Time Series Analysis in Python: User Manual

Wenyan Gong, Zongxi Li, Cong Ma
Qingcan Wang, Zhuoran Yang, Hao Zhang

January 16, 2017

1 Introduction

Time series analysis comprises methods for analyzing time series data in order to extract meaningful statistics and other characteristics of the data. It is widely used in signal processing, pattern recognition, mathematical finance, weather forecasting, earthquake prediction, control engineering, and largely in any domain of applied science and engineering which involves temporal measurements.

In this project, we will play a game with time series in finance. It has gained its popularity in Wall Street recently, since it is fundamental to most promising quantitative investment strategies. We develop a system that can predict future prices of stocks using various kinds of methods for time series analysis.

This manual is a brief introduction of the functionality of our package. In specific, we illustrate how to use the provided functions to fit the model, predict the stock price, trade, cluster etc. Introduction of the main source files and key functions is also included.

We utilize standard packages from python including `numpy`, `scipy`, and `matplotlib`. The integration of `C` code and `Python` code is based on the `Cython` package.

2 Program Structure

The high-level program structure is shown in Figure 1. The division of work is pretty even, and there are some minor work that are too trivial to mention extensively here. Overall, the program consists of two parts. The first part deals with a single time series, and exploits the time correlation within the time series. There is data preprocessing module that takes the raw data and get the right data format for later analysis. Optimization, model, and solver are used all together to fit the models using maximum likelihood estimation. Finally, there is a post processing module that makes use of the information from model. The post processing module includes parameter inference, trading strategy, and option pricing. The second part deals with a collection of time series of data, basically it exploits the correlation between different time series. In this way, we can gain more insights into the financial market. While these insights are impossible to obtain from a single time series.

3 Functionality

To start with, we will give an description of the source files.

- `basemodel.py`: Provided the basemodel of time series.

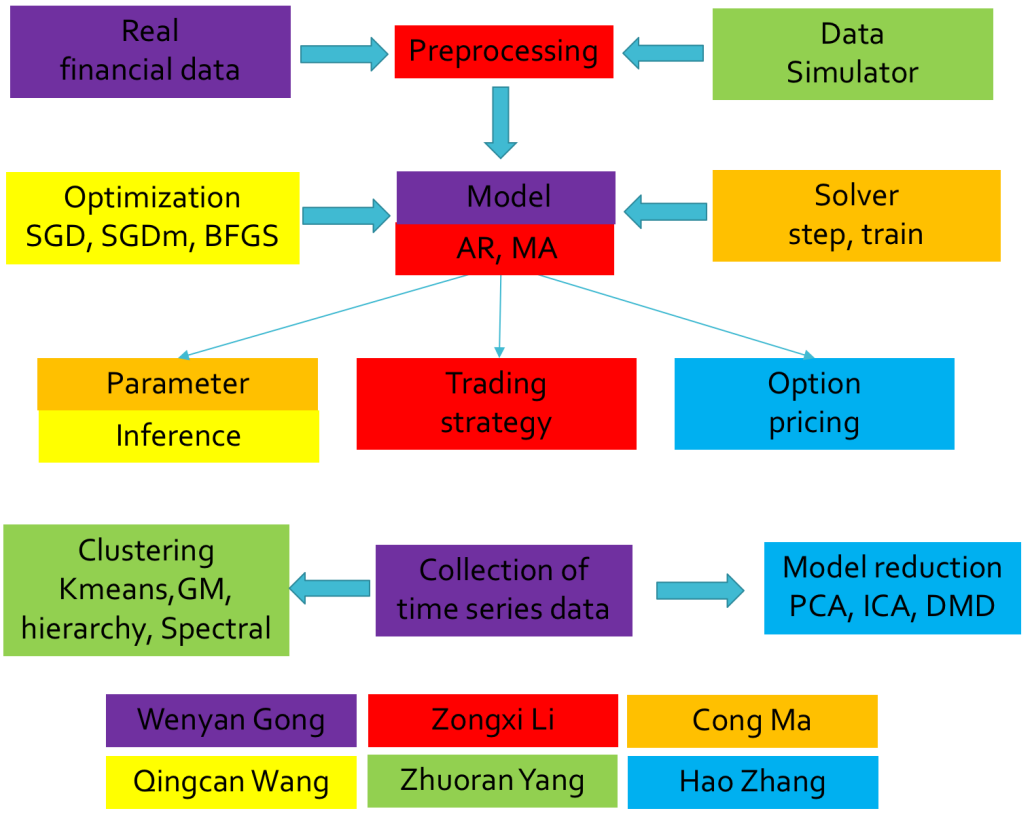

Figure 1: Program structure and division of work

- `cluster.py`: Provided the `Cluster` class for clustering. It includes four clustering methods: k-means, hierarchy clustering, Gaussian mixture modeling, and spectral clustering.
- `Data_prossessor.py`: Provided the transforming function between stock price and return, the function getting the maximum drawdown and the function returning the indicator of peak and trough.
- `Gradient_check.py`: Provided the function for deriving numerical gradient.
- `Inference.py`: Provided functions for computing the auto-covariance of `AR` model and testing significance of parameters.
- `Model.py`: Provided model class `AR` and `MA`, including functions for computing the log-likelihood and gradient given the data and an `AR` or `MA` model and the function for future price prediction.
- `Optim.py`: Provided several optimization methods: stochastic gradient descent, stochastic gradient descent with momentum update and Broyden–Fletcher–Goldfarb–Shanno algorithm
- `Option_pricing.py`: Provided functions for option pricing.
- `Reduction.py`: Provided functions for dimensional reduction for data visualization.
- `Solver.py`: Provided functions for model fitting using maximum likelihood estimation given the data and the specified model.
- `Trading.py`: Provided functions for deciding the transaction time point based on future price prediction and computing profit over the time.

- `Ts_gen.pyx` & `c_ts_gen.c`: Provided functions to generate simulation data.

3.1 Data Preparation and Preprocessing

We provide both a real dataset and methods to generate simulated data from various time series models. For data simulation, we provide functions that samples from the `AR(p), MA(q), ARMA(p,q), GARCH(p,q)` models, which is written in `C` code. See `c_ts_gen.c` for the source code. To integrate `C` code into `Python`, we write a `Cython` wrapper `ts_gen.pyx`, which generates a `Python` module named `ts_gen`. This module consists of five functions. The inputs of these functions all consist of four elements:

1. `param`: a `numpy Nd array` or a few `float` numbers – the model parameters.
2. `time`: an `integer` – the total time of the time series.
3. `num`: an `integer` – number of independent sample paths,
4. `burnin`: an `integer` – the length of burnin period. This parameter has default value 2000.

These functions all output a `numpy Nd array` with dimension (`num` $\times$ `time`) which stores `num` independent paths of the time series data until $T =$ `time`. Functions that generate simulated data are listed as follows.

```
-ar1_gen(double rho, double sigma, int time, int num, int burnin)
-ma1_gen( double rho, double constant,  int time,  int num,  int burnin
-arch1_gen(double a0, double a1, int time, int num, int burnin)
-garch11_gen(double a, double b, double c, const int time, const int num,
    const int burnin
-garch11_gen(double a, double b, double c, const int time, const int num,
    const int burnin )
-arma_gen( np.ndarray[double, ndim=1, mode="c"] ar, int p, np.ndarray[double
    , ndim=1, mode="c"]  ma, const int q, const double sigma,
const int time, const int num, const int burnin)
```

For an example of usage, see the following example.

```
from tsap.ts_gen import arma1_gen
import numpy as np
Y = ar1_gen(0.5, sigma = 1.0, time = 200, num = 10, burnin = 2000)
print(Y.shape)
```

The output, as expected, is

```
>>(10, 200)
```

3.2 Model

The model related functions are written in `model.py`, which contains two model classes: AR and MA. They are used to fit an AR or MA model by given parameters, to compute the log-likelihood and gradients given an AR or MA model and the data and to give predictions. The functions are introduced below. Firstly, you can fit an AR or MA model by given parameters. The inputs are as follows:

1. `lag`: the lag parameter for the fitted model, a number;
2. `phi`: the coefficients for the fitted model, a `numpy` array whose dimension is the `lag` given above;
3. `sigma`: the variance of noise for the fitted model, a number;
4. `intercept`: the intercept of noise for the fitted model, a number.

The output is a model object. To fit the model, we can run

```
AR_model = AR(lag=3, phi=np.array([[1],[1],[1]]), sigma=1, intercept=0.1)
MA_model = MA(lag=3, phi=np.array([[1],[1],[1]]), sigma=1, intercept=0.1)
```

Secondly, you can compute the log-likelihood and gradients given an AR or MA model and the data. The input is as follows:

1. `data`: One or several time series. The input data should be a 2-dimensional array. Each row of data represents a time series over the time, e.g. the stock return over the year of a single stock. The number of rows should be the number of time series.

The output is a number for the computed loglikelihood and a tuple for the computed gradients. This function is public but mostly it serve for the `Solver` class. To compute the log-likelihood and the gradients, we can run

```
AR_llh, AR_grads= AR_model.loss(data)
MA_llh, MA_grads= MA_model.loss(data)
```

Finally, you can make prediction based on a fitted model. The input is as follows:

1. `data`: the time series before the first prediction. The number of columns (the length of the time series) should exceed the given lag for a reasonable prediction;
2. `nstep`: the number of predictions.

To make predictions, we can run

```
AR_future=AR_model.predict(data,nstep=5)
MA_future=MA_model.predict(data,nstep=5)
```

3.3 Optimization

The optimization part `optim.py` implements various optimization update rules: stochastic gradient descent (`sgd`), stochastic gradient descent with momentum update (`sgd_momentum`) and Broyden–Fletcher–Goldfarb–Shanno algorithm (`bfgs`).

Each update rule accepts current iteration point and the gradient of the object function and produces the next iteration point. Each update rule has the same interface:

```
def update(x, dx, config=None):
```

The inputs are as follows:

1. `x`: A `numpy` array giving the current iteration point.

2. `dx`: A `numpy` array of the same shape as `x` giving the gradient of the object function with respect to `x`.

3. `config`: A dictionary containing hyper-parameter values such as learning rate, momentum, etc. If the update rule requires caching values over many iterations, then config will also hold these cached values.

The outputs are as follows:

1. `next_x`: The next point after the update.

2. `config`: The config dictionary to be passed to the next iteration of the update rule.

The update rules are called in `Solver` class like this:

```
objfcn, grads = self.model.objfcn(self.X)
for p, x in self.model.params.iteritems():
    dx = grads[p]
    config = self.optim_configs[p]
    next_x, next_config = self.update_rule(x, dx, config)
    self.model.params[p] = next_x
    self.optim_configs[p] = next_config
```

3.4 Solver

We provide a class named Solver which bridges the model class and the optimization methods. The solver class is used to solve the maximum likelihood estimation of various time series models. To construct an instance of the solver class, you need the following inputs:

1. `solver`: an instance of the model class. Most importantly, given parameters of the model and the data, it can return the negative log-likelihood of the data and the gradients with respect to the parameters.

2. `data`: an `numpy 2d array` which specifies the data we obtain. The size of the `numpy 2d array` is $N \times T$, where N is the number of time series we get and T is the length of each time series.

3. `kwargs`: a list of command line arguments. They include

   (a) `update_rule`: a `string` used to specify the optimization method to use. It could be either `sgd` or `sgd_momentum`.

   (b) `optim_config`: a `dictionary` used to specify the hyper-parameters for the optimization methods. For example, the key can be a `string` like `learning_rate` and the value can be `1e-5`.

   (c) `num_epochs`: an integer number used to specify the number of epochs for the optimization process

   (d) `batch_size`: an integer number used to specify the number of examples for one step of optimization process.

   (e) `print_every`: an integer number used to specify how often the program displays the information of the loss value.

For an example of usage, see the following.

```
from tsap.solver import Solver
from tsap.model import AR, MA
from tsap.ts_gen import ar1_gen
lag = 1
sigma = 2.0
intercept = 0.1
phi = np.random.randn(lag, 1)
AR_model = AR(lag=lag, phi=phi, sigma=sigma, intercept=intercept)

# use the Solver to solve the AR model
AR_solver = Solver(AR_model, Y,
update_rule='sgd',
optim_config={
'learning_rate': 3e-5,},
num_epochs=3000, batch_size=1,
print_every=100)
AR_solver.train()
AR_model.params
```

The output, as expected, is

```
the loss is 424.110945
the loss is 399.237624
the loss is 378.012857
the loss is 357.765501
the loss is 337.557537
the loss is 317.542124
the loss is 299.271549
the loss is 285.752707
the loss is 278.916407
the loss is 276.705150
the loss is 276.152286
the loss is 276.012074
the loss is 275.972341
{'intercept': array([-0.12022489]),
'phi': array([[ 0.47215311]]),
'sigma': array([ 0.96874162])}
```

For more examples. please refer to the demos.

3.5 Inference

The methods that perform statistical inference is written in `inference.py`, which consists of five functions, i.e., `acovf`, `acf`, `BL_stat`, `yule_walker`, and `ar_select`. In the following, we give a detailed introduction of the usages of these functions. Firstly, `acovf` and `acf` returns the auto-covariance and auto-correlations of one path of time series. The input include

1. `x`: an one dimensional `numpy` array.
2. `demean`: a boolean variable that tells whether to subtract the mean.
3. `nlags`: maximum order of lags in autocorrelation.
4. `fft`: boolean, whether use fast Fourier transform.

The output is an array of auto-covariance or auto-correlation. To use these function, we can run

```
auto_cov = acovf(x, demean=True, fft=False)
auto_cor = acf(x, nlags = 40, demean=True, fft=False)
```

Secondly, `BL_stat` performs Box-Ljung test that tests if the time series data are correlated. The input are `acf_array`, which is an array of auto-correlation coefficients (ACF), `order`, which is the maximum order of ACF used, and `nobs`, which is the time length that is used to compute the ACF. The outputs are the test statistic, a float number , the p-value, also a float number, and the test result, which a boolean. To use this function, an example is

```
test_stat, pval, test_result = BL_stat(acf_array, order, nobs):
```

Finally, `yule_walker` and `ar_select` are specified for the `AR(p)` model. In specific, `ar_select` takes an array of time series as input and outputs the estimated order of the autoregresive model. Whereas `yule_walker` takes the time series and the order as input and outputs the estimated parameters using Yule Walker method. The usage will be clear from the following example.

```
import tsap.inference as inference
from tsap.ts_gen import ar1_gen

ts = ar1_gen(0.3, sigma = 1.0, time = 500, num = 1, burnin = 2000)
order_est = inference.ar_select(ts)
print("The estimated order of AR model is "+ str(order_est))
rho, sigma = inference.yule_walker(ts, order = 1)
print("The estimated model parameter is "+ str(rho))
```

The output is

```
>>The estimated order of AR model is 2
>>The estimated model parameter is [ 0.2973149]
```

3.6 Trading Strategy

The functions for trading is written in `trading.py`, which consists of four functions, i.e., `signal_generation`, `profit_loss`, `trade`, and `rolltrade`. In the following, we give a detailed introduction of the usage of these functions.

Firstly, `signal_generation` returns the trading signal, like what time to buy and what time to sell. The input includes

1. `X`: a one dimensional `numpy` array, which is the price series.

2. `window`: a positive integer that specifies the largest length of one trade. If `window`=5, that means there's at most one trade activity within 5 days. The default number is 2.

The output is a one dimensional numpy array, whose elements are 1 (buying signal), -1 (selling signal), 0 (no signal). To use these function, we can run

```
import trading
signal = trading.signal_generation(X, window = 5)
```

Secondly, `signal_generation` returns the profit based on given trading signals. The input includes

1. `X`: a one dimensional `numpy` array, which is the price series.
2. `signal`: a one dimensional `numpy` array that contains trading signals. Usually, it takes the output of `signal_generation`.
3. `money`: a positive float number, which is the initial wealth. The default number is 1.

The output is a one dimensional numpy array, which is the time series of profit at different time points. To use these function, we can run

```
import trading
signal = trading.signal_generation(X, window = 5)
profit = trading.profit_loss(Y, signal, money = 100)
```

Next, `trade` returns the profit in certain time period. Specifically, given a fitted model, it uses this model to forecast future returns based on historical returns, which are obtained by transfering historical prices to historical returns. After that, it again transfers the returns to prices and then generates trading signal based on predicted prices. At last it calculates the profit based on the real price series and the trading signals. The input includes

1. `X`: a one dimensional `numpy` array, which is historical price.
2. `Y`: a one dimensional `numpy` array, which is future prices.
3. `M`: a class from `model.py`, which is a fitted model.
4. `nstep`: an positive integer, which specifies how many steps to predict.
5. `window`: an positive integer that specifies the largest length of one trade. If `window`=5, that means there's at most one trade activity within 5 days.
6. `money`: a positive float number, which is the initial wealth. The default number is 1.0.

The output is a one dimensional numpy array, which is the time series of profit at different time points. To use these function, we can run

```
import trading
import model
AR_model = model.AR(lag=3, phi=np.array([[1],[1],[1]]), sigma=1, intercept
    =0.1)
profit = trading.trade(X, Y, M = AR_model, nstep = 10, window = 5, money =
    100)
```

Lastly, `rolltrade` returns the profit, trading signal, and the predicted prices by rolling trading in the whole time period, which is an extension of `trade`. It will repeat the following procedure until the data is used up. Specifically, given a fitted model, it uses this model to forecast future returns based on historical returns, which are obtained by transfering historical prices to historical returns. After that, it again transfers the returns to prices and then generates trading signal based on predicted prices. At last it calculates the profit based on the real price series and the trading signals. The input includes

1. `X`: a one dimensional `numpy` array, which is historical price.
2. `M`: a class from `model.py`, which is a fitted model.

3. `l`: an integer number, which specifies how many data points are used to train for each trading.

4. `nstep`: an positive integer, which specifies how many steps to predict.

5. `window`: an positive integer that specifies the largest length of one trade. If `window`=5, that means there's at most one trade activity within 5 days.

6. `money`: a positive float number, which is the initial wealth. The default number is 1.0.

The output is three one-dimensional numpy array, which are the time series of profit, trading signal, and the predicted prices at different time points. To use these function, we can run

```
# import library
import trading
import model
import numpy as np
import data_processor as dp

# read data and do the preprocessing
data = np.loadtxt("../data/GOOG.csv", delimiter=',')
X = np.array([data[0:100]])
Y = dp.get_return(X)

# initialize the model
lag = 5
sigma = 1.0
intercept = 0.1
phi = np.array([[ 0.04560256],[ 0.0535601 ],[-0.78190871],[ 1.30062633],[
    0.46616754]])
AR_model = AR(lag=lag, phi=phi, sigma=sigma, intercept=intercept)

# solve the model
_, grads = AR_model.loss(Y)
solver = Solver(AR_model, Y, update_rule='sgd_momentum', optim_config={'
    learning_rate': 1e-6,}, num_epochs=10000, batch_size=1,print_every=10)
solver.train()

# get the trading profit, signal and predicted price
profit, signal, out_pred_price = trading.rolltrade(X, M = AR_model, l = 100,
     nstep = 20, window = 5, money = 100)
```

3.7 Option Pricing

The class `OptionPricing` calculates the call option price of an underlying stock based on the Black-Scholes model.

The following parameters are needed to construct an instance of the `OptionPricing` class:

1. `sigma`: the volatility of the underlying stock price, which is the standard deviation of the stock's returns.

2. `K`: the strike price of the option.

3. `T`: the expiry time of the option.

4. `r`: the risk-free interest rate.

5. `Smax`: the maximum stock price we want to consider.

The method `solve_black_scholes` calculates the option pricing of an instance, given the grid size of price and time (`nS` and `nt`) as input parameters. The option price as an function of stock price and time will be stored in the instance.

After running `solve_black_scholes`, we can use `get_option_price` method to get the option price of given underlying stock price `S` and time `t`.

Following is a usage example of the `OptionPricing` class:

```
goog = np.genfromtxt("../data/GOOG.csv", delimiter=",")
sigma = np.std((goog[1:] - goog[:-1]) / goog[:-1])

option_price = OptionPricing(sigma=sigma, T=90, K=800, r=0.005, Smax=1200)
option_price.solve_black_scholes(nS=100, nt=300)
print(option_price.get_option_price(S=810, t=30))
```

3.8 Clustering

We also provide realizations of four clustering methods: K-means, spectral clustering, hierarchical clustering, and Gaussian mixture modeling. These methods are organized by the `Cluster` class, which is initialized by an input data matrix of dimension (`nsample` $\times$ `nfeature`) where `nsample` is the number of data points and `nfeature` is the number of features. An object in the class consists of three attributes:`_X`, which stores the data matrix, `_nsample`, and `_nfeature`. All clustering functions in this class is has the form

```
centroid, labels, clusters =Clustering_method(self, nClusters, maxIter)
```

Here `Clustering_method` is one of `Kmeans`, `H_clustering`, `Spectral`, and `Gaussian_mixture`. In addition, `nClusters` is the number of clusters and `maxIter` is the maximum number of iterations, which has default value 300. The output consists of three items. `centroid`, a `nClusters` $\times$ `_nfeatures` matrix storing the centers of clusters; `labels` is the predicted labels; `clusters` stores the index of data points in each cluster. The following example shows how to use our package to cluster the `S&P 500` dataset. First we import and standardize the data.

```
import numpy as np
# read SP500 data
SP500 = np.genfromtxt('../data/SP500array.csv', delimiter=',')
SP500 = SP500.T
nStock = len(SP500[:,0])
nTime = len(SP500[0,:])

# preprocessing, standardize data
X = np.copy(SP500)
for i in range(nStock):
    X[i,:] = (X[i,:] - np.mean(X[i,:]))/np.std(X[i,:])
```

The we import our package and fit an K-means clustering model on the data with 3 clusters.

```
from src.cluster import Cluster
model = Cluster(X)
# run K-means

import time
```

```
start = time.time()
centroid, labels, clusters = model.kMeans(nClusters = 5)
end = time.time()

print("K-means takes "+str(end-start)+" seconds")
```

The output is

```
K-means takes 0.873986959457 seconds
```

3.9 Model Reduction

When multiple time series are available, we can analysis the data with modell reduciton methods. The model reduction function is provided in `reduction.py`. In this file, we provide a class `Reduction`.

```
class Reduction(object):
    """Callable modal reduction object.
    Example usage:
    xreduction = Reduction(X), X shape [n_features, n_samples], make sure X is zero-mean
    xmean, ux, at, energy_content = xreduction.PCA(n_components=3)
    """
```

The member functions of class `Reduction` consists of Principal Component Analysis (PCA) [2], Independent Component Analysis (ICA) [1], and Dynamic Mode Decomposition (DMD) [3] [4]. The definitions of three functions are declared below:

```
    def PCA(self, n_components=None):
        """
        Principal component analysis (PCA) of data in matrix
        Inputs:
        n_components: integer, number of principal components
        Returns:
        ux: principal components
        at: principal components coefficients
        energy_content: energy content percentage in the principal components
        """

    def ICA(self, n_components, gfunc='logcosh', tol=1e-4, max_iter=200):
        """
        Independent component analysis(ICA) of data in matrix X
        Inputs:
        n_components: integer, number of independent components
        gfunc: string, 'logcosh' or 'exp', default 'logcosh', Non-gaussian function
        tol: float, tolerance of iteration, default 1e-4
        max_iter: integer, maximum iteration steps, default 200
        Returns:
        Ex: array, mean of data
        T: array [n_features, n_features], whitening matrix, st, xtilde = Tx
        A: array [n_features, n_components], mixing matrix, st, xtilde = As
        W: array [n_components, n_features], orthogonal rows, unmixing matrix, st, W = inv(A), s = W*xtilde
```

```
        S: array, [n_components, n_samples], source data, st, S = W*Xtilde
        """

    def DMD(self, n_components=None):
        """
        Dynamic mode decomposition(DMD) of time series data x(k), find
    square
        matrix A such that x(k+1) = Ax(k). Find eigendecomposition of A, and
        corresponding DMD modes, and DMD eigenvalues.
        """
```

To get a better sense of how to analyse real data with this class. Now we will give an example using `S&P 500`.

```
import numpy as np
import reduction

# preprocessing, subtract each stock price mean and normalize by std
x = np.copy(SP500)
for i in range(nStock):
    x[i,:] = (x[i,:] - np.mean(x[i,:]))/np.std(x[i,:])

# read SP500 data
SP500 = np.genfromtxt('../data/SP500array.csv', delimiter=',')
SP500 = SP500.T

# call and test PCA
n_components = 5
SPreduction = reduction.Reduction(x)
ux, at, energy_content = SPreduction.PCA(n_components)

# call and test ICA
n_components = 10
SPreduction = reduction.Reduction(x)
Ex, T, A, W, S = SPreduction.ICA(n_components)

# call and test DMD
n_components = 20
SPreduction = reduction.Reduction(x)
evals, modes, energy_content = SPreduction.DMD(n_components)
```

References

[1] Aapo Hyvärinen and Erkki Oja. Independent component analysis: algorithms and applications. *Neural networks*, 13(4):411–430, 2000.

[2] Ian Jolliffe. *Principal component analysis*. Wiley Online Library, 2002.

[3] Clarence W Rowley, Igor Mezić, Shervin Bagheri, Philipp Schlatter, and Dan S Henningson. Spectral analysis of nonlinear flows. *Journal of fluid mechanics*, 641:115–127, 2009.

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