.033479 & -0.600273 & -0.4508390 & 0.233450 & -0.305476 \\ -0.429707 & 0.236167 & -0.6858630 & 0.331860 & 0.164928 \\ -0.469748 & -0.350891 & 0.3874450 & 0.158736 & -0.518257 \\ 0.293359 & 0.576262 & -0.0208529 & 0.379078 & -0.652552 \end{bmatrix} \\ \Sigma &= \begin{bmatrix} 27.4687 & & & & \\ & 22.6432 & & & \\ & & 8.55839 & & \\ & & & 5.98572 & \\ & & & & 2.0149 \end{bmatrix} \\ T^{-1} &= \begin{bmatrix} -0.251383 & -0.396846 & -0.692151 & -0.366170 & -0.4076350 \\ 0.814837 & 0.358662 & -0.248888 & -0.368594 & -0.0979626 \\ -0.260619 & 0.700768 & -0.220811 & 0.385938 & -0.4932500 \\ 0.396724 & -0.450711 & 0.251321 & 0.434249 & -0.6226840 \\ -0.218028 & 0.140210 & 0.589119 & -0.626528 & -0.4395520 \end{bmatrix}. \end{aligned} \tag{4.2}$$

Next, the spatial complexity k_{spat} is computed, equation 3.2, with the chosen $\epsilon_1 = e^{-2} \approx 0.135335$ and results for this example in $k_{\text{spat}} = 4$. This means, that now only the first 4 columns of U, entries of Σ and rows of T^{-1}, which are all highlighted in the respective matrices, will be used in the remaining procedures.

The reduced snapshots matrix from equation 3.3 is calculated by multiplying the reduced singular values with the reduced right-singular values, $\hat{T} = \Sigma T^*$,

$$\hat{T} = \begin{bmatrix} -6.90517 & -10.90080 & -19.01250 & -10.05820 & -11.19720 \\ 18.45050 & 8.12124 & -5.63562 & -8.34613 & -2.21818 \\ -2.23047 & 5.99745 & -1.88979 & 3.30301 & -4.22143 \\ 2.37468 & -2.69783 & 1.50434 & 2.59929 & -3.72721 \end{bmatrix}. \tag{4.3}$$

After this, the modified Koopman operator $\hat{R}$ will be computed as shown in equation 3.5 and results in

$$\hat{R} = \begin{bmatrix} 0.75011 & -0.408482 & -1.57524 & -0.715062 \\ 0.452508 & 0.390752 & 0.204795 & 1.89209 \\ -0.156436 & 0.188596 & -1.00869 & -0.342071 \\ -0.170374 & -0.0870758 & -0.641562 & -1.55755 \end{bmatrix} \tag{4.4}$$

The eigendecomposition in equation 3.6 now yields for the eigenvalues Λ and eigenvectors Q of $\hat{R}$

$$\Lambda = \begin{bmatrix} -1.96024 & & & \\ & 0.57413 + 0.287924i & & \\ & & 0.57413 - 0.287924i & \\ & & & -0.613397 \end{bmatrix},$$

$$Q = \begin{bmatrix} -0.304603 & 0.7633660 & 0.7633660 & -0.481405 \\ 0.598140 & 0.4965710-0.4006190i & 0.4965710+0.4006190i & -0.563874 \\ -0.394273 & -0.0106582-0.0550223i & -0.0106582+0.0550223i & -0.482279 \\ -0.627689 & -0.0723216+0.0426929i & -0.0723216-0.0426929i & 0.466586 \end{bmatrix}. \tag{4.5}$$

Now, the growth rates δ and frequencies ω get computed as shown in equation 3.8, where $\Delta t = 0.1$ and yield respectively

$$\delta = \begin{bmatrix} 6.73068 \\ -4.42729 \\ -4.42729 \\ -4.88743 \end{bmatrix}, \qquad \omega = \begin{bmatrix} 31.4159 \\ 4.64844 \\ -4.64844 \\ 31.4159 \end{bmatrix}. \tag{4.6}$$

The amplitudes of the reduced snapshots matrix are being calculated in equation 3.12 and result for this example in

$$a = \begin{bmatrix} 0.578684 \\ -0.811451+31.627i \\ -0.811451-31.627i \\ 11.404200 \end{bmatrix}. \tag{4.7}$$

In equation 3.13 the DMD modes are computed, where this example yields

$$u = \begin{bmatrix} -0.591142 & -0.469748 & -0.600273 & 0.355430 \\ -0.397567 & 0.293359 & 0.236167 & -0.222390 \\ -0.033479 & 0.263168 & -0.350891 & -0.450839 \\ -0.429707 & 0.243799 & 0.576262 & -0.685863 \end{bmatrix} \tag{4.8}$$

Now, the amplitudes of the snapshots matrix V can be calculated with equation 3.14, where this α is already sorted.

$$\alpha = \begin{bmatrix} 12.915900 \\ 12.915900 \\ 4.655740 \\ 0.236247 \end{bmatrix} \tag{4.9}$$

The spectral complexity k_{spec} can now easily be computed as shown in equation 3.15, where $\epsilon_0 = e^{-1} \approx 0.367879$ and results in $k_{\text{spec}} = 2$. This means that only the first 2 growth rates, frequencies, amplitudes and DMD modes are being used for the reconstruction step.

For the reconstruction of the reduced snapshots matrix $\hat{T}$, the calculation in equation 3.16 is used and results in

$$\hat{T}_{\text{reconst}} = \begin{bmatrix} -1.23887 & -14.613900 & -16.269500 & -12.65300 & -7.8173600 \\ 24.53490 & 4.855280 & -4.546130 & -7.223060 & -6.4185600 \\ 3.49768 & 2.176520 & 1.056330 & 0.315071 & -0.0739792 \\ -2.58312 & -0.145952 & 0.898012 & 1.091360 & 0.8827110 \end{bmatrix}. \tag{4.10}$$

This leads to the reconstruction of the snapshots matrix V in equation 3.17, where

$$V_{\text{reconst}} = \begin{bmatrix} 7.62054 & 10.64440 & 9.07888 & 6.03382 & 3.183130 \\ 7.64256 & 6.61967 & 4.44834 & 2.37707 & 0.894444 \\ -16.86610 & -3.44057 & 3.00701 & 4.87215 & 4.354020 \\ 3.07051 & 5.88514 & 5.49100 & 3.87733 & 2.187000 \\ -7.08200 & 5.98131 & 9.78959 & 8.77355 & 6.035860 \\ 12.72300 & -1.58993 & -7.07418 & -7.46711 & -5.655900 \end{bmatrix}. \tag{4.11}$$

Evidently, this result varies relatively much from the original snapshots matrix V, which will also be seen in the errors. For these, the equations 4.13 and 4.14 were used for the root-mean-square error and the max error, respectively.

$$E_{\text{RMS}} = 0.450876 \qquad\qquad E_{\text{Max}} = 0.560655 \tag{4.12}$$

Reasons for these errors can be speculated to be connected to the snapshots matrix, which first of all doesn't provide a lot of snapshots. Furthermore, the data isn't actually realistic, since each snapshot is completely independent from each other, because it only consists of random numbers. The choice of ϵ_1 and ϵ_0 also played a role. They weren't set to be very small so that the DMD will actually reduce the dimensions but that comes, especially with this snapshot data, with the price of quite big errors.

4.2 Testing

For the testing of the final implementation of the dynamic mode decomposition, a snapshots matrix was created with the MATLAB program `monodomain_1D_Order1`. This matrix has a total of 68 grid points and 600 snapshots, with time step width $\Delta t = 0.01$s.

4.2.1 Reduction of dimensions

Different combinations of values for ϵ_0 and ϵ_1 were chosen. First, the significance of the reduction in dimensions is measured by choosing 0 for ϵ_0 when testing k_{spat} and choosing 0 for ϵ_1 when testing k_{spec}. Since the calculation step for the spatial complexity is only dependant on ϵ_1, the value of ϵ_0 does not matter. For the spectral complexity it is different, since it can always only be at most equal to the spatial complexity, thus setting $\epsilon_1 = 0$ leaves all 68 dimensions for the spectral complexity. The respective significant ϵ will then be increased, first to e^{-15} and then in each step multiplied by e, i.e. the next step is e^{-14} and so on, up to e^{-1}. The results for the spatial complexity can be seen in table 4.1; the results for the spectral complexity are depicted in table 4.2.

Now, the reduction in dimensions, which is depicted by the reduction of the respective k, can be seen in relation to their respective ϵ. For the changes in ϵ_1, the spatial complexity k_{spat} already more than halves, i.e. it goes from being 68 initially to only 31, by setting $\epsilon_1 = e^{-15} \approx 3.05902 \times 10^{-1}$. Comparable results are achieved with the varying of ϵ_0 and reduction of k_{spec} only between $\epsilon_0 = e^{-6} \approx 2.47875 \times 10^{-3}$ and $\epsilon_0 = e^{-5} \approx 6.73795 \times 10^{-3}$, where $k_{\text{spec}} = 37$ and $k_{\text{spec}} = 21$, respectively. This suggests that the value of ϵ_1 can be kept very small, while the value of ϵ_0, in order for it to achieve a significant reduction, should be chosen to be greater.

Table 4.1: $\epsilon_0 = 0$ to show the effect of ϵ_1

ϵ_1	k_{spat}
0	68
$e^{-15} \approx 3.05902 \times 10^{-7}$	31
$e^{-14} \approx 8.31529 \times 10^{-7}$	30
$e^{-13} \approx 2.26033 \times 10^{-6}$	27
$e^{-12} \approx 6.14421 \times 10^{-6}$	23
$e^{-11} \approx 1.67017 \times 10^{-5}$	21
$e^{-10} \approx 4.53999 \times 10^{-5}$	17
$e^{-9} \approx 1.2341 \times 10^{-4}$	13
$e^{-8} \approx 3.35463 \times 10^{-4}$	12
$e^{-7} \approx 9.11882 \times 10^{-4}$	10
$e^{-6} \approx 2.47875 \times 10^{-3}$	9
$e^{-5} \approx 6.73795 \times 10^{-3}$	9
$e^{-4} \approx 1.83156 \times 10^{-2}$	8
$e^{-3} \approx 4.97871 \times 10^{-2}$	6
$e^{-2} \approx 1.35335 \times 10^{-1}$	3
$e^{-1} \approx 3.67879 \times 10^{-1}$	2

Table 4.2: $\epsilon_1 = 0$ to show the effect of ϵ_0

ϵ_0	k_{spec}
0	68
$e^{-15} \approx 3.05902 \times 10^{-7}$	64
$e^{-14} \approx 8.31529 \times 10^{-7}$	64
$e^{-13} \approx 2.26033 \times 10^{-6}$	64
$e^{-12} \approx 6.14421 \times 10^{-6}$	64
$e^{-11} \approx 1.67017 \times 10^{-5}$	64
$e^{-10} \approx 4.53999 \times 10^{-5}$	64
$e^{-9} \approx 1.2341 \times 10^{-4}$	64
$e^{-8} \approx 3.35463 \times 10^{-4}$	56
$e^{-7} \approx 9.11882 \times 10^{-4}$	51
$e^{-6} \approx 2.47875 \times 10^{-3}$	37
$e^{-5} \approx 6.73795 \times 10^{-3}$	21
$e^{-4} \approx 1.83156 \times 10^{-2}$	9
$e^{-3} \approx 4.97871 \times 10^{-2}$	7
$e^{-2} \approx 1.35335 \times 10^{-1}$	2
$e^{-1} \approx 3.67879 \times 10^{-1}$	2

4.2.2 Errors

Next, the errors were measured. For this, every combination of the values for ϵ_1 and ϵ_0 from tables 4.1 and 4.2 were tested. The results for the root-mean-square error E_{RMS} can be seen in figure 4.1, where ϵ_1 is located on the x-axis and the error value on the y-axis, while the different values for ϵ_0 are shown as different graphs. To calculate E_{RMS}, the Frobenius norm of the difference of the original snapshots matrix V and the reconstructed snapshots matrix V_{reconst} is divided by the Frobenius norm of just the original snapshots matrix V,

$$E_{\text{RMS}} = \frac{\|V - V_{\text{reconst}}\|_2}{\|V\|_2}. \tag{4.13}$$

The maximal error E_{Max} was also computed and is shown in figure 4.2, again with the value of ϵ_1 on the x-axis, the value of E_{Max} on the y-axis and the values for ϵ_0 as the graphs. The calculation of the maximal error is similar to the one of the root-mean-square error, but instead of the Frobenius norm, the max norm is used,

$$E_{\text{Max}} = \frac{\|V - V_{\text{reconst}}\|_\infty}{\|V\|_\infty}. \tag{4.14}$$

In both figures it can be observed that by choosing $\epsilon_1 \geq e^{-4} \approx 1.83156 \times 10^{-2}$, the value of ϵ_0 doesn't make a difference. This shows that ϵ_1 should be at most e^{-5},

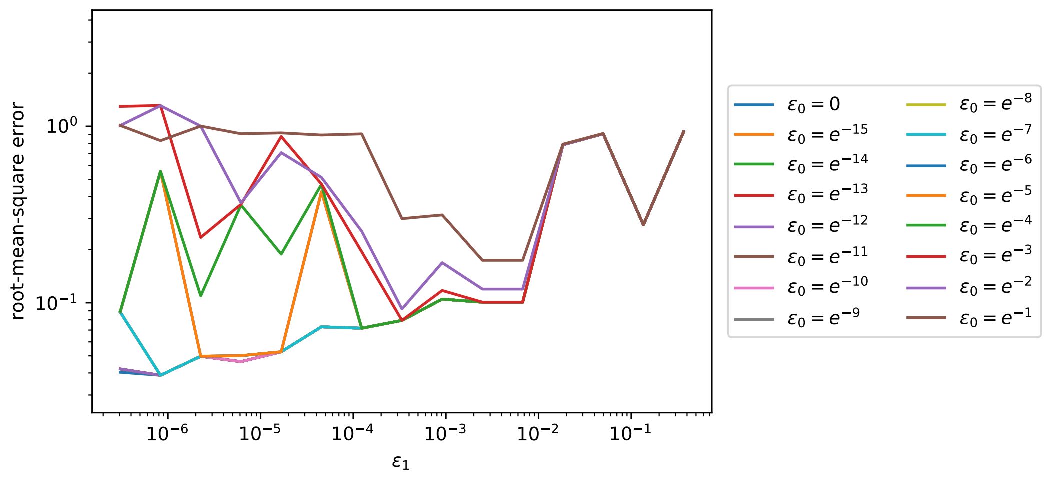


Figure 4.1: The root-mean-square error E_{RMS} of the DMD implementation.

which is further supported by the jump of the errors from e^{-4} to e^{-5}. These results suggest that good values to start further testing of the DMD might be $\epsilon_1 = e^{-5}$ and $\epsilon_0 = e^{-3}$ and decreasing their values evenly, since it can be seen that by just decreasing ϵ_1 for example, the error might do unexpected jumps upwards.

4.2.3 Runtime

For measuring the runtime of the DMD implementation the same values for ϵ_1 and ϵ_0 as above have been chosen. The results are shown in figure 4.3, where, as before, ϵ_1 is located on the x-axis, the y-axis is for the runtime in seconds and the graphs are for the different values of ϵ_0. As was expected, the runtime is anti-proportional to both, ϵ_1 and ϵ_0. There are no significant jumps other than small outliers that might be attributed to other reasons, such as unfortunate CPU scheduling.

4.2.4 Frequencies, growth rates and amplitudes

For the plots of the amplitudes α in figure 4.5 and the growth rates δ in figure 4.4 with their respective frequencies ω, both, ϵ_0 and ϵ_1, were chosen to be $e^{-15} \approx 3.05902 \times 10^{-7}$. This way, there is only limited reduction in dimensions, so that there are enough modes show the scattering.

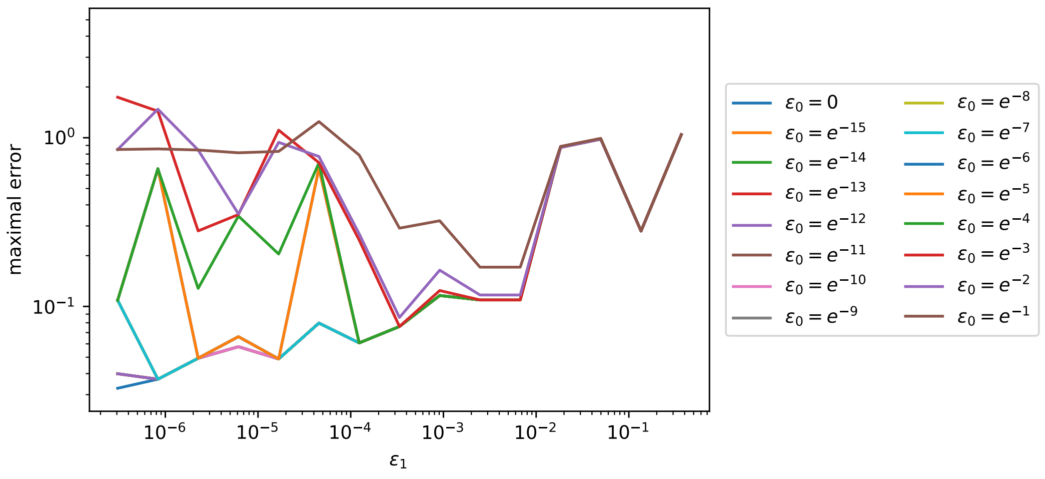


Figure 4.2: The maximal error E_{Max} of the DMD implementation.

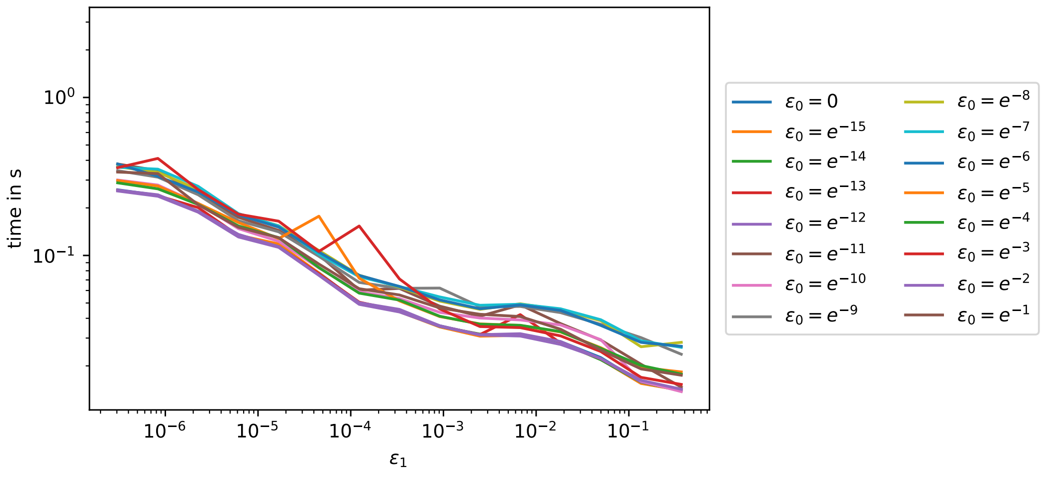


Figure 4.3: The runtime of the DMD implementation in seconds.

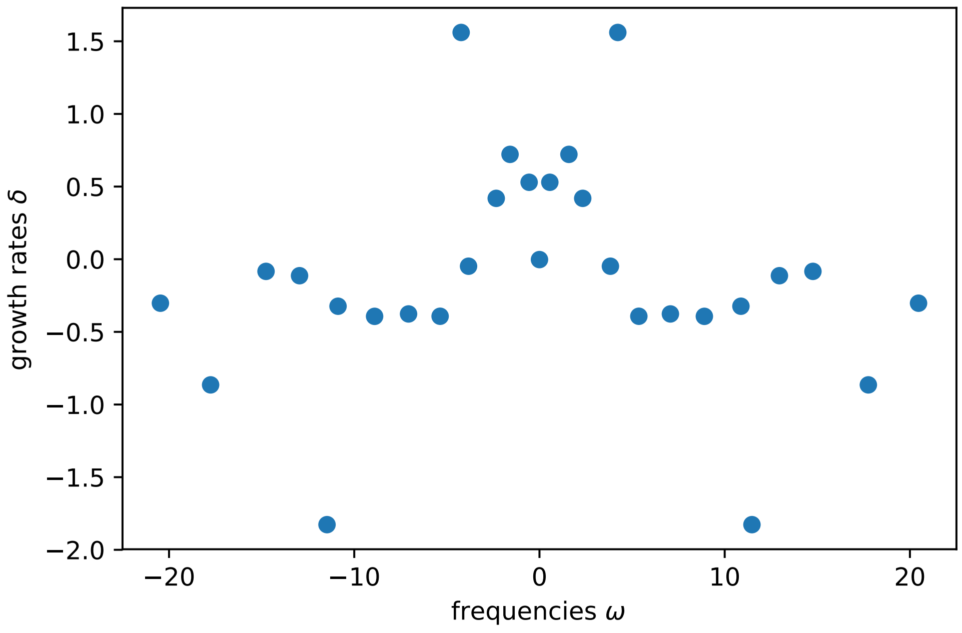


Figure 4.4: The growth rates δ in relation to their frequencies ω.

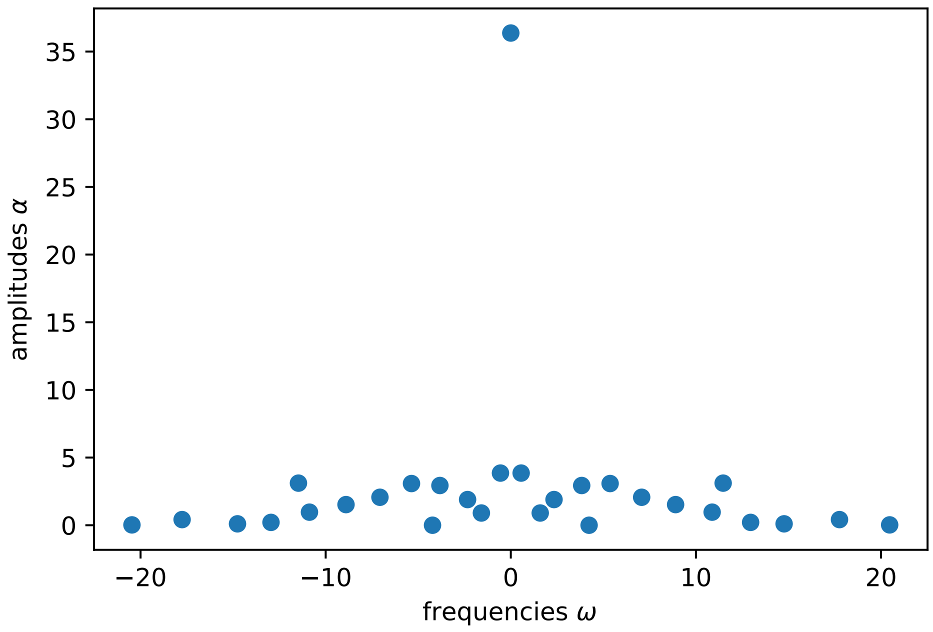


Figure 4.5: The amplitudes α in relation to their frequencies ω.

5 Outlook

The tests show promising results for further pursuing the DMD approach for model order reduction of monodomain equation models. More investigation has to be done with different models and parameter values, in order to draw a more definite conclusion.

The next steps could involve increasing the performance of the DMD implementation. For this, parallelization with ScaLAPACK [24] could be used, so that all the subroutines, currently only implemented with LAPACK, could be made more efficient. Especially for the computational excessive procedures, such as singular value decomposition, matrix multiplication and eigendecomposition, this might achieve a substantial performance gain. In order to take full advantage of the parallelization capabilities of ScaLAPACK, matrix multiplication with complex numbers should also be implemented to use the subroutine `zgemm()` by LAPACK. Another, less meaningful optimization may include a better sorting algorithm for the reduction step of the spectral complexity. Here, much better sorting algorithms exist, that unfortunately couldn't be implemented, yet.

With these optimizations in mind, a final big step forward should consist of implementing the DMD-d algorithm. Whereas this thesis only considered the standard DMD, also written as DMD-1 to signify the one-dimensional DMD, there exists the higher order dynamic mode decomposition (HODMD), denoted as DMD-d, where d equals to the order of the HODMD. For this, the higher order Koopman assumption is applied, so that not only one time step will be used for the computation of the Koopman operator, as in the DMD-1 algorithm, but d time steps [25]. To implement this, the C++ code can be extended to also implement HODMD, since the entire algorithm has been made modular, so that the single steps of the DMD should be expandable to fit DMD-d.

Bibliography

[1] Monica Aleman, Yvette S Nout-Lomas, and Stephen M Reed. "Disorders of the Neurologic System". In: *Equnie Internal Medicine.* Vol. 4. Elsevier, 2018.

[2] Neil J Smelser, Paul B Baltes, et al. "Neuromuscular System". In: *International encyclopedia of the social & behavioral sciences.* Vol. 11. Elsevier Amsterdam, 2001.

[3] James Keener and James Sneyd. "Muscle". In: *Mathematical Physiology: II: Systems Physiology.* New York, NY: Springer New York, 2009, pp. 717–772.

[4] Craig S Henriquez. "Simulating the electrical behavior of cardiac tissue using the bidomain model." In: *Critical reviews in biomedical engineering* 21.1 (1993), pp. 1–77.

[5] Birgit Stender. "Parametrization of activation based cardiac electrophysiology models using bidomain model simulations". In: *Current Directions in Biomedical Engineering* 2.1 (2016), pp. 611–615.

[6] Alan L Hodgkin and Andrew F Huxley. "A quantitative description of membrane current and its application to conduction and excitation in nerve". In: *The Journal of physiology* 117.4 (1952), pp. 500–544.

[7] Paul R Shorten et al. "A mathematical model of fatigue in skeletal muscle force contraction". In: *Journal of muscle research and cell motility* 28.6 (2007), pp. 293–313.

[8] Oliver Röhrle. *DiHu.* 2016. URL: `https://ipvs.informatik.uni-stuttgart.de/SGS/digital_human/` (visited on 04/03/2019).

[9] Peter J Schmid. "Dynamic mode decomposition of numerical and experimental data". In: *Journal of fluid mechanics* 656 (2010), pp. 5–28.

[10] Soledad Le Clainche and José M Vega. "Higher order dynamic mode decomposition". In: *SIAM Journal on Applied Dynamical Systems* 16.2 (2017), pp. 882–925.

[11] James Weldon Demmel. "Computing stable eigendecompositions of matrices". In: *Linear Algebra and its Applications* 79 (1986), pp. 163–193.

[12] Gene Golub and William Kahan. "Calculating the singular values and pseudo-inverse of a matrix". In: *Journal of the Society for Industrial and Applied Mathematics, Series B: Numerical Analysis* 2.2 (1965), pp. 205–224.

[13] Benjamin Maier. *opendihu*. 2017. URL: `https://github.com/maierbn/opendihu` (visited on 04/03/2019).

[14] Netlib. *dgesvd*. URL: `http://www.netlib.org/lapack/explore-html/d1/d7e/group__double_g_esing_ga84fdf22a62b12ff364621e4713ce02f2.html` (visited on 04/03/2019).

[15] Netlib. *zgesvd*. URL: `http://www.netlib.org/lapack/explore-html/d3/da8/group__complex16_g_esing_gad6f0c85f3cca2968e1ef901d2b6014ee.html` (visited on 04/06/2019).

[16] Moritz Widmayer. *dmd_utility.cpp*. 2019. URL: `https://github.com/maierbn/opendihu/blob/develop_DMD/core/src/utility/dmd_utility.cpp` (visited on 04/06/2019).

[17] Netlib. *dgeev*. URL: `http://www.netlib.org/lapack/explore-html/d9/d8e/group__double_g_eeigen_ga66e19253344358f5dee1e60502b9e96f.html#ga66e19253344358f5dee1e60502b9e96f` (visited on 04/06/2019).

[18] Netlib. *dgemm*. URL: `http://www.netlib.org/lapack/explore-html/d1/d54/group__double__blas__level3_gaeda3cbd99c8fb834a60a6412878226e1.html` (visited on 04/06/2019).

[19] Netlib. *zgemm*. URL: `http://www.netlib.org/lapack/explore-html/dc/d17/group__complex16__blas__level3_ga4ef748ade85e685b8b2241a7c56dd21c.html` (visited on 04/06/2019).

[20] Netlib. *dlange*. URL: `http://www.netlib.org/lapack/explore-html/de/d39/group__double_g_eauxiliary_gaefa80dbd8cd1732740478618b8b622a1.html` (visited on 04/06/2019).

[21] Netlib. *zlange*. URL: `http://www.netlib.org/lapack/explore-html/d0/d9e/group__complex16_g_eauxiliary_ga7908bb12a6f02dbfa4d5a92a27c0e9b7.html` (visited on 04/06/2019).

[22] Netlib. *dgetri*. URL: `http://www.netlib.org/lapack/explore-html/dd/d9a/group__double_g_ecomputational_ga56d9c860ce4ce42ded7f914fdb0683ff.html` (visited on 04/06/2019).

[23] Netlib. *zgesv*. URL: `http://www.netlib.org/lapack/explore-html/d6/d10/group__complex16_g_esolve_ga531713dfc62bc5df387b7bb486a9deeb.html` (visited on 04/06/2019).

[24] Netlib. *ScaLAPACK*. 2017. URL: `http://www.netlib.org/scalapack/` (visited on 04/10/2019).

[25] Soledad Le Clainche and José M Vega. “Higher order dynamic mode decomposition to identify and extrapolate flow patterns”. In: *Physics of Fluids* 29.8 (2017), p. 084102.