Lecture 9. Principal and independent component analysis

Outline / overview

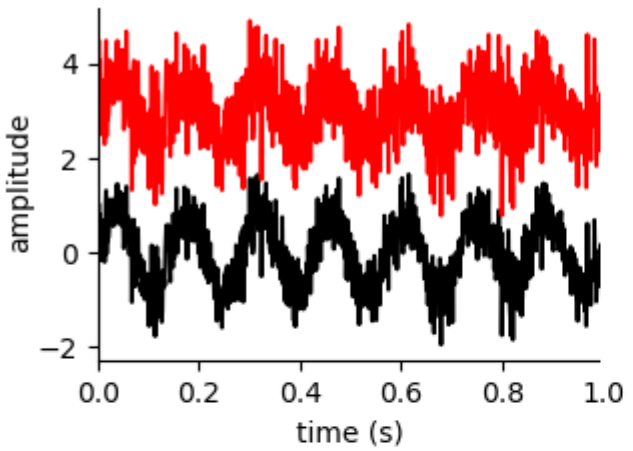
- **Section 1.** Covariance
- **Section 2.** Principal component analysis
- **Section 3.** Independent component analysis

Section 1. **Covariance**

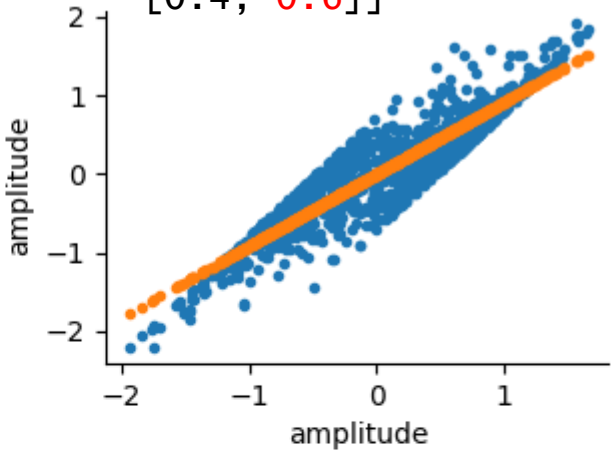
Covariance

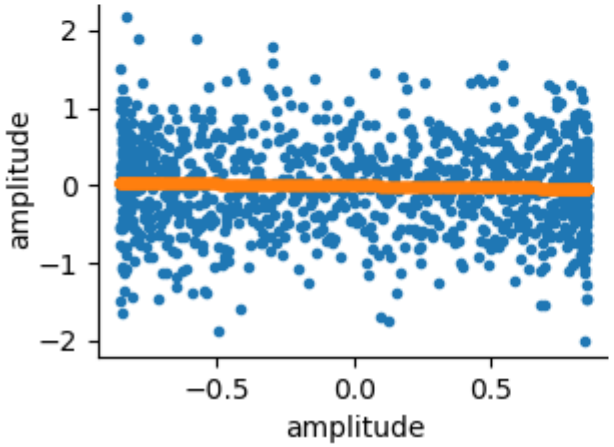
What would be the graphical interpretation of covariance?

```
# sources
S1 = np.sin(2 * np.pi * 7 * t)
S2 = np.random.randn(N)
# mixing
A = np.array([[0.6, 0.4], \
              [0.4, 0.6]])
X = np.dot(A, S)

# covariance
C = np.sum((X0 - np.mean(X0)) *
           (X1 - np.mean(X1))) / N

# linear relationship between X0 and X1
p = np.polyfit(X0, X1, 1)
X2 = p[0] * X0 + p[1]

# C is proportional to p[0] (slope)
```

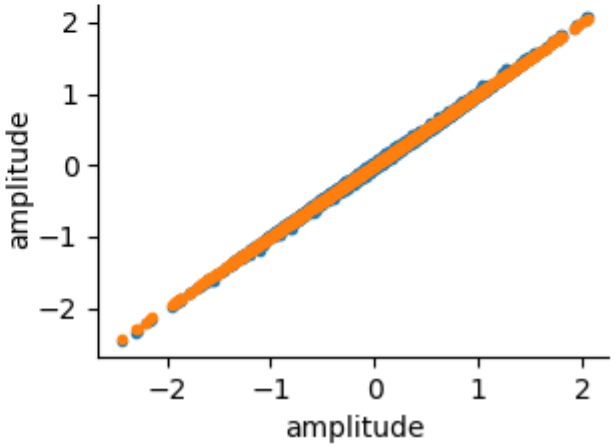


[[0.6, 0.4],
 [0.4, 0.6]]





[[0.6, 0.0],
 [0.0, 0.6]]



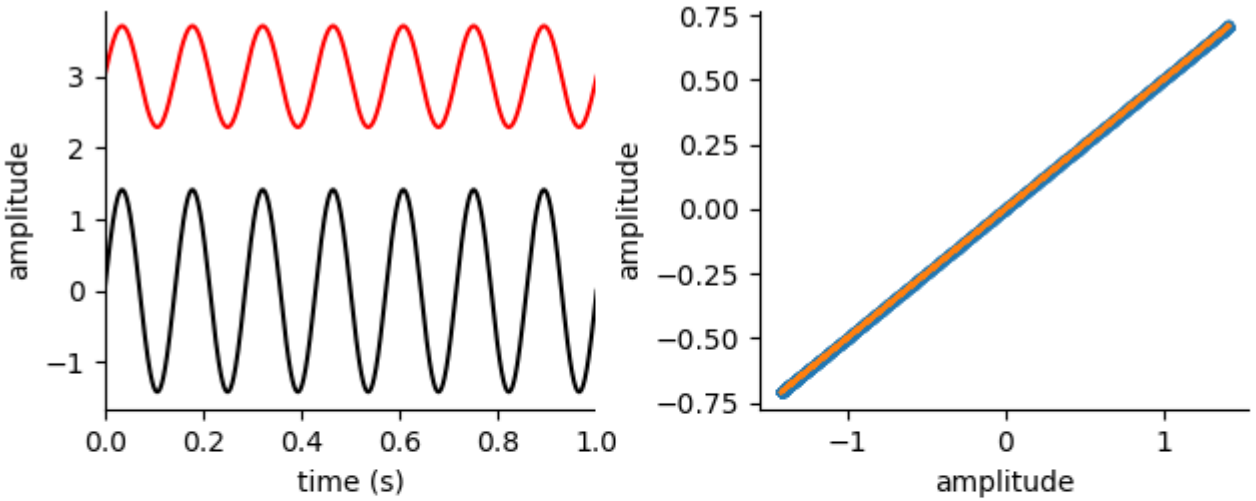
[[0.6, 0.6],
 [0.6, 0.6]]

See, "L10_covariance_2_sources.py"

Linear dependency

Zero-mean X0 and X1 are **linearly dependent** if X0 = k * X1

```
# sources
S1 = np.sin(2 * np.pi * 7 * t)
S2 = np.random.randn(N)
# mixing
A = np.array([[1.0, 0.0], \
              [0.5, 0.0]])
X = np.dot(A, S)
```



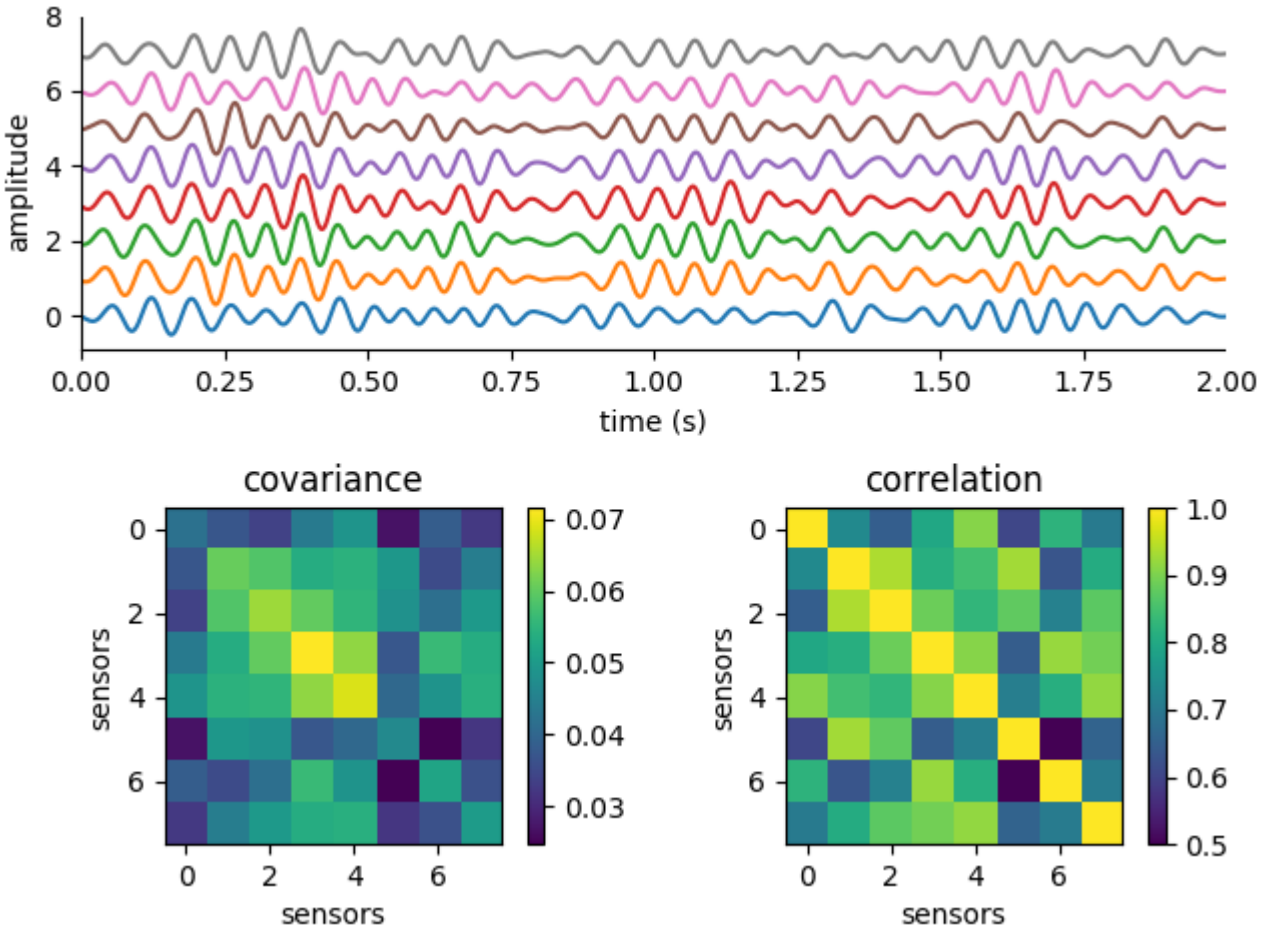

```
# linear dependency estimation
p = np.polyfit(X0, X1, 1)
X2 = p[0] * X0 + p[1] # p[0] == slope

# slope via ratio
p0r = np.sum(X1 / X0) / N

# slope via covariance
p0c = np.cov(X)[0,1]
```

See, “L10_covariance_2_sources_lin_dep.py”

Covariance matrix

What is the difference between covariance and correlation matrix?

```
# sources
S = np.random.randn(M, N)

# mixing
A = np.random.rand(M, M)
X = np.dot(A, S)


# covariance
C = np.zeros((M, M))
for i in range(0, M):
  for j in range(0, M):
  C[i,j] = np.sum((X[i] - np.mean(X[i])) *
                  (X[j] - np.mean(X[j]))) / N

# correlation
R = np.corrcoef(X)
```

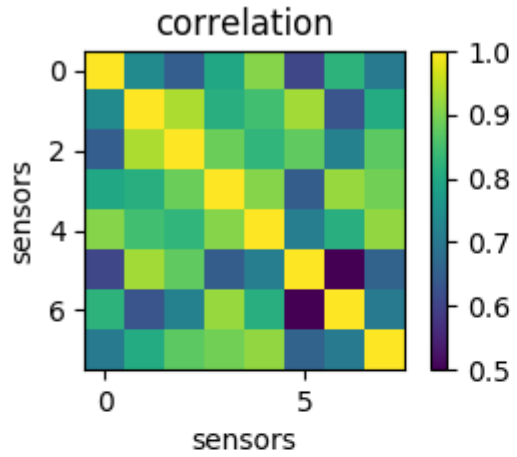


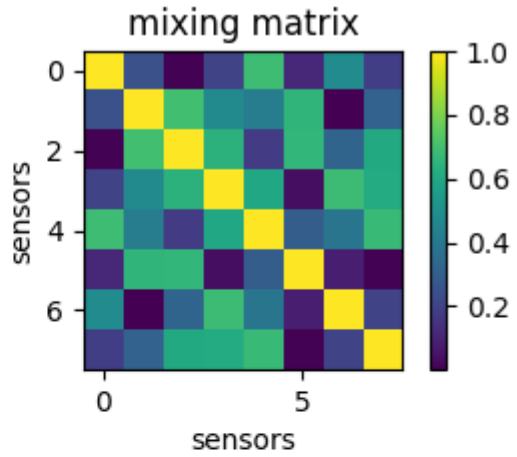
See, “L10_covariance.py”

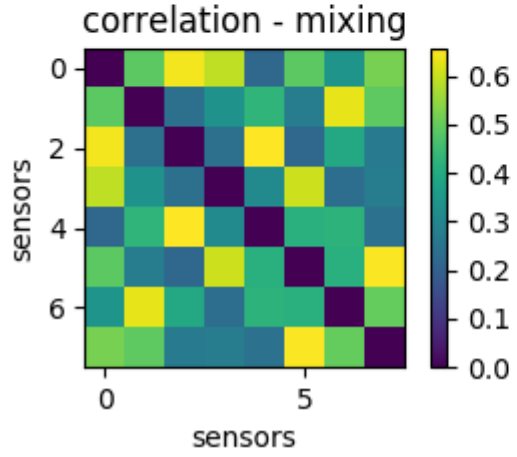
Covariance and mixing matrices

Covariance can be seen as an approximation of the original mixing matrix.

```
# covariance matrix
C = np.cov(X)

# correlation
R = np.corrcoef(X) # similar to mixing matrix A


# difference
B = R - A
```





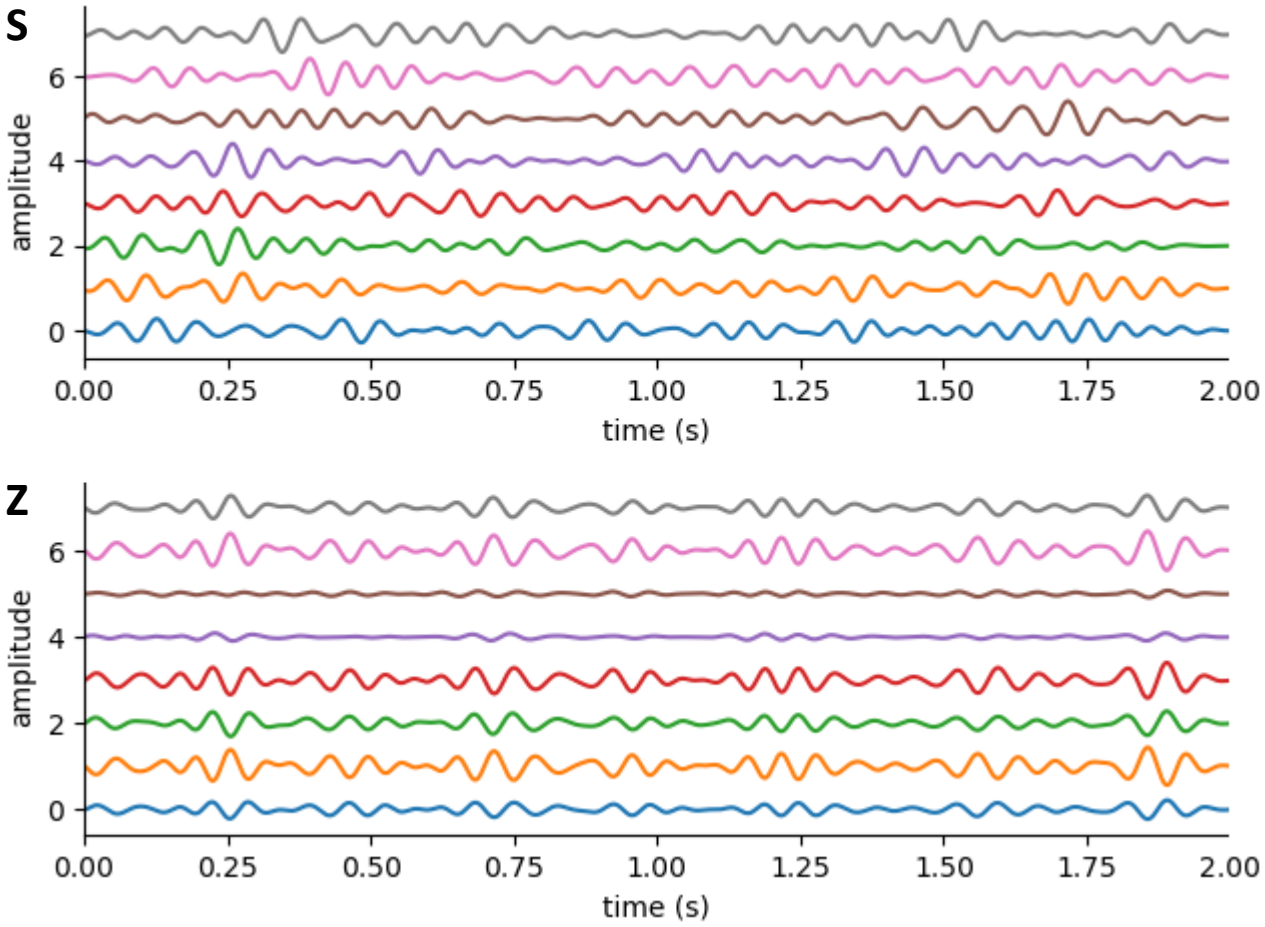


See, "L10_covariance.py"

Un-mixing matrix

We can reconstruct the original sources by inverting mixing matrix, **if A is known**.

```
# inverse mixing matrix
W = np.linalg.inv(A)

# reconstruct original sources
S = np.dot(W, X)


# such reconstruction does not work if we
simply invert the covariance matrix

# covariance
C = np.cov(X)

# un-mixing
H = np.linalg.inv(C) / N

# reconstruct sources
Z = np.dot(H, X)
```



See, "L10_covariance.py"

Section 2. **Principal component analysis**

PCA via eigen-decomposition

Eigen-decomposition decomposes a matrix (e.g., covariance matrix, **C**) into a multiplication of a matrix of eigenvectors (**V**) and a diagonal matrix of eigenvalues (**D**), so that

```
np.dot(C, V) = np.dot(D, V)
```

Eigenvectors (**V**) with distinct eigenvalues (**D**) are **linearly independent.** By multiplying data and eigenvectors of covariance matrix, we transform our data to a new space of linearly independent variables.

Eigenvalues and eigenvectors

```
D = [[0.7, 0.0],
     0.0, 0.2]]
```

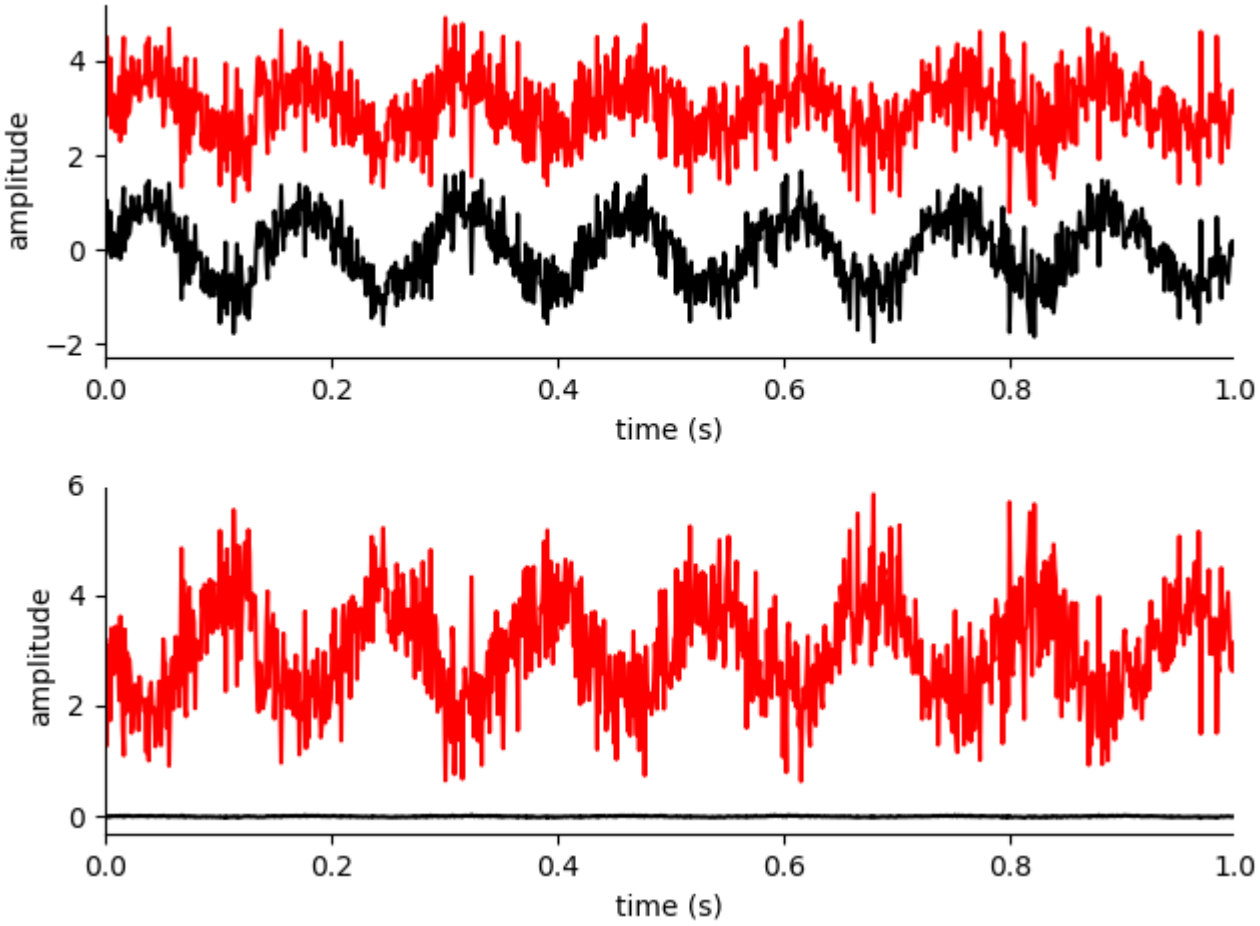
```
D * V = D[0,0] * V[:,0], D[1,1] * V[:,1]
```

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

Principal components

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)
```



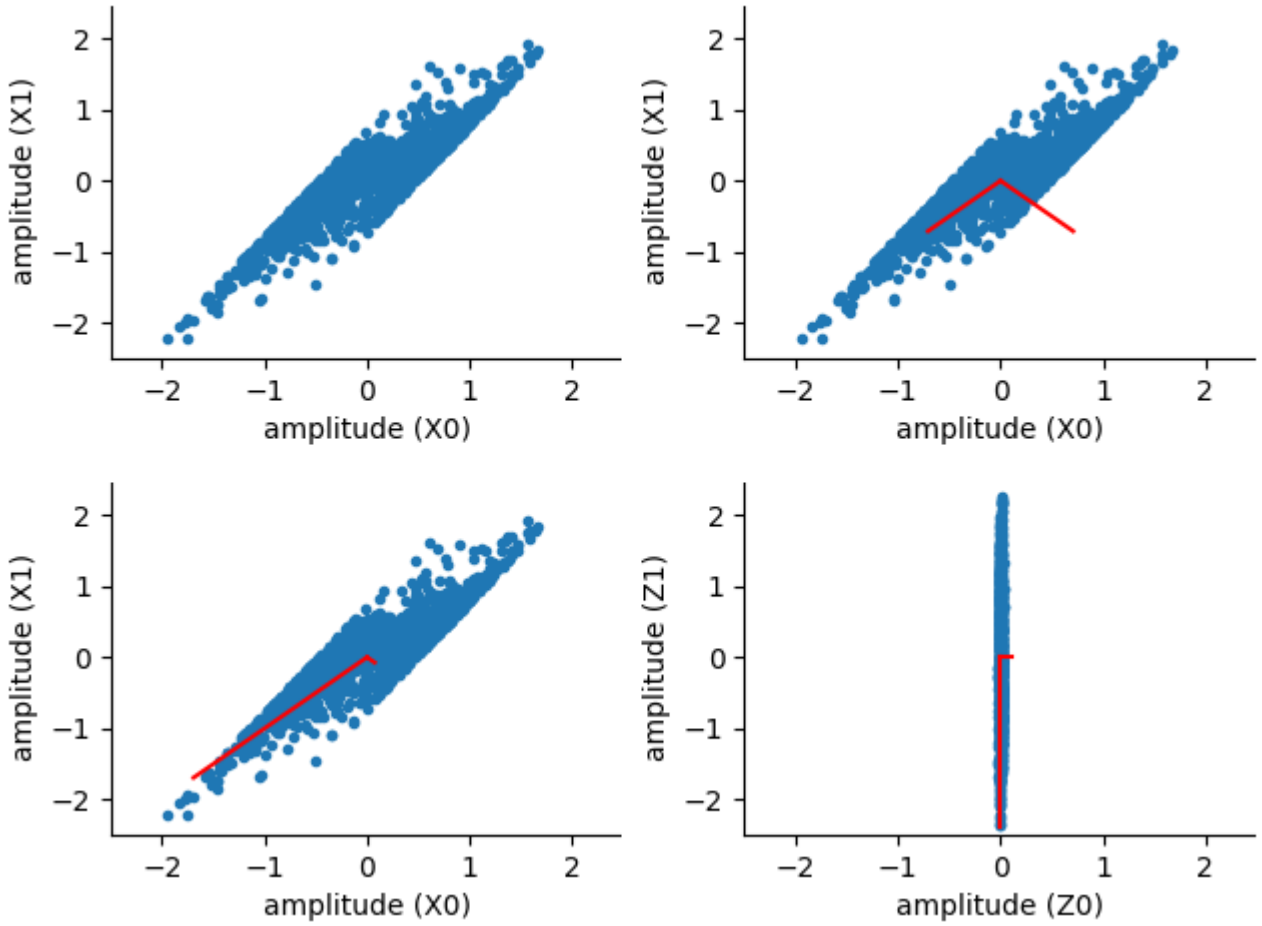
See, "L10_pca_2_sources.py"

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)
```

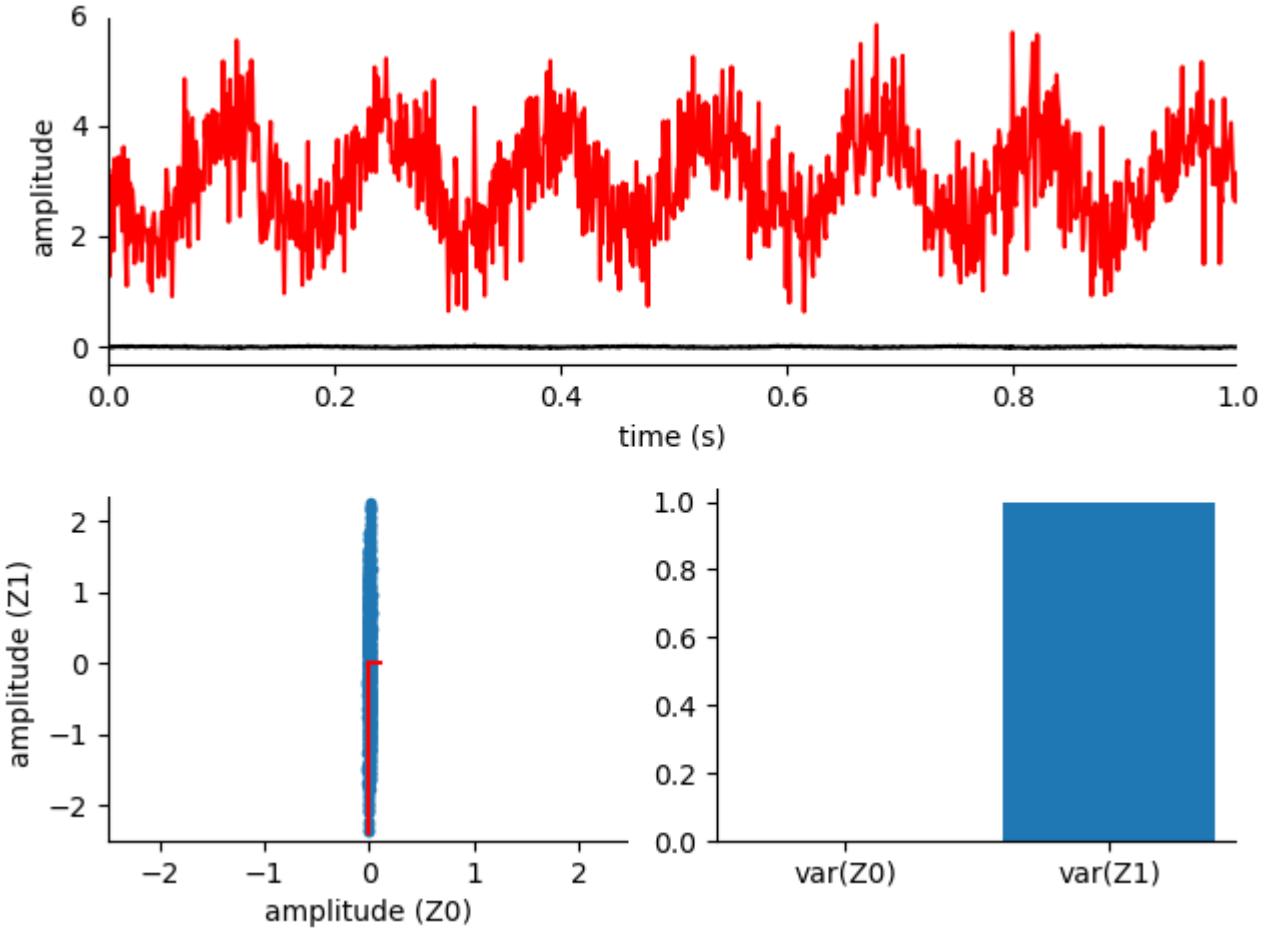


See, “L10_pca_2_sources.py”

Explained variance

Explained variance shows how much variance is explained by each component.

```
# covariance
C = np.cov(X)

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

# explained variance by components
explained_variance = (D ** 2) / (N - 1)

total_variance = np.sum(explained_variance)
explained_variance_ratio = explained_variance
/ total_variance
```

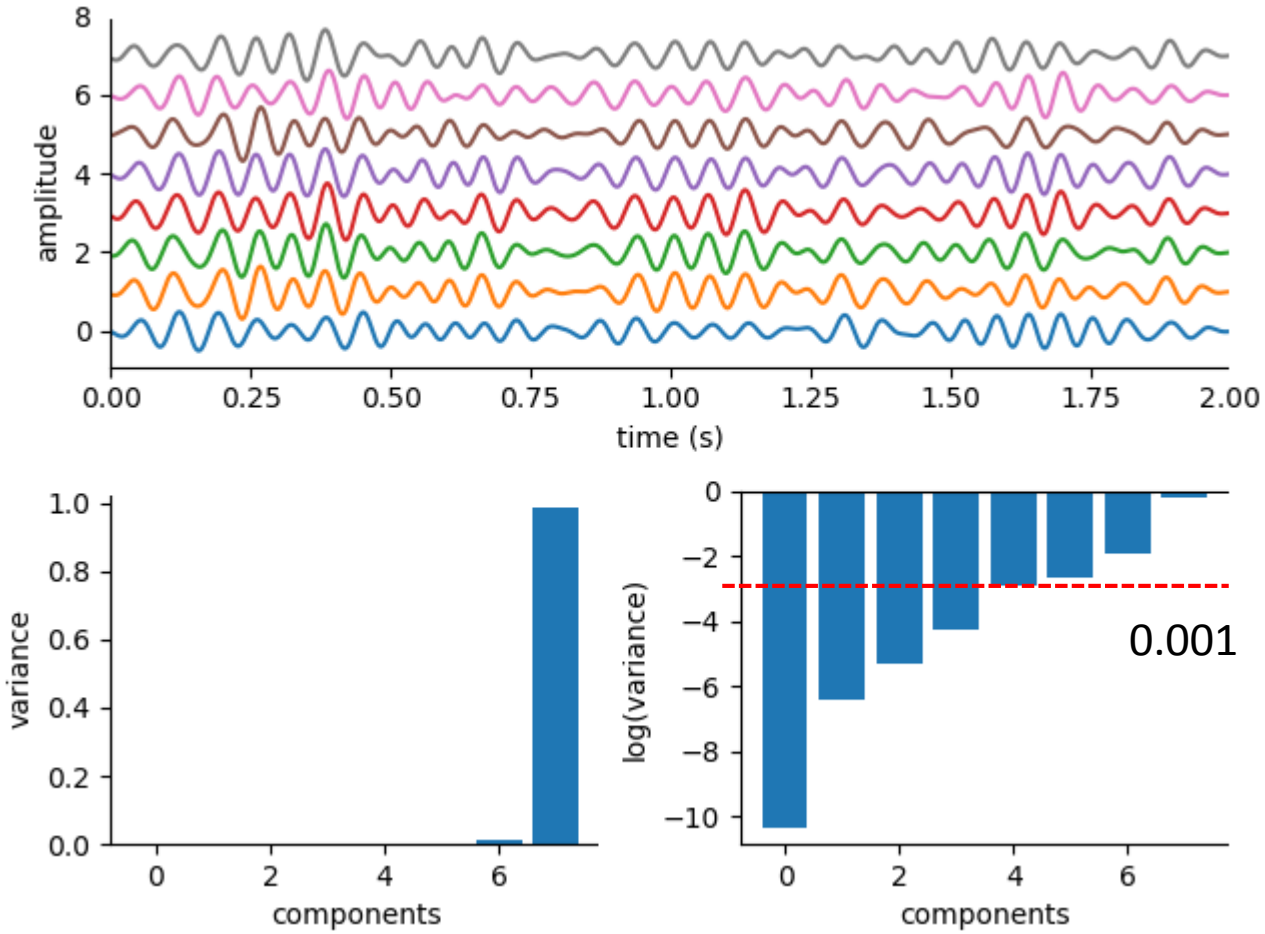


See, “L10_pca_2_sources.py”

Dimensionality reduction (1/2)

Could we exclude components that explain small portion of variance?

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

# explained variance by components
explained_variance = (D ** 2) / (N - 1)
explained_variance_ratio = explained_variance
/ np.sum(explained_variance)

# select components with high variance
threshold = 0.001
indices = np.where(explained_variance_ratio >
threshold))
```

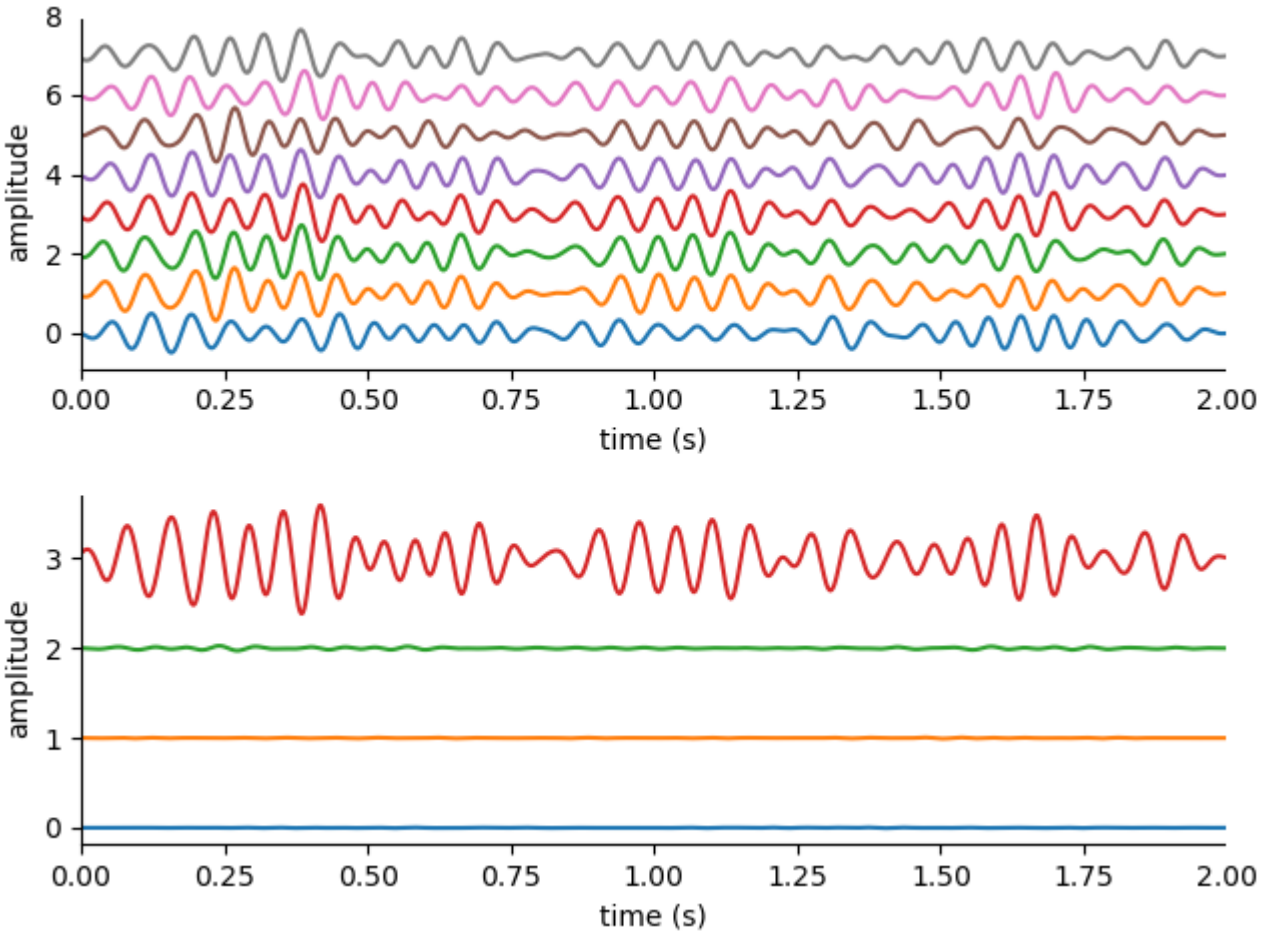


See, “L10_pca_dim_reduction.py”

Dimensionality reduction (2/2)

Could we exclude components that explain small portion of variance?

```
# select components with high variance
threshold = 0.001
indices = np.where(explained_variance_ratio > threshold))
V = V[:, indices]
D = D[indices]

# project data into orthogonal space
W = np.dot(np.diag(D), V.T)
Z = np.dot(W, X)
```

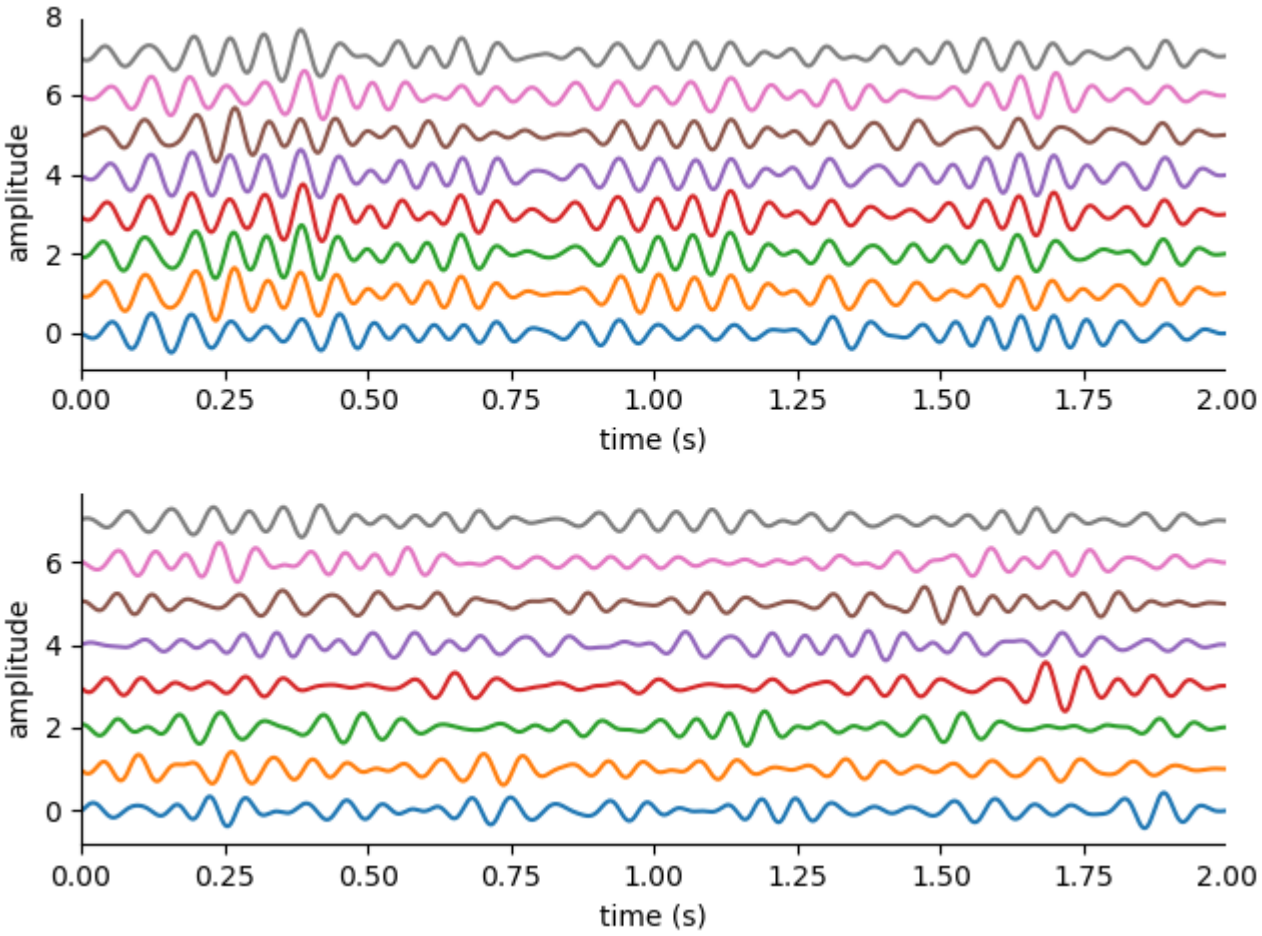


See, “L10_pca_dim_reduction.py”

Whitening (1/2)

What is whitening?

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

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

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

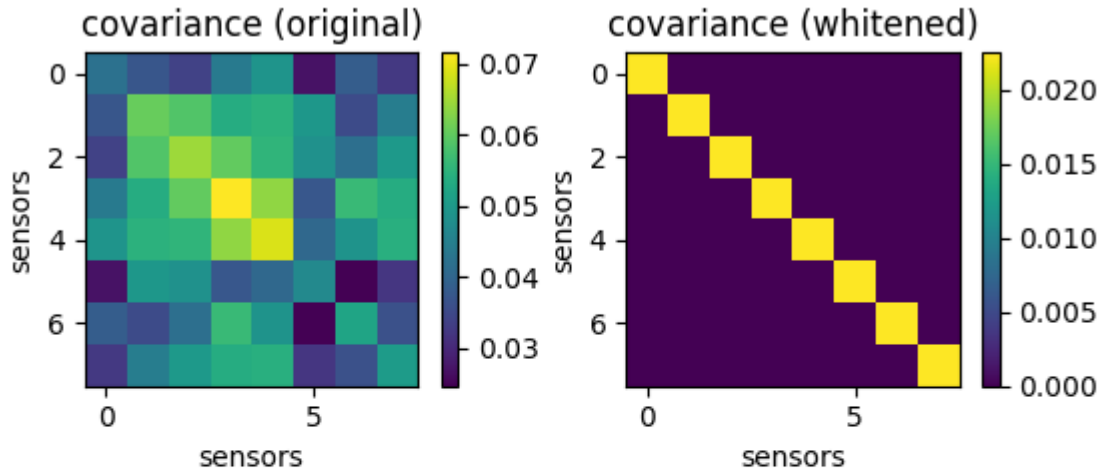


See, “L10_pca_whitening.py”

Whitening (1/2)

Whitening removes correlation between time series.

```
# covariance matrix (original)
plt.imshow(np.cov(X))
plt.colorbar()

# covariance matrix (whitened)
plt.imshow(np.cov(Z))
plt.colorbar()
```



See, “L10_pca_whitening.py”

scikit-learn implementation

```
from sklearn.decomposition import PCA

# init PCA
pca = PCA()

# reconstruct signals based on orthogonal components
H = pca.fit_transform(X)
```

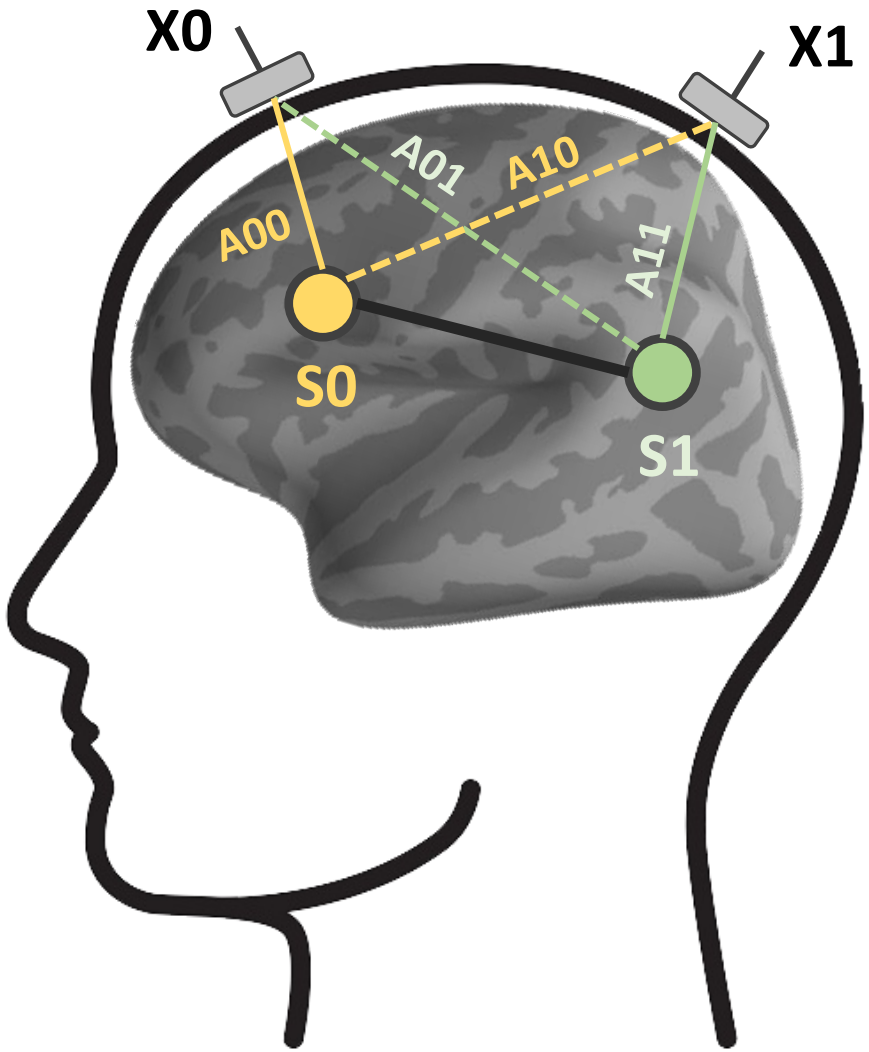
http://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html

Section 4. Independent component analysis

Problem statement

Is it possible to reconstruct sources **S** using measurements **X**?

```
# measurement model
X0 = A00 * S0 + A01 * S1
X1 = A10 * S0 + A11 * S1

# matrix notation
X = np.dot(A, S)

# inversion of the model
S0 = A00 * X0 + A01 * X1
S1 = A10 * X0 + A11 * X1

# find matrix W = inv(A)
# using only X measurements
S = W * X
```



See, “L10_fpica.py”

Assumptions in ICA

In general the model **cannot be estimated** without specific assumptions about the sources.

1. The independent components are assumed **statistically independent**

2. The independent components must have **non-gaussian distribution**

3. The number of independent components should be less or equal to the **number of observations**

Hyvarinen et al., 2001. Independent Component Analysis

Ambiguities of ICA

1. We cannot determine the variances of the independent components.

2. We cannot determine the order of the independent components.

Hyvarinen et al., 2001. Independent Component Analysis

ICA algorithm

ICA = contrast function + optimization algorithm

For instance in **fast-ICA** algorithm,

iteratively:

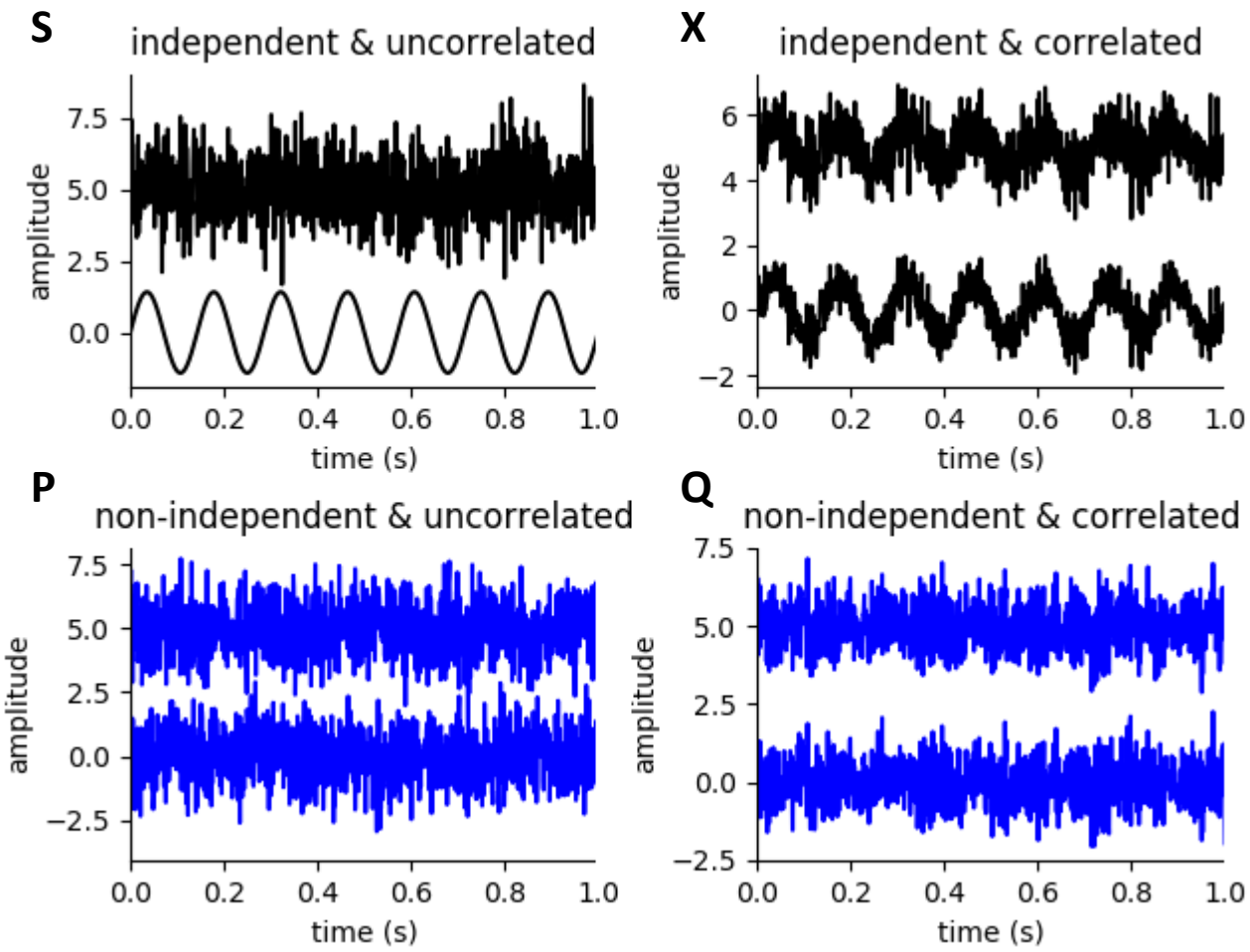
- contrast function is g(**x**) = exp(**x**^{2}) // measure the distance to gaussian variable
- optimization algorithm is *Newton method* // maximize the distance, i.e., make as non-gaussian as possible

Hyvarinen et al., 2001. Independent Component Analysis

Independence and uncorrelatedness (1/3)

What is the difference between independent and uncorrelated sources?

```
# sources
S0 = np.sin(2 * np.pi * 7 * t)
S1 = np.random.randn(N)
P0 = np.random.randn(N)
P1 = np.random.randn(N)

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

# mix sources
X = np.dot(A, S)
Q = np.dot(A, P)
```

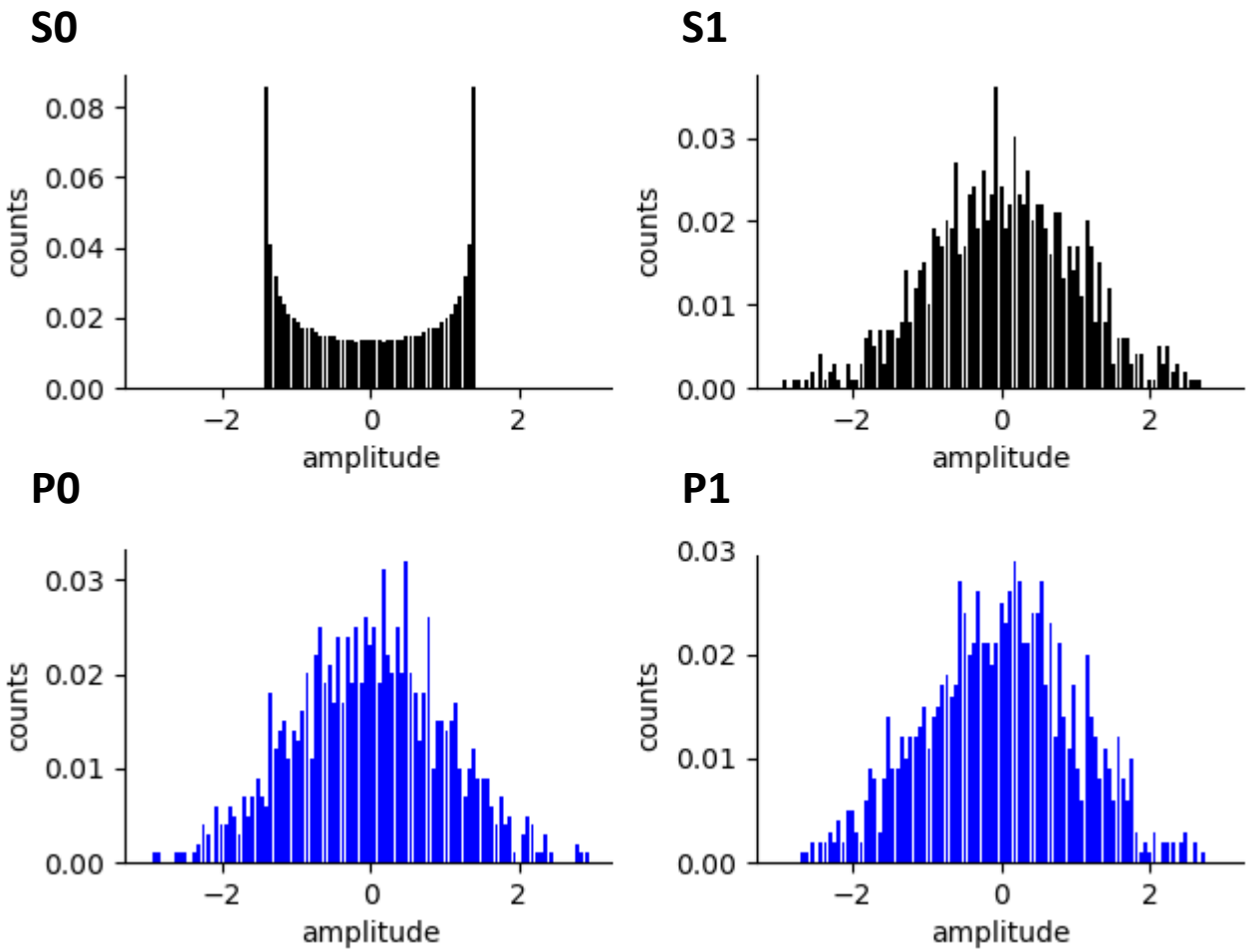


See, “L10_fpica_2_sources_uncorr_indep.py”

Independence and uncorrelatedness (2/3)

Distribution of amplitudes of independent (**S**) and non-independent sources (**P**).

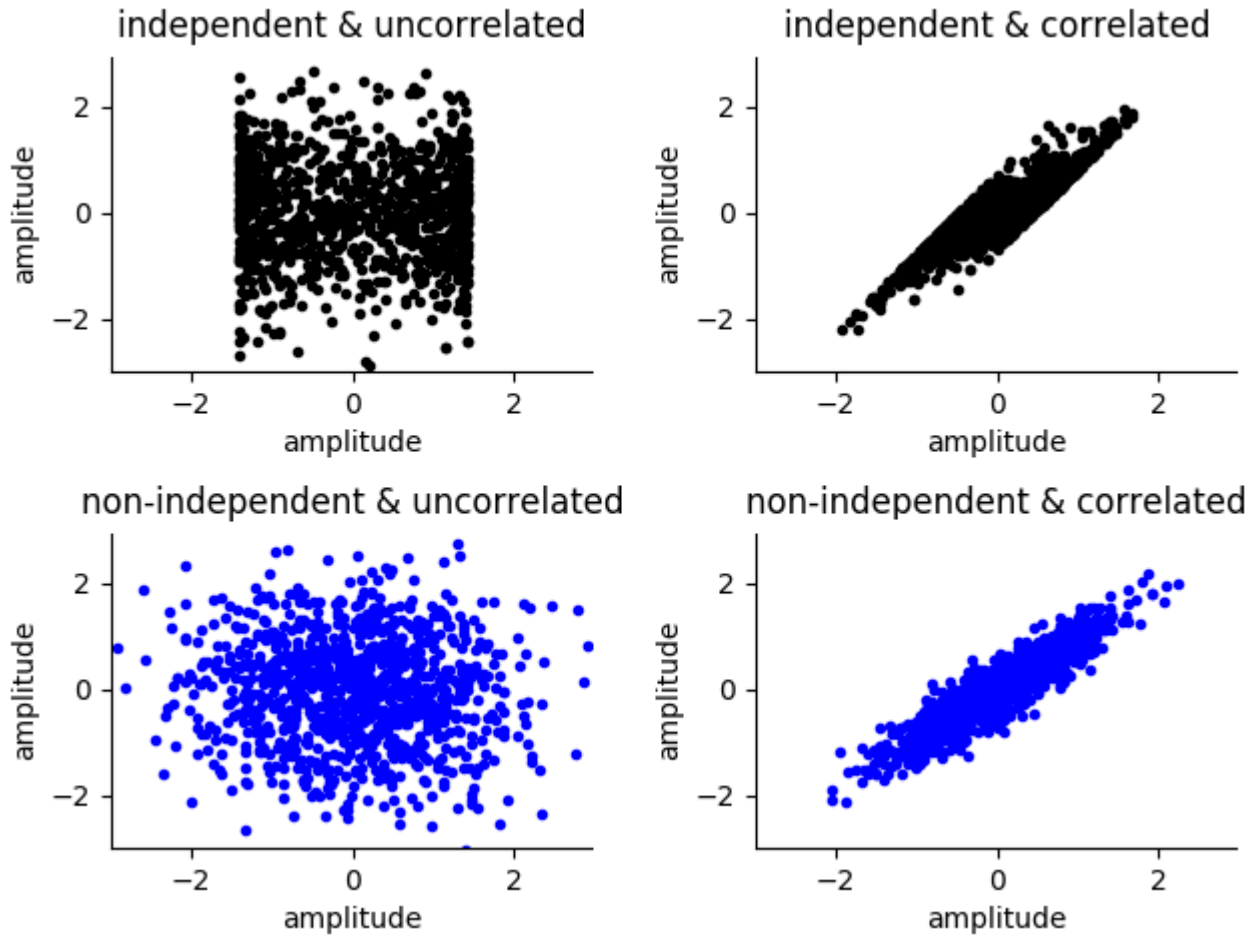
```
# sources
S0 = np.sin(2 * np.pi * 7 * t)
S1 = np.random.randn(N)
P0 = np.random.randn(N)
P1 = np.random.randn(N)

# histogram
b = np.linspace(xmin, xmax, bins)
h, b = np.histogram(x, b)
h = h / np.sum(h)
b = b[:-1] + (b[1] - b[0]) / 2
```



See, “L10_fpica_2_sources_uncorr_indep.py”

Independence and uncorrelatedness (3/3)

Scatter plot clarifies the difference between independent and uncorrelated sources.

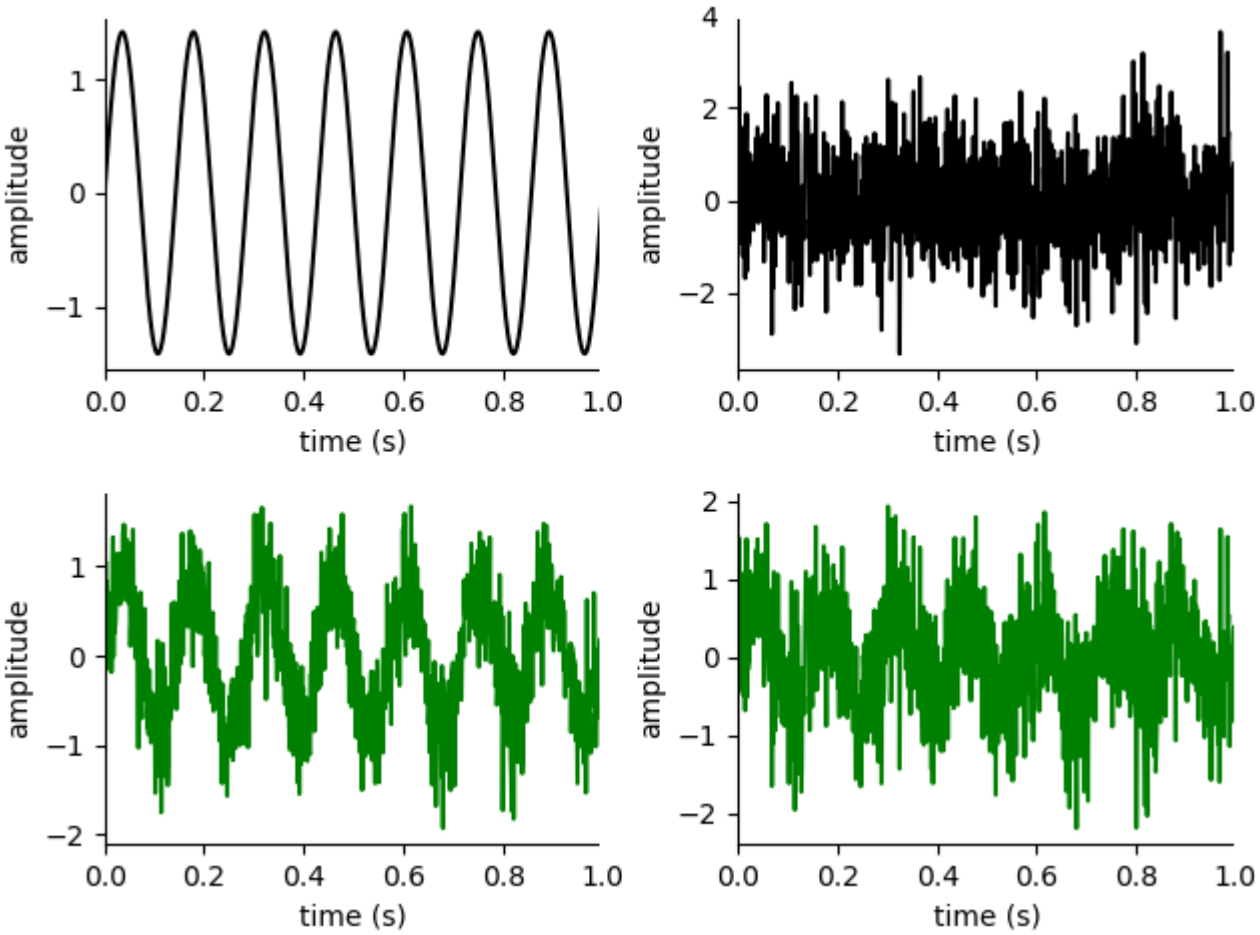


See, “L10_fpica_2_sources_uncorr_indep.py”

ICA example 1 (1/2)

We generate two independent sources and apply linear mixing.

```
# sources
M = 2
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]])


# mix sources
X = np.dot(A, S)
```

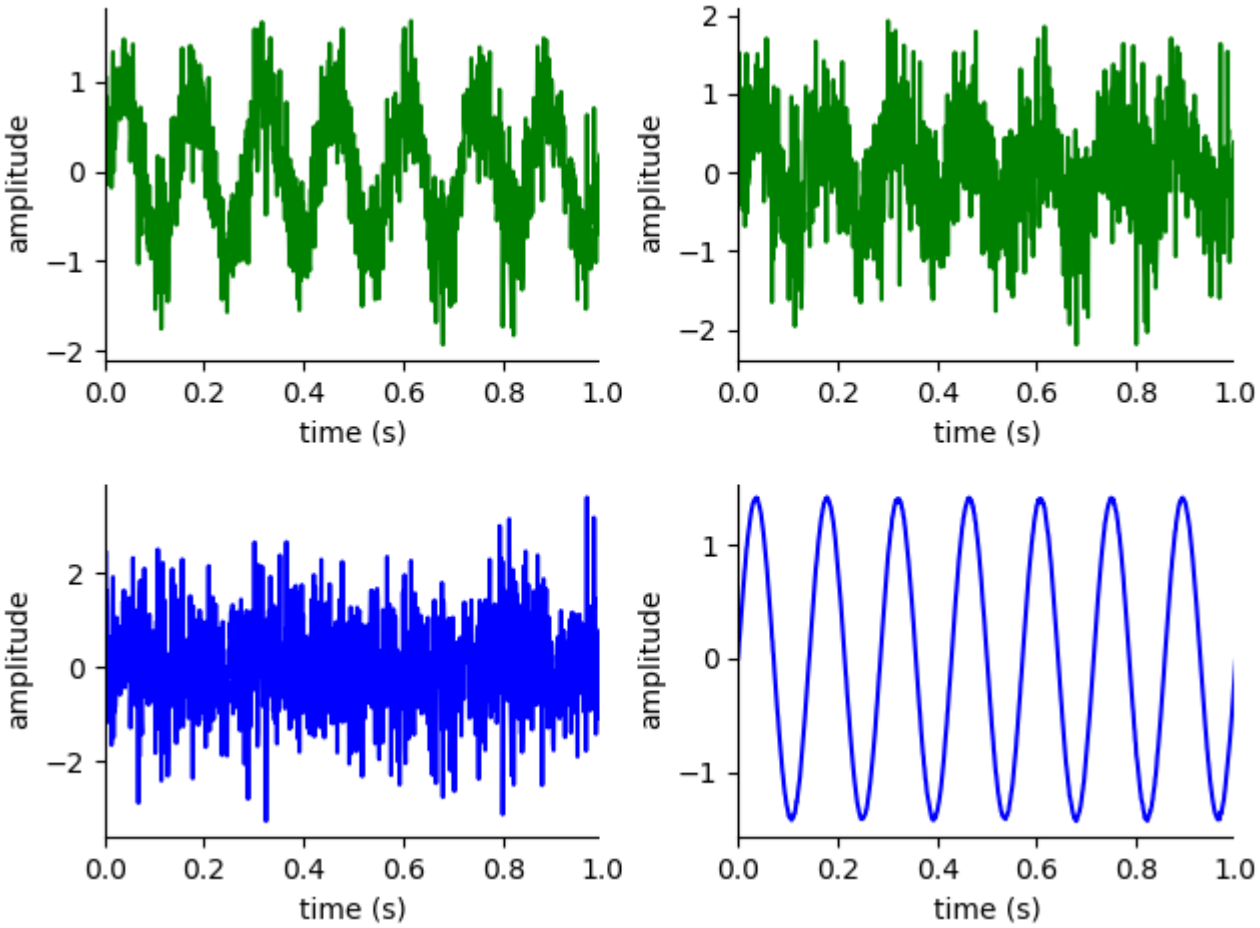


See, “L10_fpica_2_sources.py”

ICA example 1 (2/2)

How do the signals look after ICA?

```
# set parameters and run ICA
max_iter = 1000
tol = 1e-4
W = fpica(X, max_iter, tol) # fast-ica


# unmixing
Z = np.dot(W, X)
```

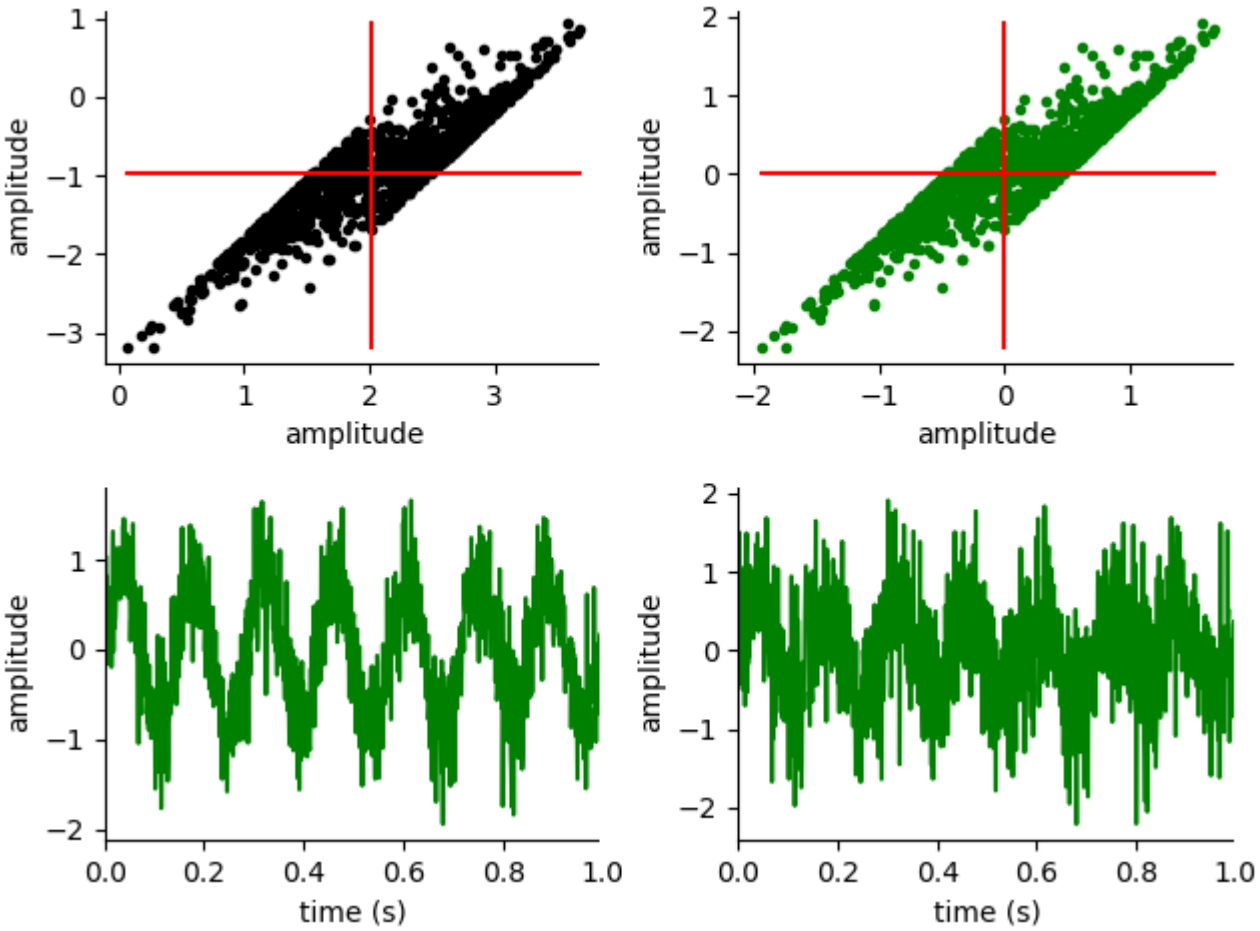


See, “L10_fpica_2_sources.py”

ICA step 1: 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
```

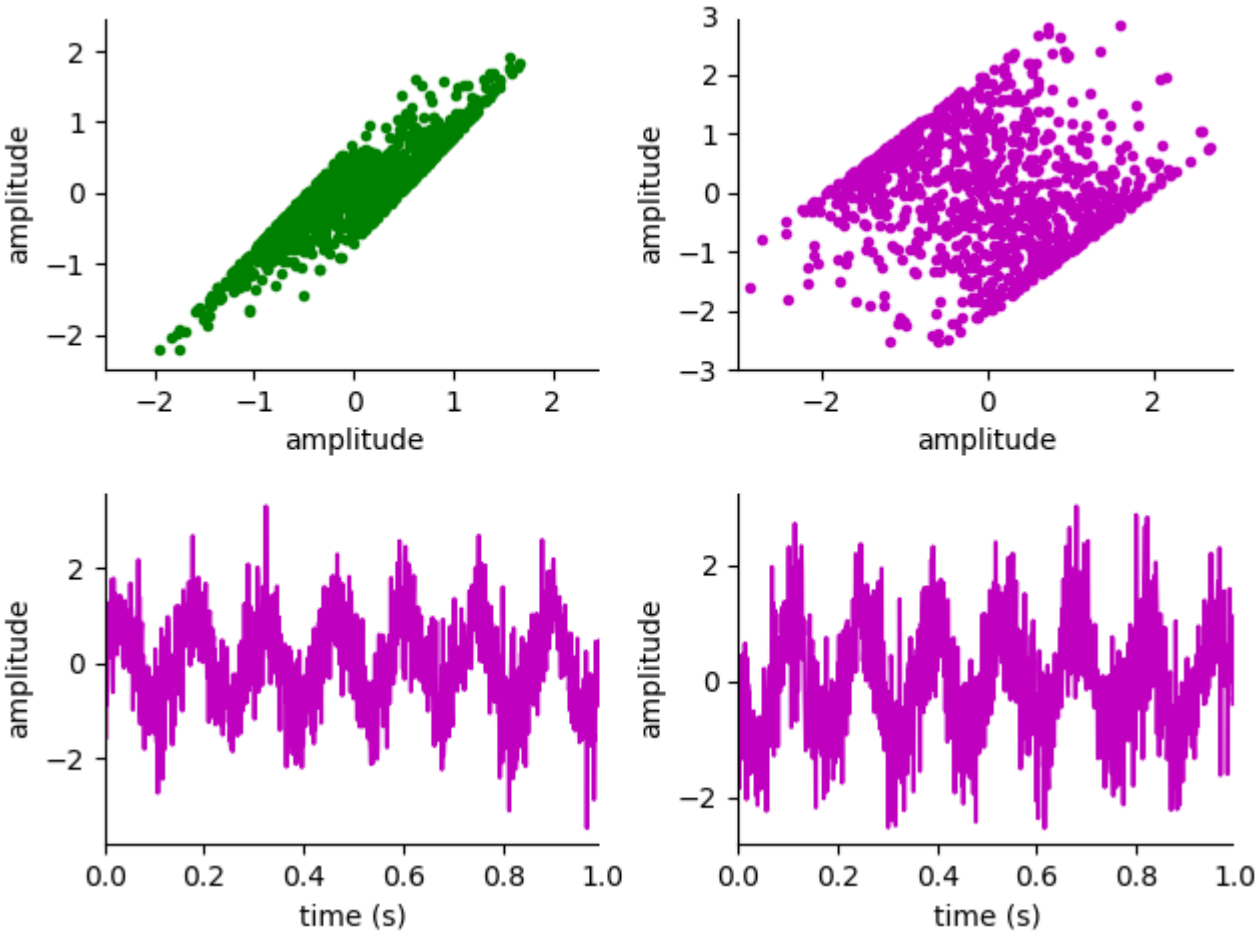


See, “L10_fpica_2_sources_steps.py”

ICA step 2: 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)
```

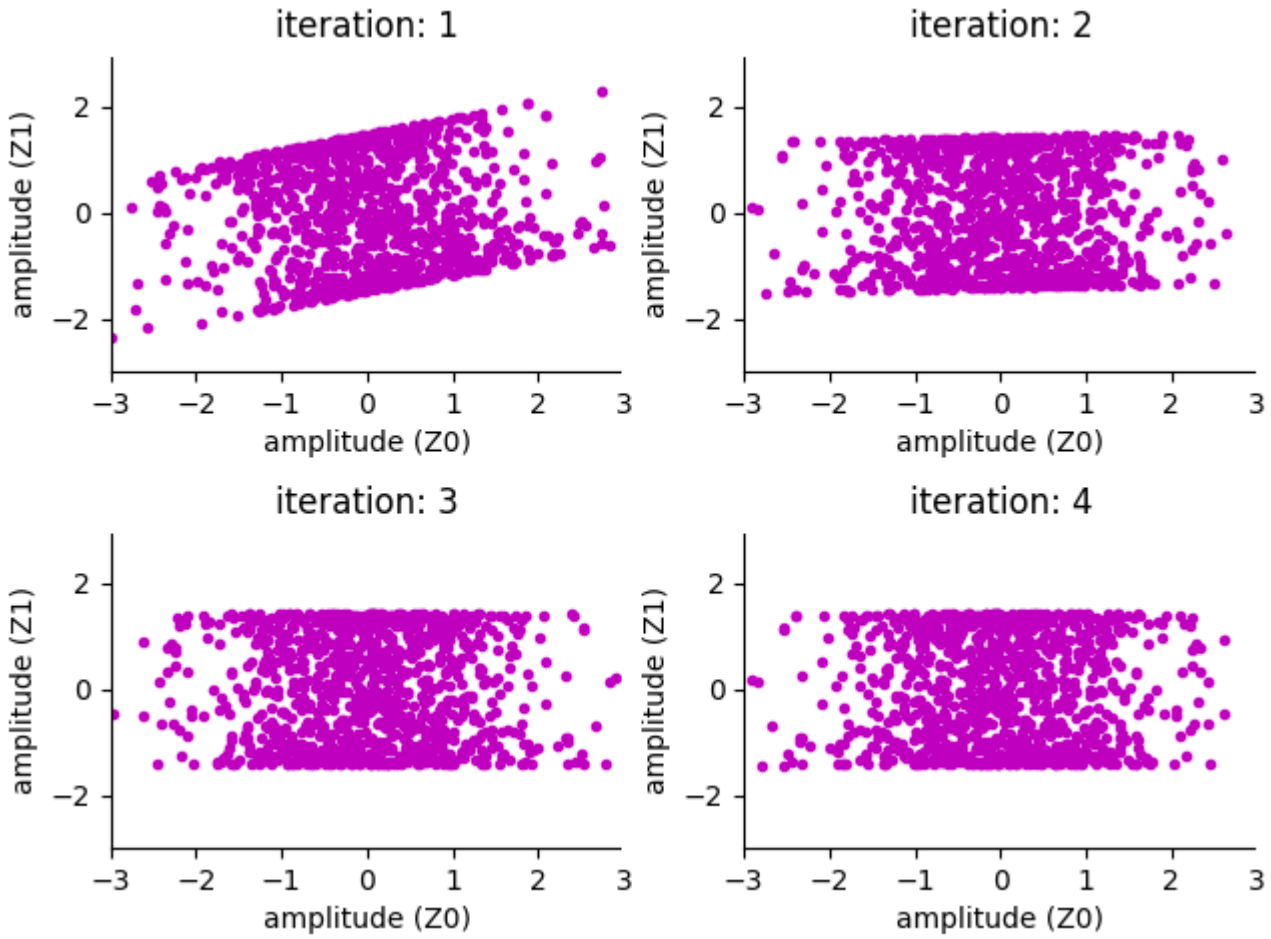


See, "L10_fpica_2_sources_steps.py"

ICA step 3: un-mixing (1/2)

Third, we iteratively maximize independence between 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
```

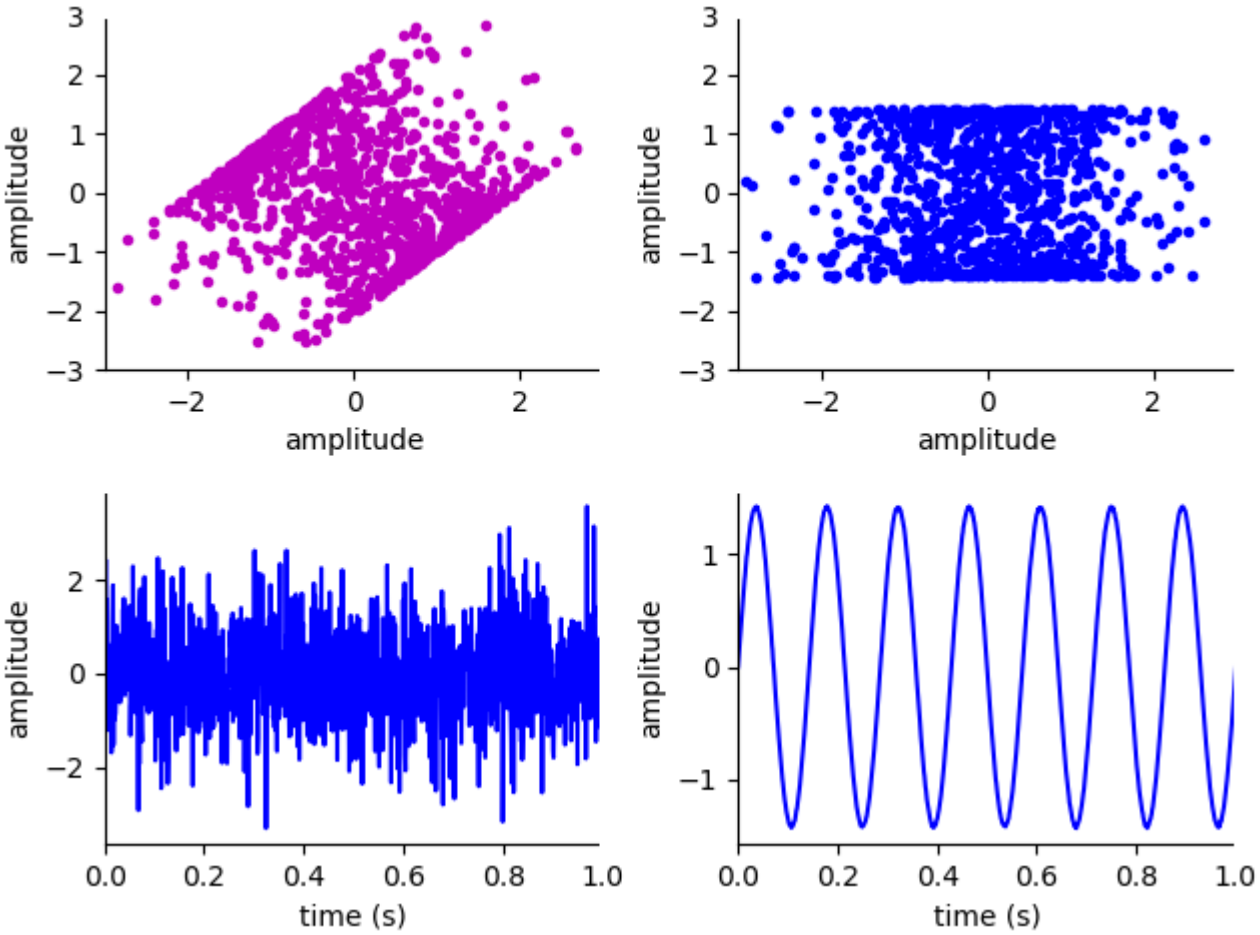


See, “L10_fpica_2_sources_steps.py”

ICA step 3: un-mixing (2/2)

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
```

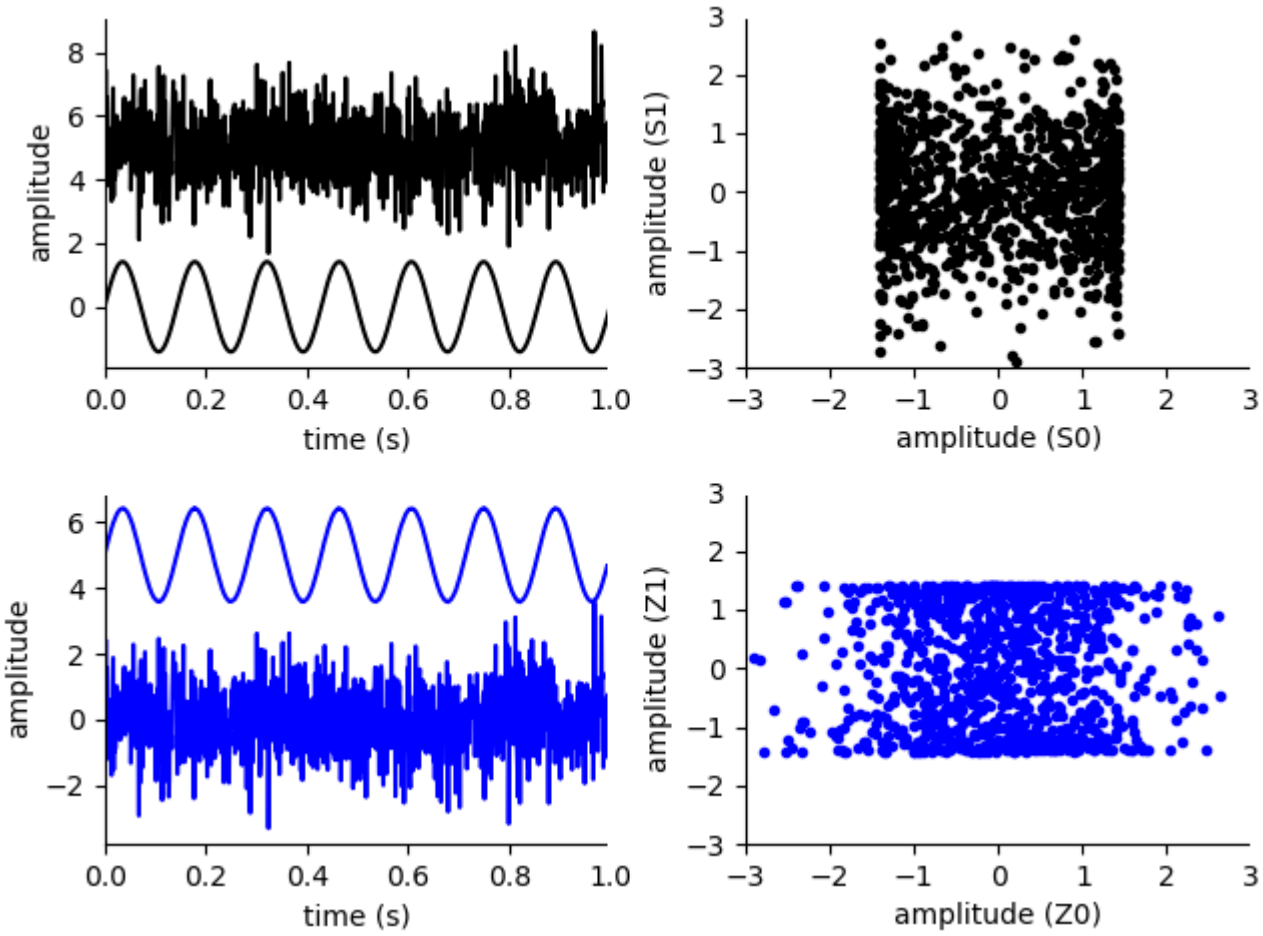


See, “L10_fpica_2_sources_steps.py”

Original and estimated sources

The original and estimated sources may have different order and variance.

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

# 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)

# A' = pinv(W) ==
#      [[0.3963804  0.5848915]
#       [0.5975067  0.3770956]]

# unmixing
Z = np.dot(W, X)
```



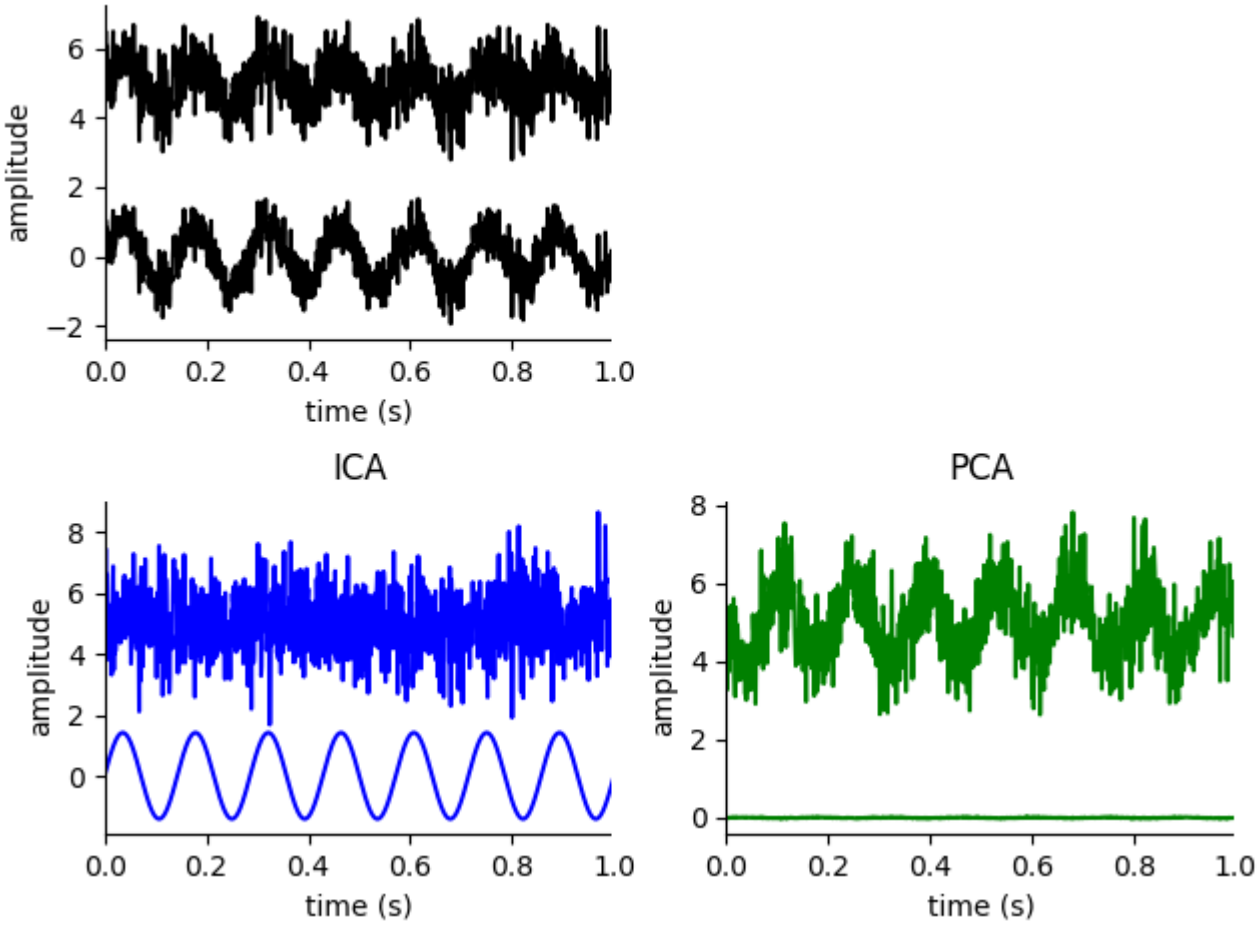
See, "L10_fpica_2_sources_steps.py"

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, "L10_fpica_2_sources_vs_PCA.py"

Un-mixing matrix in MEG/EEG (1/3)

In case of MEG/EEG data, mixing matrix **A** = *pinv*(W) represents **topography** of the sources, and **S** = W*X represents **time course** of the sources.

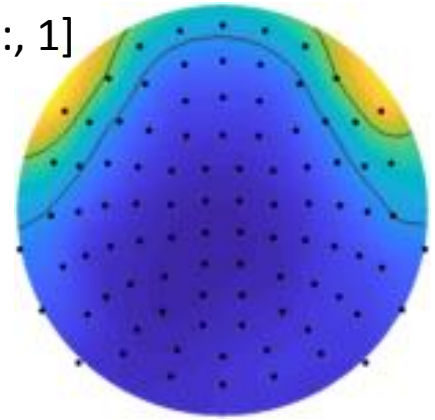
Importantly, ICA does not know about the physical location of sensors, but it manages to identify “true” sources of neuronal activity.

```
# X are measurements [sensors=102 x samples=100000]

# do ICA
W = fpica(X, max_iter=1000, tol=1e-4)

# mixing matrix
A = np.linalg.pinv(W)

# each column of A gives the weights for all sensor
IC_component_0 = A[:, 0] # [sensors x 1]
```

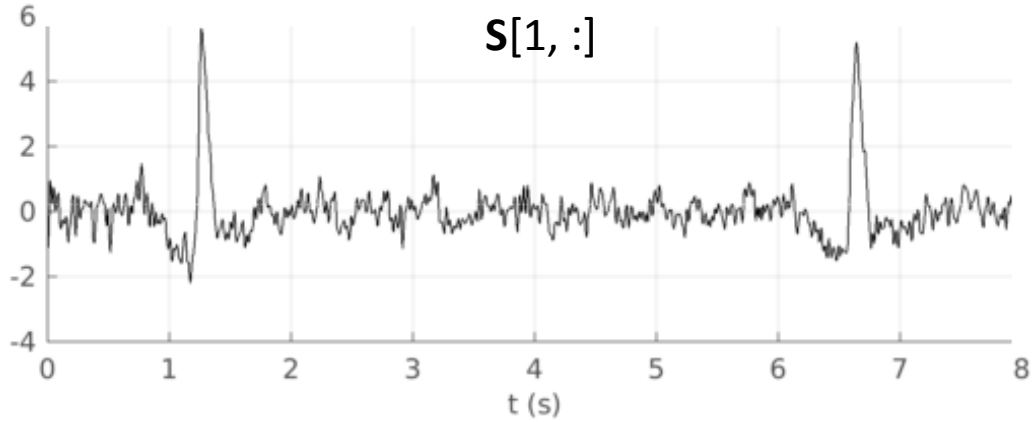
Un-mixing matrix in MEG/EEG (2/3)

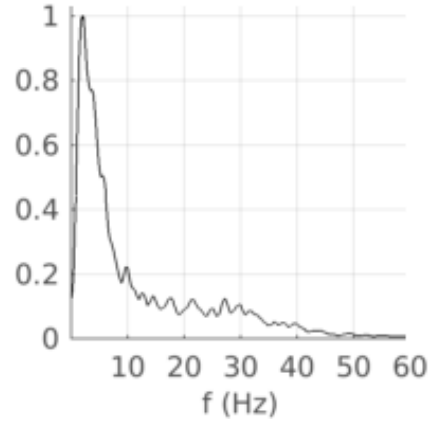
How do the components look in MEG?

A[:, 1]

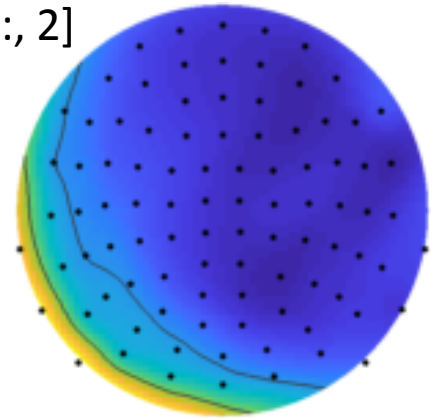


S[1, :]

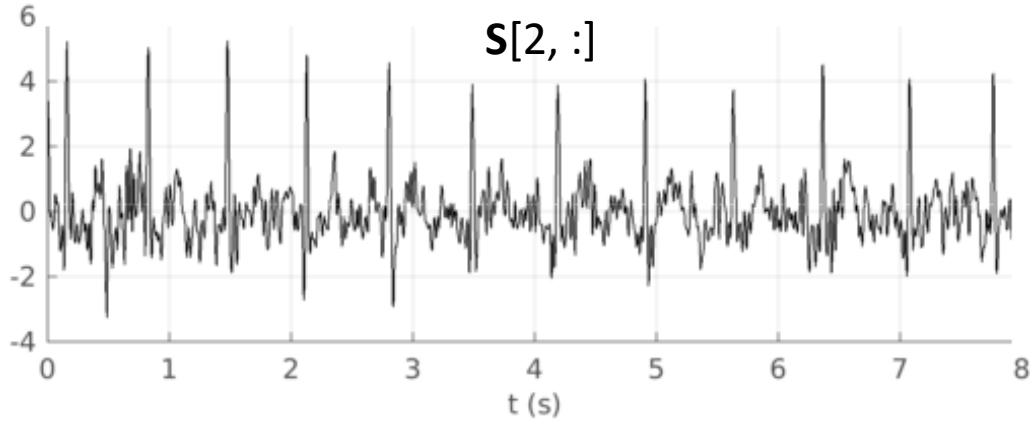


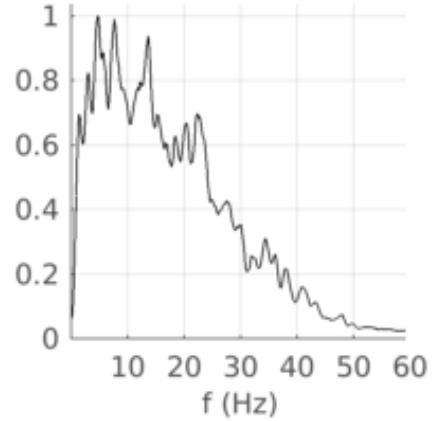


A[:, 2]



S[2, :]

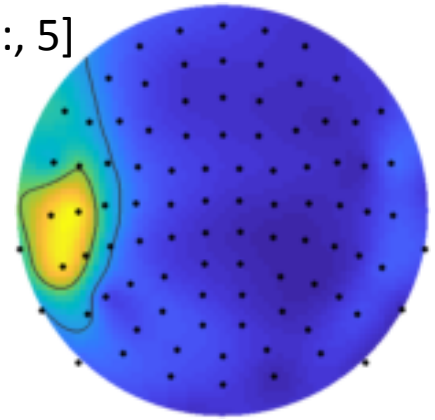




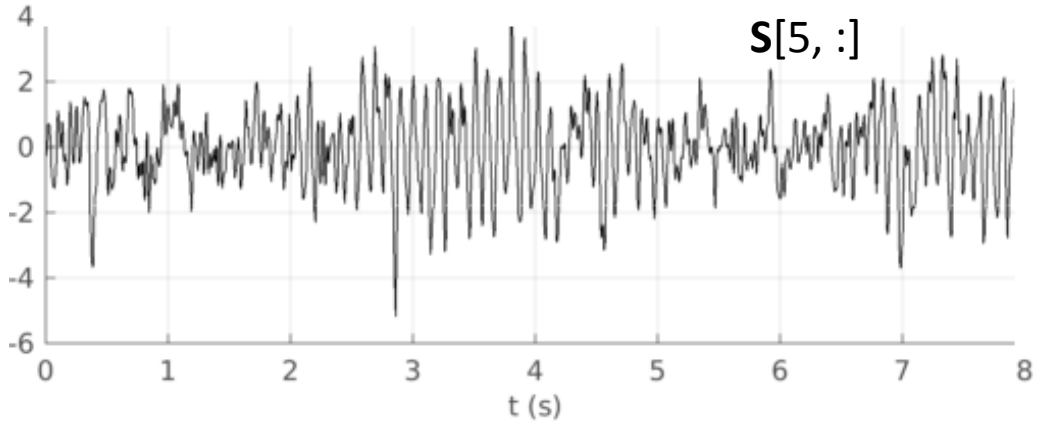
Un-mixing matrix in MEG/EEG (3/3)

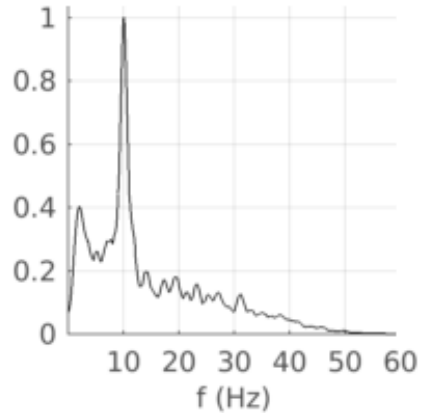
How do the components look in MEG?

A[:, 5]

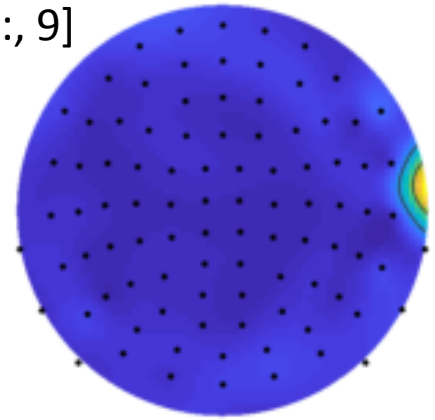


S[5, :]

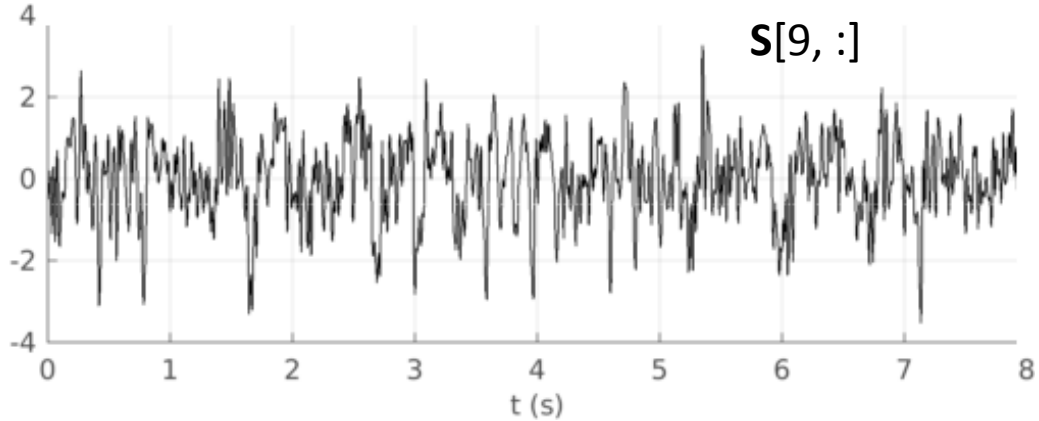


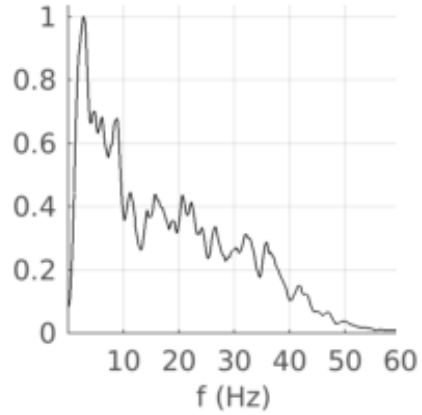


A[:, 9]



S[9, :]





Artefacts removing

How to remove artefacts from the measurements using ICA?

```
# X are measurements [sensors=102 x samples=100000]

# do ICA
W = fpica(X, max_iter=1000, tol=1e-4)

# mixing matrix
A = np.linalg.pinv(W)

# plot and identify the artificial components
S = W * X
plt.plot(S)
indices_of_artefacts = [1, 2, 3]

A[:, indices_of_artefacts] = 0.0

# remove artefacts and reconstruct artefact-free measurements
X = A * S
```

Un-mixing matrix in fMRI (1/3)

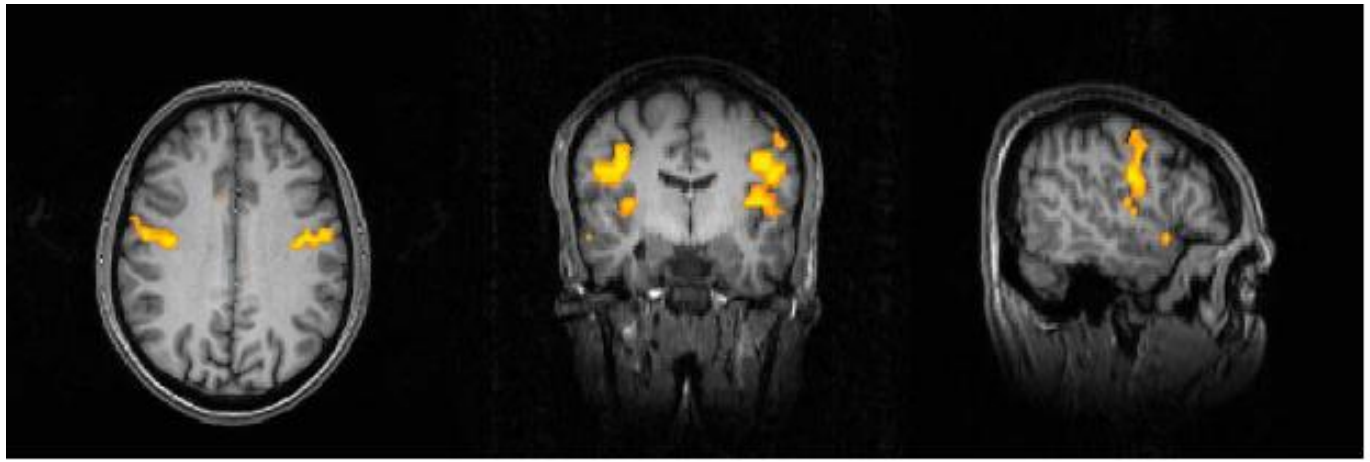
In case of fMRI data, mixing matrix **A** = *pinv*(W) represents **time course** of voxels, **S** = W*X represents **topography** of the sources.

```
# X are measurements [voxels=30000 x samples=200]

# do ICA
W = fpica(X, max_iter=1000, tol=1e-4)

# mixing matrix
A = np.linalg.pinv(W)

# components
IC_component_0 = A[:, 0] # [samples x 1]
```

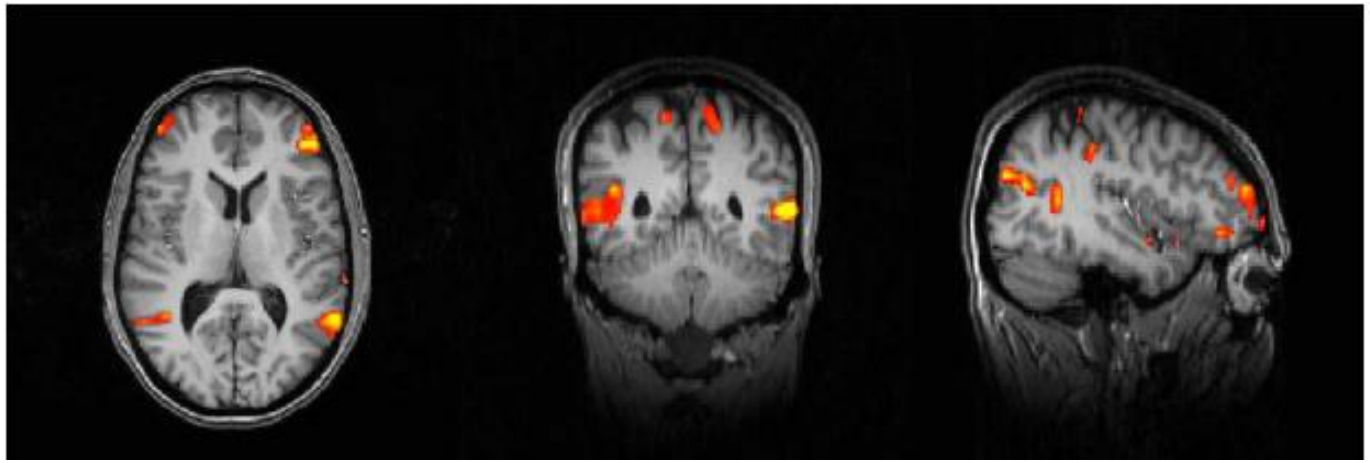
Un-mixing matrix in fMRI (2/3)

How do the components look in fMRI?

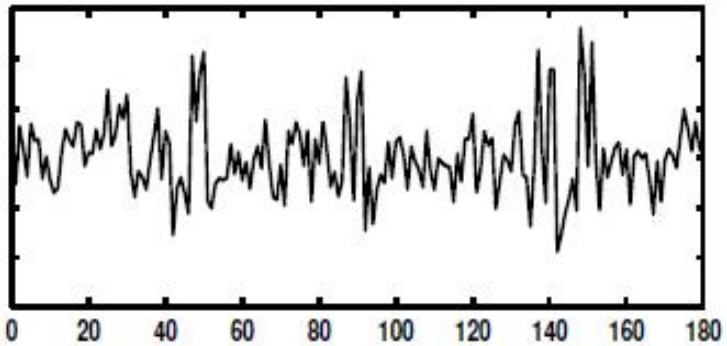
S[2, :]



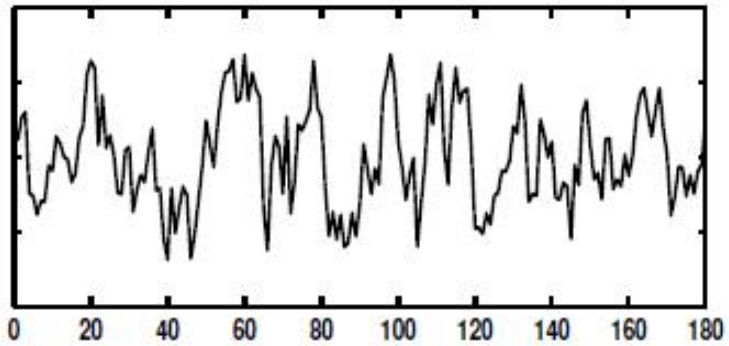
S[4, :]



A[:, 2]



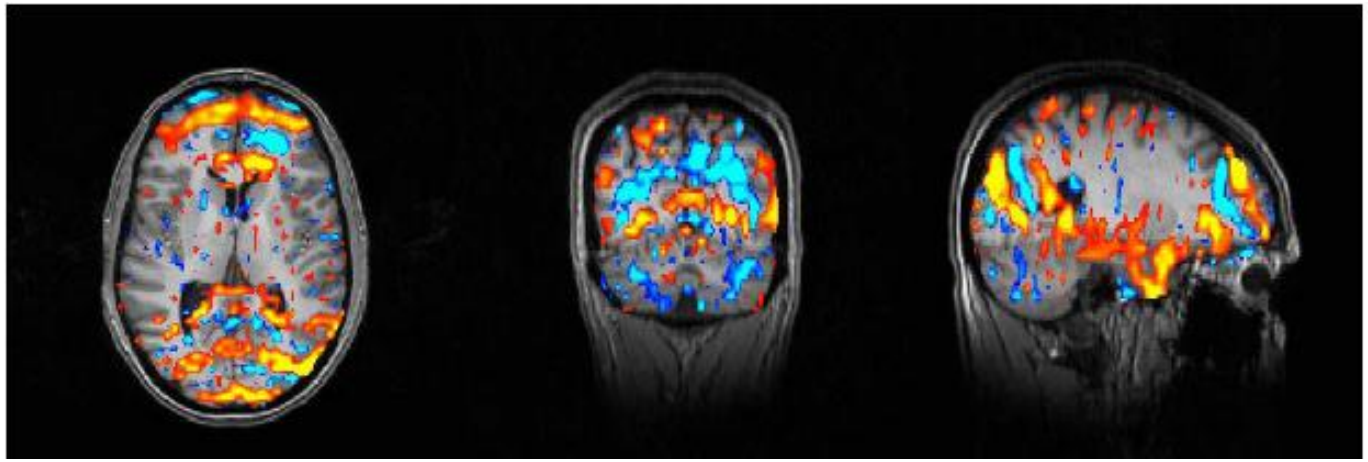
A[:, 4]



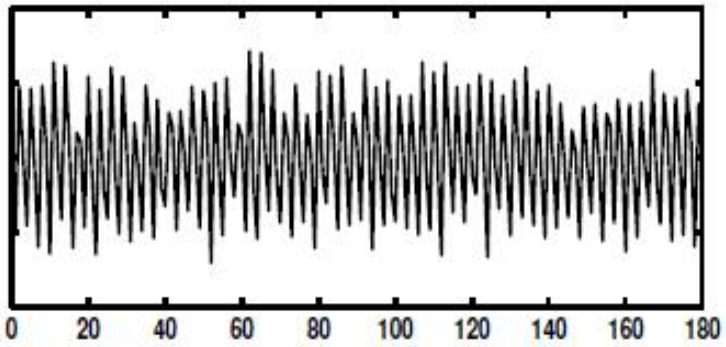
Un-mixing matrix in fMRI (3/3)

How do the components look in fMRI?

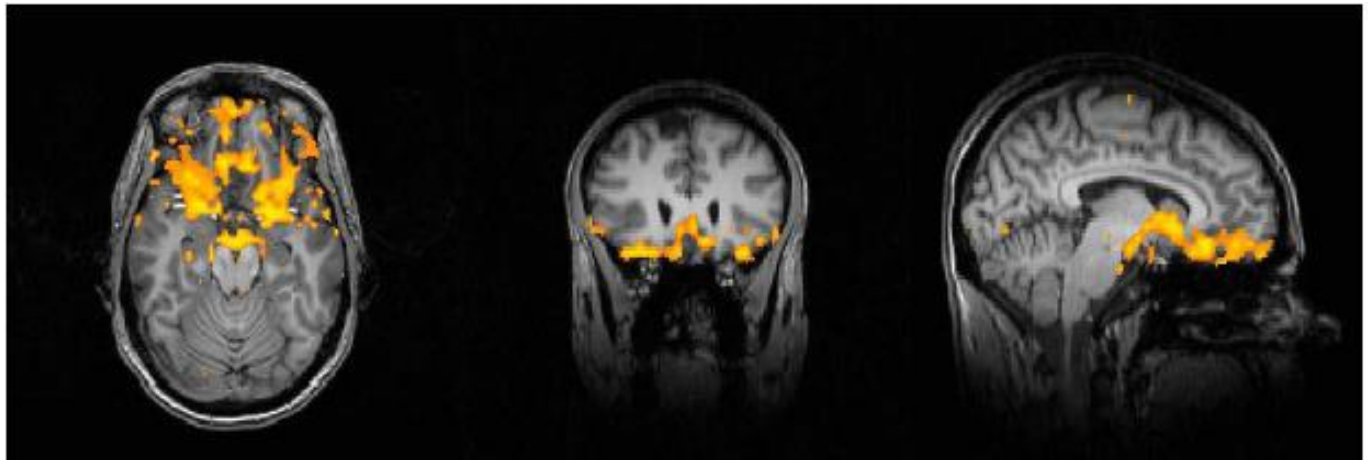
S[1, :]



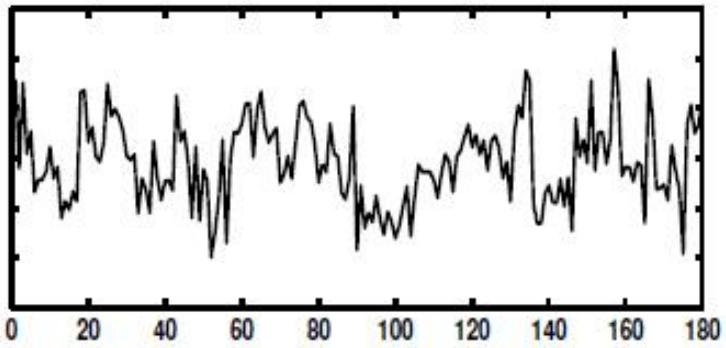
A[:, 1]



S[3, :]



A[:, 3]



scikit-learn implementation

```
from sklearn.decomposition import FastICA

# init model
ica = FastICA()

# fit model / reconstruct sources
S = ica.fit_transform(X)

# unmixing matrix
W = ica.mixing_
```

http://scikit-learn.org/stable/modules/generated/sklearn.decomposition.FastICA.html

Literature

- **Python programming language**

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

- **Data analysis**

- Andreas Müller and Sarah Guido “**Introduction to Machine Learning with Python: A Guide for Data Scientists**”