FOURIER ANALYSIS

Reference

- http://www.gaussianwaves.com/2015/11/interpreting-fft-results-complex-dft-frequency-bins-and-fftshift/

Fourier analysis

- Any complex, 1-dimensional function can be expressed as an additive series of sinusoidal functions varying in (1) frequency, (2) amplitude and (3) phase.
- (Continuous) Fourier transform

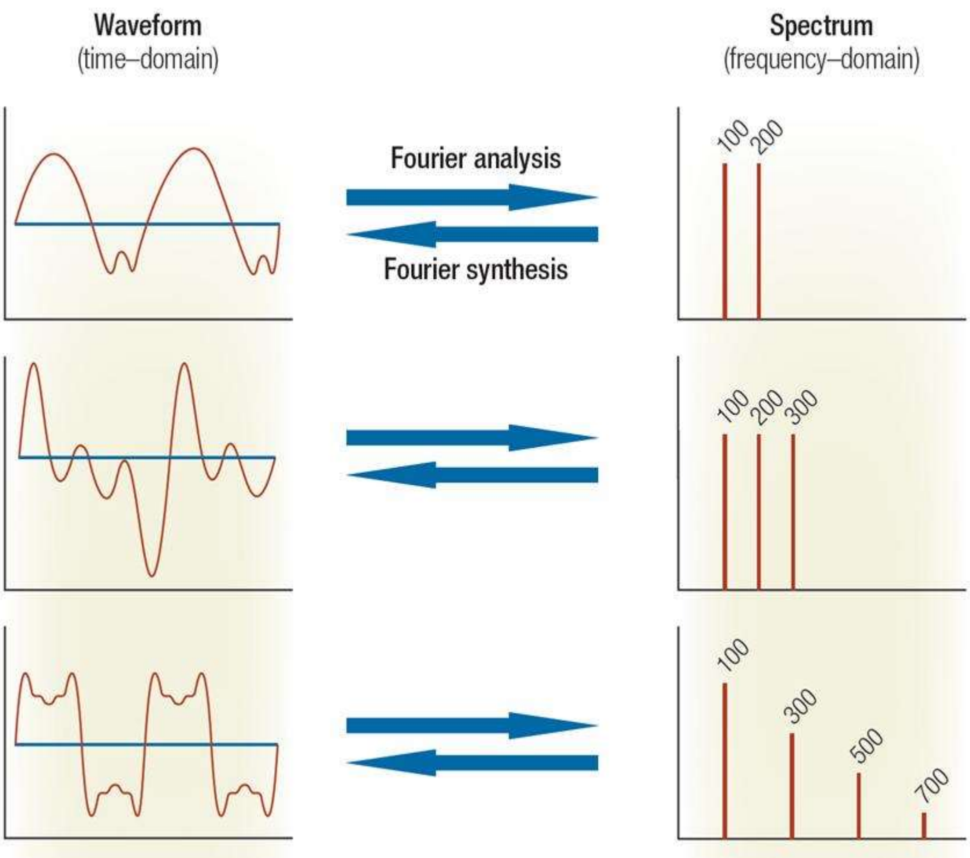
$$S(f) = \int_{-\infty}^{\infty} s(t) \cdot e^{-i2\pi ft}\, dt.$$

frequency-domain function

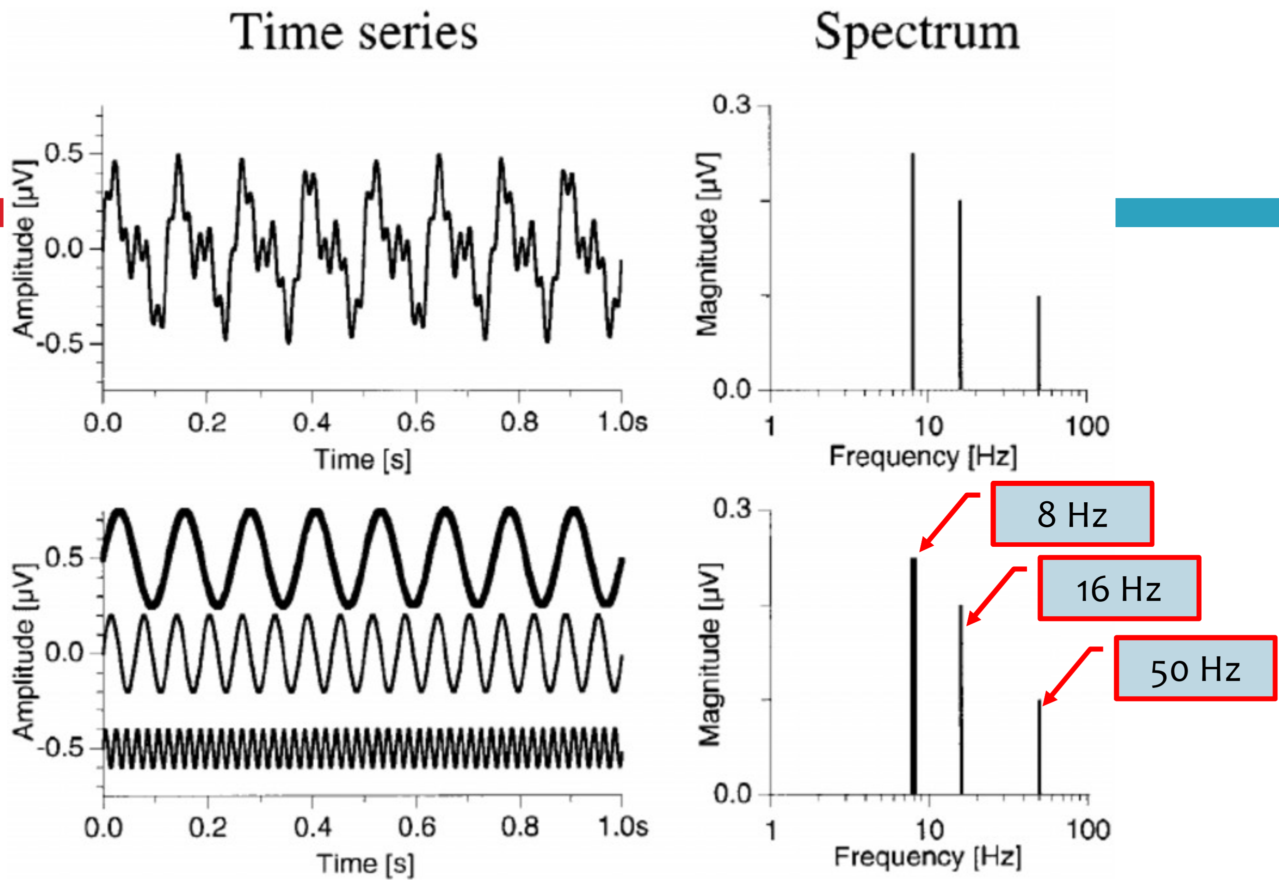
time-domain function

Euler's formula
$e^{ix} = \cos x + i \sin x$

Time domain



Frequency domain



Fourier analysis

- The top part shows a somewhat irregular waveform with both slow and fast oscillations.
- The bottom part shows the three sinusoidal waveforms which, when added together, produce the top trace.
- The lowest frequency (thick trace) contains exactly **8** periods in the recording interval (=analysis interval) of 1 s length. Thus the corresponding spectral line (right) is located at **8 Hz**.
- The spectrum further reveals the second frequency of 16 Hz and a third 50 Hz component.

Four types of Fourier Transforms

- In signal processing , a time domain signal can be continuous or discrete and it can be aperiodic or periodic.
- This gives rise to four types of Fourier transforms.

Four types of Fourier Transforms

Transform	Nature of time domain signal	Nature of frequency spectrum
Fourier Transform (FT), (a.k.a Continuous Time Fourier Transform (CTFT))	continuous, non-periodic	continuous, non-periodic
Discrete-time Fourier Transform (DTFT)	discrete, non-periodic	continuous, periodic
Fourier Series (FS)	continuous, periodic	discrete, non-periodic
Discrete Fourier Transform (DFT)	discrete, periodic	discrete, periodic

Four types of Fourier Transforms

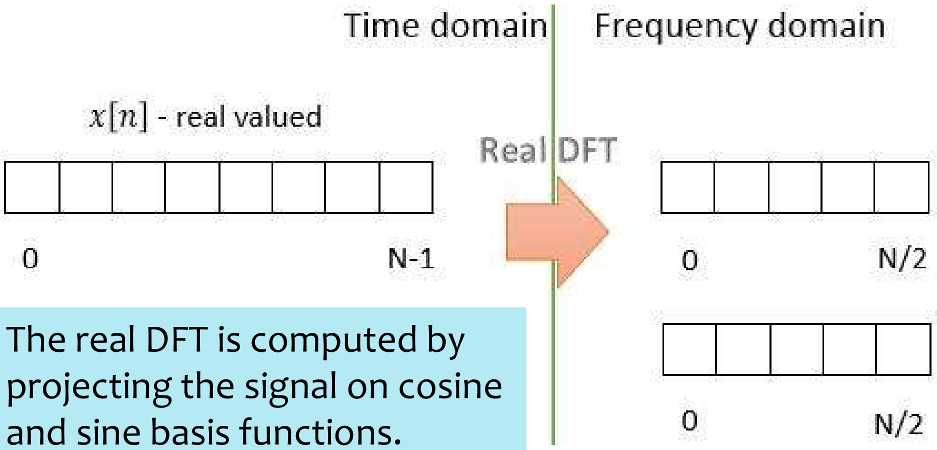
- We will limit our discussion to DFT, that is widely available as part of software packages like Matlab, Scipy(python) etc.., however we can approximate other transforms using DFT.
- The DFT can be computed efficiently in practice using a fast Fourier transform (FFT) algorithm.

Real version and Complex version

- For each of the listed transforms above, there exist a real version and complex version.
- The real version of the transform, takes in a real numbers and gives two sets of real frequency domain points
  - one set representing coefficients over cosine basis function
  - and the other set representing the coefficients over sine basis function.
- The complex version of the transforms represent positive and negative frequencies in a single array.
  - The complex versions are flexible that it can process both complex valued signals and real valued signals.

Real DFT

11



$$X_{re}[k] = \frac{2}{N}\sum_{n=0}^{N-1} x[n]cos\left(\frac{2\pi kn}{N}\right)$$

$X_{re}[k]$ – real valued
(cosine terms)

$X_{im}[k]$ – real valued
(sine terms)

$$X_{im}[k] = -\frac{2}{N}\sum_{n=0}^{N-1} x[n]sin\left(\frac{2\pi kn}{N}\right)$$

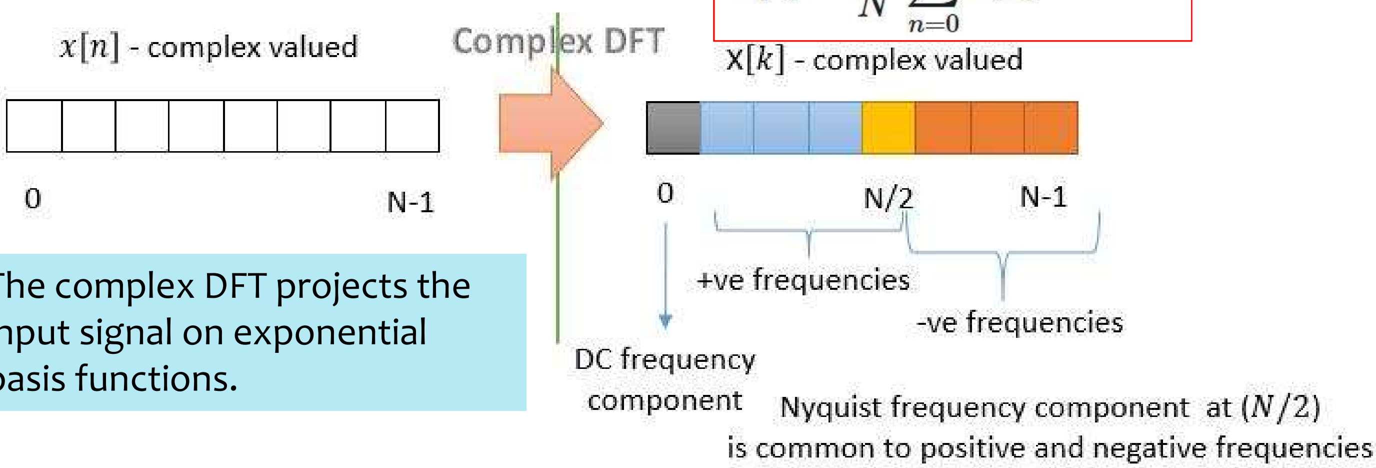
The real DFT is computed by projecting the signal on cosine and sine basis functions.

Complex DFT

12

Euler's formula
$e^{ix} = \cos x + i\sin x$

$$X[k] = \frac{1}{N}\sum_{n=0}^{N-1} x[n]e^{-j2\pi kn/N}$$



The complex DFT projects the input signal on exponential basis functions.

Complex DFT

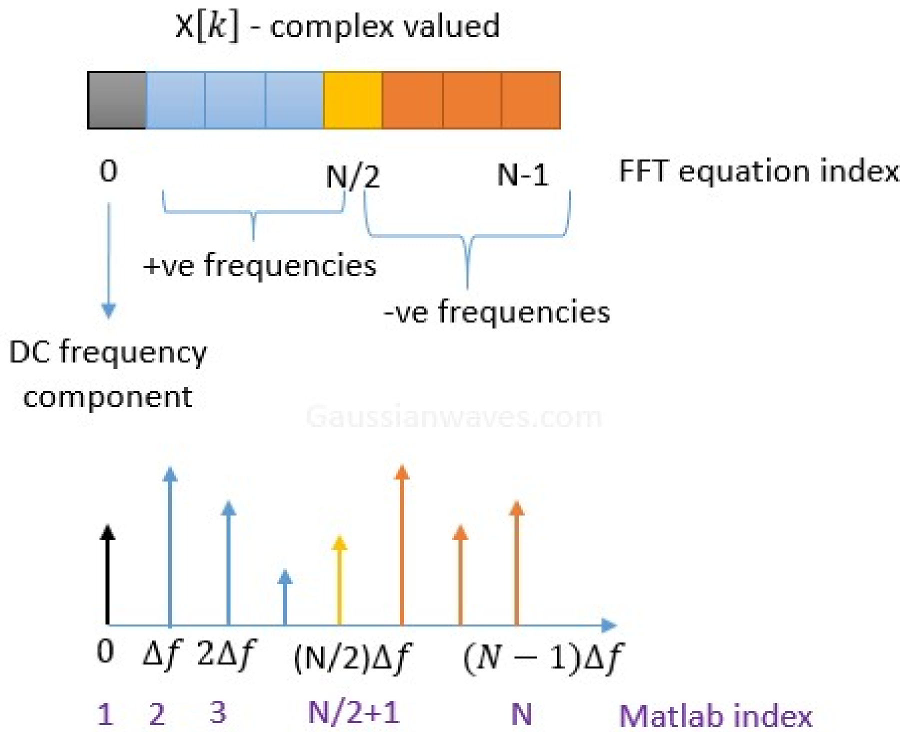
13

- The arrays values are interpreted as follows
  - X[0] represents DC frequency component
  - Next N/2 terms are positive frequency components with X[N/2] being the Nyquist frequency (which is equal to half of sampling frequency)
  - Next N/2 –1 terms are negative frequency components
    - note: negative frequency components are the phasors rotating in opposite direction, they can be optionally omitted depending on the application

Complex DFT

14

- FFT is widely available in software packages like Matlab, Scipy etc...
- FFT in Matlab/Scipy implements the complex version of DFT.



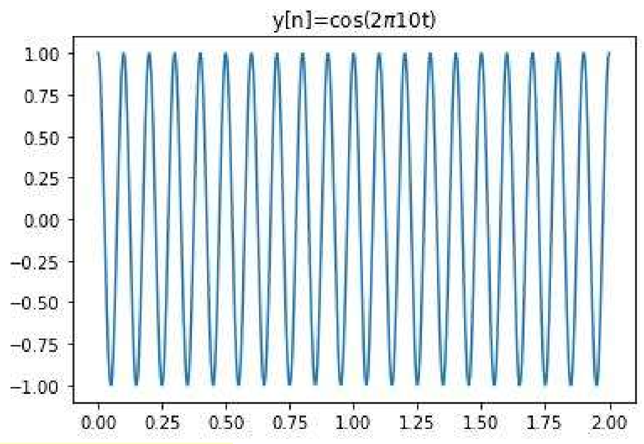
Generate a cosine signal

- Lets assume that the y[n] is the time domain cosine signal of frequency fc=10Hz that is sampled at a frequency fs=32*fc for representing it in the computer memory.

$$y[n] = cos(2\pi f_c t)$$

Generate a cosine signal


```
import matplotlib.pyplot as plt
import numpy as np
# %matplotlib inline
# frequency of the carrier
fc = 10
# sampling frequency factor=32
fs = 32 * fc
duration = 2  # 2 seconds duration
t = np.linspace(0, duration, duration*fs)
# time domain signal (real number)
y = np.cos(2*np.pi*fc*t)
plt.title(r'y[n]=cos(2$\pi$10t)')
plt.plot(t, y)
```



Interpreting the FFT results

- Compute the one-dimensional discrete Fourier Transform.

```
numpy.fft.fft(a, n=None, axis=-1, norm=None)
```

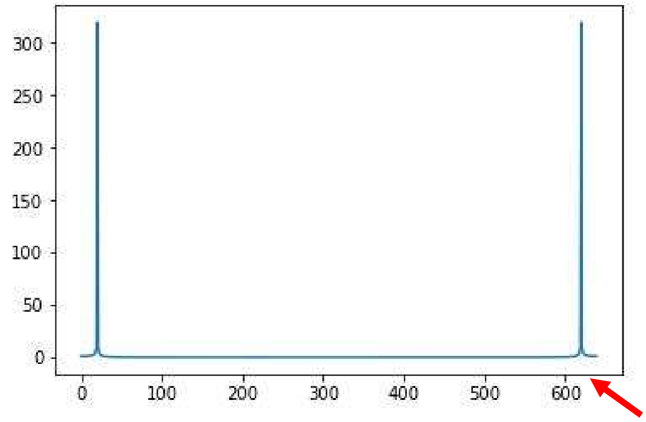
- Parameters:
  - a : array_like, Input array, can be complex.
  - n : int, optional, Length of the transformed axis of the output.
    - If n is smaller than the length of the input, the input is truncated .
    - If it is larger, the input is padded with zeros.
    - If n is not given, the length of the input along the axis specified by axis is used.
- Returns a complex ndarray

Interpreting the FFT results

- Let's consider taking a N=256 point FFT, which is the 8th power of 2.
  - FFT length is generally considered as power of 2 – this is called radix–2.
- Note:
  - In our case, the cosine wave is of 2 seconds duration and it will have 640 points
    - A 10Hz frequency wave sampled at 32 times oversampling factor will have $2 \times 32 \times 10 =$ 640 samples in 2 seconds of the record.
  - Since our input signal is periodic, we can safely use N=256 point FFT, anyways the FFT will extend the signal when computing the FFT .

Interpreting the FFT results

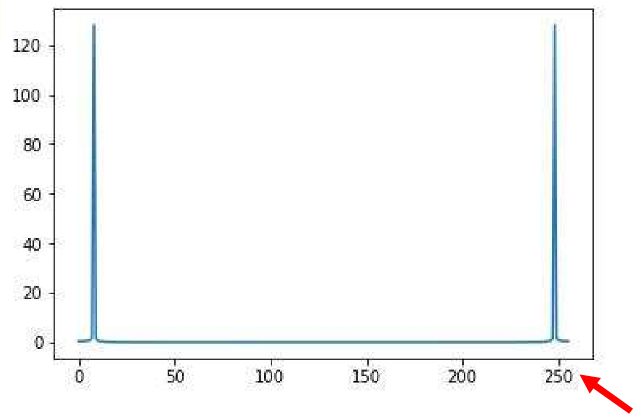

```
freqY = np.fft.fft(y)
spectrum = np.sqrt(freqY.real**2+freqY.imag**2)
plt.plot(spectrum)
```



Interpreting the FFT results


```
N=256  #FFT size
freqY = np.fft.fft(y, N)
spectrum = np.sqrt(freqY.real**2+freqY.imag**2)
plt.plot(spectrum)
```

- Note that the index for the raw FFT are integers from 0→N-1
- We need to convert the integer indices to frequencies.



Frequency axis transform

21

- **numpy.fft.fftfreq(n, d=1.0)**
  - The returned float array *f* contains the frequency bin centers in cycles per unit of the sample spacing (with zero at the start).
  - For instance, if the sample spacing is in seconds, then the frequency unit is cycles/second.

```
>>> signal = np.array([-2, 8, 6, 4, 1, 0, 3, 5], dtype=float)
>>> fourier = np.fft.fft(signal)
>>> n = signal.size
>>> timestep = 0.1
>>> freq = np.fft.fftfreq(n, d=timestep)
>>> freq
array([ 0.  ,  1.25,  2.5 ,  3.75, -5.  , -3.75, -2.5 , -1.25])
```

Frequency axis transform

22

- frequency signal fc=10Hz, sampling frequency fs=32*fc
- One second has 10*32=320 samples
- sample spacing in second = 1/320

```
In [86]: freq = np.fft.fftfreq(N, d=1/320)
In [87]: freq[8]
Out[87]: 10.0
In [89]: freq
Out[89]: array([ 0. , 1.25, 2.5 , ..., -3.75, -2.5 , -1.25])
```

Frequency axis transform

- The cosine signal has a peak at 10Hz. In addition to that, it has also a peak at 256−8=248th sample that belongs to negative frequency portion.

```
spectrum[8]
Out[62]:  128.06700088210565
spectrum[248]
Out[65]:  128.06700088210559
```

```
In [91]: freq[8]
Out[91]: 10.0
In [92]: freq[248]
Out[92]: -10.0
```

The 10Hz cosine signal will leave a peak at the 8th sample (10/1.25=8)

$$\Delta f = \frac{f_s}{N} = \frac{32 * f_c}{256} = \frac{320}{256} = 1.25Hz$$

Frequency axis transform

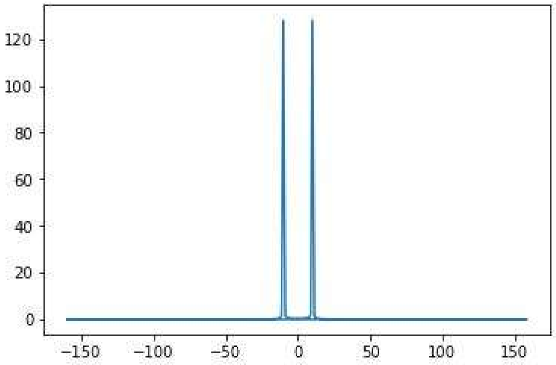
- The sample at the Nyquist frequency (fs/2) mark the boundary between the positive and negative frequencies.

```
In [89]: freq
Out[89]: array([ 0.  , 1.25, 2.5 , ..., -3.75, -2.5 , -1.25])
In [100]: nyquistIndex=int(N/2)
In [101]: freq[nyquistIndex]
Out[101]: -160.0
In [102]: freq[nyquistIndex-1]
Out[102]: 158.75
```

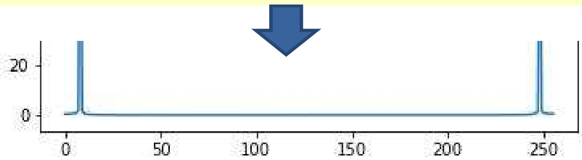
FFTShift


```
plt.plot(freq, spectrum)
```

Plot x, y pair
The sequence order is not changed.



```
In [118]: freq
Out[118]:
[ 0.     1.25   2.5   ...,  -3.75 -2.5   -1.25]
In [119]: spectrum[:10]
Out[119]:
array([ 0.39984513, 0.40617256, 0.42641611,
0.46504507, 0.53258955, 0.65487813, 0.91034296,
1.6889461, 128.06700088, 1.52828374])
```

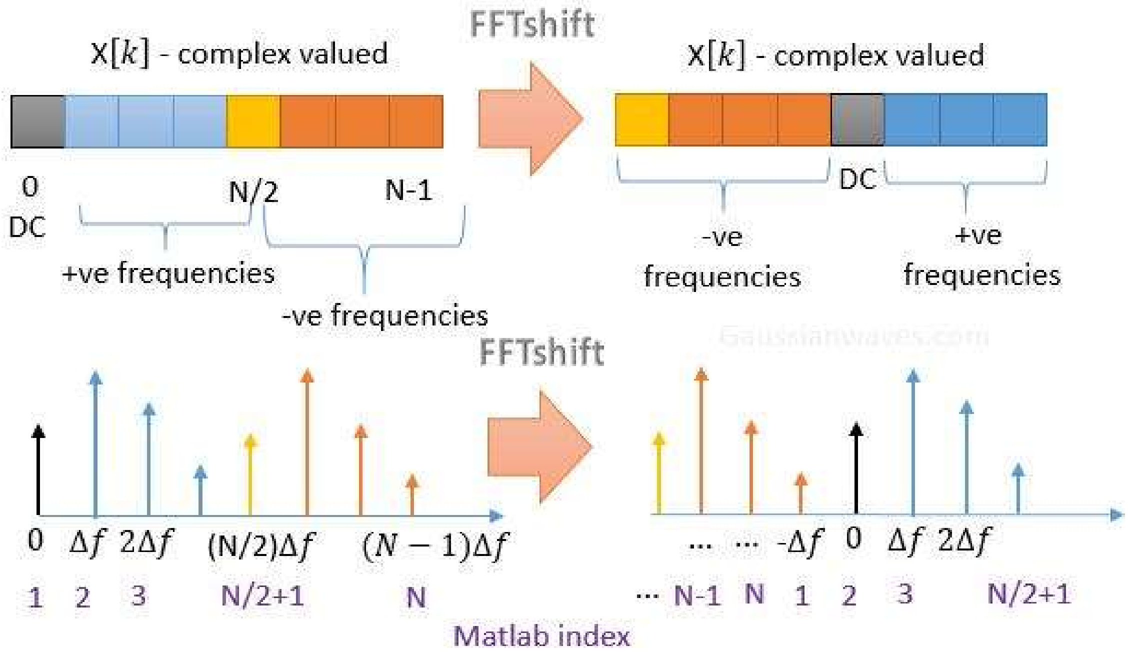


FFTShift

- From the plot we see that the frequency axis starts with DC, followed by positive frequency terms which is in turn followed by the negative frequency terms.
- To introduce proper order in the x-axis, one can use FFTshift function, which arranges the frequencies in order:
  - negative frequencies → DC → positive frequencies.
  - The fftshift function need to be carefully used when N is odd.

FFTShift

27



FFTShift

28

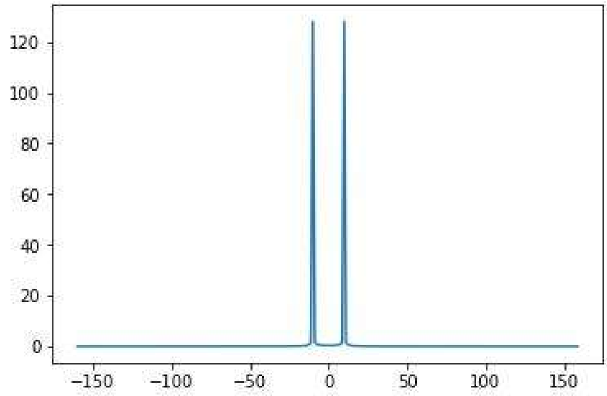
- **numpy.fft.fftshift(x, axes=None)**
  - Shift the zero-frequency component to the center of the spectrum.
  - Note that y[0] is the Nyquist component only if len(x) is even.
- **numpy.fft.ifftshift(x, axes=None)**
  - The inverse of fftshift.
  - Although identical for even-length *x*, the functions differ by one sample for odd-length *x*.

```
>>> freqs = np.fft.fftfreq(10, 0.1)
>>> freqs
array([ 0.,  1.,  2.,  3.,  4., -5., -4., -3., -2., -1.])
>>> np.fft.fftshift(freqs)
array([-5., -4., -3., -2., -1.,  0.,  1.,  2.,  3.,  4.])
```

FFTShift


```
shift_freq = np.fft.fftshift(freq)
shift_spec = np.fft.fftshift(spectrum)
plt.figure()
plt.plot(shift_freq, shift_spec)
```

```
shift_freq
Out[118]:  array([-160.   , -158.75, -
157.5  , ...,   156.25,   157.5  ,   158.75])
In [120]: shift_spec[:10]
Out[120]:
array([ 0.00232974,  0.00233041,
0.00233242,  0.00233576,  0.00234045,
0.00234648,  0.00235386,  0.0023626  ,
0.0023727  ,  0.00238418])
```



Frequency filtering

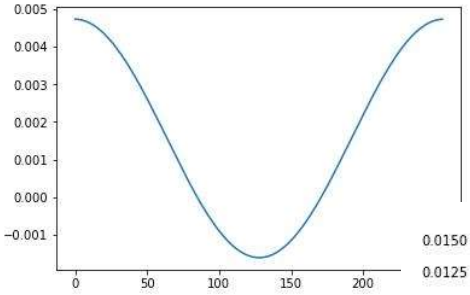
- filtering by setting cut-off frequency

```
lowPassMask = abs(freq) <=2 # cut-off frequency=2
print('non_zero= ', np.count_nonzero(lowPassMask))
lowPassFy = freqY.copy()
lowPassFy[~lowPassMask] = 0 # ~, equivalent to logical_not
lowPassY = np.fft.ifft(lowPassFy)
plt.figure()
plt.plot(lowPassY.real)
```

ifft() output is complex values. We only get the real part.

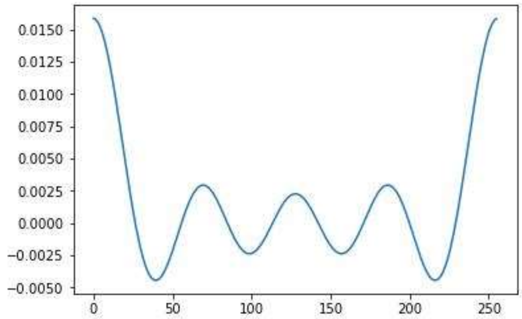
Low-pass filter

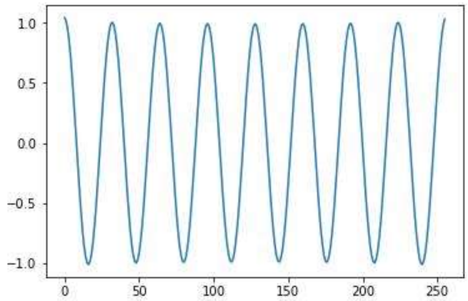
31



cut-off frequency =2

cut-off frequency =5





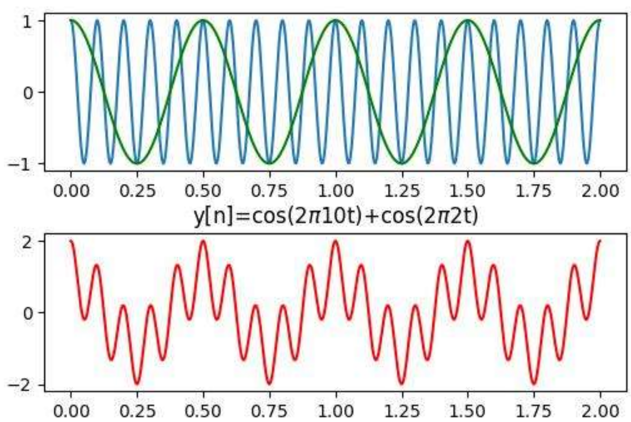
cut-off frequency =10

Example: Different frequency components in a signal

32

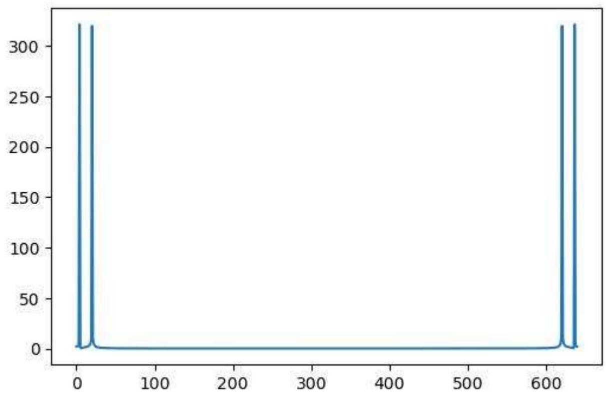
```
fs = 320 # sampling frequency
duration = 2  # 2 seconds duration
t = np.linspace(0, duration, duration*fs)
y10 = np.cos(2*np.pi*10*t)
y2 = np.cos(2*np.pi*2*t)
y = y10 + y2

plt.figure()
f, (ax1, ax2) = plt.subplots(2, 1)
plt.subplots_adjust(hspace = 0.4)
ax1.plot(t, y10)
ax1.plot(t, y2, 'g')
ax2.set_title(r'y[n]=cos(2$\pi$10t)+cos(2$\pi$2t)')
ax2.plot(t, y, 'r')
```

Signal: $y(t)=\cos(2\pi*10*t)+\cos(2\pi*2*t)$

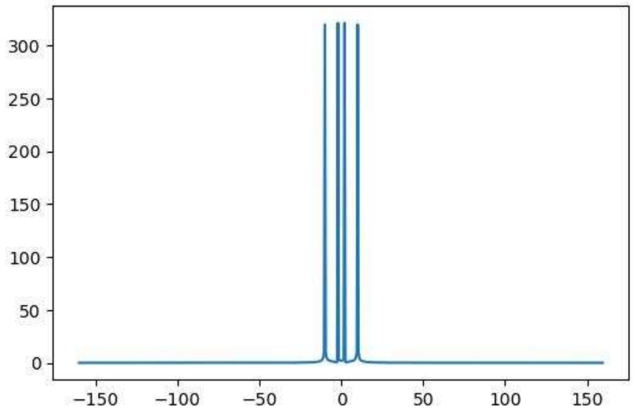


Many frequency components

- Fourier spectrum (N=640 #FFT size)



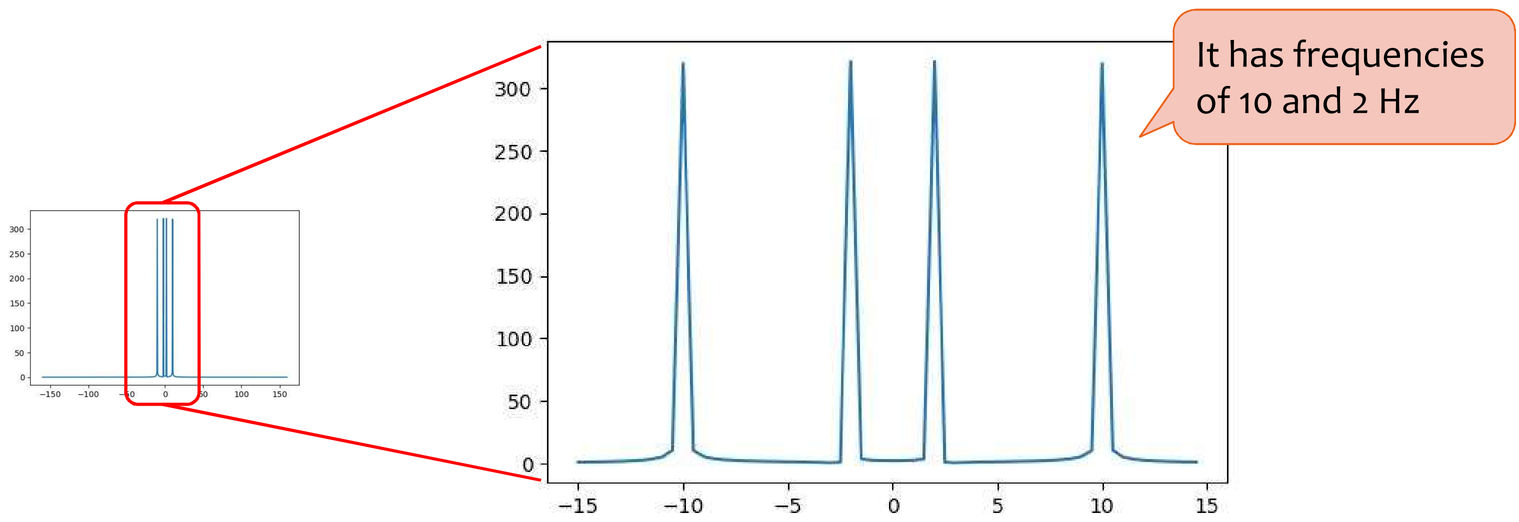
Before fftshift



After fftshift

Many frequency components

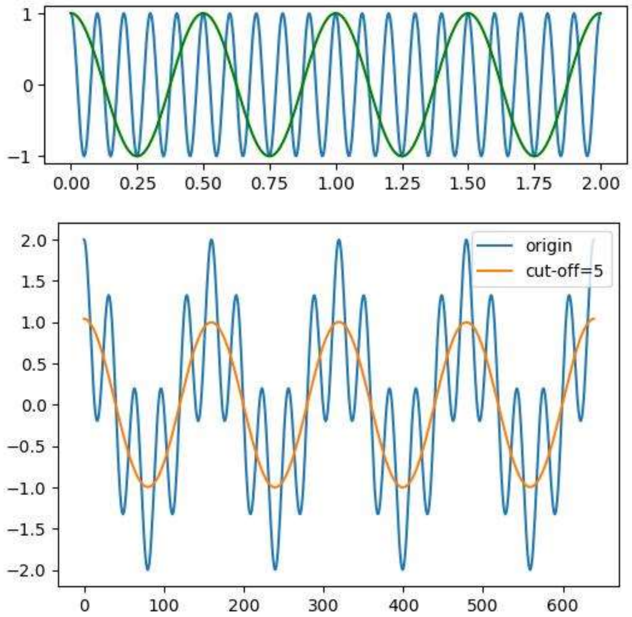
- plt.plot(shift_freq[290:350], shift_spec[290:350])



Low-pass

- cut-off frequency =5

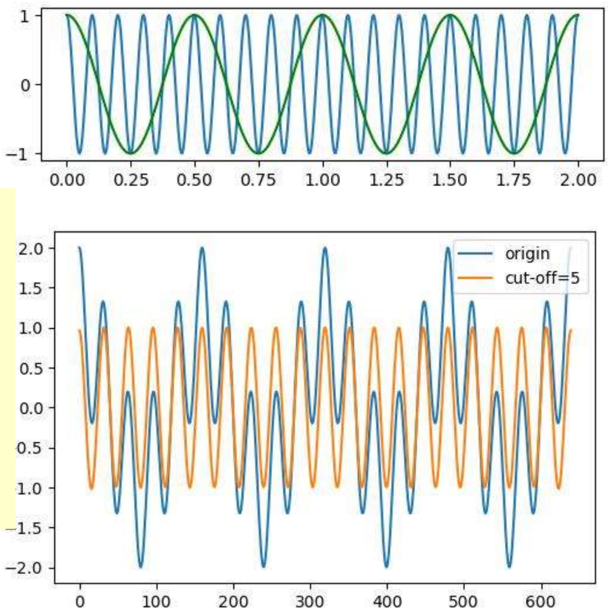
```
n = len(y)
freqY = np.fft.fft(y)
freq = np.fft.fftfreq(n, d=1/fs)
lowPassMask = abs(freq) <=5
lowPassFy = freqY.copy()
lowPassFy[~lowPassMask] = 0
lowPassY = np.fft.ifft(lowPassFy)
plt.figure()
fig, ax = plt.subplots()
ax.plot(y, label='origin')
ax.plot(lowPassY.real, label='cut-off=5')
legend = ax.legend()
```



High-pass

- cut-off frequency =5

```
highPassMask = abs(freq) >=5
highPassFy = freqY.copy()
highPassFy[~highPassMask] = 0
highPassY = np.fft.ifft(highPassFy)
plt.figure()
fig, ax = plt.subplots()
ax.plot(y, label='origin')
ax.plot(highPassY.real, label='cut-off=5')
legend = ax.legend()
```



ALTERNATIVE METHOD FOR CREATING LOWPASS FILTER IN SCIPY

Reference

- http://stackoverflow.com/questions/25191620/creating-lowpass-filter-in-scipy-understanding-methods-and-units
- **Scipy Signal processing (scipy.signal)**
  - https://docs.scipy.org/doc/scipy/reference/signal.html

Butterworth filter

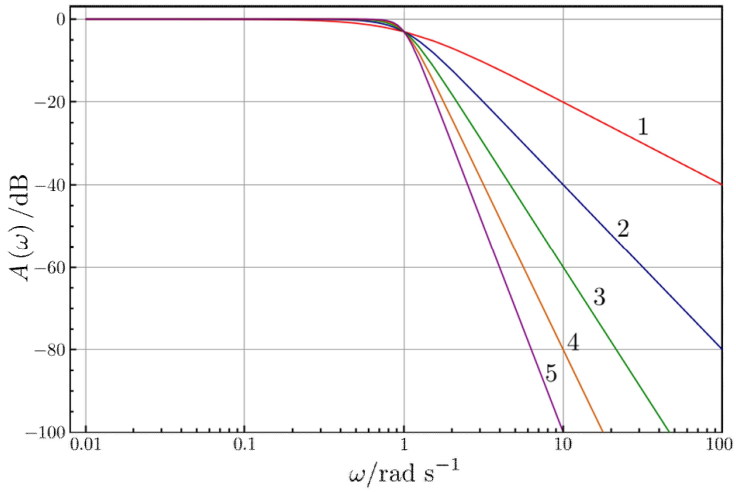
- scipy.signal.butter(*N*, *Wn*, *btype='low'*, *analog=False*, *output='ba'*)
  - Butterworth digital and analog filter design.
  - Parameters
    - **N** : int, The order of the filter.
    - **Wn** : array_like, A scalar or length-2 sequence giving the critical frequencies.
    - **btype** : {'lowpass', 'highpass', 'bandpass', 'bandstop'}, optional
      - Default is 'lowpass'.

Butterworth filter

- Parameter **Wn**
  - For a Butterworth filter, this is the point at which the gain drops to 1/sqrt(2) that of the passband (the "-3 dB point").
  - For digital filters, Wn is normalized from 0 to 1, where 1 is the Nyquist frequency, pi radians/sample. (Wn is thus in half-cycles / sample.)
  - For analog filters, Wn is an angular frequency (e.g. rad/s).
- Returns
  - **b, a** : ndarray, ndarray
    - Numerator (*b*) and denominator (*a*) polynomials of the IIR filter. Only returned if output='ba'.

Butterworth filter

- Plot of the gain of Butterworth low-pass filters of orders 1 through 5, with cutoff frequency w_0=1.



scipy.signal.lfilter

- scipy.signal.lfilter(*b*, *a*, *x*, *axis*=-1, *zi*=*None*)
  - Filter a data sequence, x, using a digital filter.
  - Parameters
    - **b, a** : ndarray, ndarray
      - Numerator (*b*) and denominator (*a*) coefficient vectors in a 1-D sequence.
    - **x** : array_like, An N-dimensional input array.
  - Return
    - **y** : array, The output of the digital filter.

This has phase shift problem.

- Use scipy.signal.filtfilt(*b*,*a*, *x*, ...) instead of lfilter()
  - This function applies a linear filter twice, once forward and once backwards.

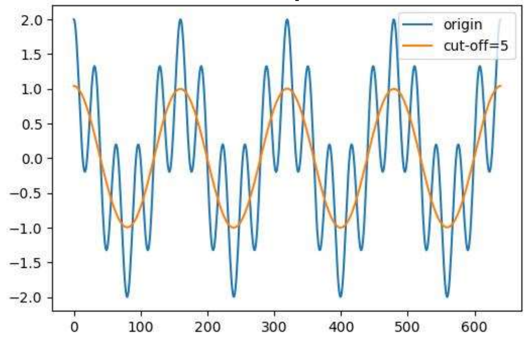
Creating lowpass filter


```
import scipy.signal as sg
def butter_lowpass(cutoff, fs, order=5):
    nyq = 0.5 * fs
    normal_cutoff = cutoff / nyq
    b, a = sg.butter(order, normal_cutoff, btype='low', analog=False)
    return b, a

def butter_lowpass_filter(data, cutoff, fs, order=5):
    b, a = butter_lowpass(cutoff, fs, order=order)
    y = sg.filtfilt(b, a, data)
    return y
```

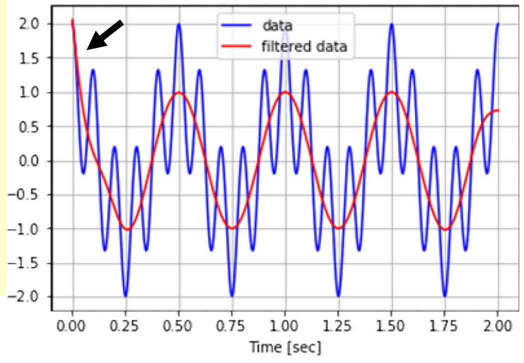
Creating lowpass filter


```
# Filter the data, and plot both the original
# and filtered signals.
order = 6
cutoff = 5

out = butter_lowpass_filter(y, cutoff, fs, order)
plt.figure()
plt.plot(t, y, 'b-', label='data')
plt.plot(t, out, 'g-', label='filtered data')
plt.xlabel('Time [sec]')
plt.grid()
plt.legend()
```

FFT low-pass



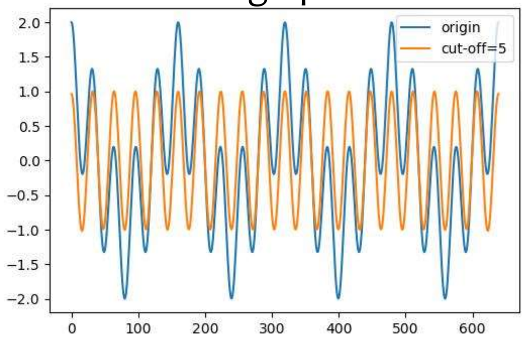
Butterworth low-pass



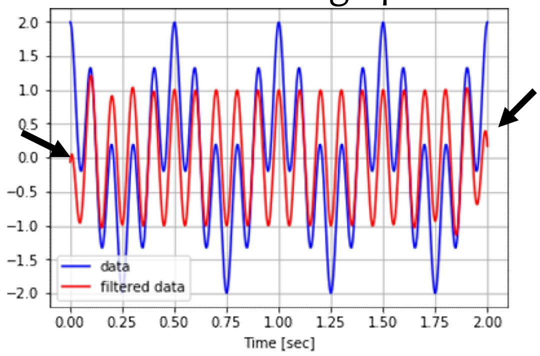
Creating highpass filter


```
def butter_highpass(cutoff, fs, order=5):
    nyq = 0.5 * fs
    normal_cutoff = cutoff / nyq
    b, a = sg.butter(order, normal_cutoff, btype='high', analog=False)
    return b, a
```

FFT high-pass



Butterworth high-pass



Computation time

- Butterworth filter is faster than FFT
- Measure the computation time

```
from time import time
t1 = time()
ax_low_buf = butter_lowpass_filter(ax, cutoff, fs)
ax_high_buf = butter_highpass_filter(ax, cutoff, fs)
t2 = time()
print('Butterworth low high pass takes %f seconds for
ax, ay, az' % (t2-t1))
```