
Università degli Studi di Messina

Dipartimento di Ingegneria

Corso di Laurea Triennale in Ingegneria Elettronica e Informatica

# Metodi data-driven e tecniche multi-resolution per l'identificazione di processi industriali

Tesi di Laurea di:
Trovatello Alessandro

Relatore:
Chiar.mo Prof. Luca Patanè

ANNO ACCADEMICO 2022–2023

# Indice

# Introduzione

Negli ultimi anni, la crescente disponibilità di dati in diversi settori ha aperto nuove possibilità per l'analisi dei dati. L'approccio data-driven, ovvero basato sui dati, ha assunto un ruolo fondamentale nel fornire soluzioni intelligenti per la manipolazione dei dati. Questa tesi di laurea si concentra sull'applicazione delle metodologie data-driven nell'ambito industriale, esplorando come i dati possono essere elaborati e utilizzati per ottenere risultati utili a migliorare i processi industriali coinvolti. I dati vengono acquisiti da varie fonti, in questo particolare caso da sensori ed analisi di laboratorio. Questi dati, se analizzati in modo appropriato, possono fornire informazioni preziose per l'ottimizzazione dei sistemi, l'identificazione di anomalie, la previsione delle prestazioni e molto altro ancora. L'utilizzo delle metodologie data-driven consente di sfruttare al meglio questa vasta quantità di dati disponibili. Durante lo sviluppo di questa tesi, saranno considerati sia gli aspetti teorici che pratici dell'approccio data-driven. Saranno esaminati algoritmi come il Dynamic Mode Decomposition (DMD), il Dynamic Mode Decomposition with Control (DMDc) e il Multi-Resolution Dynamic Mode Decomposition (mrDMD). L'obiettivo di questa tesi è dunque quello di investigare gli aspetti peculiari di questi algoritmi al fine di proporre un implementazione integrata in un nuovo algoritmo denominato Multi-Resolution Dynamic Mode Decomposition with Control (mrDMDc) che permette di includere azioni di controllo all'interno di un analisi di tipo multi-resolution. Per far cio' siamo partiti dallo studio degli algoritmi più semplici come il DMD, che si basa sulla costruzione di matrici con i valori raccolti dalle varie simulazioni per poi, tramite la Singular Value Decomposition (SVD), selezionare i modi dinamici più rilevanti utili a ricostruire la dinamica del sistema originale. Il risultato ottenuto dal DMD è stato migliorato introducendo il DMDc che utilizza il DMD integrando il concetto di controllo, questo può essere utile per progettare e ottimizzare un determinato sistema, nonché per comprendere meglio

l'interazione tra le variabili di stato e le azioni di controllo nel sistema dinamico. Successivamente abbiamo analizzato algoritmi più complessi come il mrDMD, mentre il DMD tradizionale considera l'intero intervallo temporale o lo spazio completo per l'analisi delle dinamiche, il mrDMD suddivide l'intervallo temporale o lo spazio in segmenti più piccoli e applica il DMD a ciascun segmento per ottenere una decomposizione a risoluzioni multiple. Uno dei vantaggi principali è quello di poter determinare l'andamento dei dati nel lungo, medio e breve periodo grazie all'analisi dei dati in più scale, inoltre permette al mrDMD di essere molto versatile e di poterlo utilizzare in diversi ambiti scientifici. Il tutto è stato implementato in Python 3.10 sfruttando l'editor Jupyter Notebook.

Nel Capitolo 1 andremo a vedere la teoria dietro ai vari algoritmi, come sono strutturati e come manipolano i dati per ottenere dei risultati; nel Capitolo 2 vedremo più nel dettaglio i tipi di dato utilizzato, in particolare nella prima parte i dati sintetici suddivisi in reali e complessi, per poi concentrarci sui dati misurati in un processo industriale denominato Sulfur Recovery Unit (SRU); nel Capitolo 3 andremo a visualizzare i risultati ottenuti con tutti gli algoritmi per poterli analizzare e commentare.

# Capitolo 1

# Aspetti Metodologici

**Introduzione al Capitolo** In questo capitolo verrano presentati gli aspetti metodologici dei metodi identificati presi in esame. I metodi proposti sono stati inizialmente analizzati tramite dati generati con esempi sintetici. La creazione di questi dati iniziali consiste nella concatenazione di funzioni sinusoidali per ottenere un dato che variasse nel tempo in modo da poterne apprezzare i modi dinamici ottenuti. Successivamente andremo ad utilizzare dati reali per capire le differenze, sopratutto perchè entrano in gioco i vari rumori aggiunti da qualsiasi misurazione reale.

## 1.1 Dynamic Mode Decomposition

Il Dynamic Mode Decomposition è un approccio relativamente recente che ci permettere di approssimare sistemi dinamici in termini di strutture con dinamiche lineari che evolvono nel tempo.[1] Nonostante la procedura per identificare i modi DMD e gli autovalori sia puramente lineare, il sistema stesso può non essere lineare. Oltre che per analizzare il funzionamento interno di un sistema, può essere utilizzato anche per prevedere lo stato futuro del sistema, il che potrebbe darci un vantaggio non indifferente.

**Definizione e descrizione matematica del DMD** Prendiamo in considerazione un insieme di $n$ coppie di dati

$$\{(x_0, y_0), (x_1, y_1), \dots (x_n, y_n)\} \tag{1.1}$$

In cui ciascuna $x_i$ e $y_i$ è un vettore colonna di dimensione $m$.
Definiamo adesso due matrici $m$ x $n$:

$$X = [x_0\, x_1\, \dots\, x_n], \quad Y = [y_0\, y_1\, \dots\, y_n] \tag{1.2}$$

Se definiamo un operatore $A$ come

$$A = YX^{\dagger} \tag{1.3}$$

Dove $X^{\dagger}$ è la pseudo-inversa [2] di **X**, quindi la Dynamic Mode Decomposition della coppia **(X,Y)** è data dall'autodecomposizione di **A**. Cioè, i modi DMD e gli autovalori sono autovettori e autovalori di **A**.
La definizione di cui sopra è definita come DMD ed è la definizione più generale che si può applicare a qualsiasi dataset che soddisfa i requisiti indicati. Siamo principalmente interessati ai casi in cui **A** soddisfa l'equazione $y_i = Ax_i$ per qualsiasi $i$, più precisamente:

$$Y = AX \tag{1.4}$$

Chiaramente, **X** è l'insieme dei vettori di input e **Y** è l'insieme corrispondente dei vettori di output. Questa particolare interpretazione del DMD è

estremamente potente, in quanto fornisce un metodo conveniente per analizzare (e prevedere) sistemi dinamici per i quali le equazioni governanti sono sconosciute. Ci sono una serie di teoremi che accompagnano questa definizione del DMD. Uno dei teoremi più utili afferma **Y=AX** esattamente se e solo se **X** e **Y** sono linearmente consistenti ogni volta che $X_v = 0$ per qualche vettore $v$, avrò che $Y_v = 0$ [3]. La coerenza lineare è relativamente semplice da testare. Detto questo, la coerenza lineare non è un prerequisito obbligatorio per l'utilizzo del DMD. Anche se la soluzione DMD per **A** non soddisfa esattamente l'equazione **Y=AX**, è ancora una soluzione ai minimi quadrati, che minimizza l'errore in senso $L^2$.

**Algoritmo DMD** A prima vista, il compito di trovare decomposizione agli autovalori (eigendecomposition) di $A = YX^+$ non sembra essere un lavoro troppo grande. Infatti, quando **X** è **Y** sono di dimensioni ragionevoli, un paio di chiamate a **pinv** e **eig** metodi da Numpy o MATLAB faranno il loro dovere. Il problema viene quando **A** è veramente grande. Notare che **A** è $m$ x $m$, così quando $m$ (il numero di segnali in ogni campione temporale) è molto grande, trovare la decomposizione agli autovalori può diventare ingombrante. Fortunatamente, il problema può essere suddiviso in parti più piccole con l'aiuto dell'esatto algoritmo DMD.

1. Calcola l'SVD di **X**, eseguendo facoltativamente un troncamento di basso rango allo stesso tempo:

$$X = U\Sigma V^* \tag{1.5}$$

2. Calcolare Ã, la proiezione della matrice completa **A** su **U**:

$$\tilde{A} = U^* AU = U^* YV\Sigma^{-1} \tag{1.6}$$

3. Calcolare gli autovalori $\lambda_i$ e autovettori $\omega_i$ di Ã:

$$\tilde{A} = U^* AU = U^* YV\Sigma^{-1} \tag{1.7}$$

4. Ricostruire la decomposizione agli autovalori di $\mathbf{A}$ da $\mathbf{W}$ e $\lambda$. Gli autovalori per $\mathbf{A}$ sono equivalenti a quelli di $\tilde{A}$. Gli autovettori di $\mathbf{A}$ sono date dalle colonne di $\Phi$.

$$A\Phi = \Phi\Lambda, \quad \Phi = YV\Sigma^{-1}W \tag{1.8}$$

Adesso esaminiamo l'algoritmo passo dopo passo in Python iniziando dall'installazione di tutti i pacchetti necessari.

```
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from numpy import dot, multiply, diag, power
from numpy import pi, exp, sin, cos, cosh, tanh, real, imag
from numpy.linalg import inv, eig, pinv
from scipy.linalg import svd, svdvals
from scipy.integrate import odeint, ode, complex_ode
from warnings import warn
```

**Codice 1.1:** Importazione delle librerie necessarie.

Generiamo alcuni dati di prova. Bisogna tenere in considerazione che solitamente non si conoscono necessariamente le equazioni che governano un dato. In questo caso stiamo generando in maniera totalmente casuale alcune equazioni per creare un set di dati. Una volta generati possiamo anche dimenticarci che siano stati generati.

```
# define time and space domains
x = np.linspace(-10, 10, 100)
t = np.linspace(0, 6*pi, 80)
dt = t[2] - t[1]
Xm,Tm = np.meshgrid(x, t)

# create three spatiotemporal patterns
f1 = multiply(20-0.2*power(Xm, 2), exp((2.3j)*Tm))
f2 = multiply(Xm, exp(0.6j*Tm))
f3 = multiply(5*multiply(1/cosh(Xm/2), tanh(Xm/2)), 2*exp((0.1+2.8j)*Tm))

# combine signals and make data matrix
D = (f1 + f2 + f3).T

# create DMD input-output matrices
X = D[:,:-1]
Y = D[:,1:]
```

**Codice 1.2:** Generazione casuale di funzioni per la creazione di un dataset di prova.

Ora ci calcoliamo l'**SVD** di **X**. La prima matrice di maggior interesse è $\Sigma$, i valori singolari di **X**. Prendendo l'**SVD** di **X** ci permette di estrarre i suoi modi "veloci" e ridurre la dimensionalità del sistema alla Proper Orthogonal Decomposition (POD). La visualizzazione dei valori singolari ci permette di decidere quanti modi troncare.

```
# SVD of input matrix
U2,Sig2,Vh2 = svd(X, False)
```

**Codice 1.3:** Singular Value Decomposition.

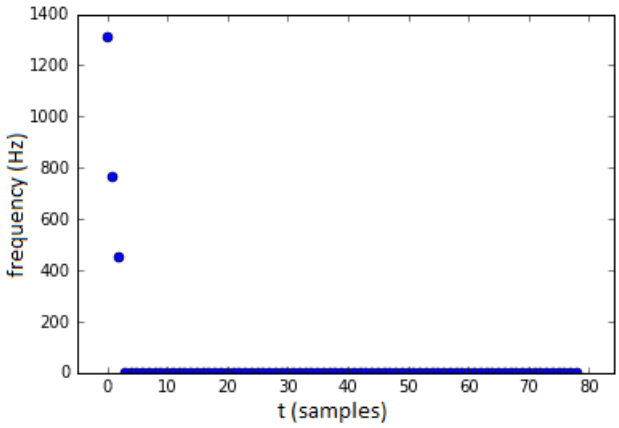



**Figura 1.1:** Rappresentazione grafica dei valori singolari. Sull'asse delle ordinate troviamo la frequenza in Hertz mentre sull'asse delle ordinate il numero di snapshot.

Dati i valori singolari mostrati sopra, concludiamo che i dati hanno tre modi di interesse significativo. Pertanto, troncheremo l'**SVD** per includere solo quei modi. Poi costruiamo $\tilde{A}$ e troviamo la sua decomposizione agli autovalori.

```
# rank-3 truncation
r = 3
U = U2[:,:r]
Sig = diag(Sig2)[:r,:r]
V = Vh2.conj().T[:,:r]

# build A tilde
Atil = dot(dot(dot(U.conj().T, Y), V), inv(Sig))
mu,W = eig(Atil)
```

**Codice 1.4:** Troncamento SVD e creazione di $\tilde{A}$.

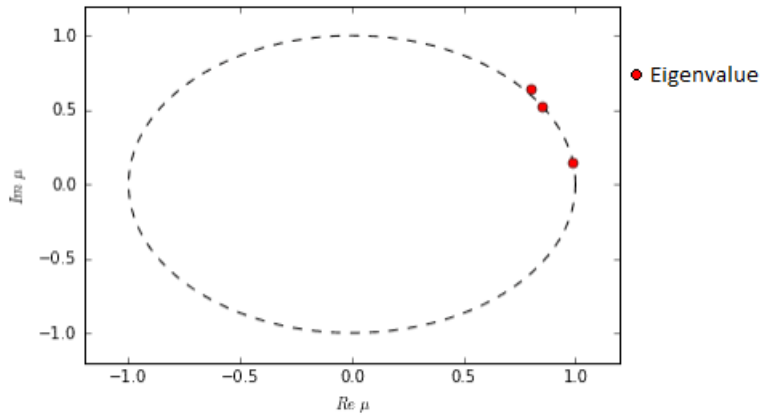



**Figura 1.2:** Rappresentazione grafica degli autovalori trovati tramite SVD. Gli autovalori si trovano sulla circonferenza unitaria quindi deduciamo che oscillano [1].

Ogni autovalore in $\Delta$ ci dice qualcosa sul comportamento dinamico del suo modo DMD. La sua esatta interpretazione dipende dalla natura della relazione tra $\mathbf{X}$ e $\mathbf{Y}$. Nel caso di equazioni differenziali possiamo trarre una serie di conclusioni. Se l'autovalore ha una parte immaginaria diversa da zero, allora c'è oscillazione nel modo DMD corrispondente. Se l'autovalore è all'interno del cerchio unitario, allora la modalità sta decadendo; se l'autovalore è esterno, allora il modo è crescente. Se l'autovalore cade esattamente sulla circonferenza unitaria, allora il modo non cresce nè decade.
Ora costruiamo i modi DMD.

```
# build DMD modes
Phi = dot(dot(dot(Y, V), inv(Sig)), W)
```

**Codice 1.5:** Costruzione dei modi DMD tramite il calcolo di Phi.

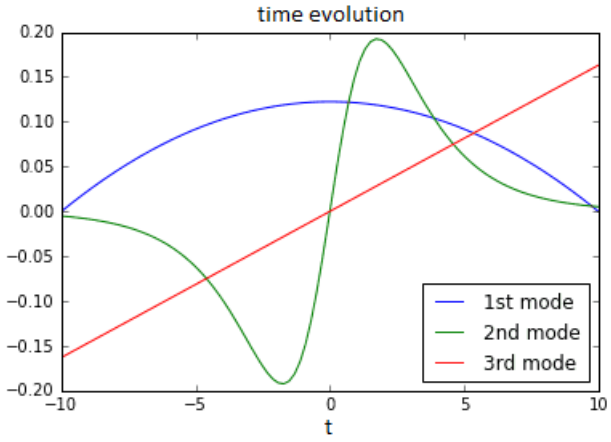



**Figura 1.3:** Rappresentazione grafica di Phi [1].

Le colonne di $\Phi$ sono i modi DMD tracciati sopra. Sono le strutture coerenti che crescono/decadono/oscillano nel sistema secondo diverse dinamiche temporali.
Qui è dove finisce tecnicamente l'algoritmo DMD. Dotato della decomposizione agli autovalori di $\mathbf{A}$ e una comprensione di base della natura del sistema $\mathbf{Y{=}AX}$, è possibile costruire una matrice $\Psi$ corrispondente all'evoluzione temporale del sistema. Per comprendere appieno il codice seguente, bisogna studiare la funzione $\mathbf{x(t)}$ per le equazioni differenziali nella sezione successiva:

```
# compute time evolution
b = dot(pinv(Phi), X[:,0])
Psi = np.zeros([r, len(t)], dtype='complex')
for i,_t in enumerate(t):
    Psi[:,i] = multiply(power(mu, _t/dt), b)
```

**Codice 1.6:** Costruzione dell'evoluzione nel tempo dei modi tramite il calcolo di Psi.

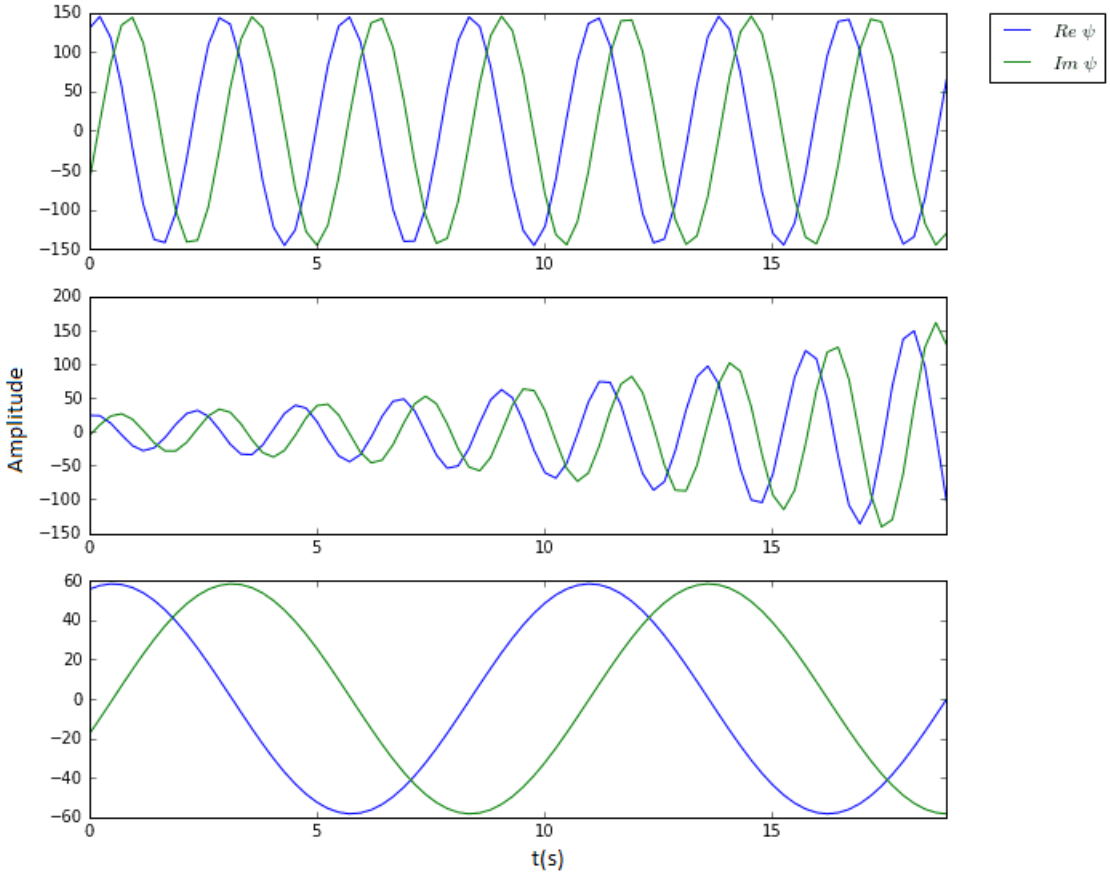



**Figura 1.4:** Rappresentazione grafica di tre dinamiche temporali di tre modi DMD [1].

Nella figura 1.4 troviamo le dinamiche temporali di tre modi DMD. Da notare come tutti e tre oscillano. Inoltre, il secondo modo sembra crescere in modo esponenziale, il che è confermato nel grafico degli autovalori.
Se desideriamo costruire un'approssimazione alla matrice di dati originali, bisogna moltiplicare semplicemente $\Phi$ e $\Psi$. L'originale e l'approssimazione coincidono esattamente in questo caso particolare.

```
# compute DMD reconstruction
D2 = dot(Phi, Psi)
np.allclose(D, D2) # True
```

**Codice 1.7:** Ricostruzione DMD.

**Conclusioni** Nonostante i suoi limiti, il DMD è uno strumento molto potente per analizzare e prevedere i sistemi dinamici. Dopo aver studiato la definizione formale del DMD, analizzato l'algoritmo passo dopo passo, è ora possibile estendere la conoscenza attraverso le estensioni del DMD.

## 1.2 Dynamic Mode Decomposition with Control

**Definizione e descrizione matematica del DMDc** Il Dynamic Mode Decomposition with Control (DMDc) è un estensione dell'algoritmo DMD per l'analisi di una grande quantità di dati provenienti da sistemi complessi che richiedono il controllo[4]. Utilizzando sia le misurazioni del sistema che il controllo esterno applicato, le dinamiche sottostanti non forzate possono essere estratte e specificate in modo indipendente dall'equazione, cioè non è necessario conoscere le equazioni di movimento sottostanti. Inoltre, viene scoperta e caratterizzata una descrizione di come gli input di controllo influenzano il sistema. Con una comprensione quantitativa delle caratteristiche input-output, è possibile generare un modello a basso ordine per la previsione e la progettazione dei controllori per sistemi complessi ad alta dimensione.

Il controllo di sistemi ad alta dimensione rimane una sfida estremamente complessa poiché molte strategie di controllo non si adattano bene alla dimensione del sistema. In particolare, i controllori sviluppati su un sistema completo possono richiedere calcoli computazionalmente proibitivi da implementare, introducendo latenze inaccettabilmente elevate. Inoltre, molte leggi di controllo sono determinate risolvendo un'ampia equazione di Riccati ($H_2$) o attraverso una procedura iterativa ($H_\infty$), che comporta un costo iniziale enorme. Pertanto, le strategie di controllo per affrontare i dati osservazionali ad alta dimensione ruotano attorno alle tecniche di riduzione della dimensionalità.

Il DMDc ha diversi vantaggi per i sistemi complessi ad alta dimensione. Innanzitutto, si basa sull'algoritmo DMD che è un'architettura senza equa-

zioni basata sui dati che ricostruisce le dinamiche sottostanti del sistema solo dalle misurazioni istantanee. Sono stati ottenuti notevoli successi nell'applicazione del DMD in campi come la dinamica dei fluidi che sono storicamente difficili da analizzare e per i quali costruire controllori risulta complesso a causa dell'enorme numero di stati spaziali richiesti per la simulazione. In secondo luogo, il DMD ha acquisito popolarità come metodo per sistemi con dinamiche non lineari, grazie a una forte connessione tra la DMD e la teoria dell'operatore di Koopman[5].

Come esempio motivante, il DMDc può essere applicata al campo dell'epidemiologia computazionale, focalizzandosi sull'eradicazione delle malattie. L'avvento di nuovi strumenti di monitoraggio e un sostanziale focus sull'analisi quantitativa dell'allocazione delle risorse sta iniziando a generare grandi insiemi di dati che descrivono la diffusione delle malattie infettive. Esiste una vasta letteratura incentrata sulla modellizzazione matematica della diffusione delle malattie infettive e sull'effetto del controllo esterno (ad esempio, vaccinazioni per la polio e zanzariere per la malaria). Una sfida comune nell'epidemiologia computazionale è decidere come modellare la diffusione della malattia, il che porta a un enorme numero di modelli fenomenologici. Tecniche senza equazioni come la DMD e la DMDc forniscono uno strumento complementare per l'analisi della diffusione spazio-temporale delle malattie infettive. Concentrandosi solo sui dati storici che contengono informazioni sullo stato (ad esempio, il numero di infezioni in una determinata posizione spaziale in un dato momento) e se sono state applicate misure di controllo (ad esempio, il numero di vaccinazioni in una determinata posizione spaziale in un dato momento), la DMDc scopre le proprietà dinamiche dei sistemi complessi.

Il metodo DMDc modifica l'assunzione di base del DMD. Il sistema dinamico lineare che collega lo stato futuro $x_{k+1}$ si basa ora sullo stato corrente $x_k$ e il controllo corrente $u_k$, come segue:

$$\mathbf{x}_{k+1} = \mathbf{A}\mathbf{x}_k + \mathbf{B}\mathbf{u}_k, \tag{1.9}$$

Dove $x_j \in \Re^n$, $u_j \in \Re^l$, $\mathbf{A} \in \Re^{n\times n}$, and $\mathbf{B} \in \Re^{n\times l}$ dove $n$ è il numero di variabili di stato e $l$ è il numero delle variabili di controllo. Le matrici di dati possono essere costruite con snapshots temporali dello stato e degli input di

controllo nel tempo. Gli snapshots dello stato **X** e **X'** vengono raccolti come segue:

$$\mathbf{X} = \begin{bmatrix} | & | & & | \\ \mathbf{x}_1 & \mathbf{x}_2 & \dots & \mathbf{x}_{m-1} \\ | & | & & | \end{bmatrix},$$
$$\mathbf{X}' = \begin{bmatrix} | & | & & | \\ \mathbf{x}_2 & \mathbf{x}_3 & \dots & \mathbf{x}_m \\ | & | & & | \end{bmatrix}, \tag{1.10}$$

Indichiamo una nuova sequenza di snapshots di input di controllo raccolte dalla seguente descrizione:

$$\boldsymbol{\Upsilon} = \begin{bmatrix} | & | & & | \\ \mathbf{u}_1 & \mathbf{u}_2 & \dots & \mathbf{u}_{m-1} \\ | & | & & | \end{bmatrix}. \tag{1.11}$$

L'equazione 1.9 può essere riscritta come segue:

$$\mathbf{X}' = \mathbf{A}\mathbf{X} + \mathbf{B}\boldsymbol{\Upsilon}. \tag{1.12}$$

Utilizzando le tre matrici di dati, è possibile trovare approssimazioni delle matrici lineari **A** e **B**. Nelle due sezioni seguenti, descriviamo come trovare i modi dinamici di **A** dato l'inclusione degli snapshots di controllo. La prima sezione delinea l'analisi e l'algoritmo se la matrice **B** è conosciuta o ben stimata. Se è sconosciuta, la seconda sezione descrive come scoprire sia **A** che **B** dalle matrici di osservazione.

**1° caso:** La matrice **B** è nota:

La sezione seguente descrive come trovare i modi dinamici e gli autovalori del sistema sottostante **A** quando la matrice **B** è nota. L'assunzione che **B** sia nota o ben stimata rappresenta una visione idealizzata della maggior parte dei sistemi complessi, ma aiuta a fornire una delle principali motivazioni di questo lavoro. Trovare i modi dinamici di **A** in

un sistema complesso in cui è stato applicato il controllo è essenziale per la progettazione di controllori e il posizionamento dei sensori. Se il controllo esterno è stato applicato al sistema, il DMD standard produrrebbe informazioni sulle dinamiche errate. Il caso più generale in cui **B** è sconosciuta sarà descritta nella sezione successiva.
L'equazione 1.12 può essere scritta accoppiando la matrice degli snapshots dello stato spostate nel tempo con la matrice degli snapshots di controllo e la matrice **B** nota.

$$\mathbf{X'} - \mathbf{B\Upsilon} = \mathbf{AX} \tag{1.13}$$

La matrice **A** può essere risolta con qualche passaggio matematico. Ancora una volta, la decomposizione ai valori singolari troncati di **X** fornisce la fattorizzazione delle matrici $\tilde{U}\tilde{\Sigma}\tilde{V}^*$. Quindi, l'approssimazione di **A** è data dalla seguente descrizione:

$$\mathbf{A} \approx \bar{\mathbf{A}} = (\mathbf{X'} - \mathbf{B\Upsilon})\tilde{\mathbf{V}}\tilde{\mathbf{\Sigma}}^{-1}\tilde{\mathbf{U}}^* \tag{1.14}$$

Nota che se gli snapshots di controllo sono $u_j = 0$, $\forall\ j \in [1, m]$, la derivazione è equivalente al DMD. Un modello dinamico del processo calcolato e della matrice di input può essere costruito come descritto di seguito:

$$\mathbf{x}_{k+1} = \bar{\mathbf{A}}\mathbf{x}_k + \mathbf{B}\mathbf{u}_k \tag{1.15}$$

Dove **x**, $\bar{A}$, and **B** hanno le stesse dimensioni delle matrici descritte in precedenza nell'equazione:

$$\mathbf{x}_{k+1} = \mathbf{A}\mathbf{x}_k \tag{1.16}$$

Tuttavia, se $r << n$, un modello più compatto ed efficiente dal punto di vista computazionale può essere ottenuto utilizzando la stessa trasformazione di base $\mathbf{Px}=\tilde{x}$ come descritto in precedenza per il DMD.

Ancora una volta, una trasformazione conveniente è già stata caclcolata tramite la decomposizione ai valori singolari di $\mathbf{X}$, data da $\mathbf{P}=\tilde{U}$. Il modello a bassa dimensione può essere derivato come segue:

$$\begin{aligned}\tilde{\mathbf{x}}_{k+1} &= \tilde{\mathbf{U}}^*\bar{\mathbf{A}}\tilde{\mathbf{U}}\tilde{\mathbf{x}}_k + \tilde{\mathbf{U}}^*\mathbf{B}\mathbf{u}_k, \\ &= \tilde{\mathbf{U}}^*(\mathbf{X}' - \mathbf{B}\boldsymbol{\Upsilon})\tilde{\mathbf{V}}\tilde{\boldsymbol{\Sigma}}^{-1}\tilde{\mathbf{x}}_k + \tilde{\mathbf{U}}^*\mathbf{B}\mathbf{u}_k \\ &= \tilde{\mathbf{A}}\tilde{\mathbf{x}}_k + \tilde{\mathbf{B}}\mathbf{u}_k\end{aligned} \tag{1.17}$$

L'approssimazione a bassa dimensione di $\mathbf{A}$ è data dalla seguente descrizione:

$$\tilde{\mathbf{A}} = \tilde{\mathbf{U}}^*(\mathbf{X}' - \mathbf{B}\boldsymbol{\Upsilon})\tilde{\mathbf{V}}\tilde{\boldsymbol{\Sigma}}^{-1} \tag{1.18}$$

La decomposizione in autovalori di $\tilde{A}$ definita da $\tilde{A}W = W\Delta$ fornisce gli autovettori che possono essere utilizzati per trovare le modalità dinamiche. Come nel DMD, i modi dinamici possono essere trovati con la seguente equazione:

$$\phi = (\mathbf{X}' - \mathbf{B}\boldsymbol{\Upsilon})\tilde{\mathbf{V}}\tilde{\boldsymbol{\Sigma}}^{-1}\mathbf{w} \tag{1.19}$$

Se $\lambda \neq 0$, allora questa è il modo DMD per $\lambda$. Se l'autovalore è 0, allora il modo dinamico viene calcolato utilizzando $\phi = \tilde{U}\omega$.

**2° caso:** La matrice $\mathbf{B}$ NON è nota:

L'assunzione che $\mathbf{B}$ sia conosciuto indica una significativa quantità di conoscenza su come il controllo influisce sul sistema. Questa sezione rilassa tale assunzione e dimostra in modo evidente che è possibile che è possibile trovare approssimazioni delle matrici $\mathbf{A}$ e $\mathbf{B}$ a partire dagli snapshot dello stato e del controllo. Per l'analista, questo è di gran

lunga più interessante poichè sono necessarie solo gli snapshot del controllo e dello stato per trovare le proprietà del processo sottostante **A** e come tale processo è influenzato dal controllo **B**. Il sistema dinamico descritto nell'equazione 1.13 può essere manipolato per ottenere la seguente rappresentazione:

$$\mathbf{X}' = [\mathbf{A} \quad \mathbf{B}] \begin{bmatrix} \mathbf{X} \\ \boldsymbol{\Upsilon} \end{bmatrix} = \mathbf{G}\boldsymbol{\Omega} \tag{1.20}$$

dove $\Omega$ contiene sia gli snapshot dello stato che del controllo. Qui, cerchiamo di trovare una soluzione migliore dell'operatore **G**, che ora contiene le dinamiche del processo **A** e la matrice di dati aggiunti $\Omega$, ottenendo $\Omega = U\Sigma V^* \approx \tilde{U}\tilde{\Sigma}\tilde{V}^*$. Il valore di troncamento della **SVD** per $\Omega$ viene definito come $p$. Si noti che il valore di troncamento di $\Omega$ dovrebbe essere maggiore di quello di **X**. La seguente computazione fornisce un'approssimazione di **G**:

$$\mathbf{G} \approx \bar{\mathbf{G}} = \mathbf{X}'\tilde{\mathbf{V}}\tilde{\mathbf{\Sigma}}^{-1}\tilde{\mathbf{U}}^*, \tag{1.21}$$

dove $\mathbf{G} \in \Re^{n\times(n+l)}$. Possiamo ora trovare approssimazioni delle matrici **A** e **B** suddividendo l'operatore lineare $\tilde{U}$ in due componenti separate come segue:

$$\begin{aligned} [\mathbf{A}, \quad \mathbf{B}] &\approx [\bar{\mathbf{A}}, \quad \bar{\mathbf{B}}] \\ &\approx [\mathbf{X}'\tilde{\mathbf{V}}\tilde{\mathbf{\Sigma}}^{-1}\tilde{\mathbf{U}}_1^*, \quad \mathbf{X}'\tilde{\mathbf{V}}\tilde{\mathbf{\Sigma}}^{-1}\tilde{\mathbf{U}}_2^*] \end{aligned} \tag{1.22}$$

dove $\tilde{U_1} \in \Re^{n\times p}$, $\tilde{U_2} \in \Re^{l\times p}$, e $\tilde{U_{1^*}}$, $\tilde{U_{2^*}^T}$. Simile all'equazione 1.22, viene costruito un modello dinamico utilizzando le matrici $\bar{A}$ e $\bar{B}$, ma per un sistema di grandi dimensioni in cui $n >> 1$, ciò è computazionalmente proibitivo. Qui, cerchiamo ancora un modello ad ordini ridotti di rango $r \neq n$, in cui è necessaria una trasformazione $\mathbf{x}=\mathbf{P}\ \tilde{x} \in \Re^r$. A differenza del DMD, i vettori singolari sinistri troncati $\tilde{U}$ non possono essere utilizzati per definire il sottospazio in cui si sviluppa lo stato. Per

l'equazione 1.22, i vettori singolari sinistri troncati di $\Omega$ definiscono lo spazio di input. Per trovare una trasformazione lineare **P** per lo stato **x**, utilizziamo un sottospazio di ordine ridotto dello spazio di output. Questa osservazione fondamentale consente alla DMDc di scoprire una rappresentazione ad ordine ridotto delle dinamiche **A** e della matrice di input **B**. Per trovare il sottospazio di ordine ridotto dello spazio di output, è necessaria una seconda decomposizione ai valori singolari. La matrice di dati dello spazio di output **X'** può essere approssimata dalla familiare **SVD** $\hat{U}\hat{\Sigma}\hat{V}^*$, dove il valore di troncamento è $r$ e $\hat{U} \in \Re^{n\times r}$, $\hat{\Sigma} \in \Re^{r\times r}$, and $\hat{V}^* \in \Re^{r\times(m-1)}$. Si noti che le due SVD avranno probabilmente valori di troncamento diversi per le matrici di input e output $p$ and $r$ and $p > r$. Utilizzando la trasformazione $\mathbf{x} = \hat{U}\tilde{x}$, è possibile calcolare le seguenti approssimazioni ad ordine ridotto di **A** e **B**:

$$\begin{aligned}\tilde{\mathbf{A}} &= \hat{\mathbf{U}}^*\bar{\mathbf{A}}\hat{\mathbf{U}} = \hat{\mathbf{U}}^*\mathbf{X}'\tilde{\mathbf{V}}\tilde{\mathbf{\Sigma}}^{-1}\tilde{\mathbf{U}}_1^*\hat{\mathbf{U}} \\ \tilde{\mathbf{B}} &= \hat{\mathbf{U}}^*\bar{\mathbf{B}} \quad = \hat{\mathbf{U}}^*\mathbf{X}'\tilde{\mathbf{V}}\tilde{\mathbf{\Sigma}}^{-1}\tilde{\mathbf{U}}_2^*\end{aligned} \tag{1.23}$$

dove $\tilde{A} \in \Re^{(r\times r)}$ e $\tilde{B} \in \Re^{(r\times l)}$. Possiamo quindi formare l'equazione ad ordine ridotto come segue

$$\tilde{\mathbf{x}}_{k+1} = \tilde{\mathbf{A}}\tilde{\mathbf{x}}_k + \tilde{\mathbf{B}}\mathbf{u}_k \tag{1.24}$$

Simile al DMD, i modi dinamici di **A** possono essere trovate risolvendo prima la decomposizione degli autovalori $\mathbf{A}\tilde{W} = W\Lambda$. La trasformazione degli autovalori ai modi dinamici di **A** è leggermente modificata e viene data nella seguente equazione:

$$\phi = \mathbf{X}'\tilde{\mathbf{V}}\tilde{\mathbf{\Sigma}}^{-1}\tilde{\mathbf{U}}_1^*\hat{\mathbf{U}}\mathbf{w}. \tag{1.25}$$

dove la relazione tra $\phi$ e $\omega$ è simile al DMD esatto.

**Algoritmo DMDc** Il seguente paragrafo descrive l'algoritmo.

1. **Raccogliere e costruire le matrici degli snapshot:**
   Raccogliere gli snapshot dello stato e del controllo e formare le matrici **X, X'** e $\Upsilon$ come descritto nell'equazione 1.10 e nell'equazione 1.11. Raccogliere le matrici dei dati **X** e $\Upsilon$ per costruire la matrice $\Omega$.

2. **Calcolare la SVD dello spazio di input $\Omega$.**
   Calcolare la decomposizione ai valori singolari di $\Omega$ come descritto nell'equazione seguente:

$$\begin{aligned}\mathbf{X} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^* &= \begin{bmatrix}\tilde{\mathbf{U}} & \tilde{\mathbf{U}}_{\text{rem}}\end{bmatrix}\begin{bmatrix}\tilde{\mathbf{\Sigma}} & 0\\ 0 & \mathbf{\Sigma}_{\text{rem}}\end{bmatrix}\begin{bmatrix}\tilde{\mathbf{V}}^*\\ \tilde{\mathbf{V}}^*_{\text{rem}}\end{bmatrix}\\ &\approx \tilde{\mathbf{U}}\tilde{\mathbf{\Sigma}}\tilde{\mathbf{V}}^*\end{aligned} \tag{1.26}$$

   ottenendo così la decomposizione $\Omega \approx \hat{U}\hat{\Sigma}\hat{V}^*$ con valore di troncamento $p$.

3. **Calcolare la SVD dello spazio di output X'.**
   Calcolare la decomposizione ai valori singolari di **X'** come descritto nell'equazione 1.26, ottenendo così la decomposizione **X'** $\approx \hat{U}\hat{\Sigma}\hat{V}^*$ con valore di troncamento $r$.

4. **Calcolare l'approssimazione degli operatori G = [A B].** Calcolare quanto segue:

$$\begin{aligned}\tilde{\mathbf{A}} &= \hat{\mathbf{U}}^*\mathbf{X}'\tilde{\mathbf{V}}\tilde{\mathbf{\Sigma}}^{-1}\tilde{\mathbf{U}}_1^*\hat{\mathbf{U}}\\ \tilde{\mathbf{B}} &= \hat{\mathbf{U}}^*\mathbf{X}'\tilde{\mathbf{V}}\tilde{\mathbf{\Sigma}}^{-1}\tilde{\mathbf{U}}_2^*\end{aligned} \tag{1.27}$$

5. **Eseguire la decomposizione degli autovalori di $\tilde{A}$.** Eseguire la decomposizione degli autovalori come segue:

$$\tilde{\mathbf{A}}\mathbf{W} = \mathbf{W}\mathbf{\Lambda} \tag{1.28}$$

**6. Calcolare i modi dinamici di A.**

$$\mathbf{\Phi} = \mathbf{X}'\tilde{\mathbf{V}}\tilde{\mathbf{\Sigma}}^{-1}\tilde{\mathbf{U}}_1^*\hat{\mathbf{U}}\mathbf{W} \tag{1.29}$$

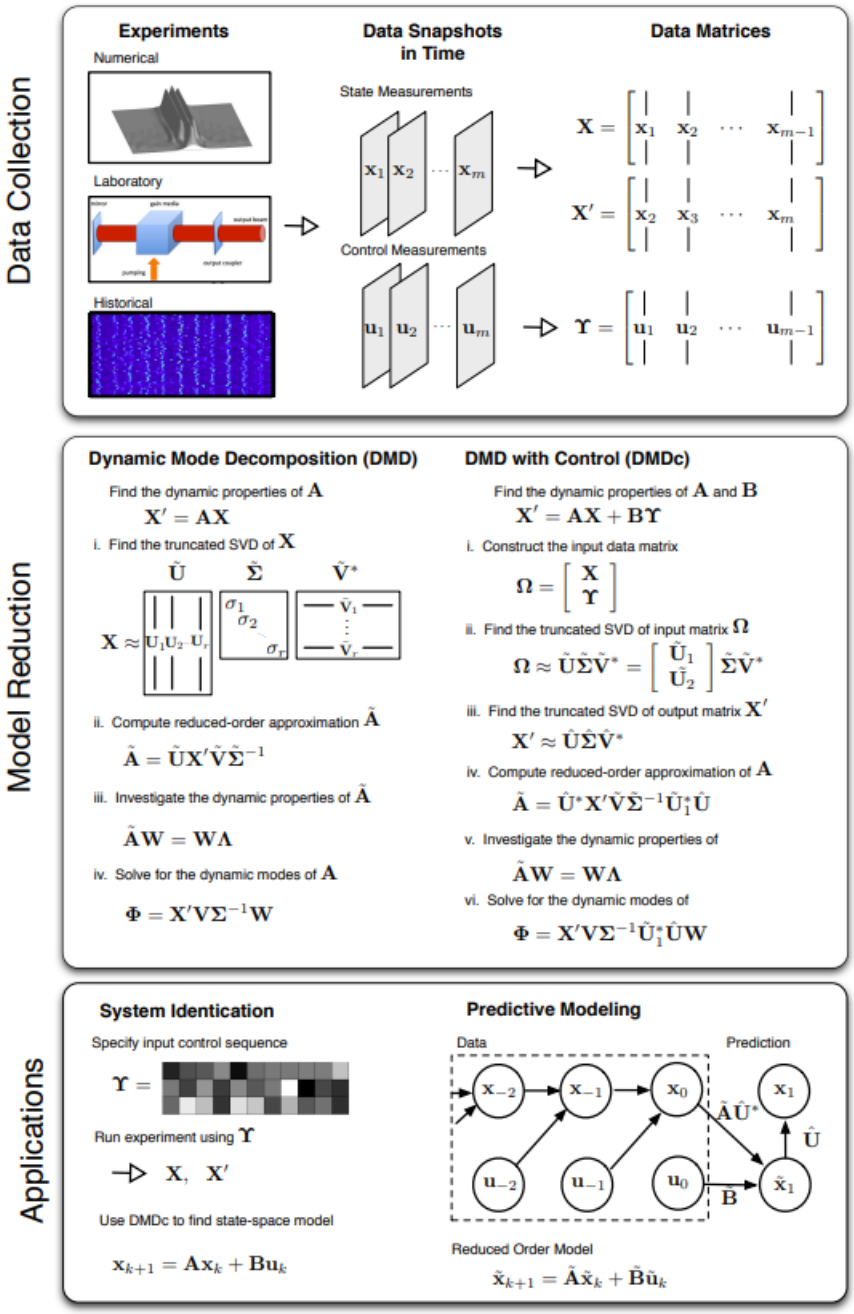



**Figura 1.5:** L'illustrazione descrive i tre principali componenti dell'applicazione del DMDc. Il pannello superiore descrive la raccolta dei dati da simulazioni numeriche, laboratorio o dati storici e la rappresentazione dei dati in matrici per i metodi. Il pannello centrale delinea la procedura per il DMD e il DMDc per scopi di confronto. Il pannello inferiore illustra due applicazioni pratiche del DMDc [4].

**Conclusioni** Il Dynamic Mode Decomposition (DMD) è un metodo data-driven e equation-free che aiuta a fronteggiare diverse sfide moderne come la riduzione della dimensionalità di grandi quantità di dati per caratterizzare e controllare sistemi complessi. Il DMD ha una forte connessione con la teoria degli operatori non lineari e riesce ad analizzare i modi dinamici in dataset spazio-temporali. Tuttavia, non produce modelli ridotti accurati per sistemi complessi con forzanti esterne. Il Dynamic Mode Decompisition with Control (DMDc) eredita i vantaggi del DMD, ma fornisce anche modelli accurati input-output per sistemi complessi con attuazione. Il metodo può essere applicato a dati provenienti da una varietà di fonti, tra cui dati storici, sperimentali e simulazioni "black-box".

## 1.3 Multi-Resolution DMD

Il metodo mrDMD è ispirato all'osservazione dei modi dinamici slow e fast che possono essere separati per applicazioni come la rimozione dello sfondo, come si può vedere nel seguente articolo [6]. Il mrDMD rimuove in modo ricorsivo il contenuto a bassa frequenza, o lentamente variabile, da un dataset di snapshot. Di solito, il numero di snapshot $M$ viene scelto in modo che i modi dinamici del DMD forniscono un'approssimazione di rango completo approssimativa della dinamica osservata. Quindi $M$ viene scelto in modo che tutto il contenuto ad alta (fast) e bassa (slow) frequenza sia presente. Nel mrDMD, $M$ viene inizialmente scelto alla stessa maniera in modo che si possa ottenere un'approssimazione di rango completo. Tuttavia, da questo primo passaggio attraverso i dati, i modi dinamici slow $m_1$ vengono rimossi e il dataset viene diviso in due segmenti con $M/2$ snapshot ciascuno. Il DMD viene nuovamente eseguito su ogni sequenza di snapshot $M/2$. Anche in questo caso, i modi dinamici slow $m_2$ vengono rimossi e l'algoritmo viene iterato fino ad una interruzione desiderata.

**Definizione e descrizione matematica del mrDMD.** Riguardo gli aspetti matematici, il mrDMD separata il risultato del DMD nel primo passaggio come segue:

$$\mathbf{x}_{\mathrm{mrDMD}}(t) = \sum_{k=1}^{M} b_k(0)\psi_k^{(1)}(\boldsymbol{\xi})\exp(\omega_k t) \;=\; \underbrace{\sum_{k=1}^{m_1} b_k(0)\psi_k^{(1)}(\boldsymbol{\xi})\exp(\omega_k t)}_{\text{(slow modes)}} + \underbrace{\sum_{k=m_1+1}^{M} b_k(0)\psi_k^{(1)}(\boldsymbol{\xi})\exp(\omega_k t)}_{\text{(fast modes)}} \tag{1.30}$$

dove $\psi_k^{(1)}(x)$ rappresentano i modi dinamici DMD calcolati dagli $M$ snapshot. La prima sommatoria in questa espressione rappresenta la dinamica del modi slow, mentre la seconda sommatoria rappresenta tutto il resto (definiti come modi fast). Quindi la seconda sommatoria può essere calcolata per ottenere la matrice:

$$\mathbf{X}_{M/2} = \sum_{k=m_1+1}^{M} b_k(0)\psi_k^{(1)}(\boldsymbol{\xi})\exp(\omega_k t)\,. \tag{1.31}$$

L'analisi DMD appena descritta può ora essere eseguita nuovamente sulla matrice dei dati $X_{M/2}$. Tuttavia, la matrice $X_{M/2}$ è ora divisa in due matrici.

$$\mathbf{X}_{M/2} = \mathbf{X}_{M/2}^{(1)} + \mathbf{X}_{M/2}^{(2)} \tag{1.32}$$

dove la prima matrice contiene i primi *M/2* snapshot e la seconda matrice contiene gli *M/2* snapshot rimanenti. Gli $m_2$ modi slow a questo livello sono dati da $\psi_k^{(2)}$, che vengono calcolati separatamente nella prima e seconda metà degli snapshot. Il processo di iterazione rimuove in modo ricorsivo le componenti a bassa frequenza costruendo man mano le nuove matrici $X_{M/2}, X_{M/4}, X_{M/8}$ e così via... fino a raggiungere il risultato desiderato. La soluzione approssimata del DMD può essere costruita nel seguente modo:

$$\begin{aligned}\mathbf{x}_{\mathrm{mrDMD}}(t) \;=\; & \sum_{k=1}^{m_1} b_k^{(1)}\psi_k^{(1)}(\boldsymbol{\xi})\exp(\omega_k^{(1)} t) + \sum_{k=1}^{m_2} b_k^{(2)}\psi_k^{(2)}(\boldsymbol{\xi})\exp(\omega_k^{(2)} t) \\ & + \sum_{k=1}^{m_3} b_k^{(3)}\psi_k^{(3)}(\boldsymbol{\xi})\exp(\omega_k^{(3)} t) + \cdots\end{aligned} \tag{1.33}$$

dove all'istante di valutazione $t$, vengono selezionati i modi dinamici esatti dalla finestra di campionamento a ogni livello di decomposizione. In particolare, $\psi_k^{(k)}$ e $\omega_k^{(k)}$ sono i modi dinamici e gli autovalori del DMD a *k-esimo* livello di decomposizione, i $b_k^{(k)}$ sono le proiezioni iniziali dei dati sull'intervallo di tempo considerato e gli $m_k$ sono il numero di modi slow mantenuti a ogni livello. Il vantaggio di questo metodo è evidente: vengono utilizzate diversi modi DMD spazio-temporali per rappresentare caratteristiche multi-resolution. Quindi non c'è un unico set di modi che domina la SVD e potenzialmente marginalizza le caratteristiche ad altre scale temporali. La figura 1.6 nel prossimo paragrafo illustra in modo visivo il processo di decomposizione multi-resolution DMD. Nella figura, viene eseguita una decomposizione a tre livelli con la scala più lenta rappresentata in blu (valori proprio e snapshot), la scala intermedia in rosso e la scala più veloce in verde. La connessione con l'analisi multi-resolution e l'analisi wavelet [7] è evidente anche dai pannelli inferiori, in cui si può veedere che il metodo mrDMD estrae successivamente informazione time-frequency in modo rigoroso. Come osservazione finale, la strategia di campionamento e l'algoritmo discusso qui (vedi figura 1.6) possono essere facilmente modificati poichè è necessario calcolare con precisione solo i modi slow ad ogni livello di decomposizione. Pertanto, è possibile modificare l'algoritmo in modo da campionare un numero fiso, ad esempio $M$, di snapshot di dati in ogni finestra di campionamento. Il valore di $M$ non deve essere grande poichè è necessario risolvere solo i modi slow. Difatti, il tasso di campionamento aumenterebbe man mano che la decomposizione procede da un livello all'altro. Ciò assicura che i livelli più bassi del mrDMD non campionino matrici molto piccole poichè il costo della SVD sarebbe notevolmente aumentato da un tasso di campionamento così piccolo. La soluzione dell'equazione 1.33 può essere resa più precisa. In particolare, bisogna tener conto del numero di livelli ($L$) della decomposizione, del numero di intervalli temporali ($J$) per ogni livello e del numero di modi ($m_L$).

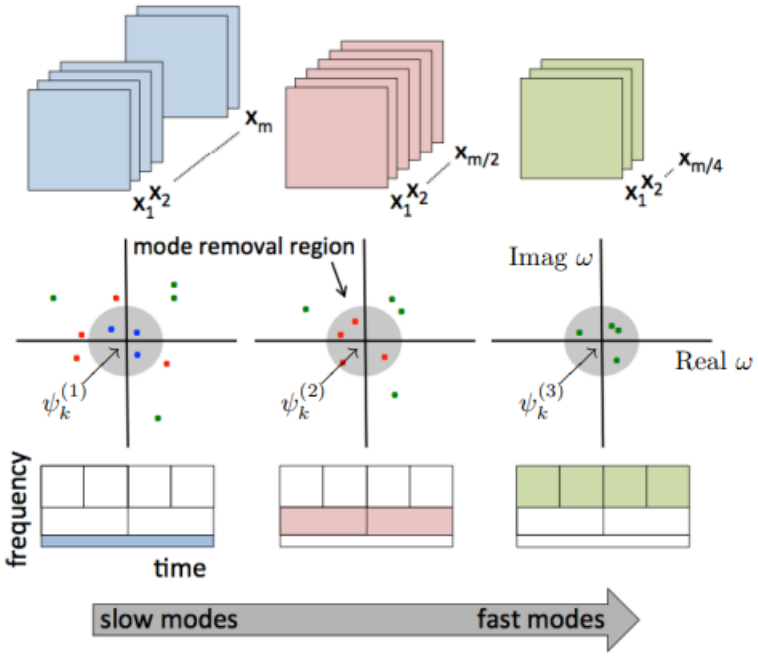



**Figura 1.6:** Rappresentazione del mrDMD, in cui il campionamento successivo dei dati, inizialmente con $M$ istantanee e ridotto di un fattore due ad ogni livello di risoluzione, è mostrato nelle figure superiori. Lo spettro DMD è mostrato nella figura centrale, dove sono presenti $m_1$ (punti blu) modi slow al livello più basso, $m_2$ (punti rossi) modi al livello successivo e $m_3$ (punti verdi) modi nella scala temporale più veloce. La regione ombreggiata rappresenta i modi rimossi a quel livello. I pannelli inferiori mostrano la decomposizione time-frequency simile a una wavelet[7] dei dati, con codifica a colori degli snapshot e delle rappresentazioni spettrali del DMD.

La soluzione è parametrizzata da questi tre indici:

$$\begin{aligned} &\ell = 1, 2, \cdots, L \ \text{ number of decomposition levels} \\ &j = 1, 2, \cdots, J \ \text{ number time bins per level } (J = 2^{(\ell-1)}) \\ &k = 1, 2, \cdots, m_L \ \text{ number of modes extracted at level } L. \end{aligned} \tag{1.34}$$

Per definire formalmente la soluzione in serie per $x_{mrDMD}(t)$, proponiamo la seguente funzione indicatrice:

$$f_{\ell,j}(t) = \begin{cases} 1 & t \in [t_j, t_{j+1}] \\ 0 & \text{elsewhere} \end{cases} \quad \text{with} \quad j = 1, 2, \cdots, J \quad \text{and} \quad J = 2^{(\ell-1)} \tag{1.35}$$

che è diversa da zero solo nell'intervallo, associato al valore di $j$. Il parametro $l$ indica il livello della decomposizione.

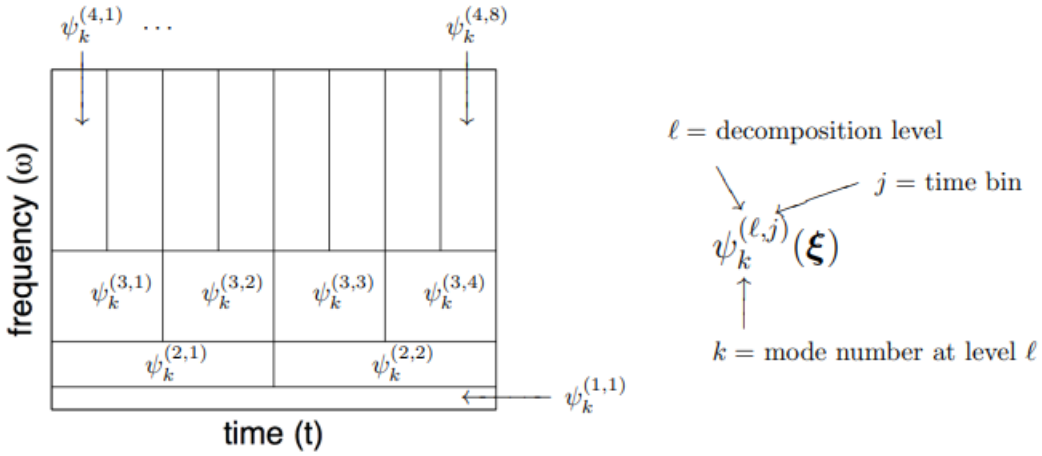



**Figura 1.7:** Illustrazione della decomposizione del mrDMD e della sua gerarchia. Sono rappresentati i modi $\psi_k^{l,j}(\Xi)$ e la loro posizione nella struttura di decomposzione. La terna di valori interi, $l$,$j$ e $k$, esprime in modo univoco il livello temporale, il bin e i modi della decomposizione [4].

I tre indici e la funzione indicatrice forniscono l'espansione della soluzione mrDMD come segue:

$$\mathbf{x}_{\text{mrDMD}}(t) = \sum_{\ell=1}^{L} \sum_{j=1}^{J} \sum_{k=1}^{m_L} f_{\ell,j}(t) b_k^{(\ell,j)} \psi_k^{(\ell,j)}(\boldsymbol{\xi}) \exp(\omega_k^{(\ell,j)} t) \tag{1.36}$$

Questa è una definizione concisa della soluzione mrDMD che include le informazioni sul livello, la posizione del time bin e il numero di modi estratti. La figura 1.8 illustra la decomposzione mrDMD in termini della soluzione. In particolare, ogni modo è rappresentata nel suo rispettivo time bin e livello. Un'interpretazione alternativa di questa soluzione e che fornisce il miglior

least-square fit, ad ogni livello $l$ della composizione, al sistema dinamico lineare.

$$\frac{d\mathbf{x}^{(\ell,j)}}{dt} = \mathbf{A}^{(\ell,j)}\mathbf{x}^{(\ell,j)} \tag{1.37}$$

dove la matrice $A^{(l,j)}$ cattura la dinamica in un dato time bin $j$ al livello $l$.
La funzione indicatrice $f_{l,j}(t)$ agisce come funzione di selezione per ciascun time bin. È interessante notare che questa funzione agisce come finestra di Gabor [8] di una trasformata di Fourier a finestre [3]. Poichè il nostro intervallo di campionamento ha un taglio netto sulla serie temporale, potrebbe introdurre alcune oscillazioni artificiali ad alta frequenza. L'analisi delle serie temporali, e in particolare i wavelet, introducono varie forme funzionali che possono essere utilizzate in modo vantaggioso. Quindi, pensando in modo più ampio, si può immaginare di utilizzare le funzioni wavelet per l'operazione di selezione, consentendo così alla funzione temporale $f_{l,j}(t)$ di assumere la forma di una delle tante basi di wavelet potenziali, ad esempio Haar, Daubechies, Mexican Hat, ecc.

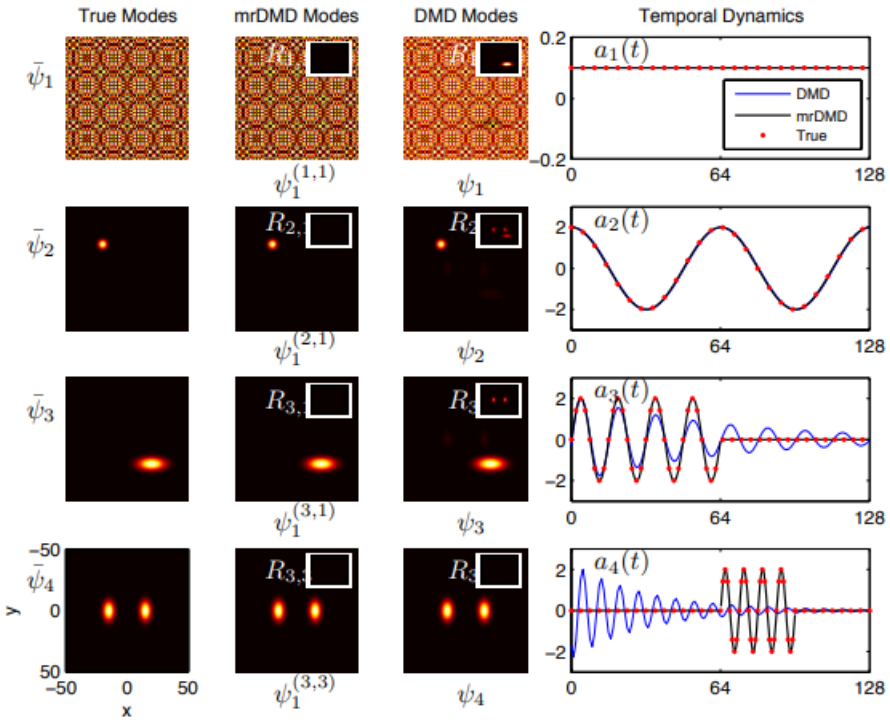



**Figura 1.8:** Confronto tra i modi reali ($\bar{\psi}_k$), i modi DMD ($\psi_k$) e i modi mrDMD ($\psi_k^j$). I modi reali, del mrDMD e del DMD sono mostrati nelle colonne da sinistra a destra. L'inserto dentro le immagini mostra l'errore relativo di modi del mrDMD ($R_{j,l} = \| \bar{\psi}_k - \psi_k \|$) rispetto ai modi reali. Le dinamiche temporali sono anche confrontate con le vere dinamiche mostrate con i punti rossi, le dinamiche ricostruite con il DMD sono mostrate con una linea blu e le dinamiche riscotruite con il mrDMD sono mostrate con una linea nera [4].

**Algoritmo mrDMD** Avendo noti i passaggi matematici del mrDMD, possiamo andare a descrivere in maniera generale quello che è l'algoritmo:

1. Calcolare il DMD per i dati disponibili.
2. Determinare i modi slow e i modi fast.
3. Trovare la migliore approssimazione DMD dei dati costruiti solo dai modi slow.
4. Sottrai l'approssimazione dei modi slow dai dati disponibili.

5. Dividi i dati disponibili a metà.

6. Ripeti la procedura per la prima metà dei dati (incluso questo passaggio).

7. Ripeti la proceduta per la seconda metà dei dati (incluso questo passaggio).

Questi appena elencati sono i passaggi di base. Alcune considerazioni:

- La procedura è ricorsiva.
- Per ottenere un tempo di esecuzione migliore, campioniamo i dati ad ogni livello della ricorsione. Ciò comporta la riduzione delle dimensioni dei dati al minimo necessario per catturare eventuali modi slow.
- Ad ogni livello possiamo eseguire una riduzione del rango utliazzando l'SVHT ottimale [9].
- Il passaggio 3 richiede di calcolare la "migliore approssimazione DMD". Utilizziamo il calcolo di **b** ottimale (preso in prestito dal spDMD) [10].
- Una volta che conosciamo il vettore ottimale **b**, possiamo costruire un'approssimazione dei modi slow dai dati. L'approssimazione viene quindi sottratta dai dati disponibili a quel livello corrispondente per produrre un nuovo insieme di dati per il livello successivo.

Nel prossimo capitolo andremo ad analizzare l'implementazione in Python dell'algoritmo, utilizzando come dataset sia dati sintetici che dati reali.

**Conclusioni** Uno dei punti forti del mrDMD è sicuramente il fatto che sia self-contained, ovvero che non ha dei parametri da modificare, il che è un grande vantaggio. Inoltre, ci consente di estrarre feature a diversi livelli di risoluzione, fornendo così un modo di identificare gli l'evoluzione dei dati sia nel lungo, medio e breve periodo. È in grado di gestire in maniera semplice fenomeni transitori e di rimuovere eventualmente il rumore senza molti sforzi aggiuntivi (soprattutto quando si utilizza l'SVHT ottimale). Infine, è efficiente e sufficientemente flessibile da poterlo utilizzare in vari ambiti scientifici, questo rende il mrDMD un ottimo strumento aggiuntivo per un ricercatore.

## 1.4 La novità: mrDMDc

L'obiettivo di questa tesi si è incentrato sulla costruzione di un nuovo metodo applicando concetti già esistenti, l'unione di queste informazioni ci ha permesso di implementare un nuovo strumento che potesse darci un risultato con dei dati industriali aventi variabili di controllo. Questo nuovo metodo è l'mrDMDc, ovvero un mrDMD che fosse in grado di costruire, ad ogni livello di decomposizione, una matrice di dati utilizzando anche le variabili di controllo.

**Algoritmo mrDMDc** L'algoritmo del mrDMDc è molto simile a quello del mrDMD, il concetto è fondamentalmente lo stesso, la caratteristica principale è stata quella di sostituire la ricostruzione DMD con la ricostruzione tramite DMDc, apportando le eventuali modifiche del caso. Nel prossimo capitolo spiegherò nel dettaglio il codice per l'implementazione.

# Capitolo 2

# Casi di Studio

**Introduzione al Capitolo** In questo capitolo andremo ad analizzare più nel dettaglio quelli che sono i codici utilizzati per la costruzione dei vari metodi, partendo da quello più semplice, ovvero il DMD, fino ad arrivare al mrDMDc.

**Implementazione in Python** Per l'implementazione di questi metodi ci siamo basati su degli esempi pratici riportati nel blog Humatic LABS, a cura di Robert Taylor [11]. L'ambiente di esecuzione dei codici che abbiamo usato è Jupyter Notebook.

## 2.1 mrDMD

Partiamo dall'implementazione del mrDMD per poi ricavarci il metodo mrDMDc.

Iniziamo ad importare le librerie necessarie (Codice 1.8)

```
import numpy as np
import matplotlib.pyplot as plt
from numpy import dot, multiply, diag, power
from numpy import pi, exp, sin, cos
from numpy.linalg import inv, eig, pinv, solve
from scipy.linalg import svd, svdvals
from math import floor, ceil # python 3.x
from sklearn.metrics import r2_score
from sklearn.metrics import mean_absolute_percentage_error
from sklearn.metrics import mean_squared_error
import math
from pydmd import DMDc

import scipy.io
```

**Codice 1.8:** Librerie necessarie per metodo mrDMD.

Costruiamo dei dati di prova tramite la concatenazione di funzioni sinusoidali (Codice 1.9)

```
# define time and space domains
x = np.linspace(-10, 10, 80)
t = np.linspace(0, 20, 1600)
Xm,Tm = np.meshgrid(x, t)

# create data
D = exp(-power(Xm/2, 2)) * exp(0.8j * Tm)
D += sin(0.9 * Xm) * exp(1j * Tm)
D += cos(1.1 * Xm) * exp(2j * Tm)
D += 0.6 * sin(1.2 * Xm) * exp(3j * Tm)
D += 0.6 * cos(1.3 * Xm) * exp(4j * Tm)
D += 0.2 * sin(2.0 * Xm) * exp(6j * Tm)
D += 0.2 * cos(2.1 * Xm) * exp(8j * Tm)
D += 0.1 * sin(5.7 * Xm) * exp(10j * Tm)
D += 0.1 * cos(5.9 * Xm) * exp(12j * Tm)
D += 0.1 * np.random.randn(*Xm.shape)
D += 0.03 * np.random.randn(*Xm.shape)
D += 5 * exp(-power((Xm+5)/5, 2)) * exp(-power((Tm-5)/5, 2))
D[:800,40:] += 2
D[200:600,50:70] -= 3
D[800:,:40] -= 2
D[1000:1400,10:30] += 3
D[1000:1080,50:70] += 2
D[1160:1240,50:70] += 2
D[1320:1400,50:70] += 2
D = D.T

# extract input-output matrices
X = D[:,:-1]
Y = D[:,1:]
```

**Codice 1.9:** Generazione dataset di prova.

Definiamo la funzione che ci permette di plottare i dati per avere un riscontro visivo (Codice 2.0)

```
def make_plot(X, x=None, y=None, figsize=(12, 8), title=''):
    """
    Plot of the data X
    """
    plt.figure(figsize=figsize)
    plt.title(title)
    X = np.real(X)
    CS = plt.pcolor(x, y, X)
    cbar = plt.colorbar(CS)
    plt.xlabel('Space')
    plt.ylabel('Time')
    plt.show()
```

**Codice 2.0:** Definizione funzione per plottare i risultati.

Adesso plottiamo la matrice dei dati che abbiamo appena generato (Figure 2.1 e 2.2)

```
make_plot(D.T, x=x, y=t)
```

**Figura 2.1:** Funzione per plottare la matrice di dati.

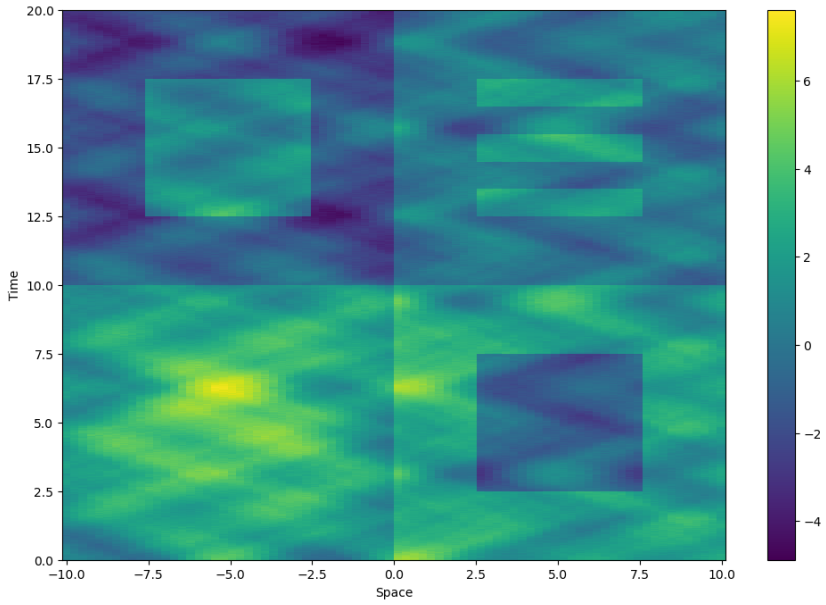



**Figura 2.2:** Rappresentazione grafica dataset di prova.

Definiamo la Singular Value Hard Treshold (SVHT) (Codice 2.2)

```
def svht(X, sv=None):
    # svht for sigma unknown                                    ## si applica la svht per trovare la matrice sigma
    m,n = sorted(X.shape) # ensures m <= n                      ## sorted(X.shape) ordina la dimensione della matrice
                                                                ## matrice ortogonale di destra sia maggiore della
                                                                ## dimensione della matrice ortogonale di sinistra.

    beta = m / n # ratio between 0 and 1                        ## beta è il median singular value della matrice X

    if sv is None:                                              ## sv -> singular value
        sv = svdvals(X)
    sv = np.squeeze(sv)                                         ## np.squeeze() remove axes of length one from a.

    omega_approx = 0.56 * beta**3 - 0.95 * beta**2 + 1.82 * beta + 1.43   ## approssimazione di omega

    return np.median(sv) * omega_approx                         ## np.median(sv) * omega_approx equivale al Tau
                                                                ## che sarebbe la soglia dell'SVHT

# determine rank-reduction
sv = svdvals(X)
tau = svht(X, sv=sv)
r = sum(sv > tau)
```

**Codice 2.2:** Definizione funzione per la Singular Value Hard Treshold.

Implementiamo il DMD (Codice 2.3)

```
def dmd(X, Y, truncate=None):
    if truncate == 0:
        # return empty vectors
        mu = np.array([], dtype='complex')
        Phi = np.zeros([X.shape[0], 0], dtype='complex')
    else:
        U2,Sig2,Vh2 = svd(X, False) # SVD of input matrix                ## U2,Sig2,Vh2 SVD senza troncamento della matrice X

        r = len(Sig2) if truncate is None else truncate # rank truncation ## len(Sig2) equivale al numero di elementi di Sig2

        U = U2[:,:r]                                                     ## U matrice troncata, da (80,15) a (80,7)
        Sig = diag(Sig2)[:r,:r]                                          ## Sig matrice troncata, da (15,) a (7,7)
        V = Vh2.conj().T[:,:r]                                           ## V matrice troncata, da (15,15) a (15,7)
                                                                         ## conj() è la congiunta

        Atil = dot(dot(dot(U.conj().T, Y), V), inv(Sig)) # build A tilde  ## A tilde è la proiezione di A su U,
                                                                         ## dot() è il prodotto scalare tra due array

        mu,W = eig(Atil)                                                 ## eig() calcola gli autovalori (mu) e gli autovettori
                                                                         ## destri di una matrice quadrata

        Phi = dot(dot(dot(Y, V), inv(Sig)), W) # build DMD modes          ## phi equivale ai modi dinamuci del DMD

    return mu, Phi                                                       ## dmd restituisce mu(autovalori) e phi(modi dinamici)

# "mu" rappresenta gli autovalori del sistema dinamico rappresentato dalla matrice di dati.
# Gli autovalori sono valori numerici che descrivono la stabilità e la dinamica del sistema.

# "phi" rappresenta i modi dinamici o le funzioni proprie del sistema.
# Questi modi dinamici sono funzioni che descrivono il comportamento del sistema in base alla sua dinamica intrinseca.
```

**Codice 2.3:** Definizione metodo DMD.

Adesso abbiamo tutti gli elementi necessari per la realizzazione del metodo mrDMD (Codice 2.4)

```
def mrdmd(D, level=0, bin_num=0, offset=0, max_levels=7, max_cycles=2, do_svht=True):
    """Compute the multi-resolution DMD on the dataset `D`, returning a list of nodes
    in the hierarchy. Each node represents a particular "time bin" (window in time) at
    a particular "level" of the recursion (time scale). The node is an object consisting
    of the various data structures generated by the DMD at its corresponding level and
    time bin. The `level`, `bin_num`, and `offset` parameters are for record keeping
    during the recursion and should not be modified unless you know what you are doing.
    The `max_levels` parameter controls the maximum number of levels. The `max_cycles`
    parameter controls the maximum number of mode oscillations in any given time scale
    that qualify as "slow". The `do_svht` parameter indicates whether or not to perform
    optimal singular value hard thresholding."""


    # 4 times nyquist limit to capture cycles                ## per il teorema di nyquist, un segnale sinusoidale puo'
                                                             ## essere ricostruito senza perdere informazioni, fintanto che
                                                             ## viene campionato ad una frequenza due volte o piu' maggiori
                                                             ## della frequenza massima. (valore tipico: 4 volte maggiori)

    nyq = 8 * max_cycles                                     ## nyq = 8 * 2

    # time bin size
    bin_size = D.shape[1]                                    ## bin_size equivale al numero di colonne (1600)
    if bin_size < nyq:                                       ## bin_size(1600)<nyq(16)
        return []

    # extract subsamples
    step = floor(bin_size / nyq) # max step size to capture cycles   ## floor è una funzione che approssima per difetto
                                                                     ## esempio floor(2.9) equivale a 2.
                                                                     ## in questo caso bin_size / nyq == 100 , quindi step=100

    _D = D[:,::step]                                          ## D[:,:] è uno slice assignment, in particolare D[:,::step]
                                                              ## considera i valori presi ogni step di tutte le righe e crea
                                                              ## una nuova matrice con solo i valori considerati.


    X = _D[:,:-1]                                             ## in X considera tutti i valori (di _D) escludendo l'ultima
                                                              ## colonna.

    Y = _D[:,1:]                                              ## in Y considera tutti i valori (di _D) escludendo la prima
                                                              ## colonna.

    # determine rank-reduction
    if do_svht:
        _sv = svdvals(_D)                                     ## svdvals(_D) trova i valori singolari della matrice campionata
                                                              ## _sv è un array con i valori singolari.
```

```
    tau = svht(_D, sv=_sv)                                  ## tau -> soglia SVHT
    r = sum(_sv > tau)                                      ## r equivale al n. di valori singolari maggiori della soglia tau
else:
    r = min(X.shape)                                        ## r -> X.shape = (80, 15) quindi r=min(X.shape) è 15.

# compute dmd
mu,Phi = dmd(X, Y, r)                                       ## applica il dmd alle matrici campionate

# frequency cutoff (oscillations per timestep)
rho = max_cycles / bin_size                                 ## frequenza di taglio rho definita come n° max di cicli
                                                            ## classificati come "slow" diviso il numero di colonne della
                                                            ## matrice D.  2/1600=0.00125

# consolidate slow eigenvalues (as boolean mask)
slow = (np.abs(np.log(mu) / (2 * pi * step))) <= rho        ## ritorna il valore assoluto [abs()] del logaritmo di mu diviso
                                                            ## 2*pi greco*step, che deve essere minore o uguale a rho.
                                                            ## in questo caso [false false false false true true false]

n = sum(slow) # number of slow modes                        ## il numero dei modi "slow" è dato dalla somma di slow.
                                                            ## in questo caso 2.

# extract slow modes (perhaps empty)                        ## si estrae i modi "slow"
mu = mu[slow]                                               ## in mu salva solo gli autovalori "slow"
                                                            ## da 7 autovalori a 2 autovalori.


Phi = Phi[:,slow]                                           ## in phi salva solo i modi dinamici "slow"
                                                            ## phi.shape era (80,7) adesso è (80,2).

if n > 0:                                                   ## se il n° di modi "slow" è maggiore di zero

    # vars for the objective function for D (before subsampling)
    Vand = np.vander(power(mu, 1/step), bin_size, True)     ## vander() restituisce una matrice di Vandermonde, come paramentri
                                                            ## vanno passati: un array 1-D (in questo caso mu elevato a potenza
                                                            ## 1/1600), il numero di colonne dell'uscita e un valore booleano
                                                            ## che indica l'incremento (se True allora le colonne saranno
                                                            ## x^0, x^1, x^2... se False saranno x^(N-1), x^(N-2),...)

    P = multiply(dot(Phi.conj().T, Phi), np.conj(dot(Vand, Vand.conj().T)))  ## multiply() serve per moltiplicare due array.
                                                                             ## in questo caso tra il [prodotto scalare della
                                                                             ## la congiunta di phi trasposta e phi] e
                                                                             ## [la congiunta del prodotto scalare di Vand
                                                                             ## e la congiunta di Vand trasposta]
    q = np.conj(diag(dot(dot(Vand, D.conj().T), Phi)))      ##
```

```
    # find optimal b solution
    b_opt = solve(P, q).squeeze()                          ## b = P^-1 * q
                                                           ## solve() trova le radici di P risolvendo per q

    # time evolution
    Psi = (Vand.T * b_opt).T                               ## Psi matrice (2,1600)

else:

    # zero time evolution
    b_opt = np.array([], dtype='complex')
    Psi = np.zeros([0, bin_size], dtype='complex')

# dmd reconstruction
D_dmd = dot(Phi, Psi)                                      ## D_dmd equivale al prodotto scalare tra i modi dinamici slow
                                                           ## e la matrice che rappresenta l'evoluzione nel tempo

# remove influence of slow modes
D = D - D_dmd                                              ## Rimuove dai dati rimanenti le componenti slow

# record keeping
node = type('Node', (object,), {})()
node.level = level             # level of recursion
node.bin_num = bin_num         # time bin number
node.bin_size = bin_size       # time bin size
node.start = offset            # starting index
node.stop = offset + bin_size # stopping index
node.step = step               # step size
node.rho = rho                 # frequency cutoff
node.r = r                     # rank-reduction
node.n = n                     # number of extracted modes
node.mu = mu                   # extracted eigenvalues
node.Phi = Phi                 # extracted DMD modes
node.Psi = Psi                 # extracted time evolution
node.b_opt = b_opt             # extracted optimal b vector
nodes = [node]

# split data into two and do recursion                     ## Qui i dati rimanenti vengono divisi in due e viene applicato
                                                           ## nuovamente il mrdmd su entrambe le metà e così via in modo
                                                           ## ricorsivo...

if level < max_levels:
    split = ceil(bin_size / 2) # where to split            ## ceil(x) approssima per eccesso x
    nodes += mrdmd(
        D[:,:split],
        level=level+1,
        bin_num=2*bin_num,
        offset=offset,
        max_levels=max_levels,
        max_cycles=max_cycles,
        do_svht=do_svht
        )
    nodes += mrdmd(
        D[:,split:],
        level=level+1,
        bin_num=2*bin_num+1,
        offset=offset+split,
        max_levels=max_levels,
        max_cycles=max_cycles,
        do_svht=do_svht
        )
return nodes                                               ## la funzione ritorna una lista di nodi che rappresenta
                                                           ## l'evoluzione dei dati spazio-temporali nel tempo
```

**Codice 2.4:** Definizione metodo mrDMD.

Adesso eseguiamo il metodo (Codice 2.5)

```
nodes = mrdmd(D)
```

**Codice 2.5:** Salviamo in una variabile i nodi prodotti dalla funzione mrDMD.

Ora abbiamo bisogno di definire una funzione che ci permette di unire i vari livelli di ricorsione. La funzione è la seguente (Codice 2.6):

```
def stitch(nodes, level):

    # get length of time dimension
    start = min([nd.start for nd in nodes])
    stop = max([nd.stop for nd in nodes])
    t = stop - start

    # extract relevant nodes
    nodes = [n for n in nodes if n.level == level]
    nodes = sorted(nodes, key=lambda n: n.bin_num)

    # stack DMD modes
    Phi = np.hstack([n.Phi for n in nodes])

    # allocate zero matrix for time evolution
    nmodes = sum([n.n for n in nodes])
    Psi = np.zeros([nmodes, t], dtype='complex')

    # copy over time evolution for each time bin
    i = 0
    for n in nodes:
        _nmodes = n.Psi.shape[0]
        Psi[i:i+_nmodes,n.start:n.stop] = n.Psi
        i += _nmodes

    return Phi,Psi
```

**Codice 2.6:** Definizione funzione per unire insieme il risultato di ogni ricorsione.

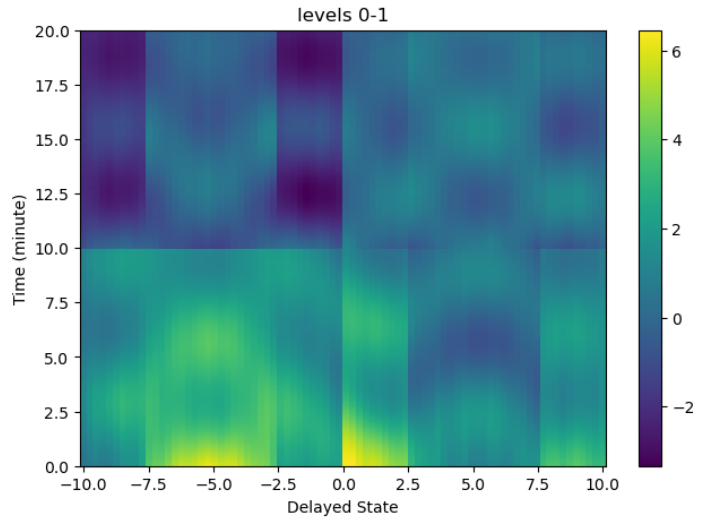
Eseguiamo lo stitch tra tutti i livelli per ottenere il risultato. In questo caso stiamo andando a mettere insieme 7 livelli di ricorsione, questo può essere cambiato nei parametri del metodo mrDMD (codice 2.4). Inoltre andiamo a stampare il Mean Squared Error (MSE) per avere un dato di confronto oltre l'immagine della ricostruzione (Codice 2.7).

```
D_mrdmd = dot(*stitch(nodes, 0))
D0 = D[0,:]
print("D0 è\n",D0)
D_mrdmd0 = D_mrdmd[0,:]
print("D_mrdmd0 è\n",D_mrdmd0)
```

```
for i in range(1, 7):
    D_mrdmd += dot(*stitch(nodes, i))
    make_plot(D_mrdmd.T, x=x, y=t, title='levels 0-' + str(i), figsize=(7.5, 5))
    print("Percentuale di errore:")
    print(round(mean_squared_error(np.abs(D0),np.abs(D_mrdmd0))*100,5))
```

**Codice 2.7:** 1. Dichiarazione della variabile D_mrdmd tramite chiamata a funzione Stitch. 2. Creazioni delle matrici unidimensionali D0 e D_mrdmd0 che ci serviranno per stimare l'errore percentuale (MSE). 3. Ciclo iterativo per unire il risultato di ogni iterazione.

Qui di seguito troviamo il risultato della ricostruzione (Figure 2.3 - 2.8)




```
Percentuale di errore:
47.6286
```

**Figura 2.3:** Risultato del primo livello di iterazione.

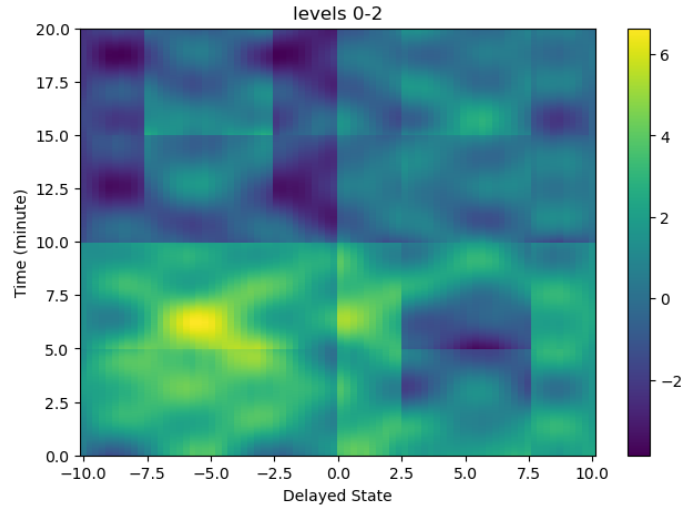



**Figura 2.4:** Risultato del secondo livello di iterazione.

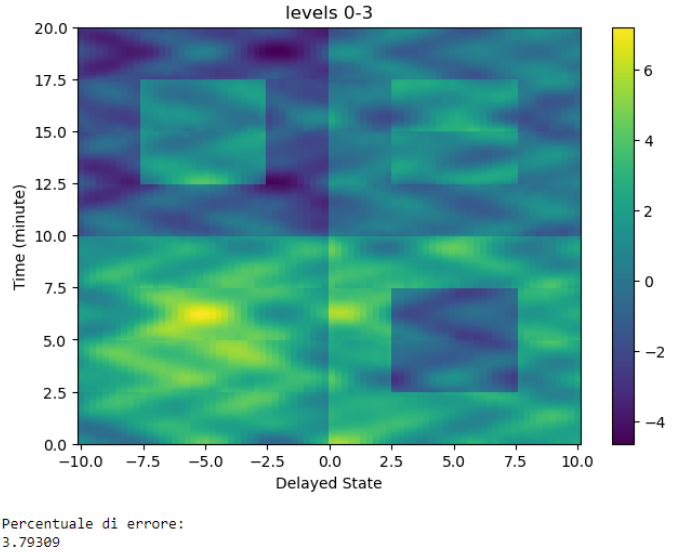
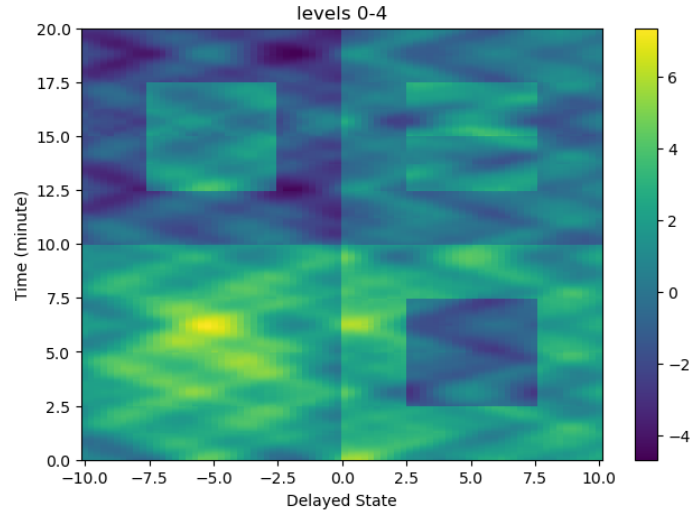
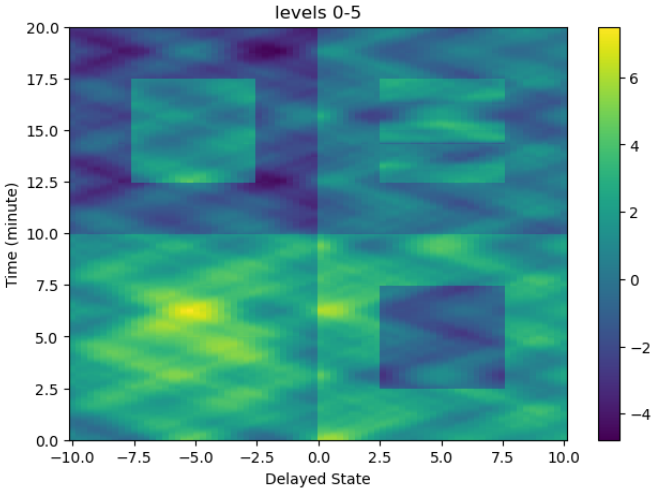



**Figura 2.5:** Risultato del terzo livello di iterazione.




```
Percentuale di errore:
1.53906
```

**Figura 2.6:** Risultato del quarto livello di iterazione.




```
Percentuale di errore:
1.04883
```

**Figura 2.7:** Risultato del quinto livello di iterazione.

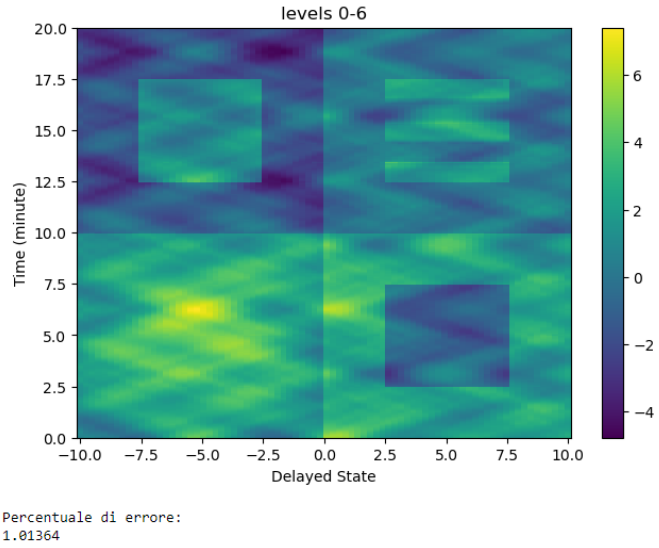



**Figura 2.8:** Risultato del sesto livello di iterazione.

Come possiamo notare, ad ogni livello di ricorsione il risultato migliora. Il miglioramento viene man mano sempre a diminuire come possiamo notare dal MSE [12], di conseguenza conviene tenere il numero di iterazione adeguato. Se mettiamo a confronto il risultato dell'ultima iterazione nel codice 2.8, quindi l'immagine "levels 0-6" con l'immagine del codice 2.1, faremo fatica a notare la differenza.
I dati utilizzati in questo esempio erano dei dati di prova generati casualmente, andando a modificare il dataset iniziale, possiamo provare ad ottenere un nuovo risultato.

## 2.2 mrDMDc

A questo punto, possiamo introdurre l'implementazione del mrDMDc. Per utilizzare questo metodo abbiamo bisogno di applicarlo ad un dataset che abbia delle variabili di controllo, il dataset in questione lo andremo ad introdurre tra qualche paragrafo, per ora concentriamoci sulle modifiche da apportare al codice. La differenza con il codice del mrDMD sta nel sostituire il metodo DMD con il metodo DMDc, quindi le modifiche da apportare so-

no solo all'interno della funzione mrdmd, per semplicità ne abbiamo creata una nuova rinominata mrdmdc. Le modifiche sostanziali sono l'aggiunta del parametro U, ovvero la matrice delle variabili di controllo, e la sostituzione della parte di codice che andava a ricostruire il dato del DMD a partire da mu e Phi, ovvero, rispettivamente gli autovalori del sistema dinamico e i modi dinamici ricavati tramite il DMD. La sostituzione della parte di ricostruzione è dovuta al fatto che la funzione DMDc ha già al suo interno la ricostruzione del dato. La funzione DMDc è stata presa in prestito dalla libreria PyDMD [13].

Qui di seguito la funzione mrDMDc (Codice 2.9):

```
def mrdmdc(D, U, level=0, bin_num=0, offset=0, max_levels=15, max_cycles=1, do_svht=True):
    """Compute the multi-resolution DMD on the dataset `D`, returning a list of nodes
    in the hierarchy. Each node represents a particular "time bin" (window in time) at
    a particular "level" of the recursion (time scale). The node is an object consisting
    of the various data structures generated by the DMD at its corresponding level and
    time bin. The `level`, `bin_num`, and `offset` parameters are for record keeping
    during the recursion and should not be modified unless you know what you are doing.
    The `max_levels` parameter controls the maximum number of levels. The `max_cycles`
    parameter controls the maximum number of mode oscillations in any given time scale
    that qualify as "slow". The `do_svht` parameter indicates whether or not to perform
    optimal singular value hard thresholding."""


    # 4 times nyquist limit to capture cycles                ## per il teorema di nyquist, un segnale sinusoidale puo'
                                                             ## essere ricostruito senza perdere informazioni, fintanto che
                                                             ## viene campionato ad una frequenza due volte o piu' maggiori
                                                             ## della frequenza massima. (valore tipico: 4 volte maggiori)

    nyq = 8 * max_cycles                                     ## nyq = 8 * 2

    # time bin size
    bin_size = D.shape[1]                                    ## bin_size equivale al numero di colonne (1600)
    if bin_size < nyq:                                       ## bin_size(1600)<nyq(16)
        return []

    # extract subsamples
    step = floor(bin_size / nyq) # max step size to capture cycles   ## floor è una funzione che approssima per difetto
                                                                     ## esempio floor(2.9) equivale a 2.
                                                                     ## in questo caso bin_size / nyq == 100 , quindi step=100

    _D = D[:,::step]                                          ## D[:,:] è uno slice assignment, in particolare D[:,::step]
                                                              ## considera i valori presi ogni step di tutte le righe e crea
                                                              ## una nuova matrice con solo i valori considerati.
    U = U[:,::step]


    X = _D[:,:-1]                                             ## in X considera tutti i valori (di _D) escludendo l'ultima
                                                              ## colonna.

    Y = _D[:,1:]                                              ## in Y considera tutti i valori (di _D) escludendo la prima
                                                              ## colonna.

    # determine rank-reduction
    if do_svht:
        _sv = svdvals(_D)                                     ## svdvals(_D) trova i valori singolari della matrice campionata
                                                              ## _sv è un array con i valori singolari.
```

```
    tau = svht(_D, sv=_sv)                                  ## tau -> soglia SVHT
    r = sum(_sv > tau)                                      ## r equivale al n. di valori singolari maggiori della soglia tau
else:
    r = min(X.shape)                                        ## r -> X.shape = (80, 15) quindi r=min(X.shape) è 15.
# determine rank-reduction
if do_svht:
    _sv = svdvals(_D)                                       ## svdvals(_D) trova i valori singolari della matrice campionata
                                                            ## _sv è un array con i valori singolari.

    tau = svht(_D, sv=_sv)                                  ## tau -> soglia SVHT
    r = sum(_sv > tau)                                      ## r equivale al n. di valori singolari maggiori della soglia tau
else:
    r = min(X.shape)                                        ## r -> X.shape = (80, 15) quindi r=min(X.shape) è 15.

# frequency cutoff (oscillations per timestep)
rho = max_cycles / bin_size                                 ## frequenza di taglio rho definita come n° max di cicli
                                                            ## classificati come "slow" diviso il numero di colonne della
                                                            ## matrice D.  2/1600=0.00125

# Ricostruzione con DMDc
dmdc = DMDc(svd_rank=-1)
D0 = _D[:,:]
U0 = U[:,1:]
print(D0.shape)
print(U0.shape)
dmdc.fit(D0,U0)
D, mu, Phi = dmdc.reconstructed_data()

# consolidate slow eigenvalues (as boolean mask)
slow = (np.abs(np.log(mu) / (2 * pi * step))) <= rho        ## ritorna il valore assoluto [abs()] del logaritmo di mu diviso
                                                            ## 2*pi greco*step, che deve essere minore o uguale a rho.
                                                            ## in questo caso [false false false false true true false]

n = sum(slow) # number of slow modes                        ## il numero dei modi "slow" è dato dalla somma di slow.
                                                            ## in questo caso 2.


if n > 0:                                                   ## se il n° di modi "slow" è maggiore di zero

    # vars for the objective function for D (before subsampling)
    Vand = np.vander(power(mu, 1/step), bin_size, True)     ## vander() restituisce una matrice di Vandermonde, come parametri
                                                            ## vanno passati: un array 1-D (in questo caso mu elevato a potenza
                                                            ## 1/1600), il numero di colonne dell'uscita e un valore booleano
                                                            ## che indica l'incremento (se True allora le colonne saranno
                                                            ## x^0, x^1, x^2... se False saranno x^(N-1), x^(N-2),...)
```

```
    P = multiply(dot(Phi.conj().T, Phi), np.conj(dot(Vand, Vand.conj().T)))  ## multiply() serve per moltiplicare due array.
                                                                             ## in questo caso tra il [prodotto scalare della
                                                                             ## la congiunta di phi trasposta e phi] e
                                                                             ## [la congiunta del prodotto scalare di Vand
                                                                             ## e la congiunta di Vand trasposta]
    q = np.conj(diag(dot(dot(Vand, D.conj().T), Phi)))    ##

    # find optimal b solution
    b_opt = solve(P, q).squeeze()                         ## b = P^-1 * q
                                                          ## solve() trova le radici di P risolvendo per q

    # time evolution
    Psi = (Vand.T * b_opt).T                              ## Psi matrice (2,1600)

else:

    # zero time evolution
    b_opt = np.array([], dtype='complex')
    Psi = np.zeros([0, bin_size], dtype='complex')

# record keeping
node = type('Node', (object,), {})()
node.level = level            # level of recursion
node.bin_num = bin_num        # time bin number
node.bin_size = bin_size      # time bin size
node.start = offset           # starting index
node.stop = offset + bin_size # stopping index
node.step = step              # step size
node.rho = rho                # frequency cutoff
node.r = r                    # rank-reduction
node.n = n                    # number of extracted modes
node.mu = mu                  # extracted eigenvalues
node.Phi = Phi                # extracted DMD modes
node.Psi = Psi                # extracted time evolution
node.b_opt = b_opt            # extracted optimal b vector
nodes = [node]

# split data into two and do recursion                ## Qui i dati rimanenti vengono divisi in due e viene applicato
                                                      ## nuovamente il mrdmd su entrambe le metà e così via in modo
                                                      ## ricorsivo...
```

```
if level < max_levels:
    split = ceil(bin_size / 2) # where to split        ## ceil(x) approssima per eccesso x
    nodes += mrdmdc(
        D[:,:split],
        U[:,:split],
        level=level+1,
        bin_num=2*bin_num,
        offset=offset,
        max_levels=max_levels,
        max_cycles=max_cycles,
        do_svht=do_svht
        )
    nodes += mrdmdc(
        D[:,split:],
        U[:,split:],
        level=level+1,
        bin_num=2*bin_num+1,
        offset=offset+split,
        max_levels=max_levels,
        max_cycles=max_cycles,
        do_svht=do_svht
        )
return nodes                                          ## la funzione ritorna una lista di nodi che rappresenta
                                                      ## l'evoluzione dei dati spazio-temporali nel tempo
```

**Codice 2.9:** Definizione metodo mrDMDc.

Come già detto, la modifica principale è stata quella di sostituire la parte di ricostruzione DMD direttamente con il DMDc della libreria PyDMD [14], dico direttamente perchè il DMDc a differenza del DMD, ritorna direttamente la matrice dei dati ricostruiti, mentre il DMD ritorna mu e phi, rispettivamente gli autovalori e i modi dinamici che poi vengono utilizzati, come visto all'interno del codice 2.4, per la ricostruzione della matrice dei dati. Qui di seguito lascio un estratto del DMDc (PyDMD), in particolare la funzione reconstructed_data() (Codice 2.10):

```python
def reconstructed_data(self, control_input=None):
        """
        Return the reconstructed data, computed using the `control_input`
        argument. If the `control_input` is not passed, the original input (in
        the `fit` method) is used. The input dimension has to be consistent
        with the dynamics.

        :param numpy.ndarray control_input: the input control matrix.
        :return: the matrix that contains the reconstructed snapshots.
        :rtype: numpy.ndarray
        """
        controlin = (
            np.asarray(control_input)
            if control_input is not None
            else self._controlin
        )

        if controlin.shape[-1] != self.dynamics.shape[-1] - 1:
            raise RuntimeError(
                "The number of control inputs and the number of snapshots to "
                "reconstruct has to be the same"
            )

        eigs = np.power(
            self.eigs, self.dmd_time["dt"] // self.original_time["dt"]
        )
        A = np.linalg.multi_dot(
            [self.modes, np.diag(eigs), np.linalg.pinv(self.modes)]
        )

        data = [self.snapshots[:, 0]]
        expected_shape = data[0].shape

        for i, u in enumerate(controlin.T):
            arr = A.dot(data[i]) + self._B.dot(u)
            if arr.shape != expected_shape:
                raise ValueError(
                    f"Invalid shape: expected {expected_shape}, got {arr.shape}"
                )
            data.append(arr)

        data = np.array(data).T
        return data
```

**Codice 2.10:** Funzione reconstructed_data() per la ricostruzione del dato con il metodo DMDc.

Questa versione è stata modificata in modo tale da tornare i valori di "mu" e "phi", rispettivamente in questo codice identificati come "eigs" e "self.modes" (Codice 2.11):

```python
def reconstructed_data(self, control_input=None):
    """
    Return the reconstructed data, computed using the `control_input`
    argument. If the `control_input` is not passed, the original input (in
    the `fit` method) is used. The input dimension has to be consistent
    with the dynamics.

    :param numpy.ndarray control_input: the input control matrix.
    :return: the matrix that contains the reconstructed snapshots.
    :rtype: numpy.ndarray
    """
    controlin = (
        np.asarray(control_input)
        if control_input is not None
        else self._controlin
    )

    if controlin.shape[-1] != self.dynamics.shape[-1] - 1:
        raise RuntimeError(
            "The number of control inputs and the number of snapshots to "
            "reconstruct has to be the same"
        )

    eigs = np.power(
        self.eigs, self.dmd_time["dt"] // self.original_time["dt"]
    )
    A = np.linalg.multi_dot(
        [self.modes, np.diag(eigs), np.linalg.pinv(self.modes)]
    )

    data = [self.snapshots[:, 0]]
    expected_shape = data[0].shape

    for i, u in enumerate(controlin.T):
        arr = A.dot(data[i]) + self._B.dot(u)
        if arr.shape != expected_shape:
            raise ValueError(
                f"Invalid shape: expected {expected_shape}, got {arr.shape}"
            )
        data.append(arr)

    data = np.array(data).T
    return data, eigs, self.modes
```

**Codice 2.11:** Funzione reconstructed_data() modificata.

Eseguiamo il nuovo metodo (Codice 2.12)

```
nodes = mrdmdc(D,U)
```

**Codice 2.12:** Salviamo i nodi della funzione mrDMDc.

A questo punto, abbiamo riscontrato un problema nell'esecuzione della funzione "stitch" (Codice 2.6) , poichè andando a inizializzare la variabile D_mrdmd che ci servirà proprio per ricostruire la matrice finale, risulta un errore riguardanti le dimensioni utilizzate (Codice 2.13):

```
D_mrdmd = dot(*stitch(nodes, 0))
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_12860\2717936194.py in <module>
----> 1 D_mrdmd = dot(*stitch(nodes, 0))

<__array_function__ internals> in dot(*args, **kwargs)

ValueError: shapes (40,7) and (0,7160) not aligned: 7 (dim 1) != 0 (dim 0)
```

**Codice 2.13:** Errore nell'inizializzazione della variabile D_mrdmd.

## 2.3 Dati Sintetici

Oltre alla generazione casuale dei dati, per approfondire il comportamento dei vari metodi, abbiamo creato dei dati sintetici tramite uno script in Matlab. La creazione di questi dati ci è stata utile per testare i vari metodi sapendo in maniera deterministica i dati forniti in ingresso, in particolare abbiamo creato sia un dataset composto da una matrice con solo elementi reali e un dataset composto con una matrice con solo elementi complessi.

Lo script, disponibile in [15], è il seguente (Codice 2.14):

```
realpole=false;
To=0;
Tend=20;
samples=1600;
Ts=Tend/samples;
n=10;
```

```
if realpole==true
    poles=[-3000 -2000 -250 -200 -50 -40 -0.1 -0.2 -0.01 -0.02];
else
    poles=[-8+80j -8-80j -2+20j -2-20j -0.5+10j -0.5-10j -0.2+2j -0.2-2j -0.01+0.5j -0.01-0.5j];
end
% A=diag(poles);
% B=zeros(10,1);
% C=zeros(1,10);
D=0;
[A,B,C,D] = zp2ss([],poles,1)
csys=ss(A,B,C,D);
```

```
opt = c2dOptions('Method','Tustin');
dsys = c2d(csys,Ts,opt);
ev=eig(dsys.A)
```

```
[y,tOut,x]=initial(dsys,10*ones(10,1),20);
zpole=pole(dsys);
zplane(zpole)
figure, plot(tOut,x)
if realpole==true
    writematrix(x','real_eig_timeseries.csv')
else
    writematrix(x','complex_eig_timeseries.csv')
end
```

**Codice 2.14:** Script in Matlab per la generazione di dataset sintetici.

La creazione del dataset con autovalori reali o complessi dipende dalla variabile "realpole" che se impostata su "false" genera un dataset con elementi complessi, mentre se impostata su "true" genera un dataset con elementi reali. Qui di seguito vediamo nel dettaglio le matrici generate e la rappresentazione grafica della loro evoluzione nel tempo (Figure 2.9 e 2.10).

```
A = 10×10
10^3 ×
   -5.0000   -2.4495         0         0         0         0         0         0         0         0
    2.4495         0         0         0         0         0         0         0         0         0
         0    0.0000   -0.4500   -0.2236         0         0         0         0         0         0
         0         0    0.2236         0         0         0         0         0         0         0
         0         0         0    0.0000   -0.0900   -0.0447         0         0         0         0
         0         0         0         0    0.0447         0         0         0         0         0
         0         0         0         0         0    0.0000   -0.0003   -0.0001         0         0
         0         0         0         0         0         0    0.0001         0         0         0
         0         0         0         0         0         0         0    0.0071   -0.0000   -0.0000
         0         0         0         0         0         0         0         0    0.0000         0

B = 10×1
     1
     0
     0
     0
     0
     0
     0
     0
     0
     0

C = 1×10
         0         0         0         0         0         0         0         0         0   70.7107

D = 0
ev = 10×1
   -0.8987
   -0.8519
   -0.2195
   -0.1111
    0.5238
    0.6000
    0.9975
    0.9988
    0.9998
    0.9999
```

**Figura 2.9:** Rappresentazione della matrice reale.

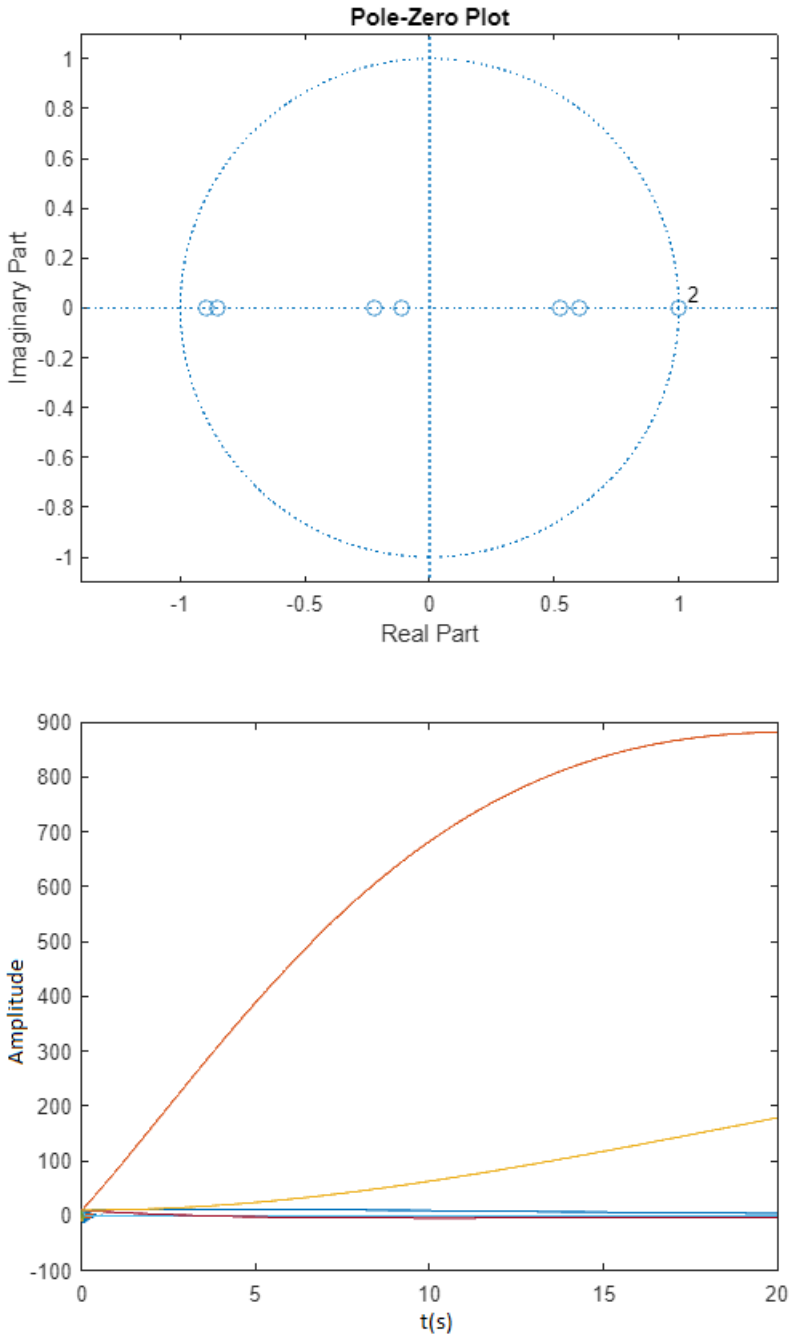



**Figura 2.10:** Rappresentazione grafica dell'evoluzione nel tempo delle variabili.

Come possiamo notare dal grafico, avendo valori unicamente reali, le funzioni si mantengono costanti nel tempo (a meno di un piccolo transitorio iniziale). Questa costanza la possiamo notare andando a compilare, ad esempio, l'mrDMD usando come dataset proprio la matrice appena generata, noteremo infatti che il risultato delle varie iterazioni non varierà e sarà sempre uguale al dataset iniziale (Figure 2.11 e 2.12).

```
A = 10×10
   -16.0000  -80.3990         0         0         0         0         0         0         0         0
    80.3990         0         0         0         0         0         0         0         0         0
          0    0.0124   -4.0000  -20.0998         0         0         0         0         0         0
          0         0   20.0998         0         0         0         0         0         0         0
          0         0         0    0.0498   -1.0000  -10.0125         0         0         0         0
          0         0         0         0   10.0125         0         0         0         0         0
          0         0         0         0         0    0.0999   -0.4000   -2.0100         0         0
          0         0         0         0         0         0    2.0100         0         0         0
          0         0         0         0         0         0         0    0.4975   -0.0200   -0.5001
          0         0         0         0         0         0         0         0    0.5001         0

B = 10×1
     1
     0
     0
     0
     0
     0
     0
     0
     0
     0

C = 1×10
          0         0         0         0         0         0         0         0         0    1.9996

D = 0
ev = 10×1 complex
   0.5527 + 0.7394i
   0.5527 - 0.7394i
   0.9457 + 0.2402i
   0.9457 - 0.2402i
   0.9861 + 0.1237i
   0.9861 - 0.1237i
   0.9972 + 0.0249i
   0.9972 - 0.0249i
   0.9999 + 0.0062i
   0.9999 - 0.0062i
```

**Figura 2.11:** Rappresentazione della matrice complessa.

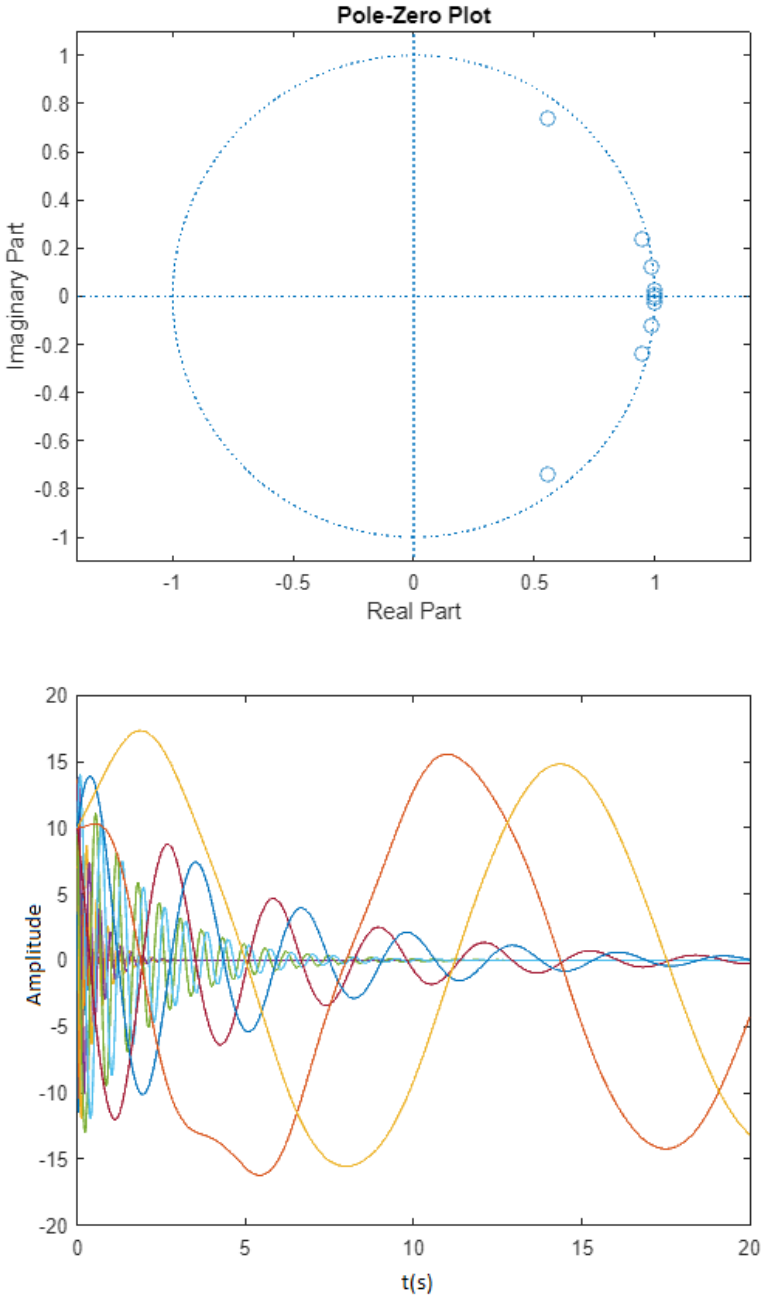



**Figura 2.12:** Rappresentazione grafica dell'evoluzione nel tempo delle variabili.

In questo caso invece i segnali oscillano nel tempo, essendo composti di una parte immaginaria. Andando ad eseguire l'mrDMD con questo dataset, riusciremmo ad apprezzarne il risultato poichè avremmo un'evoluzione nel tempo.

## 2.4 Dataset Industriale

Una volta generati e testati dei dati sintetici, bisognava fare lo step successivo, ovvero utilizzare un dataset industriale reale. Per l'estrazione di questo dataset ci siamo basati sul Sulfur Recovery Unit (SRU), ovvero un processo per estrarre dati da processi industriali.

**Sulfur Recovery Unit (SRU)** L'articolo da cui prenderemo le informazioni sarà il seguente [16]. In particolare in questo articolo sono stati sviluppati due SS (Soft Sensor) per una raffineria in Sicilia, Italia. In particolare, è stato utilizzato per l'estrazione di zolfo dai gas acidi, questo perchè il recupero di zolfo in questi tipi di processi è un processo chiave perchè permette di rimuovere gli inquinanti ambientali dai vapori generati dal gas acido che vengono rilasciati nell'atmosfera. L'SRU preso in considerazione è composto da quattro sotto-unità identiche (ovvero linee di zolfo) che lavorano in parallelo, ognuna in grado di estrarre zolfo dai gas acidi ad una velocità di 100 tonnellate al giorno, qui di seguito in figura uno schema a blocchi che ne descrive il processo (Figura 2.13).

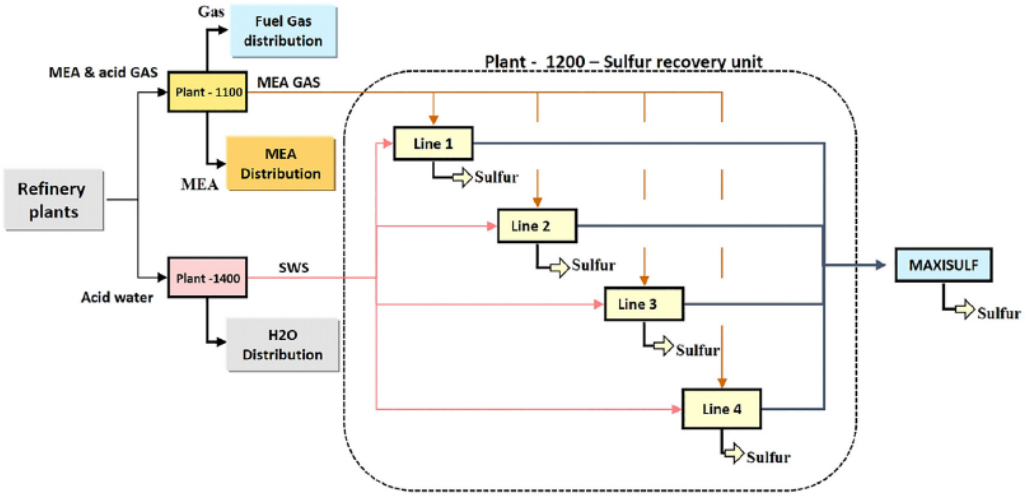



**Figura 2.13:** Schema a blocchi del sistema di recupero dello zolfo con quattro linee, utilizzato in una raffineria situata in Sicilia.

Ogni linea di zolfo ricevere due tipi di gas acidi in ingresso: gas MEA, proveniente dalle centrali di lavaggio dei gas, e gas, SWS, ricco di $H_2S$ e $NH_3$, proveniente dalla centrale di stripping dell'acqua acida. L'estrazione dello zolfo avviene in reattori, dove l'$H_2S$ viene sottoposto a una reazione di ossidazione parziale con l'aria. Il gas residuo viene alimentato nella centrale di Maxisulfur per una fase di estrazione diversa. I gas che fuoriescono (gas di coda) dall'SRU contiene $H_2S$ e $SO_2$ residui, che devono essere controllati. L'aria, che fornisce ossigeno per la reazione, è essenziale per la conversione dei gas acidi e responsabile della composizione del gas di coda.
Il processo è difficile da controllare perchè un flusso d'aria eccessivo aumenta la concentrazione di $SO_2$ rispetto a $H_2S$, mentre un flusso d'aria basso fa l'opposto. Attualmente, viene utilizzato un analizzatore online per misurare la concentrazione di $H_2S$ e $SO_2$ nel gas che fuoriesce da ciascuna linea di zolfo. Misura anche il valore $[H_2S]$ - $2[SO_2]$ (dove le parentesi indicano la concentrazione) per monitorare le prestazioni del processo di conversione e controllare il rapporto aria-alimentazione nell'SRU per un'eccellente estrazione dello zolfo. Il valore desiderato di $[H_2S]$ - $2[SO_2]$ è zero, il che indica l'assenza di questi inquinanti nel gas che fuoriesce o che i reagenti sono in proporzione stechiometrica, il che è ottimale per la rimozione totale dello zolfo.
Il controllo viene migliorato da un algoritmo a ciclo chiuso che regola un

flusso d'aria aggiuntivo *(AIR MEA 2)* in base all'analisi della composizione del gas di scarico. I gas acidi danneggiano i sensori, che richiedono una manutenzione frequente. Quando l'analizzatore è offline per la manutenzione, questo circuito di controllo non può funzionare e le prestazioni del"SRU peggiorano. Pertanto, è necessaria una SS per sostituire l'analizzatore offline. Inoltre la SS fornisce una stima ridondante delle variabili di interesse per scopi di rilevamento dei guasti.
L'SRU è spesso utilizzato come punto di riferimento per la progettazione di sistemi di stima. La maggior parte dei risultati disponibili in letteratura si applica alla linea 4 dello zolfo. In questo articolo, inizialmente sono stati considerati i dati acquisiti dalla linea 2. Per analizzare ulteriormente l'applicazione della metodologia proposta, vengono valutati anche i dati relativi alla linea 4 dello zolfo, confrontando i risultati ottenuti con alcune architetture alternative disponibili in letteratura. Per la riproducibilità degli approcci proposti, i dati di input e output normalizzati, utilizzati per lo sviluppo del SS della linea 2 dello zolfo, sono forniti nel materiale supplementare, mentre i dati relativi alla linea 4 sono disponibili in [17]. Uno schema semplificato relativo a una singola linea di processo dell'SRU (Figura 2.14).

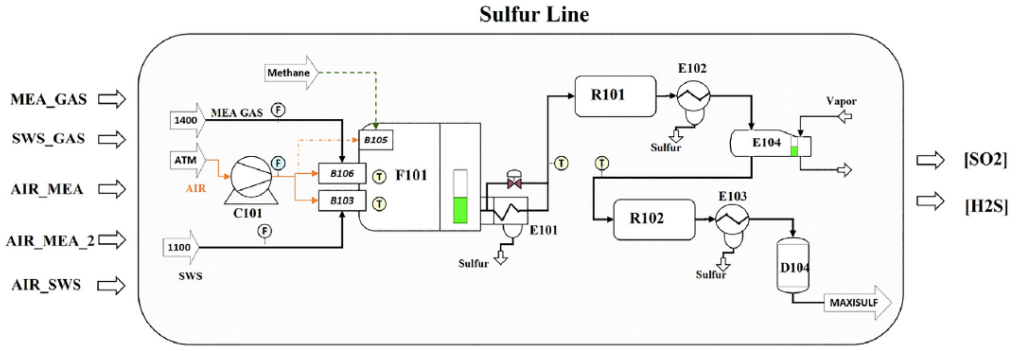



**Figura 2.14:** Descrizione schematica di una linea di zolfo in cui vengono considerati cinque ingressi e due uscite.

I cinque ingressi sono i gas MEA e SWS, i rispettivi flussi d'aria *(AIR MEA)* e *(AIR SWS)* e l'ulteriore flusso d'aria di ingresso *(AIR MEA 2)*, che viene determinato utilizzando il sistema di controllo a ciclo chiuso basato sull'analisi del gas di scarto.
Nella linea considerata, vengono sviluppati due SS, uno per il $SO_2$ e l'altro per l'$H_2S$. Oltre alla precisione della previsione, viene valutato anche $[H_2S]$

- 2[$SO_2$]. La presenza di picchi nelle variabili rappresenta una condizione critica che, se prevista corretamente, consente un miglioramento del controllo del processo. Sulla base dei suggerimenti forniti dagli esperti del settore, la soglia critica nell'analisi dei picchi è il 30% del valore medio.

# Capitolo 3

# Risultati

In quest'ultimo capitolo andremo ad analizzare i risultati ottenuti con i vari metodi utilizzando i dataset precedentemente introdotti, in modo da visualizzarli e commentarli.

## 3.1 mrDMD

Per ottenere i risultati che andremo a vedere tra poco, basterà dare in pasto al metodo mrDMD il dataset desiderato. Nel github dedicato a questo lavoro di tesi [15] è presente tutto il materiale utile a replicare i risultati presentati, compresi i dataset utilizzati. Seguendo queste istruzioni, siamo in grado di scegliere quale dataset utilizzare.

**Dati Sintetici (X,Y)** Per l'utilizzo dei dati sintetici, in particolare i dati sintetici con autovalori complessi, possiamo fare riferimento al codice 3.1.

```
### Parametri per dataset real_eig_timeseries.mat e complex_eig_timeseries.mat
x = np.linspace(0, 10, 10)
t = np.linspace(0, 1600, 1600)

### Parametri per dataset XU_DMDc.mat
#x = np.linspace(0, 40, 40)
#t = np.linspace(0, 7160, 7160)

### Scelta del dataset, se a valori complessi o reali
#D_mat = scipy.io.loadmat('real_eig_timeseries.mat')
D_mat = scipy.io.loadmat('complex_eig_timeseries.mat')
#D_mat = scipy.io.loadmat('XU_DMDc.mat')

### Trasformo il formato .mat in un Numpy Array

### Usare in caso di Dataset real/complex_eig_timeseries.mat
D_mat_list = [[element for element in upperElement] for upperElement in D_mat['xt']]
D = np.array(D_mat_list)

### Usare in caso di Dataset XU_DMDc.mat
#D_mat_list = [[element for element in upperElement] for upperElement in D_mat['X']]
#U_mat_list = [[element for element in upperElement] for upperElement in D_mat['U']]
#D = np.array(D_mat_list)
#U = np.array(U_mat_list)

### extract input-output matrices
X = D[:,:-1]
Y = D[:,1:]   #Y=X'

### Usare in caso di Dataset XU_DMDc.mat
#U = U[:,:]
```

**Codice 3.1:** Scelta del dataset complex_eig_timeseries.mat.

La rappresentazione grafica del dataset è la seguente (Figura 3.1):

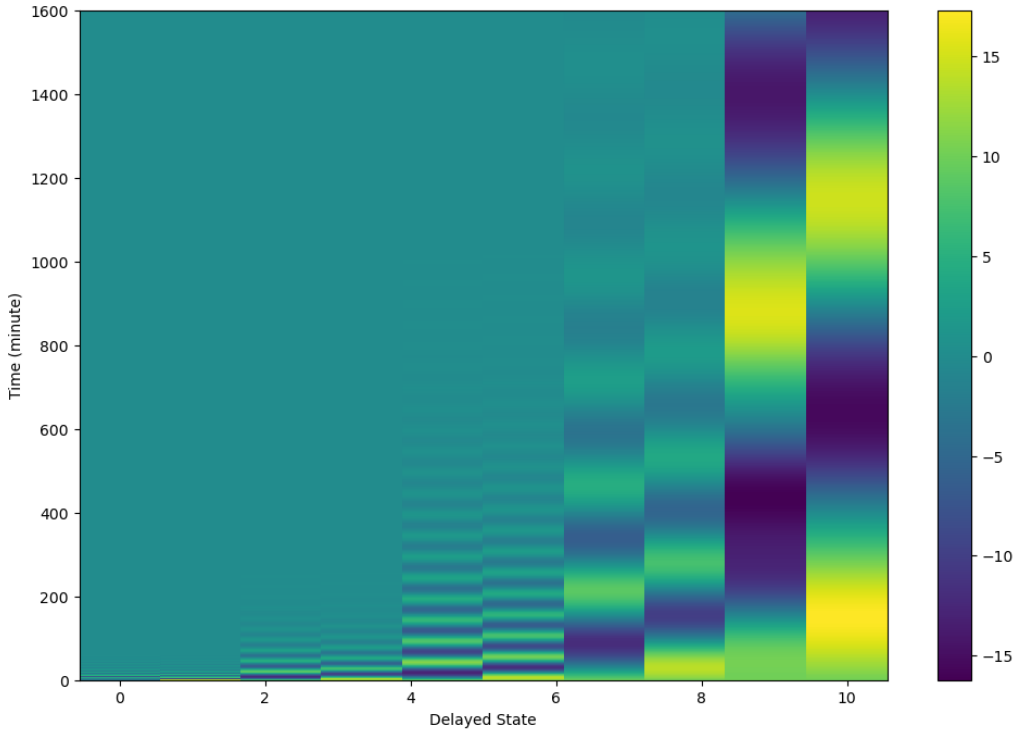



**Figura 3.1:** Rappresentazione grafica dataset complex_eig_timeseries.mat

Adesso, andando ad eseguire il codice come fatto in precedenza (Paragrafo: mrDMD, Cap.2), otterremo questo risultato (Figure 3.2 - 3.7):

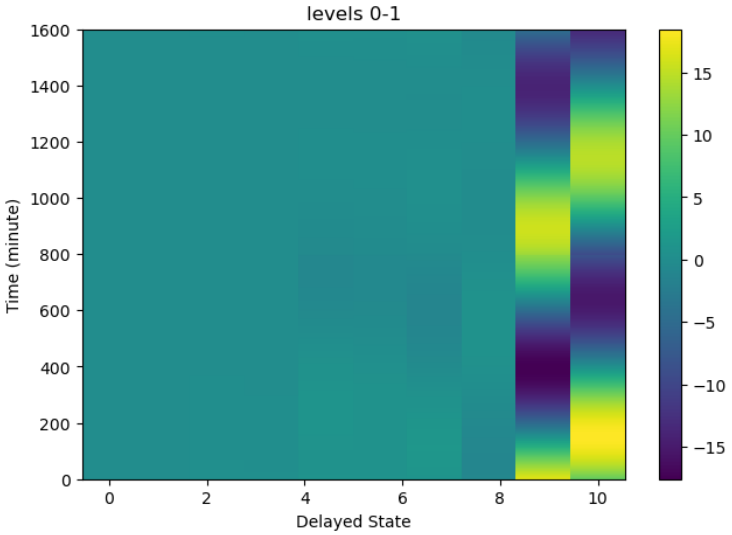



**Figura 3.2:** Risultato del primo livello di iterazione.

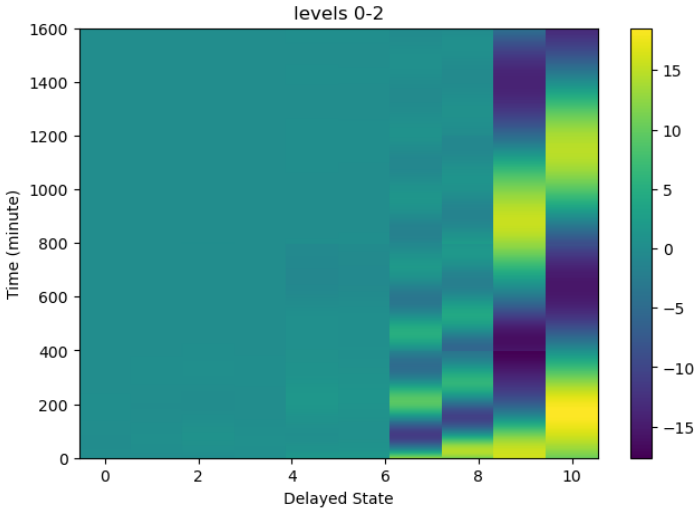



**Figura 3.3:** Risultato del secondo livello di iterazione.

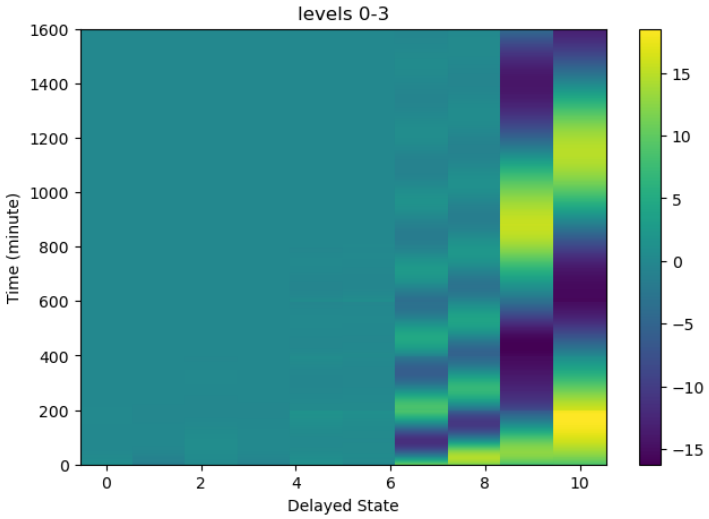



**Figura 3.4:** Risultato del terzo livello di iterazione.

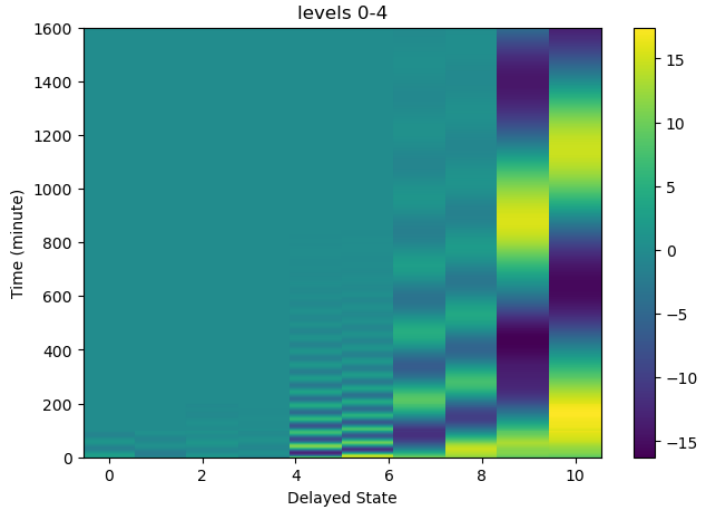



**Figura 3.5:** Risultato del quarto livello di iterazione.

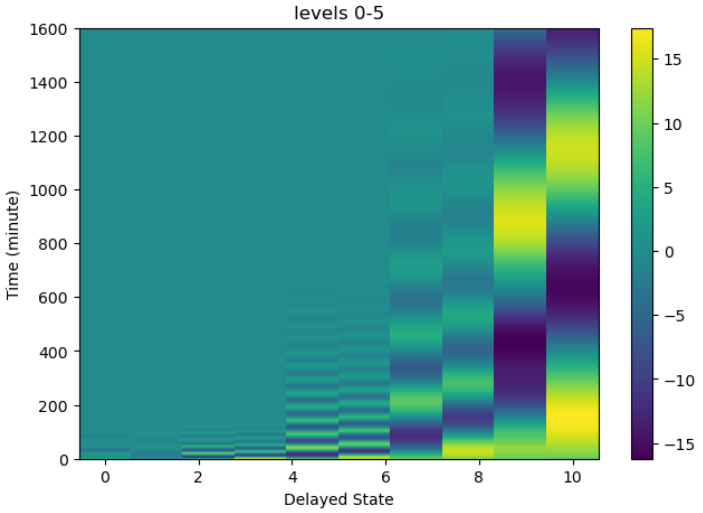



**Figura 3.6:** Risultato del quinto livello di iterazione.

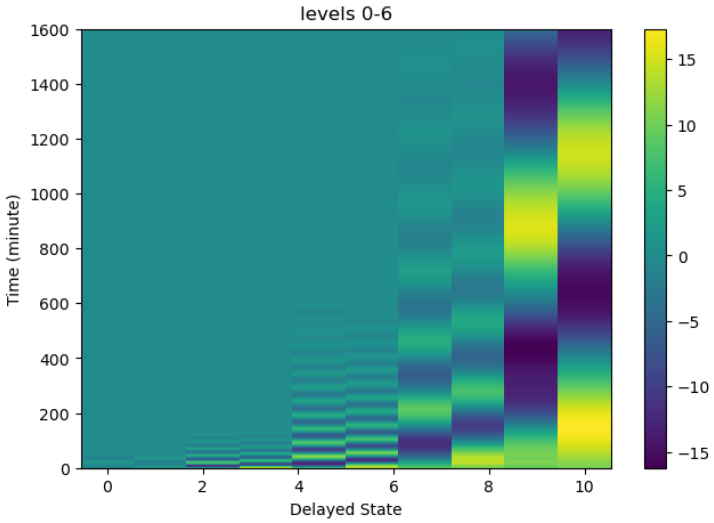



**Figura 3.7:** Risultato del sesto livello di iterazione.

Dal punto di vista grafico è difficile notare l'effettivo risultato, quindi utilizziamo un altro tipo di valutazione, ovvero come già fatto nel Codice 2.7 (punto 2), una volta create le matrici unidimensionali (vettori) del dataset di partenza e della matrice ricostruita, applichiamo una differenza tra i due vettori (Codice 3.2):

```
plt.figure()
plt.plot(t, D0, 'b', label='Misura')
plt.plot(t, D_mrdmd0.real, 'g', label='DMD')
plt.legend()
plt.show()

plt.figure()
error=np.array(D0) - np.array(D_mrdmd0)
plt.plot(t, error.real, 'b', label='Diff')
plt.legend()
plt.show()
```

**Codice 3.2:** Plotting delle misure.

e stampiamo a schermo il risultato delle misure e della differenza (Figura 3.8):

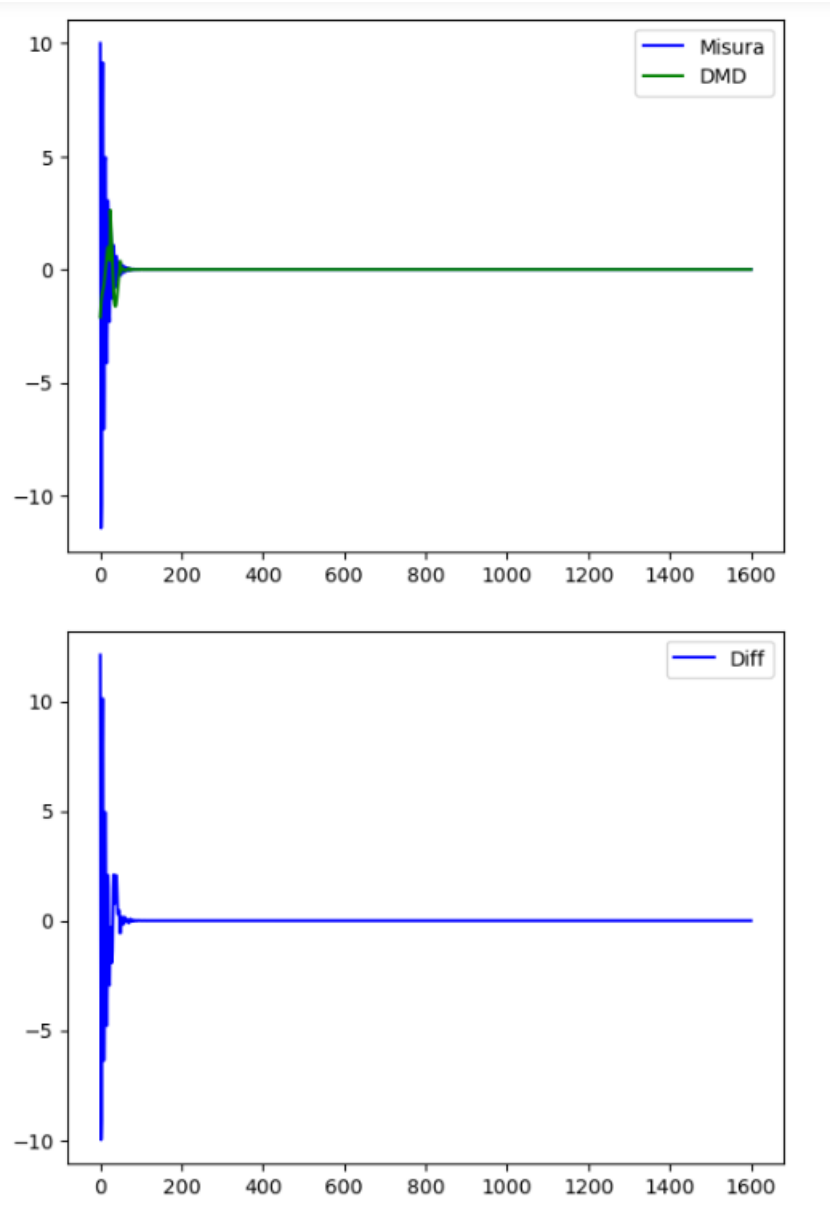



**Figura 3.8:** 1. Rappresentazione della misura del dataset complex_eig_timeseries.mat e della matrice ricostruita. 2. Differenza tra le due misure.

Adesso vediamo cosa succede se inseriamo un dataset con solo elementi reali. Innanzitutto andiamo a scegliere il dataset (Codice 3.3):

```
### Parametri per dataset real_eig_timeseries.mat e complex_eig_timeseries.mat
x = np.linspace(0, 10, 10)
t = np.linspace(0, 1600, 1600)

### Parametri per dataset XU_DMDc.mat
#x = np.linspace(0, 40, 40)
#t = np.linspace(0, 7160, 7160)

### Scelta del dataset, se a valori complessi o reali
D_mat = scipy.io.loadmat('real_eig_timeseries.mat')
#D_mat = scipy.io.loadmat('complex_eig_timeseries.mat')
#D_mat = scipy.io.loadmat('XU_DMDc.mat')

### Trasformo il formato .mat in un Numpy Array

### Usare in caso di Dataset real/complex_eig_timeseries.mat
D_mat_list = [[element for element in upperElement] for upperElement in D_mat['xt']]
D = np.array(D_mat_list)

### Usare in caso di Dataset XU_DMDc.mat
#D_mat_list = [[element for element in upperElement] for upperElement in D_mat['X']]
#U_mat_list = [[element for element in upperElement] for upperElement in D_mat['U']]
#D = np.array(D_mat_list)
#U = np.array(U_mat_list)

### extract input-output matrices
X = D[:,:-1]
Y = D[:,1:]   #Y=X'

### Usare in caso di Dataset XU_DMDc.mat
#U = U[:,:]
```

**Codice 3.3:** Scelta del dataset real_eig_timeseries.mat.

Vediamo qual è la rappresentazione grafica del dataset (Figura 3.9):

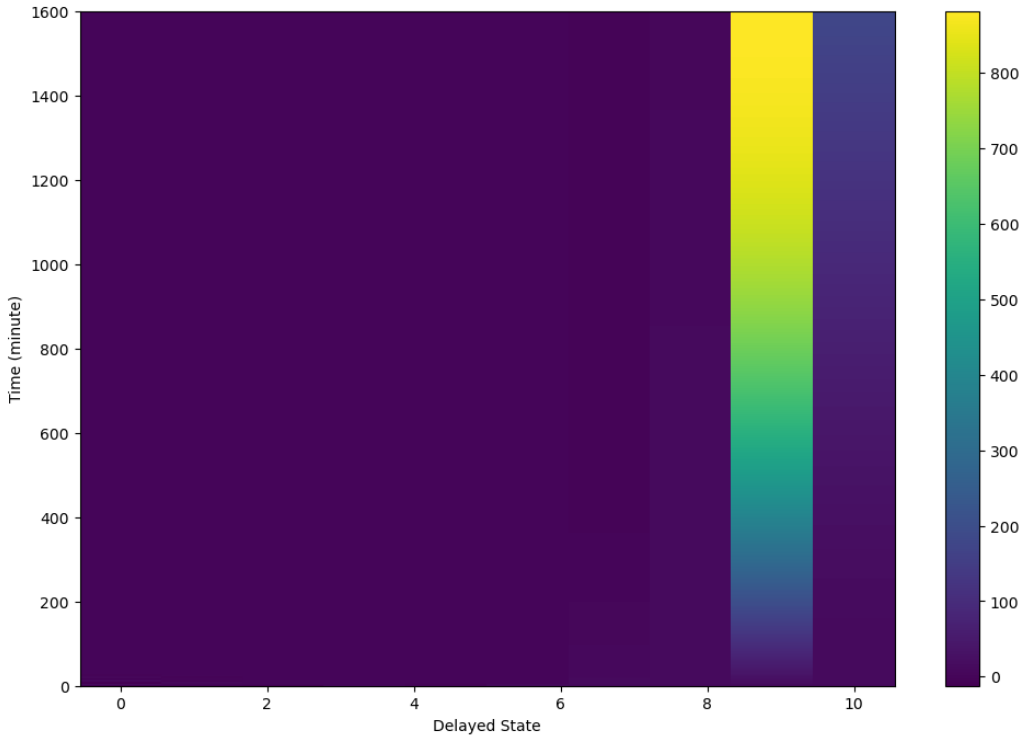



**Figura 3.9:** Rappresentazione grafica dataset real_eig_timeseries.mat

Andando ad eseguire il metodo su questo dataset, composto unicamente da variabili reali, ci aspettiamo che, non avendo evoluzione nel tempo dei dati, il risultato di ogni iterazione sarà invariato.

Adesso, andando ad eseguire il codice come fatto in precedenza (Paragrafo: mrDMD, Cap.2), otterremo questo risultato (Figure 3.10 - 3.15):

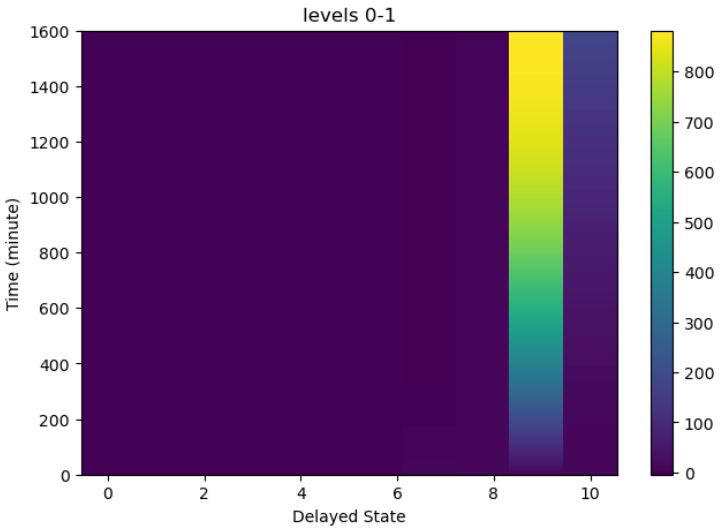



**Figura 3.10:** Risultato del primo livello di iterazione.

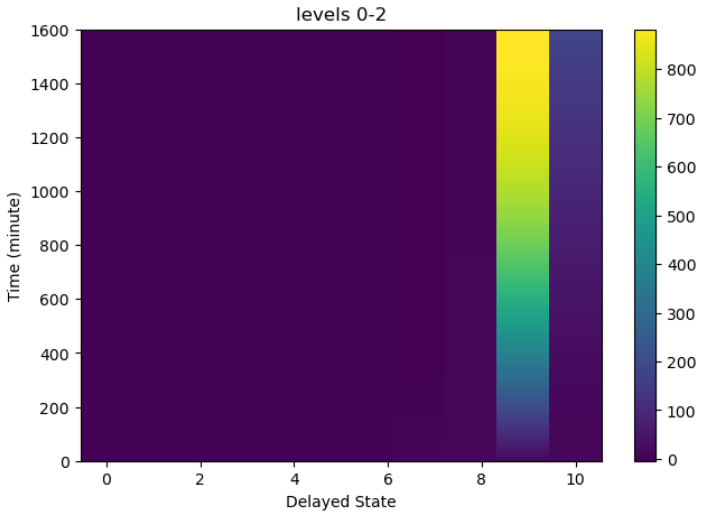



**Figura 3.11:** Risultato del secondo livello di iterazione.

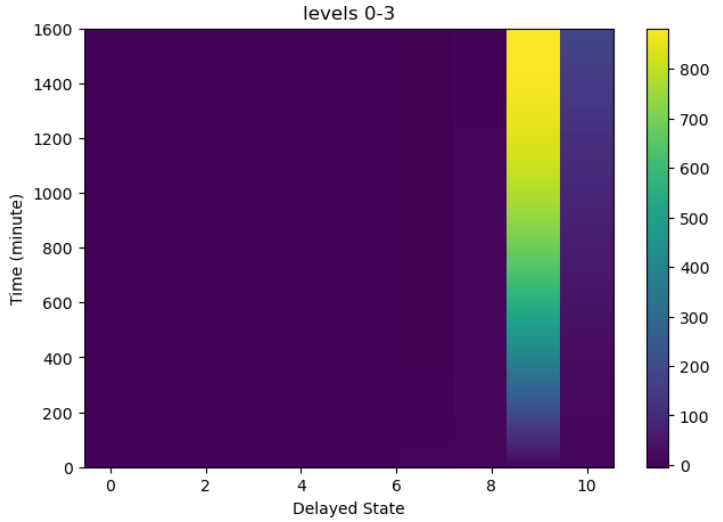



**Figura 3.12:** Risultato del terzo livello di iterazione.

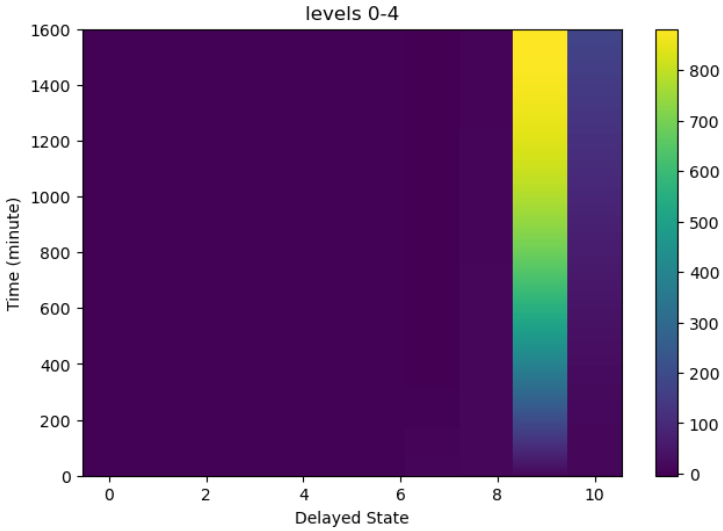



**Figura 3.13:** Risultato del quarto livello di iterazione.

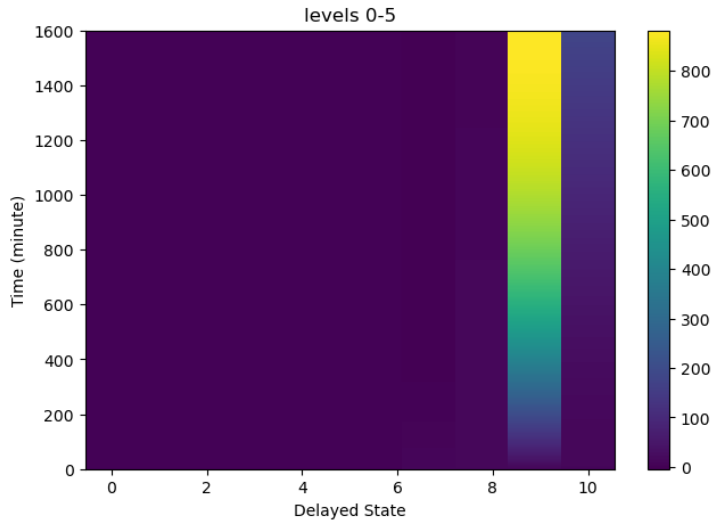



**Figura 3.14:** Risultato del quinto livello di iterazione.

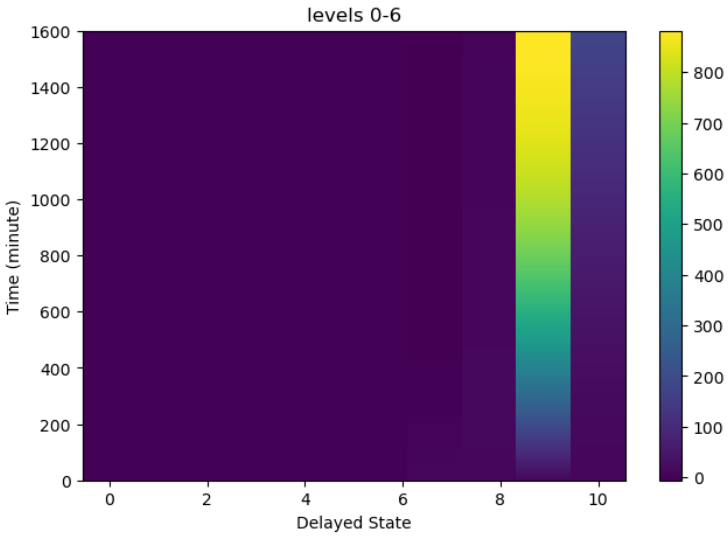



**Figura 3.15:** Risultato del sesto livello di iterazione.

Ed ecco che il risultato era proprio quello aspettato, dovuto proprio alla non-evoluzione nel tempo delle variabili che compongono il dataset. Adesso vediamo la differenza come già fatto in precedenza utilizzando il Codice 3.4 e stampiamo a schermo i risultati (Figura 3.16):

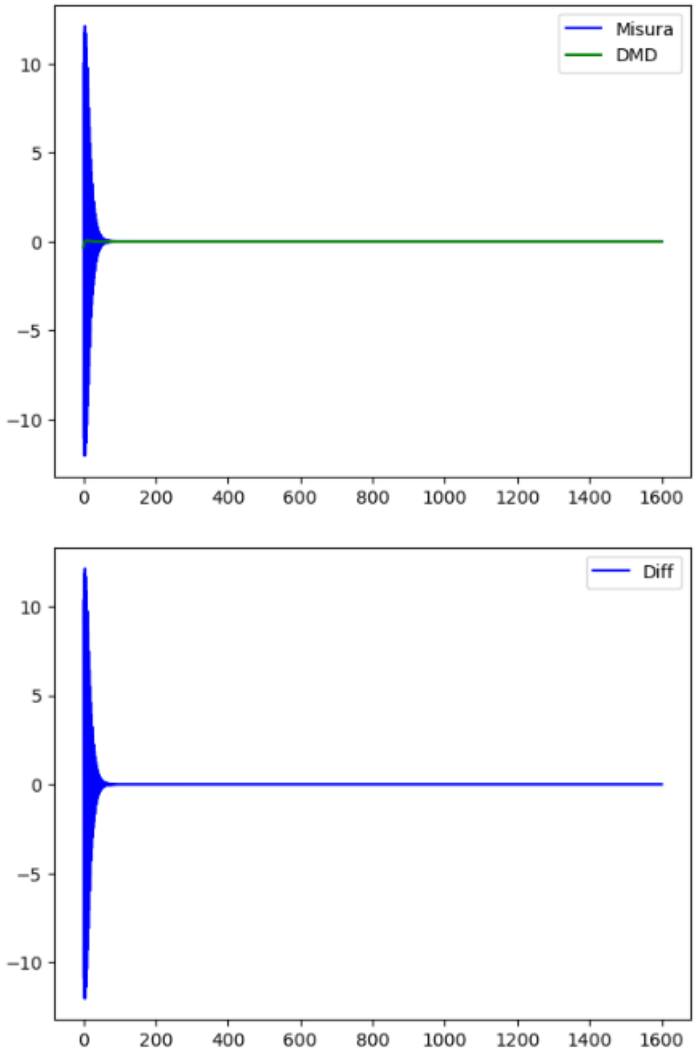



**Figura 3.16:** 1. Rappresentazione della misura del dataset real_eig_timeseries.mat e della matrice ricostruita. 2. Differenza tra le due misure.

**Dataset Industriale** Dopo aver analizzato i risultati ottenuti con dei dati sintetici, creati tramite uno script in Matlab, andiamo ad analizzare i risultati ottenuti dal dataset reale di cui abbiamo parlato nel paragrafo Sulfur Recovery Unit (SRU) nel capitolo 2.
Come fatto precedentemente, scegliamo il dataset da utilizzare (Codice 3.4):

```
### Parametri per dataset real_eig_timeseries.mat e complex_eig_timeseries.mat
#x = np.linspace(0, 10, 10)
#t = np.linspace(0, 1600, 1600)

### Parametri per dataset XU_DMDc.mat
x = np.linspace(0, 40, 40)
t = np.linspace(0, 7160, 7160)

### Scelta del dataset, se a valori complessi o reali
#D_mat = scipy.io.loadmat('real_eig_timeseries.mat')
#D_mat = scipy.io.loadmat('complex_eig_timeseries.mat')
D_mat = scipy.io.loadmat('XU_DMDc.mat')

### Trasformo il formato .mat in un Numpy Array

### Usare in caso di Dataset real/complex_eig_timeseries.mat
#D_mat_list = [[element for element in upperElement] for upperElement in D_mat['xt']]
#D = np.array(D_mat_list)

### Usare in caso di Dataset XU_DMDc.mat
D_mat_list = [[element for element in upperElement] for upperElement in D_mat['X']]
#U_mat_list = [[element for element in upperElement] for upperElement in D_mat['U']]
D = np.array(D_mat_list)
#U = np.array(U_mat_list)

### extract input-output matrices
X = D[:,:-1]
Y = D[:,1:]   #Y=X'

### Usare in caso di Dataset XU_DMDc.mat
#U = U[:,:]
```

**Codice 3.4:** Scelta del dataset XU_DMDc.mat.

Per il momento omettiamo la matrice U formata dalle variabili di controllo poichè il metodo mrDMD non è in grado di gestirla. Vediamo in ogni caso il risultato che potremmo ottenere, partendo dalla rappresentazione grafica del dataset (Figura 3.17):

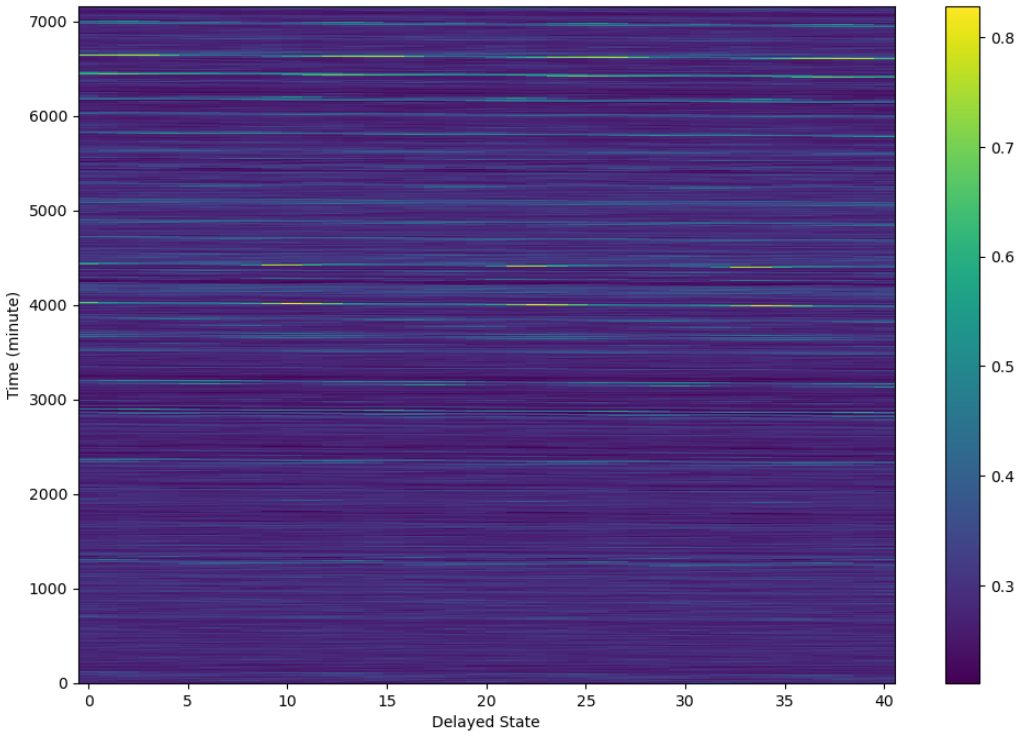



**Figura 3.17:** Rappresentazione grafica dataset XU_DMDc.mat

Notiamo delle linee molto fitte tra loro, questo è dovuto dalla quantità molto elevata di snapshot (7160), rispetto al numero di snapshot analizzati precedentemente (1600). Eseguendo il metodo otterremo (Figure 3.18 - 3.23):

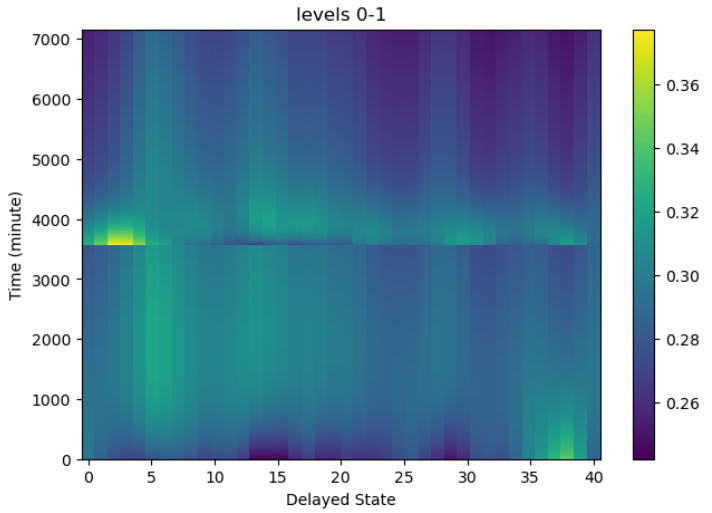



**Figura 3.18:** Risultato del primo livello di iterazione.

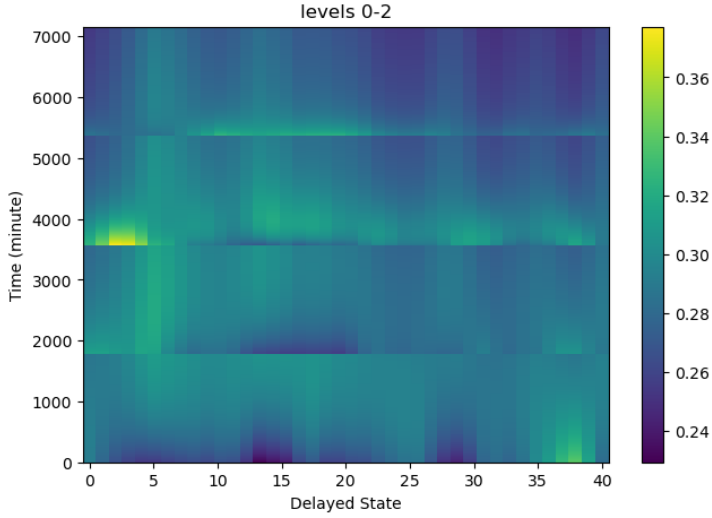



**Figura 3.19:** Risultato del secondo livello di iterazione.

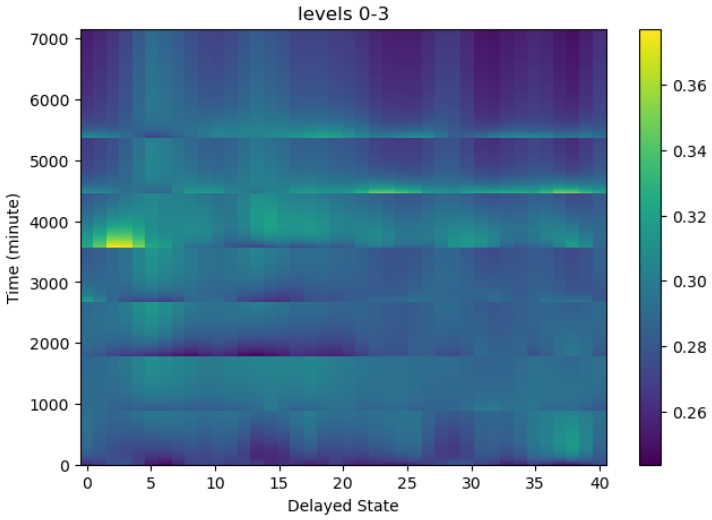



**Figura 3.20:** Risultato del terzo livello di iterazione.

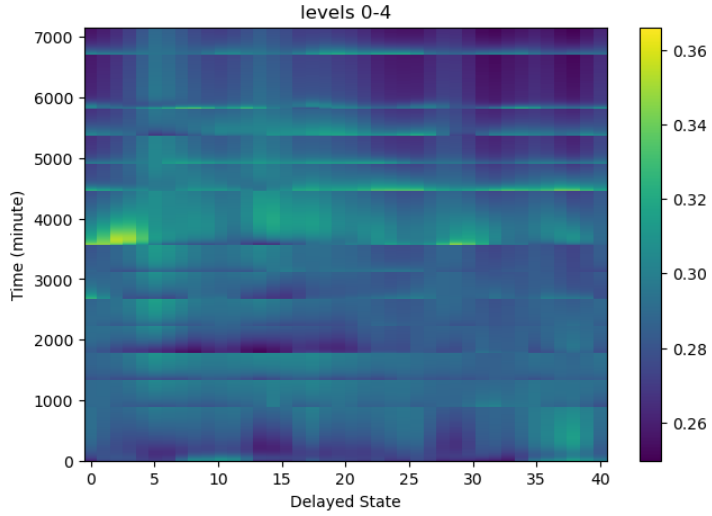



**Figura 3.21:** Risultato del quarto livello di iterazione.

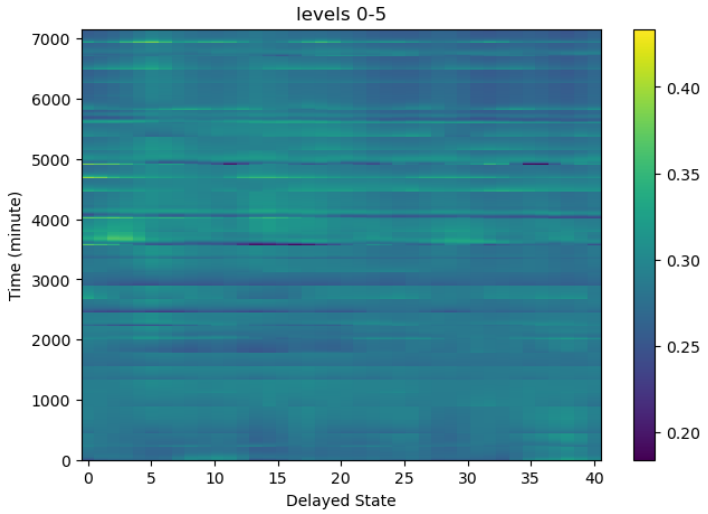



**Figura 3.22:** Risultato del quinto livello di iterazione.

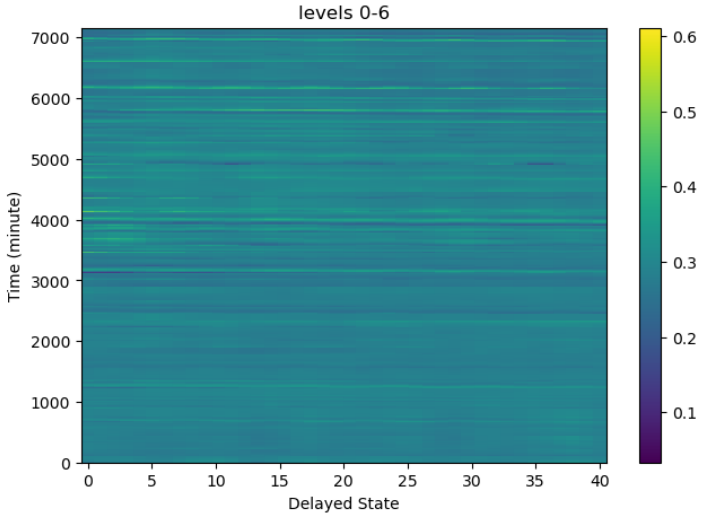



**Figura 3.23:** Risultato del sesto livello di iterazione.

Anche per questo risultato plottiamo i vettori delle misure reali (SRU) e della ricostruzione, e stampiamo a schermo il risultato della differenza (Figura 3.24):

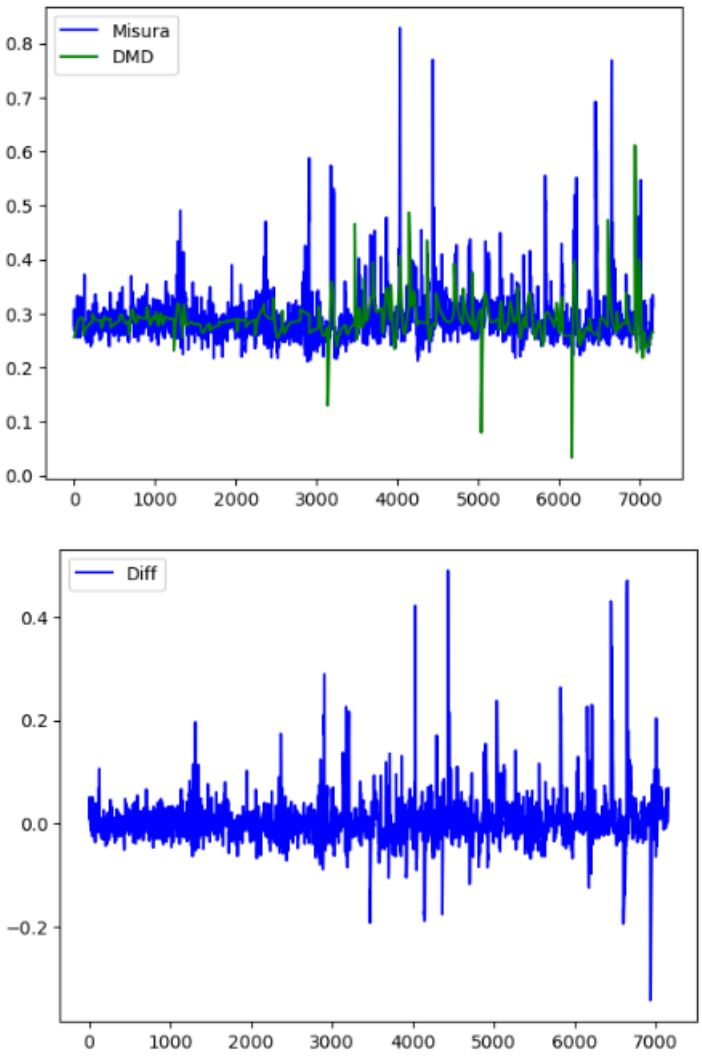



**Figura 3.24:** 1. Rappresentazione della misura del dataset SRU e della matrice ricostruita. 2. Differenza tra le due misure.

# Conclusioni

Il presente elaborato si occupa dello studio e dell'implementazione dei metodi data-driven in ambito industriale. Inizialmente sono stati analizzati gli aspetti metodologici dei vari metodi tra cui il DMD, DMDc e mrDMD. In seguito ci siamo soffermati sull'implementazione in Python, descrivendo le varie funzioni che li compongono.
L'obiettivo di questo elaborato di tesi è stato tentare di implementare un nuovo metodo data-driven chiamato mrDMDc, che fosse in grado di decomporre una matrice di dati pilotata da una matrice di controllo.
Le metodologia utilizzata consiste nella sostituzione del metodo DMD all'interno del metodo mrDMD con il DMDc.
Nonostante il lavoro svolto, abbiamo riscontrato delle problematiche riguardanti le dimensioni utilizzate all'interno della funzione "stitch". In particolare, l'elemento critico è stato la modifica della funzione DMDc, poichè avendo bisogno degli autovalori e dei modi dinamici, che prima erano generati dal DMD, generandoli con il DMDc, sono stati riscontrati errori con la gestione degli stessi.
Gli sviluppi futuri riguarderanno la modifica dell'intera libreria PyDMD e la modifica della funzione "stitch" per unire tutti i livelli di ricorsione del mrDMD, al fine di continuare lo sviluppo del metodo mrDMDc. Inoltre considerando la crescita impellente della richiesta di metodi data-driven sempre più efficienti, questo elaborato di tesi è in grado di fornire le conoscenze base dei metodi data-driven e conseguentemente di consentire lo sviluppo di nuovi metodi.

# Bibliografia


[1] Robert Taylor. *Dynamic Mode Decomposition in Python*. URL: https://humaticlabs.com/blog/dmd-python/.

[2] Wikipedia contributors. *Moore–Penrose pseudoinverse*. URL: https://en.wikipedia.org/wiki/Moore%E2%80%93Penrose_inverse.

[3] J. Nathan Kutz. «Data-driven modeling & scientific computation: methods for complex systems & big data». In: *Data-driven modeling & scientific computation: methods for complex systems & big data*. 2013.

[4] J. Nathan Kutz Joshua L. Proctor Steven L. Brunton. *Dynamic mode decomposition with control*. URL: https://arxiv.org/pdf/1409.6358.pdf.

[5] Wikipedia contributors. *Composition operator*. URL: https://en.wikipedia.org/wiki/Composition_operator.

[6] J. Nathan Kutz Jacob Grosek. *Dynamic Mode Decomposition for Real-Time Background/Foreground Separation in Video*. URL: https://arxiv.org/pdf/1404.7592.pdf.

[7] Michael P. Bishop et al. «Wavelet analysis». In: *Spatial Analysis and Modeling in Geomorphology*. 2022.

[8] Wikipedia contributors. *Gabor Transform*. URL: https://en.wikipedia.org/wiki/Gabor_transform.

[9] Robert Taylor. *Optimal Singular Value Hard Threshold*. URL: https://humaticlabs.com/blog/optimal-svht/.

[10] Robert Taylor. *Sparsity-promoting DMD*. URL: https://humaticlabs.com/blog/spdmd-python/.

[11] Robert Taylor. *Humatic LABS*. URL: https://humaticlabs.com/.

[12] scikit-learn developers. *sklearn metrics mean squared error*. URL: https://scikit-learn.org/stable/modules/generated/sklearn.metrics.mean_squared_error.html.

[13] Nicola Demo et al. *PyDMD*. URL: https://github.com/mathLab/PyDMD.

[14] Nicola Demo et al. *DMDc PyDMD*. URL: https://github.com/mathLab/PyDMD/blob/master/pydmd/dmdc.py.

[15] Alessandro Trovatello. *Metodi data-driven e tecniche multi-resolution per l'identificazione di processi industriali*. URL: https://github.com/alessandrotrovatello/data-driven_methods.

[16] Luca Patanè e Maria Gabriella Xibilia. «Echo-state networks for soft sensor design in an SRU process». In: *Information Sciences* (2021), pp. 195–214.

[17] Luigi Fortuna et al. «Soft Sensors for Monitoring and Control of Industrial Processes». In: *Soft Sensors for Monitoring and Control of Industrial Processes*. 2007.

# *Ringraziamenti*

*Ai professori per avermi seguito durante la stesura di questo elaborato di tesi, ed in particolare vorrei ringraziare il Chiar.mo Prof. Luca Patanè, la Gent.ma Prof. Maria Gabriella Xibilia e l'Ing. Francesca Sapuppo.*

*A mia madre per gli enormi sacrifici fatti durante tutto il mio percorso accademico e di vita, per essermi stata sempre vicina e per l'empatia mostratami durante i periodi più difficili. Inoltre non finirò mai di ringraziarla per avermi cresciuto e per avermi fatto diventare la persona che sono oggi.*

*A mio padre, che continua a vivere al mio fianco illuminando il mio percorso, per l'aiuto che è riuscito e che sta riuscendo a darmi nonostante la sua assenza. Spero di essere riuscito a renderlo fiero di me.*

*Ai miei colleghi, per il sostegno morale durante i periodi di studio più impegnativi, un ringraziamento speciale al mio collega, ma soprattutto amico, Andrea Aliberti, per aver affrontato buona parte di questo percorso insieme e per aver trovato sempre una parola di conforto e incoraggiamento anche nei momenti difficili.*

*A Giuseppe e Pietro, in arte Pasti e Peet, i miei due migliori amici, nonchè fratelli (non di sangue) e parte della Triplice, per essermi stati sempre vicini, per le mille esperienze vissute insieme, i mille discorsi nel cuore della notte e gli infiniti momenti di felicità e a quelli che verranno. Siete, e sarete, sempre una parte fondamentale della mia vita.*

*A Francesco e Giuseppe, in arte Ciccio (PT) e Peppone, amici di una vita, cresciuti insieme fin da piccoli. Nonostante le nostre strade stiano prendendo una direzione diversa (una volta unite dal basket) ad oggi siamo sempre qui, ad uscire e divertirci come sempre. Vi ringrazio per il rapporto speciale che c'è tra noi, fate parte della mia famiglia e ci sarà sempre una parte di voi dentro me.*

*A Morgana e Laura, in arte Morghi e Lalli, le mie due amiche del cuore, per tutte le risate condivise e per tutti i momenti indimenticabili passati insieme. Per le fantastiche e brillanti persone che sono. Mi ritengo fortunato ad avervi nella mia vita.*

*A Michela, in arte Chica, la mia fidanzata, che in così poco tempo è riuscita a farmi innamorare di lei. La ringrazio in particolar modo per essermi stata vicina nelle fasi finali del mio percorso accademico, per avermi dedicato il suo tempo e per tutti i preziosissimi consigli che mi ha dato. Sei veramente speciale, al tuo fianco, mi rendi una persona migliore.*

*A tutto il gruppo PalmaSole (dipende dalla stagione), in particolare Arturo, Anna, Alberto, Chiara, Francesco, Gianluca, Giovanni, Federica, Konrad, Maurizio, Mauro, Rita, Rosario, Sergio e Vito. Grazie per le meravigliose estati passate insieme e per tutte quelle che verranno. Il miglior modo di rilassarsi dopo la sessione estiva.*

*Ed infine, ma non per importanza, vorrei ringraziare tutte le persone che mi sono state vicino durante tutto questo percorso e non solo. Vi ringrazio di cuore.*

-Alessandro