Python and Statistics for Matlab Users

Session 6

Nicolas Barrier

OT-Med Labex

19 septembre 2016

Introduction

Maths and statistics are manipulated by using the `numpy` and Scipy libraries. The latter contains numerous sub-modules :

- Integration (`scipy.integrate`)
- Interpolation (`scipy.interpolate`)
- Fourier Transforms (`scipy.fftpack`)
- Linear algebra (`scipy.linalg`)
- Statistics (`scipy.stats`)
- ...

Table of Contents

Data for distributions

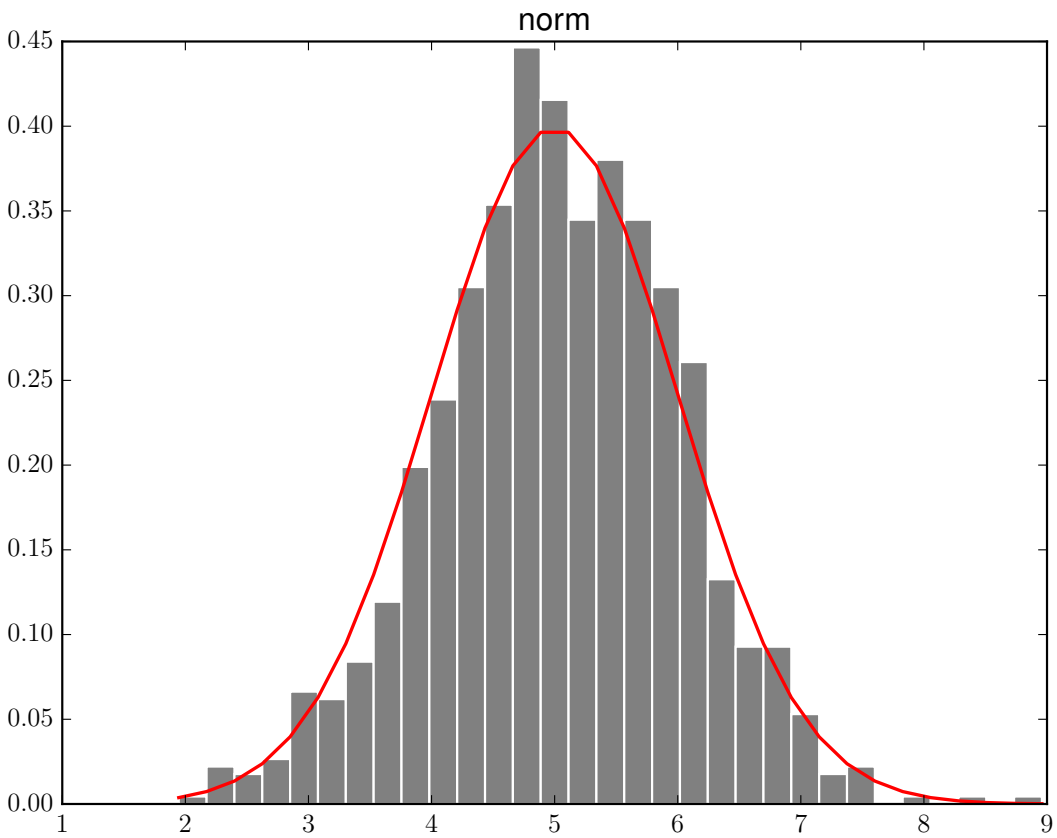
```
import numpy as np
import scipy.stats as stats
import pylab as plt

N = 1000 # number of samples to draw
nbins = int(np.sqrt(N)) # number of bins for histograms
```

Normal distribution

```
# initialises a distribution object
norm = stats.norm(loc=5, scale=1)

# random draft given a distribution
norm_rvs = norm.rvs(size=N)

# draw the histogram
(n, bins, patches) = plt.hist(norm_rvs, nbins, normed=True,
                              color='gray', edgecolor='white')

# draw the distribution pdf
plt.plot(bins, norm.pdf(bins), color='r')
```

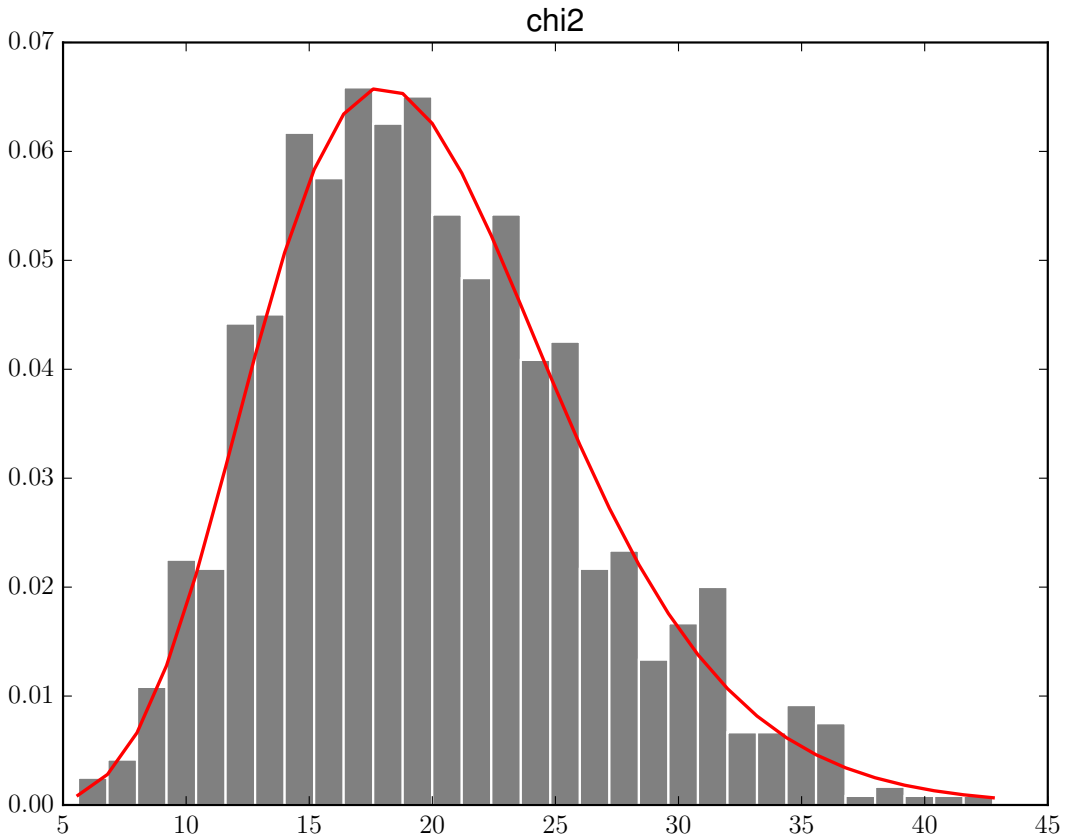
Normal distribution

```
norm = stats.norm(loc=5, scale=1)
```



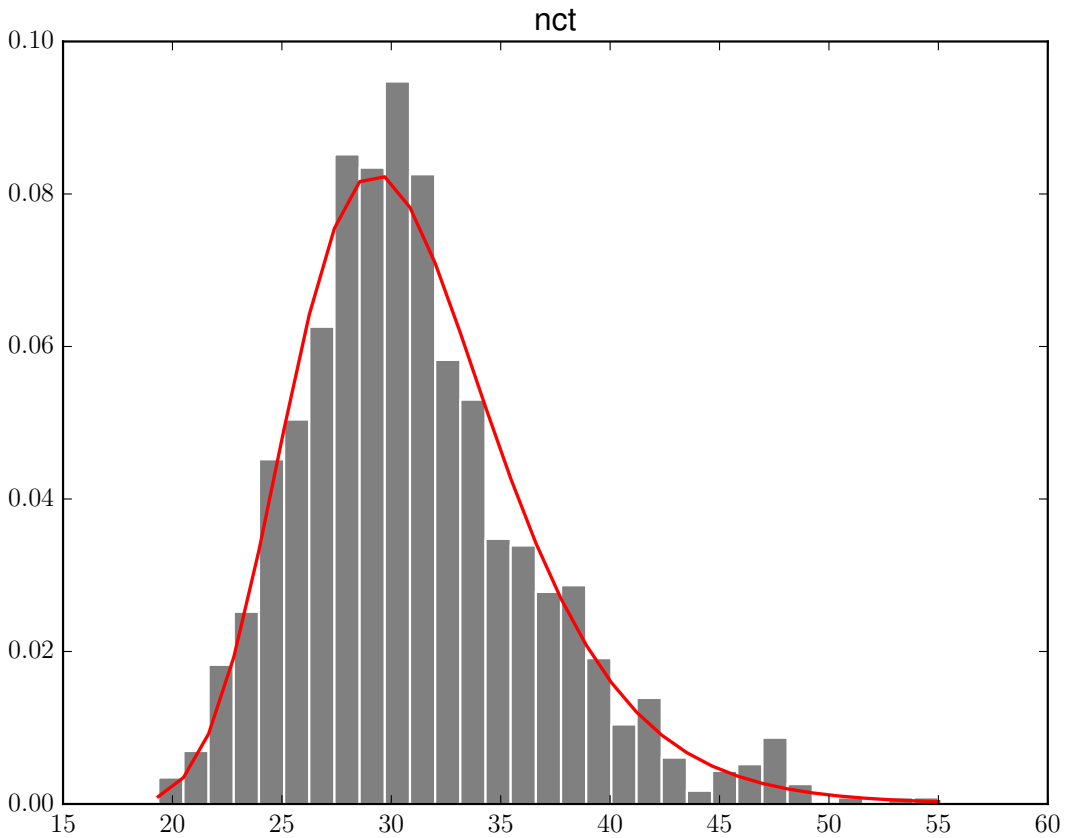
χ_2 distribution

```
chi2 = stats.chi2(df=20)
```



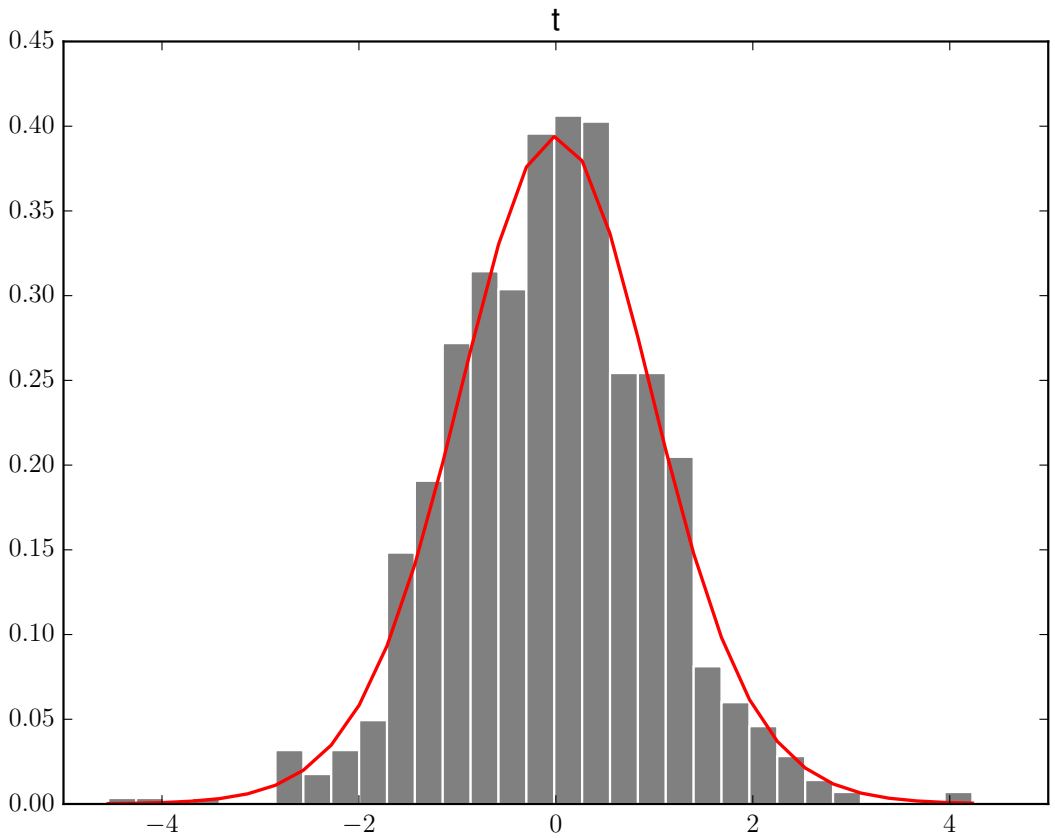
Non central Student T distribution

```
nct = stats.nct(df=20, nc=30)
```



Non central Student T distribution

```
nct = stats.t(df=20)
```



Student T test

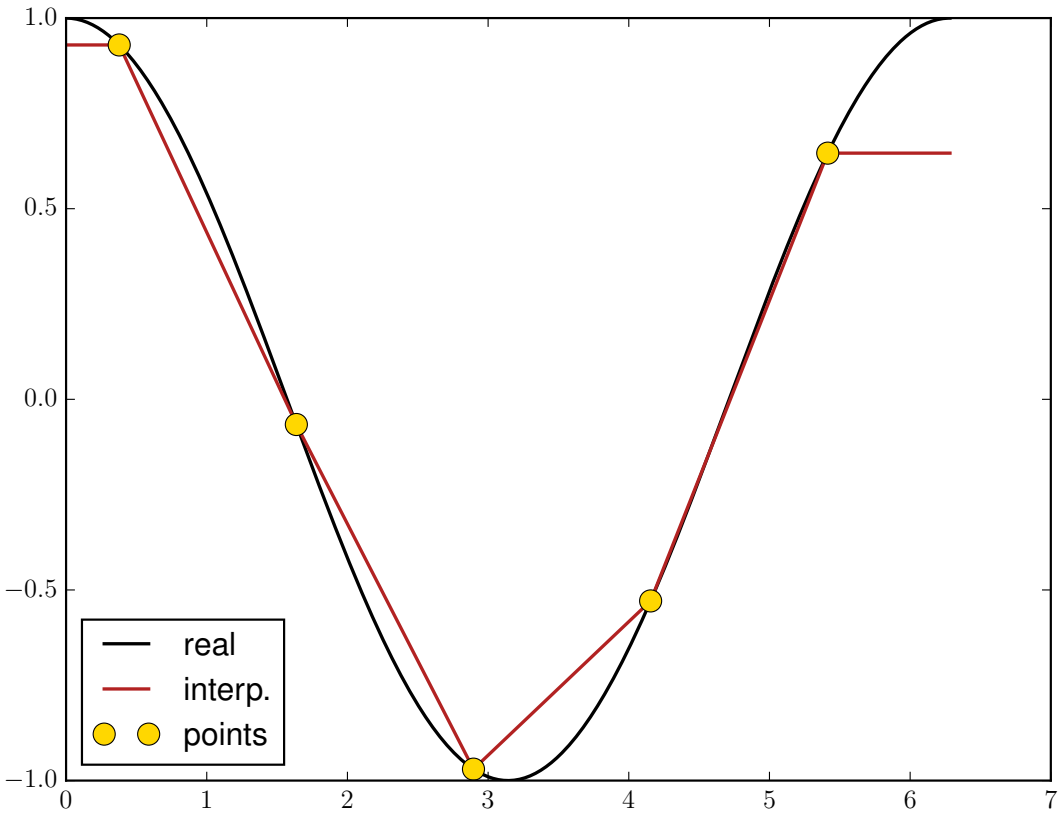
```
# random drafts of time series
me = 5; st = 1; N = 200
rvs1 = stats.norm.rvs(loc=me, scale=st, size=N)
rvs2 = stats.norm.rvs(loc=me, scale=st, size=N)

# for two independent variables with id. variances
tvalue, pvalue = stats.ttest_ind(rvs1, rvs2, equal_var=True)

# for two related variables with id. variances
tvalue, pvalue = stats.ttest_rel(rvs1, rvs2)

# recovering the value of the Student table
proba = 0.95; dof = 30
ttable = stats.t.ppf(proba, dof)
```

Table of Contents

1D interpolation

```
# x coords of interpolated values
x = np.linspace(0, 2*np.pi, 500)
y = np.cos(x) # real values

# data sampling
xp = x[30::100]
yp = y[30::100]

# interpolated values
yi = np.interp(x, xp, yp, left=yp[0], right=yp[-1])
# possibility to define the left and right values
# default: constant and set to yp[0] and yp[-1]
```

1D interpolation



2D interpolation

```
import scipy # library for griddata

# creates a dummy data set
xp = np.random.random_sample(300)*10-5
yp = np.random.random_sample(300)*10-5
zp = np.sin(xp*2+yp*2) # (300,)
points = np.array([xp, yp]).T # (300, 2)

# generate grid points
xi = np.arange(-4.01, 4.01, 0.25) # (33,)
yi = np.arange(-3.01, 3.01, 0.25) # (25)
xi, yi = np.meshgrid(xi, yi) # (25, 33)
pointsi = np.array([xi, yi]).T # (33, 25, 2)

# compute interpolation
zi = scipy.interpolate.griddata(points, zp, pointsi,
                                method='linear')
```

2D interpolation

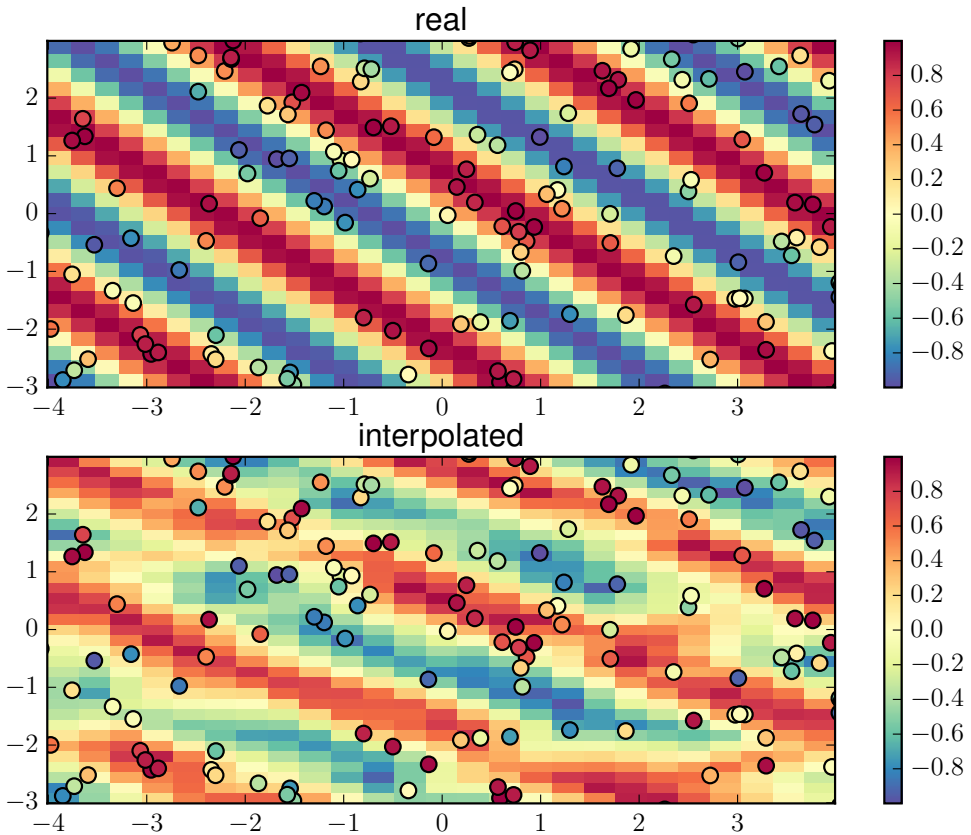
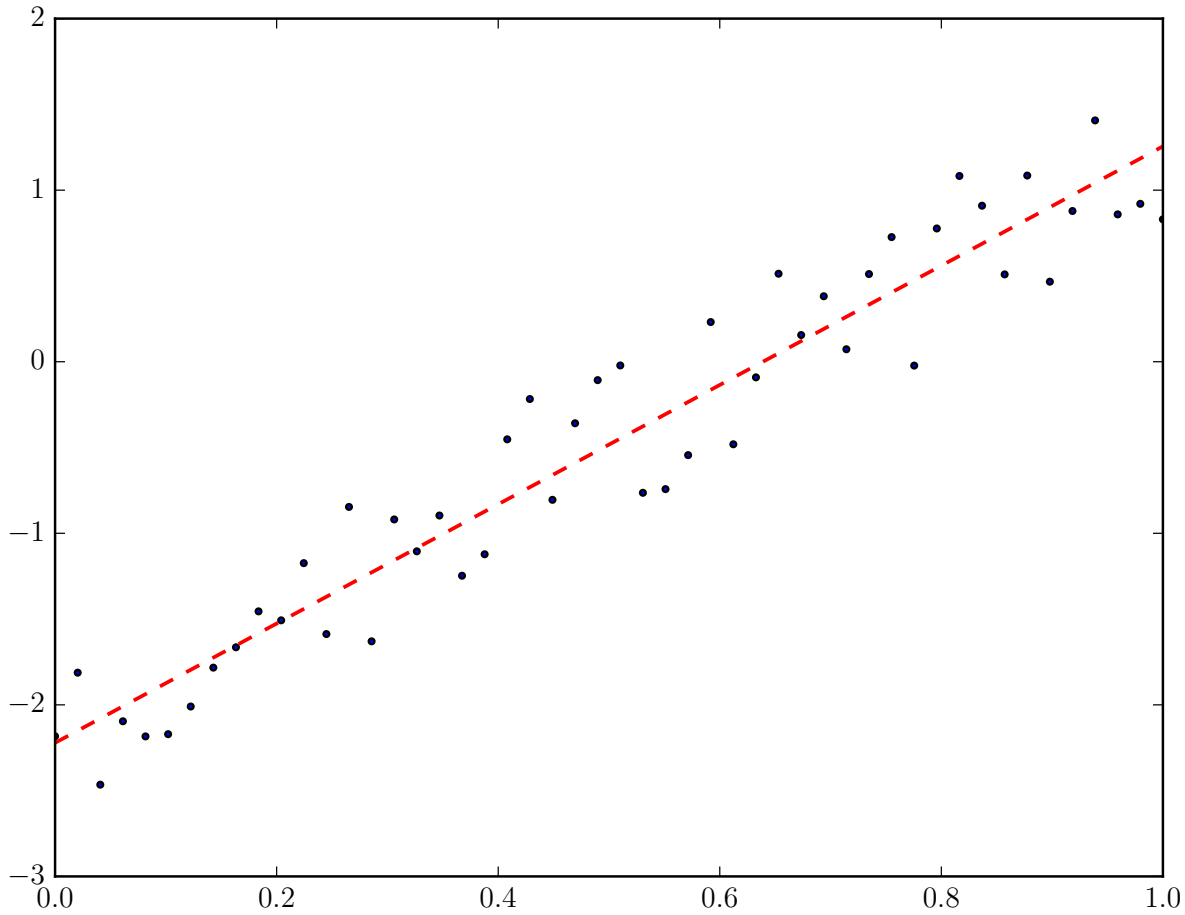


Table of Contents

Linear Regression

```
import numpy as np
import scipy.stats as stats

# creation of a dataset
N = 50; a = 3.3; b = -2.1
noise = np.random.rand(N) - 0.5
x = np.linspace(0, 1, N)
y = a*x + b + noise

slope, intercept, r_value, p_value, std_err = \
    stats.linregress(x, y)
# slope = 3.47, inter = -2.22
yr = slope*x + intercept # for plotting the slope
```

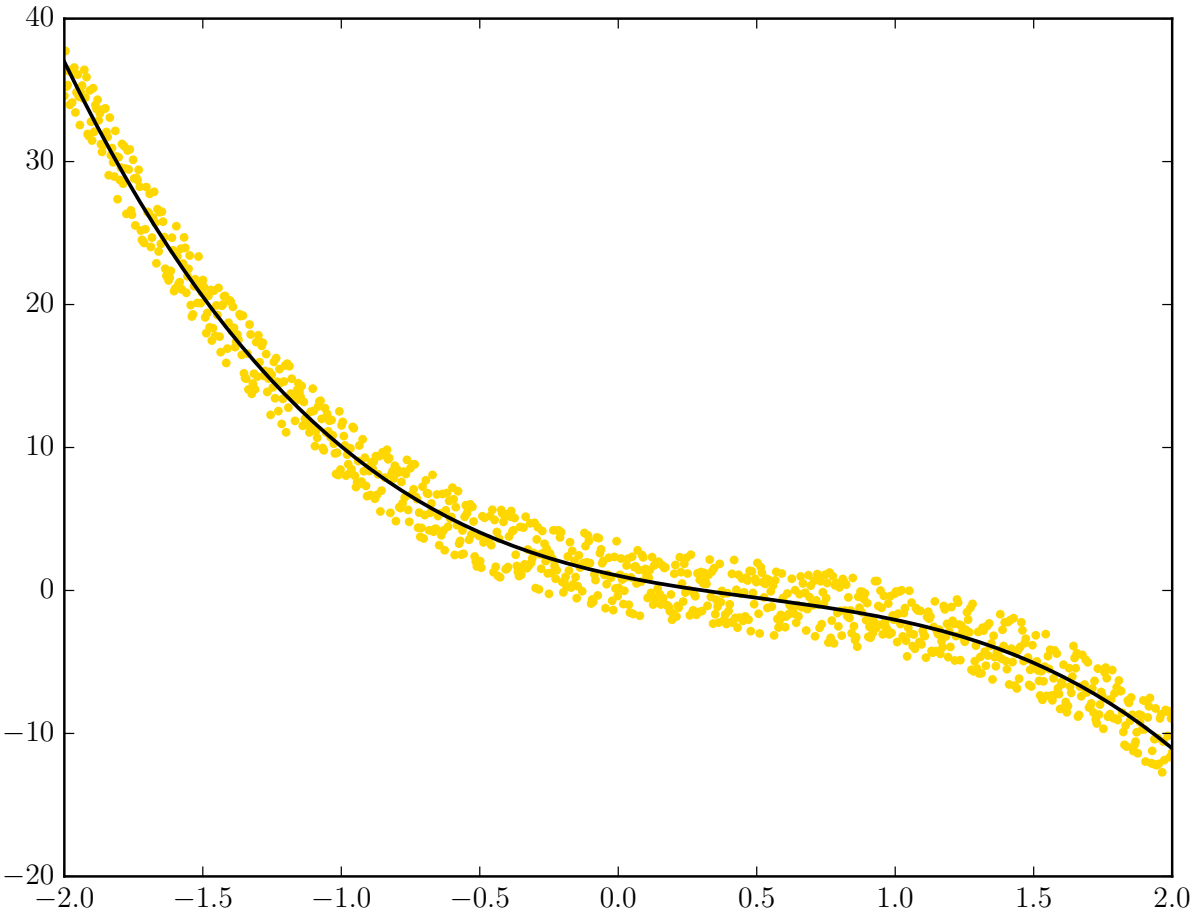
Regression



Polynom fitting

```
# creation of a 3rd order polynom
a = -2; b = 3; c = -4; d = 1
y = a*x**3 + b*x**2 + c*x + d
noise = np.random.rand(len(x)) - 0.5
noise *= 5
y += noise

# calculation of the polynom coefficients
pol = np.polyfit(x, y, 3)
# [-1.98134351  2.98419841 -4.08023535  1.02895507]

# calculates the polynome values
yr = np.polyval(pol, x)
```

Polynom fitting



Data for correlation

```
from netCDF4 import Dataset
import numpy as np

# load the dataset
fin = Dataset("../../data/combined_gridded_index_data.nc", "r")
amo = fin.variables['amo'][:]
nao = fin.variables['nao'][:]
psl = fin.variables["msl"][:]
lon = fin.variables["lon"][:]
lat = fin.variables["lat"][:]
fin.close()

# transfors the NAO/AMO index into
# an array with dims. (nt, 2)
index = np.vstack([nao, amo]).T
```

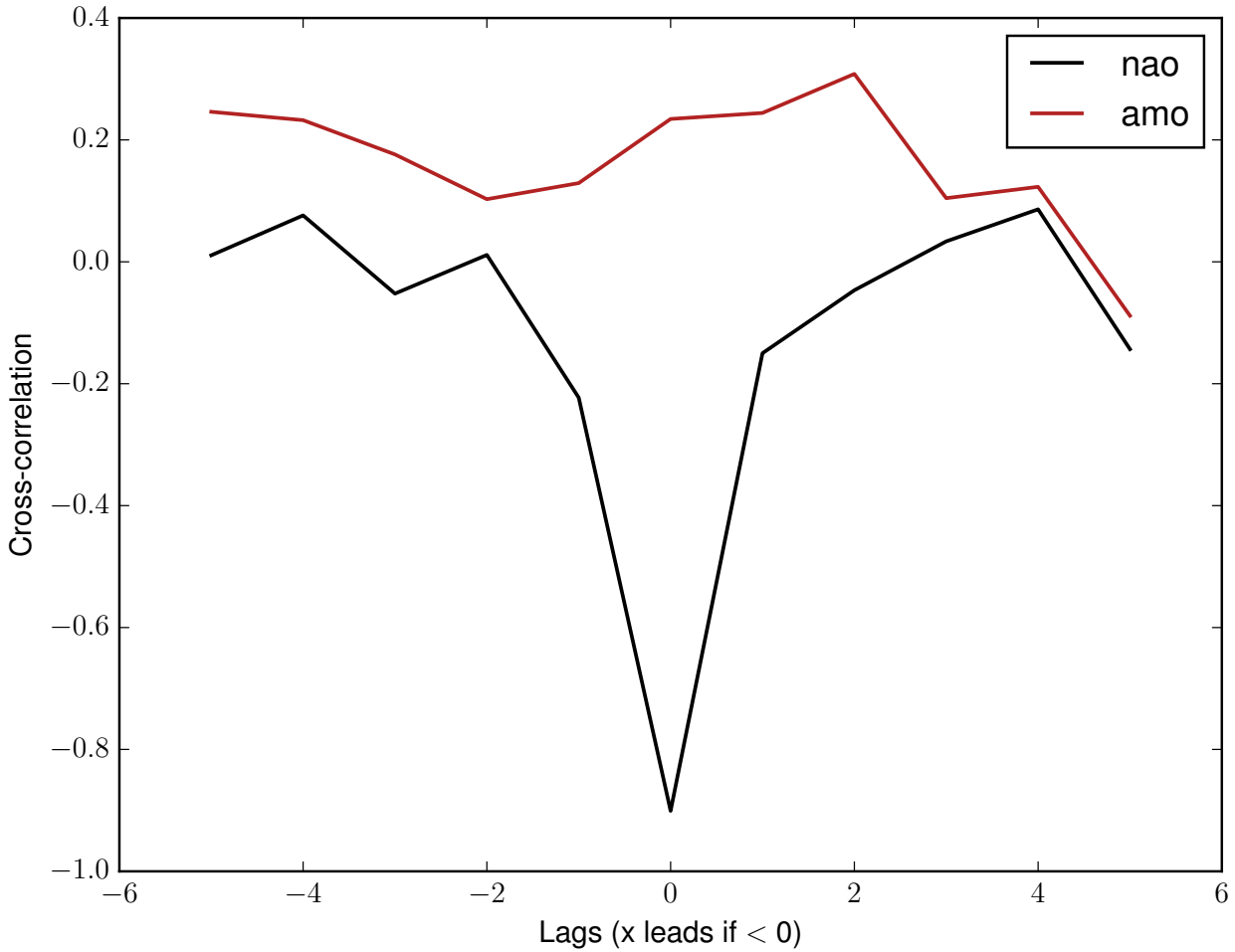
1D, 0-lag correlation/covariance

```
# extracts one slp grid point
ytest = psl[:, 13, 63]

# computes correlation coefficients
np.corrcoef(nao, ytest)[0, 1]
np.corrcoef(amo, ytest)[0, 1]

# computes covariance coefficients
np.cov(nao, ytest)[0, 1]
np.cov(amo, ytest)[0, 1]
```

1D, cross-correlation

```
import pylab as plt

# common arguments of the xcorr function
args = {'usevlines':False, 'marker':None, 'linestyle':'-'}

# nao/amo leads for NEGATIVE lags
lags, cnao, line, b = plt.xcorr(nao, ytest,
                                maxlags=5, label='nao', **args)
lags, camo, line, b = plt.xcorr(amo, ytest,
                                maxlags=5, label='amo', **args)

plt.legend(loc=0)
plt.ylabel('Cross-correlation')
plt.xlabel('Lags (x leads if $<$ 0)')
```

1D, cross-correlation



ND, cross-correlation

```
import nbtools.ts

# computes cross-correlation for each index and each grid point
lag, corr = nbtools.ts.xcorr_ND(index, psl,
                                maxlag=5, use_covariance=False)

# computes cross-covariance for each index and each grid point
lag, cov = nbtools.ts.xcorr_ND(index, psl,
                               maxlag=5, use_covariance=True)


# nao/amo leads for POSITIVE lags
# cov.shape = corr.shape = (2, nlat, nlon, nlag)

# contour at lag 0 for index 0 (NAO)
cs = bmap.contourf(lon, lat, corr[0, :, :, 6])
```

ND, cross-correlation

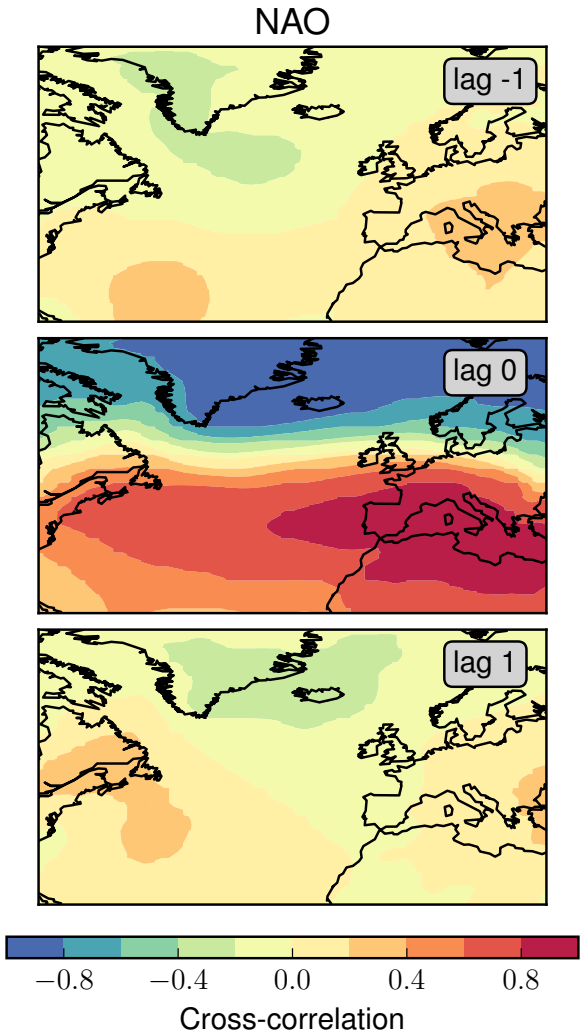


Table of Contents

Introduction

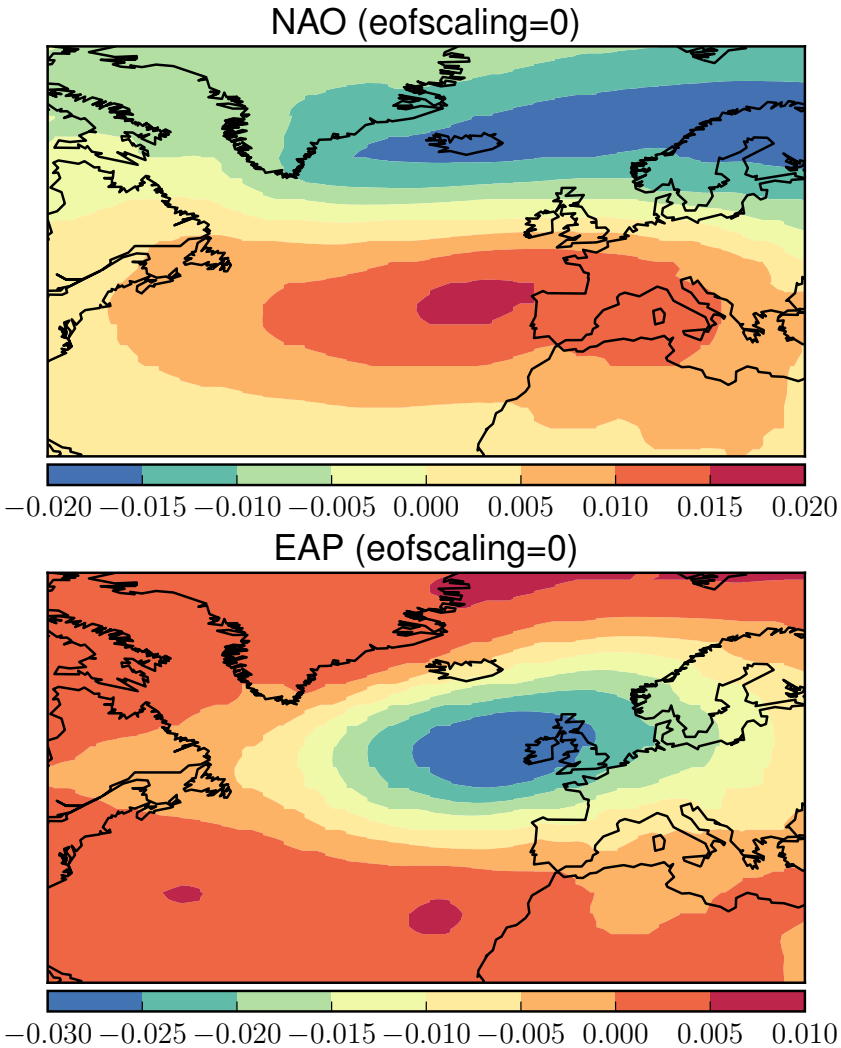
EOF analysis is performed by using the eofs library.

```
import eofs.standard as eofs
import numpy as np
from netCDF4 import Dataset

fin = Dataset("../../data/combined_gridded_index_data.nc", "r")
psl = fin.variables["msl"][:]
lon = fin.variables["lon"][:]
lat = fin.variables["lat"][:]
year = fin.variables["year"][:]
fin.close()

# weight array. For maps, sqrt(cos(lat))
weights = np.sqrt(np.cos(np.deg2rad(lat))) # (nlat)
weights = np.expand_dims(weights, axis=0) # (1, nlat)
weights = np.expand_dims(weights, axis=-1) # (1, nlat, 1)
bd = np.broadcast_arrays(psl, weights) # conform array size
weights = bd[1] # (nt, nlat, nlon)
```

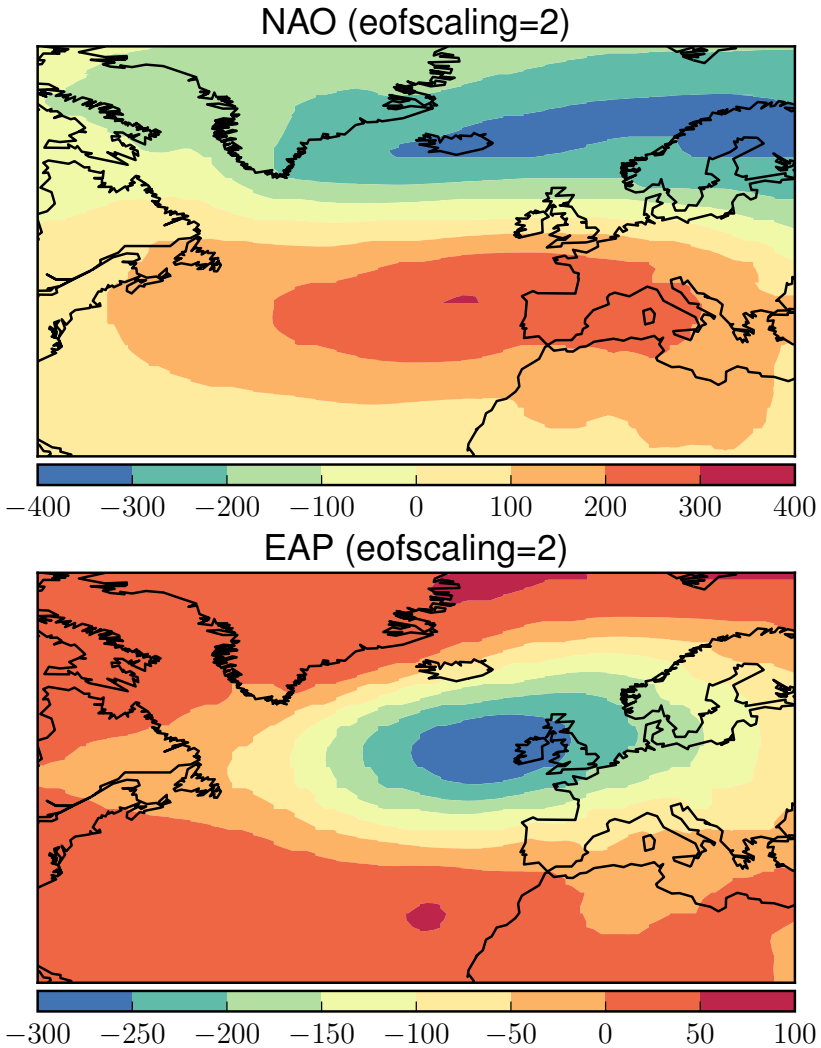
Extraction of EOF maps

```
# intialisation of solver
eofobj = eofs.Eof(psl, weights=weights)

# Un-scaled EOFs (default).
eofmap0 = eofobj.eofs(eofscaling=0, neofs=2)

# EOFs are divided by the square-root of their eigenvalues.
eofmap1 = eofobj.eofs(eofscaling=1, neofs=2)

# EOFs are multiplied by the square-root of
# their eigenvalues (units are in Pa)
eofmap2 = eofobj.eofs(eofscaling=2, neofs=2)

# array size: (2, nlat, nlon)
```

Extraction of EOF maps



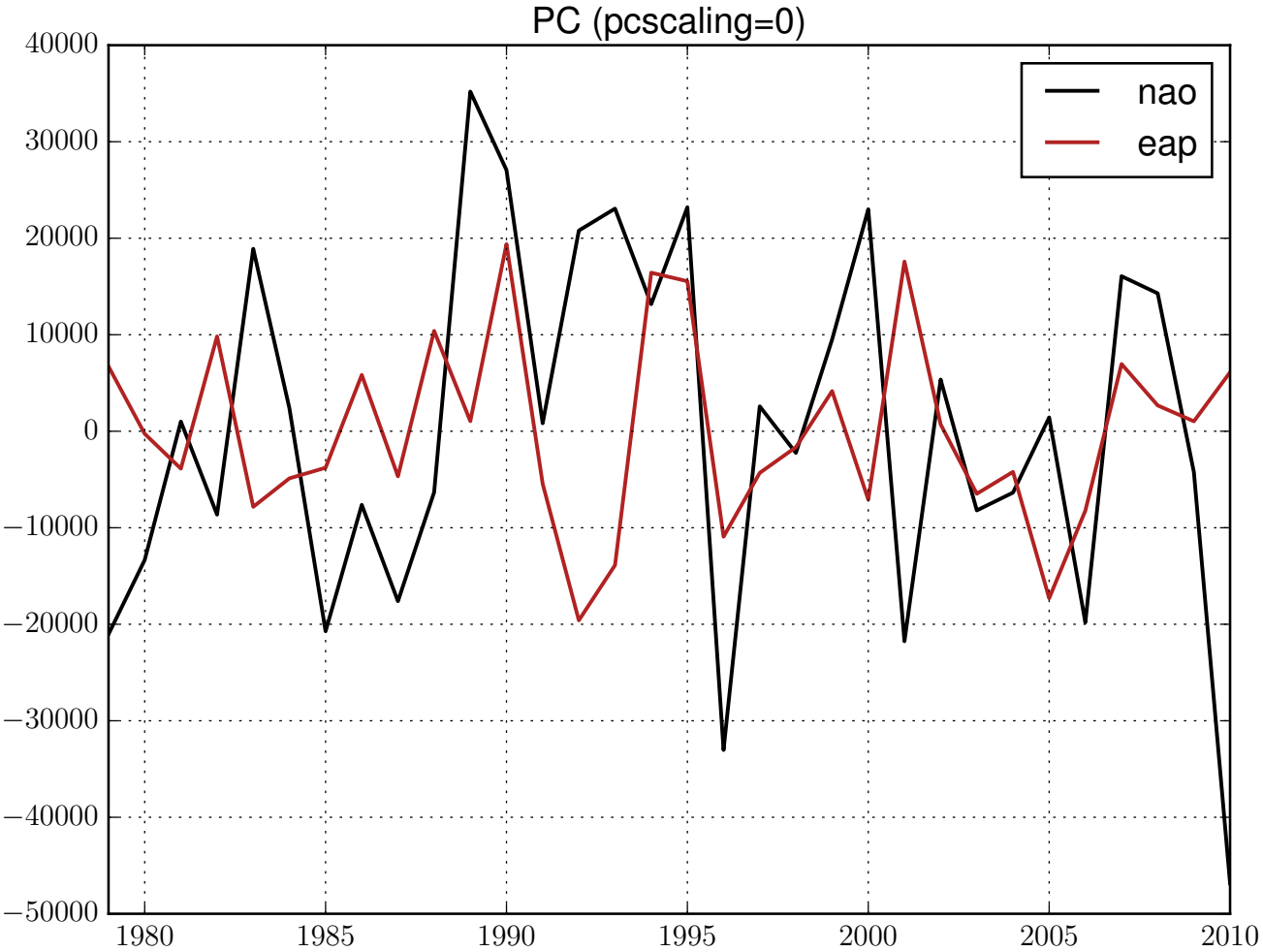
Extraction of EOF maps



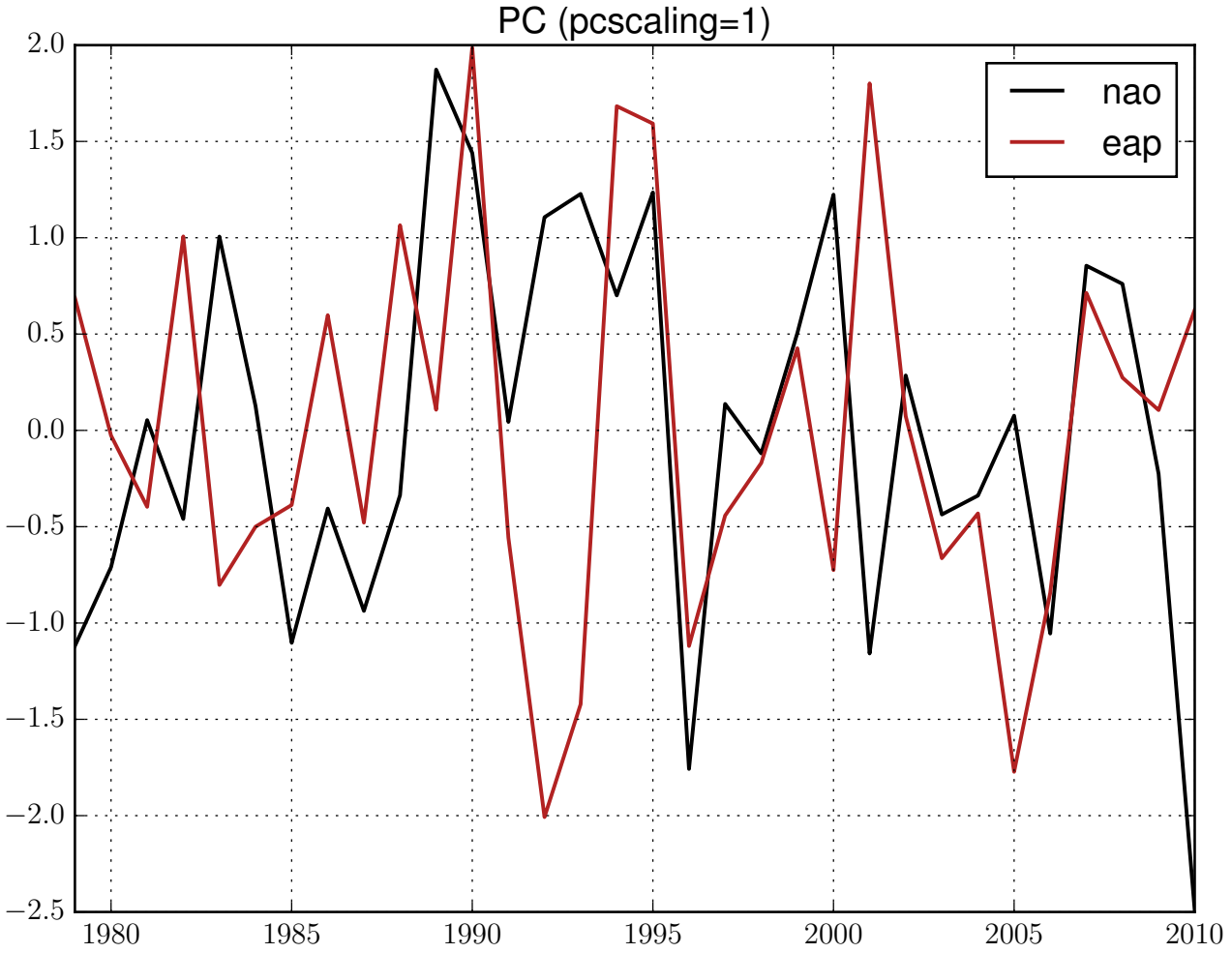
Extraction of EOF PCs

```
# Un-scaled PCs (default).
pc0 = eofobj.pcs(pcscaling=0, npcs=2)

# PCs are scaled to unit variance
# (divided by the square-root of their eigenvalue).
pc1 = eofobj.pcs(pcscaling=1, npcs=2)

# PCs are multiplied by the square-root of their eigenvalue.
pc2 = eofobj.pcs(pcscaling=2, npcs=2)

# array size: (nt, 2)
```

Extraction of EOF PCs



Extraction of EOF PCs



Miscellaneous methods

```
# projection of a field on the EOFs
# weighting is automatically done
proj = eofobj.projectField(psl[[0, 1, -1], :, :],
                           neofs=2, eofscaling=0)
# size of the array: (3, 2)

# recovering the explained variance
expvar = eofobj.varianceFraction(neigs=2) * 100
# [53.8 14.5]
```