Lecture 4. Autoregressive models

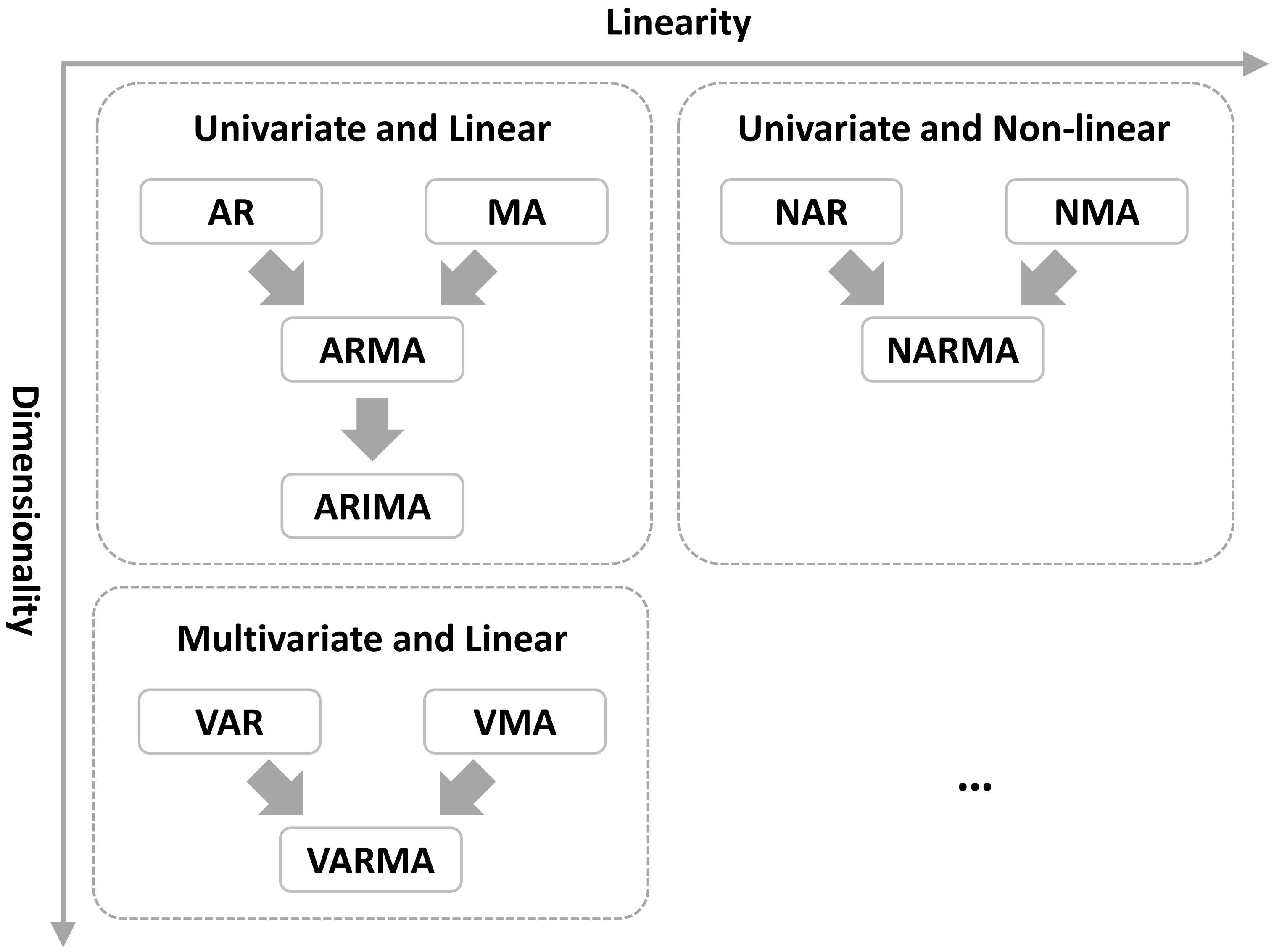
Alexander Zhigalov / Dept. of CS, University of Helsinki and Dept. of NBE, Aalto University

Outline / overview

- **Section 1.** Models hierarchy
- **Section 2.** Autoregressive (AR) model
- **Section 3.** Moving average (MA) model
- **Section 4.** Autoregressive moving average (ARMA) model
- **Section 5.** Estimation of power spectrum using AR model

NOTE: Prepare one/two questions about the lectures material

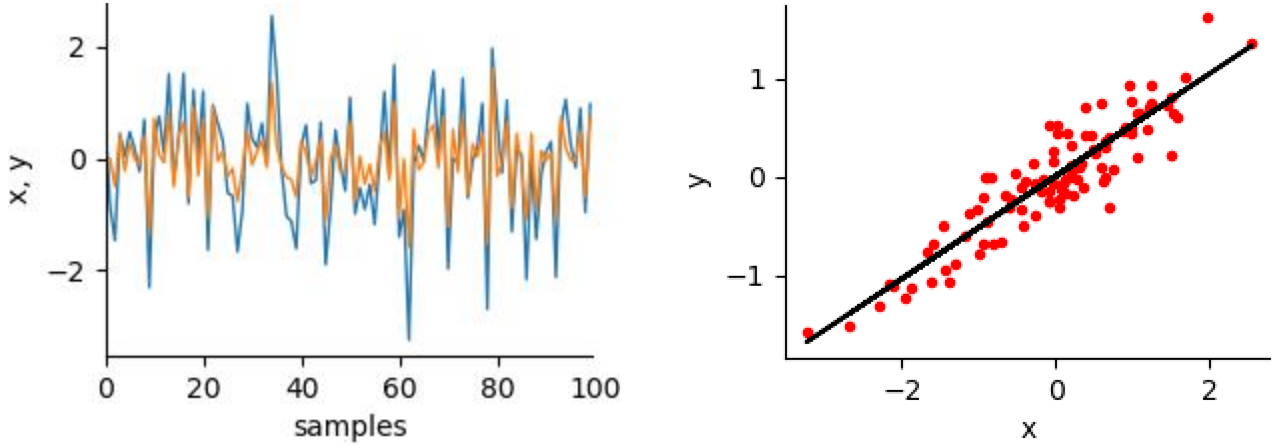
Section 1. Models hierarchy



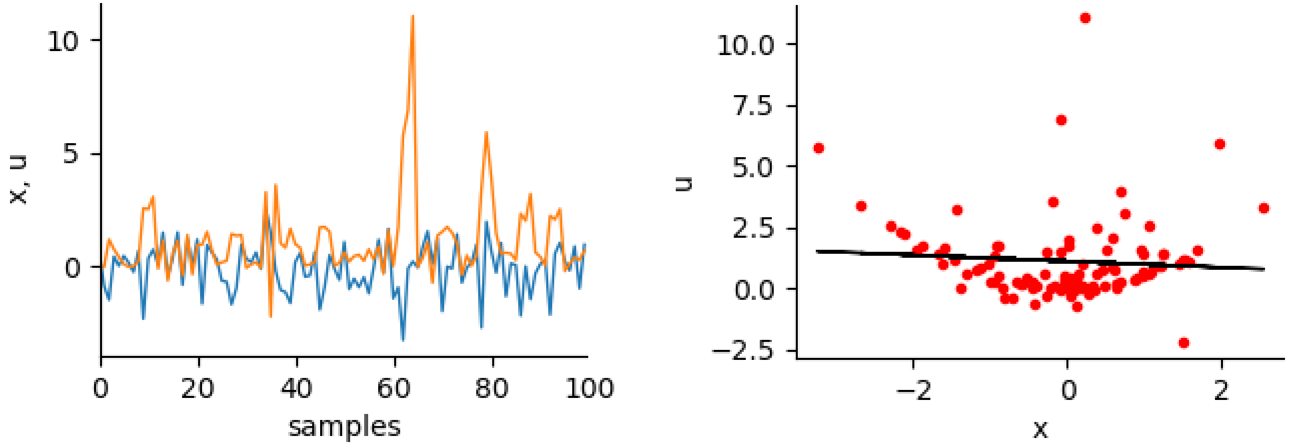
To understand the complicated methods, we first need to understand the basic concepts

Linear vs. non-linear temporal dependency

```
# random data
x = np.random.randn(N)

# linear temporal dependency
y = np.zeros(N)
for i in range(2, N):
  y[i] = 0.5 * x[i] -
         0.2 * x[i-1] +
         0.1 * x[i-2]
```



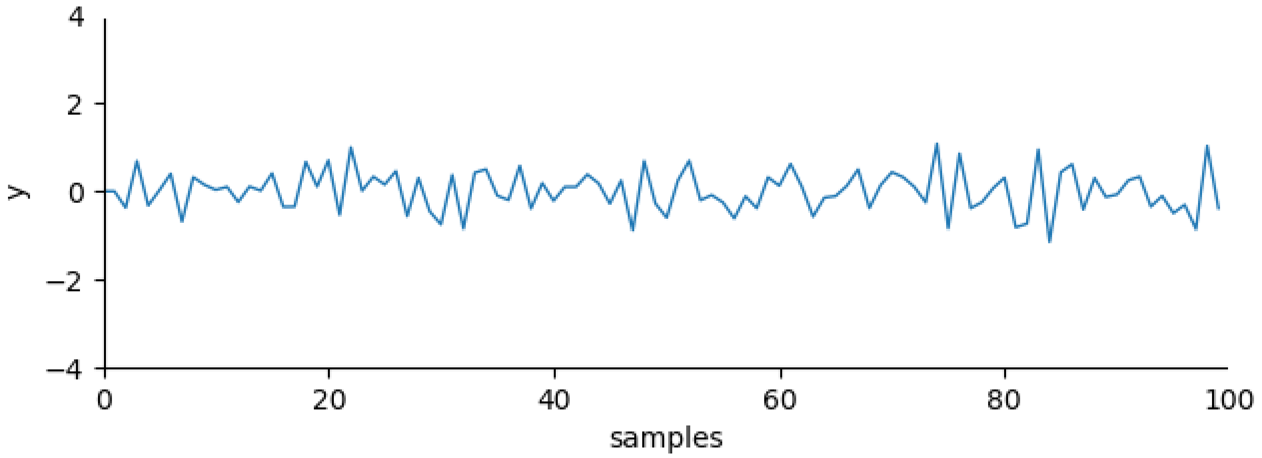

```
# non-linear temporal dependency
u = np.zeros(N)
for i in range(2, N):
  u[i] = 0.5 * x[i] ** 2 -
         0.2 * x[i-1] ** 3 +
         0.1 * x[i-2] ** 4
```



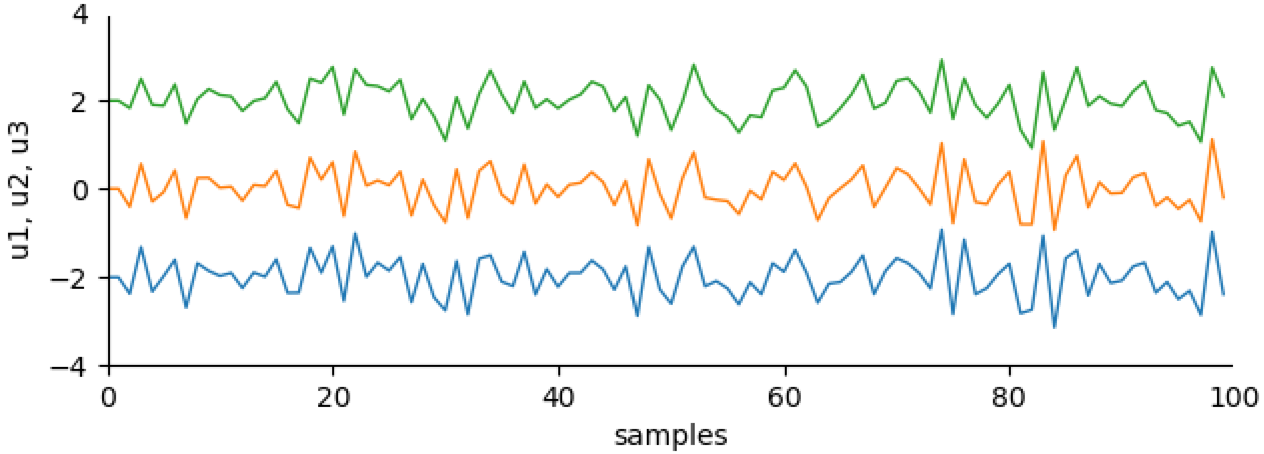
See, “L04_linear_vs_nonlinear.py”

Univariate vs. multivariate timeseries

```
# random data
x = np.random.randn(N)

# univariate time series
y = np.zeros(N)
for i in range(2, N):
  y[i] = 0.5 * x[i] -
         0.2 * x[i-1] +
         0.1 * x[i-2]
```




```
# multivariate time series
u = np.zeros([N, 3])
for i in range(2, N):
  u[i, 0] = 0.5 * x[i]
  u[i, 1] = 0.5 * x[i] -
            0.2 * x[i-1]
  u[i, 2] = 0.5 * x[i] -
            0.2 * x[i-1] +
            0.1 * x[i-2]
```



See, “L04_univariate_vs_multivariate.py”

Section 2. Autoregressive (AR) model

AR model and its parameters

An autoregressive (AR) model is a representation of a type of random process.

The autoregressive model specifies that the output variable depends linearly on its own **previous values** and on a **stochastic term**.

https://en.wikipedia.org/wiki/Autoregressive_model

The notation **AR**(*p*) indicates an autoregressive model of order *p*. The **AR**(*p*) model is defined as

convolution

$$X_n = c + \underbrace{\sum_{i=1}^{p} a_i X_{n-i}} + e_n$$

```
x[n] = c + np.sum(a * x[np.arrange((n-1),(n-p-1),-1)]) + e[n]
```

where a_i are the parameters of the model, c is a constant, and e_t is white noise.

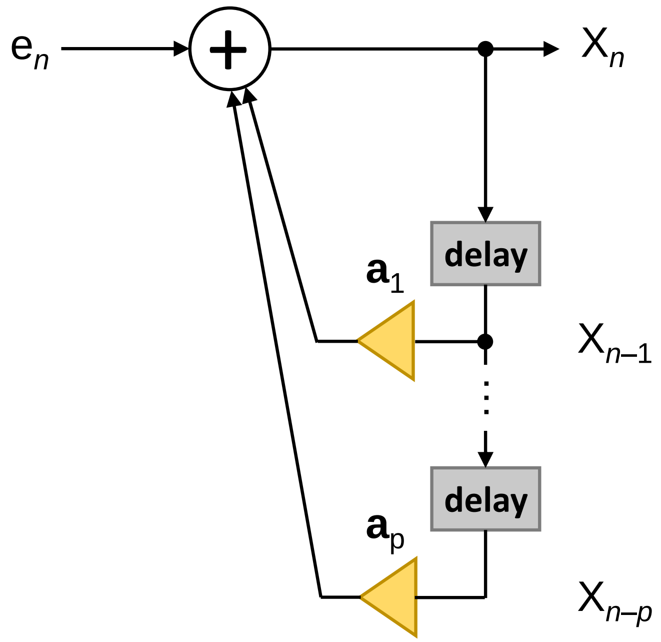
See, “L04_ar_python_equation.py”

Indexing in Python

```
# array of items
X = np.array([1, 2, 3, 4, 5])
N = len(X)
p = 2

# loop
for n in range(p, N):
  print(X[(n-1):(n-p-1):-1])

# output
[]
[3, 2]
[4, 3]
```

```
# array of items
X = np.array([1, 2, 3, 4, 5])
N = len(X)
p = 2

# loop
for n in range(p, N):
  print(X[np.arange((n-1),(n-p-1),-1)])

# output
[2, 1]
[3, 2]
[4, 3]
```

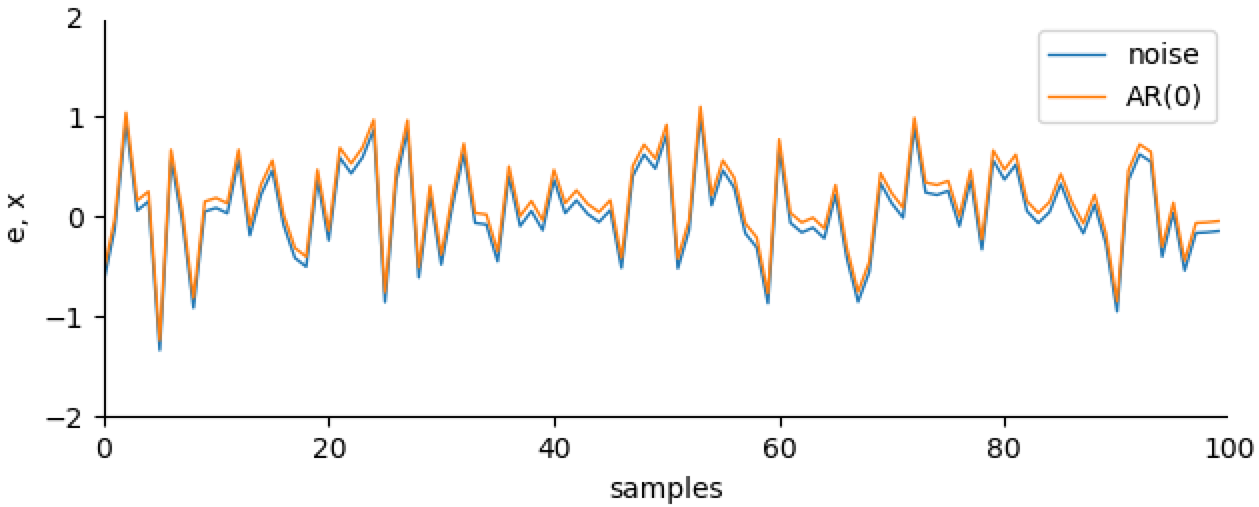
Graphical representation of AR model



AR time series (1/3)

The simplest AR process is AR(0), which has no dependence between the terms. Only the error/innovation/noise term contributes to the output of the process.

```
# gaussian noise
e = np.random.randn(N)

# AR model
a = []
p = len(a)
x = np.zeros(N)
for i in range(p, N):
  x[i] = 0.1 + e[i]
```



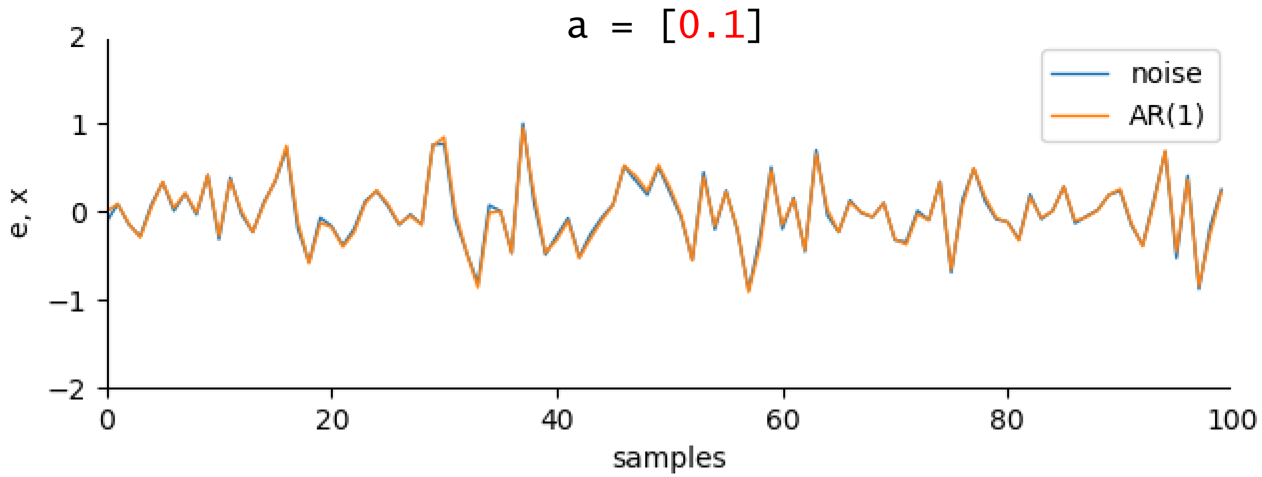
See, “L04_graph_ar_0_process.py”

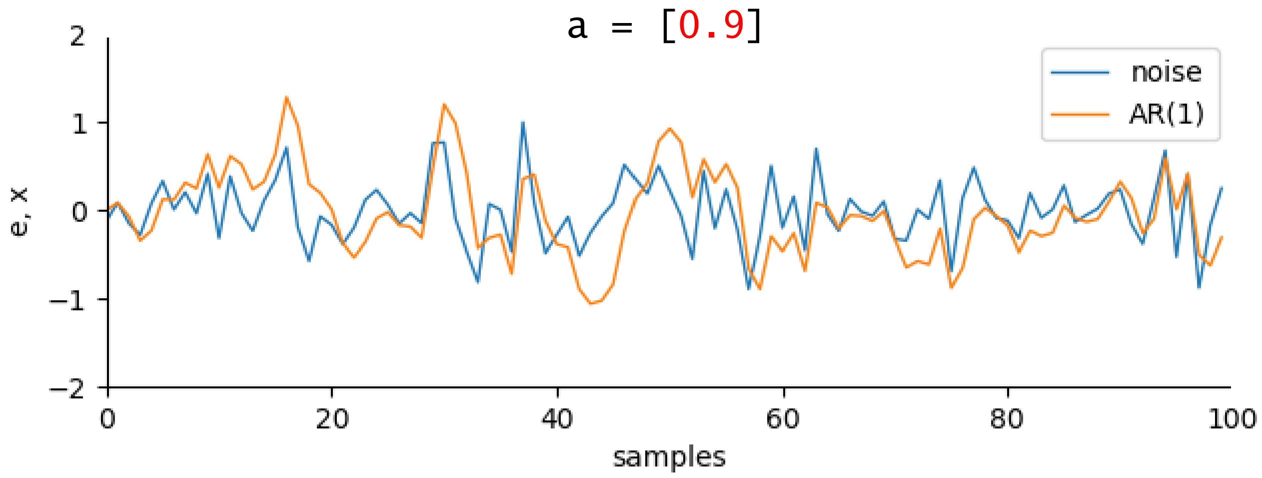
AR time series (2/3)

For an AR(1) process with a positive a_1, only the previous term in the process and the noise term contribute to the output.

If a_1 is close to 0, then the process still looks like white noise, but as a_1 approaches 1, the output gets a larger contribution from the previous term relative to the noise.

```
# gaussian noise
e = np.random.randn(N)

# AR model
a = [0.5]
p = len(a)
x = np.zeros(N)
for i in range(p, N):
  x[i] = a[0] * x[i-1] + e[i]
```





See, “L04_graph_ar_1_process.py”

AR time series (3/3)

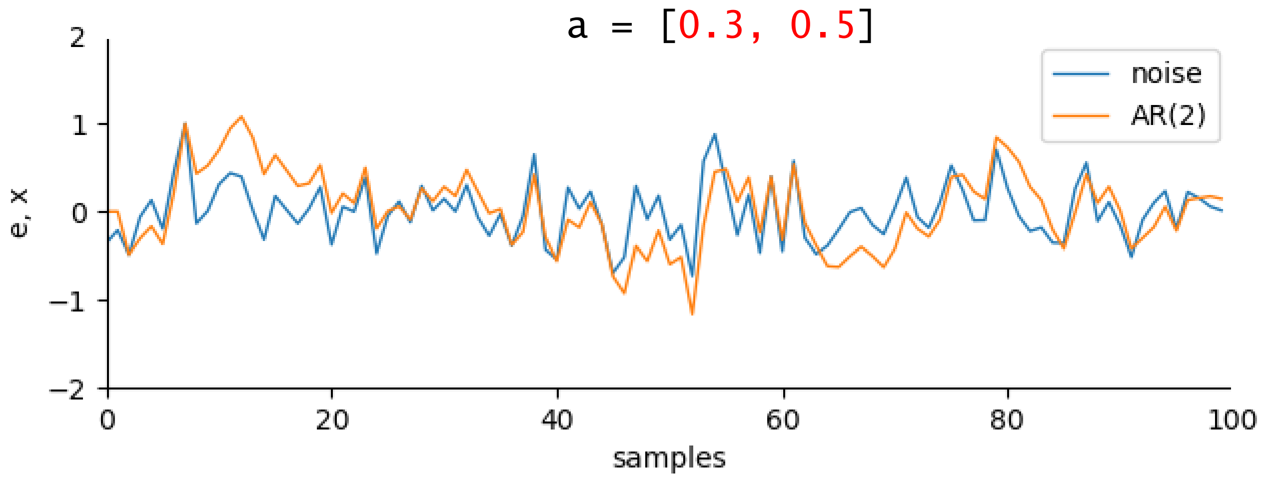
For an AR(2) process, the previous two terms and the noise term contribute to the output.

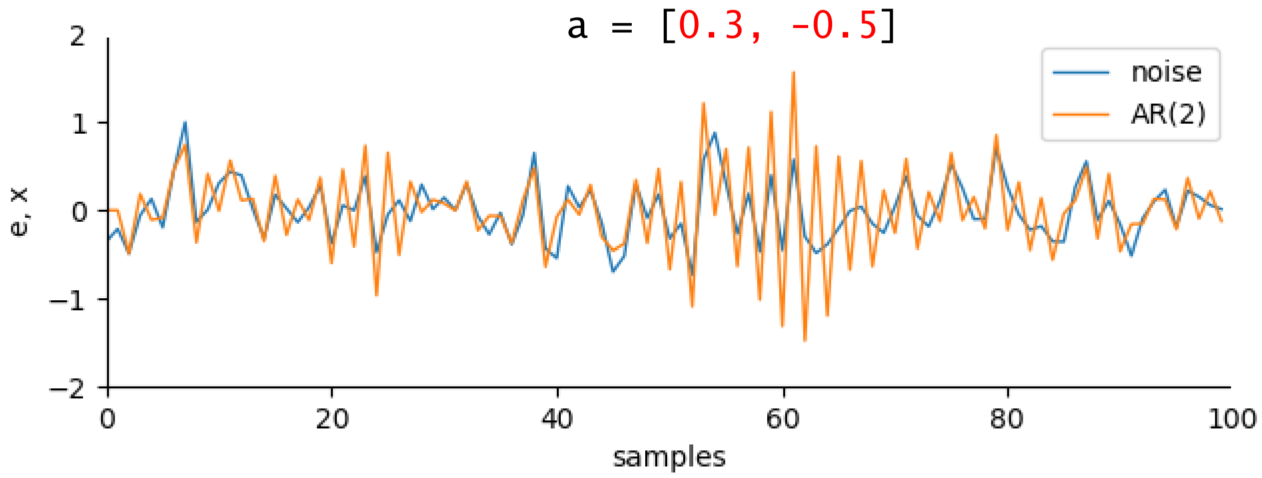
If both a_1 and a_2 are positive, the output will resemble a low pass filter, with the high frequency part of the noise decreased.

If a_1 is positive while a_2 is negative, then the process favors changes in sign between terms of the process. The output oscillates.

```
# gaussian noise
e = np.random.randn(N)

# AR model
a = [0.5, 0.5]
p = len(a)
x = np.zeros(N)
for i in range(p, N):
  x[i] = a[0]*x[i-2] + a[1]*x[i-1] + e[i]
```

See, “L04_graph_ar_2_process.py”



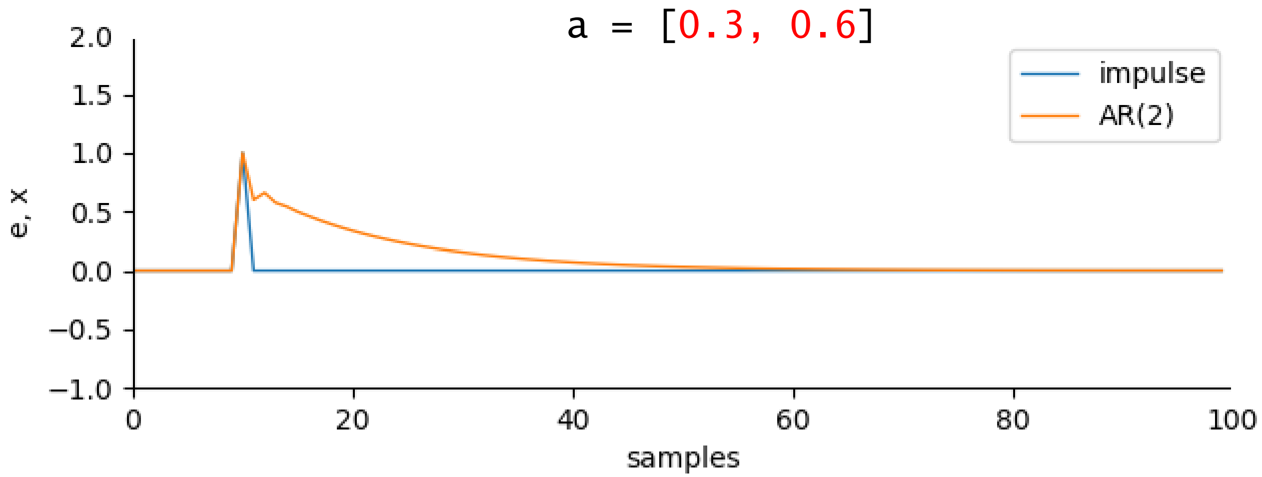


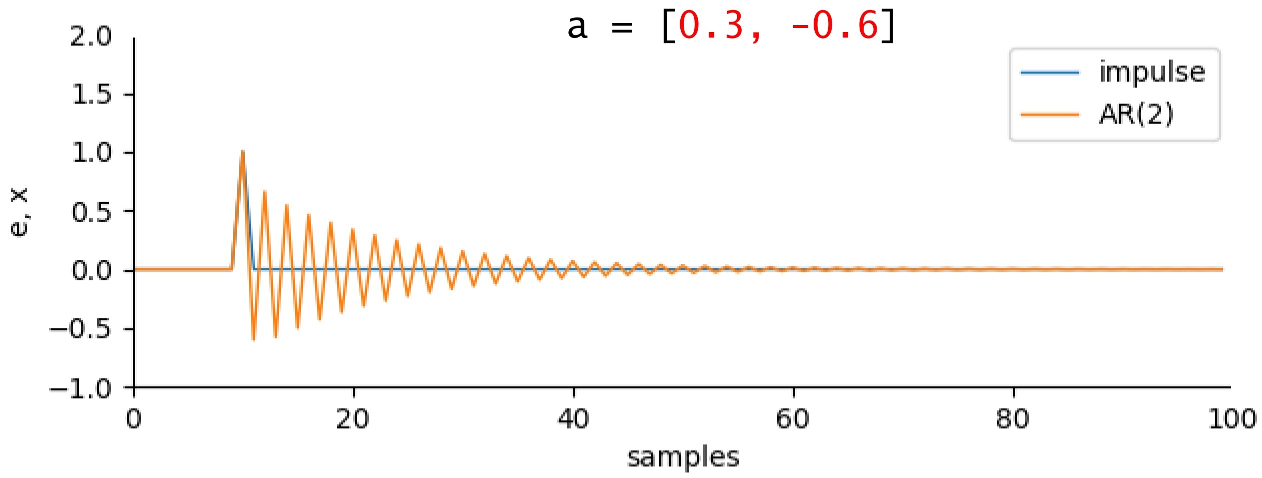
AR impulse response

The impulse response of a system is the change in an evolving variable in response to a change in the value of a shock term *k* periods earlier, as a function of *k*.

An autoregressive model can thus be viewed as the output of an all-pole **infinite impulse response** filter.

```
# impulse
e = np.zeros(N)
e[10] = 1

# AR model
a = [0.5, 0.5]
p = len(a)
x = np.zeros(N)
for i in range(p, N):
  x[i] = a[0]*x[i-2] + a[1]*x[i-1] + e[i]
```



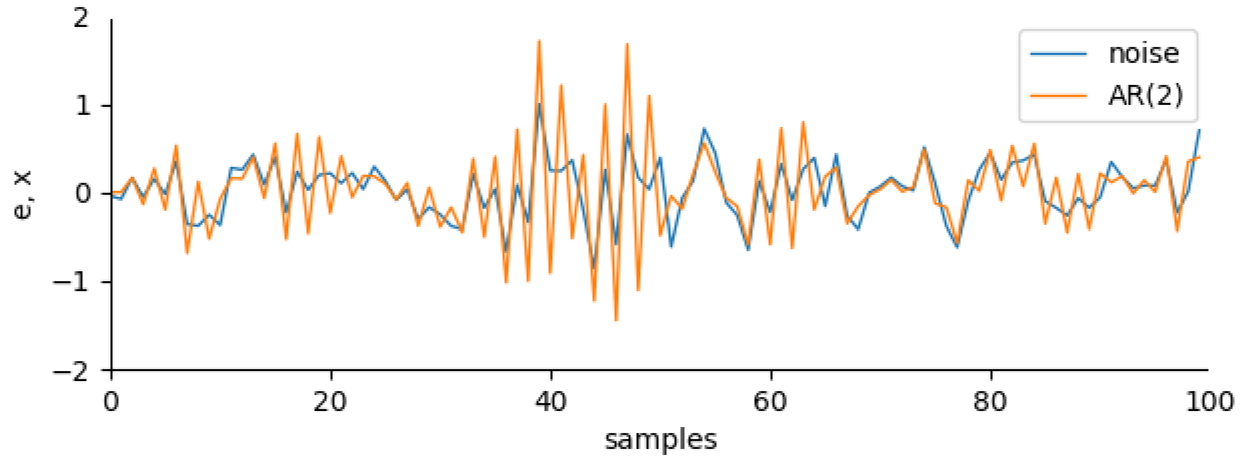


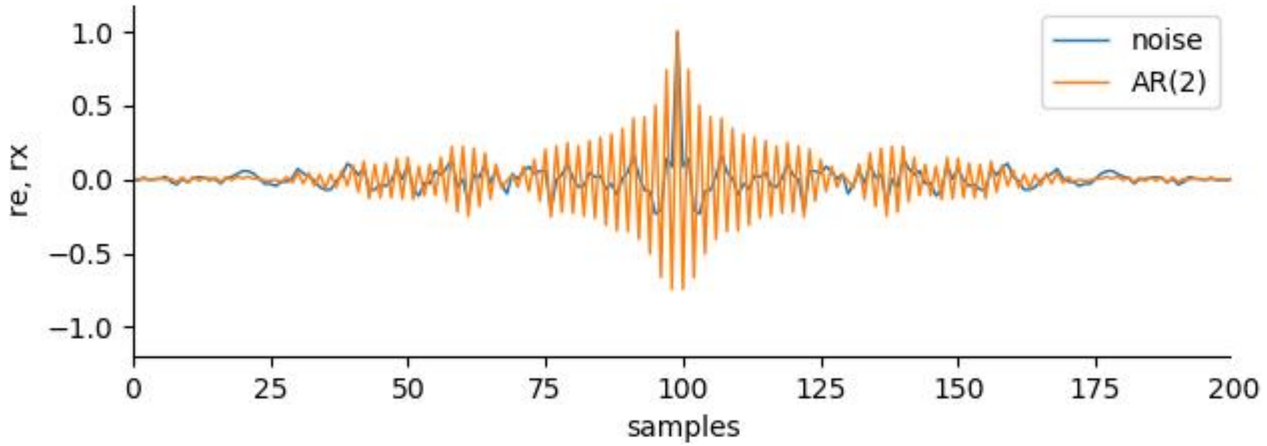
See, “L04_ar_impulse_response.py”

AR autocorrelation function

The autocorrelation function of an AR(*p*) process is a sum of decaying exponentials.

```
# gaussian noise
e = np.random.randn(N)

# AR model
a = [0.3, -0.5]
p = len(a)
x = np.zeros(N)
for i in range(p, N):
  x[i] = a[0]*x[i-2] + a[1]*x[i-1] + e[i]

# autocorrelation function
re = signal.correlate(e, e)
rx = signal.correlate(x, x)
```





See, “L04_ar_acf.py”

AR parameters estimation

We assume that the noise is Gaussian, and for known output **y** and model order **p**, we estimate AR coefficients.

Algorithms for computing the least squares AR model,

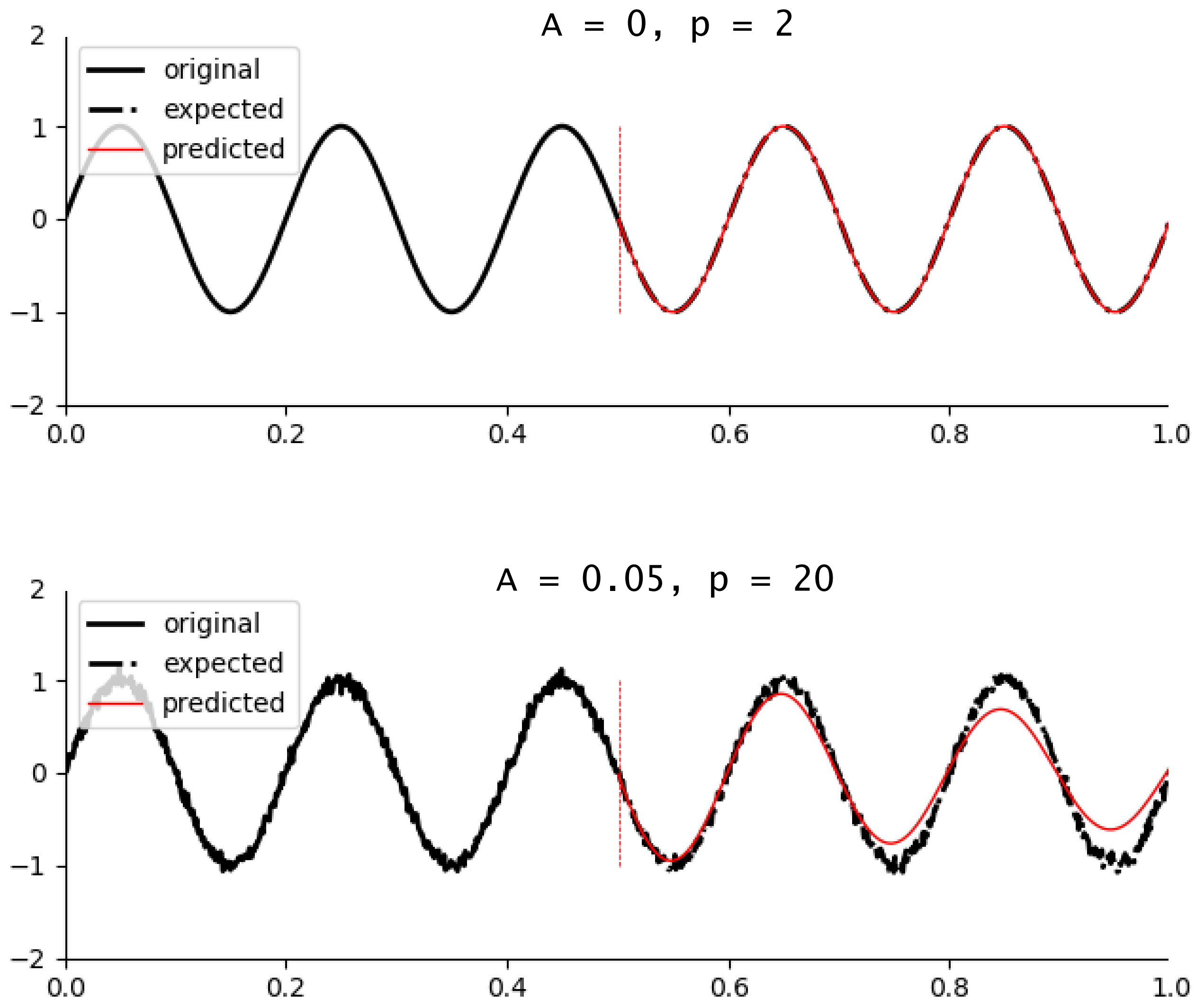
- **Burg's lattice-based method**. Solves the lattice filter equations using the harmonic mean of forward and backward squared prediction errors.
- **Forward-backward approach**. Minimizes the sum of a least-squares criterion for a forward model, and the analogous criterion for a time-reversed model.
- **Geometric lattice approach**. Similar to Burg's method, but uses the geometric mean instead of the harmonic mean during minimization.
- **Least-squares approach**. Minimizes the standard sum of squared forward-prediction errors.
- **Yule-Walker approach**. Solves the Yule-Walker equations, formed from sample covariances.

https://se.mathworks.com/help/ident/ref/ar.html

AR n-step-ahead forecasting

Once the parameters of the autoregression have been estimated, the autoregression can be used to forecast an arbitrary number of periods into the future.

```
# signal
A = 0.05
X = np.sin(2 * np.pi * 5 * t) +
    A * np.random.randn(N)

# split dataset
x = X[:L] # data to fit
y = X[L:] # data to test

# autoregressive model
p = 20 # AR model order
model = AR(x)
model_fit = model.fit(maxlag=p)
u = model_fit.predict(start, stop)
```



See, “L04_ar_forecasting.py”

Section 3. Moving average (MA) model

MA model and its parameters (1/2)

The moving-average (MA) model is a common approach for modeling univariate time series.

The moving-average model specifies that the output variable depends linearly on the **current** and various **past values** of a stochastic term.

https://en.wikipedia.org/wiki/Moving-average_model

The notation **MA**(q) refers to the moving average model of order q:

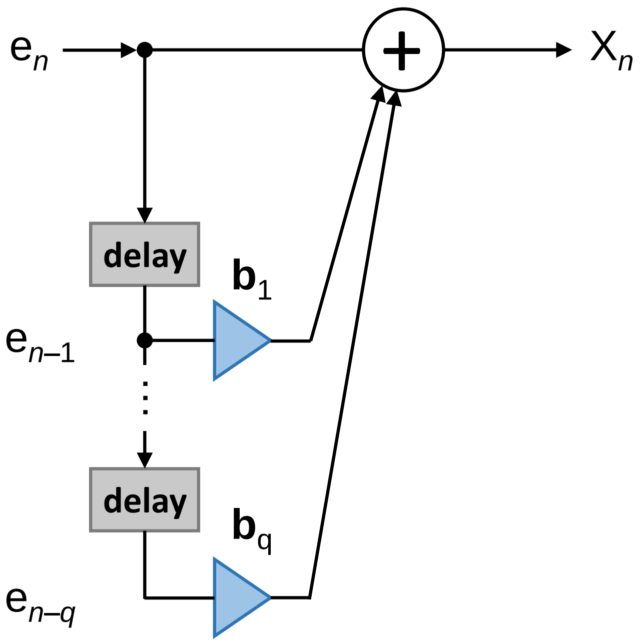
$$X_n = \mu + \sum_{i=1}^{q} b_i\, e_{n-i} + e_n$$

```
x[n] = mu + np.sum(b * e[np.arange((n-1),(n-p-1),-1)]) + e[n]
```

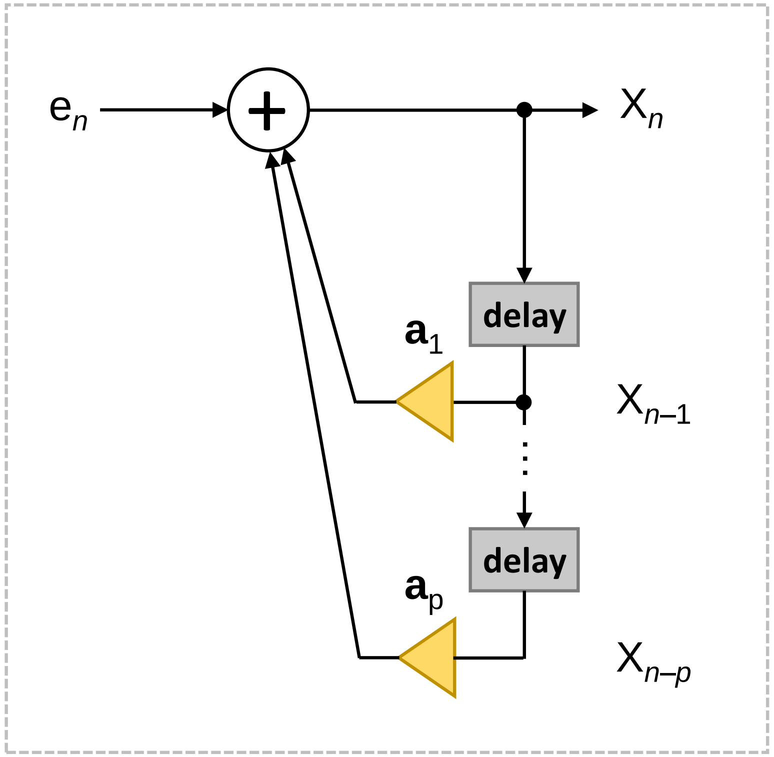
where b_i are the parameters of the model, μ is the expectation of X_n (often assumed to equal 0), and e_t is white noise.

See, “L04_ma_python_equation.py”

Graphical representation of MA model



AR model



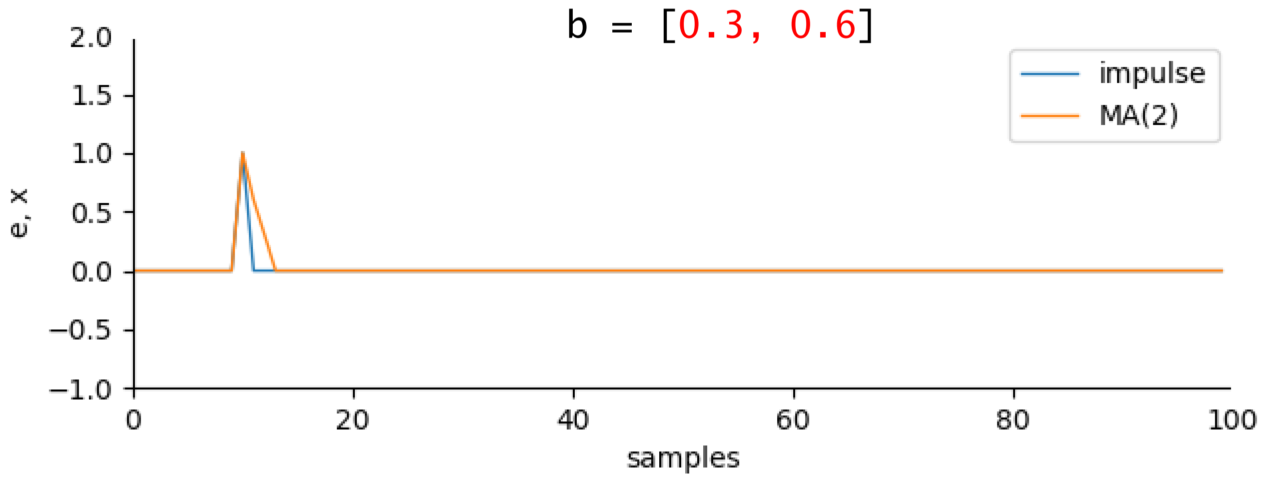
MA impulse response

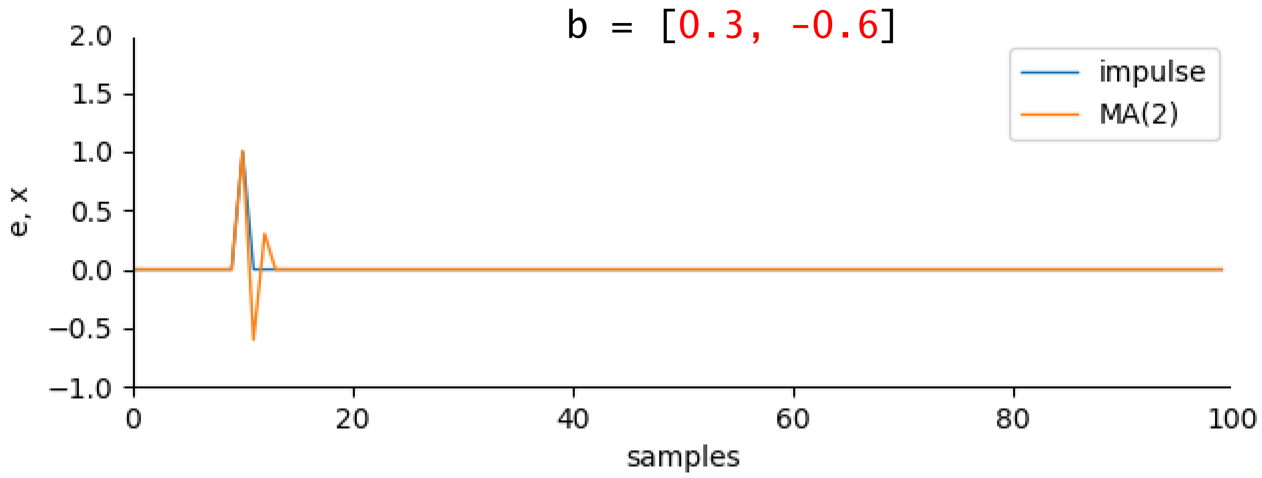
The moving-average model is essentially a **finite impulse response** filter applied to white noise, with some additional interpretation placed on it.

In an MA process, a one-time shock affects values of the evolving variable non-infinitely far into the future.

```
# impulse
e = np.zeros(N)
e[10] = 1

# AR model
b = [0.5, 0.5]
p = len(a)
x = np.zeros(N)
for i in range(p, N):
  x[i] = b[0]*e[i-2] + b[1]*e[i-1] + e[i]
```

See, "L04_ma_impulse_response.py"





MA parameters estimation

Fitting the MA estimates is more complicated than with autoregressive models because the lagged error terms are not observable. This means that iterative non-linear fitting procedures need to be used in place of linear least squares.

https://en.wikipedia.org/wiki/Moving-average_model

Section 4. Autoregressive moving average (ARMA) model

ARMA model and its parameters

Autoregressive–moving-average (ARMA) models provide a parsimonious description of a stationary stochastic process in terms of two polynomials, one for the **autoregression** and the second for the **moving average**.

The model consists of two parts, an autoregressive (AR) part and a moving average (MA) part. The AR part involves regressing the variable on its own past values. The MA part involves modeling the error term as a linear combination of error terms occurring contemporaneously and at various times in the past.

https://en.wikipedia.org/wiki/Autoregressive-moving-average_model

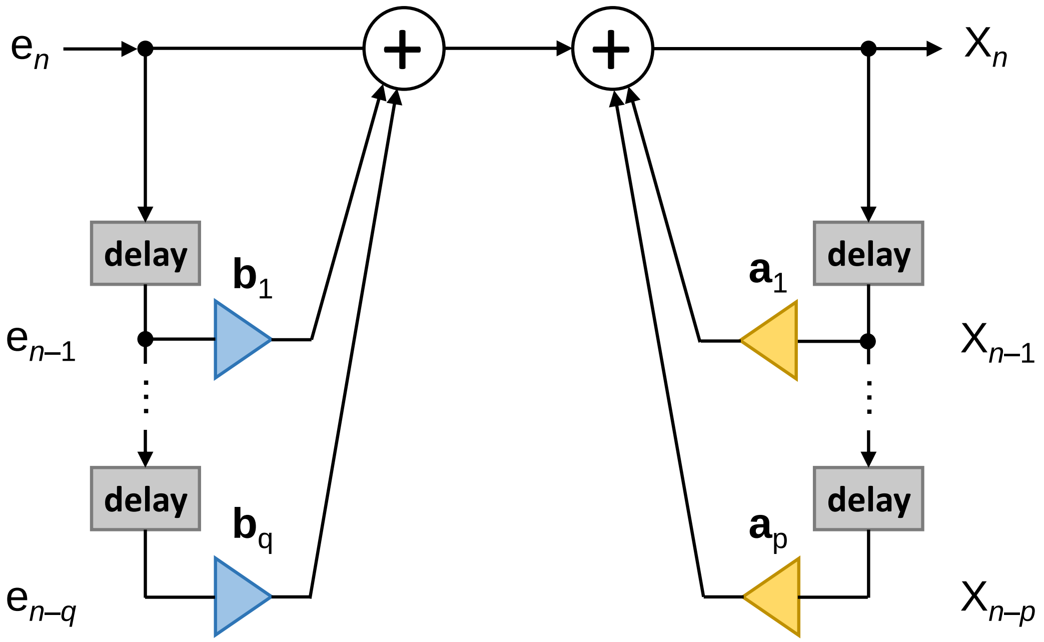
The notation **ARMA**(*p*, *q*) refers to the model with *p* autoregressive terms and *q* moving-average terms

$$X_n = c + e_n + \sum_{i=1}^{p} a_i X_{n-i} + \sum_{i=1}^{q} b_i e_{n-i}$$

```
X[n] = c + e[n] + np.sum(a * X[np.arange((n-1),(n-p-1),-1)]) +
                  np.sum(b * e[np.arange((n-1),(n-q-1),-1)])
```

where a_i and b_i are the parameters of the model, c is a constant, and e_t is white noise.

Graphical representation of ARMA model

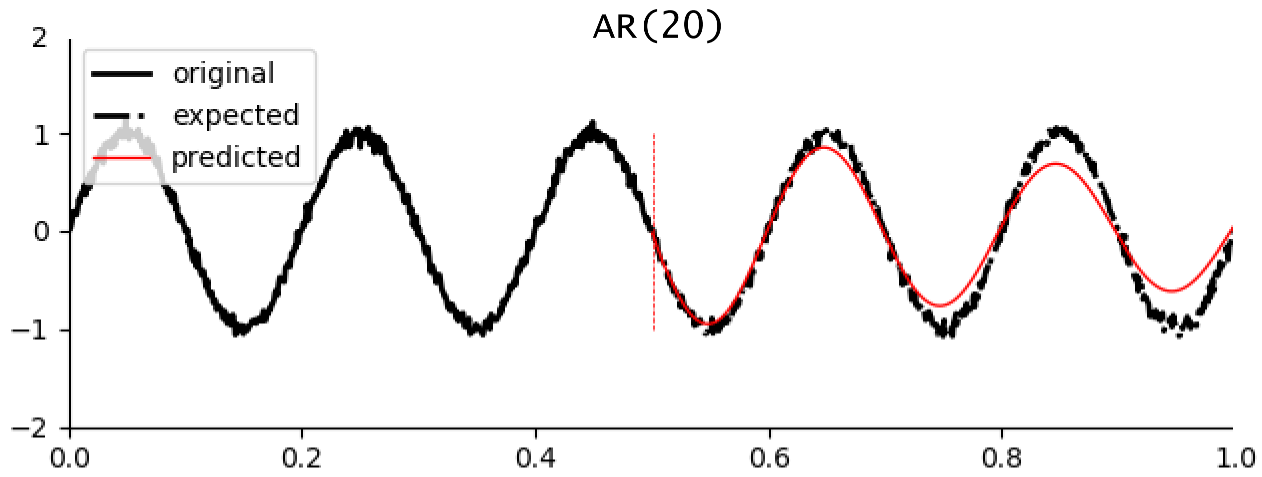


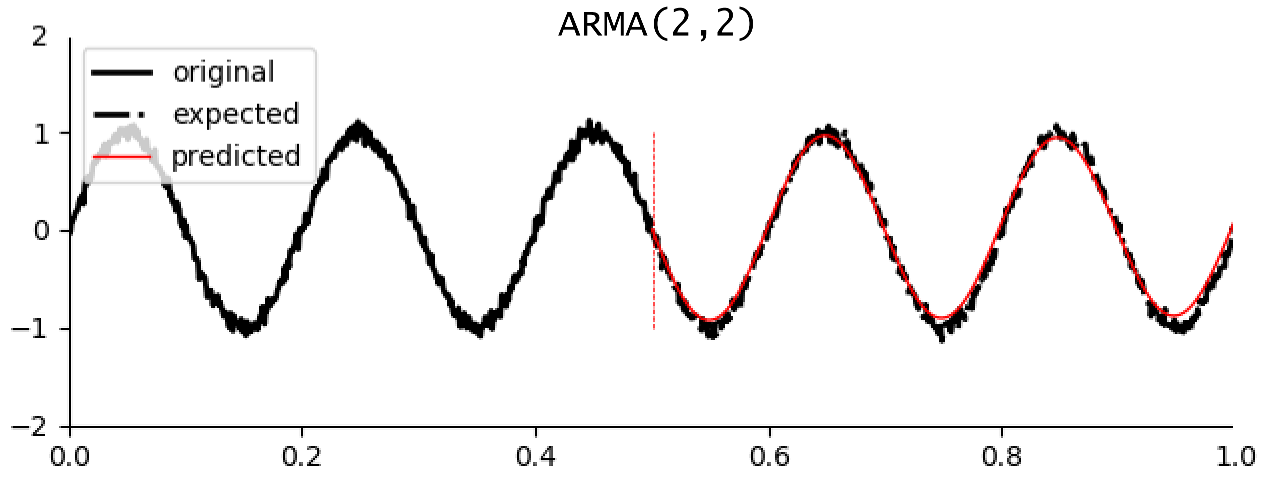
ARMA n-step-ahead forecasting

ARMA models in general cannot be, after choosing p and q, fitted by least squares regression to find the values of the parameters which minimize the error term.

```
# signal
A = 0.05
X = np.sin(2 * np.pi * 5 * t) +
    A * np.random.randn(N)

# split dataset
x = X[:L] # data to fit
y = X[L:] # data to test

# autoregressive model
p = 2 # AR model order
q = 2 # MA model order
model = ARMA(x, (p, q))
model_fit = model.fit()
u = model_fit.predict(start, stop)
```

See, “L04_arma_forecasting.py”

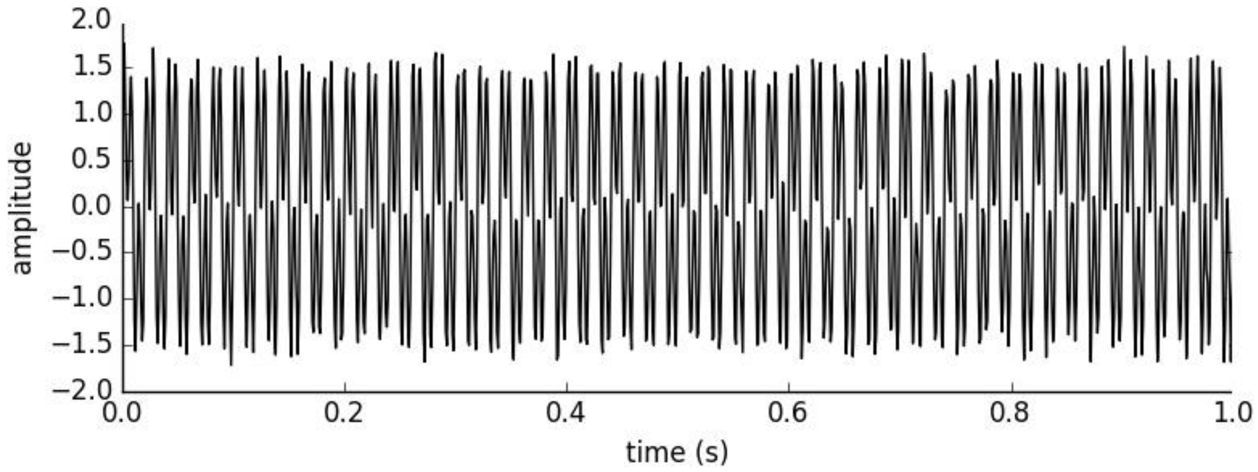




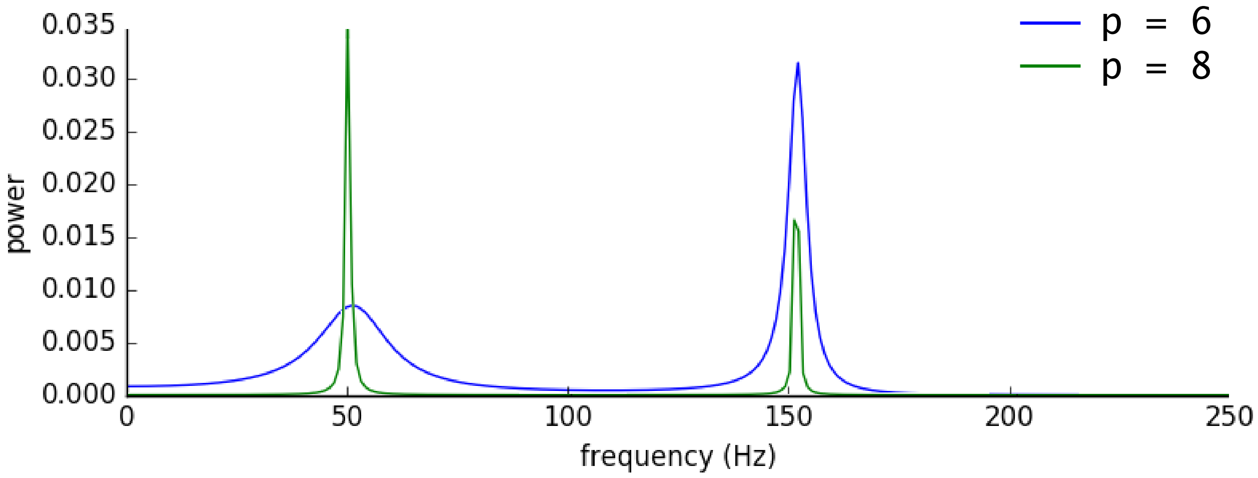
Section 5. Estimation of power spectrum

Burg estimation of power spectrum

Estimate AR coefficients and then compute Fourier transform of these coefficients




```
from spectrum import arburg, arma2psd

# estimate AR model
p = 6
AR, P, k = arburg(x, p)

# compute power spectrum
PSD = arma2psd(AR, NFFT=NFFT)
PSD = PSD / np.sum(PSD)
```



See, “L04_ar_spectrum_estimation.py”

http://thomas-cokelaer.info/software/spectrum/html/user/ref_param.html

Literature

- **Python programming language**

- http://www.scipy-lectures.org/, see “materials/L02_ScipyLectures.pdf”

- **Data analysis**

- Cohen M., “**Analyzing Neural Time Series Data: Theory and Practice**”