Fourier Analysis for neuroscientists
A practical guide using Matlab

Dr Cyril Pernet – February 2012

Introduction

The goal of the *Fourier transform* is to perform a frequency analysis of a signal, i.e. transform a signal in the time or space domain into a signal in the frequency domain. The idea is that one can decompose any signal as a sum of cosine and sine functions of various frequencies.

Such decomposition is not unusual. Consider a ray of natural (white) light, it is describe both as a set of photons and a wave. When this wave hits a prism, it is decomposed into different colour bands (figure 1). As neuroscientists, we know that colour is a percept made by our brain as a result of the stimulation of different photo-pigments in the retina; stimulation which depends on the frequency of the waves received.

Figure 1: decomposition of light
into different frequencies
(credit: https://lasers.llnl.gov/education/fusion_fun/recipes/crystals.php)

A. decomposition of a 1D signal (Matlab Signal Processing Toolbox)

1. Real Fourier Series

$$f(t)=\frac{a_0}{2}+\sum_{n=1}^{\infty} a_n\cos(nt)+\sum_{n=1}^{\infty} b_n\sin(nt) \quad \text{eq (1)}$$

with t representing the time, a0 a constant standing for the non oscillatory part of the signal and an, bn representing the strength of each frequency component.

From 0 to 360 degrees or -pi to pi (radian), cosine and sine functions of different frequencies are orthogonal, meaning that the integral of their product = 0. Furthermore, the integral of the square = pi. Using these properties, we can split the above equation for cosine and sine to obtain an independent estimation of parameters a and b.

$$a_n=\frac{1}{\pi}\int_{-\pi}^{\pi} f(t)\cos(nt)dt \quad \text{and} \quad b_n=\frac{1}{\pi}\int_{-\pi}^{\pi} f(t)\sin(nt)dt \quad \text{eq(2) eq(3)}$$

In the real word, instead of integrating form -pi to pi we would, for each time point t, sum from -L to L (L being half of the length of the signal). This mean we can write this equation into Matlab using a loop and sum. However, if the signal is very long this will take a long time to compute.

2. Complex, Discrete and Fast Fourier Transforms

One can take advantage of the Euler identity (eq 4) to rewrite equation (1), giving the *complex Fourier transform* (eq 5). Application to the discrete domain is called the *Discrete Fourier Transform* (DFT) and special algorithms the speed up the analysis have been developed and are regrouped under the term *Fast Fourier Transforms* (FFT). We do not present more equations at this point, as this is beyond the scope of this practical guide - all equations can be found e.g. on Wikipedia. Note however that we moved from the real series which uses sines and cosines (a and b are real numbers) to a complex series, i.e. we now use a complex exponential (a is real and b imaginary). As we will see latter this has advantages to quickly compute the phase of the signal.

$$e^{iwt} = \cos wt + i \sin wt \quad \text{Eq (4)}$$

$$f(t) = \sum_{n=-\infty}^{\infty} c_n e^{int} \quad \text{Eq (5)}$$

Matlab offers a convenient way to perform such decomposition using the **fft** function (figure 2).

```
% Generate a signal consisting of 3 sinusoids, one at 50 Hz,
% one at 120 Hz with twice the amplitude and one at 300Hz.

% Time
sampling_rate = 1/1000; % 1KHz
t = (0:sampling_rate:1)';
L = length(t) / 2;

% Signal
y = sin(2*pi*50*t) + 2*cos(2*pi*120*t) + sin(2*pi*300*t);
figure; subplot(2,2,1); plot(t(1:100),y(1:100),'LineWidth',3);
grid on; title(sprintf('50Hz sine + 120Hz cos \n + 300Hz sine'),'Fontsize',14);
xlabel('time','Fontsize',12); ylabel('Amplitude','Fontsize',12)

% decompose/reconstruct the signal
coef = fft(y);
subplot(2,2,2); plot(real(coef(1:L)),'LineWidth',3); hold on
plot(imag(coef(1:L)),'r','LineWidth',3); grid on;
title(sprintf('Real and imaginary part \n of the fft'),'Fontsize',14);
xlabel('frequency','Fontsize',12); ylabel('Amplitude','Fontsize',12)

reconsty = ifft(coef);
subplot(2,2,3); plot(t(1:100),reconsty(1:100),'g','LineWidth',3);
grid on; title('inverse fft','Fontsize',14)
xlabel('time','Fontsize',12); ylabel('Amplitude','Fontsize',12)

diff = y - reconsty;
subplot(2,2,4); plot(t(1:100),diff(1:100),'k','LineWidth',3);
grid on; title('Difference','Fontsize',14)
xlabel('time','Fontsize',12); ylabel('Amplitude','Fontsize',12)
```

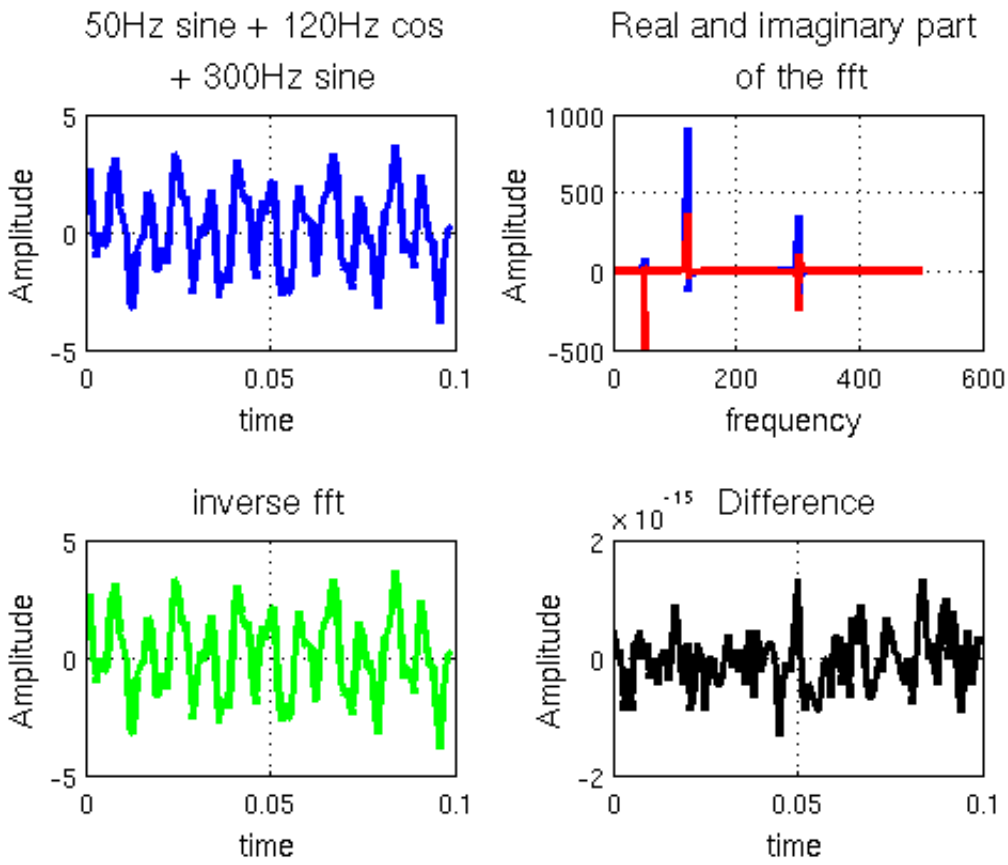

Figure 2: illustration of fft and inverse fft - note that the back projection is nearly perfect (the scale is 10^{-15})

3. Amplitude Spectrum

The goal of the FFT is not to reconstruct the signal but investigate the relative weight of the different frequencies in the signal (the coefficient a, b). Note that the FFT returns both real and imaginary numbers (figure 2). The strength of the coefficients is then their absolute values.

Parseval's theorem states that the energy of a signal in the time domain equals the energy of the transformed signal in the frequency domain. Since one sums over all data points it is essential to normalize the amplitude spectrum. The normalization is then simply a division by the number of data point.

By default, Matlab assumes that the frequencies in the signal are evenly spaced from 0 to the *Nyquist limit*. Nyquist showed that for discrete signals, the maximum frequency is half of the sampling rate (figure 3). When plotting the amplitude spectrum, it is therefore necessary to input the frequency limits.

```
coef = fft(y) / length(t); % normalize by the number of time points
Nyquist_Limit = (1/sampling_rate)/2; % = 500Hz
x = linspace(0,1,L)*Nyquist_Limit;
figure; subplot(1,2,1);
plot(x,abs(coef(1:L),'LineWidth',3);
xlabel('frequency','Fontsize',12); ylabel('amplitude','Fontsize',12)
axis tight; grid on; title('Original Signal')

% imagine we recorded this signal at 200Hz
ys = downsample(y,5);
Nyquist_Limit = ((1/sampling_rate)/5)/2; % = 100Hz
x = linspace(0,1,length(ys)/2)*Nyquist_Limit;
subplot(1,2,2); coef = fft(ys) / length(ys);
plot(x,abs(coef(1:length(ys)/2)),'LineWidth',3);
xlabel('frequency','Fontsize',12); ylabel('amplitude','Fontsize',12)
axis tight; grid on; title('Sampled at 200Hz')
```

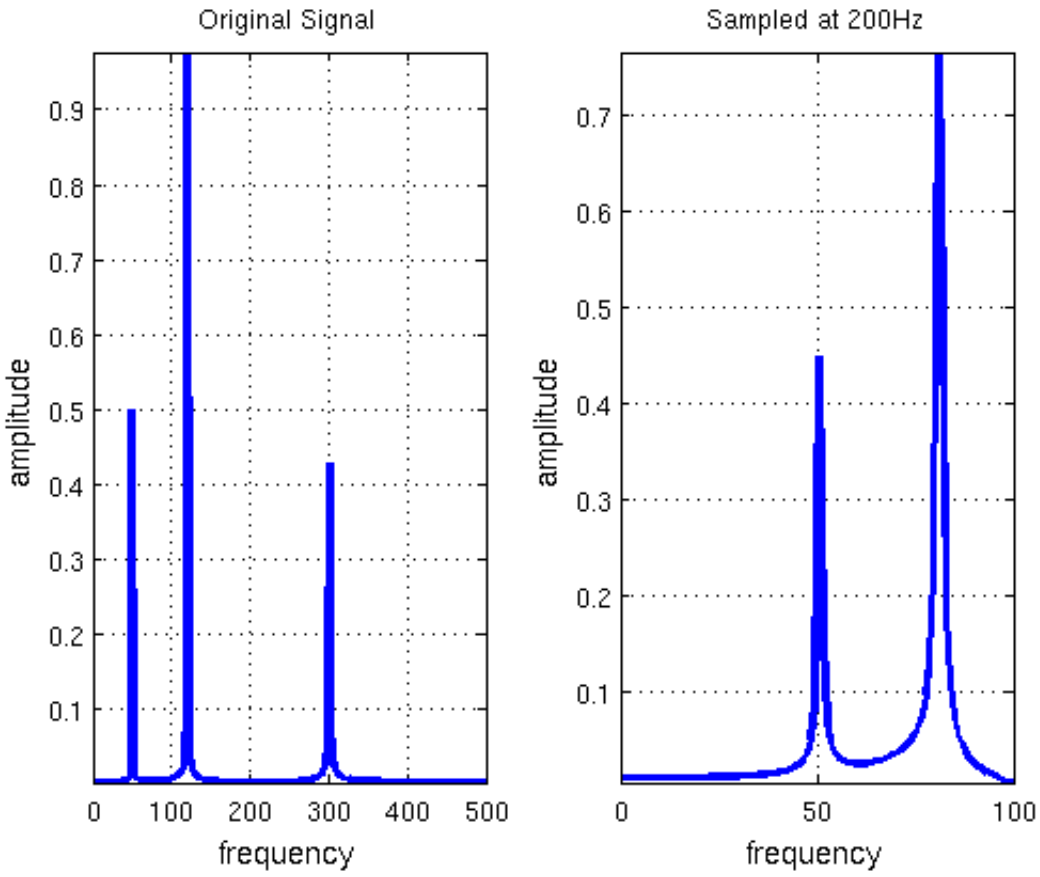

Figure 3: Illustration of the Nyquist's theorem: the original spectrum includes the 3 frequencies of the original data (50, 120, 300Hz) – sampling this signal at 200Hz, only the 50Hz component is properly recovered. The subsampling introduced a spurious frequency at 80Hz.

4. Power

The power of a signal is simply its' squares normalized. For complex values in the frequency domain this is equivalent to a multiplication of the complex values by the normalized complex conjugate – function **conj** in Matlab.

```
% recompute on the original signal
coef = fft(y) / length(t);
Nyquist_Limit = (1/sampling_rate)/2; % = 500Hz
x = linspace(0,1,L)*Nyquist_Limit;
subplot(1,2,1); plot(x,abs(coef(1:L)),'LineWidth',3);
xlabel('frequency','Fontsize',12); ylabel('amplitude','Fontsize',12)
axis tight; grid on; title('amplitude spectrum')
% get the power
P = coef(1:L).*conj(coef(1:L)) / length(t);
subplot(1,2,2); plot(x,P,'LineWidth',3);
xlabel('frequency','Fontsize',12); ylabel('Power','Fontsize',12)
axis tight; grid on; title('Power spectrum')
```

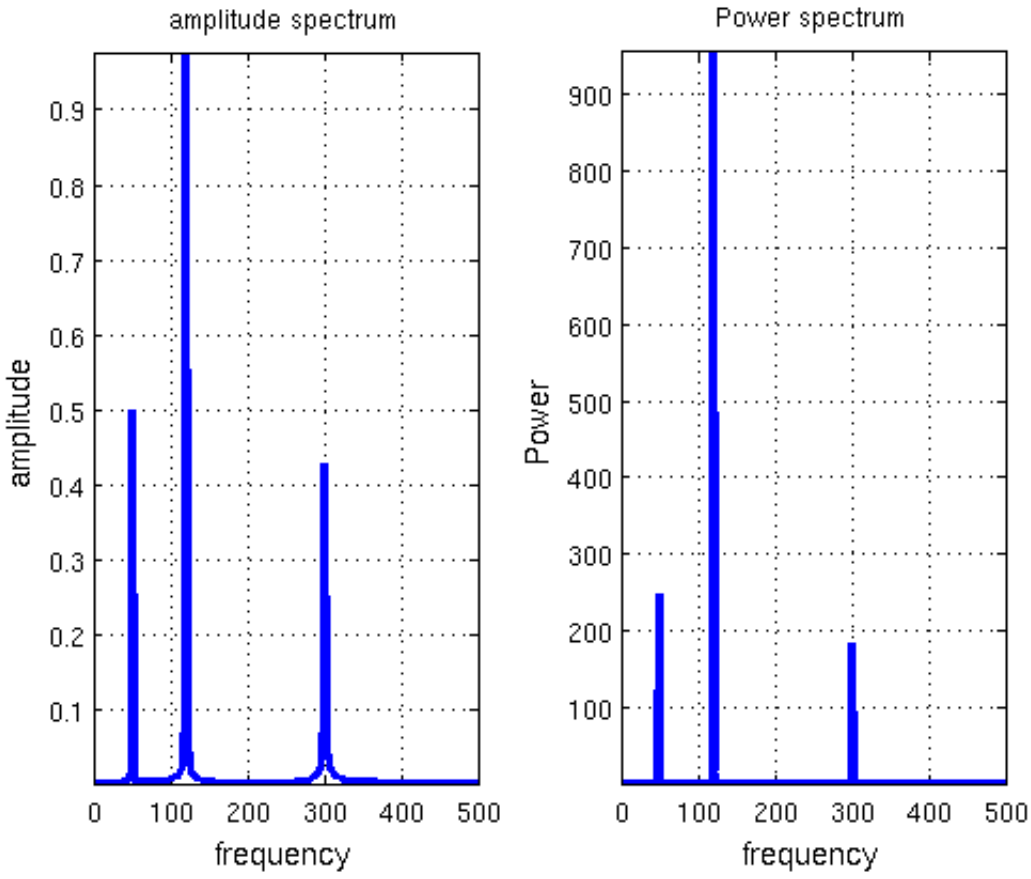

Figure 4: Amplitude and power spectra.

Although it is interesting to know the power of each frequencies, it can also be of interest to compute the overall power of the signal. This can be achieve in the frequency domain by integrating the power spectral density. The Matlab signal processing toolbox allows to do this easily by 1. computing the power automatically at each frequency and 2. convert into power spectral density, i.e a value of power per frequency unit. To do this we call the mean square spectrum (**msspectrum**) and the power spectral density (**psd**) functions.

```
hp = spectrum.periodogram('hamming');  % Create periodogram object
hpopts = psdopts(hp,x); % Create options object
set(hpopts,'Fs',sampling_rate,'SpectrumType','twosided','centerdc',true); % set
properties
msspectrum(hp,y,hpopts); % like the normalized power
hpsd = psd(hp,y,hpopts); % measures power per unit of frequency
figure; plot(hpsd);
```

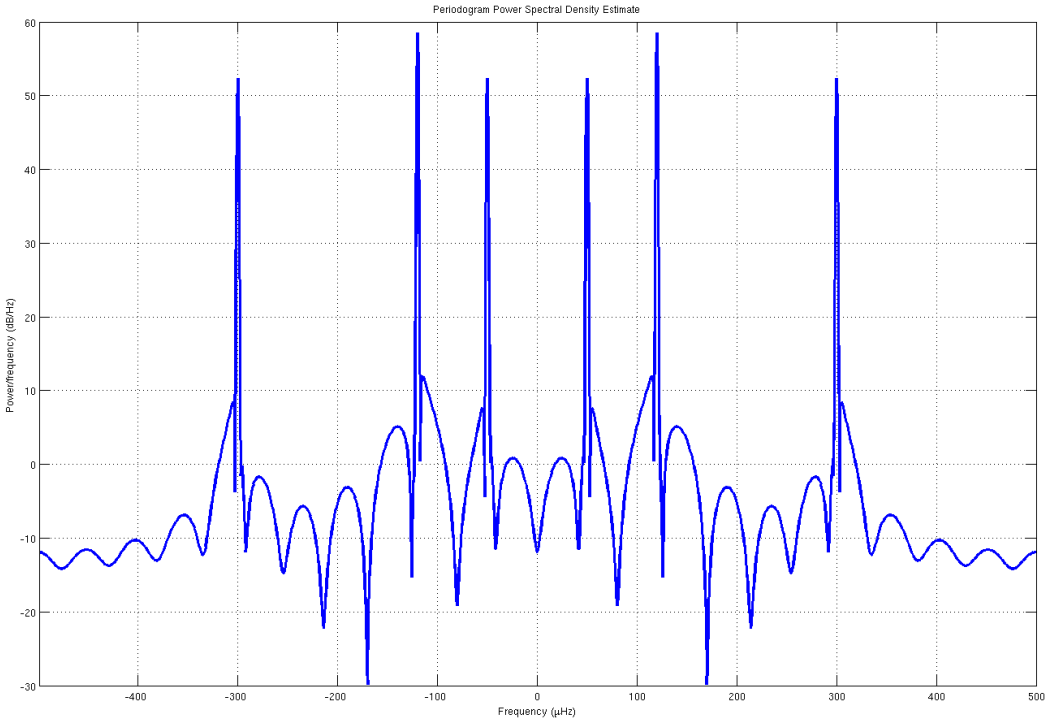

Figure 5. Power spectral density

```
power_freqdomain = avgpower(hpsd);
power_timedomain = sum(abs(y).^2)/length(y);
```

Simply averaging the psd allows to obtain the average power of the signal. However, we can get this result much faster – remember the *Parseval's theorem?* We can simply integrate the signal^2 and normalize to obtain the same result.

5. Phase analysis and Coherence

Since the signal is modelled by cosines and sines, this is not just the amplitude (or power) that can be obtained but also the phase of the different components (figure 6). Since we used an fft, coefficients represent now a real and imaginary exponential. On a complex plane, the x abscissa represents real values whilst the y ordinate represents imaginary values. Thus, a complex value from the FFT is represented by a pair (x,y) with the magnitude being the distance to the origin and the phase being the angle between the x-axis and the vector formed by this pair. Using standard trigonometry rule

angle theta = $(\tan)^{-1}(\frac{imaginary}{real})$ Eq. (6)

```
figure; subplot(1,2,1);
%y = sin(2*pi*50*t) + 2*cos(2*pi*120*t) + sin(2*pi*300*t);
y1 = sin(2*pi*50*t);
y2 = 2*cos(2*pi*120*t);
y3 = sin(2*pi*300*t);
plot(t(1:60),y1(1:60),'LineWidth',3); hold on
plot(t(1:60),y3(1:60),'g','LineWidth',3);
plot(t(1:60),y2(1:60),'r','LineWidth',3);
grid on; title('Original data','Fontsize',14);
xlabel('time','Fontsize',12); ylabel('Amplitude','Fontsize',12)

theta = atan(imag(coef)./real(coef));
subplot(1,2,2); plot(x(1:60),theta(1:60)./(pi/180),'LineWidth',3);
grid on; title('Phase','Fontsize',14); axis tight
xlabel('time','Fontsize',12); ylabel('Phase','Fontsize',12)

% signal processing as a function to do just that
phi = unwrap(angle(coef));  % Phase
figure;plot(x(1:60),phi(1:60)./(pi/180));
```

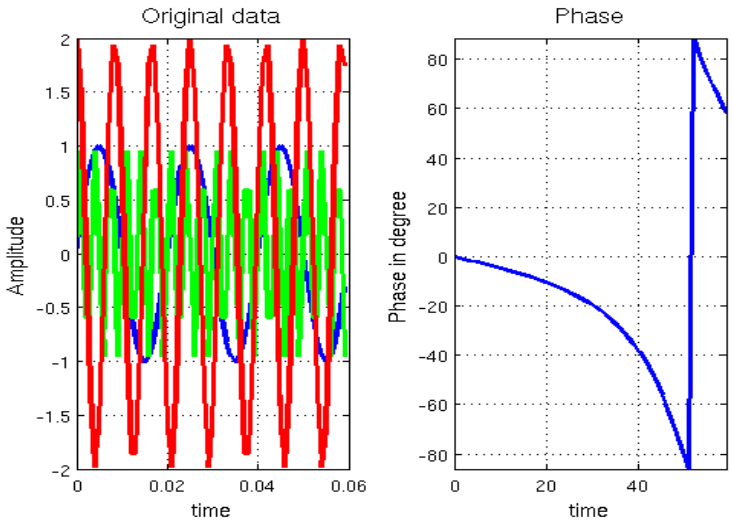

Figure 6: decomposed signal and phase plot

6. Coherence

The coherence represents by how much two signals are in phase. It is defined by the cross-spectrum normalized by the product of the spectrum of each signal. Note in eq. 7 the coherence is defined for a given frequency and analyses are usually performed to obtain coherence values across the range of available frequencies (figure 7).

$$C(\omega)=\frac{|S_{xy}(\omega)|^2}{S_x(\omega)S_y(\omega)} \quad \text{Eq. (7)}$$

This is easily done in Matlab using the **mscohere** function.

```
[C13,F13] =
mscohere(y1,y3,hanning(round(L)),round(L/2),round(L),1/sampling_rate);
figure; plot(F13,C13,'r','LineWidth',3);
title('Coherence at each frequencies','Fontsize',14); axis tight
xlabel('frequencies','Fontsize',12); ylabel('Coherence','Fontsize',12)
[C12,F12] =
mscohere(y1,y2,hanning(round(L)),round(L/2),round(L),1/sampling_rate);
hold on; plot(F12,C12,'LineWidth',3); grid on
```

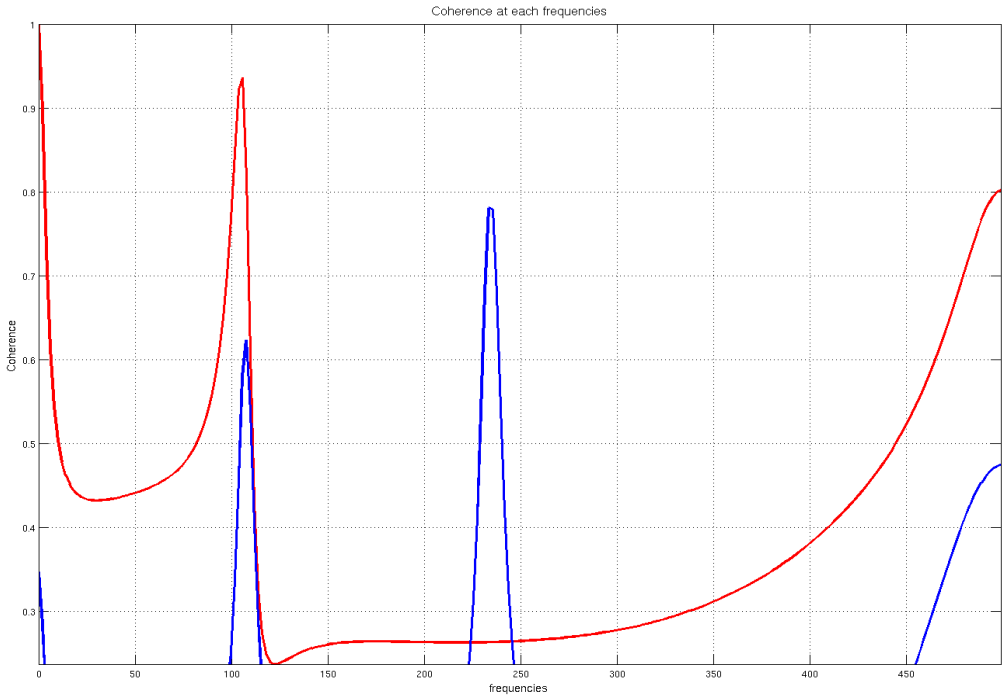

Figure 7: Coherence values between y1 (50Hz) and y2 (120Hz) in blue and between y1 and y3 (300Hz) in red. Because y1 and y3 are multiple of each other they exhibit periods of higher coherence. Note the coherence is very smooth because we defined fft over overlapping hanning windows.

7. Time-frequency analysis

So far we examined a signal that is *stationary*. This doesn't mean that the signal is the same (since we used sine and cosine functions clearly it changed over time) but the properties of the signal are the same in time. For instance if the signal we analysed above was twice as long, the amplitude/power spectra would be the same, and the Fourier transform assumes stationarity.

How to analyse a non stationary signal? Assuming the signal is stationary by small windows, one can divide the signal in small windows and compute multiple Fourier transforms. To ensure that the spectrum is well computed over the whole signal, overlapping windows are also used, which forms the basis of the *short-time Fourier transform* (STFT). In addition because on the edge of each window one would 'break' abruptly the signal, special shapes (tapered windows) are used. The main role of the window is to damp out the effects of the *Gibbs phenomenon* that results from truncation of an infinite series. Informally, it means that the Gibbs phenomenon is the observation of oscillations near sharp transitions and it reflects the difficulty in approximating a non stationary signal (or a discontinuous function) by a *finite* series of continuous sine and cosine waves.

The Matlab function to perform such analysis is: **[S,F,T,P]=spectrogram(x,window,noverlap,F,fs)**

- x is the signal to be analyzed
- window is the size of the Hamming window
- noverlap is the number of samples that each segment overlaps. The default value is the number producing 50% overlap between segments.
- F a vector of frequencies a which the spectrogram is computed
- fs is the sampling frequency, which defaults to normalized frequency

S is the STFT, F is the rounded values of F (input) used in each DFT bin, T the time points corresponding to the STFT and P is the *power spectral density* (PSD) of each segment.

```
%% Time - frequency analysis
clear all
[data,sampling_rate]=wavread('song1.wav');
t = [1:length(data)];
L = length(t) / 2;

% Signal
figure; subplot(2,2,1); plot(data,'LineWidth',3);
grid on; title(sprintf('bird song'),'Fontsize',14);
xlabel('time','Fontsize',12); ylabel('Amplitude','Fontsize',12); axis tight

% what if we do an fft
coef = fft(data);
P = coef(1:L).*conj(coef(1:L))/ length(t);
Nyquist_Limit = (1/sampling_rate)/2;
x = linspace(0,1,length(t)/2)*Nyquist_Limit;
subplot(2,2,2); plot(x,P,'LineWidth',3);
xlabel('frequency','Fontsize',12); ylabel('Power','Fontsize',12); axis tight
axis tight; grid on; title('Power spectrum')

% time-frequency decomposition
subplot(2,2,3); spectrogram(data,256,[],[],sampling_rate,'yaxis')
title('Spectrogram')
```

```
% [S,F,T,P]=spectrogram(x,window,noverlap,F,fs)
[S,F,T,P]=spectrogram(data,256,[],[],sampling_rate);
subplot(2,2,4); mesh(abs(P));
xlabel('Time (Seconds)'); ylabel('Hz'); zlabel('power')
title('Power Spectral density')
```

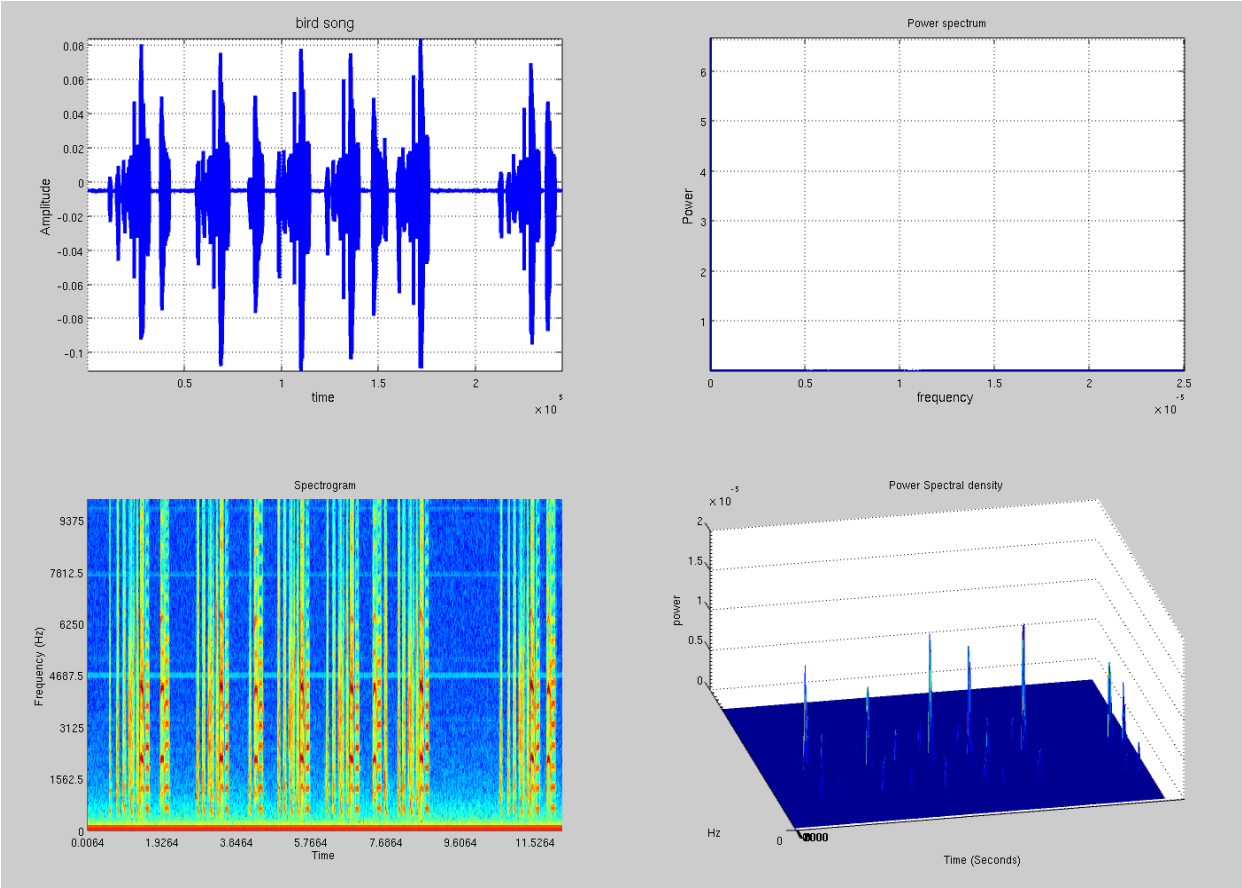

Figure 8: Time-frequency decomposition. At the top a non stationary signal and it's FFT – we can see the FFT fails at capturing any relevant information. Using time bins, the STFT shows frequencies at which one has maximum signal (bottom).

```
% we can also look at the effect of windowing
figure; subplot(4,1,1); w = rectwin(256); % rectangle
spectrogram(data,w,[],[],sampling_rate,'yaxis'); title('rectangular
window','Fontsize',14)

subplot(4,1,2); w = bartlett(256); % like a triangle
spectrogram(data,w,[],[],sampling_rate,'yaxis'); title('bartlett
window','Fontsize',14)

subplot(4,1,3); w = gausswin(256); % gaussian
spectrogram(data,w,[],[],sampling_rate,'yaxis'); title('gaussian
window','Fontsize',14)
```

```
subplot(4,1,4); w = hamming(256); % like gaussian - default
spectrogram(data,w,[],[],sampling_rate,'yaxis'); title('hamming
window','Fontsize',14)
```

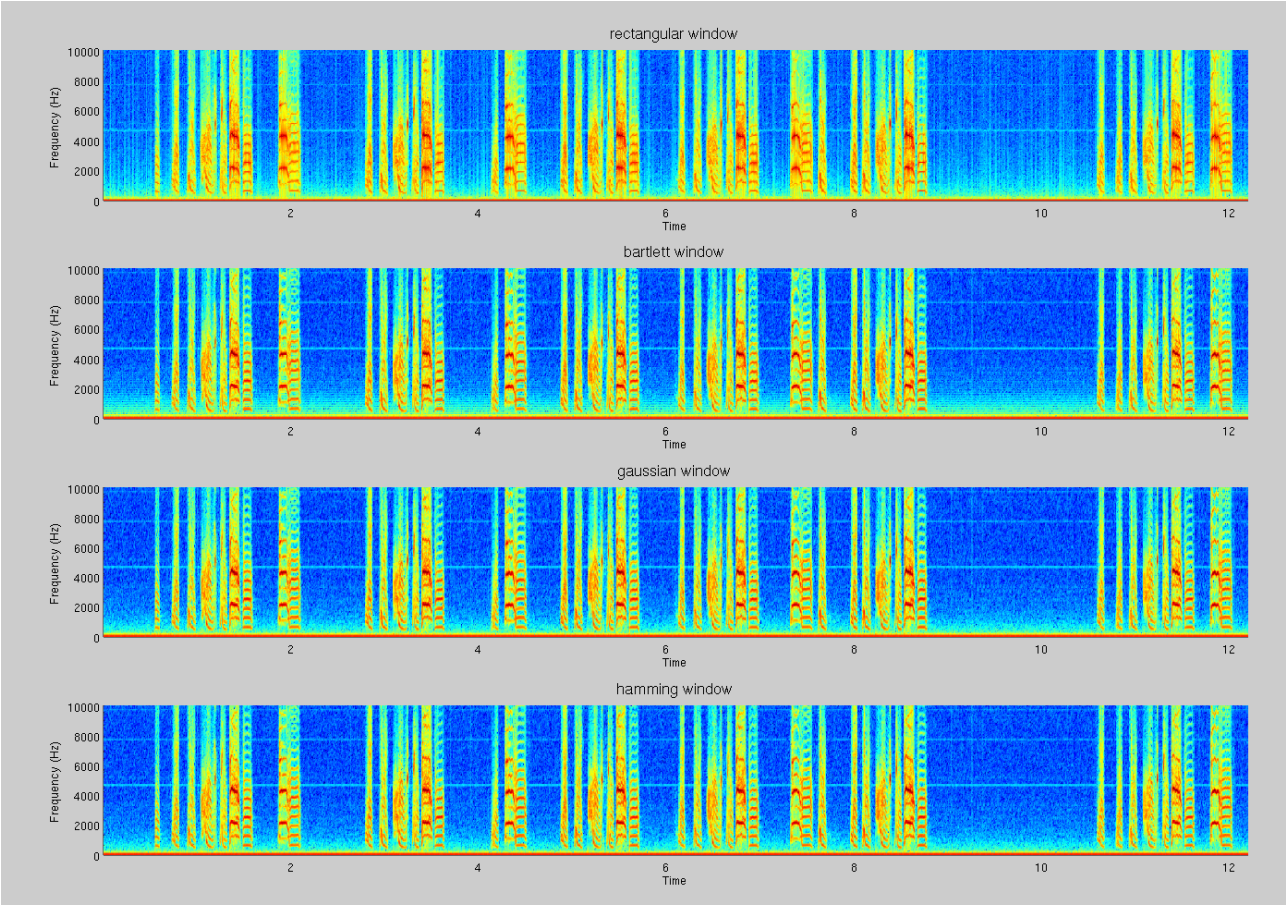

Figure 9: effect of using different windows on the STFT.

Finally, it is also important to understand that there is a trade-off time vs. frequency in terms of resolution – with short windows we have lots of time information but the frequency resolution is poor and vise versa

```
% we can also illustrate the time/frequency trade-off
% if we use long windows we can see well low-frequencies
figure
windows = [256, 2048, 8192];
for w=1:3
    subplot(1,3,w); spectrogram(data,windows(w),[],[],sampling_rate,'yaxis')
    title(['Spectrogram with ' num2str((windows(w)/sampling_rate)*1000) 'ms
window'])
    [S,F,T,P]=spectrogram(data,256,[],[],sampling_rate);
    set(gca,'YTick',F(1:20:end)); set(gca,'XTick',T(1:300:end))
end
```

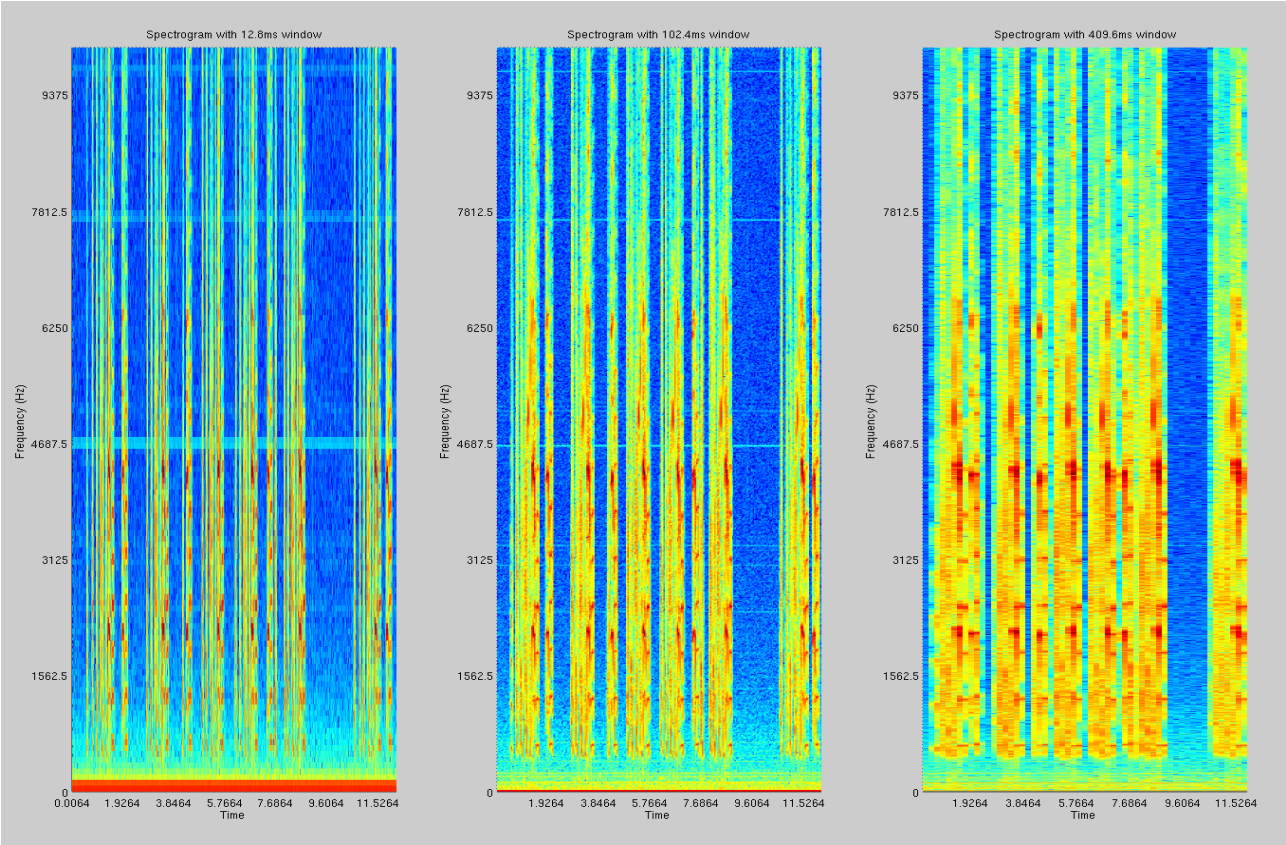

Figure 10: Time/frequency trade off of the STFT

B. Application to 2D signal (Matlab Image processing toolbox)

As we will see the same principals as presented above can be applied to e.g. images, the Matlab function to do that is **fft2** and **fftn.** Similar operations can be performed (usually better) using filters, also available in Matlab.

The Matlab help reads 'The Fourier transform is a representation of an image as a sum of complex exponentials of varying magnitudes, frequencies, and phases. The Fourier transform plays a critical role in a broad range of image processing applications, including enhancement, analysis, restoration, and compression.'

1. Definition

If *f*(*m*,*n*) is a function of two discrete spatial variables *m* and *n*, then the *two-dimensional Fourier transform* of *f*(*m*,*n*) is defined by the relationship

$$F(\omega 1,\omega 2)=\sum_{m=-\infty}^{\infty}\sum_{n=-\infty}^{\infty} f(m,n)\mathrm{e}^{-j\omega 1\mathrm{m}}\mathrm{e}^{-j\omega 2\mathrm{n}} \quad \text{Eq (8)}$$

with ω_1 and ω_2 the frequency variables in radians per sample. $F(\omega_1,\omega_2)$ is the *frequency-domain*

representation of $f(m,n)$. $F(\omega_1,\omega_2)$ is a complex-valued function that is periodic both in ω_1 and ω_2, with period 2π.

Just as for 1D signal, we can use a DFT to manipulate discrete objects (rather than continuous functions). For an image of dimension m by n the DFT is

$$F(p,q)=\sum_{m=0}^{M-1}\sum_{n=0}^{N} f(m,n)\mathrm{e}^{-j2\pi qn/M}\mathrm{e}^{-j2\pi qn/N} \quad \text{Eq. (9)}$$

2. Image analysis

Just as we did with a 1D signal, one can compute the fft, investigate the distribution of amplitudes and compute the power.

```
% import with imread
im=double(imread('Roberto.jpg'));

% note the last dimention is for the RGB colour channels / here we don't
% care so we average
im = mean(im,3);

% make the image square either by zero padding or truncate
N = min(size(im));
index = (max(size(im)) - N) / 2;
im = im((1+index):size(im,1)-index,:);

% check the image ;
figure('Name','Roberto'); imagesc(im); title('Original image','Fontsize',14)
xlabel('Pixel number','Fontsize',12); ylabel('Pixel number','Fontsize',12);
colormap('gray');
```

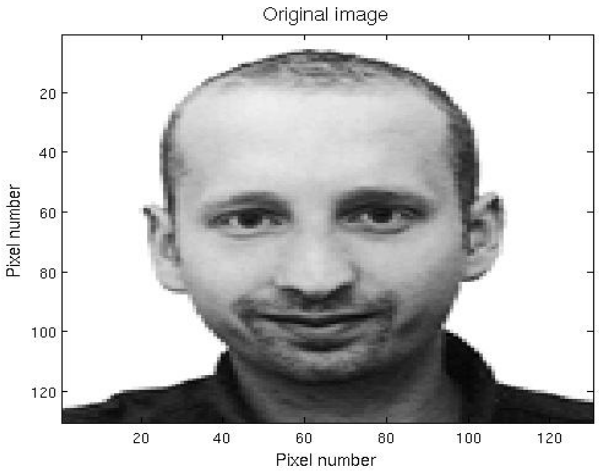

Figure 11: original picture of Roberto

The **fft2** function performs the transform. In the resulting matrix the 0 frequency (DC component) is located at the top left corner and frequencies increases in each column/rows. A more traditional representation is to have 0 at the centre of the matrix and frequencies to increase toward the borders. This is perform using the **fftshift** function.

```
% compute the fft using the ff2 function + use the fftshift function to shift
% the 0 frequency component (DC) from top left corner to the center
imf=fftshift(fft2(im));

% just as for 1D signal the frequencies are half the length of the signal
figure('Name','2D Fourier transform of Roberto')
freq =-N/2:N/2-1; subplot(2,2,1);
imagesc(freq,freq,log(abs(imf)));
title('FFT','Fontsize',14)
xlabel('Freqencies','Fontsize',12); ylabel('Frequencies','Fontsize',12);

% To obtain a finer sampling of the Fourier transform,
% add zero padding to the image (more pixels = more frequencies)
% when computing its DFT. The zero padding and DFT computation
% can be performed in a single step
imf2=fftshift(fft2(im,256,256)); % zero-pads im to be 256-by-256
freq =-256/2:256/2-1; subplot(2,2,2);
imagesc(freq,freq,log(abs(imf2)));
title('FFT at higher resolution','Fontsize',14)
xlabel('Freqencies','Fontsize',12); ylabel('Frequencies','Fontsize',12);

% power spectrum (ignore the vertical and horizontal streak - its
% an artifact due to the boundaries of the image).
impf=abs(imf2).^2; subplot(2,2,3);
imagesc(freq,freq,log10(impf)); title('Power spectrum','Fontsize',14)
xlabel('Freqencies','Fontsize',12); ylabel('Frequencies','Fontsize',12);

% rotational average to get the profile of the power along the freq axis
% for a given frequency coordinate in x and y we have a power value for
% instance at freq(0,0) impf(128,128) = 1.5293e+12 ; we can express this
% coordinate [x,y] = (128,128) in polar coordinates using cart2pol giving
% theta the angle between the x axis and the vector (0,0)(128,128) and rho
% the length of this vector - once in polar coordinates, reading rho is
% like reading the frequency value in the image - theta can be used to
% investigate which frequencies are in phase

[X Y]=meshgrid(freq,freq); % get coordinates of the power spectrum image
[theta rho]=cart2pol(X,Y); % equivalent in polar coordinates

rho=round(rho);
f=zeros(256/2+1);
for r=0:256/2
    i{r+1}=find(rho==r); % for each freq return the location in the polar
                        % array ('find') of the corresponding frequency
    f(r+1)=mean(impf(i{r+1})); % average power values of all the same freq
end
```

```
% freqency spectrum
freq2=0:256/2; subplot(2,2,4);
loglog(freq2,f); title('frequency spectrum','Fontsize',14); axis tight
xlabel('Freqencies','Fontsize',12); ylabel('Power','Fontsize',12);
```

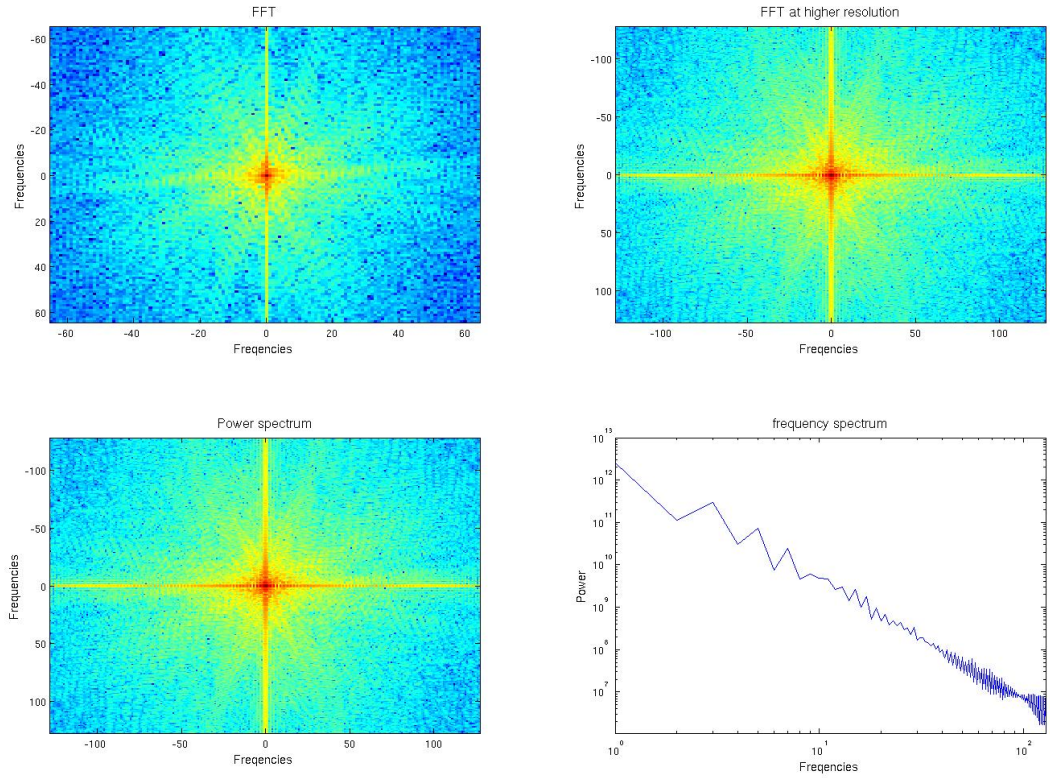

Figure 12. At the top DFT of initial picture with two different resolution (note the differences in the x-axis). At the bottom the power spectrum in 2D and averaged over angles.

3. Image decomposition: filtering

Using **ifft2** we can reconstruct back the image. To filter frequencies one can simply remove some of the frequencies in the frequency domain and reconstruct then in the spatial domain.

```
figure('Name','Inverse Fourier transforms of Roberto')

bound = round((N/10)/2);
low_freq = [N/2-bound:N/2+bound]; % from middle to image expand 1/10
b = zeros(N,N); b(low_freq,low_freq) = 1;
subplot(3,2,1); imagesc(b);
subplot(3,2,2); imagesc(real(ifft2(fftshift(fft2(im)).*b)));
title('Roberto low freq','Fontsize',14);

middle_freq = [N/2-5*bound:N/2+5*bound];
b2 = zeros(N,N); b2(middle_freq,middle_freq) = 1; b2 = b2 - b;
subplot(3,2,3); imagesc(b2);
subplot(3,2,4); imagesc(real(ifft2(fftshift(fft2(im)).*b2)));
```

```
title('Roberto mid freq','Fontsize',14);

b3 = ones(N,N) - b2 - b;
subplot(3,2,5); imagesc(b3);
subplot(3,2,6); imagesc(real(ifft2(fftshift(fft2(im)).*b3)));
title('Roberto high freq','Fontsize',14);
colormap('gray')
```

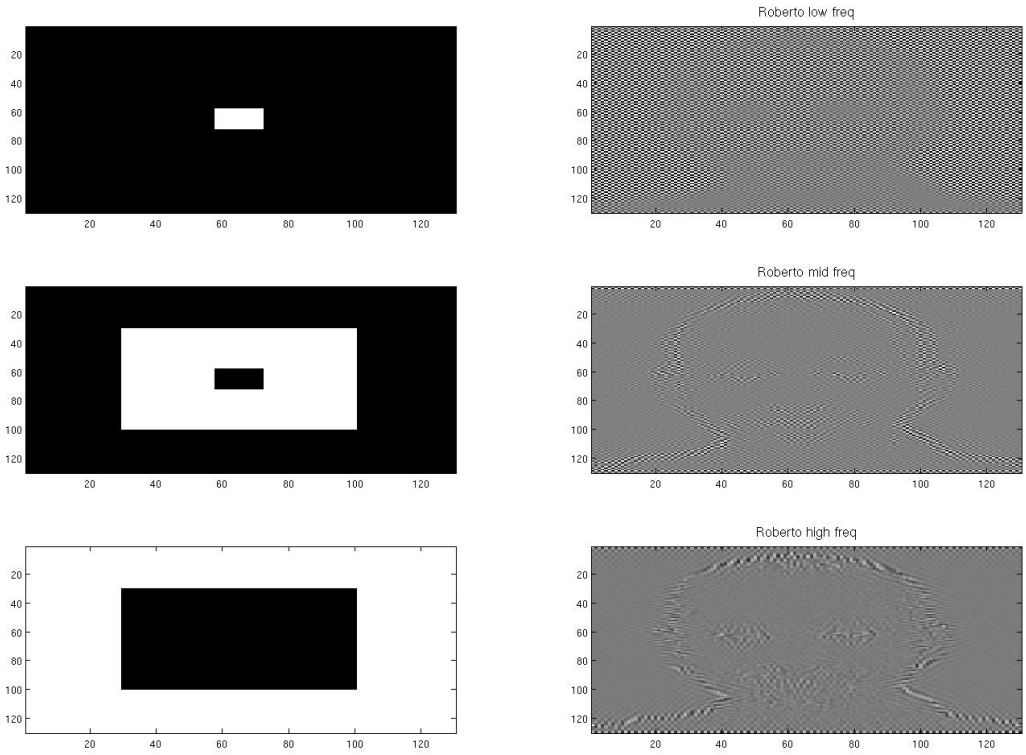

Figure 13: illustration of the distribution of frequencies in the frequencies domain.

For a good filter, since the data are organized in a circular fashion, the filter has to be circular. That is from the centre (0,0) compute the diameter of your circle and set frequencies to 0.

```
% say we want to filter up to 10Hz (low-pass)
unit = 1/(N/2);
ffilter = 10*unit;
imf=fftshift(fft2(im));

for i=1:N
    for j=1:N
        r2=(i-round(N/2))^2+(j-round(N/2))^2;
        if r2>round((N*ffilter)^2)
            imf(i,j)=0;
        end
    end
end

Ifilter=real(ifft2(fftshift(imf)));
```

```
figure('Name','frequency filtering')
subplot(1,3,1); imagesc(im); title('Original Image','Fontsize',14)
subplot(1,3,2); imagesc(Ifilter); title('10Hz low-pass','Fontsize',14)
subplot(1,3,3); imagesc(im-Ifilter); title([num2str(N/2-10) 'Hz High-
pass'],'Fontsize',14);
colormap('gray')
```

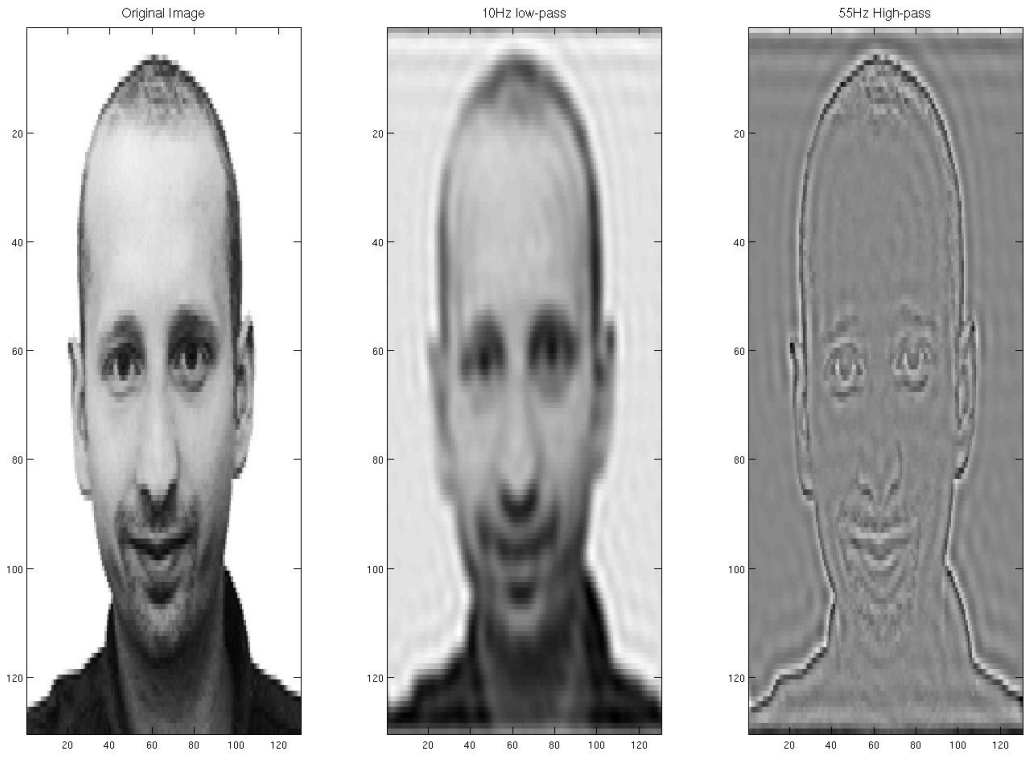

Figure 14. Frequency filtering

Another way to filter the image is to work in the spatial domain using a convolution. However, complex convolution are usually computed in the Fourier domain as a convolution in the time or spatial domain is equal to the multiplication in the frequency domain which is faster. Consider the following simple example as an empirical demonstration where the convolution **conv2** gives the same result as the multiplication in the Fourier domain

```
% 1. Create two matrices.
A = magic(3);
B = ones(3);
Conv = conv2(A,B);

% 2. Zero-pad A and B so that they are at least (M+P-1)-by-(N+Q-1).
A(8,8) = 0;
B(8,8) = 0;

% 3. Compute the two-dimensional DFT of A and B using fft2,
% multiply the two DFTs together, and compute the inverse
% two-dimensional DFT of the result using ifft2
```

```
DFT = ifft2(fft2(A).*fft2(B));

% 4. Extract the nonzero portion of the result and remove the imaginary
% part caused by roundoff error.
DFT = DFT(1:5,1:5);
DFT = real(DFT)
```

Matlab offers many way to built filters – I here only illustrate a simple smoothing = low-pass filter. The function fspecial offers the standard filters (averaging, gaussian, laplace, etc).

```
%% filtering using convolution
filter = ones(5,5);
filtered_im = (convn(im,filter))./numel(filter);
padding = (size(filtered_im)-N)/2; N2 = size(filtered_im,1);
filtered_im = filtered_im(padding:(N2-padding-1),padding:(N2-padding-1));
figure;
subplot(2,2,1); imagesc(uint8(filtered_im)); title('Convolution','Fontsize',14)
subplot(2,2,2); imagesc(uint8(im-filtered_im)+127);

% the dedicated function is
h = fspecial('average', [5 5]);
Y = filter2(h,im);
subplot(2,2,3); imagesc(Y); title('Filter','Fontsize',14)
subplot(2,2,4); imagesc(im-Y); colormap('gray')
```

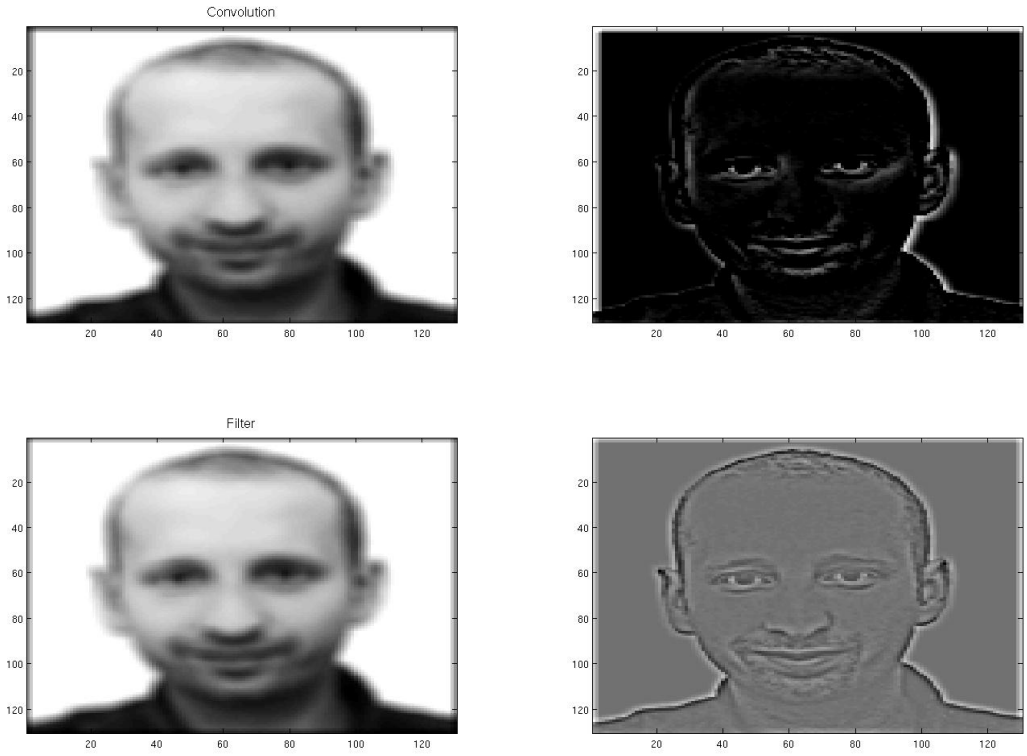

Figure 15: filtering in the spacial domain via convolution

4. Image randomization

To finish this tutorial, we will explore the effect of changing the phase of an image. Because the amplitude spectrum doesn't change, the energy is the same – that is one create a scrambled image which has the same first order statistical properties has the original one.

```
clear all
im=mean(double(imread('Roberto.jpg')),3);
N = min(size(im));
index = (max(size(im)) - N) / 2;
im = im((1+index):size(im,1)-index,:);
imf=fftshift(fft2(im));
freq =-N/2:N/2-1;
impf=abs(imf).^2;
[X Y]=meshgrid(freq,freq);
[theta rho]=cart2pol(X,Y);
rho=round(rho);
f=zeros(N/2+1);
for r=0:N/2
    i{r+1}=find(rho==r);
    f(r+1)=mean(impf(i{r+1}));
end
figure;
subplot(2,4,1); imagesc(im); title('original','Fontsize',14);
subplot(2,4,2); imagesc(angle(imf)); title ('Phase','Fontsize',14);
subplot(2,4,3); hist(im(:));
mytitle = sprintf('mean %s, std %s \n skewness %s kurtosis %s', mean(im(:)),
std(im(:)), skewness(im(:)), kurtosis(im(:)));
title(mytitle)
subplot(2,4,4); freq2=0:N/2; loglog(freq2,f); title('frequency
spectrum','Fontsize',14); axis tight


%generate random phase structure
RandomPhase = angle(fft2(rand(N, N)));
ImFourier = fft2(im);
Amp = abs(ImFourier);
Phase = angle(ImFourier);
Phase = Phase + RandomPhase;
ImScrambled = ifft2(Amp.*exp(sqrt(-1)*(Phase)));
ImScrambled = real(ImScrambled);

N = min(size(ImScrambled));
index = (max(size(ImScrambled)) - N) / 2;
im = ImScrambled((1+index):size(ImScrambled,1)-index,:);,
imf=fftshift(fft2(im));
freq =-N/2:N/2-1;
impf=abs(imf).^2;
[X Y]=meshgrid(freq,freq);
[theta rho]=cart2pol(X,Y);
rho=round(rho);
f=zeros(N/2+1);
for r=0:N/2
    i{r+1}=find(rho==r);
```

```
    f(r+1)=mean(impf(i{r+1}));
end
subplot(2,4,5); imagesc(ImScrambled); title('scrambled','Fontsize',14);
subplot(2,4,6); imagesc(angle(imf)); title ('Phase','Fontsize',14);
subplot(2,4,7); hist(ImScrambled(:));
mytitle = sprintf('mean %s, std %s \n skewness %s kurtosis %s', mean(im(:)),
std(im(:)), skewness(im(:)), kurtosis(im(:)));
title(mytitle)
subplot(2,4,8); freq2=0:N/2; loglog(freq2,f); title('frequency
spectrum','Fontsize',14); axis tight
imwrite(ImScrambled,'Roberto_scrambled.jpg','jpg');
```

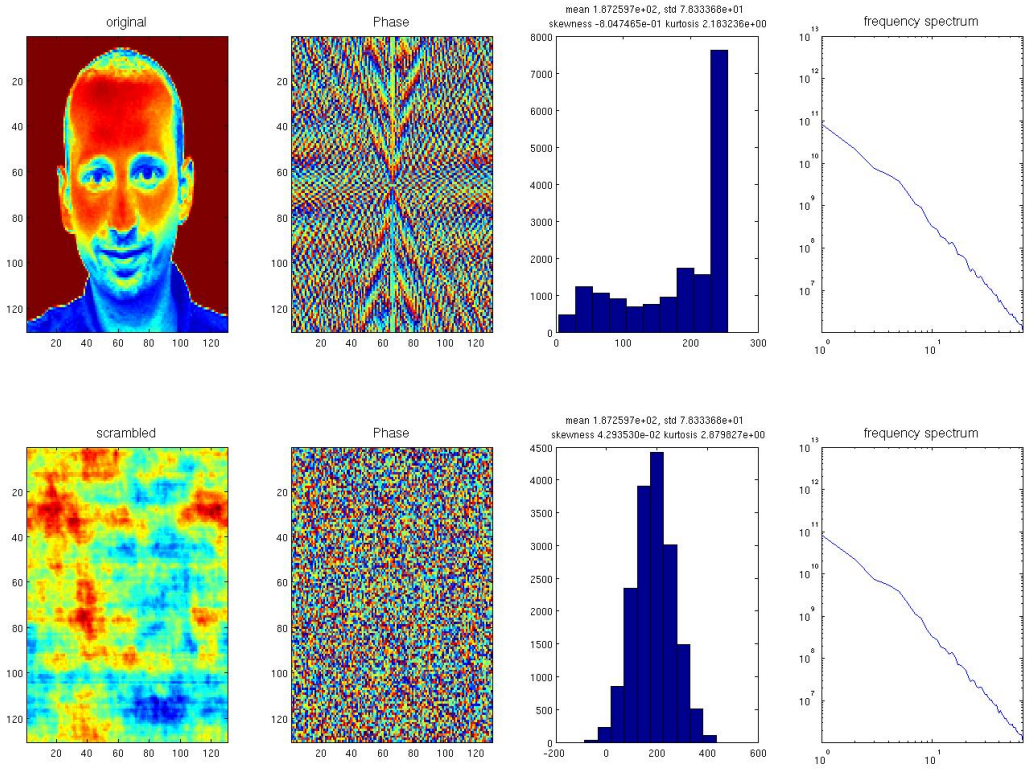

Figure 16: Original and phase scrambled image.