A3: Fourier Properties

Audio Signal Processing for Music Applications

Introduction

This assignment is to get a better understanding of some of the Fourier theorems and useful properties of the DFT. You will write short pieces of code that implement and verify several properties of the DFT that were discussed in the class. You will also learn to use the dftModel python function of sms-tools. There are five parts in this assignment: 1) Minimize energy spread in DFT of sinusoids, 2) Optimal zero-padding, 3) Symmetry properties of the DFT, 4) Suppressing frequency components using DFT model, and 5) FFT size and zero-padding. The Part-5 of this assignment is optional and will not contribute towards your final grade.

Relevant Concepts

DFT of sinusoids: When a real sinusoid has an integer number of cycles in N samples, the frequency of the sinusoid exactly matches one of the bin frequencies in an N point DFT. Hence the DFT spectrum of the sinusoid has a value of zero at every DFT bin except at the two bins that match the frequency of the sinusoid, as shown in Equation 1. Otherwise, the energy of the sinusoid is spread over all the bins. When there are multiple sinusoids, the equations extend to each sinusoid.

$$x[n] = A_0 \cos(2\pi k_0 n/N) = \frac{A_0}{2}e^{j2\pi k_0 n/N} + \frac{A_0}{2}e^{-j2\pi k_0 n/N} \quad (1)$$

$$X[k] = \frac{A_0}{2} \text{ for } k = k_0, -k_0; \;\; 0 \text{ otherwise} \quad (2)$$

Zero-padding: Zero-padding a signal is done by adding zeros at the end of the signal. If we perform zero-padding to a signal before computing its DFT, the resulting spectrum will be an interpolated version of the spectrum of the original signal. In most implementations of the DFT (including the FFT algorithms) when the DFT size is larger than the length of the signal, zero-padding is implicitly done.

Real, even and odd signals: A signal is real when it does not have any imaginary component, and all sounds are real signals. A signal f is even if $f[n] = f[-n]$, and odd if $f[n] = -f[-n]$. For a signal of length N (and N is odd), in the context of a signal and its DFT, the signal is even if $f[n] = f[N-n]$ and odd if $f[n] = -f[N-n], 1 \leq n \leq N-1$. The DFT properties show that for real input signals, the magnitude spectrum is even and the phase spectrum is odd. Furthermore, when the input signal is both real and even, the DFT is real valued, with an even magnitude spectrum and a phase spectrum that is zero. In summary, if f is an input signal of length N (N is odd) and $F = \text{DFT}(f, N)$, then for $1 \leq k \leq N-1$

- If f is real, $|F[k]| = |F[N-k]|$ and $<F[k] = - <F[N-k]$
- If f is real and even, $|F[k]| = |F[N-k]|$ and $<F[k] = 0$

Positive half of the DFT spectrum: Audio signals are real signals. Due to the symmetry properties of the DFT of a real signal, it is sufficient to store only one half of the magnitude and phase spectrum. For the case of even length DFT, we also need to store just the first $(N/2)+1$ samples for a perfect reconstruction of the signal from its DFT. To save on both storage and computation, we will just store just the half spectrum when possible.

From an N point DFT (N even), we can obtain the positive half of the spectrum by considering only the first $(N/2)+1$ samples of the DFT. We can compute the magnitude spectrum of the positive half (in dB) as $m_X = 20\log_{10}|X[0:(M/2)+1]|$, where X is the DFT of the input.

Zero phase windowing: Zero phase windowing of a frame of signal puts the centre of the signal at the 'zero' time index for DFT computation. By moving the centre of the frame to zero index by a circular shift, the computed DFT will not have the phase offset which would have otherwise been introduced (recall that a shift of the signal causes the DFT to be multiplied by a complex exponential, which keeps the magnitude spectrum intact but changes the phase spectrum). When used in conjunction with zero-padding, zero phase windowing is also useful for the creation of a frame of length of power of 2 for FFT computation (fftbuffer).

If the length of the signal x is M and the required DFT size is N, the zero phase windowed version of the signal (dftbuffer) for DFT computation can be obtained as (works for both even and odd M):

```
hM1 = floor((M+1)/2)
hM2 = floor(M/2)
dftbuffer = zeros(N)
dftbuffer[:hM1] = x[hM2:]
dftbuffer[-hM2:] = x[:hM2]
```

Filtering: Filtering involves selectively suppressing certain frequencies present in the signal. Filtering is often performed in the time domain by the convolution of the input signal with the impulse response of a filter. The same operation can also be done in the DFT domain using the properties of DFT, by multiplying the DFT of the input signal by the DFT of the impulse response of the filter. In this assignment, we will consider a very simple illustrative filter that suppresses some frequency components by setting some DFT coefficients to zero. It is to be noted that the convolution operation here is circular convolution with a period N, the size of the DFT.

If $x_1[n] \Leftrightarrow X_1[k]$ and $x_2[n] \Leftrightarrow X_2[k]$, $x_1[n] * x_2[n] \Longleftrightarrow X_1[k]\, X_2[k]$

Part-1: Minimize energy spread in DFT of sinusoids (*2 points*)

Complete the function `minimizeEnergySpreadDFT(x, fs, f1, f2)` in the file `A3Part1.py` that selects the first M samples from a given signal consisting of two sinusoids and returns the positive half of the DFT magnitude spectrum (in dB), such that it has only two non-zero values.

M is to be calculated as the smallest positive integer for which the positive half of the DFT magnitude spectrum has only two non-zero values.

The input arguments to this function are the input signal x (of length $W \geq M$) consisting of two sinusoids of frequency $f1$ and $f2$, the sampling frequency *fs* and the value of frequencies $f1$ and $f2$. The function should return the positive half of the magnitude spectrum m_X. For this question, you can assume the input frequencies $f1$ and $f2$ to be positive integers and factors of fs, and that M is even.

Due to the precision of FFT computation, the zero values of the DFT are not zero but very small values less than 10^{-12} (or -240 dB) in magnitude. For practical purposes, all values with absolute value less than 10^{-6} (or -120 dB) can be considered to be zero.

For an input `x` sampled at 10000 Hz and composed of sinusoids of frequency 80 Hz and 200 Hz, the DFT size for the required condition is `M` = 250 samples and the non-zero values in the DFT spectrum are at bin indices 2 and 5 (corresponding to the frequency values of 80 and 200 Hz, respectively). The output `mX` that the function returns is 126 samples in length.

```
def minimizeEnergySpreadDFT(x, fs, f1, f2):
    """
    Inputs:
        x (numpy array) = input signal
        fs (float) = sampling frequency in Hz
        f1 (float) = frequency of the first sinusoid
                     component in Hz
        f2 (float) = frequency of the second sinusoid
                     component in Hz
    Output:
        The function should return
        mX (numpy array) = The positive half of the DFT spectrum
                           of the M sample segment of x.
                           mX is (M/2)+1 samples long
                           (M is to be computed)
    """
    ## Your code here
```

Part-2: Optimal zero-padding (*2 points*)

Complete the function `optimalZeropad(x, fs, f)` in the file `A3Part2.py` that given a sinusoid, computes the DFT of the sinusoid after zero-padding and returns the positive half of the magnitude spectrum (in dB). Zero-padding needs to be done such that one of the bin frequencies of the DFT coincides with the frequency of the sinusoid. Choose the minimum zero-padding length for which this condition is satisfied.

The input arguments are the sinuosid x of length W, sampling frequency fs and the frequency of the sinusoid f. The output is the positive half of the magnitude spectrum mX computed using the M point DFT ($M >= W$) of x after zero-padding x to length M appropriately as required. For this exercise, you can assume that the frequency of the sinusoid f is a positive integer and a factor of the sampling rate fs, and that M will be even.

Due to the precision of FFT computation, the zero values of the DFT are not zero but very small values less than 10^{-12} (or -240 dB) in magnitude. For practical purposes, all values with absolute value less than 10^{-6} (or -120 dB) can be considered to be zero.

For a sinusoid `x` with `f` = 100 Hz, `W` = 15 samples and `fs` = 1000 Hz, you will need to zero-pad by 5 samples and compute an `M` = 20 point DFT. In the magnitude spectrum, you can see a maximum value at bin index 2 corresponding to the frequency of 100 Hz. The output `mX` you return is 11 samples in length.

```
def optimalZeropad(x, fs, f):
    """
    Inputs:
        x (numpy array) = input signal of length W
        fs (float) = sampling frequency in Hz
        f (float) = frequency of the sinusoid in Hz
    Output:
        The function should return
        mX (numpy array) = The positive half of the DFT spectrum
               of the M point DFT after zero-padding input x
               appropriately (zero-padding length to be computed).
               mX is (M/2)+1 samples long
    """
    ## Your code here
```

Part-3: Symmetry properties of the DFT (*3 points*)

Complete the function `testRealEven(x)` in the file `A3Part3.py` that checks if the input signal is real and even using the symmetry properties of its DFT. The function will return the result of this test, the zero-phase windowed version of the input signal (dftbuffer), and the DFT of the dftbuffer.

Given an input signal x of length M, do a zero phase windowing of x without any zero-padding (a dftbuffer, on the same lines as the fftbuffer in sms-tools). Then compute the M point DFT of the zero phase windowed signal and use the symmetry of the computed DFT to test if the input signal x is real and even. Return the result of the test, the dftbuffer computed, and the DFT of the dftbuffer.

The input argument is a signal x of length M. The output is a tuple with three elements (`isRealEven`, `dftbuffer`, `X`), where '`isRealEven`' is a boolean variable which is True if x is real and even, else False. `dftbuffer` is the M length zero phase windowed version of x. X is the M point DFT of the `dftbuffer`.

To make the problem easier, we will use odd length input sequence in this question (M is odd). Due to the precision of the FFT computation, the zero values of the DFT are not zero but very small values less than 10^{-12} (or -240 dB) in magnitude. For practical purposes, all values with absolute value less than 10^{-6} (or -120 dB) can be considered to be zero. Use an error tolerance of 1e-6 on the linear scale to compare if two floating point arrays are equal.

If `x = np.array([ 2, 3, 4, 3, 2 ])`, which is real and even, the function returns (`True, array([ 4., 3., 2., 2., 3.]), array([14.0000+0.j, 2.6180+0.j, 0.3820+0.j, 0.3820+0.j, 2.6180+0.j])`). The values shown here are approximate and correct to four decimal places.

```
def testRealEven(x):
    """
    Inputs:
        x (numpy array)= input signal of length M (M is odd)
    Output:
        The function should return a tuple (isRealEven, dftbuffer, X)
        isRealEven (boolean) = True if the input x is real and even,
                            and False otherwise
        dftbuffer (numpy array, possibly complex) = The M point zero
                            phase windowed version of x
        X (numpy array, possibly complex) = The M point DFT
                                            of dftbuffer
    """
    ## Your code here
```

Part-4: Suppressing frequency components using DFT model (*3 points*)

Complete the function `suppressFreqDFTmodel(x, fs, N)` in the file `A3Part4.py` that uses the `dftModel` functions to suppress all the frequency components till the first bin $\geq$ 70 Hz from a frame of signal and returns the output of the `dftModel` with and without filtering.

You will use the DFT model to implement a very basic form of filtering to suppress frequency components. When working close to mains power lines, there is a 50/60 Hz hum that can get introduced into the audio signal. You will try to remove that using a basic DFT model based filter. You will work on just one frame of a synthetic audio signal to see the effect of filtering.

You can use the functions `dftAnal` and `dftSynth` provided by the `dftModel` file of sms-tools. Use `dftAnal` to obtain the magnitude spectrum (in dB) and phase spectrum of the audio signal. Set the values of the magnitude spectrum that correspond to frequencies $\leq$ 70 Hz to -120dB (there may not be a bin corresponding exactly to 70 Hz, choose the nearest bin of equal or higher frequency than 70 Hz). The log of zero is not well defined and hence we set it to -120 dB instead of zero. Use `dftSynth` to synthesize the filtered output signal and return the output. The function should also return the output of `dftSynth` without any filtering (without altering the magnitude spectrum in any way). You will use a hamming window to smooth the signal. Hence, do not forget to scale the output signals by the sum of the window values (as done in `sms-tools/software/models_interface/dftModel_function.py`).

To understand the effect of filtering, you can plot both the filtered output and non-filtered output of the `dftModel`. Please note that this question is just for illustrative purposes and filtering is not usually done this way - such sharp cutoffs introduce artifacts in the output.

The input is a M length input signal x that contains undesired frequencies below 70 Hz, sampling frequency fs and the FFT size N. The output is a tuple with two elements (`y`, `yfilt`), where `y` is the output of `dftModel` with the unaltered original signal and `yfilt` is the filtered output of the `dftModel`.

For an input signal with 40 Hz, 100 Hz, 200 Hz, 1000 Hz components, `yfilt` will only contain 100Hz, 200Hz and 1000 Hz components.

```
def suppressFreqDFTmodel(x, fs, N):
    """
    Input:
        x (numpy array) = input signal of length M (odd)
        fs (float) = sampling frequency (Hz)
        N (positive integer) = FFT size
    Output:
        The function should return a tuple (y, yfilt)
        y (numpy array) = Output of the dftSynth() without
                          filtering (M samples long)
        yfilt = Output of the dftSynth() with filtering
               (M samples long)
    The first few lines of the code have been written for you,
    do not modify it.
    """
    M = len(x)
    w = get_window('hamming', M)
    outputScaleFactor = sum(w)

    ## Your code here
```

Part-5: FFT size and zero padding (*Optional*)

Complete the function `zpFFTsizeExpt(x, fs)` in the file `A3Part5.py` that takes in an input signal, computes three different FFTs on the input and returns the first 80 samples of the positive half of the FFT magnitude spectrum (in dB) in each case.

This optional exercise is a walk-through example to provide some insights into the effects of the length of signal segment, the FFT size, and zero-padding on the FFT of a sinusoid. The input to the function is `x`, which is 512 samples of a real sinusoid of frequency 110 Hz and the sampling frequency fs = 1000 Hz. You will first extract the first 256 samples of the input signal and store it as a separate variable `xseg`. You will then generate two 'hamming' windows `w1` and `w2` of size 256 and 512 samples, respectively (code provided). The windows are used to smooth the input signal. Use `dftAnal` to obtain the positive half of the FFT magnitude spectrum (in dB) for the following cases:

1. Input signal `xseg` (256 samples), window `w1` (256 samples), and FFT size of 256

2. Input signal `x` (512 samples), window `w2` (512 samples), and FFT size of 512
3. Input signal `xseg` (256 samples), window `w1` (256 samples), and FFT size of 512 (Implicitly does a zero-padding of `xseg` by 256 samples)

Return the first 80 samples of the positive half of the FFT magnitude spectrum output by `dftAnal`.

To understand better, plot the output of `dftAnal` for each case on a common frequency axis. Let `mX1, mX2, mX3` represent the outputs of `dftAnal` in each of the Cases 1, 2, and 3 respectively. You will see that `mX3` is the interpolated version of `mX1` (Zero-padding leads to interpolation of the DFT). You will also observe that the 'mainlobe' of the magnitude spectrum in `mX2` will be much smaller than that in `mX1` and `mX3`. This shows that choosing a longer segment of signal for analysis leads to a narrower mainlobe with better frequency resolution and less spreading of the energy of the sinusoid.

If we were to estimate the frequency of the sinusoid using its DFT, a first principles approach is to choose the frequency value of the bin corresponding to the maximum in the DFT magnitude spectrum. Some food for thought, if you were to take this approach, which of the Cases 1, 2, or 3 will give you a better estimate of the frequency of the sinusoid ?

If the input signal is `x` (of length 512 samples), the output is a tuple with three elements: (`mX1_80, mX2_80, mX3_80`) where `mX1_80, mX2_80, mX3_80` are the first 80 samples of the magnitude spectrum output by `dftAnal` in cases 1, 2, and 3, respectively.

```
def zpFFTsizeExpt(x, fs):
    """
    Inputs:
        x (numpy array) = input signal (2*M samples long)
        fs (float) = sampling frequency in Hz
    Output:
        The function should return a tuple (mX1_80, mX2_80, mX3_80)
        mX1_80 (numpy array): The first 80 samples of the magnitude
                              spectrum output of dftAnal for Case-1
        mX2_80 (numpy array): The first 80 samples of the magnitude
                              spectrum output of dftAnal for Case-2
        mX3_80 (numpy array): The first 80 samples of the magnitude
                              spectrum output of dftAnal for Case-3

    The first few lines of the code to generate xseg and the windows
    have been written for you, please use it and do not modify it.
    """
    M = len(x)/2
    xseg = x[:M]
    w1 = get_window('hamming',M)
    w2 = get_window('hamming',2*M)
    ## Your code here
```

Grading

Only the first four parts of this assignment are graded and the fifth part is optional. The total points for this assignment is 10.