Universidade Federal de Santa Catarina
Centro Tecnológico
Engenharia de Controle e Automação

Data-Driven Algorithms for Problems in Science and Engineering

Final Project
Model identification and mode decomposition applied to Schröedinger's Equation

Jean Guilherme Neiverth

Florianópolis
2023

1 Introduction

As the analytical complexity of quantum physics topics increases, numerical approaches are increasingly used. Two common tasks for data-driven applications in this field are (I) model identification, in order to identify a model based on data and (II) model complexity reduction, to perform simulations with a lower computational cost.

The objectives of this project are:

- Run the numerical solver proposed in [3] for Schröedinger's Equation, in order to acquire data. Perform the algorithm for 5 different initial conditions, being one of them the particular case of the harmonic oscillator;
- Apply DMD to identify systems modes, enabling a less costly approach for simulation;
- Reproduce the results in [2], i.e., identify Schröedinger's PDE using SINDy algorithm.

2 Numerical solver for time-dependent Schröedinger's Equation

The time-dependant Schröedinger's Equation shows how the quantum state $\psi(x) \in \mathbb{C}$ of an particle evolve in time:

$$i\hbar\frac{\partial}{\partial t}\psi(x,t) = -\frac{\hbar^2}{2m}\frac{\partial^2}{\partial x^2}\psi(x,t) + V(x)\psi(x,t) \tag{1}$$

Here, $\psi(x,t)$ is a complex wavefunction $\psi(x)$ that changes with time, which means that it is a valid function for any $x \in \mathbb{R}$. However, as we are going to perform a numerical solver and it is impossible to work with an infinitely large array x, we set boundaries for $\psi(x,t)$ in terms of x, which will be inside a restricted length L:

$$\begin{aligned} \psi(x,t) &= 0 \qquad \forall x: \ x \leq 0 \\ \psi(x,t) &= 0 \qquad \forall x: \ x \geq L \end{aligned}$$

We can interpret $||\psi(x,t_o)||^2$ as the probability distribution of finding the particle in some position x in $t = t_o$. This way, $\psi(x,t)$ satisfies the following property for any time t:

$$\int_{-\infty}^{\infty} ||\psi(x,t)||^2 dx = 1 \tag{2}$$

The element $\hbar = 6.62607015 \times 10^{-34} JHz^{-1}$ is the Planck's constant and m is the particle mass. $V(x)$ is the energy function and depends on the environment where the particle is inserted and how the particle interacts with it.

As $\hbar$ is a very small number and it is inconvenient to work with numbers with a very small magnitude order in numerical solvers, we can apply some transformations to make the computing feasable:

- $x' = \frac{x}{L\hbar}$
- $t' = \frac{t}{mL^2\hbar}$
- $V'(x') = mL^2V(x')$
- $\hbar = 1$

After multiplying the in Equation 1 by mL^2 in both sides:

$$imL^2\hbar\frac{\partial}{\partial t}\psi(x,t) = -\frac{1}{2}\hbar^2L^2\frac{\partial^2}{\partial x^2}\psi(x,t) + mL^2V(x)\psi(x,t) \tag{3}$$

Applying the transformations in Equation 3:

$$i\frac{\partial}{\partial t'}\psi(x,t) = -\frac{1}{2}\frac{\partial^2}{\partial x'^2}\psi(x,t) + V'(x')\psi(x,t) \tag{4}$$

We can divide both sides by i, obtaining:

$$\frac{\partial}{\partial t'}\psi(x,t) = i\frac{1}{2}\frac{\partial^2}{\partial x'^2}\psi(x,t) - iV'(x')\psi(x,t) \tag{5}$$

Choosing a discrete grid with spacing $\Delta x'$ and $\Delta t'$ and letting $\psi_j^k = \psi(j\Delta x', k\Delta t')$, we can pass the equation to the discrete form:

$$\frac{\psi_j^{k+1} - \psi_j^k}{\Delta t'} = i\frac{1}{2}\frac{\psi_{j+1}^k - 2\psi_j^k + \psi_{j-1}^k}{\Delta x'^2} - iV'(x')\psi_j^k \tag{6}$$

Rearranging the elements:

$$\psi_j^{k+1} = \psi_j^k + \frac{i}{2}\frac{\Delta t'}{\Delta x'^2}\left(\psi_{j+1}^k - 2\psi_j^k + \psi_{j-1}^k\right) - i\Delta tV'(x')\psi_j^k \tag{7}$$

The continuous time-dependent Schröedinger's Equation naturally satisfies the condition 2. However, as we will be dealing with discrete-time numerical operations, numerical error can show up and make the distribution to don't satisfy this property. This way, we can implement a correction that force the system to satisfy this property.

$$Norm(\psi) = \sum_j ||\psi_j^{k+1}||^2 dx' \tag{8}$$

$$\psi^{k+1} \leftarrow \frac{\psi^{k+1}}{Norm(\psi)} \tag{9}$$

Using the Equation 7 and the correction in 9 , it is possible to numerically solve the Schröedinger's Equation for a given initial condition $\psi(x, t_o)$ and energy function $V(x)$. The image below shows the implementation of a function that performs this task using Python (Nt is the length of grid in time and Nx is the length of grid in space).

```python
def compute_psi(psi):
    for t in range(0, Nt-1):
        for i in range(1, Nx-1):
            psi[t+1][i] = psi[t][i] + 1j/2 * dt/dx**2 * (psi[t][i+1] - 2*psi[t][i] + psi[t][i-1]) - 1j*dt*V[i]*psi[t][i]

        normal = np.sum(np.absolute(psi[t+1])**2)*dx
        for i in range(1, Nx-1):
            psi[t+1][i] = psi[t+1][i]/normal

    return psi
```

Figure 1: Function in Python for numerically computing $\psi(x,t)$.

Another important issue is to guarantee that $\frac{\Delta t'}{\Delta x'^2}$ is a small number, so the increase step of ψ_j^{k+1} will not be very large. Once we have the simulation data, it is in the transformed version (Our space grid is x' and temporal grid is t'). So, in order to get back to SI units, we must invert the transformation:

- $x = x' L\hbar$
- $t = t' m L^2 \hbar$

Using the function in Figure 1, the 4 first simulations were performed using the same V'(x') energy function, which is in Figure 2. The initial conditions are visible in the Figure 3 and the final states after simulation, in Figure 4. The solving parameters are in the Table 1. These 4 datasets were used in Section 3.

Also, a 5th simulation was performed, representing the harmonic oscillator that will be used in Section 4. The energy function V'(x') is shown in Figure 5, the initial condition, in Figure 6, and the final state, in Figure 7. The parameters are the same as the 4 first cases.

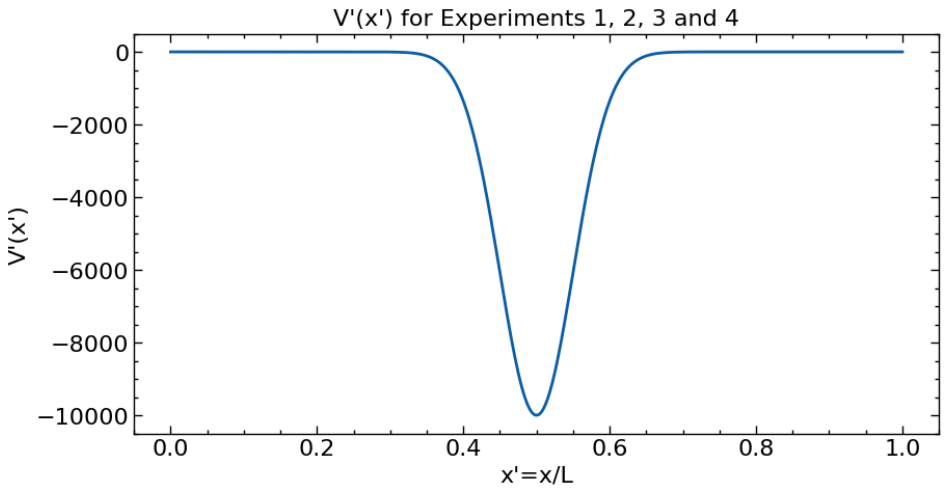

Figure 2: V'(x') for simulations 1, 2, 3 and 4.

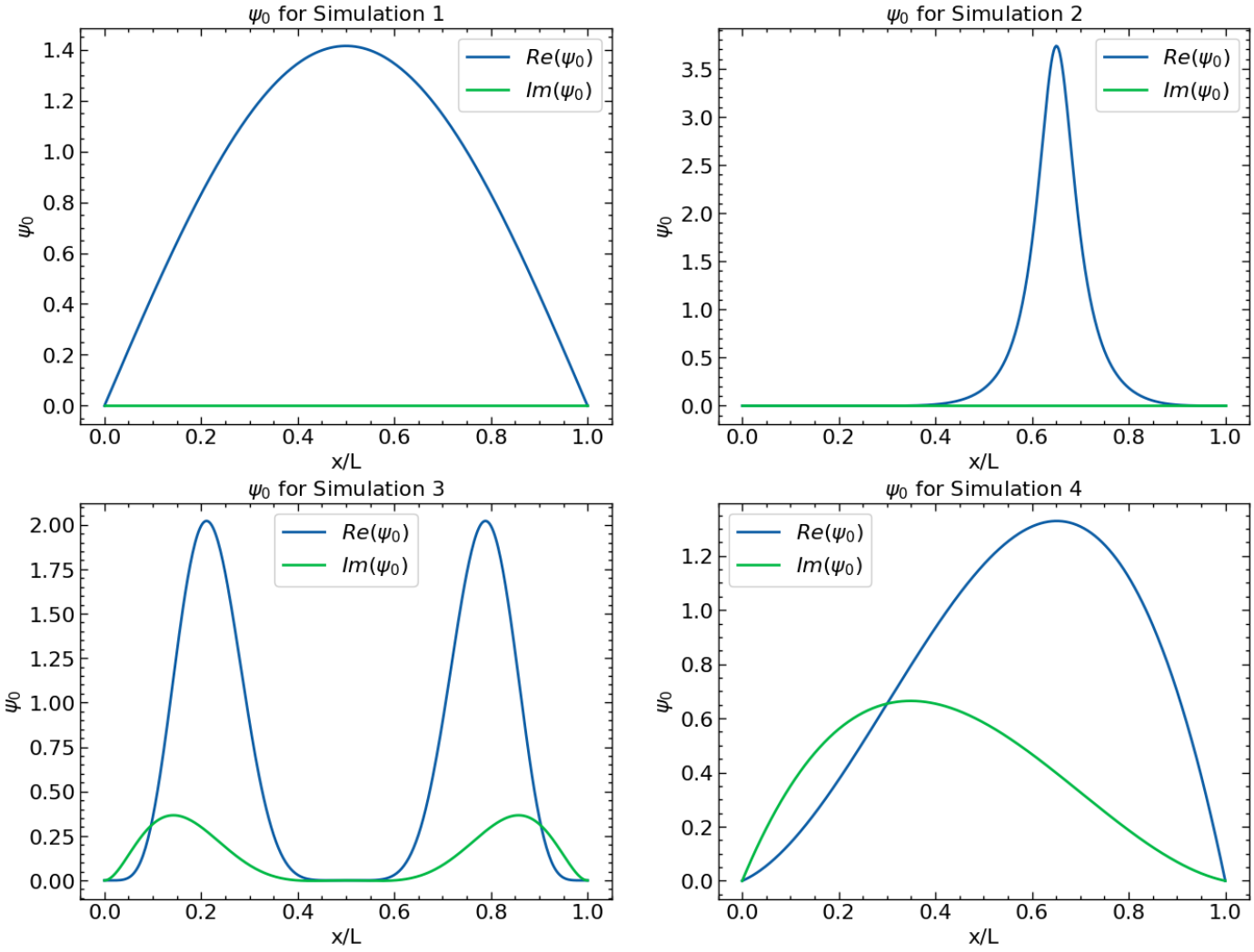

Figure 3: Initial conditions for simulations 1, 2, 3 and 4.

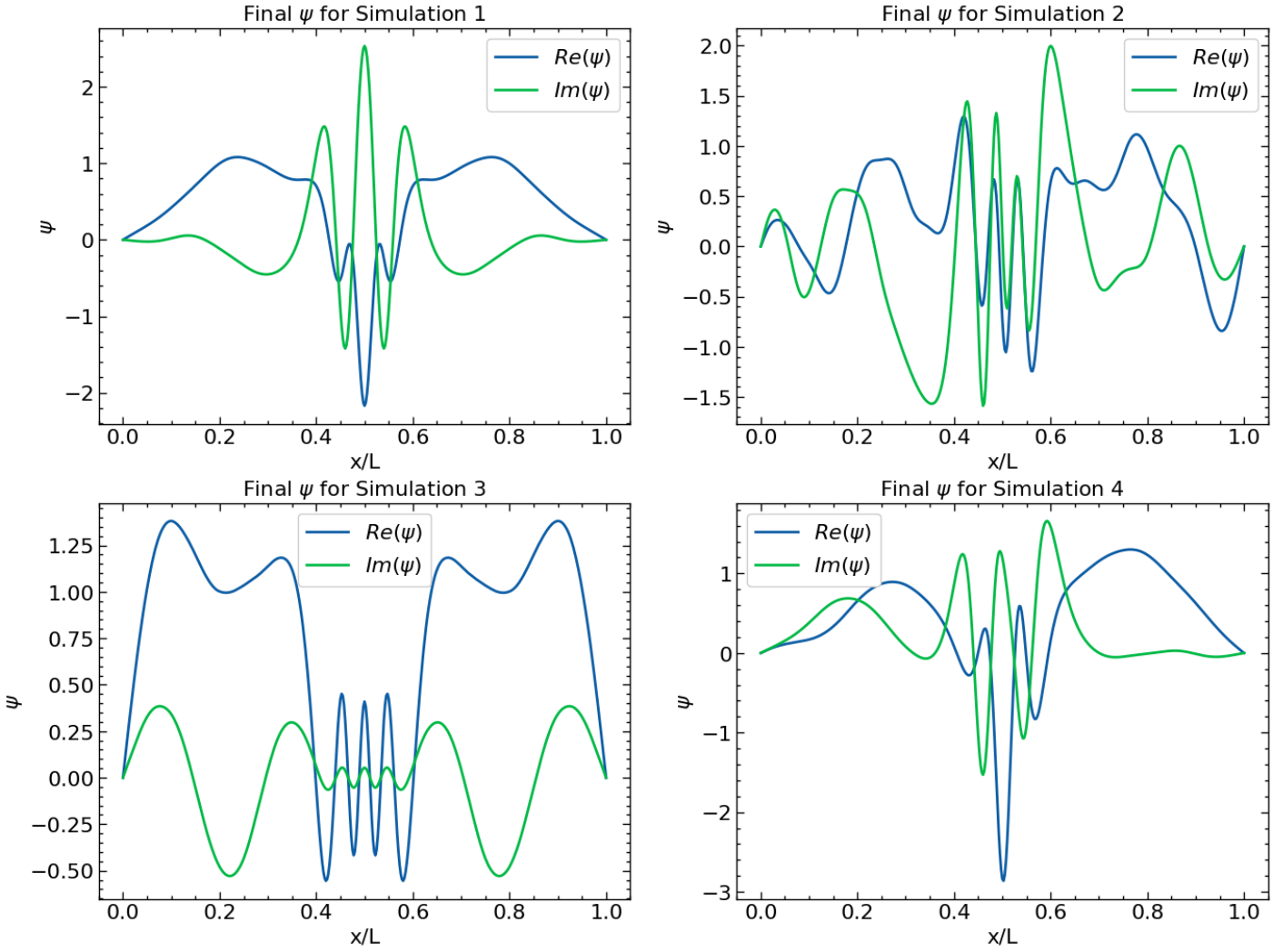

Figure 4: Final state for simulations 1, 2, 3 and 4.

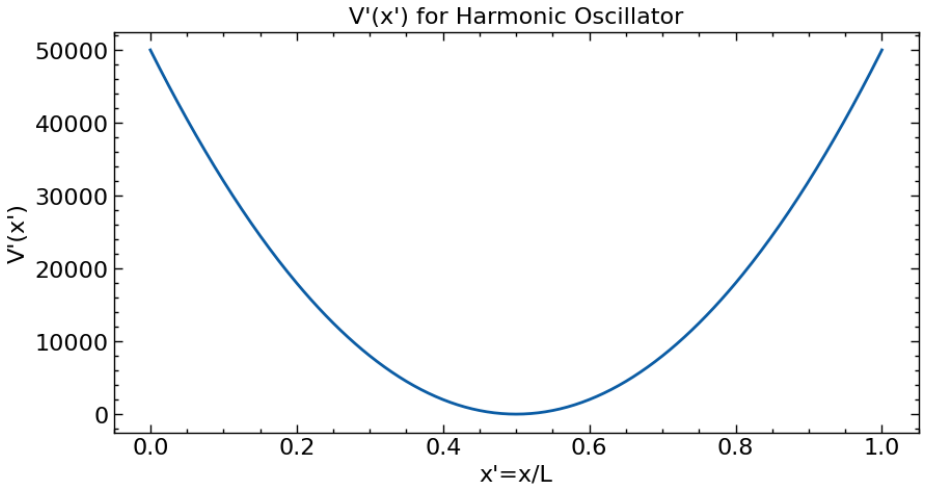

Figure 5: V'(x') for simulation 5.

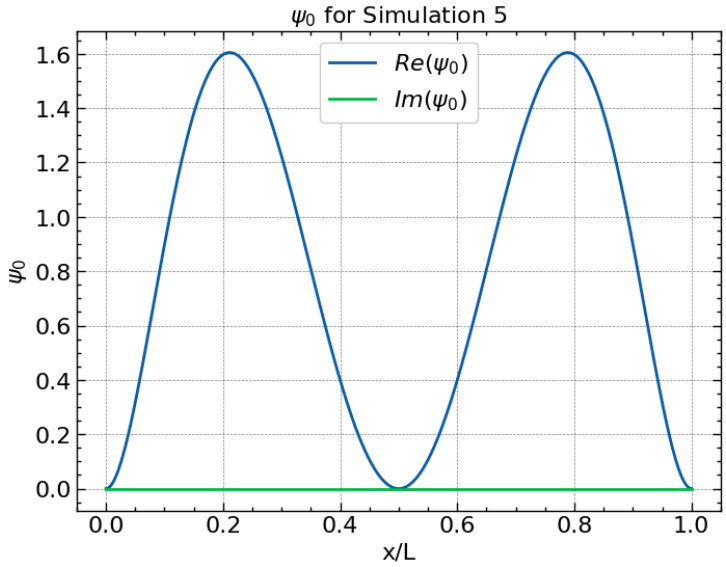

Figure 6: Initial conditions for simulation 5.

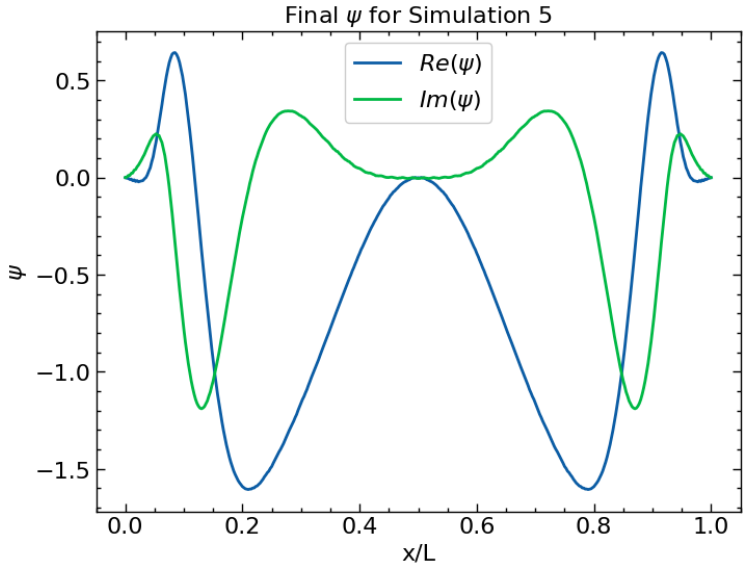

Figure 7: Final state for simulation 5.

Table 1: Used parameters.

Parameter	Description	value
Nt	Iterations in time in solving	10^6
dt	temporal step	10^{-8}
Nt'	Iterations in time after compression	10^4
Nx	Number of spatial measures	401
dx	Spatial step	0.0025
$\frac{\Delta t'}{\Delta x'^2}$	Low constant	0.0016

3 Dynamic Mode Decomposition

3.1 The algorithm

The DMD can have as objective two possible tasks. The first is to find a matrix A such that:

$$\dot{X} = AX, \quad \dot{X} = \frac{X_{K+1} - X_K}{\Delta t} \tag{10}$$

The second is also to find a matrix A, but that satisfies:

$$X_{K+1} = AX_K \tag{11}$$

In a more general way, we can say that:

$$Y = AX \tag{12}$$

The difference between the two tasks is basically how the data is arranged before being inputted into the algorithm. Independently of what task is performed, the algorithm is the same and consists of the following steps:

I. Perform SVD on X, truncating on rank r:

$$\tilde{U}\tilde{\Sigma}\tilde{V}^* = SVD(X) \tag{13}$$

II. Build $\tilde{A}$:

$$\tilde{A} = \tilde{U}^* Y V \Sigma^{-1} \tag{14}$$

III. Find DMD modes:

$$\mu = \text{eigenvalues}(A) \tag{15}$$
$$W = \text{eigenvectors}(A) \tag{16}$$
$$\Phi = YV\Sigma^{-1}W \tag{17}$$

IV. Compute A:

$$A = \Phi \text{Diag}(\mu)\Phi^{-1} \tag{18}$$

Where Φ^{-1} can be obtained using SVD.

The Python implementation code is shown in Figure 8 below.

```python
from numpy import diag
from numpy.linalg import inv, eig, pinv
from scipy.linalg import svd

def DMD(X, Y, r=10):

    # Step 1, perform SVD for X
    U2,Sig2,Vh2 = svd(X, False)

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

    # Step 2, build A tilde
    Atil = U.conj().T @ Y @ V @ inv(Sig)

    # Step 3 build DMD modes
    mu,W = eig(Atil)
    Phi = Y @ V @ inv(Sig) @ W

    # Step 4, compute A
    Phi_inv = pinv(Phi)
    A = Phi @ diag(mu) @ Phi_inv

    return A
```

Figure 8: Python implementation of DMD.

3.2 Finding the best DMD type and rank

The first experiment consisted in find the matrix A for each of the 4 first simulations, considering both tasks in Equation 10 and Equation 11. Then, the data were reconstructed from the initial conditions and the mean absolute error were computed for each of them. The results are in the Figure 9. It was concluded that the task $X_{K+1} = AX_K$ is the one that best fits the problem.

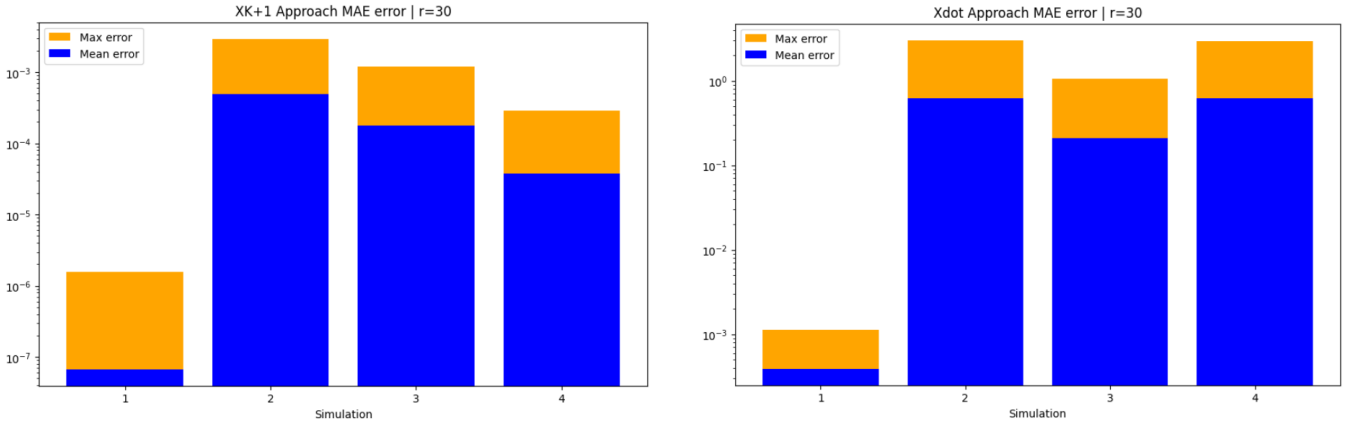

Figure 9: Comparison between two possible DMD tasks.

After that, it was necessary to choose a rank to truncate. For all the simulations, many different ranks were tested, also evaluating using the mean absolute error. The results are in the Figure 10 and the choosen rank was r=60.

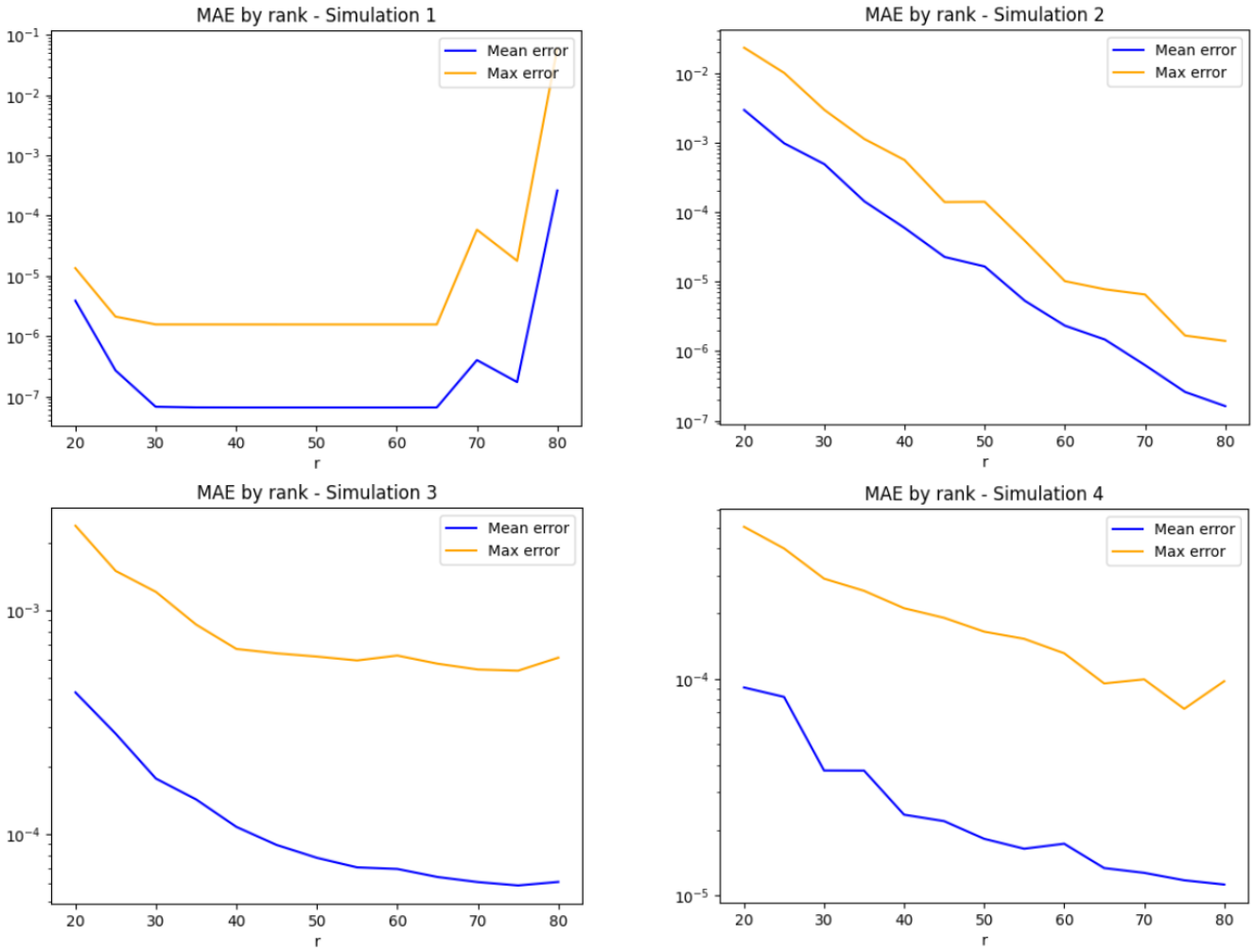

Figure 10: Comparison between different ranks to truncate.

3.3 Utility

It is possible to check that the data could be nicely reconstructed for all simulations with rank r=60 (Figure 10). It means that it is possible to represent the original 401x10000 matrix as the initial condition plus the matrix A, with a low reconstruction error.

If we consider that the complex numbers have a size of 128 bites, it means that the size of the original dataset is given by:

$$S_o = \frac{401 \times 10000 \times 128}{1024^2} \approx 489.50\text{MB} \tag{19}$$

Using the reconstruction, however, it is possible to store an approximation of the results within a lower size, since we will only have the initial condition (401×128) and the matrix A ($401 \times 401 \times 128$). This way, the reconstruction storage size is:

$$S_R = \frac{401 \times 402 \times 128}{1024^2} \approx 19.67\text{MB} \tag{20}$$

So the ratio between them is:

$$Ratio = \frac{S_R}{S_o} = \frac{19.67}{489.50} = 4.02\% \tag{21}$$

This means that, using DMD to approximate the simulation data, it is possible to save around 96%

storage.

One could think that a negative issue with this approach is the time for running the reconstruction. However, in the experiments performed, the execution times for reconstructing the data were lower than for loading the original data from the hard disk. The results are shown in the Figures 11 and 12. This possibly happens because of hard disk processing limitations.

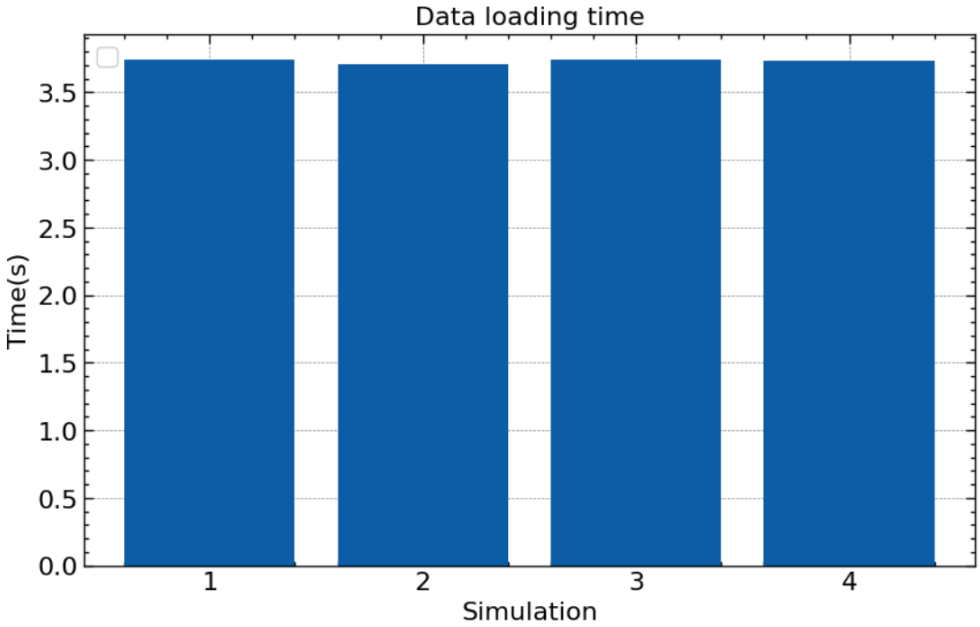

Figure 11: Time for data loading.

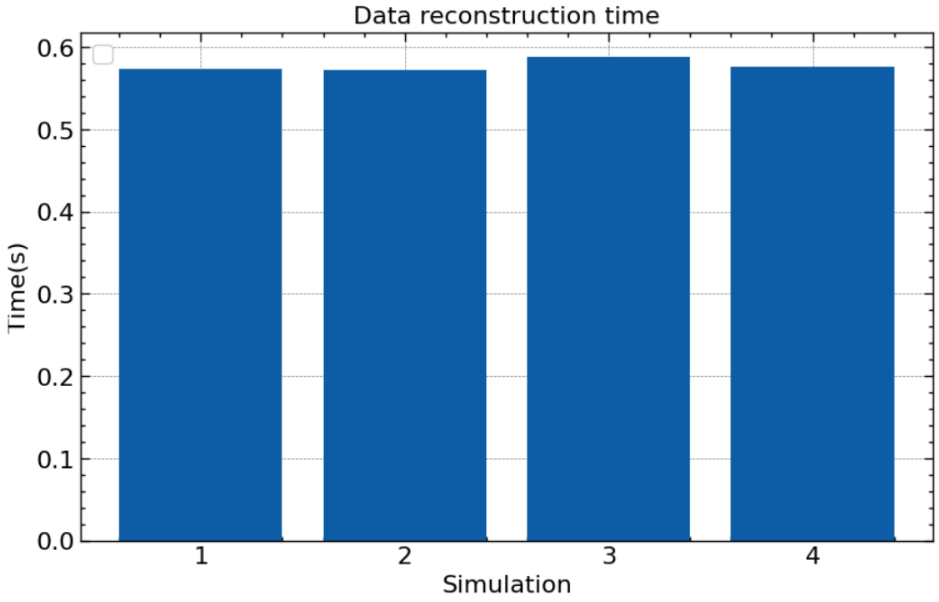

Figure 12: Time for data reconstruction.

3.4 Limitations

Despite reconstructing well the behaviour from the fitted data, the matrix A cannot do the same for simulations in different initial conditions. If we use DMD to find the matrix A for the first simulation, for example, it will reconstruct badly the data from the second, third and fourth simulations.

The Figure 13 below shows how the models fitted for each data performed in other simulations. The conclusion we can take from here is that the matrix obtained with DMD is just like a footprint of our data, but we can not consider as a general simplification of the Schröedinger's Equation.

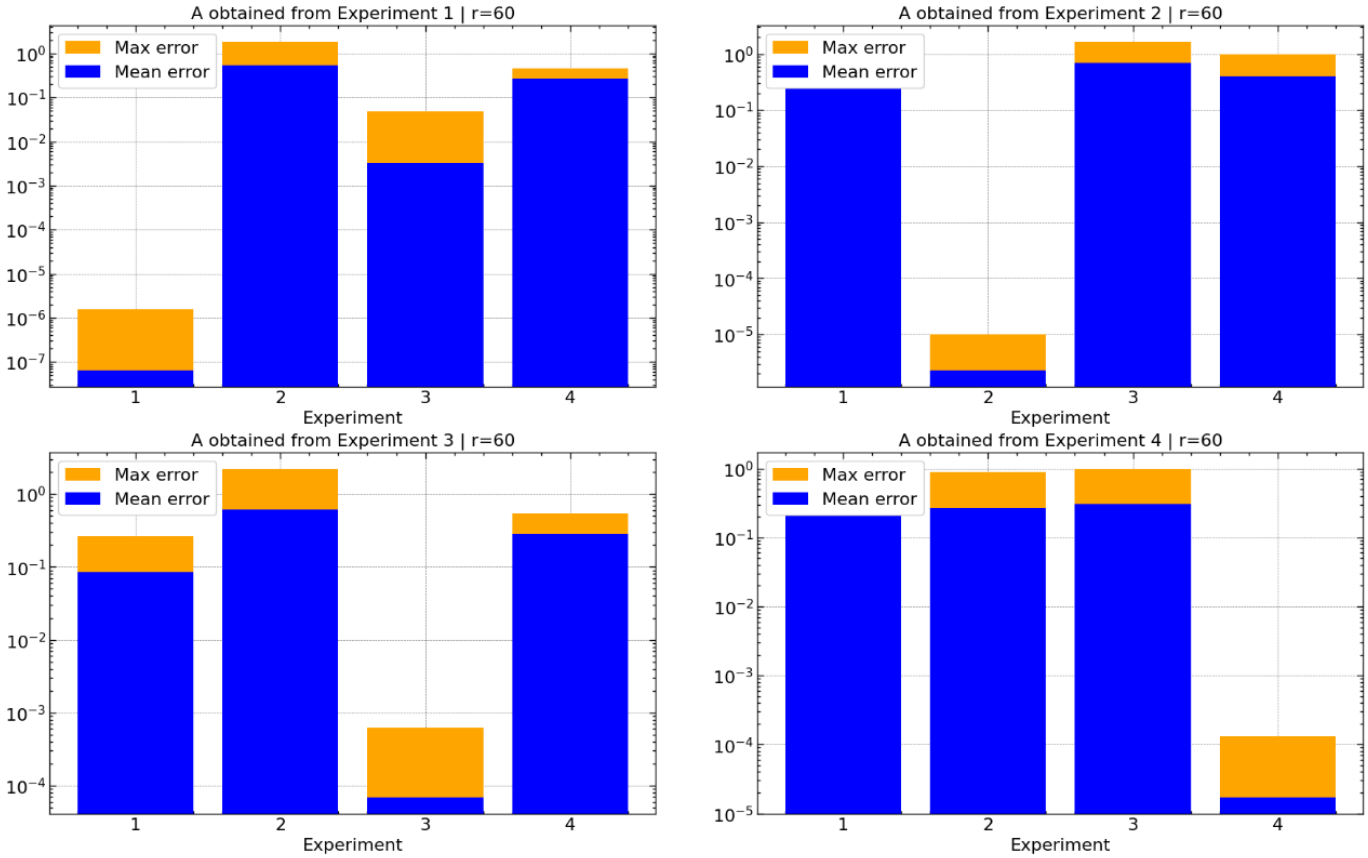

Figure 13: DMD reconstruction performance out of original data.

4 Sparse Identification of Non-Linear Systems

4.1 The algorithm

The objective of SINDy is to find a representation of a system based on experimental data:

$$\frac{d}{dt}x = f(x), \qquad f(x) \approx \sum_{k=1}^{p} \theta_k(x)\xi_k = \Theta(x)\xi \tag{22}$$

Where $\Theta(x)$ is a set of candidate functions.
First, data is obtained and the numerical derivatives are computed.

$$X = [x(t_1)\ x(t_2)\ ...\ x(t_m)]^T, \qquad \dot{X} = [\dot{x}(t_1)\ \dot{x}(t_2)\ ...\ \dot{x}(t_m)]^T \tag{23}$$

Then, it is necessary a algorithm to find a sparse set of functions to represent the data.

$$\xi_k = \text{argmin} \quad ||\dot{X}_K - \Theta(X)\xi_k'||_2 + \lambda||\xi_k'||_1 \tag{24}$$

However, in the present project, the STLS (Structured Total Least Squares) optimization process was used. It basically relies on using the least squares method to obtain the weights ξ_k and iteratively cut off the parameters lower than a previuosly determined thershold. The Python code used is shown in the figure 14.

```python
def get_df_by_index(u_index):

    # u -> quantum state
    u_minus1 = D5[u_index-1,1:] # u in the previous x position
    u = D5[u_index,1:] # u
    u_plus1 = D5[u_index+1,1:] # u in the next x position

    ud = D5[u_index,0:-1] # u befor

    ux = (u_plus1 - u)/dx
    uxx = (u_plus1 - 2*u + u_minus1)/dx**2
    ut = (u - ud)/dt

    x = np.array([x_grid[u_index]]*len(u))

    df = pd.DataFrame({'u':u,'x':x,'ux':ux,'uxx':uxx,'ut':ut})

    return df

def get_df():
    index_list = np.arange(10,400,10)
    l = [get_df_by_index(u_index) for u_index in index_list]
    data = pd.concat(l,axis=0)
    data.reset_index(drop=True,inplace=True)
    return data
```

```python

data = get_df()

data['x^2'] = data['x']**2
data['u^2'] = data['u']**2

data['u*ux'] = data['u']*data['ux']
data['u*uxx'] = data['u']*data['uxx']

data['u^2*ux'] = data['u^2']*data['ux']
data['u^2*uxx'] = data['u^2']*data['uxx']

data['x*ux'] = data['x']*data['ux']
data['x*uxx'] = data['x']*data['uxx']

data['x^2*ux'] = data['x']*data['ux']
data['x^2*uxx'] = data['x']*data['uxx']

data['x*u'] = data['x']*data['u']
data['x^2*u'] = data['x^2']*data['u']
data['x*u^2'] = data['x']*data['u^2']
data['x^2*u^2'] = data['x^2']*data['u^2']
```

Figure 14: SINDy implementation in Python.

4.2 Identification of the Harmonic Oscillator PDE

In the particular case of the Harmonic Oscilator, the Schröedinger's Equation becomes:

$$u_t = \frac{i}{2}u_{xx} - \frac{i}{2}Kx^2u \tag{25}$$

Where u is the quantum state, previously called $\psi(x,t)$

The equation above is valid when $V(x) = \frac{K}{2}x^2$. However, as it is possible to visualize in Figure 5, the energy function for the simulation 5 is:

$$V'(x') = \frac{K}{2}(x' - 0.5)^2 = \frac{K}{2}(x'^2 - x' + 0.25) \tag{26}$$

$$K = 4.10^5 \tag{27}$$

So the PDE turns into:

$$u_t = \frac{i}{2}u_{xx} - \frac{i}{2}Kx^2u + \frac{i}{2}Kxu - \frac{i}{8}Ku \tag{28}$$

Using the data generated in the simulation 5, which is relative to the Harmonic Oscillator, the algorithm was applied using the candidate functions shown in Figure 15, obtaining the result in Figure 16.

```
['u',
 'x',
 'ux',
 'uxx',
 'x^2',
 'u^2',
 'u*ux',
 'u*uxx',
 'u^2*ux',
 'u^2*uxx',
 'x*ux',
 'x*uxx',
 'x^2*ux',
 'x^2*uxx',
 'x*u',
 'x^2*u',
 'x*u^2',
 'x^2*u^2']
```

Figure 15: SINDy candidate functions.

```
u (608.7066-39399.3978j)
x (1635.9325-68.2057j)
x^2 (-1773.0934+72.9495j)
u^2 (-641.5033-255.6025j)
x*u (-2078.1972+107987.867j)
x^2*u (2078.4225-107985.5314j)
x*u^2 (3089.7056+1323.8074j)
x^2*u^2 (-3089.6852-1323.6781j)
```

Figure 16: First SINDy result.

It was not possible to achieve the expected results at first, because of the issue involving the magnitude order of the spring constant K, that is very high. That given, it was necessary to perform a normalization in the variables, given by:

$$x_{normal} = \frac{x}{max||x_j||} \tag{29}$$

This makes all variables inside the same range of values, so it is possible to pick a cut off threshold that is in a magnitude order compatible with the terms u_{xx} and those related to the spring constant. After finding the sparse dependence, it was possible to transform back the coefficients using:

$$x_{OriginalCoef} = x_{Coef}\frac{max||u_{tj}||}{max||x_j||} \tag{30}$$

After this normalization process, it was possible to reproduce the result in [2] (discover Schröendiger's PDE with SINDy), which was the objective of this section. The result can be visualized in figure 17. The final cut-off parameter was 0.5.

```
u -49991.08952626009j
uxx 0.4998913888713031j
x*u 199957.62371113925j
x^2*u -199957.62371113925j
```

Figure 17: SINDy result after normalization.

4.3 Limitations

In the current experiment, the PDE was previosly known, so it was possible to verify that the SINDy algorithm hadn't performed well before the normalization. In a real discovery process where the actual dependence is unknown, it is necessary to use a verification method, such as cross validation. Also, one will never know if the system found is a global law or just a good representation of the system under the specific measurement conditions.

The combinatory of the candidate functions can become huge for PDEs, so it is necessarry to be careful in their choice. Finally, it is important to keep in mind the importance of the cut-off threshold when using the STLS method and, if there is no good performance, it is possible to try other regularization process, such as L1 norm.

5 Conclusions

Data-driven algorithms can be useful in Quantum Physics field not just by identifying experimental data models, but can also help with data handling like storage using DMD, that was able to save 96% of storage size. The DMD algorithm may not perform well for predicting trajectories of non linear systems. It only detects the linear behaviour for each singular experiment.

SINDy algorithm was able to find the Schröedinger's PDE (Harmonic Oscillator). However, there were some critical points:

- It was only possible after data normalization;
- One must be careful when identifying unknown PDEs, because of the influence of the cut-off threshold and the necessity of good candidate functions;
- The combination of possible partial derivatives and functions can make the candidate functions set become huge.

6 References

[1] Steven L. Brunton and J. Nathan Kutz, "Data-Driven Science and Engineering: Machine Learning, Dynamical Systems, and Control," Cambridge University Press, 2019.

[2] S. H. Rudy, S. L. Brunton, J. L. Proctor, and J. N. Kutz. Data-driven discovery of partial differential equations. Science Advances, 3(e1602614), 2017.

[3] Base code - Schröedinger Equations numerical solver