CM3106: Multimedia
Tutorial/Lab Class 1 (Week 2)

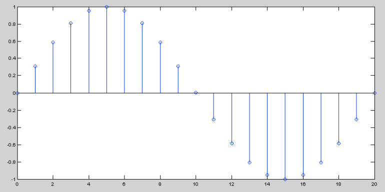
Prof David Marshall
dave.marshall@cs.cardiff.ac.uk

School of Computer Science & Informatics
Cardiff University, UK

All Lab Materials available at:
http://users.cs.cf.ac.uk/Dave.Marshall/Multimedia/PDF/tutorial.html

All Lab class support files available as a zip download

The Sine Wave and Sound



The general form of the sine wave we shall use (quite a lot of) is as follows:

$$y = A.sin(2\pi.n.F_w/F_s)$$

where:

A is the amplitude of the wave,
F_w is the frequency of the wave,
F_s is the sample frequency,
n is the sample index.

MATLAB function: `sin()` used — works in radians

MATLAB Sine Wave Radian Frequency Period

Basic 1 period Simple Sine wave — **1 period is 2π radians**

Basic 1 period Simple Sine wave

```
% Basic 1 period Simple Sine wave

i=0:0.2:2*pi;
y = sin(i);
figure(1)
plot(y);

% use stem(y) as alternative plot as in lecture notes
%  to  see sample values

title('Simple 1 Period Sine Wave');
```

MATLAB Sine Wave Amplitude

Sine Wave Amplitude is -1 to +1.

To change amplitude multiply by some gain (amp):

Sine Wave Amplitude Amplification

```
% Now Change amplitude

amp = 2.0;

y = amp*sin(i);

figure(2)
plot(y);
title('Simple 1 Period Sine Wave Modified Amplitude');
```

MATLAB Sine Wave Frequency

Sine Wave Change Frequency

```
% Natural frequency is 2*pi radians
% If sample rate is F_s HZ then 1 HZ is 2*pi/F_s
% If wave frequency is F_w then freequency is F_w* (2*pi/F_s)
% set n samples steps up to sum duration nsec*F_s where
% nsec is the duration in seconds
% So we get y = amp*sin(2*pi*n*F_w/F_s);

F_s = 11025;
F_w = 440;
nsec = 2;
dur= nsec*F_s;

n = 0:dur;

y = amp*sin(2*pi*n*F_w/F_s);

figure(3)
plot(y(1:500));
title('N second Duration Sine Wave');
```

MATLAB Sine Wave Plot of *n* cycles

Plotting of *n* cycles of a Sine Wave

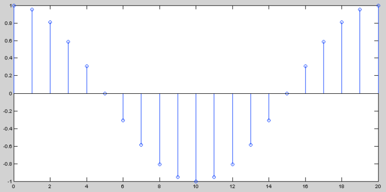
```
% To plot n cycles of a waveform

ncyc = 2;

n=0:floor(ncyc*F_s/F_w);

y = amp*sin(2*pi*n*F_w/F_s);

figure(4)
plot(y);
title('N Cycle Duration Sine Wave');
```

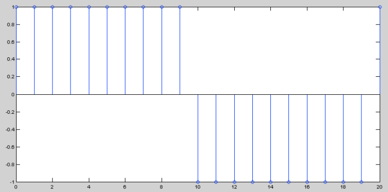
MATLAB Square and Sawtooth Waveforms

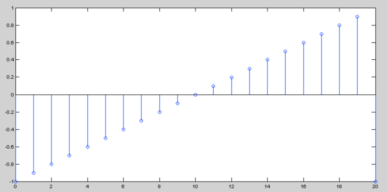
MATLAB Square and Sawtooth Waveforms

```
% Square and Sawtooth Waveforms created using Radians

ysq = amp*square(2*pi*n*F_w/F_s);
ysaw = amp*sawtooth(2*pi*n*F_w/F_s);

figure(6);
hold on
plot(ysq,'b');
plot(ysaw,'r');
title('Square (Blue)/Sawtooth (Red) Waveform Plots');
hold off;
```

MATLAB functions `cos()` (cosine), `square()` and `sawtooth()` similar.







Filtering with IIR/FIR

We have **two filter banks** defined by vectors: $A = \{a_k\}$, $B = \{b_k\}$.

These can be applied in a *sample-by-sample* algorithm:

- MATLAB provides a generic `filter(B,A,X)` function which filters the data in vector `X` with the filter described by vectors `A` and `B` to create the filtered data `Y`.
  The filter is of the standard difference equation form:

$$
\begin{aligned}
a(1) * y(n) \quad = \quad & b(1) * x(n) + b(2) * x(n-1) + \ldots + b(nb+1) * x(n-nb) \\
& -a(2) * y(n-1) - \ldots - a(na+1) * y(n-na)
\end{aligned}
$$

- If a(1) is **not equal** to **1**, filter **normalizes** the filter coefficients by a(1). If **a(1) equals 0**, `filter()` **returns** an **error**

Creating Filters

How do I create Filter banks `A` and `B`

- Filter banks can be created manually — Hand Created: **See next slide** and **Equalisation** example later in slides
- MATLAB can provide some predefined filters — **a few slides on, see lab classes**
  - Many standard filters provided by MATLAB
- See also `help filter`, online MATLAB `docs` and lab classes.

MATLAB filters

Matlab `filter()` function implements an IIR/FIR hybrid filter.

Type `help filter`:

```
FILTER One-dimensional digital filter.
    Y = FILTER(B,A,X) filters the data in vector X with the
    filter described by vectors A and B to create the filtered
    data Y.  The filter is a "Direct Form II Transposed"
    implementation of the standard difference equation:

    a(1)*y(n) = b(1)*x(n) + b(2)*x(n-1) + ... + b(nb+1)*x(n-nb)
                          - a(2)*y(n-1) - ... - a(na+1)*y(n-na)

    If a(1) is not equal to 1, FILTER normalizes the filter
    coefficients by a(1).

    FILTER always operates along the first non-singleton dimension,
    namely dimension 1 for column vectors and non-trivial matrices,
    and dimension 2 for row vectors.
```

Filtering with IIR/FIR: Simple Example

The MATLAB file IIRdemo.m sets up the filter banks as follows:

IIRdemo.m

```
fg=4000;
fa=48000;
k=tan(pi*fg/fa);

b(1)=1/(1+sqrt(2)*k+k^2);
b(2)=-2/(1+sqrt(2)*k+k^2);
b(3)=1/(1+sqrt(2)*k+k^2);
a(1)=1;
a(2)=2*(k^2-1)/(1+sqrt(2)*k+k^2);
a(3)=(1-sqrt(2)*k+k^2)/(1+sqrt(2)*k+k^2);
```

Using MATLAB to make filters for `filter()` (1)

MATLAB provides a few built-in functions to create ready made filter parameter$A$ and B:

Some common MATLAB Filter Bank Creation Functions

E.g: `butter`, `buttord`, `besself`, `cheby1`, `cheby2`, `ellip`.

See `help` or `doc` appropriate function.

Using MATLAB to make filters for `filter()`(2)

For our purposes the **Butterworth** filter will create suitable filters, :

help butter

```
BUTTER Butterworth digital and analog filter design.
    [B,A] = BUTTER(N,Wn) designs an Nth order lowpass digital
    Butterworth filter and returns the filter coefficients in
     length N+1 vectors B (numerator) and A (denominator).
    The coefficients are listed in descending powers of z.
    The cutoff frequency  Wn must be 0.0 < Wn < 1.0, with 1.0
    corresponding to half the sample rate.

    If Wn is a two-element vector, Wn = [W1 W2], BUTTER returns
    an order 2N bandpass filter with passband  W1 < W < W2.
    [B,A] = BUTTER(N,Wn,'high') designs a highpass filter.
    [B,A] = BUTTER(N,Wn,'low') designs a lowpass filter.
    [B,A] = BUTTER(N,Wn,'stop') is a bandstop filter
    if Wn = [W1 W2].
```

Fourier Transform in MATLAB

`fft()` and `fft2()`

MATLAB provides functions for 1D and 2D **Discrete Fourier Transforms** (**DFT**):

fft(X)
: is the 1D discrete Fourier transform (DFT) of **vector** X. For **matrices**, the FFT operation is applied to **each column** — **NOT** a 2D DFT transform.

fft2(X)
: returns the 2D Fourier transform of matrix X. If X is a vector, the result will have the same orientation.

fftn(X)
: returns the N-D discrete Fourier transform of the **N-D array X**.

Inverse DFT
: **ifft()**, **ifft2()**, **ifftn()** perform the **inverse** DFT.

See appropriate MATLAB **help/doc** pages for **full details**.

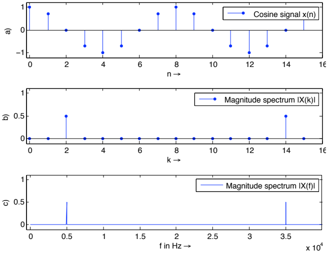
Plenty of examples to Follow.

See also: **MALTAB Docs Image Processing** → **User's Guide** → **Transforms** → **Fourier Transform**

Visualising the Fourier Transform

Having computed a DFT it might be useful to visualise its result:

- It's useful to visualise the Fourier Transform
- Standard tools
- Easily plotted in MATLAB



The Magnitude Spectrum of Fourier Transform

Recall that the Fourier Transform of our **real** audio/image data is always **complex**

- **Phasors**: This is how we encode the **phase** of the underlying signal's **Fourier Components**.

How can we visualise a complex data array?

Back to Complex Numbers:

Magnitude spectrum **Compute the absolute value of the complex data**:

$$|F(k)| = \sqrt{F_R^2(k) + F_I^2(k)} \textbf{ for } k = 0, 1, \ldots, N-1$$

where $F_R(k)$ is the **real** part and $F_I(k)$ is the **imaginary** part of the N sampled Fourier Transform, $F(k)$.

Recall MATLAB: `Sp = abs(fft(X,N))/N;`
(**Normalised form**)

The Phase Spectrum of Fourier Transform

The Phase Spectrum

Phase Spectrum

The Fourier Transform also represent phase, the **phase spectrum** is given by:

$$\varphi = \arctan \frac{F_I(k)}{F_R(k)} \textbf{ for } k = 0, 1, \ldots, N-1$$

Recall MATLAB: `phi = angle(fft(X,N))`

Relating a Sample Point to a Frequency Point

When **plotting graphs** of *Fourier Spectra* and doing other DFT processing we may wish to **plot** the x-axis in **Hz** (**Frequency**) rather than **sample point** number $k = 0, 1, \ldots, N-1$

There is a **simple relation** between the two:

- The sample points go in steps $k = 0, 1, \ldots, N-1$
- For a given sample point k the frequency relating to this is given by:

$$f_k = k\frac{f_s}{N}$$

  where f_s is the *sampling frequency* and N the **number** of samples.
- Thus we have **equidistant frequency steps** of $\frac{f_s}{N}$ ranging from 0 Hz to $\frac{N-1}{N}f_s$ Hz

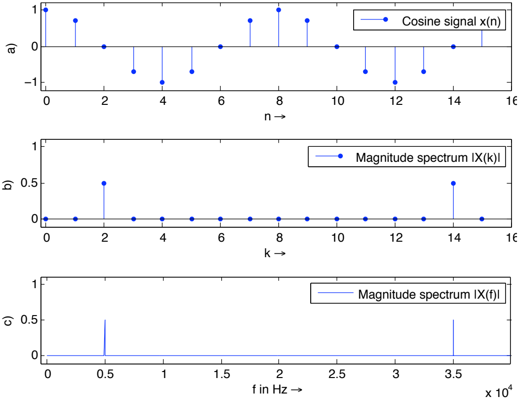
MATLAB Fourier Frequency Spectra Example

fourierspectraeg.m

```
N=16;
x=cos(2*pi*2*(0:1:N-1)/N)';

figure(1)
subplot(3,1,1);
stem(0:N-1,x,'.');
axis([-0.2 N -1.2 1.2]);
legend('Cosine signal x(n)');
ylabel('a)');
xlabel('n \rightarrow');

X=abs(fft(x,N))/N;
subplot(3,1,2);stem(0:N-1,X,'.');
axis([-0.2 N -0.1 1.1]);
legend('Magnitude spectrum |X(k)|');
ylabel('b)');
xlabel('k \rightarrow')

N=1024;
x=cos(2*pi*(2*1024/16)*(0:1:N-1)/N)';
```

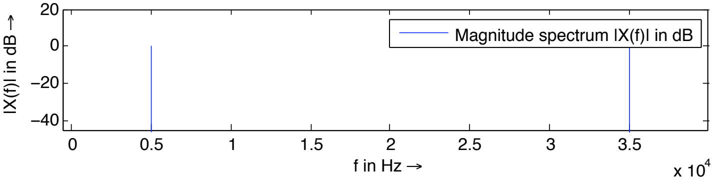
```
FS=40000;
f=((0:N-1)/N)*FS;
X =abs(fft(x,N))/N;
subplot(3,1,3);plot(f,X);
axis([-0.2*44100/16 max(f) -0.1 1.1]);
legend('Magnitude spectrum |X(f)|');
ylabel('c)');
xlabel('f in Hz \rightarrow')

figure(2)
subplot(3,1,1);
plot(f,20*log10(X./(0.5)));
axis([-0.2*44100/16 max(f) ...
-45 20]);
legend('Magnitude spectrum |X(f)| ...
in dB');
ylabel('|X(f)| in dB \rightarrow');
xlabel('f in Hz \rightarrow')
```

fourierspectraeg.m produces the following:



Magnitude Spectrum in dB

Note: It is common to plot both spectra magnitude (also frequency ranges not show here) on a dB/log scale:
(Last Plot in fourierspectraeg.m)



Time-Frequency Representation: Spectrogram

Spectrogram

It is often **useful** to look at the **frequency distribution** over a **short-time**:

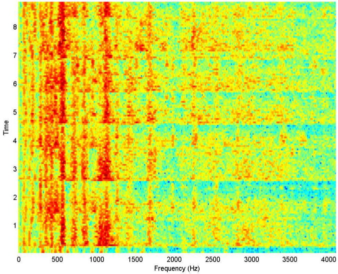
- Split signal into N segments
- Do a **windowed Fourier Transform** — **Short-Time Fourier Transform** (**STFT**)
  - Window needed to reduce *leakage* effect of doing a shorter sample SFFT.
  - Apply a **Blackman**, **Hamming** or **Hanning** Window
- MATLAB function does the job: `Spectrogram` — see `help spectrogram`
- See also MATLAB's `specgramdemo`

MATLAB spectrogram Example

spectrogrameg.m

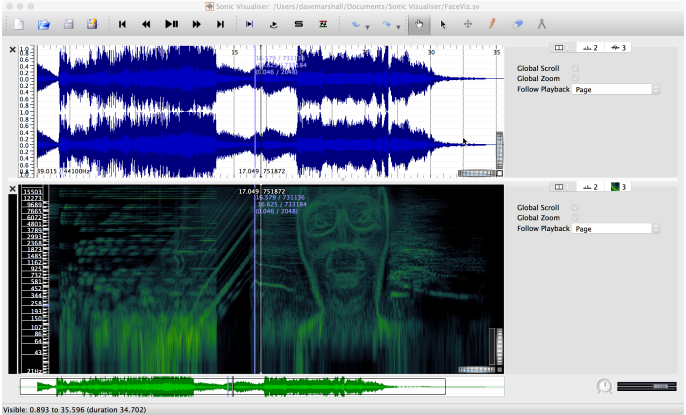
```
load('handel')
[N M] = size(y);
figure(1)
spectrogram(y,512,20,1024,Fs);
```

Produces the following:



Aphex Twin Spectrogram

Aphex Twin famously[1] embedded images in the spectrogram of a few tracks on his Windowlicker EP. His face on Track 2 "Formula" or "Equation" (Full title: $\Delta M_{i-1} = -\alpha \sum_{n=1}^{N} D_i[n][\sum_{\sigma \in C[i]} F_{ji}[n-1] + Fext_i[n-1]]$!!:

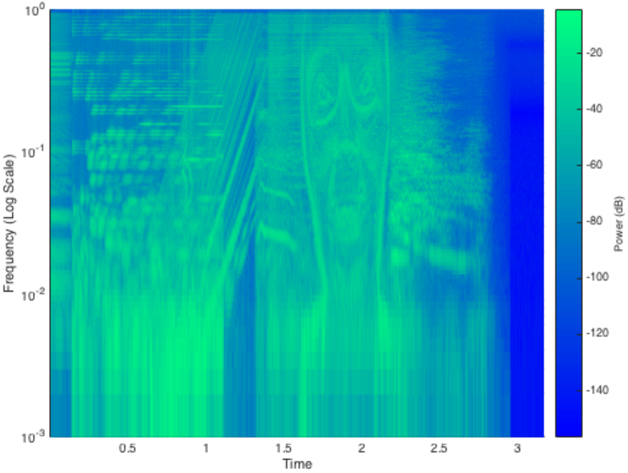


[1]See here for web link to other examples of embedded image Spectrograms

Matlab Code to show the Aphex Twin Spectrogram

Previous slide use the free and excellent **Sonic Visualiser**

We of course know how to display the image in MATLAB:



Matlab Code to show the Aphex Twin Spectrogram

Aphex_Spectrogram.m

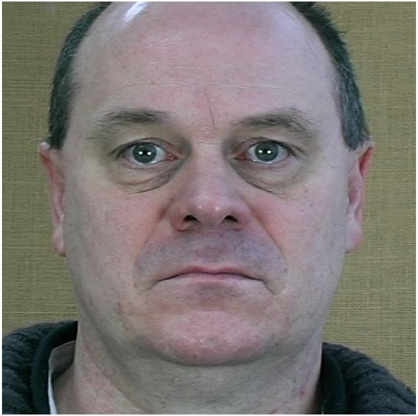
```
aphex = audioread('FormulaSnippet.wav');

mono = (aphex(:,1) + aphex(:,2))/2;

spectrogram(mono,1024,120,2048,'power','yaxis');
set(gca,'YScale','log');
colormap('winter');
xlabel('Time')
ylabel('Frequency (Log Scale)')
```

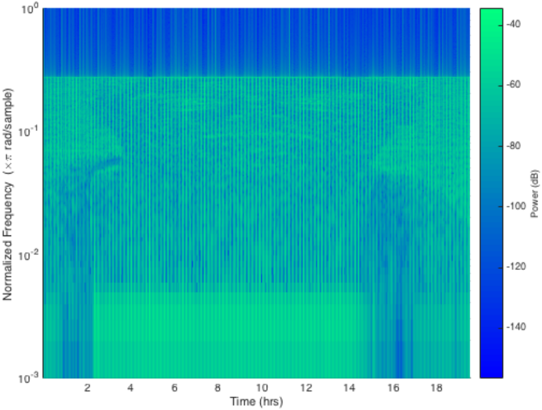
Note: we change the display of the spectrogram to a **log scale**, which looks better.

Audio clip here: FormulaSnippet.wav

So what does my face sound like?

Let’s embed my face in spectrogram:





It sounds like this:

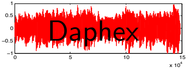


Image to Sound Conversion

Daphex.m

```
figure(1);
imshow(imread('Dave_Frame0001.jpg'));

dave_im2snd = im2sound('Dave_Frame0001', 'jpg', 44100, 40,6000,0.00002, 10);

sound(dave_im2snd,44100);

figure(2);

spectrogram(dave_im2snd,1024,120,2048,'power','yaxis');
set(gca,'YScale','log');

colormap('winter');

shg;
```

Image used here: Dave_Frame0001.jpg

Image to Sound Conversion[2]

im2sound.m (Usage)

```
function [final_sound] = im2sound(filename, ext, f_sample, f_low, ...
    f_high, amp_mod, sample_t)

%INPUTS:
%'filename' - Name of the image to be encoded (not including extension
%ext' - Extension of the image (not including "." at the beginning).
%'f_sample' - Sampling frequency (Hz)
%'f_low' - Lowest frequency (Hz) (e.g. 40)
%'f_high' - Highest frequency (Hz) (e.g. 6000)
%'amp_mod' - Multiplication factor for the amplitude.  Decrease until
%image is clear. Too high and the waveform clips.  Too low and the image
%is very dark (e.g. 0.00002)
%'sample_t' - Duration of the sample in seconds.  Longer samples have
%better quality (e.g. 10)

%OUTPUTS:
%'final_sound' - the final sound containing the image.  This is
%automatically saved to a .wav file with the original image filename
```

[2]Original Code from MATLAB Central

Image to Sound Conversion

im2sound.m (Code)

```
function [final_sound] = im2sound(filename, ext, f_sample, f_low, ...
    f_high, amp_mod, sample_t)

........

%INITIALISING VARIABLES:
%The waveform at each time point.  This is reset at the beginning of each
%time point
temp_sound = 0;
%The final waveform
final_sound = 0;


%MAIN BODY
%Loading the sample image and calculating the image size
raw_im = imread(strcat(filename,'.',ext));
size_raw_im = size(raw_im);

%Making a frequency table for the height of the image.  Each row of the
%image is assigned a particular frequency from the corresponding row of
%this table.  The frequencies are linearly distributed between the highest and
%lowest user-definied frequencies.  "f_step" is the increment between each
%adjacent frequency
f_step = (f_high - f_low)/size_raw_im(1,1);
f_table = (f_high:-f_step:f_low);
```

Image to Sound Conversion

im2sound.m (Code)

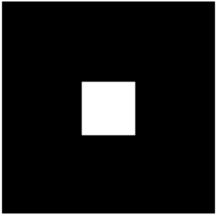
```
%The final sound will dwell on each column of the image for a specific
%time.  This time is defined by "t_start" and "t_end".  It depends on how
%long the user determined the sound-clip should be and how wide (how many
%columns) the image is.
t_step = (sample_t/size_raw_im(1,2));

%Initial values for the start and end times.  These will be increased at
%the end of each loop iteration (when the script moves onto the next column
%of the image).
t_start = 0;
t_end = t_step;

%The loop which generates the sound file.  At each iteration it generates a
%segment of the final sound file, which is temporarily saved to
%"temp_sound".  This segment is built up of frequencies from that
%particular column of the image.
for j = 1:size_raw_im(1,2)
    %Initialising the variable (the sound for each frequency (row) is added
    %to the existing sound)
    temp_sound = 0;

    %Setting the time in matrix format
    t = t_start:1/f_sample:(t_end);
```

Image to Sound Conversion

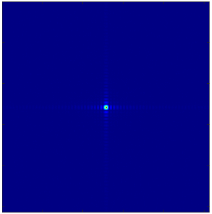
im2sound.m (Code)

```
    %For each iteration of this loop, the script goes down the current
    %column of the image and generates a waveform of the frequency
    %specified in "f_table".  The amplitude of the waveform is determined
    %by the pixel intensity.  This generated waveform is added to all the
    %previously generated waveforms in that particular column
    for i = size_raw_im(1,1):-1:1
        temp_sound = temp_sound+ sin(2*pi*t*f_table(i))*...
            double(raw_im(i,j))*amp_mod;

    end

    %At the end of each column the segment of sound generated is added to
    %the end of the existing sound file ("final_sound").
    final_sound = cat(2,final_sound,temp_sound);

    %The temporary sound is cleared ready for the start of the next column
    clear temp_sound

    %Moving to the next time frame
    t_start = t_start + t_step;
    t_end = t_end + t_step;

end

%This saves "final_sound" to the '.wav' file of the same name as the input
%file
audiowrite(strcat(filename, '.wav'), final_sound, f_sample);
```

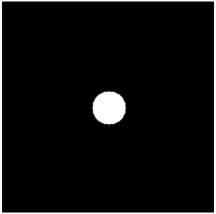
Ideal Low Pass Filter Example 1



(a) Input Image



(b) Image Spectra



(c) Ideal Low Pass Filter

(d) Filtered Image

Ideal Low-Pass Filter Example 1 MATLAB Code

lowpass.m:

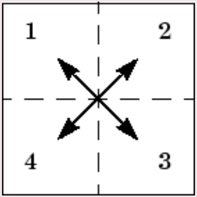
```
% Create a white box on a
% black background image
M = 256; N = 256;
image = zeros(M,N)
box = ones(64,64);
%box at centre
image(97:160,97:160) = box;

% Show Image

figure(1);
imshow(image);

% compute fft and display its spectra

F=fft2(double(image));
figure(2);
imagesc((abs(fftshift(F))/(M*N)));
colormap(jet);
axis off;
```

```
% Compute Ideal Low Pass Filter
u0 = 20; % set cut off frequency

u=0:(M-1);
v=0:(N-1);
idx=find(u>M/2);
u(idx)=u(idx)-M;
idy=find(v>N/2);
v(idy)=v(idy)-N;
[V,U]=meshgrid(v,u);
D=sqrt(U.^2+V.^2);
H=double(D<=u0);

% display
figure(3);
imshow(fftshift(H));

% Apply filter and do inverse FFT
G=H.*F;
g=real(ifft2(double(G)));

% Show Result
figure(4);
imshow(g);
```

Shifting the Fourier Transform, `fftshift()`

Centring the Frequency of a Fourier Transform

- Most computations of FFT represent the frequency from 0 — $N-1$ samples (similarly in 2D, 3D *etc.*) with corresponding frequencies ordered accordingly — the 0 frequency is not really the **centre**.
- We frequently like to visualise the FFT as the **centre of the spectrum**.
- In 1D (Audio/Vector): **swaps the left and right halves of the vector**
- Similarly in 2D (Image/Matrix) we swap the first quadrant with the third and the second quadrant with the fourth:



The fftshift() MATLAB Command

help fftshift()

Y = fftshift(X) rearranges the outputs of **fft**, **fft2**, and **fftn** by **moving** the zero-frequency component to the **center of the array**.

It is useful for **visualising** a Fourier transform with the zero-frequency component in the **middle** of the spectrum.

For **vectors**, **fftshift(X)** **swaps** the **left** and **right** halves of X.

For **matrices**, **fftshift(X)** swaps the **first** quadrant with the **third** and the **second** quadrant with the **fourth**.

Butterworth Low-Pass Filter Example Code

butterworth.m:

```
% Load Image and Compute FFT as
%  in Ideal Low Pass Filter Example 1
.......
% Compute Butterworth Low Pass Filter
u0 = 20; % set cut off frequency

u=0:(M-1);
v=0:(N-1);
idx=find(u>M/2);
u(idx)=u(idx)-M;
idy=find(v>N/2);
v(idy)=v(idy)-N;
[V,U]=meshgrid(v,u);

for i = 1: M
    for j = 1:N
      %Apply a 2nd order Butterworth
      UVw = double((U(i,j)*U(i,j) + V(i,j)*V(i,j))/(u0*u0));
      H(i,j) = 1/(1 + UVw*UVw);
    end
end
% Display Filter and Filtered Image as before
```

Phasors (Recap from CM2104/CM2208 (CM2202))

General Phasor Form: $re^{i\phi}$

More generally we use $re^{i\phi}$ where:

$$re^{i\phi} = r(\cos\phi + i\sin\phi)$$

MATLAB Speaks the Phasor Language

MATLAB Complex No. Phasor Declaration

```
>> exp( i*(pi/4) )

ans =  0.7071 + 0.7071i

>> [abs(z), angle(z)]

ans =  1.0000    0.7854
```

Rotating a Phasor

Rotating a Phasor

Could not be more simpler, to rotate by an angle θ:

- multiply the phasor by the the phasor

$$e^{i*\theta}$$

So given a phasor, $re^{i\phi}$ to rotate it by an angle θ do :

$$re^{i\phi} * e^{i*\theta} = re^{i(\phi+\theta)}$$

MATLAB Example

MATLAB Phaser Rotation, phasor_rotate_eg.m

```
syms x;  % Create our symbolic variable

fcos = exp(i*0);  % A Phasor (cosine) with no phase.

% Rotate phaser by pi/4 radians  (45 degrees)
frot = fcos*exp(i*pi/4);

% convert back (check) to non-phasor way of thinking

fcos_angle = angle(fcos);  % It's zero!

frot_angle = angle(frot);  % Should be pi/4!
```

Phase Shifting via the Fourier Transform

fft_phase_eg.m

```
% Set Up
sample_rate=10000;
dt=1/sample_rate;
len=0.01;
t=0:dt:(len-dt);
f=500;
N = length(t);

% Generate signal
signal=sin(2*pi*f*t);

% Define a phase shift
phase = pi/4;
num_samp =
 round((sample_rate/f)
  *(phase/(2*pi)));

% Get the FFT of the signal
signalfft =fft(signal);

% Rotate each FFT component
k=1:length(signalfft);

% Range of Phasor phase values
w = 2*pi/N*(k-1);
spec=signalfft.*
   exp(-j*w*num_samp);

% Get the new signal
newsignal=(ifft(spec));

% Plot the signals
figure;plot(t,real(signal));
hold on;
plot(t,real(newsignal),'g');
```

Phase Shifting via the Fourier Transform

Heart of fft_phase_eg.m

```
% Rotate each FFT component
k=1:length(signalfft);

% Range of Phasor phase values
w = 2*pi/N*(k-1);
spec=signalfft.*exp(-j*w*num_samp);
```