Lecture 02.06 Discrete Fourier transforms

The source for this lecture is in *SageMath* kernel *Jupyter* notebook. For more information, see jupyter.org and sagemath.org.

See ricopic.one/measurement/notebooks for the source code notebook. First, we import packages and all that. We use `matplotlib` for plotting, `numpy` for numerics, and `scipy` for discrete (fast) Fourier transforms.

Modern measurement systems primarily construct spectra by sampling an analog electronic signal $y(t)$ to yield the sample sequence (y_n) and perform a *discrete Fourier transform*.

Definition 02.06.1: discrete Fourier transform

The *discrete Fourier transform* (DFT) of a sample sequence (y_n) of length N is (Y_m), where $m \in [0, 1, \cdots, N-1]$ and

$$Y_m = \sum_{n=0}^{N-1} y_n e^{-j2\pi mn/N}.$$

The *inverse discrete Fourier transform* (IDFT) reconstructs the original sequence for $n \in [0, 1, \cdots, N-1]$ and

$$y_n = \frac{1}{N} \sum_{n=0}^{N-1} Y_m e^{j2\pi mn/N}.$$

The DFT (Y_m) has a frequency interval equal to the sampling frequency ω_s/N and the IDFT (y_n) has time interval equal to the sampling time T. The first $N/2+1$ DFT (Y_m) values correspond to frequencies

and the remaining $N/2-1$ correspond to frequencies

In practice, the definitions of the DFT and IDFT are not the most efficent methods of computation. A clever algorithm called the *fast Fourier transform* (FFT) computes the DFT much more efficiently. Although it is a good exercise to roll our own FFT, in this lecture we will use `scipy`'s built-in FFT algorithm, loaded with the following command.

```
from scipy import fft
```

Now, given a time series array `y` representing (y_i), the DFT (using the FFT algorithm) can be computed with the following command.

```
fft(y)
```

In the following example, we will apply this method of computing the DFT.

02.06.0.1 A DFT/FFT example

We would like to compute the DFT of a sample sequence (y_n) generated by sampling a spaced-out sawtooth. Let's first generate the sample sequence and plot it.

We define the sampling rate `fs`, which defines the sampling interval `Ts`. Furthermore, we define the frequency of the spaced sawtooth signal `f_signal`.

```
fs = 200 # sampling rate
Ts = 1.0/fs # sampling interval
f_signal = 10 # frequency of the signal
```

We want an interval of ramp followed by an interval of "space" (zeros). The following method of generating the sampled signal `y` helps us avoid *leakage*, which we'll describe after the example.

```
arr_zeros = np.zeros(fs/f_signal/2) # half signal period worth of zeros
arr_ramp = np.arange(fs/f_signal/2) # half signal period worth of ramp
y = [] # initialize time sequence
j = 0
for i in range(fs):
    if i % (fs/f_signal/2) == 0:
        # if we are at the start of a signal period
        if j % 2 == 0:
            # every other signal period
            y = np.append(y,arr_zeros)
        else:
            y = np.append(y,arr_ramp)
        j += 1
```

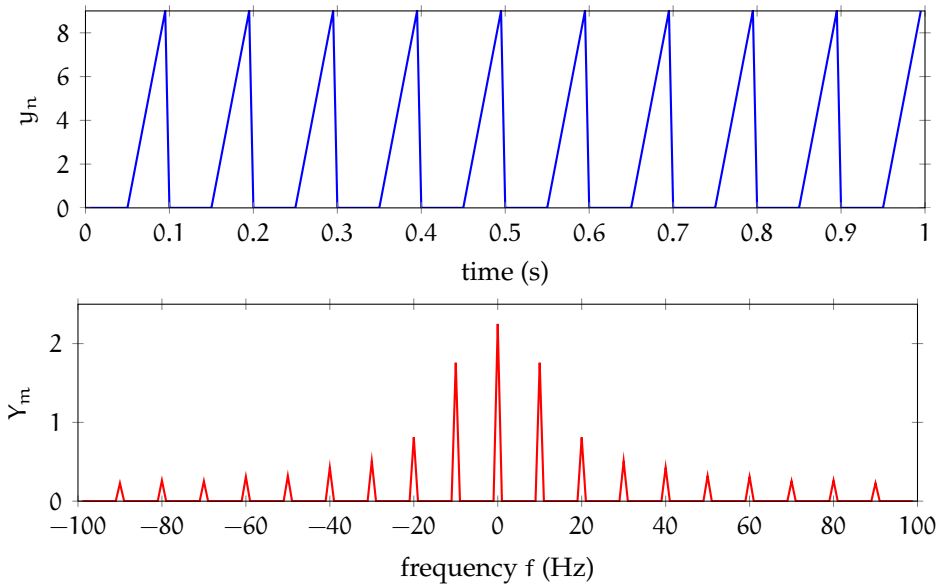

Figure 02.6: (top) a sampled sequence (y_n) plotted through time and (bottom) its discrete Fourier transform sequence (Y_m) plotted through frequency.

From this sequence, we can compute the following parameters.

```
N = len(y) # number of samples
t_a = np.arange(0,N*Ts,Ts) # time array
time_total = N*Ts # total time in series
```

Plotting this with `matplotlib` is fairly straightforward. The result is shown in the top plot of Figure 02.6.

```
plt.figure()
plt.plot(t_a,y,'b-',linewidth=2)
plt.xlabel('time (s)')
plt.ylabel('$y_n$');
```

Display the plot with the following command.

```
plt.show()
```

Now we have a nice time sequence on which we can perform our DFT. It's easy enough to compute the FFT.

```
Y = fft(y)/N # FFT with proper normalization
```

Recall that the latter values correspond to negative frequencies. In order to plot it, we want to rearrange our `Y` array such that the elements corresponding to negative frequencies are first. It's a bit annoying, but *c'est la vie*.

```
Y_positive_zero = Y[range(N/2)]
Y_negative = np.flip(
    np.delete(
        Y_positive_zero,
        0
    ),
    0
)
Y_total = np.append(Y_negative,Y_positive_zero)
```

Now all we need is a corresponding frequency array.

```
freq_total = np.arange(-N/2+1,N/2)*fs/N
```

Now, just to plot.

```
plt.figure()
plt.plot(freq_total, abs(Y_total),'r-',linewidth=2)
plt.xlabel('frequency $f$ (Hz)')
plt.ylabel('$Y_m$');
```

And now display the plot of the spectrum, shown on the bottom of Figure 02.6.

```
plt.show();
```

02.06.0.2 Leakage

The DFT assumes the sequence (y_n) is periodic with period N. An implication of this is that if any periodic components have period $N_{\text{short}} < N$, unless N is divisible by N_{short}, spurious components will appear in (Y_n).

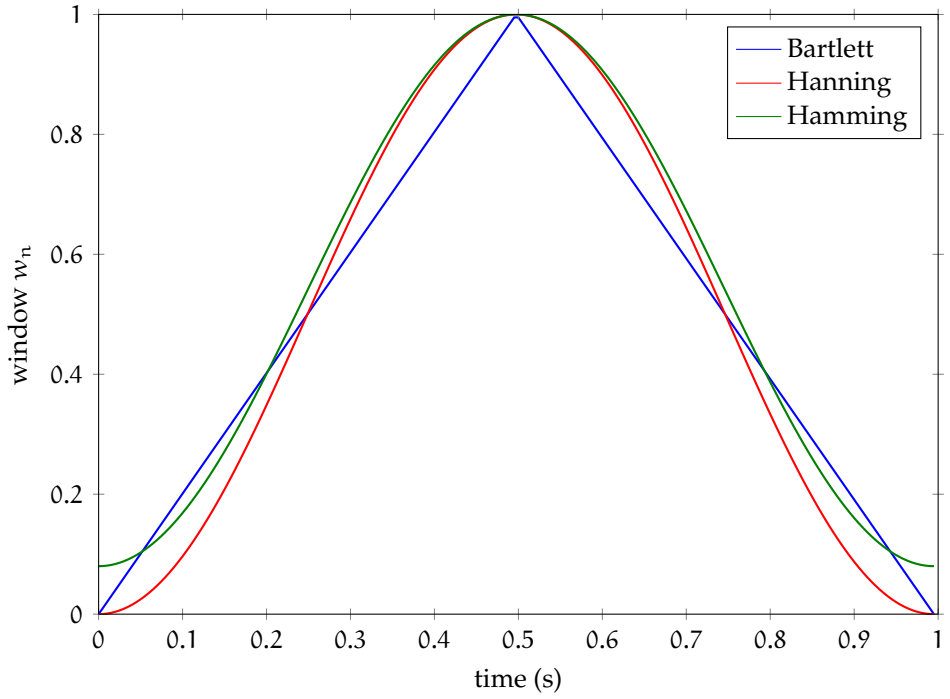

Figure 02.7: three sample window functions.

Avoiding leakage is difficult, in practice. Instead, typically we use a *window function* to mitigate its effects. Effectively, windowing functions—such as the *Bartlett*, *Hanning*, and *Hamming windows*—multiply (y_n) by a function that tapers to zero near the edges of the sample sequence.

Numpy has several window functions such as `bartlett()`, `hanning()`, and `hamming()`. For usage information on a function, the following `?` idiom is useful.

```
np.hanning?
```

Let's plot the windows to get a feel for them.

```
bartlett_window = np.bartlett(N)
hanning_window = np.hanning(N)
hamming_window = np.hamming(N)
```

```
plt.figure()
plt.plot(t_a,bartlett_window,'b-',label='Bartlett',linewidth=2)
plt.plot(t_a,hanning_window,'r-',label='Hanning',linewidth=2)
plt.plot(t_a,hamming_window,'g-',label='Hamming',linewidth=2)
plt.xlabel('time (s)')
plt.ylabel('window $w_n$')
plt.legend();
```

Show the figure Figure 02.7.

```
plt.show()
```