COE3DY4 Lab #1
DSP Primitives in Python

Objective

The purpose of this lab is to develop the understanding of some basic digital signal processing (DSP) primitives used in Software Defined Radios (SDRs) and implement in `Python`.

Preparation

- Revise the fundamental material from signals and systems, in particular the linear time-invariant (LTI) systems, impulse response, convolution, Nyquist rate, Fourier transform
- Revise basic programming in both `Python` and `MATLAB`, because of the inherent similarities between `SciPy/NumPy` and `MATLAB`
- Get familiar with `SciPy/NumPy` at `https://www.scipy.org/`

Discrete Fourier Transform

The Fourier transform is widely used in the fields of science and engineering, ranging from physics and astronomy to multimedia processing and communications to data science and machine learning. One simple way to think of the Fourier transform is as a change of a signal from one domain (time) to another one (frequency) for more intuitive and efficient data representation and computational processing. It relies on complex exponentials, used to manipulate periodic trigonometric functions, i.e., $\sin(\theta)$ and $\cos(\theta)$, using Euler's formula $e^{j\theta} = \cos(\theta) + j(\theta)$, where $j = \sqrt{-1}$.

Formally, the continuous-time Fourier Transform of a time domain signal $x(t)$ is defined as:

$$X(f) = \int_{-\infty}^{\infty} x(t)e^{-j2\pi ft}\,dt \qquad \text{or} \qquad X(\omega) = \int_{-\infty}^{\infty} x(t)e^{-j\omega t}\,dt$$

$X(f)$ or $X(\omega)$ gives the frequency spectrum of the signal, where the frequency f is measured in Hz and the angular frequency ω is measured in radians/second. The above is also referred to as the *analysis* formula. The *synthesis* formula is defined using the inverse Fourier transform:

$$x(t) = \int_{-\infty}^{\infty} X(f)e^{j2\pi ft}\,df \qquad \text{or} \qquad x(t) = \frac{1}{2\pi}\int_{-\infty}^{\infty} X(\omega)e^{j\omega t}\,d\omega$$

In Fourier analysis, both the time and frequency domain variables can be continuous or discrete. The above formulas assume continuous time/frequency across infinite intervals. A detailed and rigorous mathematical coverage of the Fourier transform, which goes beyond the scope of this project course on SDRs, can be found online at `https://ccrma.stanford.edu/~jos/st/`. For our project, it is still worth stating that the discrete-time Fourier Transform discretizes the time variable only, whereas the Discrete Fourier Transform (DFT) has both the time and the frequency variables discretized. Since DFT is the most commonly used Fourier-type transform in practical applications, due to its computational nature, we will use it for the rest of this project on SDRs.

Assuming we have an input signal represented as a block of N samples $x(k)$, with $k = 0 \ldots N-1$, the DFT is defined as:

$$X(m) = \sum_{k=0}^{N-1} x(k) e^{-2\pi j \frac{km}{N}}$$

where $X(m)$ is the m^{th} frequency component (or bin) in the spectrum of the input signal. The above formula has a straightforward translation to pseudocode:

Input: Time domain samples $x(k)$ with $k = 0 \ldots N-1$
Output: Discrete frequency bins $X(m)$ with $m = 0 \ldots N-1$

for $m \in [0, N-1]$ **do**
  $X(m) \Leftarrow 0$
  for $k \in [0, N-1]$ **do**
    $X(m) \Leftarrow X(m) + x(k) \times e^{-2\pi j \frac{km}{N}}$
  end for
end for

Note, using the notation of **twiddle factors**, i.e., $W_N^r = e^{j\frac{2\pi r}{N}}$, the DFT can be rewritten as:

$$X(m) = \sum_{k=0}^{N-1} x(k) W_N^{-km}$$

W_N^r are in fact the N^{th} roots of unity and simple math can show that $(W_N^r)^N = 1$, for each $r = 0 \ldots N-1$. Note, the properties of these twiddle factors are central to the derivation of the faster implementations of the DFT, such as the Fast Fourier Transform (FFT). For the rest of this course we will use the notation of twiddle factors when discussing DFT/FFT.

Based on the same inversion principle as for the continuous-time Fourier transform, the Inverse Discrete Fourier Transform (IDFT) is defined as:

$$x(k) = \frac{1}{N} \sum_{m=0}^{N-1} X(m) W_N^{km}$$

In the lab you should perform the following tasks, which are aimed at strengthening the link between *mathematical formalism* and *algorithmic thinking* in DSP (and system design in general):

- Write your **own** functions for DFT and IDFT in Python. Confirm their functional correctness against the built-in implementations of the FFT and inverse FFT (IFFT) from `SciPy`.
- Generate a random signal using `NumPy` with 1,000 samples, whose magnitude is normalized to -10 to +10, and confirm that in your implementation of DFT/IDFT the signal energy holds (as proven formally by Parseval's theorem). Note, the signal energy in the time domain is summation of the squares of samples' absolute values, i.e., $|x(k)|^2$. In the frequency domain, it is the summation across all frequency components of their individual energy spectral density, i.e., $|X(m)|^2$. Stated differently, $\sum_{k=0}^{N-1} |x(k)|^2$ must equal $\sum_{m=0}^{N-1} |X(m)|^2$.
- Using the reference code for a pure tone of 100 Hz that is plotted it in time and frequency domain, write a function to generate a multi-tone signal (up to 3 tones is sufficient) where the sines have different amplitudes, frequencies and phases. Visualize your multi-tone signal in both time and frequency domain.

Digital Filter Design

For linear-time invariant (LTI) systems that are stable and causal, digital filtering can be performed via convolution (using operator $*$). Consider two signals $x(a)$ of size A and $h(b)$ of size B. The discrete convolution of these two sequences $y(c)$ is defined as :

$$y(c) = (x * h)(c) = \sum_{l=-\infty}^{\infty} x(l)h(c-l) = (h * x)(c)$$

In the above formula, when the indices for x and y are below zero or above their maximum size, i.e., $A-1$ for x and $B-1$ for h, the values of the corresponding $x(l)$ and $h(c-l)$ are assumed to be zero. The total number of non-zero elements in signal y will be equal to $A+B-1$.

A central theorem of the Fourier theory is that the transform of the convolution of two sequences in the time domain is equal to pointwise multiplication of their transforms in the frequency domain (referred to as the convolution theorem). One of the implications of this theorem is:

$$DFT(x * h) = DFT(x)DFT(h) \implies x * h = IDFT(DFT(x)DFT(h))$$

The above can be leveraged for the computational speed-up of the convolution operation (only when DFT is replaced via FFT). However, since computational speed-ups are beyond the scope of this lab, at this time, the convolution theorem will be used to understand how to derive the impulse response of a digital filter, i.e., the coefficients that are used to weigh the input samples.

An ideal low-pass filter will eliminate all the frequency components of the input signal that are above a cutoff frequency f_c. This can be achieved by multiplying the frequency response of the input sequence with a rectangular window; the height of the rectangular window defines the filter gain and the window width from -f_c to f_c defines the filter's pass band. The inverse Fourier transform of the rectangular window is the *sinc* function, defined as $\frac{sin(\pi t)}{\pi t}$ (when t is 0 *sinc* is defined by its limiting value of 1). The shapes of the rectangular window and the *sinc* function are shown below primarily for illustrative purposes.

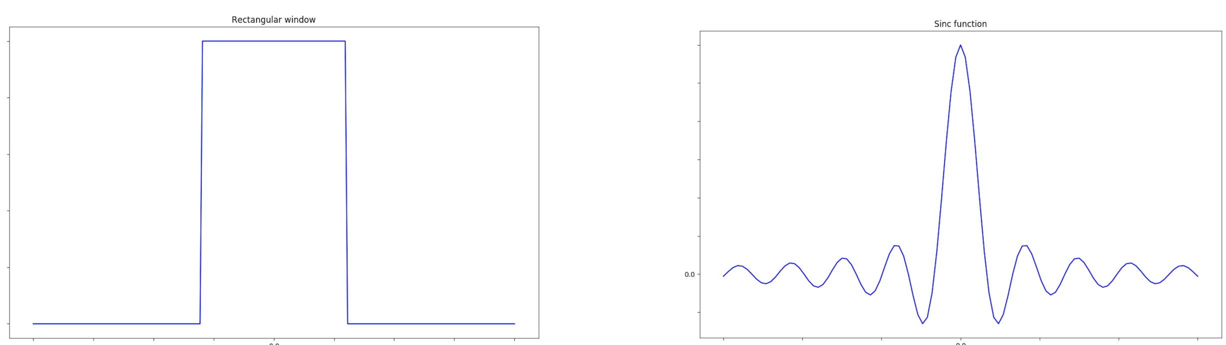

Figure 1: The Fourier transform pair: rectangular window and *sinc* function

To summarize the above, the impulse response of a low-pass filter is derived from the *sinc* function, whose argument and scale factor are dependent on the normalized cutoff frequency as follows:

$$h(i) = \frac{f_c}{f_s/2} sinc(\frac{f_c}{f_s/2} i)$$

In the above formula f_c is the cutoff frequency and f_s is the sampling rate. Note, when computing the normalized frequency $\frac{f_c}{f_s/2}$ the cutoff frequency f_c is divided by $f_s/2$, which is the Nyquist frequency that must be greater than the bandwidth of the input signal. Note also, from the the above formula, it is implied the impulse response converges to $\frac{f_c}{f_s/2}$ at its center point.

One point that must be clarified before translating the previous mathematical formalism into pseudocode is that the impulse response derived via the *sinc* function needs to be *windowed.* In practice the filters based on convolution are limited in size, hence the term finite impulse response (FIR) filters. Because the ideal filter assumes an infinite response, FIRs will truncate the *sinc* function based on the number of filter taps N_{taps}. If a rectangular window is used for truncation, one of its well-understood side-effects is excessive *spectral leakage* in the frequency domain. Spectral leakage is the phenomena where new frequencies arise in the spectrum because of the behavior induced by truncation at the boundaries of the FIR. To alleviate this problem, the impulse response is *windowed.* The trade-offs involved in different types of windows and their impact on spectral leakage are well-studied, however they are beyond the scope of our project. In our filter design we will rely on the Hann window, which is one of the oldest ones, yet most commonly used in practice.

The pseudocode for deriving the impulse response coefficients of a low pass filter is given below.

Input: Filter parameters: f_c, f_s and N_{taps}
Output: Taps of the FIR filter $h(i)$ with $i = 0 \ldots N_{taps} - 1$

$Norm_{cutoff} \Leftarrow \frac{f_c}{f_s/2}$ ▷ define explicitly the normalized cutoff frequency $Norm_{cutoff}$

for $i \in [0, N_{taps} - 1]$ **do**
  if $i = (N_{taps} - 1)/2$ **then**
    $h(i) \Leftarrow Norm_{cutoff}$ ▷ avoid division by zero in *sinc* for the center tap when N_{taps} is odd
  else
    $h[i] \Leftarrow Norm_{cutoff} \frac{sin(\pi Norm_{cutoff}(i-(N_{taps}-1)/2))}{\pi Norm_{cutoff}(i-(N_{taps}-1)/2)}$
  end if
  $h(i) \Leftarrow h(i) sin^2(\frac{i\pi}{N_{taps}})$ ▷ apply the Hann window
end for

It is important to mention that deciding on the number of taps for an FIR filter can be a cumbersome task. It depends on many factors, including pass band gain/ripple, stop band attenuation/ripple, transition band width, roll-off ... On the one hand, the larger the number of taps, the better the FIR approximation of the infinite impulse response, hence better signal quality; on the other hand, the more taps, the more multiplications and accumulations need to be done, hence there is an increased computational demand. An empirical yet effective decision can be reached using the "harris rule of thumb", where the number of taps are determined as:

$$N_{taps} = \frac{f_s}{\Delta f_t} \frac{Attn(dB)}{22}$$

In the above formula, in addition to the already-introduced number of taps (N_{taps}) and the sampling frequency (f_s), the Δf_t is the transition band width and $Attn(dB)$ is the stop band attenuation (in dB from the main lobe to the peak of the first side lobe in the magnitude frequency response). For our SDR application, usually at least $40dB$ attenuation is preferred; naturally, the higher the attenuation the better it is in terms of signal quality, however it is worse in terms of compute speed, which makes it more challenging to meet real-time constraints. The relative size of the transition band width to the sampling rate varies from one use case to another, however it is rarely the case that Δf_t is much higher than 10% of f_s. To conclude, the key point of this discussion is that in practice compromises are needed when implementing filters in real-time by trading off the signal quality against the compute speed. Other non-functional requirements, e.g., energy-efficiency, can add further dimensions to this complex engineering problem of system design.

Consistent with the earlier objective to link *mathematical formalism* to *algorithmic thinking*, in the lab you should perform the following:

- Implement the given pseudocode to design your **own** low-pass impulse response in `Python`.
- Compare your impulse response against the built-in methods from `SciPy` for filter design, such as `firwin`. This can be achieved by visually inspecting the frequency response of both methods using `freqz`.
- Use your impulse response to filter the lowest frequency of the multi-tone signal from the previous experiment. You can achieve this by using `lfilter`. You should visualize the outcome both in the time domain and in the frequency domain.

Digital Filtering on Input Streams Divided in Blocks

The previous sections have discussed a few foundational concepts in DSP, which have enabled you to build your own tools for the design of impulse responses and spectrum analysis. They will be useful way beyond this lab, as you transition to a more refined implementation in C/C++. This section discusses a complementary, yet important, practical consideration that must be accounted for in real-time implementations when input data is continuously streamed.

The convolution of the impulse response with an input sequence can be done in a single-pass only if **all** the input data has been buffered. In many applications, especially when the input is streamed, this is not practically feasible. In such cases the input stream is segmented into blocks that are processed one-at-a-time. However, to avoid discontinuities in the output sequence, at the end of each block the convolution needs to be stopped and resumed from a state that contains all the input samples needed to compute the outputs for the new block. The above challenge is best explained through an example. Assume the input x is convolved with an impulse response h with 10 taps. Assume also the blocks are of size 100. Consider the case when two consecutive output samples from different blocks ($y(99)$ and $y(100)$) are computed:

$$y(99) = x(90)h(9) + x(91)h(8) + ... + x(99)h(0)$$

$$y(100) = x(91)h(9) + x(92)h(8) + ... + x(100)h(0)$$

For $y(99)$ all the input samples are from the same block. However, for $y(100)$ we need one sample from the second block ($x(100)$) and 9 samples from the first block ($x(91)$ to $x(99)$). If the convolution for the second block is processed independently from the first one, then samples $x(91)$ to $x(99)$ will be missing (they are assumed to be zero by default), which clearly introduces discontinuities in the output sequence. To avoid the above problem there is the need to save the samples $x(91)$ to $x(99)$ from the first block to use them when computing the first sample from the second block.

In the lab you need to perform the following:

- The reference code filters a `.wav` file both in a single-pass, as well as in blocks, where the filter state at the end of each block is not saved. Vary the size of blocks and understand when and why there are audible artifacts due to lack of state saving when transitioning between consecutive blocks.
- Based on the `SciPy` documentation for `lfilter` understand how you can use the additional input/output arguments, e.g. `zi`, to save/resume from the filter state at the border between consecutive blocks. Update the code to properly transition between blocks.

Take-home exercises

In addition to completing and submitting your in-lab experiments, to further improve your understanding of DSP, as well as to become more proficient with `Python` and comfortable with its libraries for signal processing, you should perform the following:

- Add functionality to `fourierTransform.py` to generate periodic square waves of arbitrary duty cycles. Plot these waves in both time and frequency domain. As you try to explain the frequency spectrum of square waves, revisit your background knowledge, search for explanations, consolidate your understanding, ..., and summarize your thoughts in the report.
- Learn how to implement a bandpass filter using `firwin` from `SciPy`. Confirm your understanding both through plotting the frequency response in `filterDesign.py`, as well as by looking at filtered data in the time domain (for example, you can eliminate the middle tone from a three-tone signal). Summarize your thoughts in the report by relating your background knowledge to the exploratory work from this undertaking.
- First, implement audio filtering in `blockProcessing.py` by replacing `firwin` with your own function (from the in-lab experiment on filter design) to derive the filter coefficients. Furthermore, you should implement your own function for convolution that should replace the functionality from `lfilter`, as well as it will guarantee that data can be processed in blocks without discontinuities in the output sequence. It is not expected for your solution to deriving filter coefficients and convolution to produce bit-level equivalence against `firwin` and `lfilter`. The computational speed is also of no concern at this time (`NumPy` data structures in C/C++ are actually very efficient). Nonetheless the filtered audio should be perceived similarly under the same settings (number of taps, cutoff frequency and block size). You will have to build your solution incrementally with regular self-checks and comparisons against the reference methods from `SciPy`. This is critical to smoothen the transition from modeling DSP primitives in `Python` to their implementation in C/C++ for an SDR system. Summarize in the report the challenges and your solutions for this (more complex) development task.

Your report should have three sections, one for each of the above items. One page is sufficient and it should not be exceeded unless there are out-of-ordinary points to be made.

Your sources should stay in the `src` sub-folder of the GitHub repo; your report (in .pdf, .txt or .md format) should be included in the `doc` sub-folder.

Your submission, which is to be done via a push to the master branch of your group's GitHub repo for this lab, is due 14 hours before your next lab session is scheduled. Late submissions will be penalized.

This lab is worth 5 % of your total grade.