AIMS CDT Signal Processing Lab Session 1
Classical Signal Processing

Xiaowen Dong (xdong@robots.ox.ac.uk)

Part 1 - Auto- and cross-regressive models.

In this lab session[1], we will be looking into developing *auto- and cross-regressive models*. Recall that the generic form of the auto-regressive model is

$$\hat{y}[t] = \sum_{i=1}^{p} a_i y[t-i].$$

We can solve for the unknown set of coefficients, $\{a_i\}$, in two different ways:

1. By forming the *embedding* matrix, $\mathbf{M}$, from lagged versions of a data series. As $\mathbf{M}$ in general is non-square, we will have to use the pseudo-inverse, i.e., evaluate $(\mathbf{M^T M})^{-1}\mathbf{M^T}$.

2. Estimate the *autocorrelation* matrix, $\mathbf{R}$, and use $\mathbf{R^{-1}}$ to infer the coefficients.

Preamble Load the data, provided at `http://www.robots.ox.ac.uk/~xdong/teaching/aims/lab/qbo.txt`. This consists of three data streams from temperature readings associated with the Quasi-Biennial Oscillation (QBO)[2], sampled at one month intervals. These are shown below.

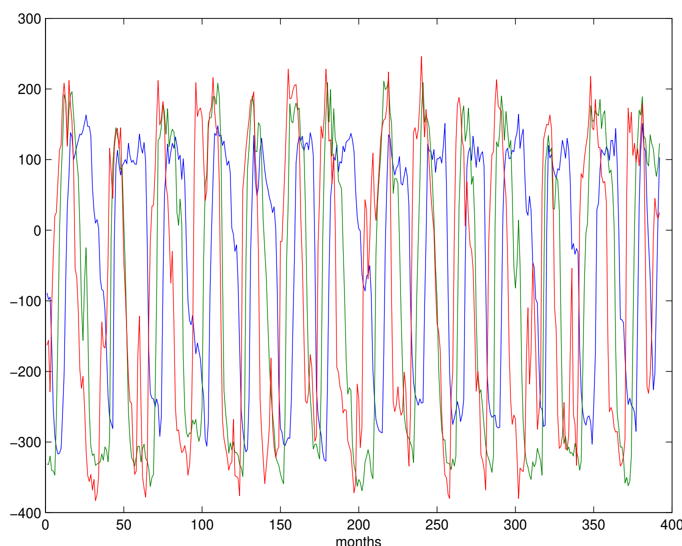

Figure 1: QBO data.

[1]This lab session was originally devised by Steve Roberts.

[2]https://en.wikipedia.org/wiki/Quasi-biennial_oscillation

Direct 1-D solutions We start out by looking at one of these time series - up to you which one.

- By evaluating the least-squares solution for $\mathbf{a}$ using the embedding method, write code to infer the coefficients.
- Re-evaluate the solution for $\mathbf{a}$ using a direct inversion of the autocorrelation matrix.
- Use a Toeplitz solver - the Yule-Walker recursions, for example, to evaluiate the coefficients rather than a direct inversion of the autocorrelation matrix. For large model orders, $p >> 1$, can you notice a difference in speed?
- As the AR models represent *one-step prediction* models, you can use this as a metric of performance.

Cross-regression Modify the embedding matrix method, so that

$$\hat{y}[t] = \sum_{i=1}^{p} a_i z[t-i]$$

in which y is being modelled by observing another timeseries, z. The coefficients of this model now describe the cross-regression i.e. how the past of z effects the present value of y. We can look at the magnitude of the coefficients as well as the predictions of y to give an idea about the information that 'flows' from z to y. Do any of the timeseries have strong interactions? If so, is there any indication of which one is driving which?

Regularisation If you have the time, you can look at regularising the solutions of the least-squares linear models. From a *Bayesian* perspective this is equivalent to putting *priors* on the coefficients which penalise large values. A simple method to achieve this is *shrinkage*, which adds 'jitter', of magnitude α say along the leading diagonal of anything we invert - in other words $\mathbf{A}^{-1}$ becomes $(\mathbf{A} + \alpha\mathbf{I})^{-1}$. This has the effect of helping reduce rank-deficiency in $\mathbf{A}$ and has the knock on, in our system, of *shrinking* the values of $\mathbf{a}$ towards zero. If you get a chance try it, and see what effect it has. You should be able to obtain coefficients for model orders close to the number of data points without getting singular solutions.

[Checkpoint] You are now done with the first part of the exercises. Please ask a lab demonstrator to evaluate your work.

Part 2 - Image processing.

In this lab session[3], we look at several classical image processing operations based on the representations enabled by the Fourier and wavelet transform. First, download the toolbox provided at `https://github.com/gpeyre/numerical-tours/raw/master/python/nt_toolbox.zip`, and make sure the folder *nt_toolbox* is in the current working directory. Then, install and import the necessary Python modules.

```
python3 -m pip install numpy scipy matplotlib scikit-learn scikit-image
import numpy as np
import scipy as scp
import pylab as pyl
import matplotlib.pyplot as plt
from nt_toolbox.general import *
from nt_toolbox.signal import *
```

Plot an image. Let's load and plot an example image.

```
n0 = 512
f = rescale(load_image("nt_toolbox/data/hibiscus.bmp", n0))
plt.figure(figsize = (5, 5))
imageplot(f, 'Image f')
```

We can zoom in to see more detail of one part of the image.

```
plt.figure(figsize = (5, 5))
imageplot(f[n0//2 - 32:n0//2 + 32, n0//2 - 32:n0//2 + 32], 'Zoom')
```

[3] This lab session was largely based on the excellent Numerical Tours by Gabriel Peyré.

Fourier transform. Now let's perform the 2-D Fourier transform.

```
F = pyl.fft2(f)/n0 # forward transform

from pylab import linalg
print("Energy of Image:   %f" %linalg.norm(f))
print("Energy of Fourier: %f" %linalg.norm(F))
```

- Why is the signal energy preserved in the frequency domain?

Display the log amplitude of the Fourier coefficients. The *fftshift* function shifts the zero-frequency component to the centre of the spectrum.

```
L = pyl.fftshift(np.log(abs(F) + 1e-1))
plt.figure(figsize = (5, 5))
imageplot(f, 'Image', [1, 2, 1])
imageplot(L, 'Log(Fourier transform)', [1, 2, 2])
```

Filtering. One of the most common signal processing operations is filtering. Let's consider a low-pass filtering example, where we only keep M coefficients corresponding to the low frequencies while setting the rest to zero, before carrying out an inverse transform.

```
M = n0**2//64
q = int(np.sqrt(M))
F = pyl.fftshift(pyl.fft2(f))
Sel = np.zeros([n0, n0])
Sel[n0//2 - q//2:n0//2 + q//2, n0//2 - q//2:n0//2 + q//2] = 1
F_zeros = np.multiply(F, Sel)
imageplot(np.log(abs(F_zeros) + 1e-1), 'Log(Fourier transform)') # plot in frequency domain

f_lpf = np.real(pyl.ifft2(pyl.fftshift(F_zeros))) # inverse transform
plt.figure(figsize = (5, 5))
imageplot(f, 'Image', [1, 2, 1])
imageplot(clamp(f_lpf), 'Low-pass filtered image', [1, 2, 2])
```

- Can you observe the difference between the original image and the filtered one?
- Now perform a high-pass filtering (how can you do this?). Can you explain the filtering outcome?

Fourier vs wavelet transform. The low-pass filtering example above corresponds to image approximation using M low-frequency Fourier coefficients. We can also use the wavelet transform for the same purpose and compare the outcome. Let's first perform the 2-D wavelet transform.

```
Jmin = 0
from nt_toolbox.perform_wavelet_transf import *
Fw = perform_wavelet_transf(f, Jmin, + 1) # forward transform
plt.figure(figsize = (5, 5))
plot_wavelet(Fw)
plt.title('Wavelet coefficients')
plt.show()
```

Let's implement a wavelet approximation by only keep the M largest wavelet coefficients while setting the rest to zero.

```
a = np.sort(np.ravel(abs(Fw)))[::-1] # sort a 1D copy of fw in descending order
Fw_zeros = np.multiply(Fw,(abs(Fw) > a[M])) # keeping the M largest wavelet coefficients
plt.figure(figsize = (5, 5))
plot_wavelet(Fw_zeros) # comapre with the plot of coefficients above
```

```
plt.title('Wavelet coefficients (thresholded)')
plt.show()

f_wav = perform_wavelet_transf(Fw_zeros, Jmin, -1) # inverse transform
plt.figure(figsize = (5, 5))
imageplot(clamp(f_lpf), 'Fourier approximation', [1, 2, 1])
imageplot(clamp(f_wav), 'Wavelet approximation', [1, 2, 2])
```

- Can you comment on the comparison between the two approximations above. Why is one better than the other?

Denoising. The final example is image denoising, i.e., removing noise from an image. First let's generate and plot a noisy image.

```
n = 256
N = n**2
name = 'nt_toolbox/data/boat.bmp'
x0 = load_image(name, n)
plt.close('all')
imageplot(x0)
sigma = .08
y = x0 + sigma*np.random.standard_normal(x0.shape)
imageplot(clamp(y))
```

We will implement a denoising strategy via convolution. Define a Gaussian convolutional kernel as follows.

```
cconv = lambda a, b : np.real( pyl.ifft2(pyl.fft2(a)*pyl.fft2(b)) )
normalize = lambda h : h/sum(h.flatten())
t = np.transpose( np.concatenate( (np.arange(0, n/2), np.arange(-n/2, 0) ) ) )
[Y, X] = np.meshgrid(t, t)
h = lambda mu: normalize(np.exp(-(X**2 + Y**2)/ (2*mu**2)))
```

Now denoise the image by convolving it with a Gaussian kernel with a given width.

```
mu = 5
plt.figure(figsize = (5, 5))
imageplot(pyl.fftshift(h(mu)), 'kernel', [1, 2, 1])
imageplot(pyl.fftshift(np.real(pyl.fft2(h(mu)))), 'FT of kernel', [1, 2, 2])
denoise = lambda x, mu : cconv(h(mu), x)
plt.figure(figsize = (5, 5))
imageplot(x0, 'Image', [1, 3, 1])
imageplot(y, 'Noisy image', [1, 3, 2])
imageplot(denoise(y, mu), 'Denoised image', [1, 3, 3])
```

- Vary the kernel width μ, and compare the denoising outcome.
- Can you find an optimal μ that minimises the ℓ_2-norm of the difference between original and denoised image?

[Checkpoint] You are now done with the second part of the exercises. Please ask a lab demonstrator to evaluate your work.

[Optional] You can try more advanced image processing exercises as part of the Numerical Tours at `http://www.numerical-tours.com/python/`.