four.dft Discrete and fast Fourier transforms

1. Python code in this section was generated from a Jupyter notebook named `discrete_fourier_transform.ipynb` with a `python3` kernel.

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 four.6: 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

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

m

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

$$(0, \omega_s/N, 2\omega_s/N, \cdots \omega_s/2)$$

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

$$(-\omega_s/2, -(N-1)\omega_s/N, \cdots, -\omega_s/N).$$

0, 1, 2, -2, -1

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.

python

```
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.

Example four.dft−1 **re: FFT of a sawtooth signal**

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.

In addition to `scipy`, let's import `matplotlib` for figures and `numpy` for numerical computation.

```
import matplotlib.pyplot as plt
import numpy as np
```

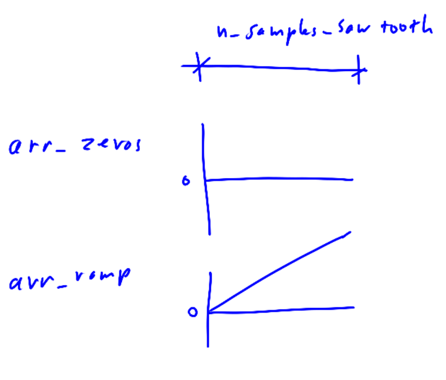
We define several "control" quantities for the spaced-sawtooth signal.

```
f_signal = 48 # frequency of the signal
spaces = 1 # spaces between sawteeth
n_periods = 10 # number of signal periods
n_samples_sawtooth = 10 # samples/sawtooth
```

These quantities imply several "derived" quantities that follow.

```
n_samples_period = n_samples_sawtooth*(1+spaces)
n_samples = n_periods*n_samples_period
T_signal = 1.0/f_signal # period of signal
t_a = np.linspace(0,n_periods*T_signal,n_samples)
dt = n_periods*T_signal/(n_samples-1) # sample time
f_sample = 1./dt # sample frequency
```

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 at the end of the example.

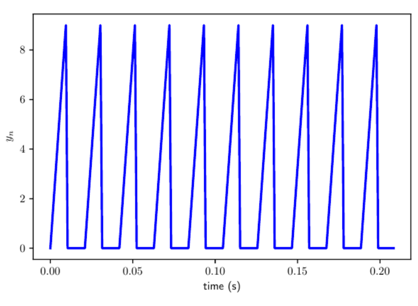



```
arr_zeros = np.zeros(n_samples_sawtooth) # frac of period
arr_ramp = np.arange(n_samples_sawtooth) # frac of period
y = [] # initialize time sequence
for i in range(n_periods):
  y = np.append(y,arr_ramp) # ramp
  for j in range(spaces):
    y = np.append(y,arr_zeros) # space
```

We plot the result in Fig. dft.1, generated by the following code.

```
fig, ax = plt.subplots()
plt.plot(t_a,y,'b-',linewidth=2)
plt.xlabel('time (s)')
plt.ylabel('$y_n$')
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_samples # 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.

0, 0.1, 0.2, -0.2, -0.1

Y_0 Y_1 Y_2 Y_3 Y_4

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

Y_4 Y_3 Y_0 Y_1 Y_2

-0.2 -0.1 0 0.1 0.2

Now all we need is a corresponding frequency array.

```
freq_total = np.arange(
  -n_samples/2+1,n_samples/2
)*f_sample/n_samples
```

The plot, created with the following code, is shown in Fig. dft.2.

```
fig, ax = plt.subplots()
plt.plot(freq_total, abs(Y_total),'r-',linewidth=2)
plt.xlabel('frequency $f$ (Hz)')
plt.ylabel('$Y_m$')
plt.show()
```

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 $N_{short} < N$, unless N is divisible by N_{short}, spurious components will appear in (Y_n). 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.

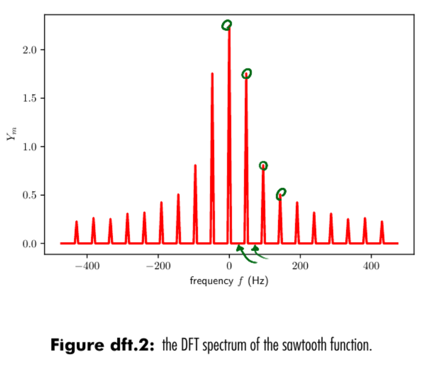


Figure dft.2: the DFT spectrum of the sawtooth function.

Numpy has several window functions such as `bartlett()`, `hanning()`, and `hamming()`.
Let's plot the windows to get a feel for them – see Fig. dft.3.

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

fig, ax = plt.subplots()
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()
plt.show()
```

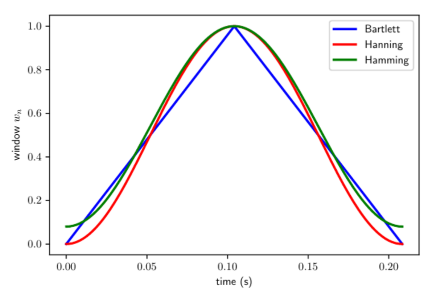


Figure dft.3: three window functions to minimize leakage.