Lecture 11. Course summary

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

Outline / overview

- **Section 1.** Signals
- **Section 2.** Models
- **Section 3.** Transformations
- **Section 4.** Filtering
- **Section 5.** Interactions
- **Section 6.** Clustering
- **Section 7.** Classification

Flowchart

Uni-variate time series

1 **Signals**
(deterministic / stochastic)

2 **Models**
(autoregressive, autocorrelation)

3 **Transforms**
(Fourier transform, convolution)

4 **Filtering**
(FIR/IIR filters / wavelets)

Multi-variate time series

5 **Interactions**

6 **Clustering**

7 **Classification**

Section 1. Signals

Signals

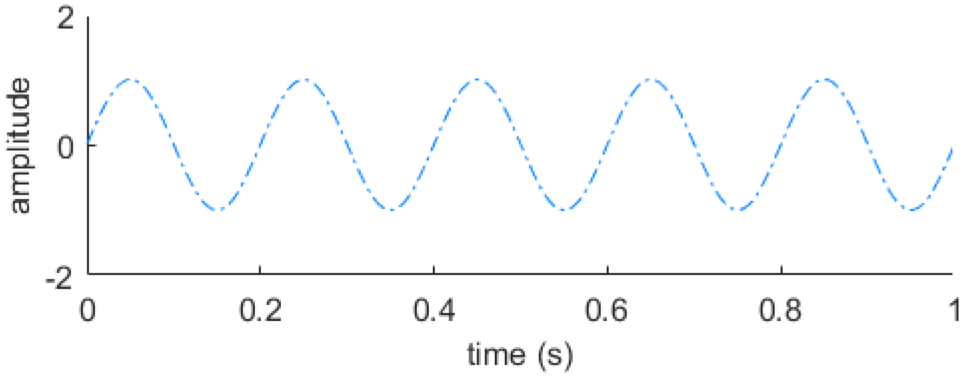
- Sinusoid time series
- Periodic and non-periodic time series
- Random time series

Sinusoid time series

Parameters of sinusoid functions

A – amplitude of signal
f – frequency of signal
phi – phase of signal

```
fs = 1000  # Hz
T = 1      # seconds
N = T * fs # duration or length

# init
t = np.linspace(0, T, N)

# function
X = A * np.sin(2 * np.pi * t * f + phi)
```

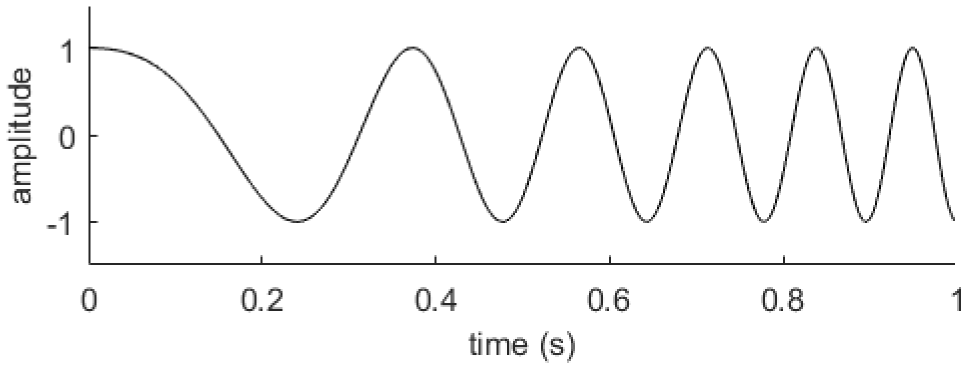



```
A = 1
f = 5
phi = 0
```

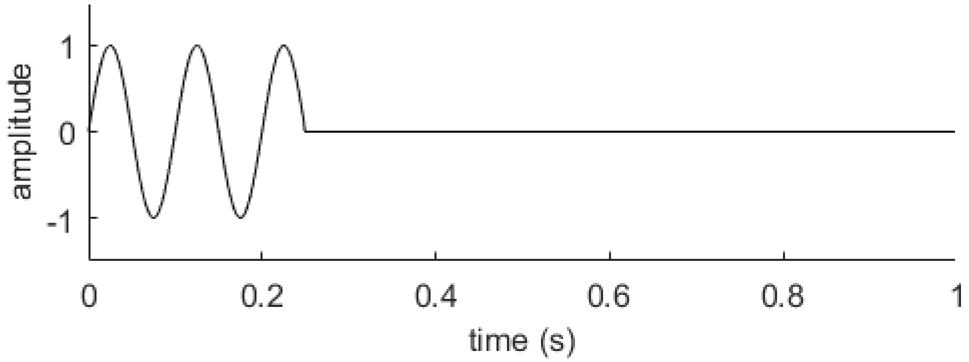
See, “L03_periodic_signal.py”

Periodic and non-periodic time series

```
# chirp signal
f0 = 1
f1 = 10
t1 = T
y = signal.chirp(t, f0, t1, f1)
```



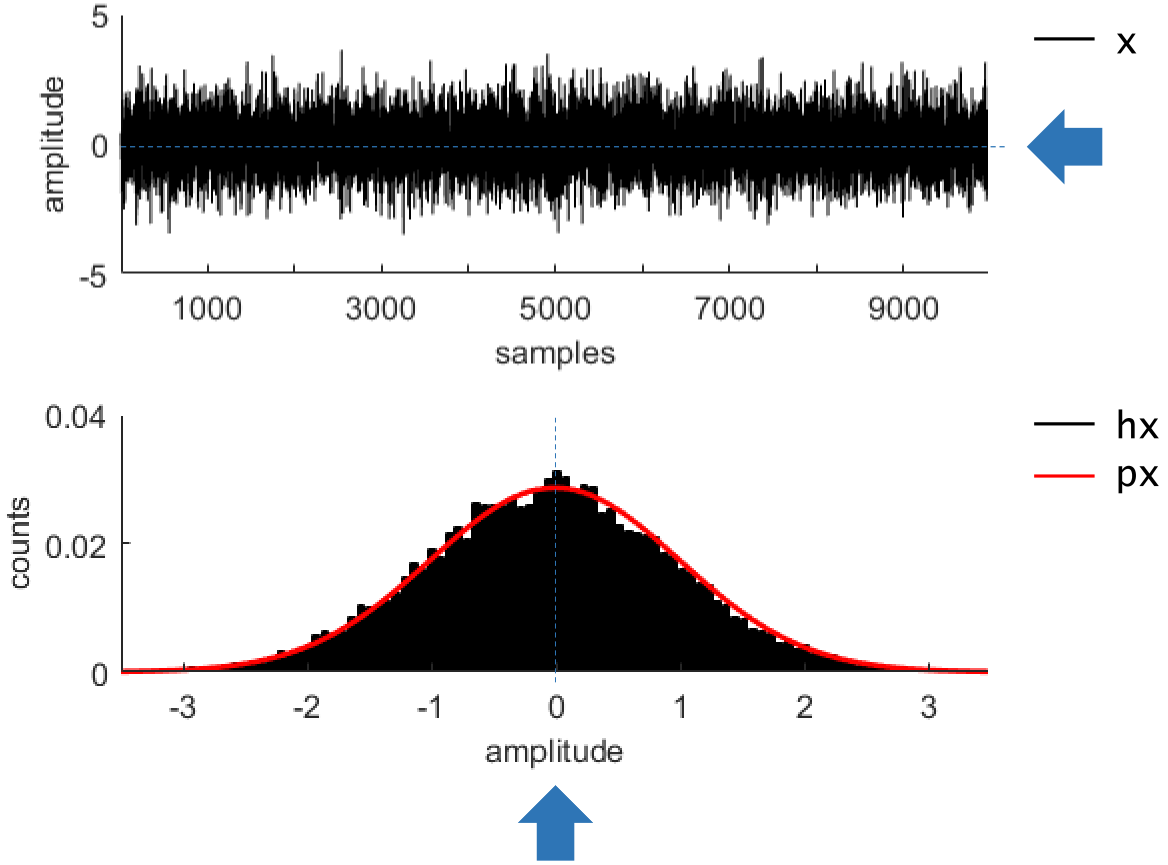

```
# non-periodic signal
f0 = 10
u = np.sin(2 * np.pi * f0 * t)
u[int(N/4):] = 0
```



See, “L03_non_periodic_signal.py”

Random time series

```
import numpy as np
from scipy import signal

# generate gaussian noise
N = 10000
x = np.random.randn(N) # mu = 0, std = 1

# histogram
bx = np.linspace(xmin, xmax, 100)
hx, bx = np.histogram(x, bx)

# pdf
mu, std = norm.fit(x)
px = norm.pdf(bx, mu, std)
```



See, “L03_noise.py”

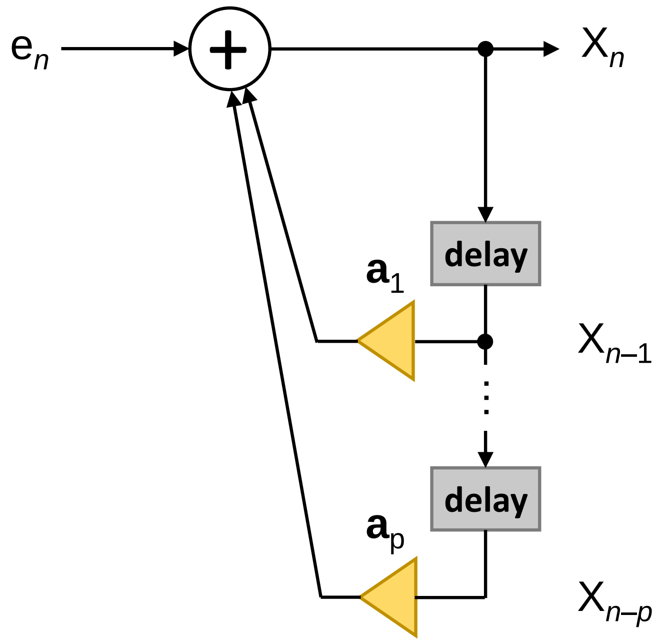
Section 2. **Models**

Models

- Autoregressive models
- Stochastic models (**?**)

https://machinelearningmastery.com/autoregression-models-time-series-forecasting-python/

Graphical representation of AR model



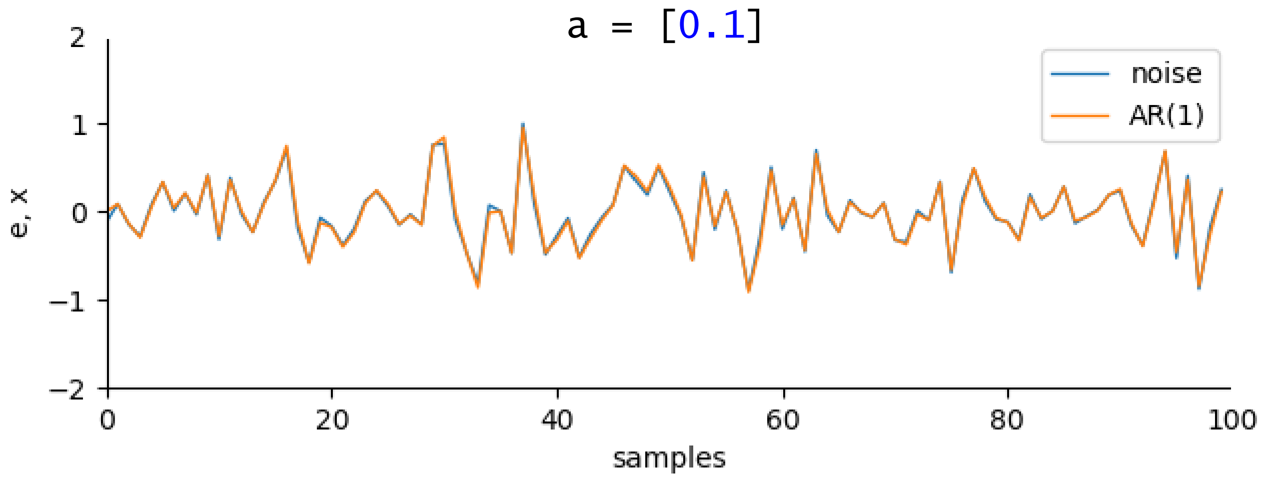
https://machinelearningmastery.com/autoregression-models-time-series-forecasting-python/

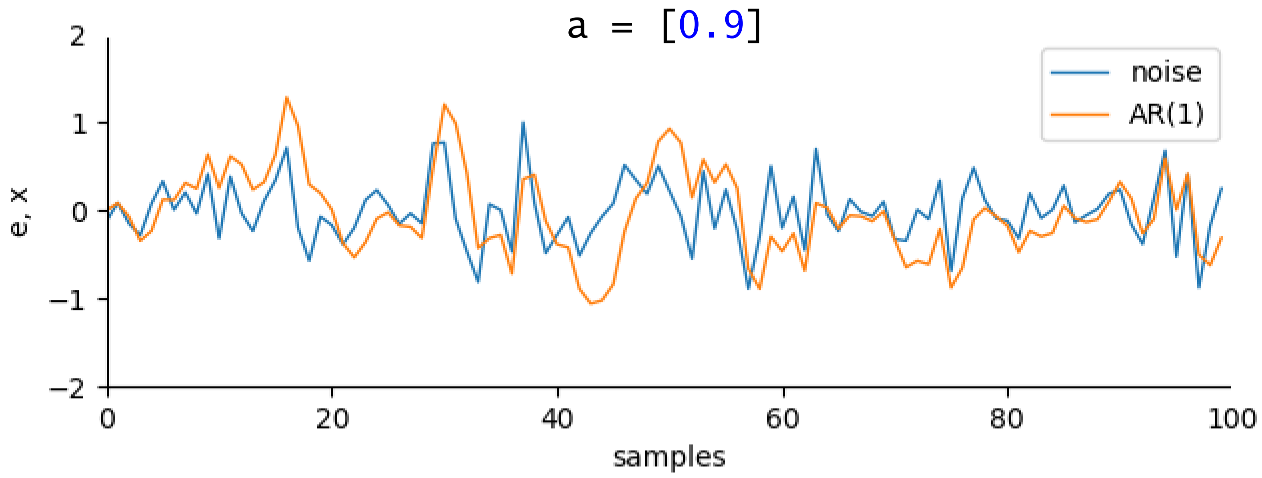
AR time series

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.

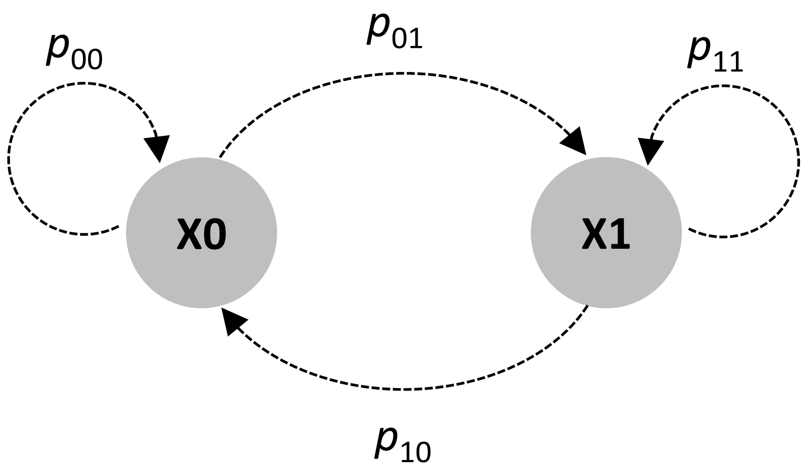
```
# gaussian noise
e = np.random.randn(N)

# 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]
```





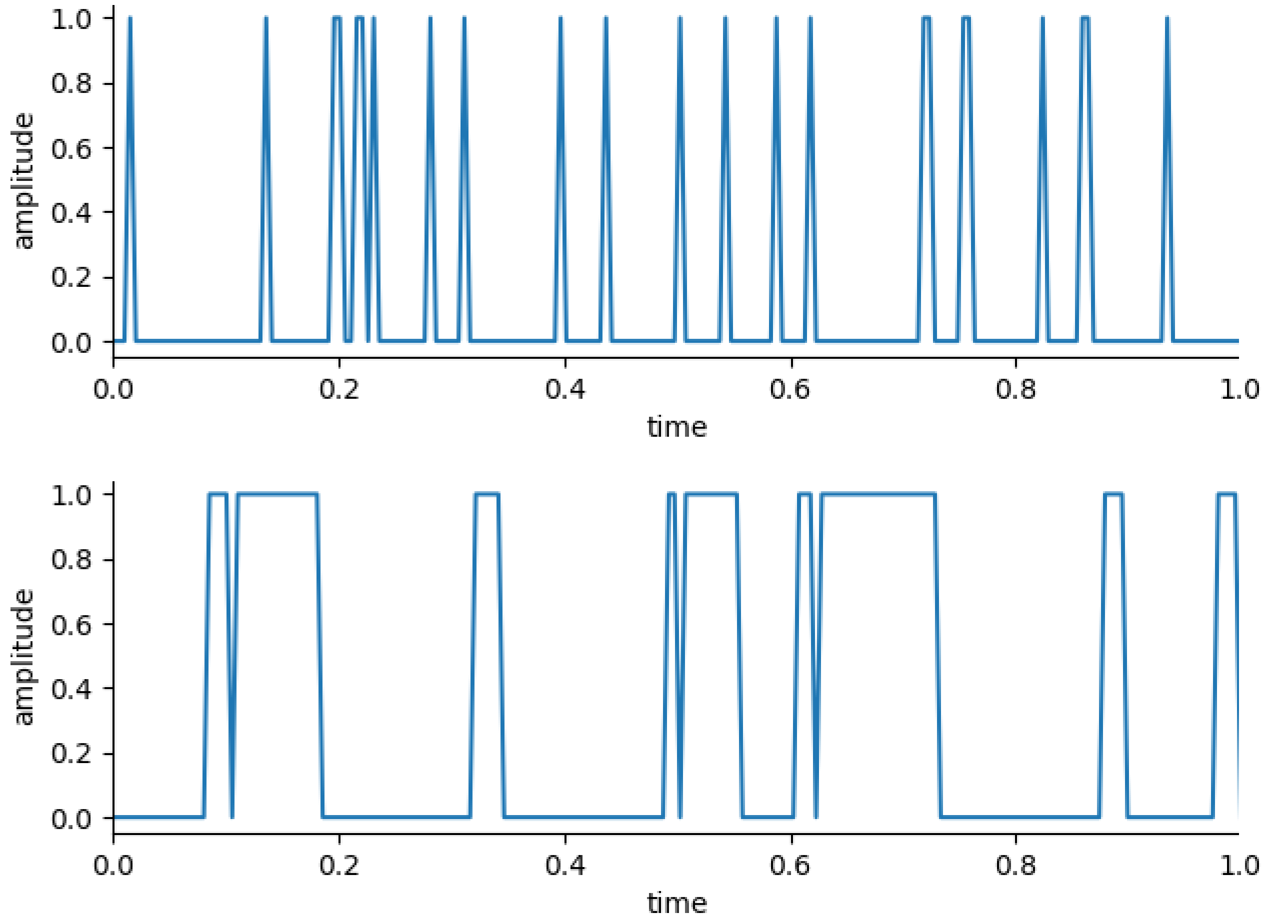
See, “L04_graph_ar_1_process.py”

Graphical representation of Markov chain



Markov chain

```
# parameters
P00 = 0.9
P11 = 0.3
p = np.array([[P00, (1 - P00)],
              [(1 - P11), P11]])

x = get_chain(p, N)


# function
def get_chain(p, N):
  x = np.zeros(N)
  state = 0
  for i in range(0, N):
    if np.random.rand(1) > p[state, state]:
          state = 1 - state
        x[i] = state
    return x
```



See, "L11_markov_chain.py"

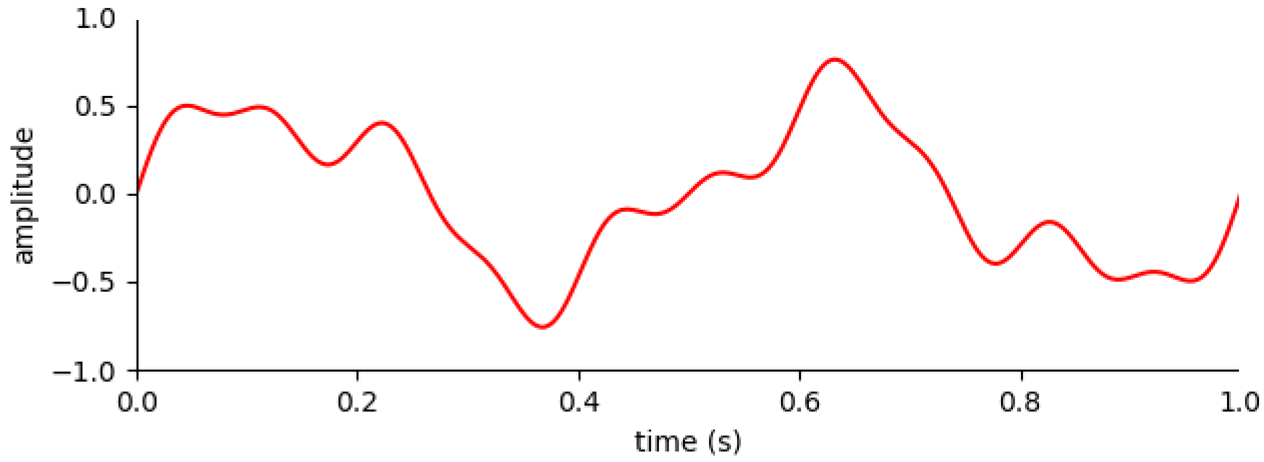
Section 3. Transformations

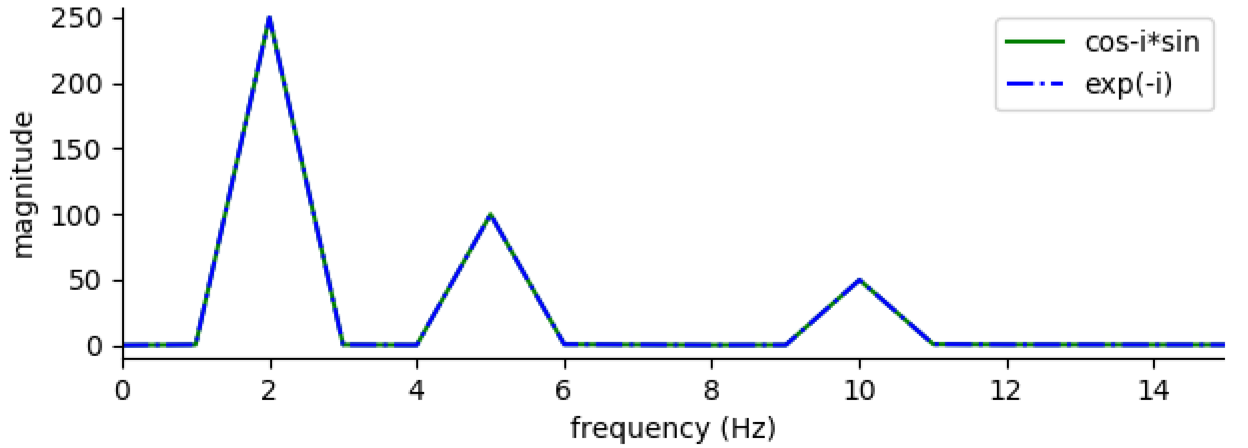
Transformations

- Fourier transform

Fourier transform

```python
# frequency resolution
nFFT = fs # fs / nFFT, in Hz

# time variable
t = np.arange(0, N)

# over frequencies
for k in range(0, nFFT):

  # relative frequency
  f = k / nFFT

  # exp
  y[k] = np.sum(np.exp(-1j * 2 * np.pi * t * f)
         * x)

  # cos + 1i * sin
  u[k] = np.sum(np.cos(2 * np.pi * t * f) * x -
           1j * np.sin(2 * np.pi * t * f) * x)
```

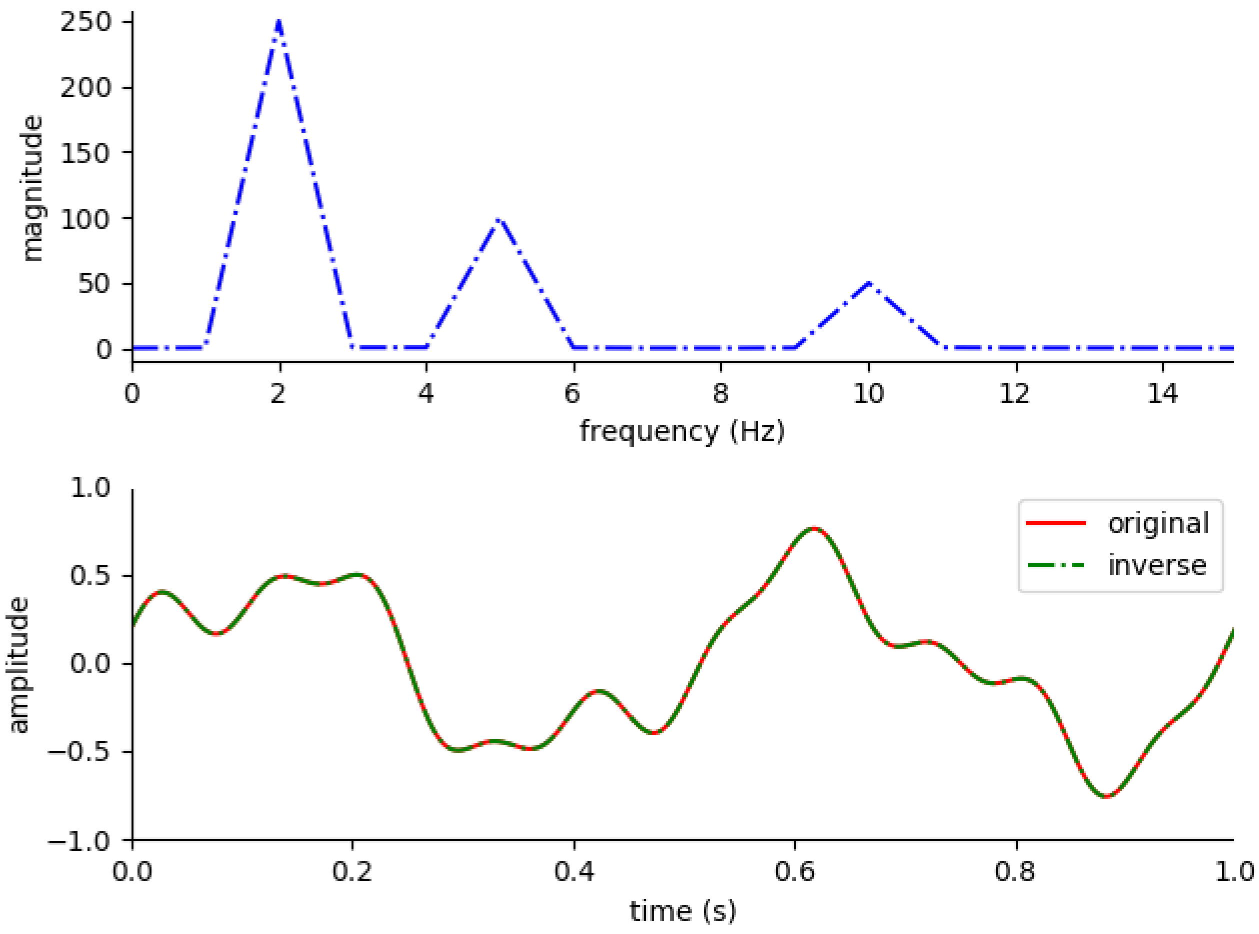




See, “L05_fourier_transform.py”

Inverse Fourier transform

```
# inverse fourier transform
x = np.zeros(N, 'complex')
t = np.arange(0, N)
for k in range(0, N):
  # relative frequency
  f = k / nFFT
  # complex exponent
  x[k] = (1/N) *
         np.sum(np.exp(1j * 2 * np.pi * t * f)
         * y)
```

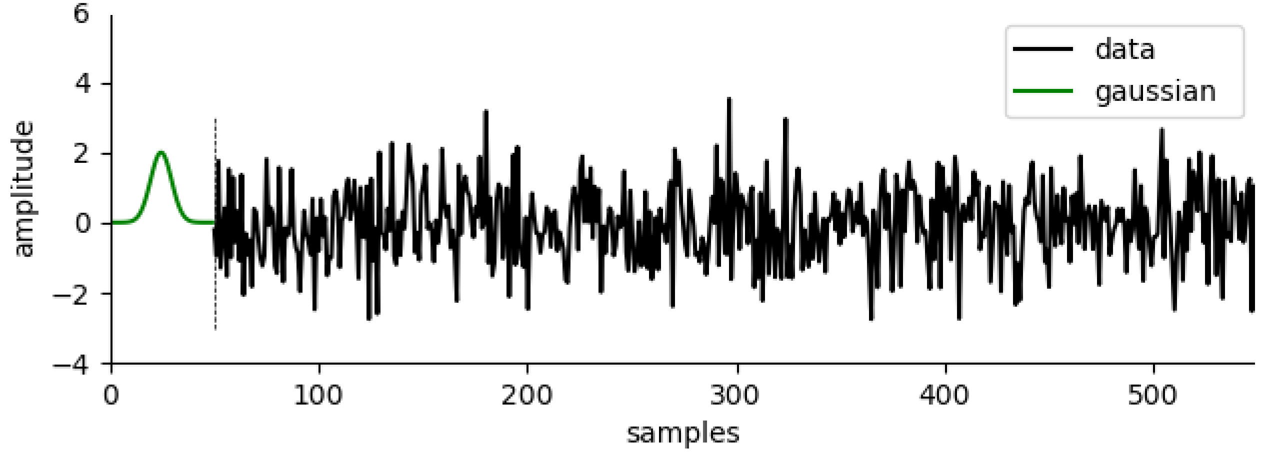


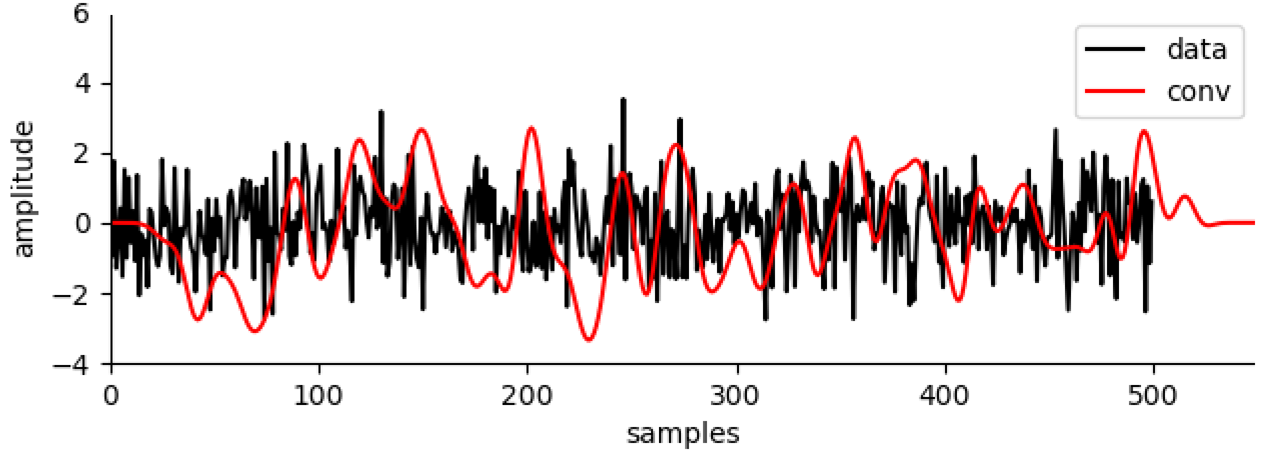
See, “L05_inverse_fourier_transform.py”

Property 1: Convolution and FFT (1/2)

Convolution in time domain equals to
Product in frequency domain.

```
# init
N = len(x)
M = len(w)

# add zeros
x = np.concatenate((np.zeros(M-1), x,
np.zeros(M-1)))
y = np.zeros(N+M-1)

# convolution
for n in range(0, (N+M-1)):
  y[n] = np.sum(x[n:(n + M)] * w[::-1])
```

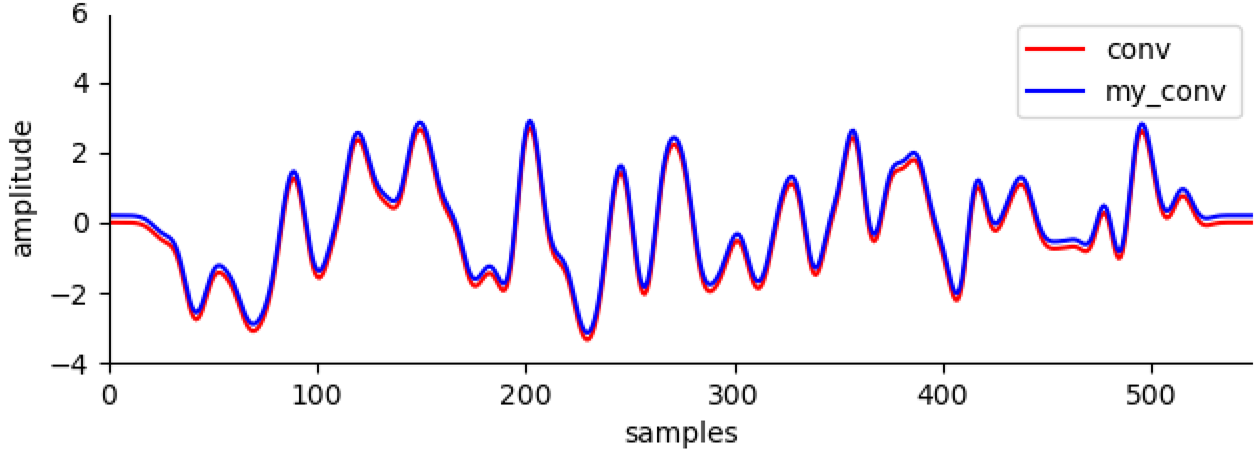


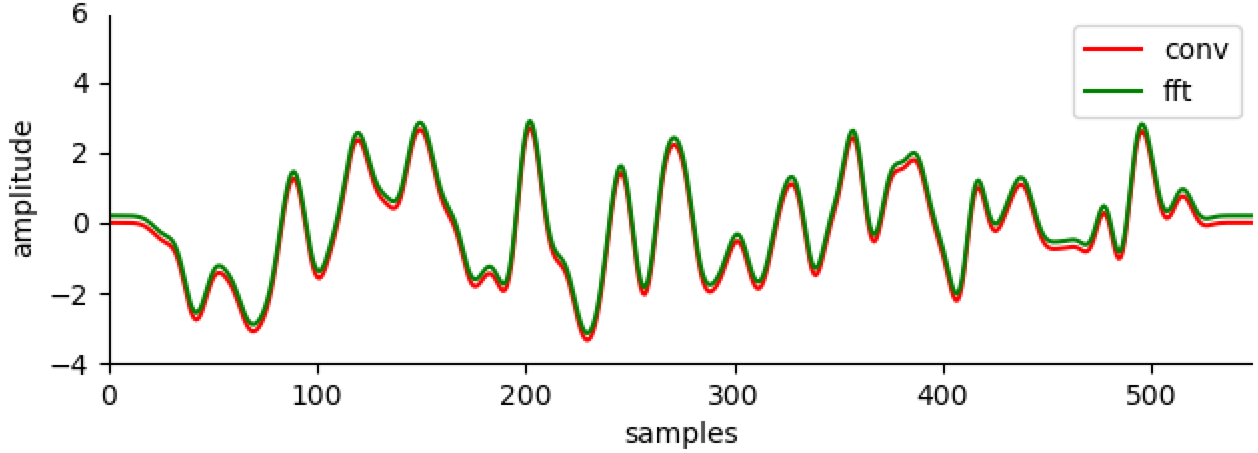


See, “L05_convolution_and_product.py”

Property 1: Convolution and FFT (2/2)

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

# window
w = signal.gaussian(M, 5)

# convolution
y = signal.convolve(x, w)

# Fourier transform
nFFT = N+M-1
u = ifft(fft(x, nFFT), fft(w, nFFT))
```



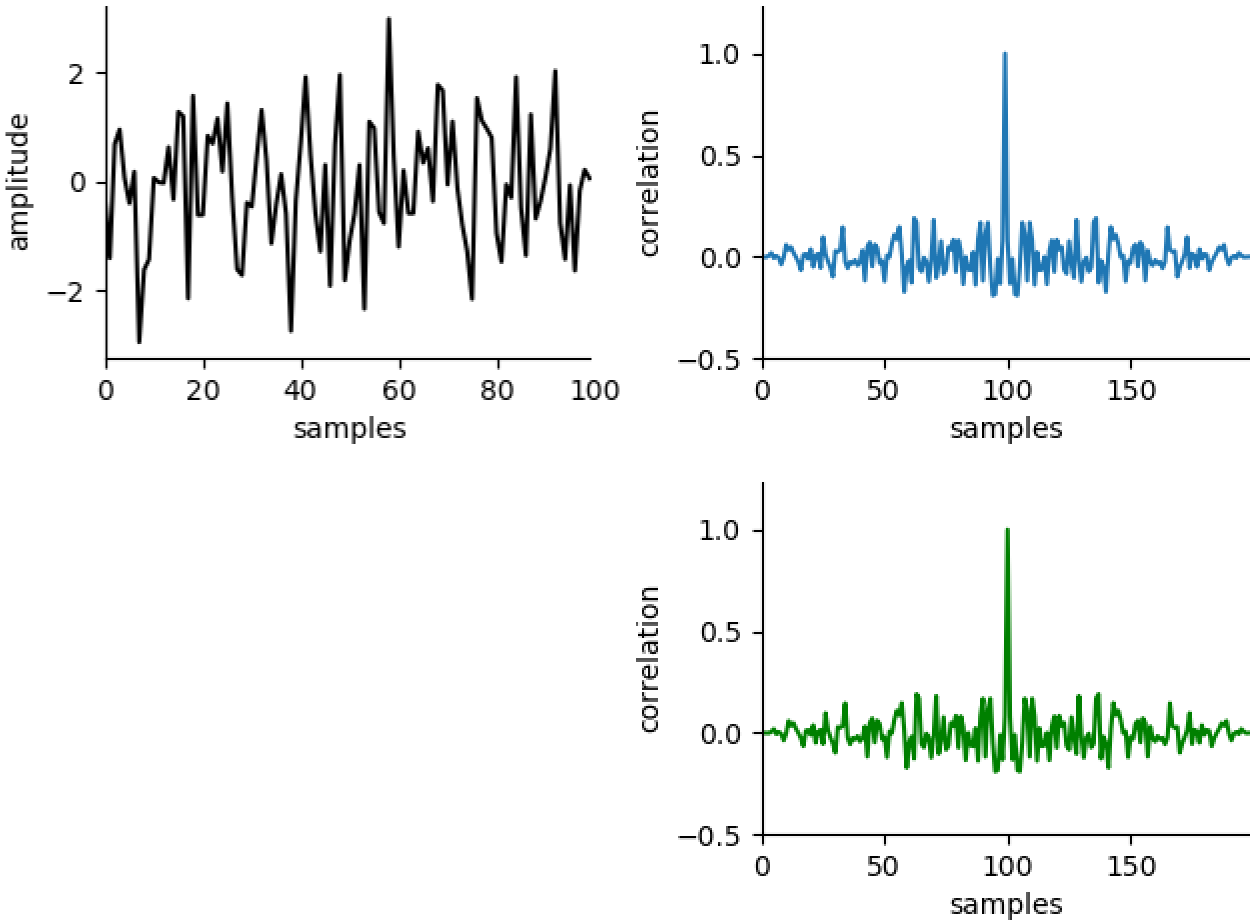


See, “L05_convolution_and_product.py”

Property 2: Autocorrelation and FFT

Autocorrelation function can be computed using FFT.

```
# compute ACF
rx = signal.correlate(x, x)
rx = rx / np.max(rx)

# compute ACF using FFT
nFFT = 2 * N
ry = np.real(ifft(fft(x, nFFT) *
          np.conj(fft(x, nFFT))))

ry = np.concatenate((ry[N::-1], ry[1:N:]))

ry = ry / np.max(ry)
```



See, “L05_acf_via_fft.py”

Section 4. Filtering

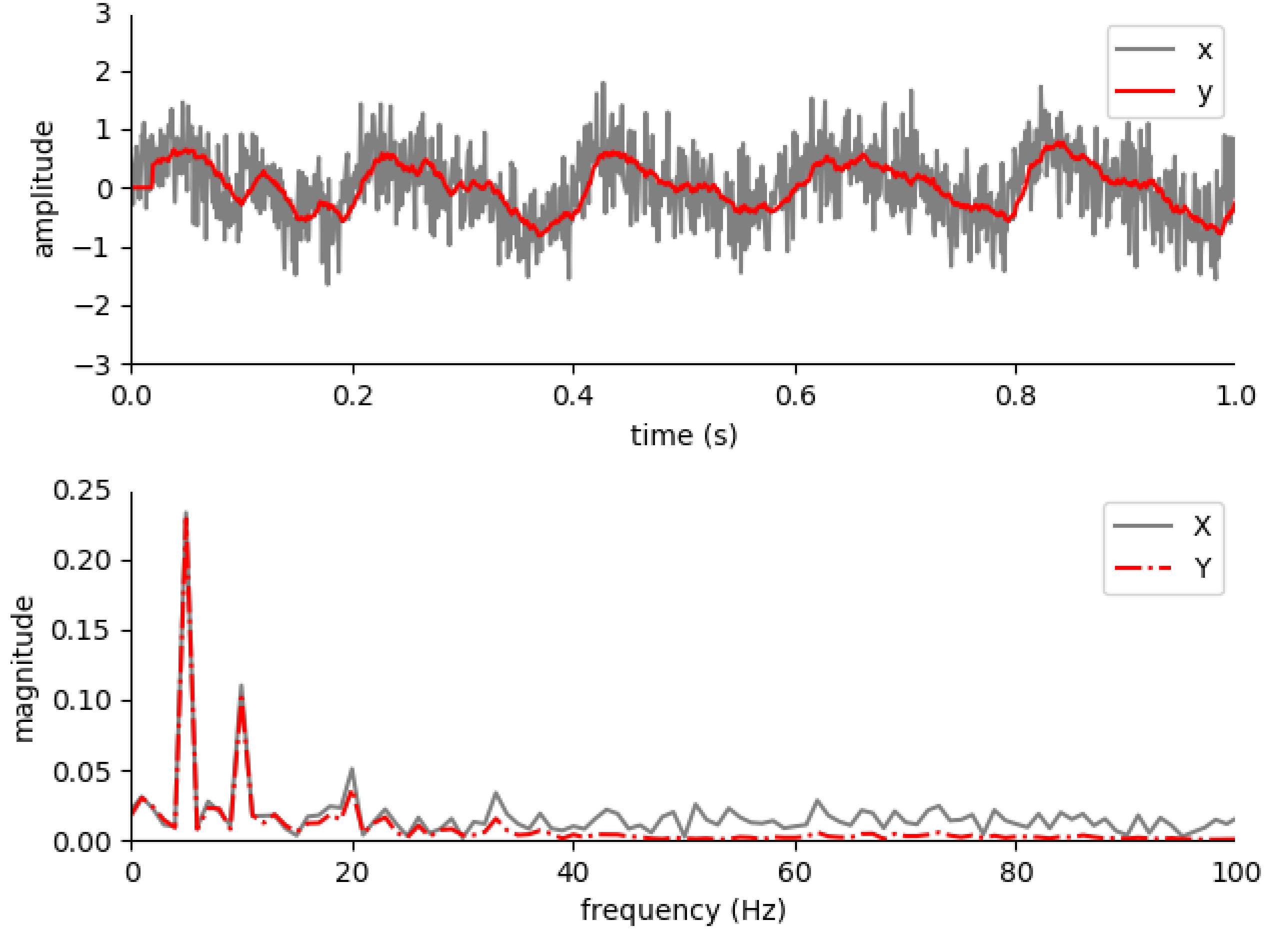
Filtering

- FIR/IIR filters
- Wavelets

Signal smoothing (1/2)

How does smoothing work?

```
def do_smoothing(x, M):

  N = len(x)
  y = np.zeros(N)

  # average
  for i in range(0, (N-M)):
    y[i+M] = np.sum(x[i:(i+M)]) / M

  return y
```

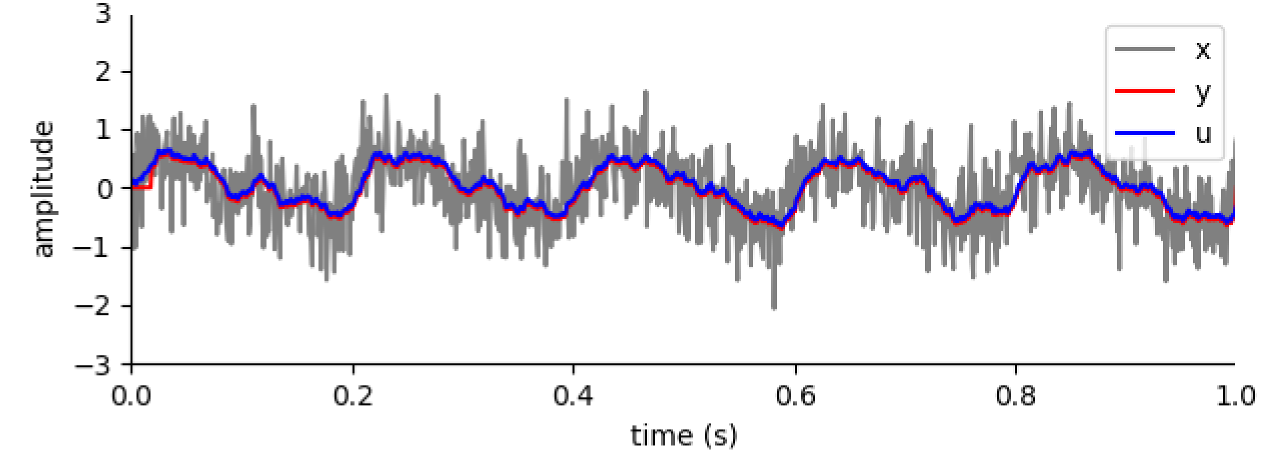


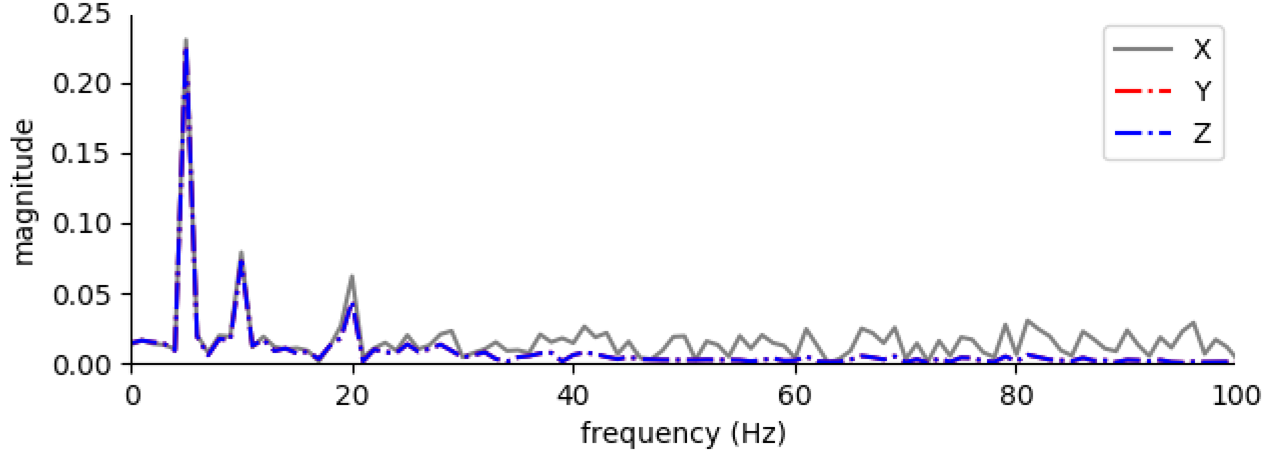
See, “L06_smoothing.py”

Signal smoothing (2/2)

Averaging is equivalent to convolution with square window.

```
# signal
x = 0.5 * np.sin(2 * np.pi * 5 * t)  + \
    0.2 * np.sin(2 * np.pi * 10 * t) + \
    0.1 * np.sin(2 * np.pi * 20 * t) + \
    0.5 * np.random.randn(N)
X = np.abs(fft(x)) / N

# smoothing, M samples
M = 20
y = do_smoothing(x, M)
Y = np.abs(fft(y)) / N

# convolution
w = np.ones(M) / M
u = do_convolution(x, w)
u = u[0:N]
U = np.abs(fft(y))
```





See, “L06_convolution_square_window.py”

Time and frequency domains

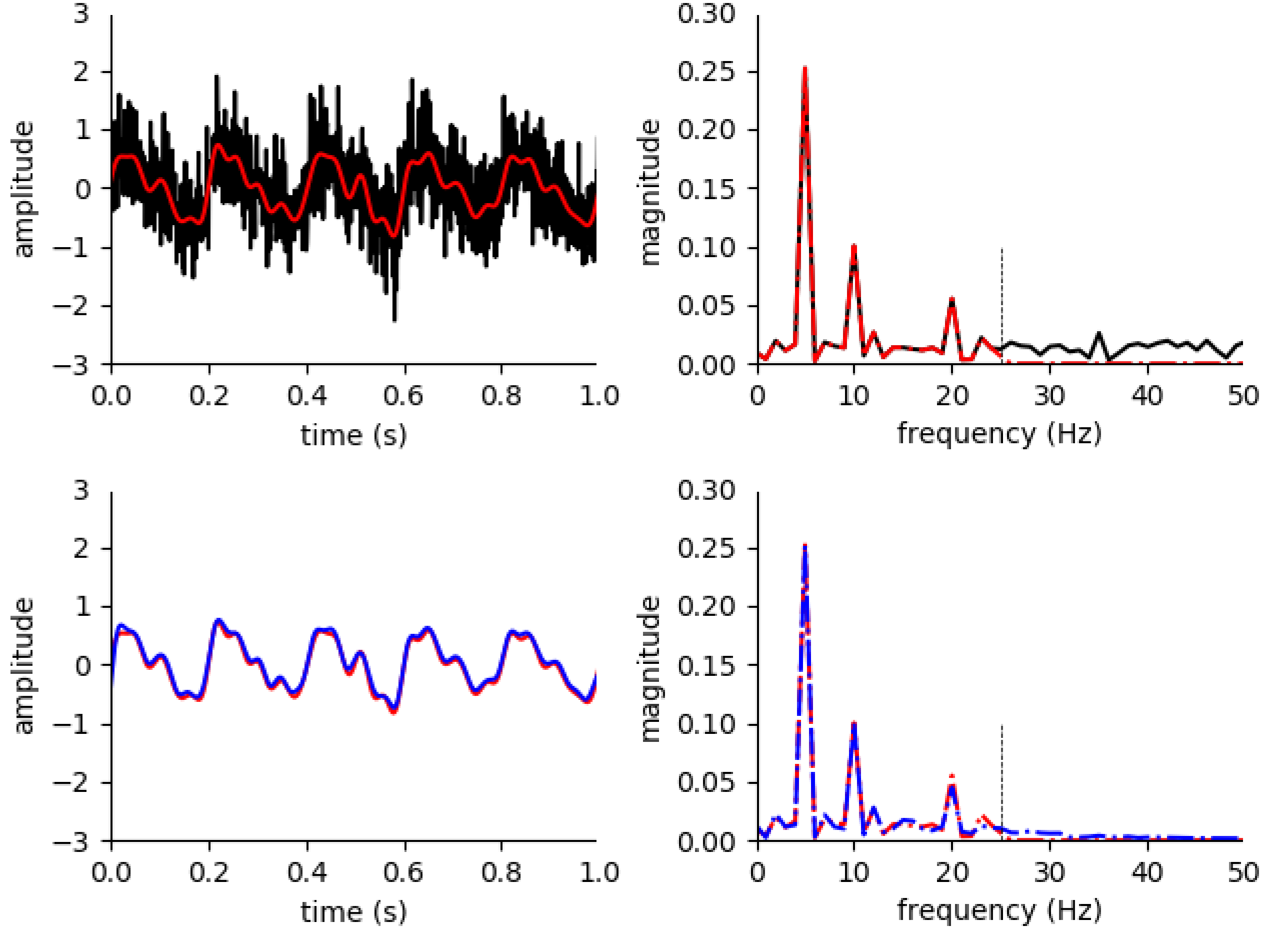
Filtering can be done in frequency and time domains.

Frequency domain

```
# fourier transform
fx = fft(x)
# remove frequencies above f0
fx[np.arange(f0, nFFT-f0)] = 0
# inverse fourier transform
y = ifft(fx)
```

Time domain

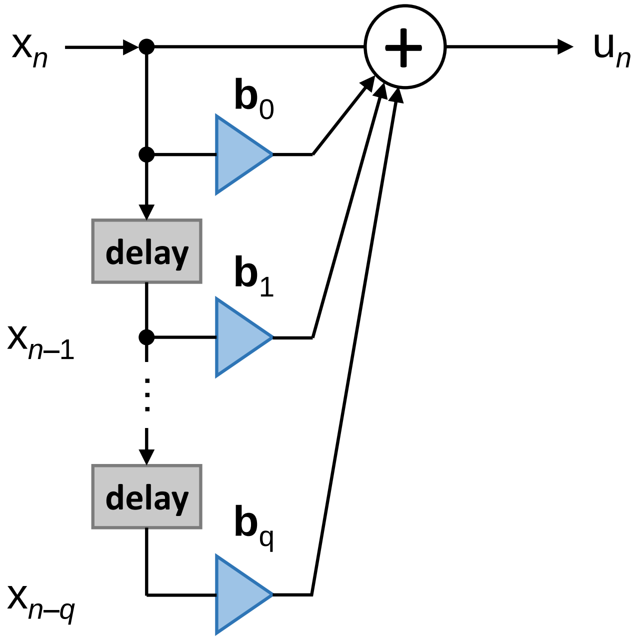
```
# design filter
[b, a] = signal.butter(4, f0 / (fs/2), 'low')
# apply filter
u = signal.filtfilt(b, a, x)
```



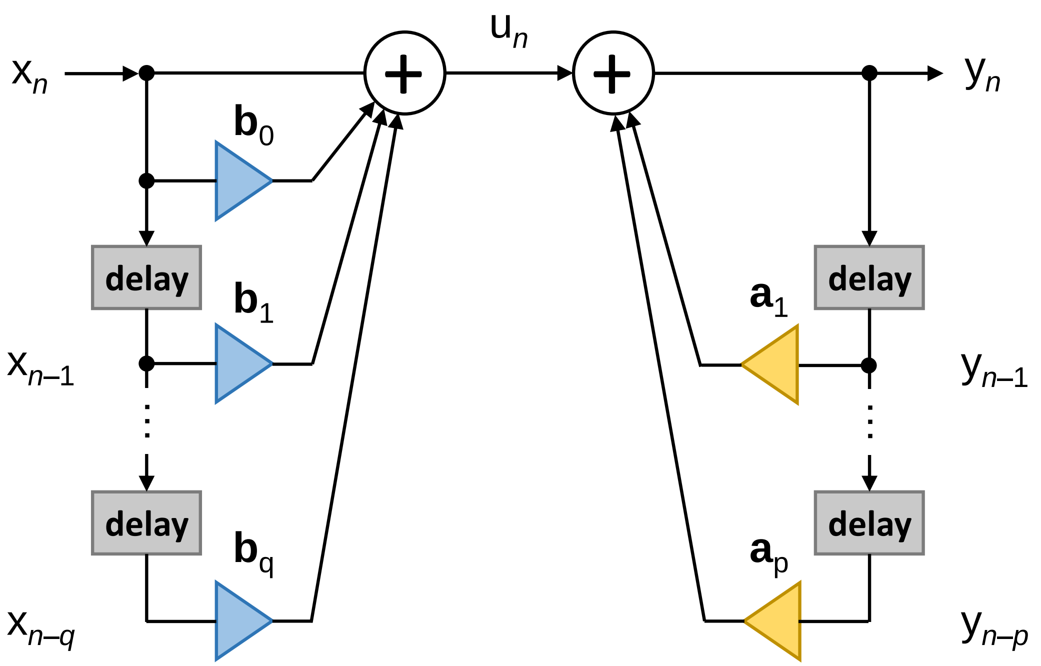
See, “L06_time_frequency_domains.py”

Impulse response (Finite IR / Infinite IR)

FIR / Direct form I



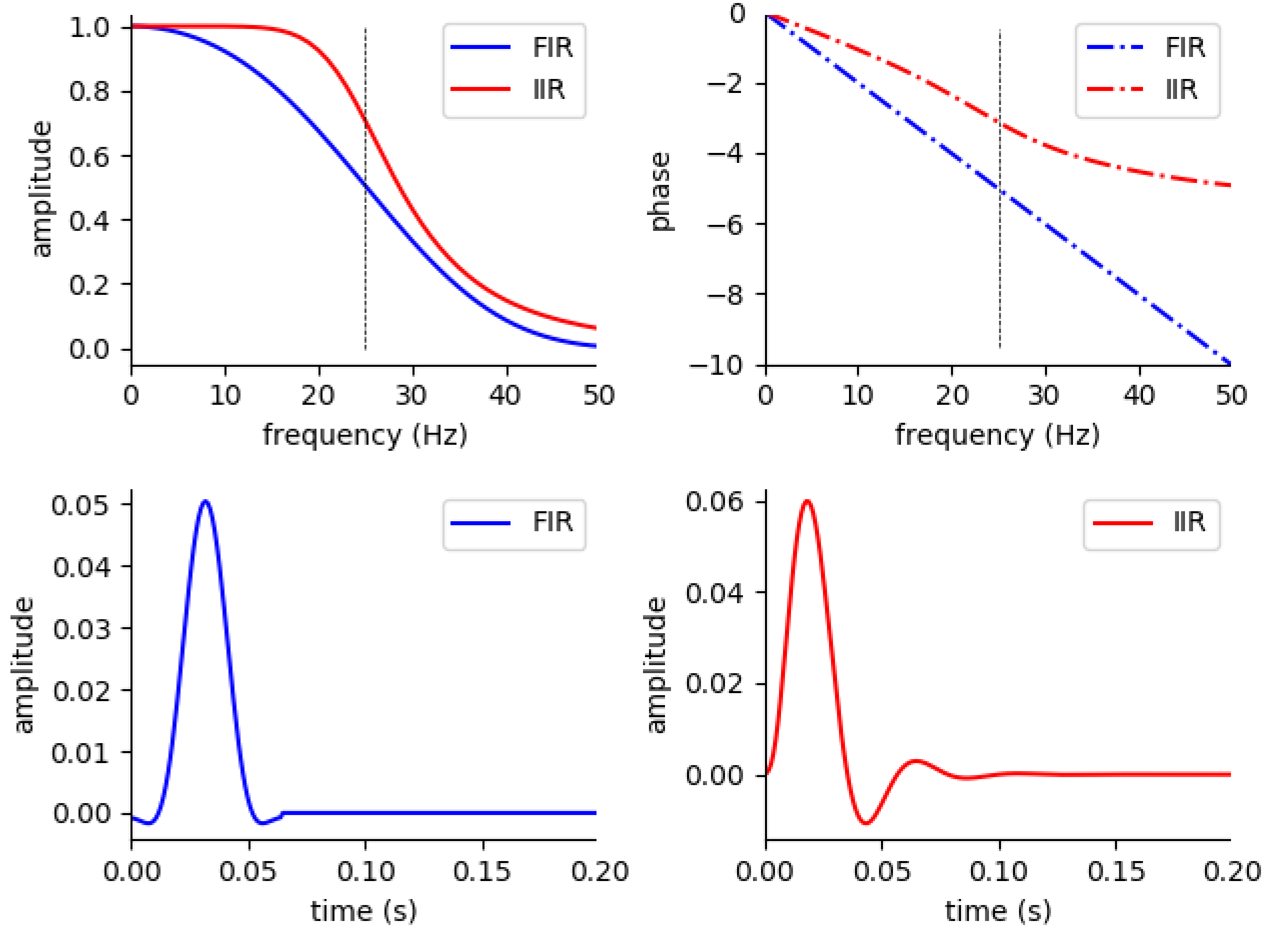
IIR / Direct form I



https://se.mathworks.com/help/signal/ref/dfilt.html

FIR vs IIR filters

```
# design filter
n1 = 65 # FIR filter order
n2 = 4  # IIR filter order

# low-pass FIR filter
a1 = 1
b1 = signal.firwin(numtaps=n1, cutoff=fc)

# low-pass IIR filter
[b2, a2] = signal.butter(n2, fc, 'low')


# impulse response
p = np.zeros(200)
p[0] = 1
resp = signal.lfilter(b, a, p)
```

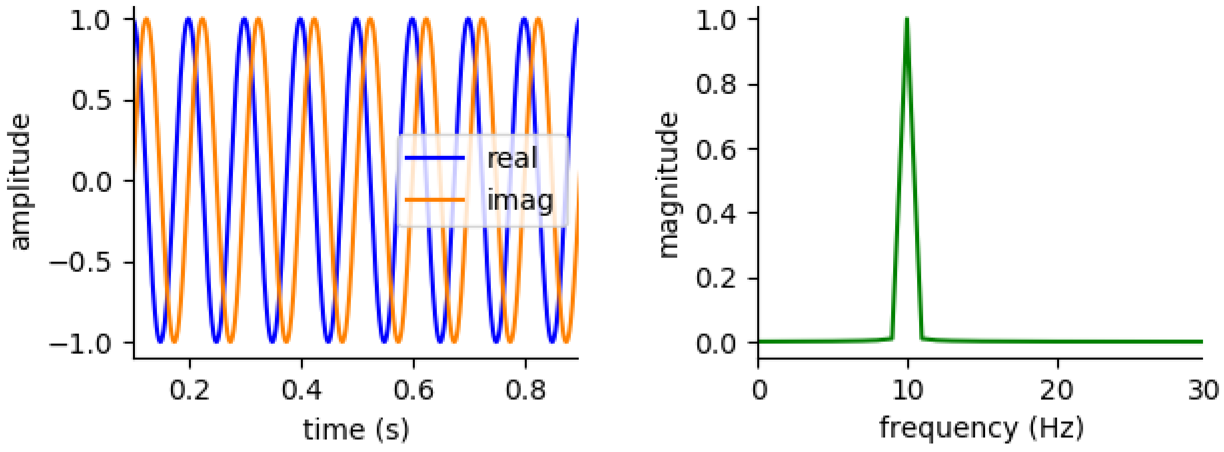


See, “L06_fir_vs_iir_filter.py”

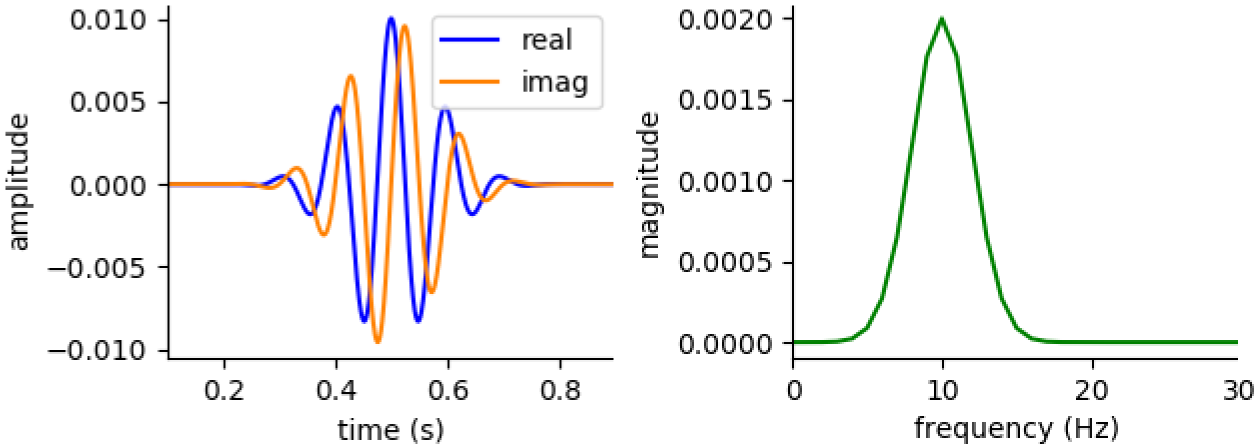
Wavelets

What is the difference between sine (*complex exponent*) and wavelet?

```
# signal
x = np.cos(2 * np.pi * 10 * t) +
    1j * np.sin(2 * np.pi * 10 * t)
X = np.abs(fft(x)) / N
```




```
# init Morlet wavelet
f0 = 10
m = 5
a, b = init_wavelet(f0, m, fs)

# shape to draw
y = np.concatenate((a, b))
Y = np.abs(fft(y)) / N
```

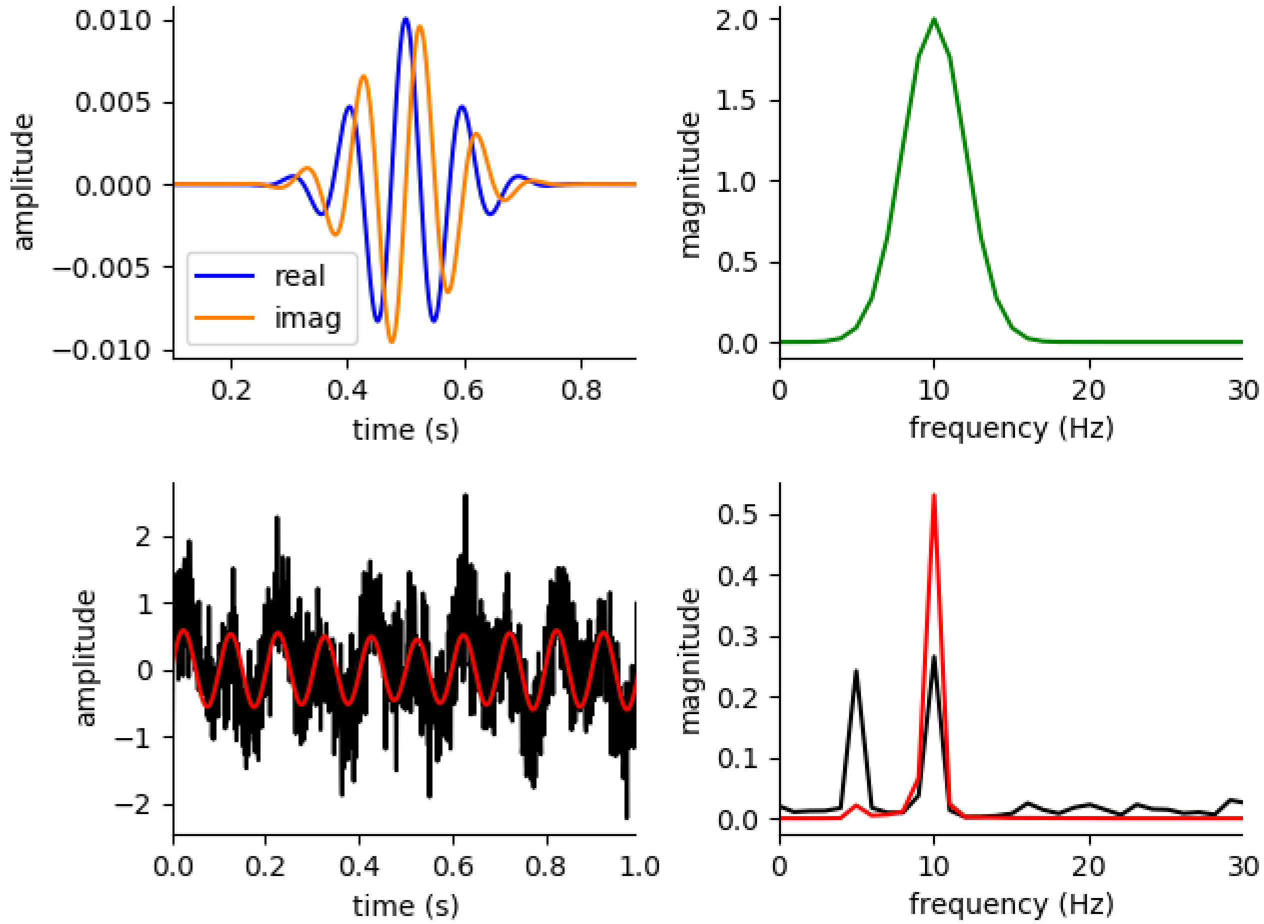


See, “L07_wavelet_vs_sine.py”

Wavelet filtering

What are the parameters of **Morlet** wavelet?

```
# signal
x = 0.5 * np.sin(2 * np.pi * 5 * t)  + \
    0.5 * np.sin(2 * np.pi * 10 * t) + \
    0.5 * np.random.randn(N)

# init Morlet wavelet
f0 = 10
m = 5
a, b = init_wavelet(f0, m, fs)

# concatenate halves
w = [b, np.zeros(2 * L, 'complex'), a)]

# filtering
y = ifft(fft(x) * fft(w))

# amplitude spectra
X = np.abs(fft(x)) / N
Y = np.abs(fft(y)) / N
```



See, “L07_wavelet_filtering.py”

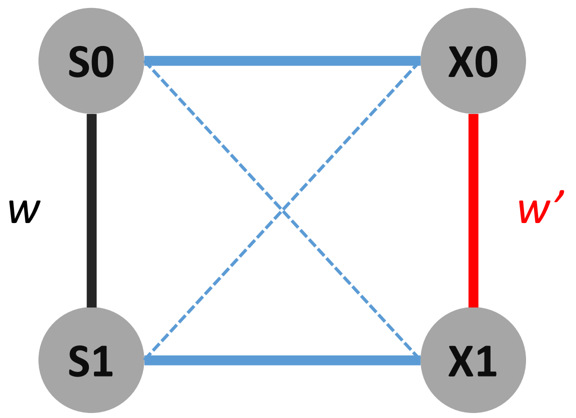
Section 5. **Interactions**

Interactions

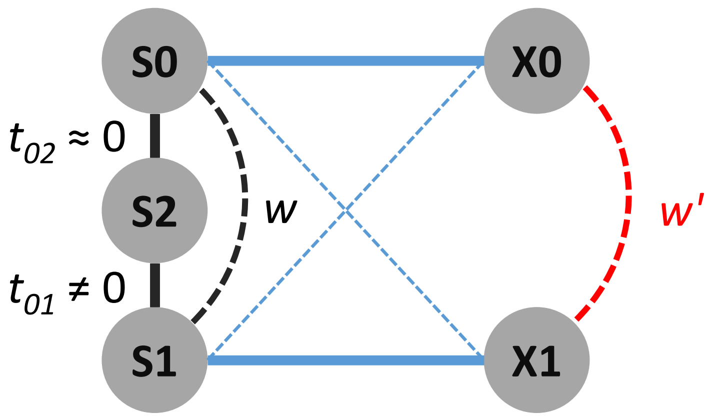
- Amplitude correlation and phase-coupling
- Granger causality

Interactions

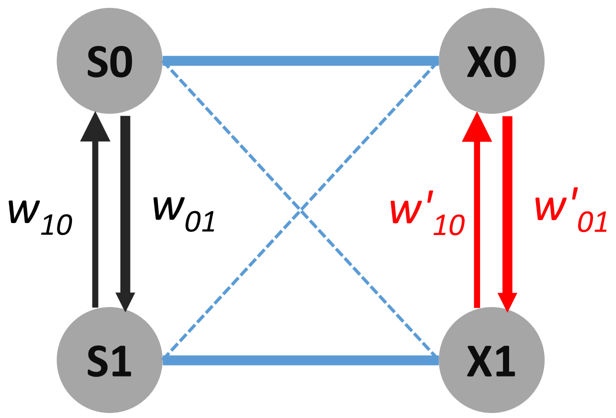
Zero-lag interactions



Non-zero-lag interactions



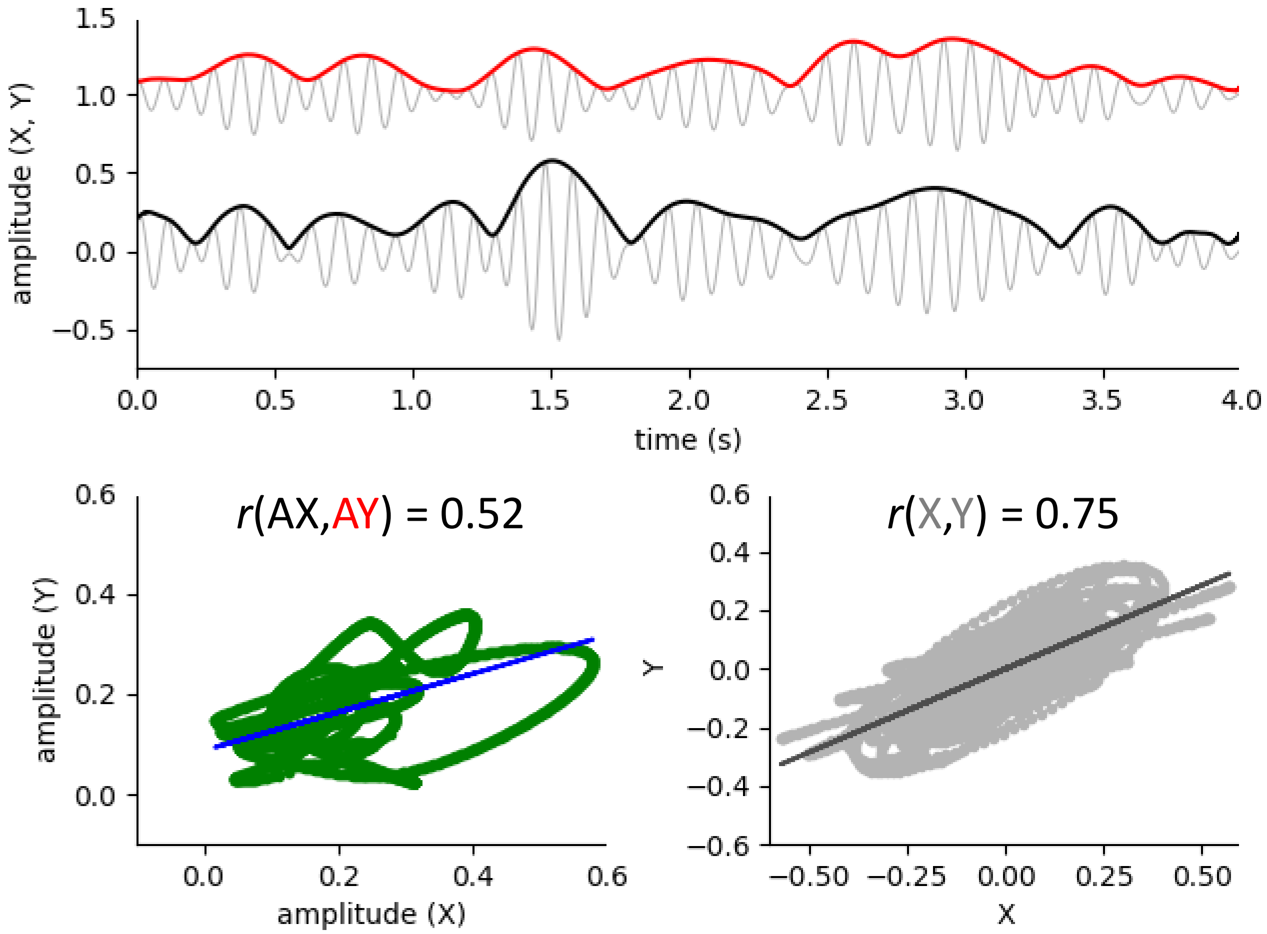
Causal interactions



I. Amplitude-amplitude correlations

Correlation between amplitudes of two signals.

```
# amplitude
AX = np.abs(signal.hilbert(X))
AY = np.abs(signal.hilbert(Y))

# linear fit (amplitudes)
p = np.polyfit(AX, AY, 1)
AU = p[0] * AX + p[1]

# linear fit (signal)
p = np.polyfit(X, Y, 1)
U = p[0] * X + p[1]

# correlation
rAA = np.corrcoef(AX, AY)[0, 1]
rXY = np.corrcoef(X, Y)[0, 1]
```

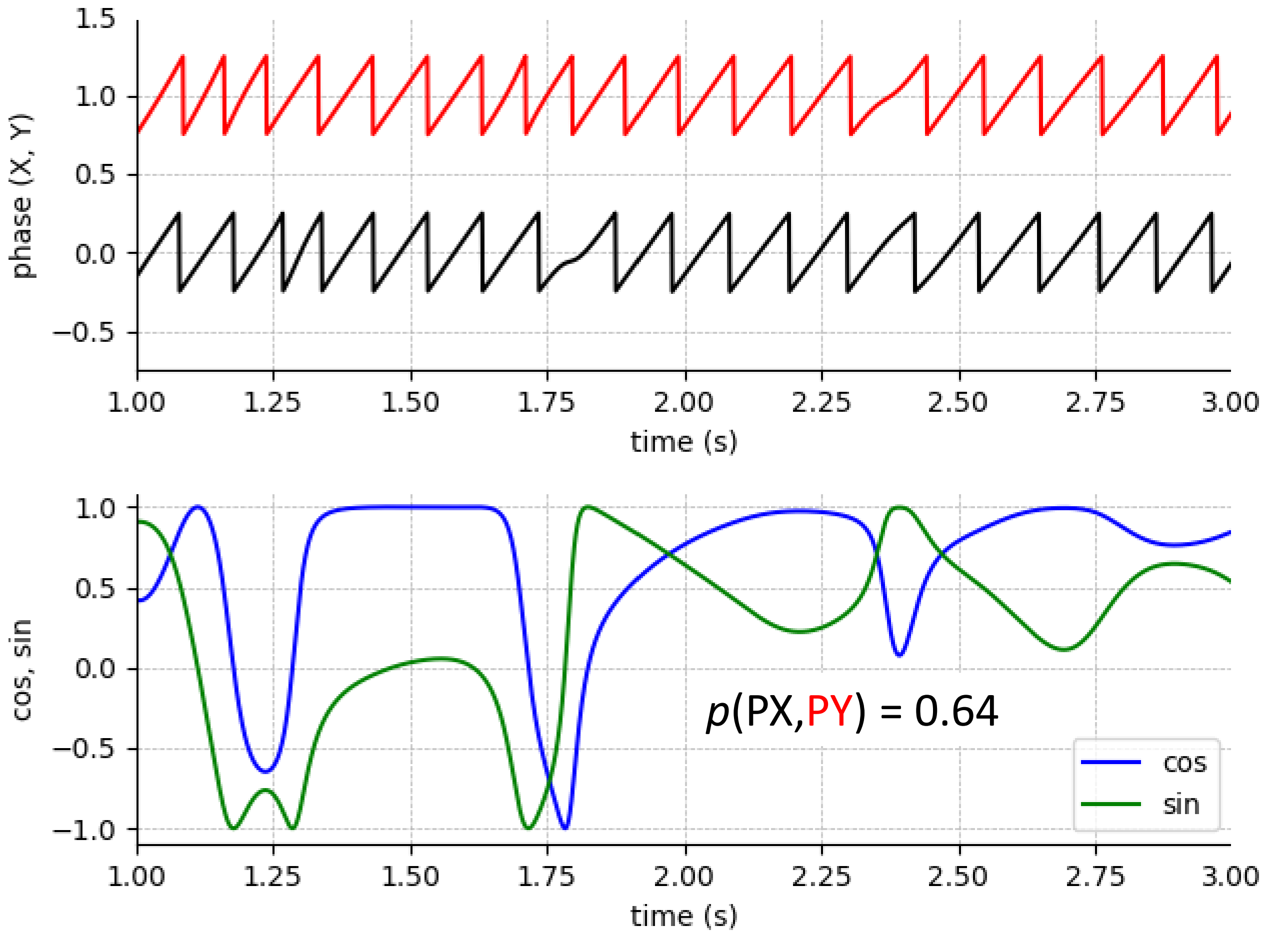


See, "L08_zero_lag_interactions_amplitude.py"

II. Phase-phase coupling

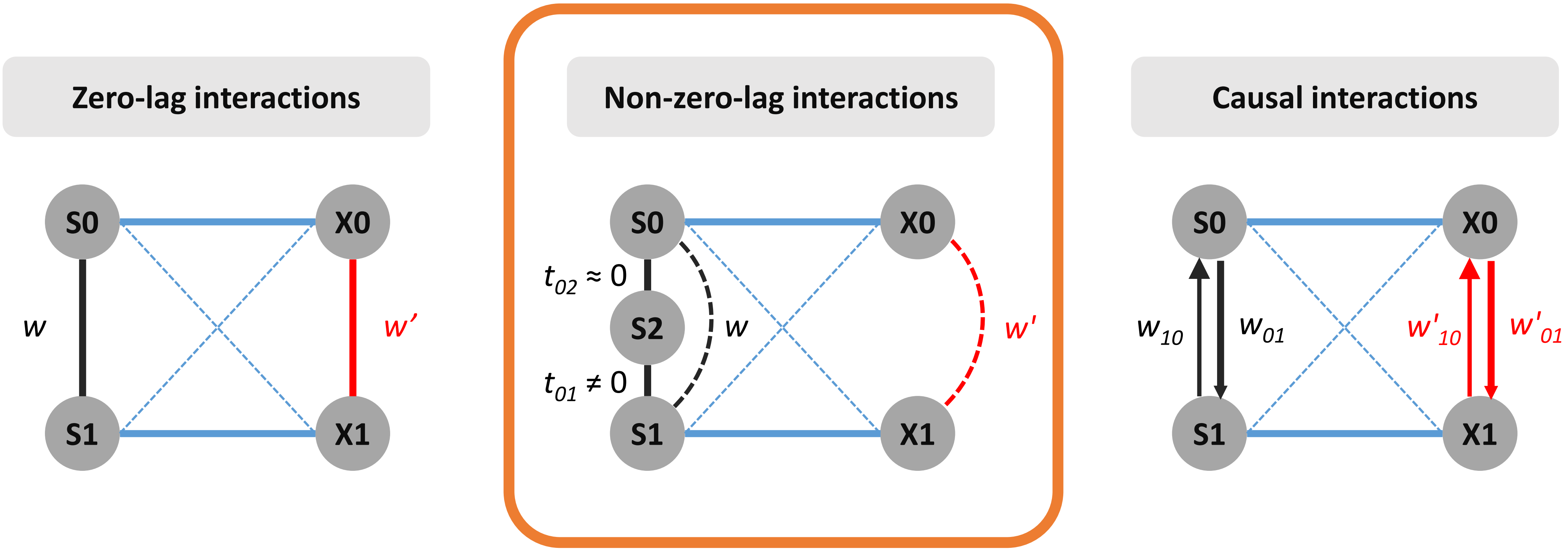
Coupling between phases of two signals can be assessed via phase-locking value.

```
# phase
PX = np.angle(signal.hilbert(X))
PY = np.angle(signal.hilbert(Y))

# phase-locking value
p = np.abs(np.sum(np.exp(1j * (PX - PY))) / N)

# phase-locking value via sine
p = np.abs(np.sum(np.cos(PX - PY) +
                  1j * np.sin(PX - PY)) / N)
```



See, “L08_zero_lag_interactions_phase.py”

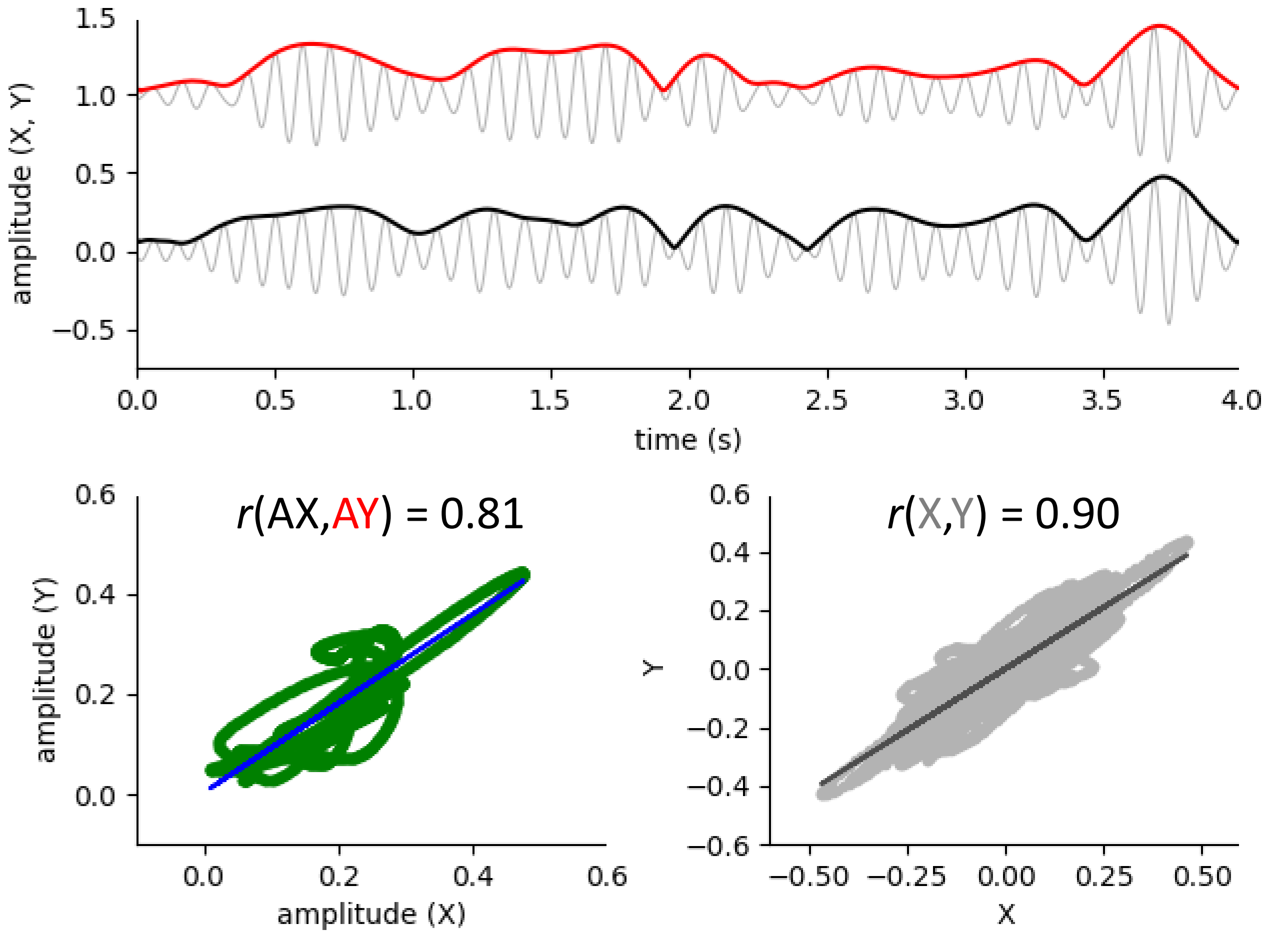
Interactions



I. Amplitude-amplitude correlations (1/2)

Correlation between amplitudes of two signals via third source.

```
# mixing matrix
A = np.array([[1.0, 0.0, 2.0], \
              [0.0, 1.0, 2.0], \
              [0.0, 0.0, 1.0]])

# amplitude
AX = np.abs(signal.hilbert(X))
AY = np.abs(signal.hilbert(Y))

# linear fit
p = np.polyfit(AX, AY, 1)
fAY = p[0] * AX + p[1]
p = np.polyfit(X, Y, 1)
fY = p[0] * X + p[1]

# correlation
rAA = np.corrcoef(AX, AY)[0, 1]
rXY = np.corrcoef(X, Y)[0, 1]
```

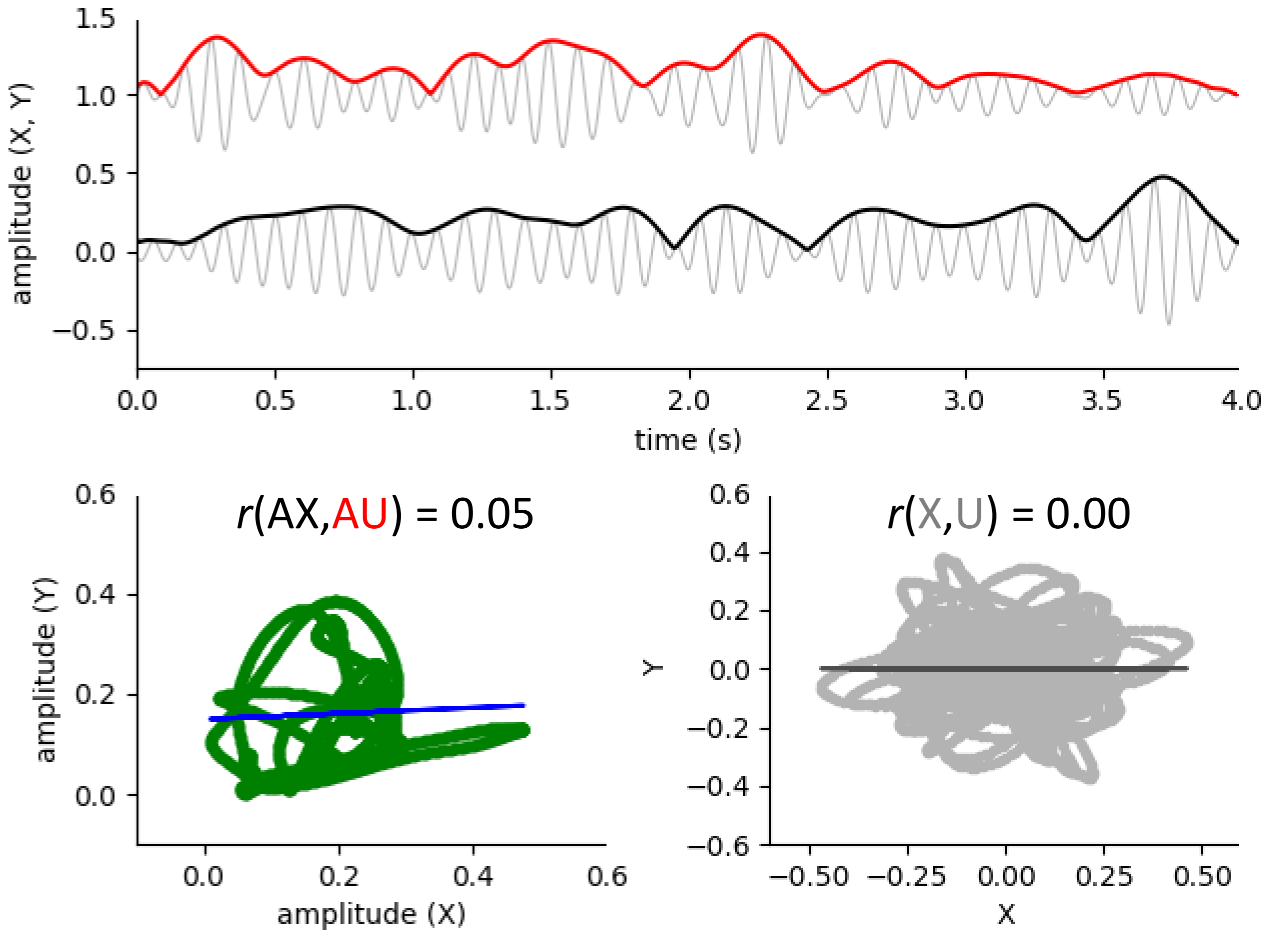


See, “L08_non_zero_lag_interactions_amplitude.py”

I. Amplitude-amplitude correlations (2/2)

Correlation between amplitudes of two signals with regressed out a common source.

```
# regression
b = np.sum((X / np.sum(X ** 2)) * Y)
U = Y - X * b

# amplitude
AX = np.abs(signal.hilbert(X))
AU = np.abs(signal.hilbert(U))

# linear fit
p = np.polyfit(AX, AU, 1)
AU = p[0] * AX + p[1]
p = np.polyfit(X, U, 1)
fU = p[0] * X + p[1]

# correlation
rAA = np.corrcoef(AX, AU)[0, 1]
rXU = np.corrcoef(X, U)[0, 1]
```

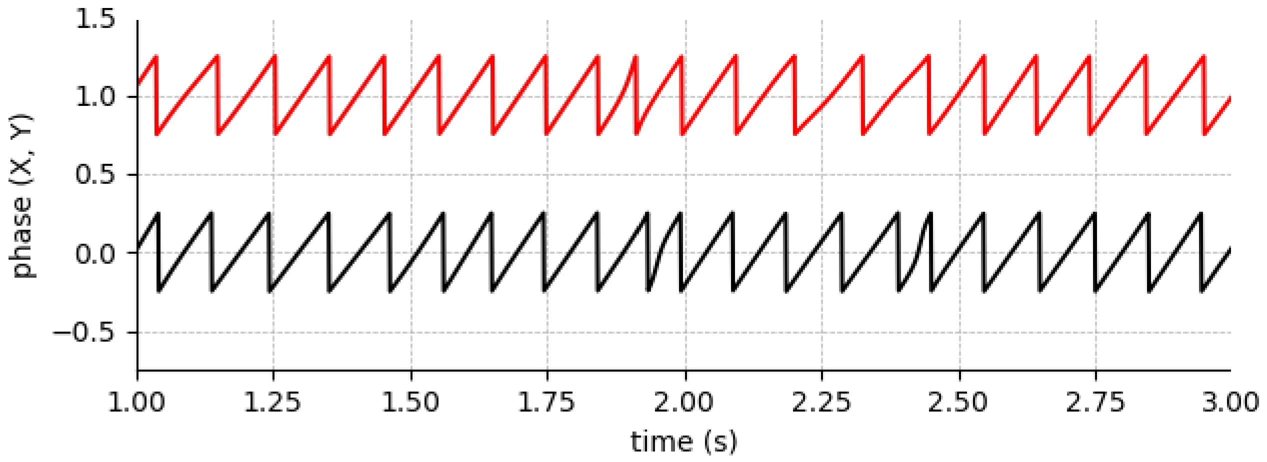


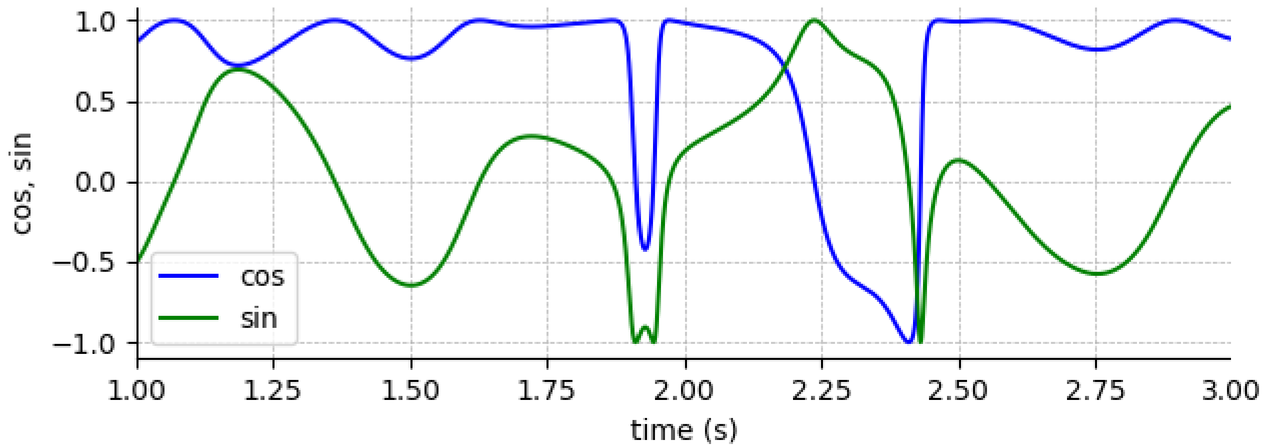
See, “L08_non_zero_lag_interactions_amplitude.py”

II. Phase-phase coupling

Coupling between phases of two signals can be assessed via phase-locking value.

```
# phase
PX = np.angle(signal.hilbert(X))
PY = np.angle(signal.hilbert(Y))

# phase-locking value
p = np.abs(np.sum(np.exp(1j * (PX - PY))) / N)


# phase-locking value
pr = np.real(np.sum(np.exp(1j * (PX - PY)))
       / N)
pi = np.imag(np.sum(np.exp(1j * (PX - PY)))
       / N))
```

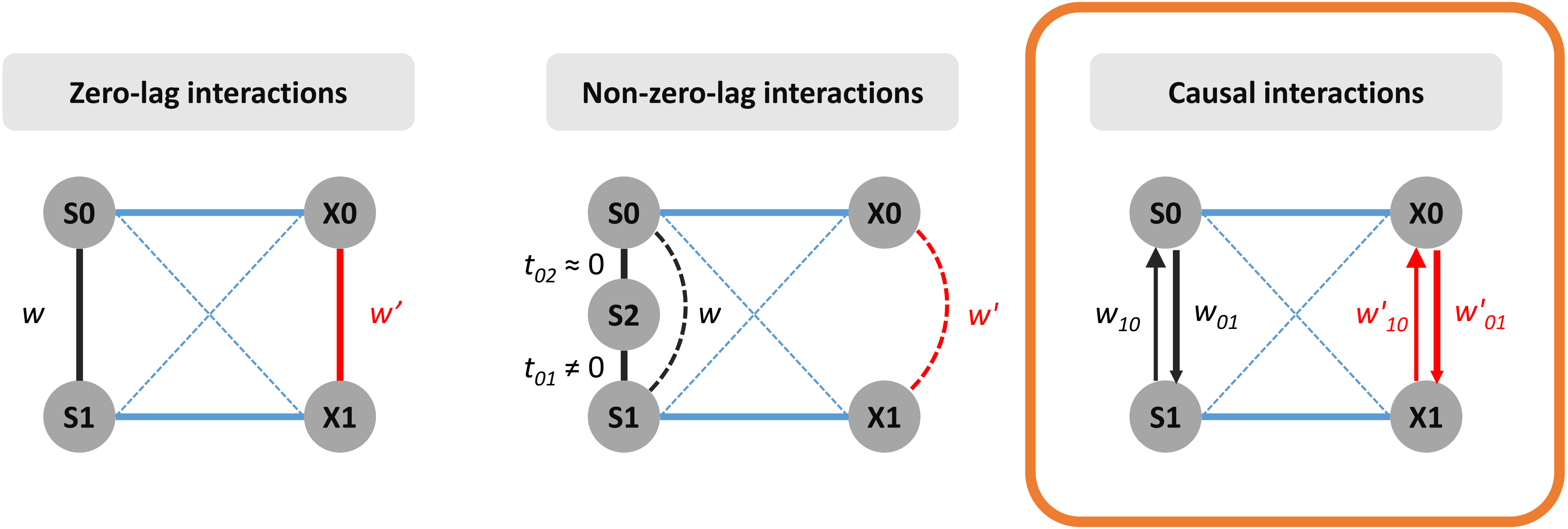




p(PX,PY) = 0.75 *p_re*(PX,PY) = 0.75 *p_im*(PX,PY) = 0.00

See, “L08_non_zero_lag_interactions_phase.py”

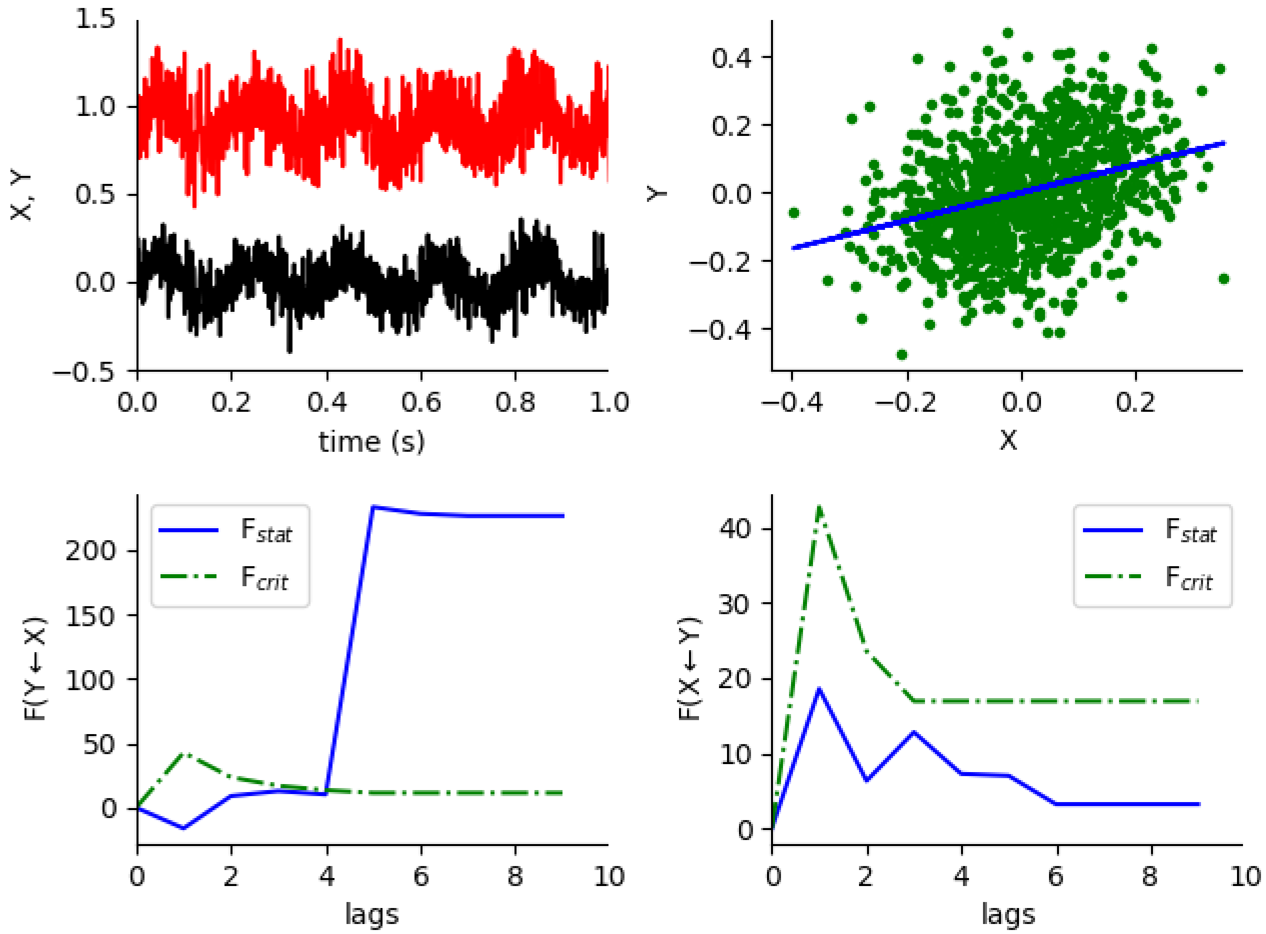
Interactions



Granger causality (1/2)

Does Y granger cause X?

```
# signal
lag = 5
X = 0.1 * np.sin(2 * np.pi * 5 * t) +
    0.1 * np.random.randn(N)
Y = np.concatenate((X[lag:], X[:lag])) +
    0.1 * np.random.randn(N)


# Granger causality
max_lags = 10

for max_lag in range(1, max_lags):
  # Y << X
  F_stat, F_crit = granger_causality(X, Y,
                   1e-10, max_lag)
  # X << Y
  F_stat, F_crit = granger_causality(X, Y,
                   1e-10, max_lag)
```

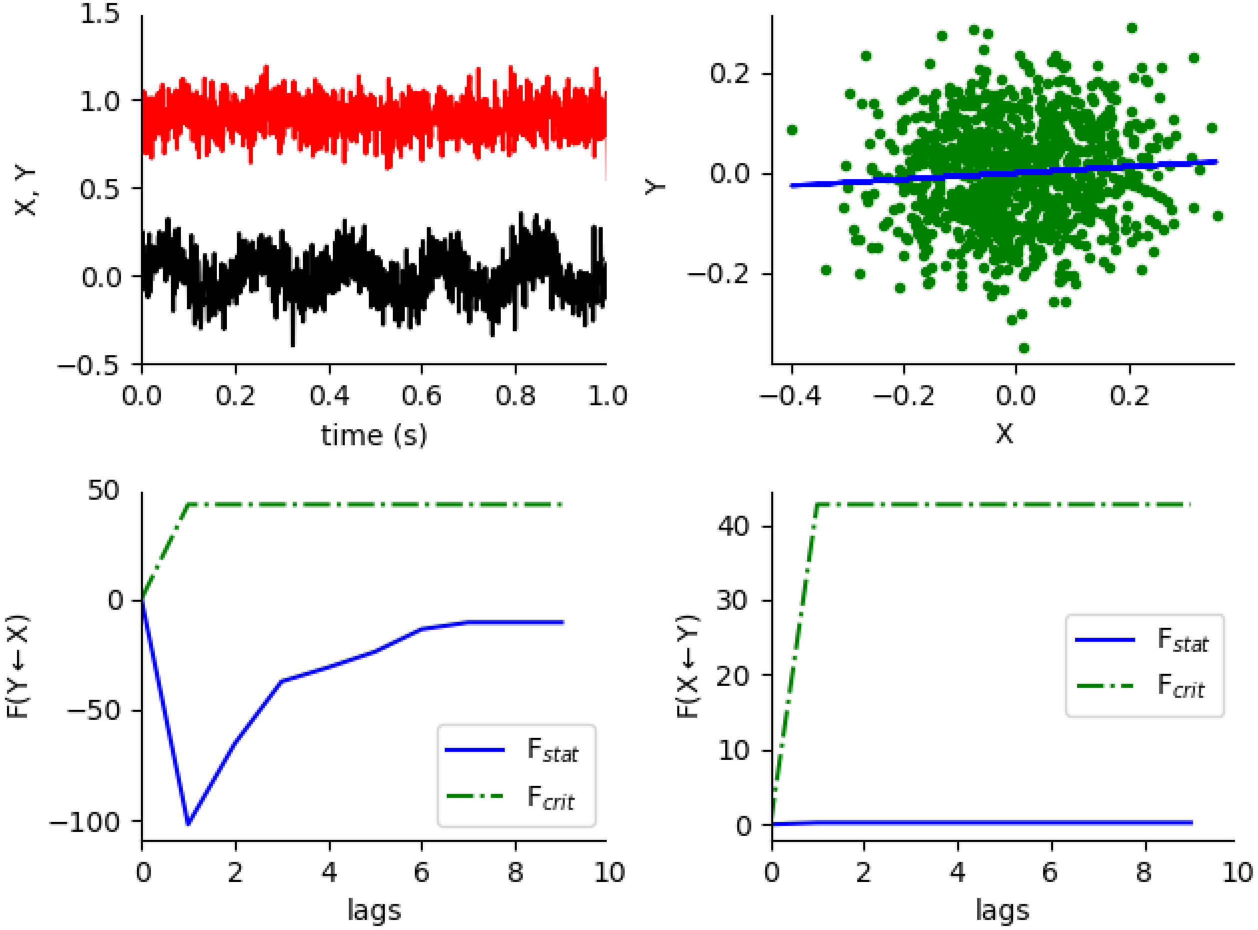


See, "L08_causal_interactions.py"

Granger causality (2/2)

Does Y granger cause X?

```
# signal
X = 0.1 * np.sin(2 * np.pi * 5 * t) +
    0.1 * np.random.randn(N)
Y = 0.1 * np.random.randn(N)


# Granger causality
max_lags = 10

for max_lag in range(1, max_lags):
  # Y << X
  F_stat, F_crit = granger_causality(X, Y,
                        1e-10, max_lag)
  # X << Y
  F_stat, F_crit = granger_causality(X, Y,
                        1e-10, max_lag)
```



See, “L08_causal_interactions.py”

Section 6. **Clustering**

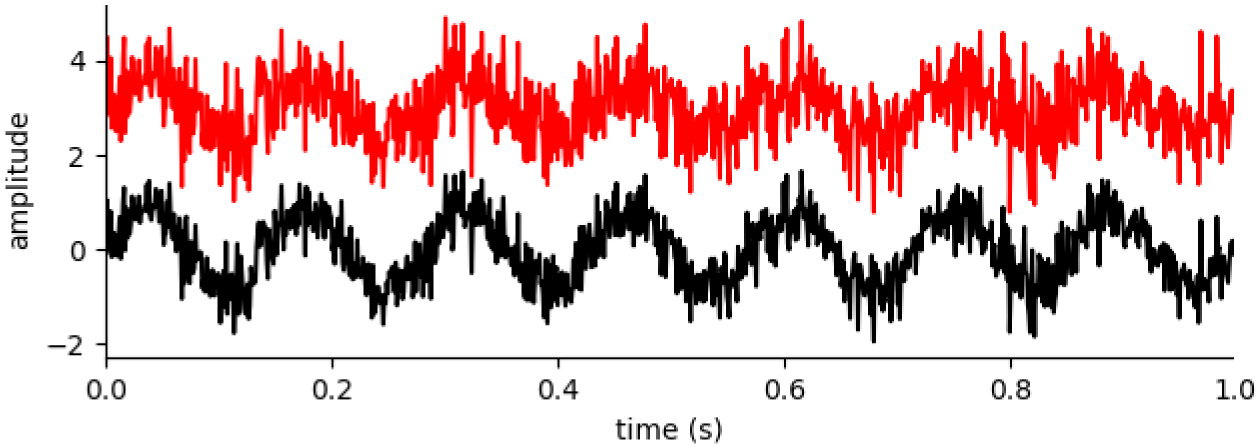
Clustering

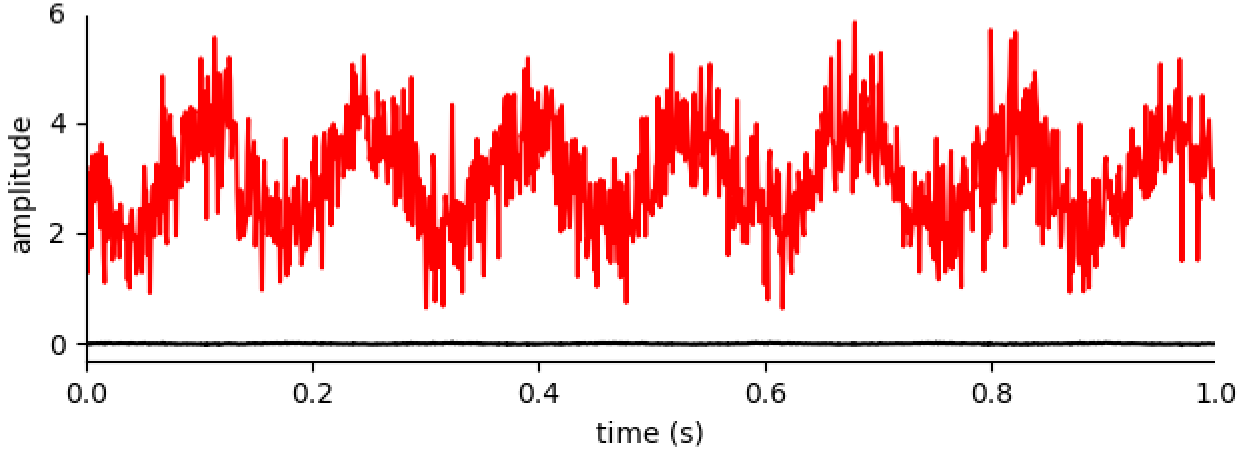
- PCA/ICA
- K-means and hierarchical clustering

Principal component analysis

How do principal component look like?

```
# sources
S0 = np.sin(2 * np.pi * 7 * t)
S1 = np.random.randn(N)

# mixing matrix
A = np.array([[0.6, 0.4], \
              [0.4, 0.6]])

# covariance
C = np.cov(X)

# eigen-decomposition
[D, V] = np.linalg.eigh(C)

# principal components
W = np.dot(np.diag(D), V)
Z = np.dot(W, X)
```



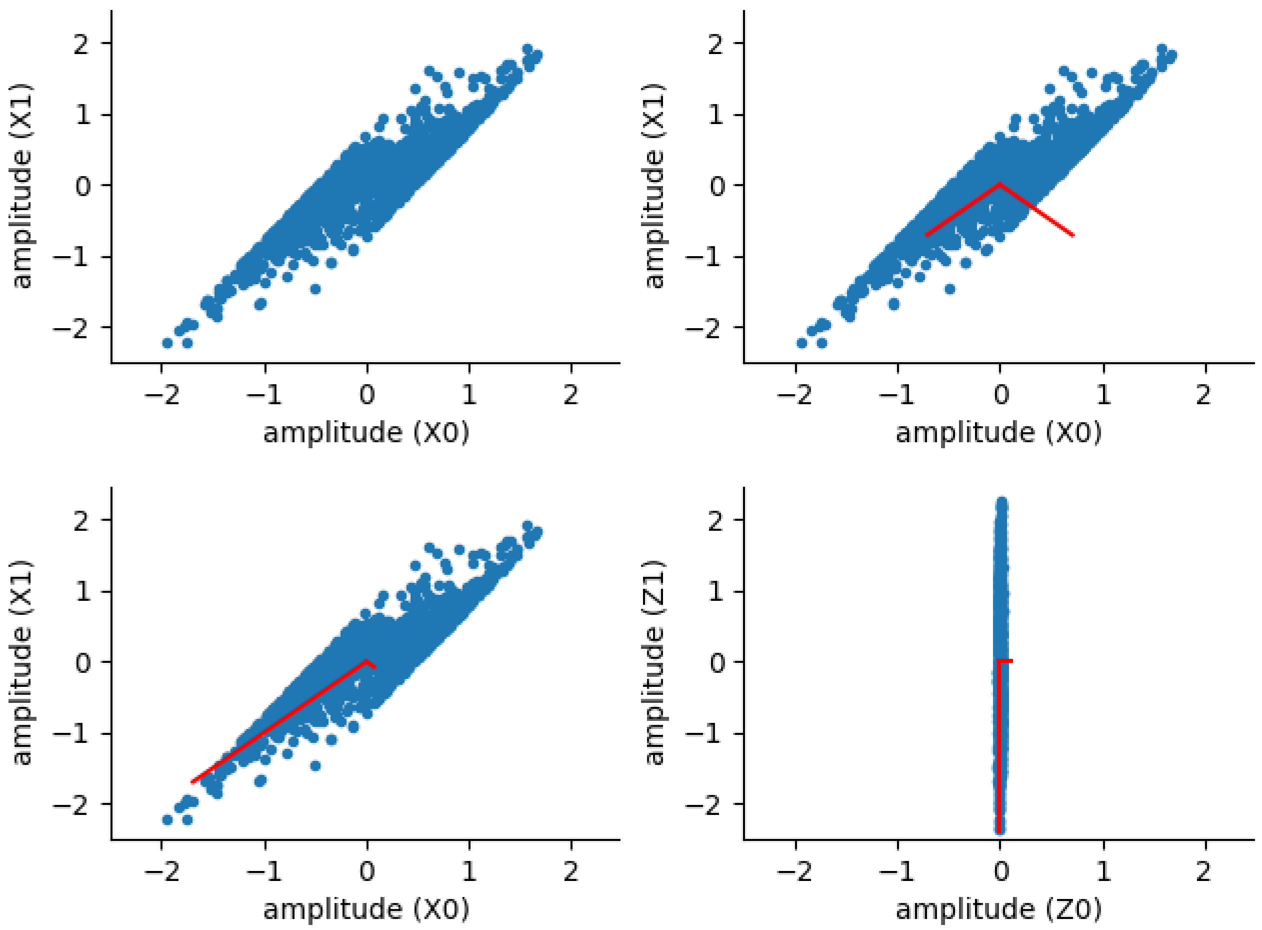


See, “L09_pca_2_sources.py”

Graphical interpretation

PCA can be thought of as fitting an n-dimensional **ellipsoid** to the data, where **each axis** of the ellipsoid represents a **principal component**.

```
# eigen-decomposition
[D, V] = np.linalg.eigh(C)
```



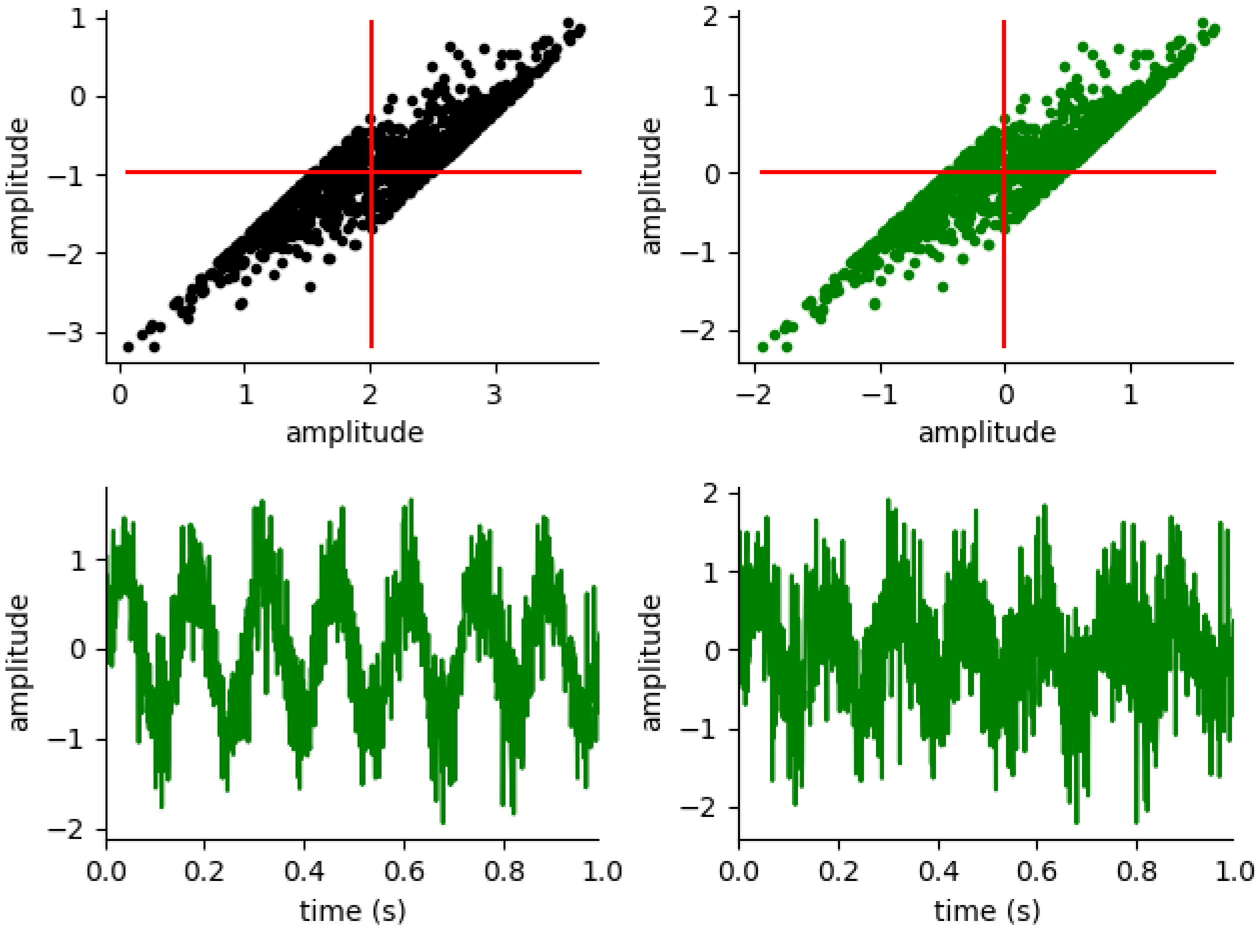

```
# V represents the coordinates of red lines
# D represents the length of red lines

# principal components
W = np.dot(np.diag(D), V)
Z = np.dot(W, X)
```

See, “L09_pca_2_sources.py”

Independent component analysis: zero mean

First, we need to remove mean from time series.

```
# remove mean
X = X - np.tile(np.mean(X, axis=1), (N, 1)).T
```

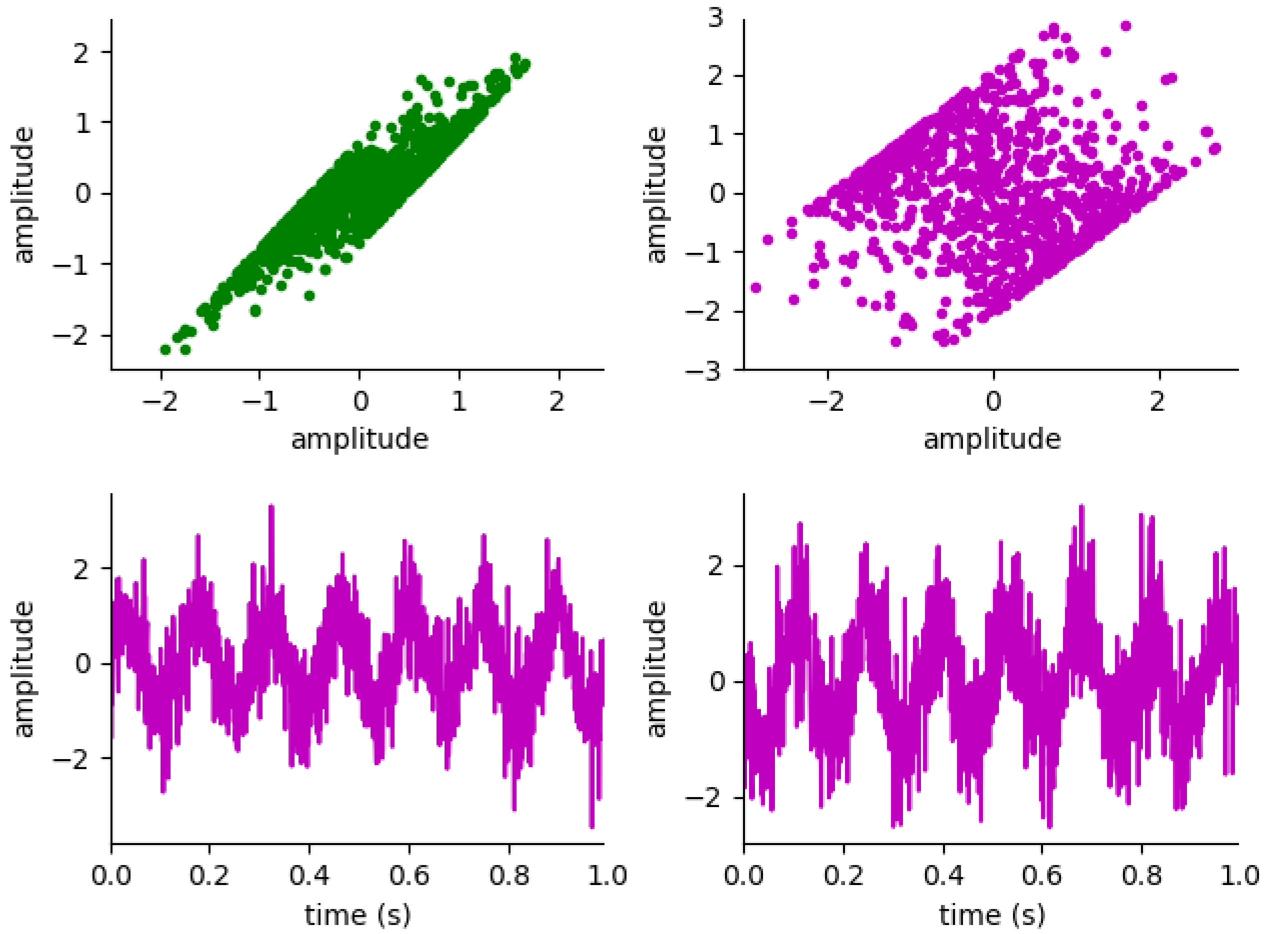


See, “L09_fpica_2_sources_steps.py”

Independent component analysis: whitening

Second, we need to remove linear dependency between time series.

```
# eigen-value decomposition
[D, E] = np.linalg.eigh(np.cov(X))
D = np.diag(D)

# whitening matrix
WM = np.dot(np.linalg.inv(np.sqrt(D)), E.T)

# whitened signals
Z = np.dot(WM, X)
```

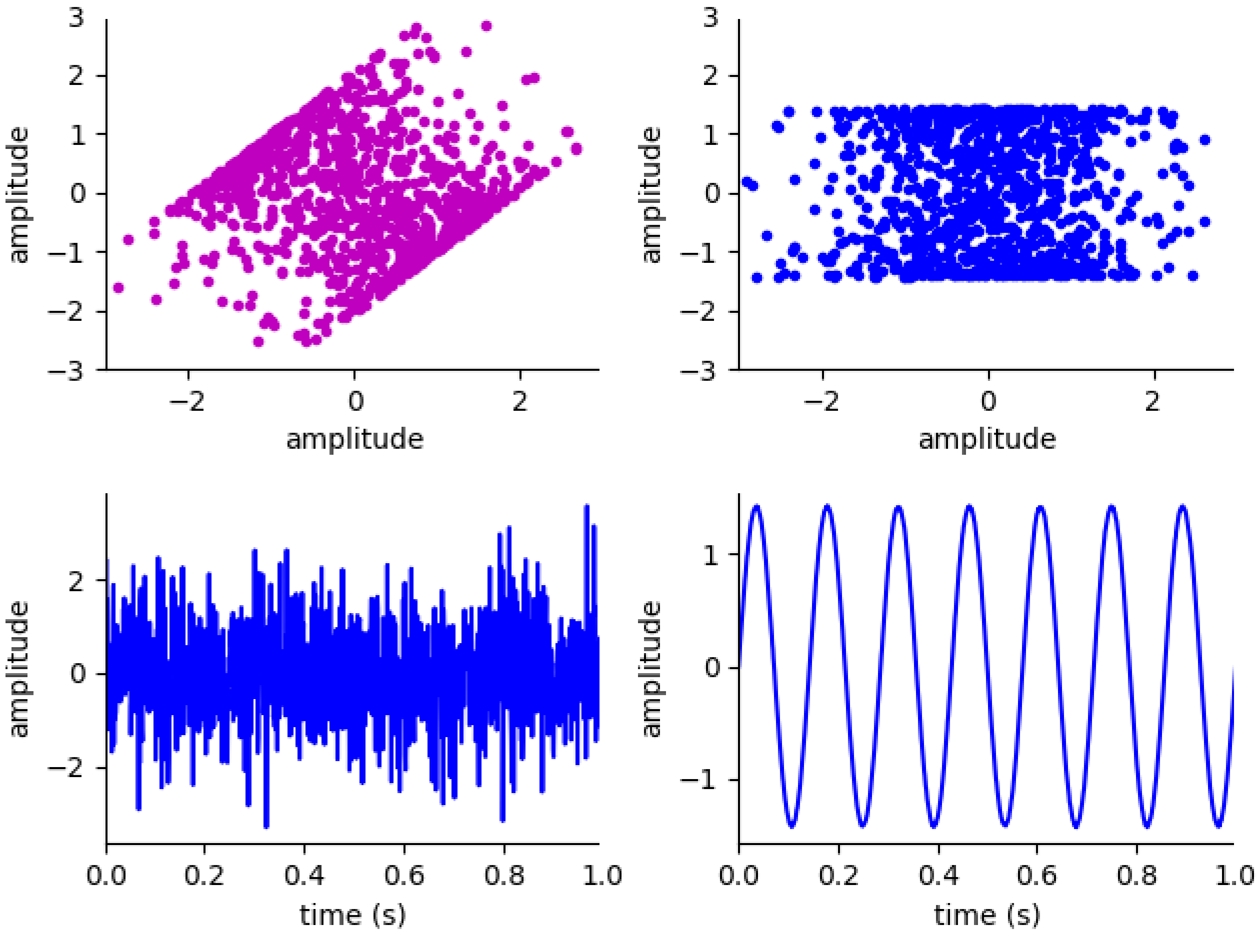


See, “L09_fpica_2_sources_steps.py”

Independent component analysis: un-mixing

Finally, we compute the un-mixing matrix to reconstruct the sources.

```
# routine
for i in range(0, max_iter):
  # orthogonalize B
  B = symmetric_decorrelation(B)
  # convergence condition
  minAbsCos = min(abs(diag(B' * BOld)))
  if (1 - minAbsCos < tol):
    break
  BOld = B # previous iteration
  # re-compute sources
  x = np.dot(Z.T, B)
  # compute nonlinearity
  g  = x * np.exp(-(x**2) / 2)
  dg = (1 - x**2) * np.exp(-(x**2) / 2)
  # updating rule
  B = np.dot(Z, g) - np.tile(np.sum(dg,
      axis=0), (n, 1)) * B
```

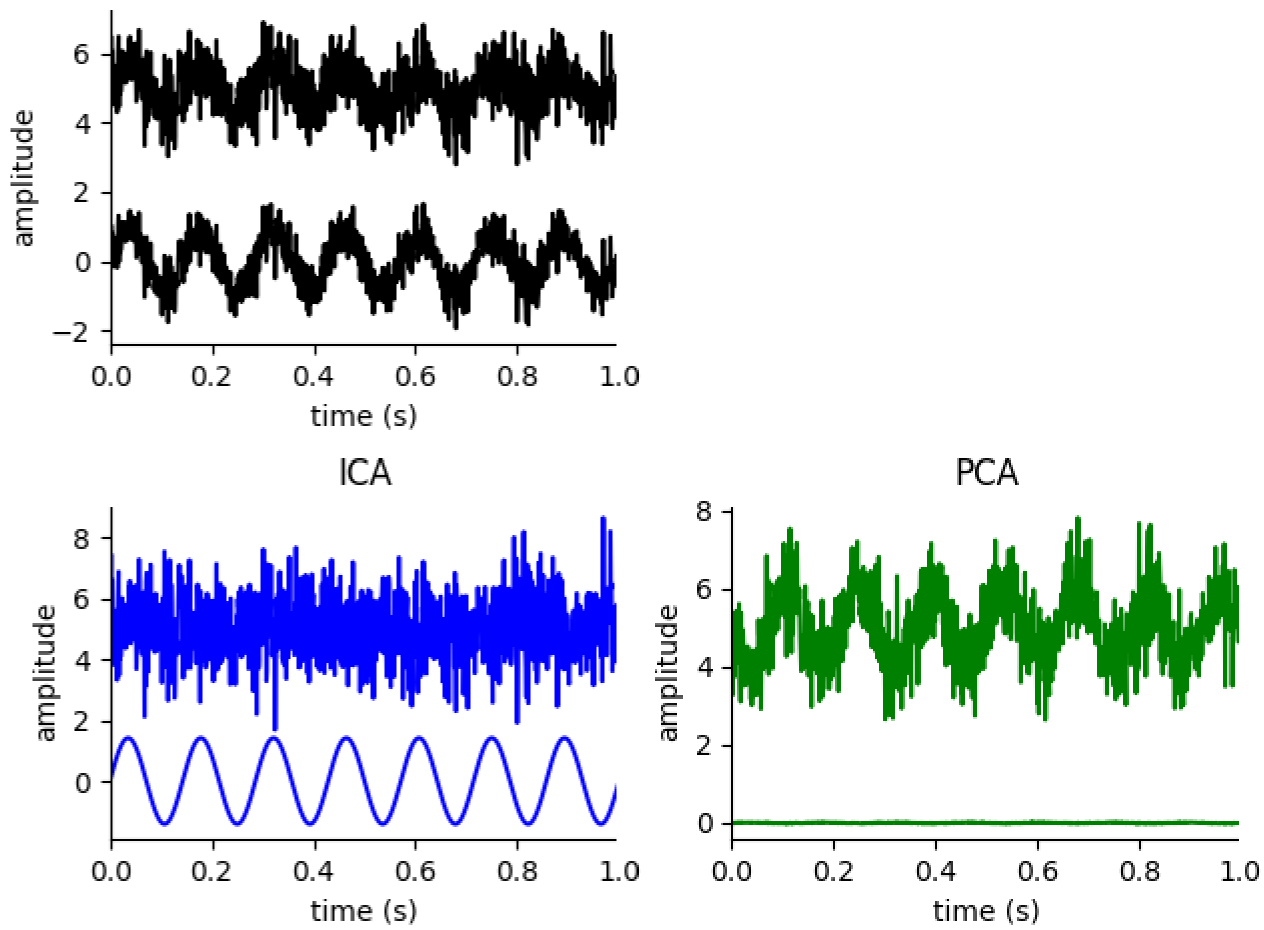


See, “L09_fpica_2_sources_steps.py”

ICA versus PCA

What is the difference between ICA and PCA?

```
# mixing
A = np.array([[0.6, 0.4], \
              [0.4, 0.6]])
X = np.dot(A, S)


# ica
W = fpica(Y, max_iter=1000, tol=1e-4)

# pca
[D, V] = np.linalg.eigh(np.cov(X))
Q = np.dot(np.diag(D), V.T)

# unmixing
ICA = np.dot(W, X)
PCA = np.dot(P, X)
```

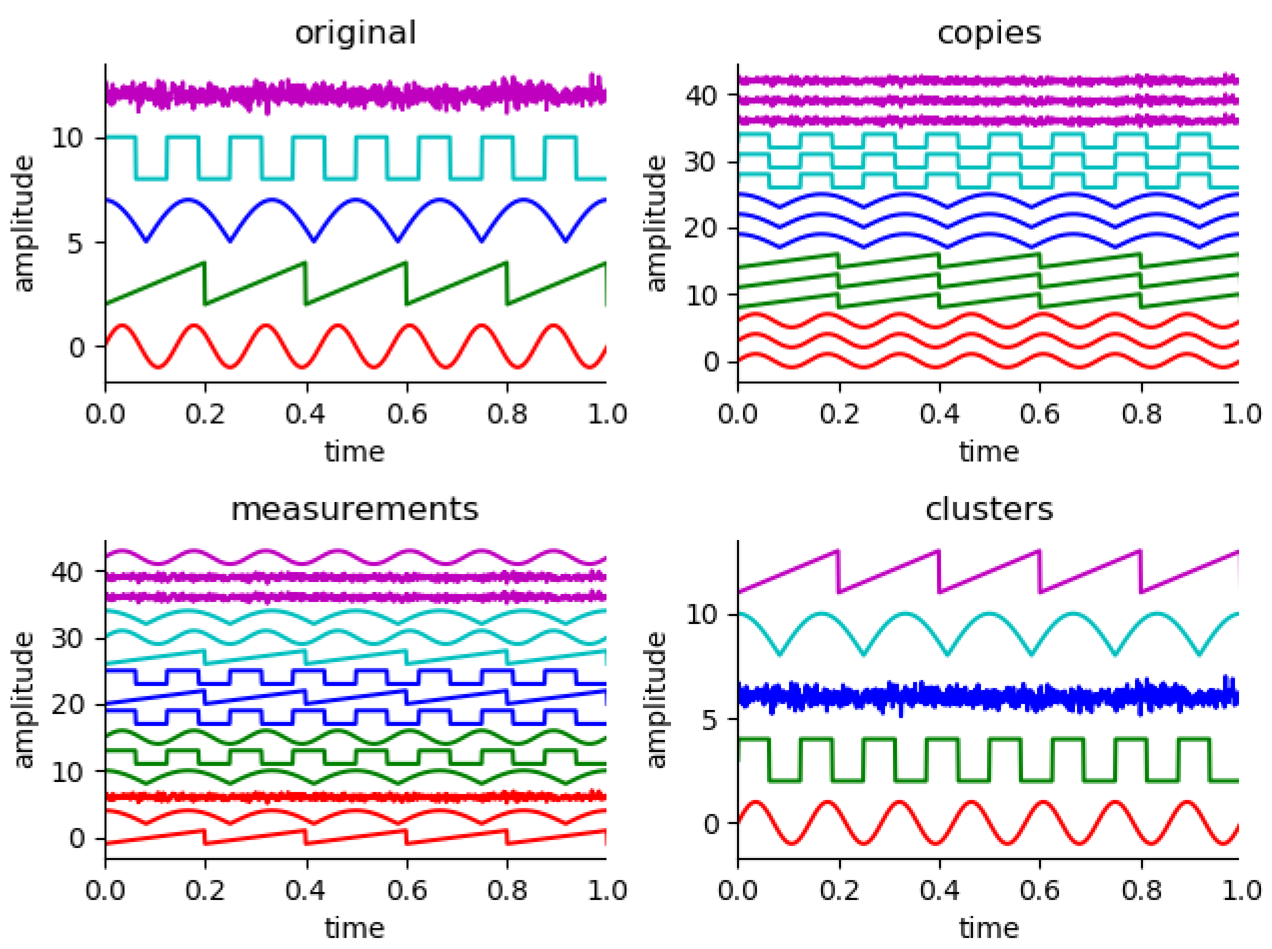


See, “L09_fpica_2_sources_vs_PCA.py”

K-means clustering (1/2)

The number of clusters (K) is equal to the number of sources.

```
# create copies
X[i] = np.tile(S[i, :], (R, 1)) +
       np.random.randn(R, N) * SNR

# measurements
Y = X[np.random.permutation(M*R), :]


# clustering using sklearn
model = cluster.KMeans(n_clusters=K)
model.fit(Y)

# clustering outcome
labels = model.labels_
Z = model.cluster_centers_
inertia = model.inertia_
print(inertia) # within-cluster sum-of-squares
```



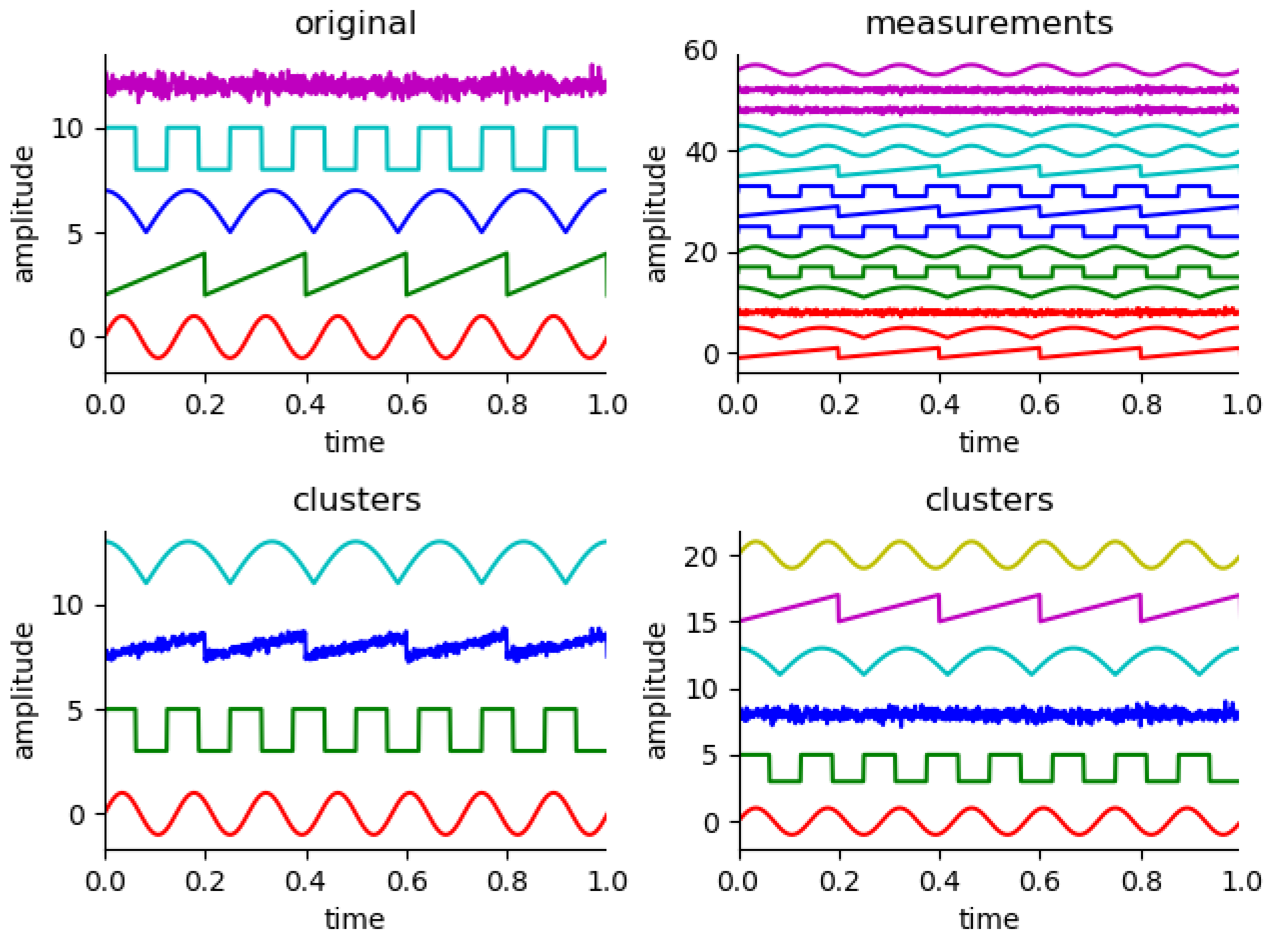
inertia = 0.0

See, “L10_clustering_kmeans.py”

K-means clustering (2/2)

The number of clusters (K) is greater or less than the number of sources.

```
# create copies
X[i] = np.tile(S[i, :], (R, 1)) +
       np.random.randn(R, N) * SNR

# measurements
Y = X[np.random.permutation(M*R), :]


# clustering using sklearn
model = cluster.KMeans(n_clusters=K)
model.fit(Y)

# clustering outcome
labels = model.labels_
Z = model.cluster_centers_
inertia = model.inertia_
print(inertia)
```



inertia = 603.5

inertia = 0.0

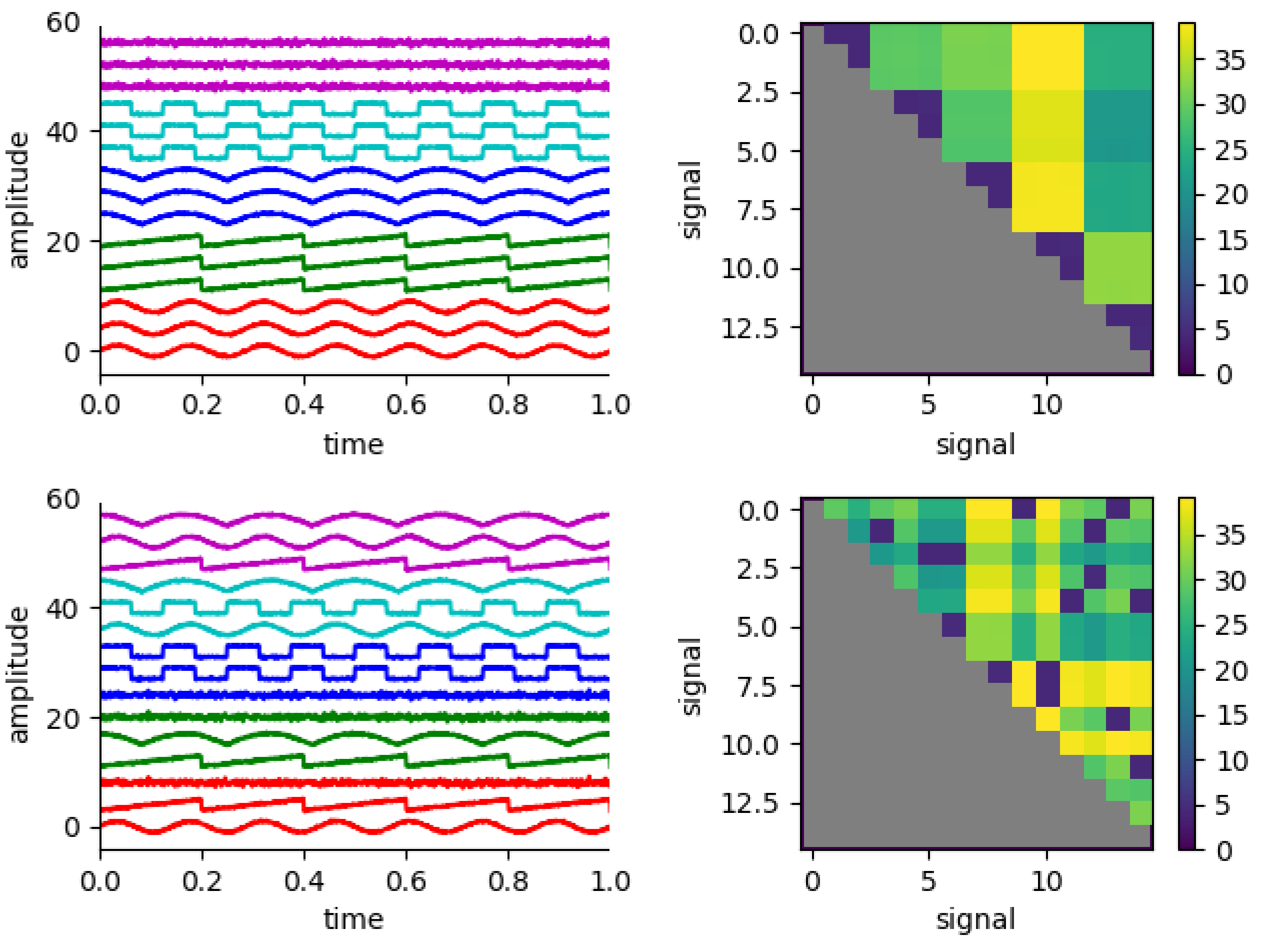
See, “L10_clustering_kmeans.py”

Hierarchical clustering (1/2)

What are the distance measures between signals?

```
# pair-wise distance between signals
PX = np.zeros((MR, MR))
PX[np.triu_indices(MR, 1)] = pdist(X,
                             'euclidean')
```

```
# distance after permutation
PY = np.zeros((MR, MR))
    PY[np.triu_indices(MR, 1)] = pdist(Y,
                                 'euclidean')
```

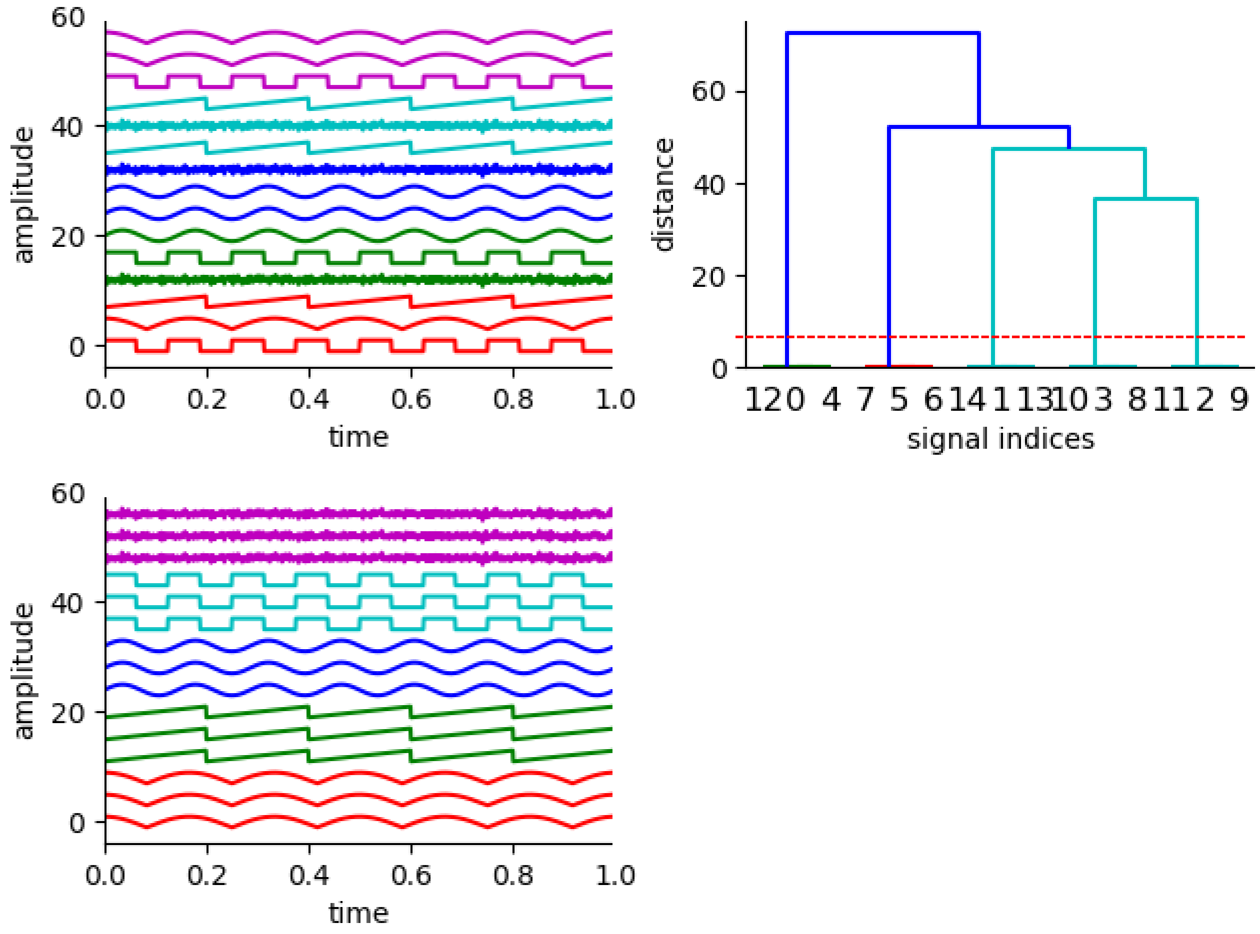


See, “L10_clustering_hierarchical.py”

Hierarchical clustering (2/2)

How does it work?

```
# clustering
model =
  cluster.AgglomerativeClustering()
model.fit(Y)
labels = model.labels_
children = model.children_
```



See, “L10_clustering_hierarchical.py”

Section 7. Classification

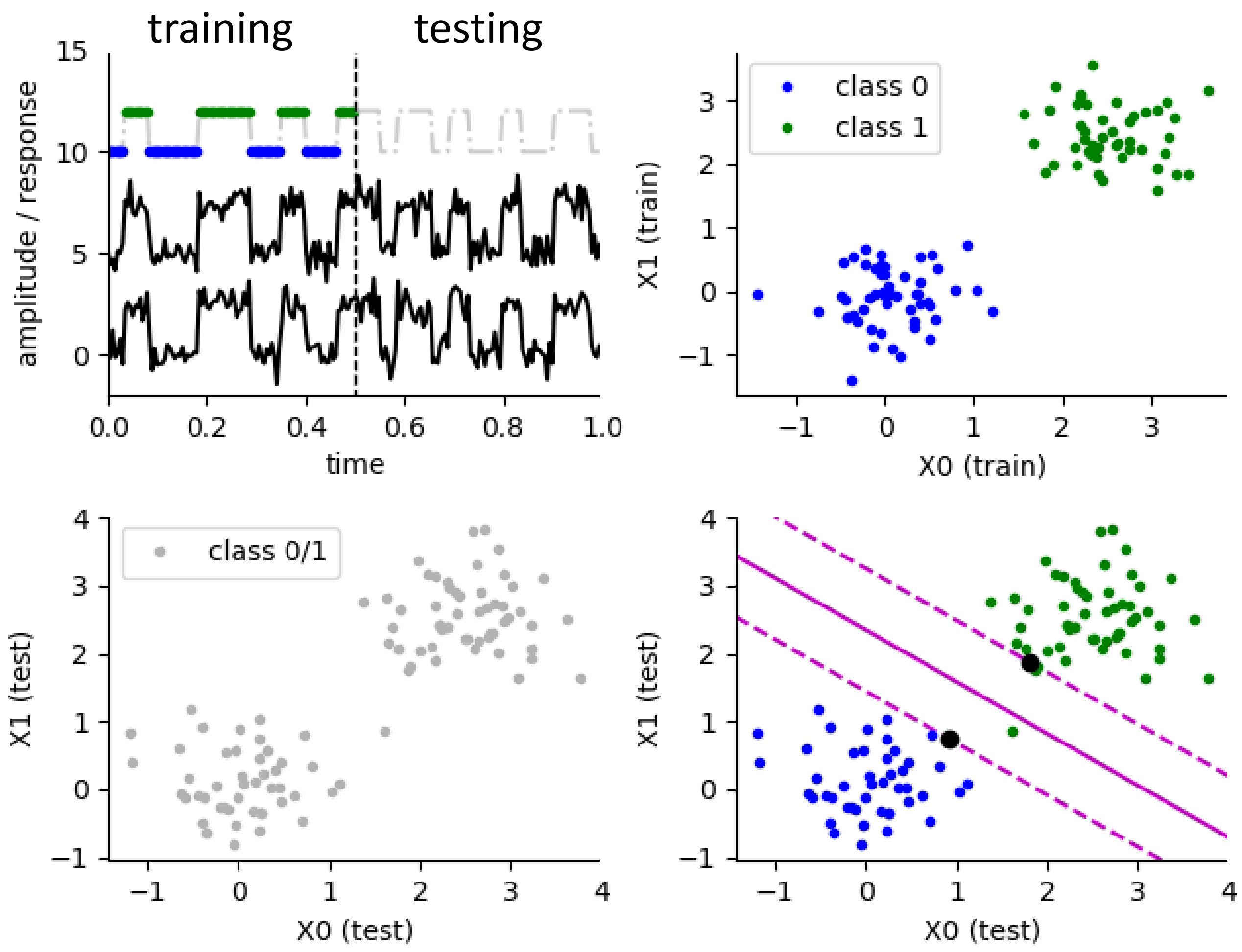
Classification and regression

- Support Vector Machine, Logistic regression
- Support Vector Regression, Linear regression

Support Vector Machine (1/2)

SVM as any other classification approach consists of two stages: training and testing.

```
# data
X = np.random.randn(M, N)

# binary labels
y = get_sequence(5, 0.8, N)

# induce some correlation between X and y
X = X + 2.0 * np.tile(y, (M, 1))

# training and testing datasets
L = N // 2
Y = y[:L] # training labels
U = y[L:] # testing labels
XY = X[:, :L] # training data
XU = X[:, L:] # testing data

# train classifier
model = SVC(kernel='linear')
model.fit(XY.T, Y)
```

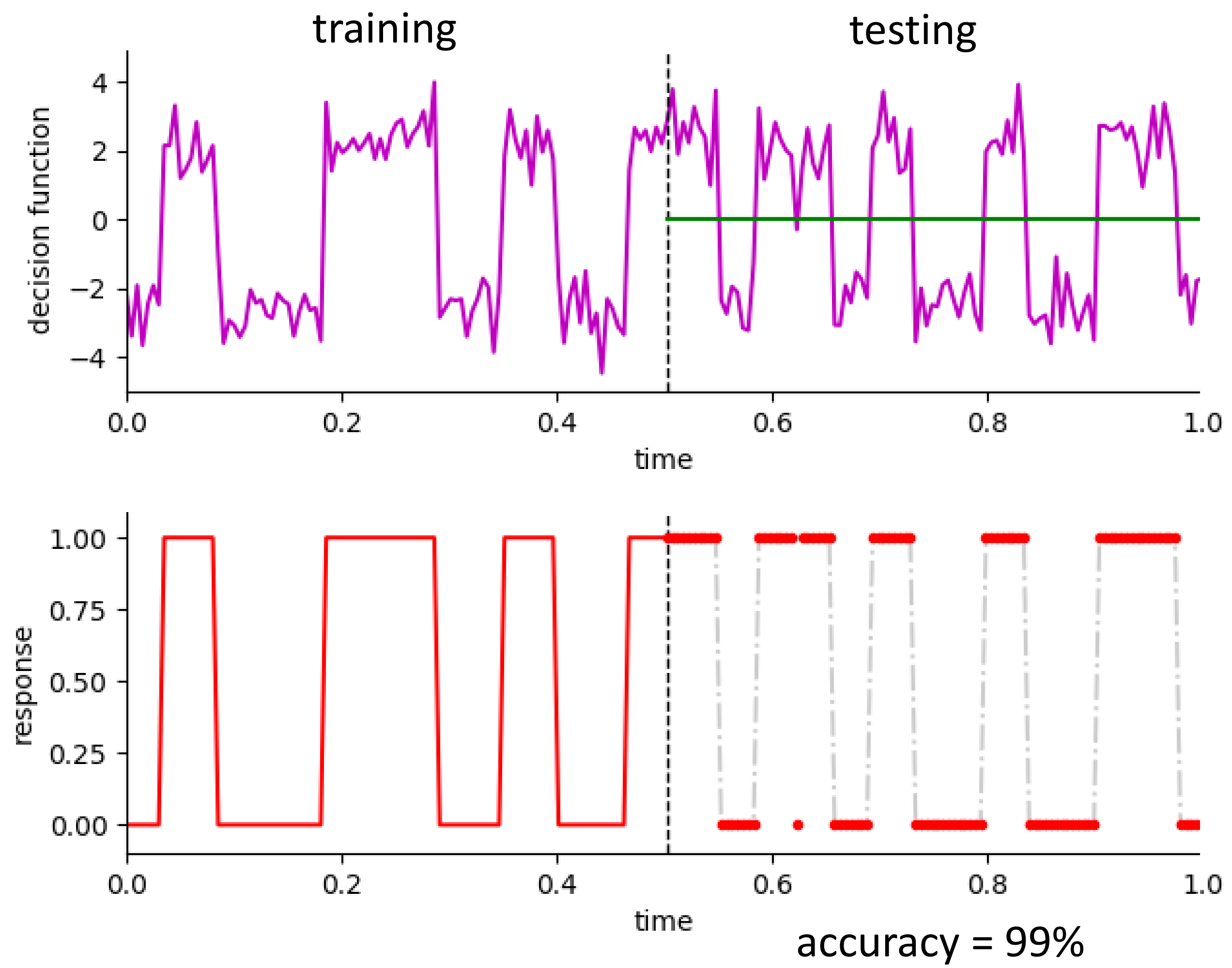


See, “L10_classification_svm_2_signals.py”

Support Vector Machine (2/2)

The classifier gives the coefficients that can be converted to the decision function.

```
# classifier outcome
coef = model._get_coef()
intercept = model.intercept_

# decision function
Z = np.zeros(N)
for i in range(0, N):
  Z[i] = np.sum(X[:, i] * coef) + intercept

# testing
v = U
u = model.predict(XU.T)
u = u > 0.5

# accuracy
a = np.mean(v == u)
print('accuracy: %1.2f' % (a))
```

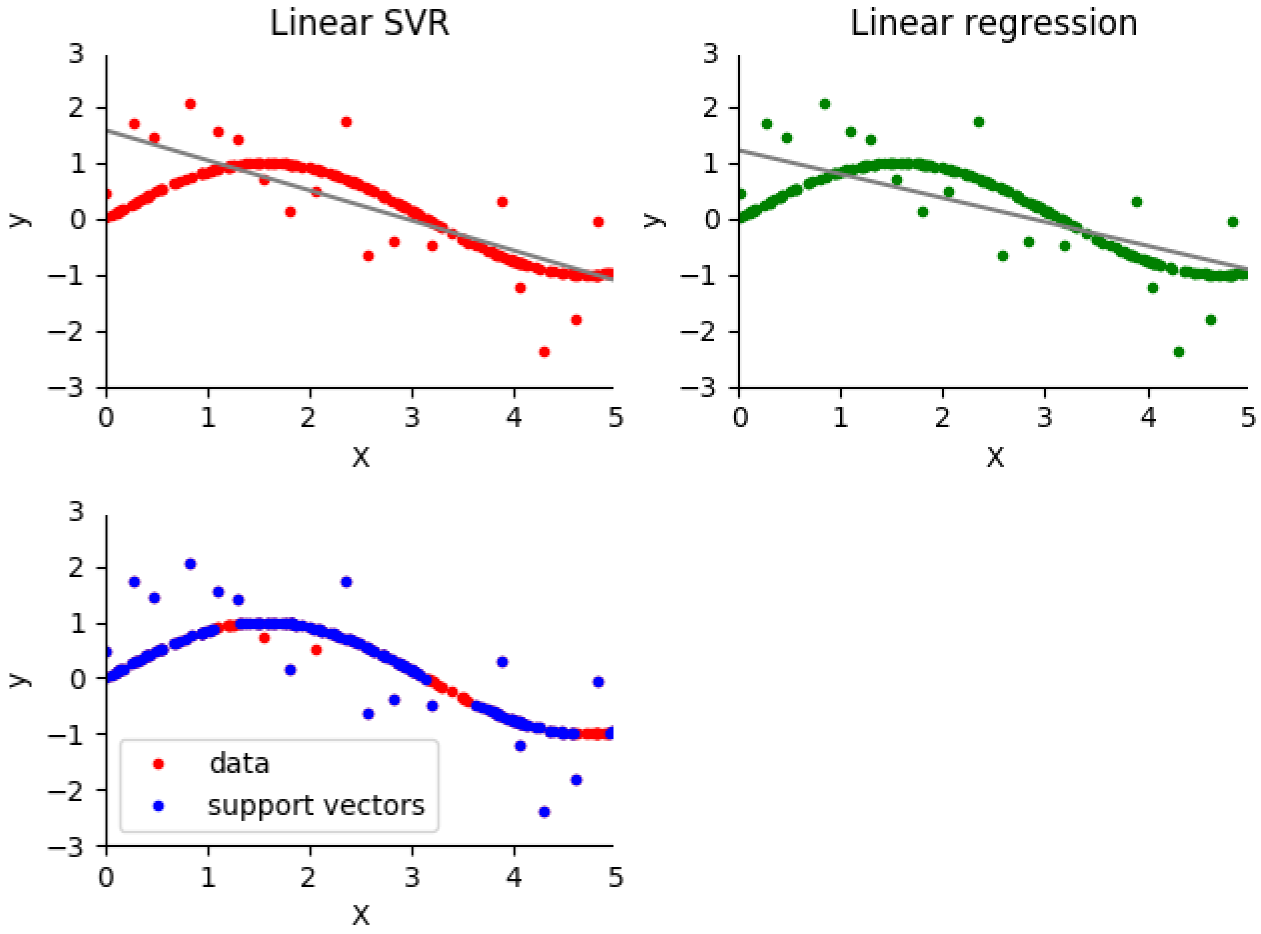


See, “L10_classification_svm_2_signals.py”

Support Vector Regression (1/2)

What is the difference between SVR and linear regression?

```
# fit model
model_svr = SVR(kernel='linear', C=1e3)
model_svr.fit(X, y)
model_lin = LinearRegression()
model_lin.fit(X, y)

# predict
u = model_svr.predict(X)
v = model_lin.predict(X)

# support vectors
support_vector_indices = model_svr.support_
```

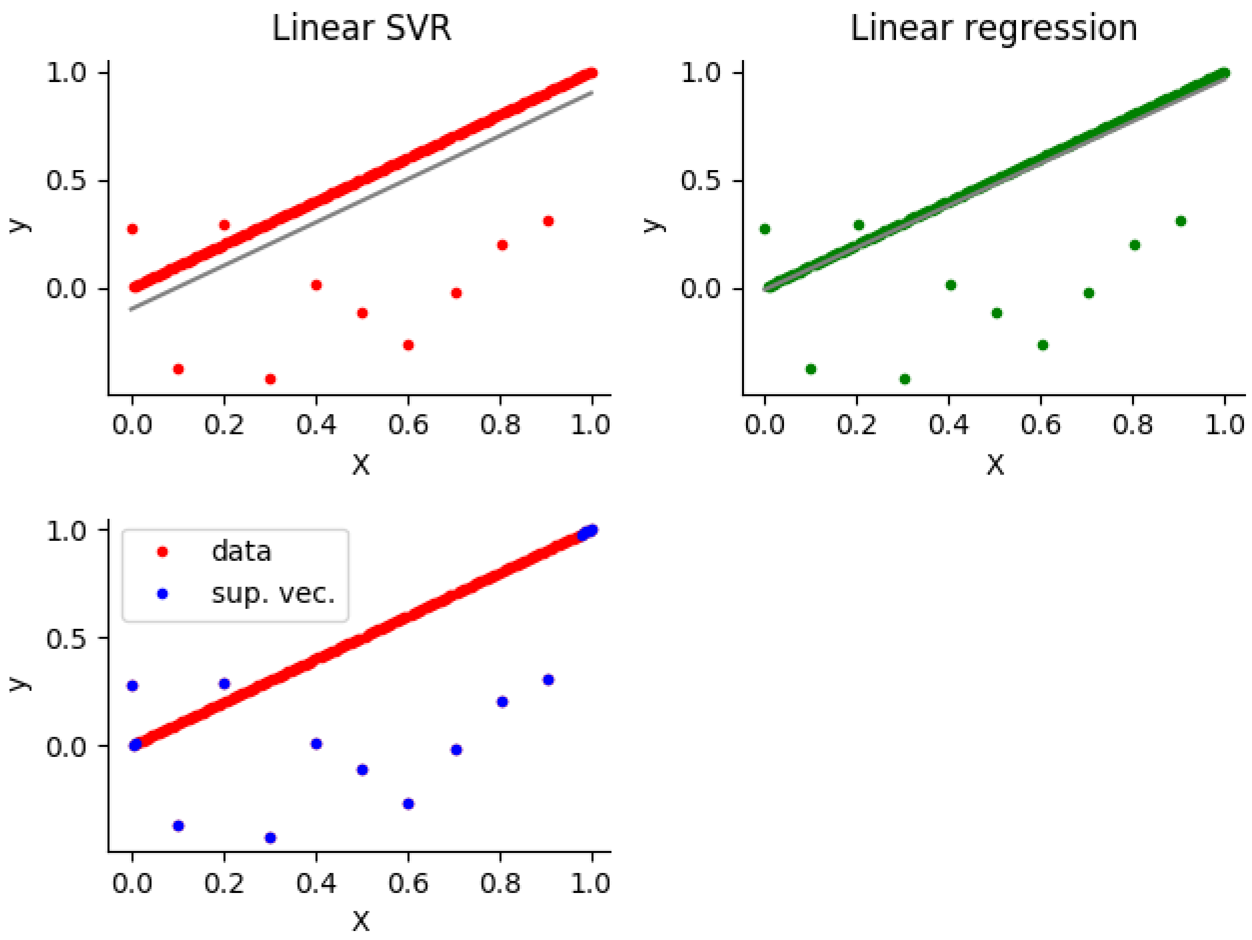


See, “L11_regression_svr.py”

Support Vector Regression (2/2)

What is the difference between SVR and linear regression?

```
# fit model
model_svr = SVR(kernel='linear', C=1e3)
model_svr.fit(X, y)
model_lin = LinearRegression()
model_lin.fit(X, y)

# predict
u = model_svr.predict(X)
v = model_lin.predict(X)

# support vectors
support_vector_indices = model_svr.support_
```



See, “L11_regression_svr_2.py”

Logistic regression

What is the difference between logistic and linear regression?

Outcome

In *linear regression*, the outcome (dependent variable) is continuous. It can have any one of an infinite number of possible values.

In *logistic regression*, the outcome (dependent variable) has only a limited number of possible values.