Dymode

A parallel dynamic mode decomposition software

Romain Futrzynski
fromain@kth.se
Linné Flow Centre
Aeronautical and Vehicle Engineering

Gunilla Efraimsson
gef@kth.se
Centre for ECO² Vehicle Design
Aeronautical and Vehicle Engineering
Dymode is distributed freely on a limited basis. It comes with no guarantees whatsoever. Problems can be reported to fromain@kth.se or gef@kth.se, but no action is promised. If results obtained by using the programs included in the dymode distribution are published, the authors would like an acknowledgment, e.g. in the form of referencing this manual.
## Contents

1. **Introduction** ........................................................................................................................................... 3
   1.1 Software description ........................................................................................................................... 3

2. **Theory** .................................................................................................................................................. 5
   2.1 Flow decompositions.......................................................................................................................... 5
   2.2 The dynamic mode decomposition ................................................................................................. 6
   2.3 DMD computation in dymode ........................................................................................................ 10

3. **Implementation** ................................................................................................................................... 13
   3.1 Computational options ................................................................................................................... 13
   3.2 Input / Output .................................................................................................................................. 14
   3.3 Singular value decomposition ......................................................................................................... 18
   3.4 Similar matrix formation ................................................................................................................. 18
   3.5 Eigen decomposition ....................................................................................................................... 18
   3.6 Linear system solving ...................................................................................................................... 19
   3.7 Sorting ............................................................................................................................................. 20

4. **Building dymode** ................................................................................................................................ 23
   4.1 Dependencies .................................................................................................................................. 23
   4.2 Dymode ........................................................................................................................................... 24

5. **Usage** .................................................................................................................................................. 27

6. **Parallel scaling** .................................................................................................................................... 29

7. **Dymodem** .......................................................................................................................................... 31

8. **References** .......................................................................................................................................... 33
1 Introduction
While numerical simulations of complex turbulent flows have become more and more common, primarily due to the development of high performance computing, visualization and post-processing remains challenging. From numerical or experimental data, one can compute time averaged quantities such as a mean velocity profile or a mean turbulent kinetic energy profile. Particular post-processing such as phase averaging can also be employed if the flow is known to exhibit specific frequencies. Alternatively significant frequencies in the flow can be identified, for instance, by means of a Fourier transform. However such an operation might overlook structures that are coherent not only in time but also in space.

For this reason mode decompositions are often used when structures with well-defined properties, such as energy or time-dependency, are to be identified without any a priori knowledge of a flow. Once the modes have been found, they can be used in applications ranging from visualizing a flow more easily to deriving reduced order models.

1.1 Software description
Dymode is a parallel program that computes dynamic mode decompositions. A description of the underlying theory is given in sec. 2 Theory. The code is written in C++ and relies on a number of libraries listed in sec. 4 Building dymode.

Several parameters can be specified in order to control the computational aspects of the program as well as the input and output of the decomposition, particularly how the modes are sorted. A description of these parameters and details about the implementation are given in sec. 3 Implementation. Finally, dymode is almost entirely parallel and is therefore particularly suitable for computing the dynamic mode decomposition of large datasets.

The dymode package also includes dymodem, a Matlab implementation of the code which accepts the same arguments as dymode, when they are relevant, and produces the same output. It can be useful to use dymodem when dealing with smaller datasets, or to validate the output from dymode. Its usage is described in sec. 7 Dymodem.
2 Theory

2.1 Flow decompositions

When a particular flow is too complex to analyze and/or understand as a whole, it is reasonable to decompose this flow into modes that can be displayed one by one or by groups of relatively few. In other words, one is looking to write the flow as a sum of spatial structures $\varphi_j$ evolving in time according to a function $\alpha_j(t)$ as follows:

$$s(t) = \sum_{j=1}^{N} \alpha_j(t) \varphi_j,$$  \hspace{1cm} (2.1)

where $s(t)$ is called a snapshot of the flow at time $t$; that is, the elements of $s(t)$ contain values that describe the flow at time $t$ at various spatial coordinates. Space is used in a general sense here and coordinates identify both a point in space and a particular quantity. For instance, in a flow described by two components of the velocity as well as by the pressure ($u$, $v$ and $p$), where values for these quantities are obtained at $N_p$ points in physical space, then a possible layout for the snapshot would be

$$s = \begin{bmatrix} u_1 \\ u_{N_p} \\ v_1 \\ v_{N_p} \\ p_1 \\ p_{N_p} \end{bmatrix}.$$

The problem is now to find pairs $\{\alpha_j(t), \varphi_j\}_{j \in \{1,N\}}$ that

- satisfy (2.1),
- are adequate for the desired use.

Several methods have been proposed to find such pairs, and perhaps the most used is currently the proper orthogonal decomposition (POD), which was introduced in 1967 (Lumley 1967) and has been used extensively ever since. The principle of POD is to create an orthogonal base that spans the space of $s(t)$, choose the vectors $\varphi_j$ as the base vectors, then find the associated $\alpha_j(t)$ that minimize energy distribution. The advantage of this method is that the energy of the flow is distributed in as few modes as possible. Detailed derivations are provided, e.g., by Cazemier, Verstappen, and Veldman (1998) or Manhart and Wengle (1993).

Another method, named dynamic mode decomposition (DMD), has been introduced relatively recently to fluid mechanics (Rowley et al. 2009; Schmid et al. 2010), although it is based upon long lived ideas of linear algebra and dynamical systems. The modes obtained from DMD, rather than being orthogonal in space between them, present the particularity of being periodic in time. The atten-
tion to DMD has been rapidly increasing due to its ability to extract coherent spatial structures presenting a frequency-based temporal evolution which is both intuitive and common in signal processing.

Even newer types of decomposition, such as the optimized DMD (Chen, Tu, and Rowley 2012) and optimal mode decomposition (Wynn et al. 2013), use modified versions of the DMD in order to decompose a flow into an arbitrary number of modes.

In its current state, dymode performs only classic DMD of a flow, and the theory behind it is presented here.

2.2 The dynamic mode decomposition

Under the assumption that the flow is composed only of periodically oscillating structures, having \( \alpha_j(t) \) in (2.1) be periodic seems natural and physically relevant. In that case, by defining \( a_j \) the magnitude of \( \alpha_j \), \( \omega_j \) its angular frequency and \( \theta_j \) its phase, one can write

\[
\alpha_j(t) = a_j e^{i(\omega_j t + \theta_j)}. \tag{2.2}
\]

A flow decomposition where the modes exhibit such a periodic behavior is commonly called dynamic mode decomposition, although the name Koopman decomposition has also been used. The assumptions made about the flow in (2.1) and (2.2) are very likely not realistic, especially in the case of non-linear flows, and the meaning of the results obtained when computing a DMD of such flows has to be discussed; see, e.g., Mezić’s (2013) review of some key papers on the subject of DMD.

For simplicity, it is considered in the following that (2.1) and (2.2) hold, and particularly that the flow can be exactly decomposed in a finite number \( N \) of distinct modes oscillating at single frequencies.

Furthermore we assume here the time step \( \Delta t \) between two consecutive snapshots is constant. In that case, rewriting (2.2) at the times when snapshots are taken gives

\[
\alpha_j(t_n) = a_j e^{i\theta_j} e^{i\omega_j t_n} \quad \left[ \gamma_j := a_j e^{i\theta_j} \right] \\
= \gamma_j e^{i\omega_j \Delta t} \\
= \gamma_j (e^{i\omega_j \Delta t})^n \quad \left[ \psi_j := \omega_j \Delta t \right] \\
= \gamma_j (e^{i\psi_j})^n \quad \left[ \lambda_j := e^{i\psi_j} \right]. \tag{2.3}
\]

Rewriting (2.1) with these notations and at discrete times yields

\[
s_{n+1} := s(t_{n+1}) = \sum_{j=1}^{N} \gamma_j \lambda_j^{n+1} \varphi_j, \tag{2.4}
\]

which we write in matrix form.
Since we have assumed that the \( N \) structures \( \varphi_j \) are independent, \( \Phi \) is invertible and \( \Phi^{-1} \) exists. Hence there is a matrix for which

\[
\begin{align*}
s_{n+1} &= \left[ \begin{array}{c|c|c} \varphi_1 & \varphi_2 & \cdots \\ \hline \end{array} \right] \begin{bmatrix} \gamma_1 \lambda_1^{n+1} \\ \gamma_2 \lambda_2^{n+1} \\ \vdots \\ 0 \end{bmatrix} \\
&= \Phi \begin{bmatrix} \vdots & \lambda_j & 0 \\ \lambda_j & 0 & \ldots \\ 0 & \ldots & \ldots \end{bmatrix} \sigma_n \\
&= \Phi \Lambda \sigma_n \\
&= \Phi \Lambda \Phi^{-1} \Phi \sigma_n \\
&= \Phi \Lambda \Phi^{-1} s_n.
\end{align*}
\]

(2.5)

Since we have assumed that the \( N \) structures \( \varphi_j \) are independent, \( \Phi \) is invertible and \( \Phi^{-1} \) exists. Hence there is a matrix

\[
A = \Phi \Lambda \Phi^{-1}
\]

(2.6)

for which

\[
s_{n+1} = As_n.
\]

(2.7)

In other words, \( A \) is the operator that shifts a snapshot of the flow from one time step to the next, and it is commonly referred to as the Koopman operator.

### 2.2.1 Eigen decomposition of the Koopman operator

Since \( \Lambda \) is a diagonal matrix, we have from (2.6) that the \( \lambda_j \) introduced in (2.3) are the eigen-values of \( A \) and that the structures \( \varphi_j \) into which we want to decompose the flow are the eigen-vectors of \( A \). Furthermore, the eigen-values contain the frequency \( f_j \) of these structures within their argument. Indeed, from (2.3) we can write

\[
f_j = \frac{\omega_j}{2\pi} = \frac{\psi_j}{2\pi \Delta t}.
\]

(2.8)

It is worth noting at this point that, by measuring the argument inside \( ] - \pi; \pi ] \), we have

\[
\left| \frac{\psi_j}{2\pi f_j} \right| \leq \frac{\pi}{2\pi f_j}
\]

(2.9)

and thus
\[ |\Delta t| \leq \left| \frac{1}{2f_j} \right|. \quad (2.10) \]

In other words, we draw a parallel with the Nyquist-Shannon theorem (see, e.g., Oppenheim, Willsky, and Nawab [1997, p. 518] for a statement of the theorem), since the sampling frequency of the snapshots \( 1/\Delta t \) has to be at least as high as twice the largest frequency representable in the decomposition.

In order to include modes presenting a growth or decay rate, their time-dependency is expressed by introducing a real constant \( r_j \) as

\[ \alpha_j(t_n) = a_j r_j^n e^{i(\omega f_j t_n + \theta_j)}. \quad (2.11) \]

With \( \lambda_j = r_j e^{i\theta_j} \) in (2.3), the derivations that follow remains valid. As a result, the growth or decay rate of mode \( j \) is given by \( r_j = |\lambda_j| \).

It is now clear that finding the eigen-values and eigen-vectors of \( A \), the Koopman operator of the flow, will be a significant step into finding the pairs \( \{\alpha_j(t), \varphi_j\} \) of the decomposition. Several methods can be employed in order to compute this eigen decomposition, usually starting with the creation of a Krylov subspace of \( A \). Such a subspace can be obtained to the order \( p \) by applying successively the operator \( A \) to an arbitrary vector \( x \), \( p - 1 \) times:

\[ K_p(A, x) = \{x, Ax, A^2x, ..., A^{p-1}x\}. \quad (2.12) \]

In the context of flow decomposition, however, \( A \) is no explicitly known but by choosing \( x = s_0 \), the first flow snapshot, then by (2.7) we have

\[ K_p(A, s_0) = \{s_0, As_0, A^2s_0, ..., A^{p-1}s_0\} \]

\[ = \{s_0, s_1, s_2, ..., s_{p-1}\}, \quad (2.13) \]

which is nothing else than a set of \( p \) consecutive snapshots of the flow. In other words one can obtain a Krylov subspace of \( A \) so long as one can obtain snapshots.

There are two methods commonly used from here—both of them described below: forming the companion matrix of \( A \), or forming a matrix similar to \( A \). After solving the eigen decomposition of either matrix, a simple transformation will then suffice to get back to the eigen decomposition of \( A \) and hence the \( \varphi_j \) and \( \lambda_j \) of the decomposition (2.4).

2.2.1.1 The companion matrix method

We start by forming a snapshot matrix \( S \). This is done by grouping all the successive snapshot vectors as columns of \( S \). The elements of each snapshot must always be in the same order, and we assume as always that the time step between two consecutive columns of the snapshot matrix is constant. In effect, \( S \) is then the Krylov subspace that is needed.
Since we have assumed that the flow can be decomposed into a finite number of structures, the matrix \( S \) will eventually become linearly dependent as the number of snapshots grows. In numerical applications linear dependence occurs gradually, and the introduction of residuals is necessary in order to maintain mathematical equality in the operations that follow (Ruhe 1984). However, for simplicity, it is considered here that the snapshot matrix becomes linearly dependent upon the addition of the snapshot \( s_t \). This means that the snapshot \( s_t \) can be written as a linear combination of the \( l \) previous snapshots:

\[
\mathbf{s}_t = \begin{bmatrix}
s_0 & s_1 & \cdots & s_{t-1} & c_0 & c_1 & \cdots & c_{l-1}
\end{bmatrix}
\]

(2.14)

Defining \( \tilde{S} = [s_0, s_1, \ldots, s_{t-1}] \) the snapshot matrix containing the first \( l \) snapshots, and \( \hat{S} = [s_1, s_2, \ldots, s_t] \), we have

\[
A\tilde{S} = \hat{S}
\]

\[
= [s_1, s_2, \ldots, s_t]
\]

\[
= \tilde{S} \begin{bmatrix}
1 & 0 & c_0 \\
& 1 & c_1 \\
& & \ddots & c_{l-2} \\
& & & 1 & c_{l-1}
\end{bmatrix}
\]

\[
= \tilde{S}C
\]

(2.15)

(2.16)

where \( C \) is a companion matrix. Its last column \( \mathbf{c} \) can be computed if we know \( l + 1 \) snapshots by solving (2.14), e.g. in the least squares sense (using normal equations, a QR- or SVD-decomposition), or using any other relevant norm.

By property of companion matrices, its eigen-values are the same as the eigen-values \( \lambda_j \) of \( A \). Moreover, if \( \mathbf{g}_j \) is the eigenvector of \( C \) associated to \( \lambda_j \) then

\[
A\tilde{S}\mathbf{g}_j = \tilde{S}C\mathbf{g}_j
\]

\[
= \hat{S}\lambda_j\mathbf{g}_j
\]

\[
= \lambda_j\tilde{S}\mathbf{g}_j.
\]

That is, \( \tilde{S}\mathbf{g}_j = \varphi_j \) is the eigen-vector of \( A \) associated to \( \lambda_j \).

2.2.1.2 The similar matrix method

Although the companion matrix method is mathematically correct, it has been noted that the method is ill-conditioned, and that a more robust method should be used (Schmid 2010). In (2.15), the matrix \( \tilde{S} \) is replaced by its thin singular value decomposition \( \tilde{S} = U\Sigma V^t \), where, if \( \tilde{S} \) is \( m \times n \), \( U \)
is \( m \times n \), \( \Sigma \) is diagonal and both \( \Sigma \) and \( V \) are \( n \times n \). Moreover, \( U \) and \( V^t \) are unitary so that \( VV^t = V^tV = I \), where \( I \) is the identity matrix. Finally, with \( \Sigma^+ \) being the pseudo-inverse of \( \Sigma \) we have

\[
AU \Sigma V^t = \hat{S} \\
U^t AU \Sigma V^t = U^t \hat{S} \Sigma V^t \\
U^t AU = U^t \hat{S} \Sigma V^t.
\] (2.18)

Since \( U \) is unitary, the matrix \( U^t AU \) is similar to \( A \), and can be computed using the right hand side of (2.18). Therefore, its eigen-values are the same as the eigen-values \( \lambda_j \) of \( A \) (see Horn and Johnson 2012, sec. 1.3, 2.2). Moreover, if \( h_j \) is the eigen-vector of \( U^t AU \) associated with \( \lambda_j \) then

\[
U^t AU h_j = \lambda_j h_j \\
AU h_j = \lambda_j U h_j
\] (2.19)

That is, \( U h_j = \varphi_j \) is the eigen-vector of \( A \) associated with \( \lambda_j \).

2.2.2 Dynamic modes

We have now found, using either method, a set of \( \lambda_j \) and associated \( \varphi_j \) from the data collected in the snapshot matrix. Comparing with the decomposition as written in (2.4), the vectors \( \varphi_j \) contain the space distribution of the structures into which we want to decompose the flow. Thanks to \( \lambda_j \) we also know the frequency at which each structure oscillates in time, as well as their growth or decay rate. But the weight and relative phase of each structure, contained in \( \gamma_j \), is still unknown. This can be dealt with by solving the linear system

\[
\begin{bmatrix}
\lambda_0 \varphi_0 & \lambda_1 \varphi_1 & \cdots \\
\end{bmatrix}
\begin{bmatrix}
\gamma_0 \\
\gamma_1 \\
\vdots
\end{bmatrix} =
\begin{bmatrix}
s_n
\end{bmatrix}
\] (2.20)

for any particular snapshot, e.g. with a least squares approach. Modes \( m_j := \gamma_j \varphi_j \) are finally obtained by scaling the vectors \( \varphi_j \) by the associated complex \( \gamma_j \). These modes satisfy the relation

\[
s(t_n) = \sum_{j=1}^{N} \lambda_j m_j.
\] (2.21)

2.3 DMD computation in dymode

In dymode, the dynamic mode decomposition is performed using the similar matrix method described in 2.2.1.2 above. The algorithm used is summarized here (see sec. 3 Implementation for more details about each operation):

1. The snapshot matrix \( S \) is read as input.
2. A singular value decomposition is performed on \( \hat{S} \), giving matrices \( U, \Sigma, V \).
3. The similar matrix \( B = U^t AU \) is computed as \( B \leftarrow U^t \hat{S} V \Sigma^+ \).
4. The eigen decomposition of $B$ is computed, giving the eigenvalues $\lambda_j$ and eigenvectors $h_j$ of $B$.
5. The linear system $H\gamma = U^T s_0$ is solved for $\gamma$.
6. The matrix of modes is computed as $M \leftarrow UH\text{diag}(\gamma)$, where $\text{diag}(\gamma)$ is the square diagonal matrix with the elements of vector $\gamma$ on the main diagonal.
7. The energy of the modes at the first snapshot is computed.
8. The eigen-values $\lambda_j$, the energy of the modes as well as the singular values from the SVD are saved.
9. The modes are sorted according to energy criteria and saved.
3 Implementation

Dymode makes calls to the ScaLAPACK (Blackford et al. 1997) library, MPI reduction operations, and functions from the Eigen (Guennebaud, Jacob, and others 2010) library that may be vectorized. For these reasons, dymode does not guarantee bit-wise reproducible results and variations in the output may occur, even from run to run. These variations, however, usually occur in a few of the least significant digits.

Furthermore, ScaLAPACK is generally cited as the library used for parallel linear algebra operations. While dymode is assumed to compile and run with any of the ScaLAPACK implementations, dymode was developed and debugged using mostly Intel® Math Kernel Library 11.0 Update 5 for Windows.

3.1 Computational options

3.1.1 The process grid

Dymode uses ScaLAPACK for almost all the linear algebra operations necessary. Therefore when dymode starts, the available processes are arranged in a grid suitable for calls to the ScaLAPACK routines. Dymode will always try to create grids that are as square as possible while keeping a high number of utilized processes. Figure 3—1 c., for instance, shows the 3-by-2 process grid created when calling dymode with 6 processes: the MPI rank is numbered and associated with a color on the figure.

If the number of processes is such that a rectangle grid cannot be created, processes may be left out of the grid. These processes will not participate in any computation; every MPI process, however, participates in reading data from the input files.

3.1.2 Matrix distribution

Matrices are divided into blocks and distributed amongst the processes of the grid in a round robin cycle. Figure 3—1 a. represents a 26-by-16 matrix divided into 6-by-6 blocks colored according to the process rank to which they are assigned. Figure 3—1 b. shows how these blocks are combined inside the memory of each process. Tints of a same color identify different blocks from the original matrix owned by the same process.

Because of some of the ScaLAPACK routines called, the blocks have to be square and at least 6-by-6. The size used can be set by using the \texttt{--block} \texttt{<size>} argument.
3.1.3 Residuals
A residual for the key operations of dymode will be computed and displayed if the program is called with the \texttt{--r} flag. See the appropriate paragraph for a definition of the residual of each operation.

3.2 Input / Output
3.2.1 Input
3.2.1.1 File naming
The only input data necessary to compute the dynamic mode decomposition is the snapshot matrix. See sec. 2 Theory for a description of the snapshot matrix. The file format used to store the snapshot matrix must be HDF5 (The HDF Group 1997).

The snapshot matrix has to be stored in one or several files named according to the pattern
\texttt{<name><filenumber>.h5},
where
\begin{itemize}
  \item \texttt{<name>} can be any valid filename identifying the data, and
  \item \texttt{<filenumber>} is a 4 digit integer indicating the order in which the files have to be read to reconstruct the snapshot matrix, left padded with 0 if necessary. \texttt{<filenumber>} must start at 0001.
\end{itemize}

For example, the snapshot matrix of a flow case named “landing\_gear” could be stored in the set of files \texttt{landing\_gear0001.h5}, \texttt{landing\_gear0002.h5}, \texttt{landing\_gear0003.h5}, etc. The number of such files that are read by dymode can be set with \texttt{--nfiles <n>}, in which case dymode will construct a snapshot matrix from only the first \texttt{<n>} files. If this argument is omitted, dymode will attempt to read only one file.

3.2.1.2 HDF5 structure
\begin{note}
See also sec. 7 Dymodem which contains Matlab scripts to save matrices as valid HDF5 files.
\end{note}

The snapshots must be present within the same dataset in every file. Dymode will look for the dataset \texttt{/snapshots\_T} by default, but the name of the dataset can be set by using
the \texttt{--dataset \textless name\textgreater} option. The data type in this dataset must be 64-bit floating point (double precision).

Furthermore, the blocks of the snapshot matrix have to be in transposed form in the file. For instance, for a snapshot matrix containing $k + 1$ snapshots of the two variables $u$ and $v$ looking like this:

\[
\begin{bmatrix}
| & | \\
\underline{u}_j & \underline{u}_{j+k} \\
| & | \\
\underline{v}_j & \underline{v}_{j+k} \\
| & |
\end{bmatrix},
\]

the dataset inside a particular file will contain a matrix like this:

\[
\begin{bmatrix}
-\underline{u}_j & -\underline{v}_j \\
-\underline{u}_{j+k} & -\underline{v}_{j+k}
\end{bmatrix}.
\]

The number of snapshots $k + 1$ stored in the dataset can vary from file to file.

\textbf{3.2.1.3 Variable selection}

It is possible to specify a name for the quantity saved in the snapshots with \texttt{--variable \textless name\textgreater}. If the snapshots contain several variables, use commas to separate the variables names like so: \texttt{--variable \textless name1\textgreater,\textless name2\textgreater,\textless name3\textgreater,...} This name will be used to identify the output files containing the modes. If no name is provided, the entire HDF5 dataset will be read as a single variable and the default name $u$ will be used to save the modes.

If the snapshots contain variables that should not be included in the mode computation, use the name \texttt{null} as a placeholder. In that case the block of the snapshot matrix corresponding to this variable will not be read from disk. For example if a snapshot contains the variables $u$, $v$, $w$ and $p$:

\[
\begin{bmatrix}
u \\
v \\
w \\
p
\end{bmatrix}
\]

and dymode is called with \texttt{--variable u,null,null,p} then $v$ and $w$ will be ignored and the following snapshot will be formed in memory:

\[
\begin{bmatrix}
u \\
p
\end{bmatrix}.
\]

\textbf{Note: All the variables must have the same number of rows, i.e. they must have the same number of sampled points.}

\textbf{3.2.1.4 Stride}

It is possible to read only one out of $k$ snapshots from the files on disk, e.g. if the DMD should be computed with a snapshot matrix where the time-step between the columns is larger than the time-step...
step between the available snapshots. The integer $k$ can be specified with the option `--stride <k>`. For instance, if files on disk contain the snapshots

$$\begin{bmatrix}
  s_0 & s_1 & s_2 & \cdots \\
  \vdots & \vdots & \vdots & \\
\end{bmatrix},$$

then the following matrix will be formed in memory:

$$\begin{bmatrix}
  s_0 & s_k & s_{2k} & \cdots \\
  \vdots & \vdots & \vdots & \\
\end{bmatrix}.$$  

### 3.2.1.5 Parallel reading

All the processes participate in reading the data, regardless of whether or not they belong to the process grid.

Regardless of how many data files are present on disk, dymode will only read the number of files that is specified with `-nfiles <n>` starting from file 0001. The `<n>` files are distributed as equally as possible amongst all the processes, with process of rank 0 being assigned the first $\text{floor}(nfiles/\text{numproc})$, process of rank 1 the next $\text{floor}(nfiles/\text{numproc})$, etc. If the number of files to read is not divisible by the number of processes, the processes of lowest rank will be assigned 1 extra file. Table 1 shows how five files would be distributed amongst three MPI processes.

**Table 1: Example of file distribution among processes.**

<table>
<thead>
<tr>
<th>Process 0</th>
<th>Process 1</th>
<th>Process 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>file0001.h5, file0002.h5</td>
<td>file0003.h5, file0004.h5</td>
<td>file0005.h5</td>
</tr>
</tbody>
</table>

All the files are accessed a first time in order to determine how many snapshots are contained in each file, and to check that all the snapshots have the same number of rows.

The files are accessed a second time in order to read the data. Only the blocks of data that are assigned to variables with a name different from "null" are read from disk. When all the necessary data has been read, the data is redistributed onto the processes belonging to the grid to form the block cyclic distributed snapshots matrix (see Figure 3—1).

### 3.2.1.6 Ensight .geo file

If an Ensight Gold mesh is associated with the data, it is possible to specify a geometry file to be parsed with `--geo <file>`. See **3.2.2.3** for more information about how the output is modified.

### 3.2.2 Output

#### 3.2.2.1 Console output

The messages and values printed in the console are almost always coming from the process of MPI rank 0. If a message displayed in the console comes from a process of different rank, the message line is preceded by the MPI rank number between parentheses.

Notable exceptions are the value of the residuals, if computed, and the singular values, if displayed, which are obtained from the global matrices.
3.2.2.2  **Output directory**
During a run, dymode writes a number of files to disk. The path to the directory where these files are saved is specified with `--outdir <path>`.

3.2.2.3  **Modes**
Once the modes have been computed, they are stored on disk in the binary version of the Ensight Gold format.

**Note:** Due to the Ensight Gold format, the modes are stored in single precision.

The naming scheme for the data files is `mode<modenumber>.<variable>.<part>`, where

- `<modenumber>` is the 6-digit number of the mode. See sec. 3.7 *Sorting* for more details about the numbering of the modes,
- `<variable>` is the name of the quantity (given with `--variable <name>`),
- `<part>` is either `abs` or `ang`. Files whose extension is `abs` and `ang` contain respectively the modulus and argument of the complex values that constitute the modes.

The data inside an Ensight Gold variable file is grouped by part number and element type. The default output from dymode creates a single part with a single element type (hexa8), then prints all the values for a variable, in the same order as the snapshot matrix.

If a geometry file was specified with `--geo <geofile>`, matching part and element type headers will be inserted inside the sequence of values.

3.2.2.4  **Text data**
Three text files are saved to disk in addition to the mode files.

- `singulars.txt` contains the singulars values computed during the singular value decomposition, in decreasing order.
- `eigenvalues.txt` contains one column whose entries are the eigen-values printed in the same order as they were returned after the eigen decomposition. The entries are formatted as: `(real part), (imaginary part)`.
- `spectrum.txt` contains two columns. The first column contains the argument of the eigen-values, taken in the same order as in `eigenvalues.txt`, and the second column contains the energy of the associated mode.

In all text files, values are printed with enough digits to recover the original number with double precision.

3.2.2.5  **Execution timing**
If `USE_PROFILER` was defined when compiling dymode, time information about different parts of the code will be output during the program’s execution. Timings displayed in the console have all been measured by the process of rank 0. However, when the program completes successfully each
process writes the file profiler-<rank>.yml in the output directory. This file contains time information for the process of rank <rank> formatted in YAML\(^1\).

### 3.3 Singular value decomposition
The first step is to compute the thin singular value decomposition of the snapshot matrix lacking the very last snapshot. This is done with a single call to the ScaLAPACK routine \(p^\ast gesvd\).

This creates additional matrices \(U, \Sigma, V\) such that

\[ \hat{S} = U \Sigma V^t. \]

#### 3.3.1.1 Residual of the SVD calculation
The residual is defined as \(\max_{i,j} |\delta_{i,j}|\) where \(\delta_{i,j}\) are the coefficients of the matrix

\[ \Delta = U\Sigma V^t - \hat{S}. \]

### 3.4 Similar matrix formation
Using the results from the SVD, a matrix \(B\), similar to the Koopman matrix \(A\), is formed according to

\[ B = U^t \hat{S} V \Sigma^+. \]

where \(\Sigma^+\) is the pseudo inverse of \(\Sigma\). Since \(\Sigma\) is diagonal, \(\Sigma^+\) can be formed by taking the inverse of the non-zero entries of the transpose of \(\Sigma\). Values below a numerical tolerance are set to 0. The value of the tolerance used is displayed in the console.

### 3.5 Eigen decomposition
The eigen-values and eigen-vectors of \(B\) are then computed. This requires 3 steps:

1. \(B\) is reduced to upper-Hessenberg form \(B = QGQ^t\)
2. The Hessenberg form is reduced to Schur form \(G = ZTZ^t\)
3. The matrix of eigen-vectors \(H\) is constructed from the Schur matrix’s entries

The first two operations can be done in parallel using calls to the routines \(pdgehrd\) and \(pdhseqr\) from ScaLAPACK. The last operation is always done in serial.

Alternatively, the whole eigen decomposition can be left entirely to the Eigen library by setting \(--eigen\ EigSerial\). In that case no call to ScaLAPACK is used and the computation is done entirely in serial.

---

**Note:** The ScaLAPACK routine \(p^\ast hseqr\) which computes the Schur reduction appears to be broken, and the problem has been reported to Intel\(^2\), whose implementation of ScaLAPACK dymode was mostly developed with.

It is unknown to the authors if it is a bug common to all ScaLAPACK implementations or only to some of them, therefore by default dymode computes the Schur reduction in serial using the Eigen library, which corresponds to setting \(--eigen\ EigHess\). Parallel computation of the Schur form using ScaLAPACK can be forced by setting \(--eigen\ EigSchur\).

\(^1\) See [http://www.yaml.org](http://www.yaml.org) for the format specification.

Table 2 provides a summary of which library will be used for which operation depending on the --eigen argument specified.

Table 2: Summary of the libraries used during the eigen decomposition for all the --eigen arguments.

<table>
<thead>
<tr>
<th>Operation</th>
<th>EigSerial</th>
<th>EigHess</th>
<th>EigSchur</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hessenberg reduction</td>
<td>Eigen</td>
<td>ScaLAPACK</td>
<td>ScaLAPACK</td>
</tr>
<tr>
<td>Schur reduction</td>
<td>Eigen</td>
<td>Eigen</td>
<td>ScaLAPACK</td>
</tr>
<tr>
<td>Forming eigen-vectors</td>
<td>Eigen</td>
<td>Eigen</td>
<td>Eigen</td>
</tr>
</tbody>
</table>

3.5.1.1 Residual of the Hessenberg reduction
The residual is defined by \( \max_{i,j} |\delta_{i,j}| \) where \( \delta_{i,j} \) are the coefficients of the matrix

\[
\Delta = Q^T BQ - G.
\]

3.5.1.2 Residual of the eigen decomposition
The residual is defined by \( \max_{i,j} |\delta_{i,j}| \) where \( \delta_{i,j} \) are the coefficients of the matrix

\[
\Delta = BH - H\Lambda,
\]

where \( \Lambda \) is the diagonal matrix formed with the eigen-values of \( B \).

3.6 Linear system solving
The shape of the modes is obtained by left multiplying the matrix of eigen-vectors \( H \) with the unitary matrix \( U \), so that the norm of each mode is equal to the norm of the corresponding eigen-vectors. However, the norm of the eigen-vectors is entirely dependent on the implementation of the eigen-solver which is free to return any multiple of a given eigen-vector. In practice, the eigen-vectors are usually normalized before being returned. As a result, although the shape of the modes would be correct, their energy content would be equal to one for all modes, and a summation of all the modes would not yield the original snapshot matrix. Therefore it is necessary to weight the modes based on their contribution to the flow.

The vector of weights \( \gamma \) can be computed by solving

\[
H\gamma = U^t s_0.
\]

Note that \( \gamma_j \) is complex, as it also includes the phase \( \theta_j \) of mode \( j \). When two modes form a pair of complex conjugate vectors \( \phi \) and \( \phi^* \), their respective phase should be opposite so that their sum adds up to a real snapshot. However, depending on the method used for solving, the weights of a pair of complex conjugate modes may not be a pair of complex conjugate weights. Dymode enforces that such a pair of weights are actually conjugates. This is done by solving a different system

\[
\tilde{H}\tilde{\gamma} = U^t s_0,
\]

where \( \tilde{H} \) is constructed based on an equivalent real formulation of the system (see e.g. Day and Heroux 2001), and using the fact that complex columns come in conjugate pairs. To construct \( \tilde{H} \) from \( H \), the first column of a pair is replaced with twice its real part, and the second column with minus two times the imaginary part. This is illustrated below, where the column \( i \) and \( i + 1 \) of \( H \) hold a pair of conjugate vectors and column \( j \) holds a real vector:
\[ H = \begin{bmatrix}
i & i + 1 & j \\
\vdots & \vdots & \vdots \\
h_i & h_i^* & h_j \\
\vdots & \vdots & \vdots
\end{bmatrix}\]

Then \( \tilde{H} \) is defined as:

\[ \tilde{H} = \begin{bmatrix}
i & i + 1 & j \\
\vdots & \vdots & \vdots \\
2\Re(h_i) & -2\Im(h_i) & h_j \\
\vdots & \vdots & \vdots
\end{bmatrix}\]

Where \( \Re \) and \( \Im \) denote respectively the real and imaginary parts.

### 3.6.1.1 Residual from the weight computation

This residual is defined by \( \max_{i,j} |\delta_{i,j}| \) where \( \delta_{i,j} \) are the coefficients of the matrix

\[ \Delta = \tilde{H}\tilde{\gamma} - U^t s_0 \]

### 3.6.2 Reconstructing the solution

We now construct \( \gamma \) by using

\[
\begin{aligned}
\gamma_i &= \tilde{\gamma_i} + i \tilde{\gamma}_{i+1} \\
\gamma_{i+1} &= \tilde{\gamma_i} - i \tilde{\gamma}_{i+1} \\
\gamma_j &= \tilde{\gamma_j}
\end{aligned}
\]

which guarantees that the weights corresponding to a pair of modes are conjugates. Finally, the weighted modes are formed from the eigen-vectors by

\[ M = UH\text{diag}(\gamma). \]

### 3.6.2.1 Residual from Modes

This residual is defined by \( \max_{i,j} |\delta_{i,j}| \) where \( \delta_{i,j} \) are the coefficients of the matrix

\[ \Delta = MV_{\lambda} - \tilde{S} \]

Where \( V_{\lambda} \) is the Vandermonde matrix based on the eigenvalues of \( B \).

### 3.6.2.2 Residual from Modes (last snapshot)

This residual is defined by \( \max_{i} |\delta_{i}| \) where \( \delta_{i} \) are the coefficients of the vector

\[ \delta = M \varpi - \varsigma, \]

Where \( \varpi \) is the last column of \( V_{\lambda} \) and \( \varsigma \) the last snapshot available.

### 3.7 Sorting

The columns \( m_j \) of \( M \), containing the weighted modes, are not sorted in any particular order and mostly depend on the order in which the eigen-vectors are found. First, the \( \ell^2 \) norm of each column is computed in a stable manner that protects against over- and under-flow, using

\[ \|m_j\| = \beta \left\| \frac{m_j}{\beta} \right\| \]

20
where $|\beta| = \max_i |m_{i,j}|$. Therefore, $\|m_j\|^2$ is the energy contribution of the mode in column $j$ at the time of the first snapshot. The values of $\|m_j\|$ are saved in the second column of the file spectrum.txt in the same order as they appear in the $M$ matrix.

If the first $\langle k \rangle$ modes are to be saved, as specified by using `--modes <k>` with $\langle k \rangle > 0$, these modes are saved in a sorted manner. The way to sort the modes can be specified by using `--sort <type>,<option1>,<option2>,...`

The only $<type>$ of sorting currently supported by dymode is energy. Because modes can have a growth or decay rate, given by the modulus of their eigen-values, their energy changes in time. It is possible to specify the snapshot number $s$ at which time the energy is evaluated and used to sort the modes by using `--sort energy,s; s = 0` being the number of the first snapshot. In that case the value used to sort the modes is

$$\|m_j\||\lambda_j|^s.$$

There are also special keywords that can be used instead of using a number: median and mean. With `--sort energy,median`, the modes will be sorted according to their energy midway through the snapshot sequence, using the value

$$\|m_j\||\lambda_j|^{n/2}$$

where $n$ is the total number of snapshots. With `--sort energy,mean`, the modes will be sorted according to their mean energy over time of the snapshot sequence, using the value

$$\begin{cases} \|m_j\| & \text{if } |\lambda_j| = 1 \\ \|m_j\| \frac{1 - |\lambda_j|^n}{n (1 - |\lambda_j|)} & \text{if } |\lambda_j| \neq 1 \end{cases}$$

where $n$ is the total number of snapshots.

By default, only the modes with eigen-values in the top half complex plane, corresponding to positive frequencies, are saved, so that only real modes and one out of a pair of conjugate modes are saved. The modes with eigen-values in the bottom half complex plane can be included to the modes saved by adding the $c$ option.

By default, modes are sorted in descending order, so that the mode with largest value of energy is saved as mode 0. Modes can be sorted in increasing order instead by using the $i$ option. For example, in order to save the first 5 modes with the smallest energy at the time of the third snapshot, saving a single mode per pair of conjugate modes, one would use `--modes 5 --sort energy,2,i`. In that case, the mode number 0 will be the mode with the lowest energy at the time of the third snapshot $s_2$. Figure 3—2 illustrates which modes will be saved and how they will be numbered for some of the possible sorting methods.

If the `--sort` option is not specified, dymode uses `--sort energy,median` by default.
Figure 3—2 Example of mode numbering of the first 5 modes for different --sort parameters. The points corresponding to saved modes are marked with a red star.
4 Building dymode

4.1 Dependencies

Dymode relies upon a number of libraries. These libraries must be acquired and built beforehand and linked when compiling dymode. Here is a summary of all external dependencies:

<table>
<thead>
<tr>
<th>Library</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI</td>
<td>Dymode is intended for very large dataset and has no serial version. Therefore an MPI distribution is necessary to compile dymode. For mode decomposition of smaller data sets, see sec. 7 Dymodem.</td>
</tr>
<tr>
<td>ScaLAPACK</td>
<td>Dymode has been tested mostly with MKL. However, other versions of ScaLAPACK should be usable.</td>
</tr>
<tr>
<td>Boost</td>
<td>The Boost library is used to perform a variety of tasks.</td>
</tr>
<tr>
<td>HDF5</td>
<td>This library is used to read the snapshots matrix from hdf5 files. It is not required to build it with parallel support.</td>
</tr>
</tbody>
</table>

In addition, dymode comes with different libraries. These libraries are header only and do not need to be built. The default makefile should already specify the right path to the proper headers and no user intervention is required. If building in Visual Studio, add the include folders to the project’s include directories. Here is a summary of these libraries:

<table>
<thead>
<tr>
<th>Library</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eigen</td>
<td>The Eigen library is used to store local matrices and perform some operations. Most notably it is used to solve all or part of the eigen decomposition. Dymode uses a modified version of Eigen.</td>
</tr>
<tr>
<td>TCLAP</td>
<td>Parsing of the input arguments is done with the Templatized C++ Command Line Parser Library3.</td>
</tr>
<tr>
<td>peigen</td>
<td>The wrappers in peigen are used to hold distributed matrices and provide a number of operations, including calls to ScaLAPACK.</td>
</tr>
<tr>
<td>tic-toc-profiler</td>
<td>This library is used to time the execution of the code.</td>
</tr>
<tr>
<td>yaml-hpp</td>
<td>Needed to save profiling information from the tic-toc-profiler. This is a modified version of yaml-cpp4.</td>
</tr>
</tbody>
</table>

4.1.1 Boost

Most of the Boost libraries are header only, so that you only need to download and extract the files, and define the path to Boost as an include folder.

It is possible that one library used in tic-toc-profiler needs to be built, namely boost/system. If that is the case, it can be done rather easily by doing the following:

cd /path-to-libraries/boost_1_55_0
./bootstrap.sh
./b2 --with-system --build-type=complete toolset=<toolset>

3 http://tclap.sourceforge.net
4 https://code.google.com/p/yaml-cpp/
Where `<toolset>` is typically `intel, gcc, msvc`, etc depending on the compiler used. See Boost instructions for more details.

4.2 Dymode

4.2.1 Getting dymode

The latest version of dymode can be requested by writing to fromain@kth.se or gef@kth.se.

4.2.2 Setting paths and options

*Makefile* needs to be edited to match a particular system. Some systems come with compiler wrappers that automatically `#define` include and library paths. The following sections explains how to modify *Makefile* assuming that no wrapper is used and all the paths need to be explicitly defined.

There are three sections in *Makefile* that need to be edited before dymode can be compiled.

4.2.2.1 Section 1. Set the compiler

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CC</td>
<td>Compiler to use, eg. <code>gcc</code>, <code>icpc</code>, <code>mpicc</code>, etc.</td>
</tr>
<tr>
<td>CC_OPTS</td>
<td>Options to pass to the compiler. These will be used in all targets.</td>
</tr>
</tbody>
</table>

4.2.2.2 Section 2. Set external dependency paths

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI</td>
<td>Parent folder of the <code>include</code> and <code>lib</code> directories of the MPI installation.</td>
</tr>
<tr>
<td>HDF5</td>
<td>Parent folder of the <code>include</code> and <code>lib</code> directories of the HDF5 library.</td>
</tr>
<tr>
<td>BOOST</td>
<td>Root folder of the Boost library, e.g. <code>/lib/boost_1_55_0</code>.</td>
</tr>
</tbody>
</table>

4.2.2.3 Section 3. Set up the ScALAPACK paths

There are two sets of macro that help linking to a ScALAPACK library. The first set (Case 1) can be used if BLAS/LAPACK/ScALAPACK was built from source.

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BLAS</td>
<td>Root folder of a BLAS build.</td>
</tr>
<tr>
<td>LAPACK</td>
<td>Root folder of a LAPACK build.</td>
</tr>
<tr>
<td>SCALAPACK</td>
<td>Root folder of a ScALAPACK build.</td>
</tr>
<tr>
<td>PACKLIBS</td>
<td>Link line relative to the above libraries, typically <code>-lscalapack -llapack -lblas</code>.</td>
</tr>
</tbody>
</table>

The second set (Case 2) can be used to link with Intel MKL. In that case, it is advised to use the MKL Link Line Advisor to determine what arguments should be used.

<table>
<thead>
<tr>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MKLROOT</td>
<td>Parent folder of the <code>include</code> and <code>lib</code> folders of the MKL installation.</td>
</tr>
<tr>
<td>USE_THIS_LINK_LINE</td>
<td>Name of the libraries to link with. Given under “Use this link line:” in MKL Link Line Advisor.</td>
</tr>
<tr>
<td>COMPILER_OPTIONS</td>
<td>Include folder and special options. Given under “Compiler options:” in MKL Link Line Advisor.</td>
</tr>
</tbody>
</table>
Note: If MKLROOT is not empty, dymode will always try to compile using MKL and ignore paths defined in the Case 1 section.

4.2.3 Compiling

There are three targets that can be built with `make debug`, `make release` or `make assert`. The `debug` target does not use optimizations and does not define `NDEBUG`. The `release` target uses optimizations and defines `NDEBUG`. The `assert` target is the same as `release` and is compiled with optimizations; however the macro `NDEBUG` is not defined.

The Eigen library used to manage matrices verifies a number of assertions at runtime, e.g. to verify that elements being accessed lie within the bounds of the matrix. While these checks are useful to identify problems (such as segmentation faults) that may have remained hidden and lead to corrupted results, they are costly and are deactivated in the release target. Such verifications are deactivated when the macro `NDEBUG` is defined. The `assert` target allows to compile dymode with optimizations but does not define `NDEBUG`, therefore maintaining these checks. The differences between the build targets are summarized here.

<table>
<thead>
<tr>
<th>Target</th>
<th>Optimizations</th>
<th>Defines NDEBUG</th>
</tr>
</thead>
<tbody>
<tr>
<td>debug</td>
<td>Level 0</td>
<td>No, assertions are checked</td>
</tr>
<tr>
<td>assert</td>
<td>Level 3</td>
<td>No, assertions are checked</td>
</tr>
<tr>
<td>release</td>
<td>Level 3</td>
<td>Yes, no verification</td>
</tr>
</tbody>
</table>

After `make <target>`, the executable `dymode.out` is created in the root `dymode` folder. This file can be deleted by running `make clean`. 
5 Usage

A typical way to call dymode looks like:

```bash
mpiexec -n 1 dymode -f "./flowcase" -n 1 -o "./dymode_results/" -m 5
```

Executing this line will run dymode on one MPI process. Dymode will then use the single file `flowcase0001.h5` located in the same folder as dymode as input for the snapshot matrix. The output of dymode, which will include 5 modes, will be saved in the subfolder `dymode_results`. A list of all the options is provided here.

<table>
<thead>
<tr>
<th>Option name</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>-f filename &lt;file&gt;</code></td>
<td><code>&lt;file&gt;</code></td>
<td>Specify the path and name of the snapshot files, without the file number or extension.</td>
</tr>
<tr>
<td><code>-g geo &lt;file&gt;</code></td>
<td></td>
<td>Specify the path and name of an Ensight Gold geometry file associated with the data.</td>
</tr>
<tr>
<td><code>-d dataset &lt;set&gt;</code></td>
<td><code>/snapshots_T</code></td>
<td>Specify the name of the HDF5 dataset containing the snapshots.</td>
</tr>
<tr>
<td><code>-n nfiles &lt;n&gt;</code></td>
<td>1</td>
<td>Specify the number of files to read.</td>
</tr>
<tr>
<td><code>-s stride &lt;n&gt;</code></td>
<td>1</td>
<td>Specify the stride to use between the available snapshots when forming the snapshot matrix.</td>
</tr>
<tr>
<td><code>-i variables &lt;var&gt;</code></td>
<td>&quot;u&quot;</td>
<td>Specify the names of the variables to use as input. Use the name &quot;null&quot; to skip a variable.</td>
</tr>
<tr>
<td><code>-t sort &lt;type&gt;</code></td>
<td>&quot;energy,median&quot;</td>
<td>Specify the type of sorting applied to the modes before numbering them.</td>
</tr>
<tr>
<td><strong>Computation</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>-b block &lt;n&gt;</code></td>
<td>6</td>
<td>Specify the block size used to distribute matrices amongst the MPI processes.</td>
</tr>
<tr>
<td><code>-e eigen &lt;type&gt;</code></td>
<td>&quot;EigHess&quot;</td>
<td>Specify the level of parallelization used when computing the eigen decomposition.</td>
</tr>
<tr>
<td><code>-r residuals</code></td>
<td></td>
<td>When present, this flag activates the computation of residuals.</td>
</tr>
<tr>
<td><code>-m singulars &lt;n&gt;</code></td>
<td>0</td>
<td>Specify the number of singular values to display.</td>
</tr>
<tr>
<td><strong>Output</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>-o outdir &lt;path&gt;</code></td>
<td></td>
<td>Specify the path where output files are saved.</td>
</tr>
<tr>
<td><code>-m modes &lt;n&gt;</code></td>
<td>1</td>
<td>Specify how many modes are saved to disk.</td>
</tr>
<tr>
<td><code>-h help</code></td>
<td></td>
<td>When present, this flag terminates the execution of the program and displays help.</td>
</tr>
</tbody>
</table>
6 Parallel scaling

The time spent by dymode to compute the DMD of a particular flow was benchmarked for different numbers of parallel processes, and the results are shown in Figure 6—1. The figure shows the time spent in four of the most time consuming tasks.

The snapshot matrix was saved into 100 files and its total size was 655360-by-1184; the block size used to distribute the matrix was 6. Residuals were not calculated and no mode was saved. Computations were benchmarked on a Cray XC30, having two 2.7 GHz, 12-core E5-2697 v2 (Ivy Bridge) processors per node. Dymode was compiled in release mode with GCC 4.9.1 and linked to Cray-libsci 13.0.1 for ScaLAPACK routines.

Several remarks can be made from the figure:

- The total execution time decreases from 1229 s using $N_p = 4$ cores to 54 s using $N_p = 576$ cores; for this problem, the execution time thus scales as $1/N_p^{0.63}$ over the range of processes benchmarked.
- The time taken to read the snapshot matrix decreases until $N_p = 100$, and then stays constant, since the number of input files was 100.
- The time taken by the eigen decomposition does not benefit much from a higher number of cores used since the Schur decomposition is computed in serial by default because of current issues within ScaLAPACK.
- The number of processes used (reported in Figure 6—1) were multiples of a compute node, so that in some cases processes were left out of the computation to form a rectangular grid.

Figure 6—1: Execution time of the four main tasks vs. number of processes used. The time spent in the four tasks shown here represents nearly the entire runtime of the program.
7 Dymodem

A Matlab version of dymode can be found in ./dymode/dymodem. The function dymodem accepts exactly the same input and the same parameters, if relevant, as dymode, and can be used for problems with smaller sizes that do not require parallel computing. The behavior and output of dymodem are the same as those of dymode.

Since dymodem is a Matlab function, calls have a different syntax. The input filename and output directory are mandatory and are the first two arguments passed to the function. Additional options can be specified by appending pairs ‘option’, ‘value’ to the argument list, where option is the full length name of a dymode option. For instance, the typical call described in sec. 5 Usage translated to a dymodem call is

\[ \text{dymodem('./flowcase', './dymode_results/', 'nfiles',1, 'modes',5);} \]

Dymodem can be useful if the size of the snapshot matrix is small, or if a serial version of the code is required. It can also be useful to run both dymode and dymodem on a subset of the data (e.g. by passing a small value to --nfiles and/or a large value to --stride) and compare the output of both in order to check that building dymode went as expected and its output is correct. It is important to note that the modes are saved from dymode with single precision due to the Ensight Gold format used. Moreover, because of the parallel implementation of dymode, variations in the least significant digits are to be expected. An appropriate tolerance should therefore be used when comparing the output of the two programs.

Some scripts can be found in ./dymodem/dymodem to use Matlab to read from and generate HDF5 files compatible with dymode.
8 References


http://www.journals.cambridge.org/abstract_S0022112010001217.

Decomposition.” *Theoretical and Computational Fluid Dynamics* 25 (1-4) (August 20): 249–

http://www.hdfgroup.org/HDF5/.

http://www.journals.cambridge.org/abstract_S0022112013004266.