Implementations of the DFT

We will program here two implementations of the DFT (naive, recursive FFT with decimation-in-time) and then apply the DFT (or rather, the DCT variant) to the compression of music files. Download the files `dft.py` and `test_dft.py` on moodle.In this file you can also include the following method to display a list of complex numbers.

```
def printc(L):
    res=[]
    for a in L:
        res.append('{:.4f}'.format(a))
    print(res)
```

1 A naive DFT implementation

We first consider the general problem of evaluating a polynomial P at a given value x (which is a real or complex number). For instance, to evaluate the polynomial $P = 2 + 3X + 5X^2 + 4X^3 + 7X^6$ (encoded in python as the list `[2,3,5,4,7]`) at a value x, we can use the so-called *Horner scheme* :

$$P(x) = 2 + x * (3 + x * (5 + x * (4 + 7 * x))).$$

Question 1. Using the Horner scheme strategy, write a method `eval_poly(L,x)` that takes as parameters a list L and a value x (real or complex number), and returns the evaluation at x of the polynomial corresponding to L. Test your code, for instance `eval_poly([1,4,3],6)` should return 133 and `eval_poly([2,3,1,5,6,4],5+1j)` should return $11047 + 14767j$ (in Python a complex number such as $5 + i$ is written `5+1j`).

Question 2. Write a method `dft_slow(L)` that takes as argument a vector L, and returns a list of complex numbers that gives the DFT of L. (Do not forget to include `from cmath import exp, pi` at the beginning of your file in order to call the exponential function and use the π constant).

To test your function, check that it gives the same result as the fft method from `numpy` (to use it, add `import numpy as np` at the beginning of your file, and then to compute the fft of a list L, call `np.fft.fft(L)`).

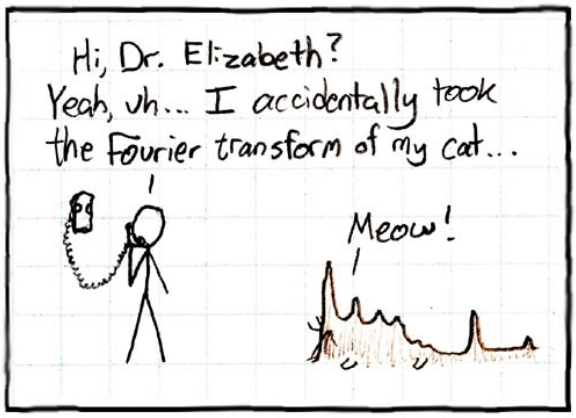

FIGURE 1 – That cat has some serious periodic components. Source : XKCD / CC BY-NC 2.5.

2 The recursive FFT

We will program here the variant "decimation in time" of the recursive FFT, as described in the exercise sheet.

For $n \geq 1$ and $N = 2^n$, and for two vectors $a = [a_0, \ldots, a_{N/2-1}]$ and $b = [b_0, \ldots, b_{N/2-1}]$ both of length $N/2$, we let

$$\tilde{b} = [b_0, b_1 * \omega, \ldots, b_r * \omega^r, \ldots, b_{N/2-1} * \omega^{N/2-1}], \quad \text{where } \omega = \exp(-2i\pi/N)$$

and we define the *FFT-sum* of a and b as the vector of length N obtained by concatenating the vectors $a + \tilde{b}$ and $a - \tilde{b}$. For instance the FFT-sum of $[a_0]$ and $[b_0]$ is $[a_0 + b_0, a_0 - b_0]$ and the FFT-sum of $[a_0, a_1]$ and $[b_0, b_1]$ is $[a_0 + b_0, a_1 - ib_1, a_0 - b_0, a_1 + ib_1]$.

Question 3. Write a method `fft_sum(a,b)` that takes as arguments two lists of a same length that is a power of 2, and returns the FFT-sum of a and b.

Question 4. With the help of the method `fft_sum`, write a recursive method `recursive_fft(L)` (in the 'decimation-in-time' variant) that takes as argument a list L of length N a power of 2, and returns the list of complex numbers of length N that represents the DFT of L. The following Python shortcut is convenient : `L[0::2]` gives the subarray of entries with even index in L, and `L[1::2]` gives the subarray of those with odd index). Check on a few examples that both methods `dft_slow` and `recursive_fft` return the same vectors.

Question 5. Compare the running times of both methods (the FFT should be much faster). You can also compare with the running time of the FFT from `numpy`. The one of `numpy` should be faster (which is due to the fact that it comes from an extension written in C). We recall the `timeit` syntax, for instance `timeit.timeit(lambda:dft_slow([1.3,-2,6,3]), number=1)` outputs the time (in seconds) spent by one execution of `dft_slow([1.3,-2,6,3])` (you should also add `import timeit` at the beginning of the file).

3 Music compression

3.1 Introduction

Music compression heavily relies on Fourier-related transforms, those are the basic blocks of virtually all music formats you know (MP3, WMA, OGG, ...). In this section, we will implement a very naive compression algorithm to illustrate this point.

When your computer or phone plays audio, it convert a digital representation to an analog signal (Digital to Analog Converter, DAC) that is sent to the headphone. This representation is a list of *samples* : instanteanous values of the left and right channels at periodic interval of times. This time interval is given by the *sampling frequency*. Virtually all hardware have a sampling frequency of 44100Hz and represent samples by a 16-bit signed integer, and therefore most music files that are distributed follow this convention [1]. Figure 2 illustrates how your DAC interpolates between the the samples $S_0, S_1, \ldots$ to produce a continuous sound $S(t)$.

The main observation of music compression is that in practice, those samples carry a lot of redundant information because the process that generates them is constrained : it can only produce some frequencies sometimes. Consider the following example, illustrated on Figure 3 of a continuous signal that looks complicated, but is in fact the sum of 5 cosine at 10, 20, 30, 40 and 50Hz. If we were to store the entire signal for half a second as shown, we would need to store $\frac{1}{2}f$ samples, where $f = 44100$ is the sampling frequencies, for a total of $2 \times \frac{1}{2}f = 44100$ integers (each stereo sample is composed of two integers : left and right). On the other hand, we could simply store the frequencies of the 5 cosines and their amplitude for a total of 10 integers, and still be able to reconstruct the signal exactly. In this example, we can compress the signal by a factor over four thousands !

1. See `https://people.xiph.org/~xiphmont/demo/neil-young.html` for a long discussion about why 44100Hz and 16-bit is a very good trade-off.

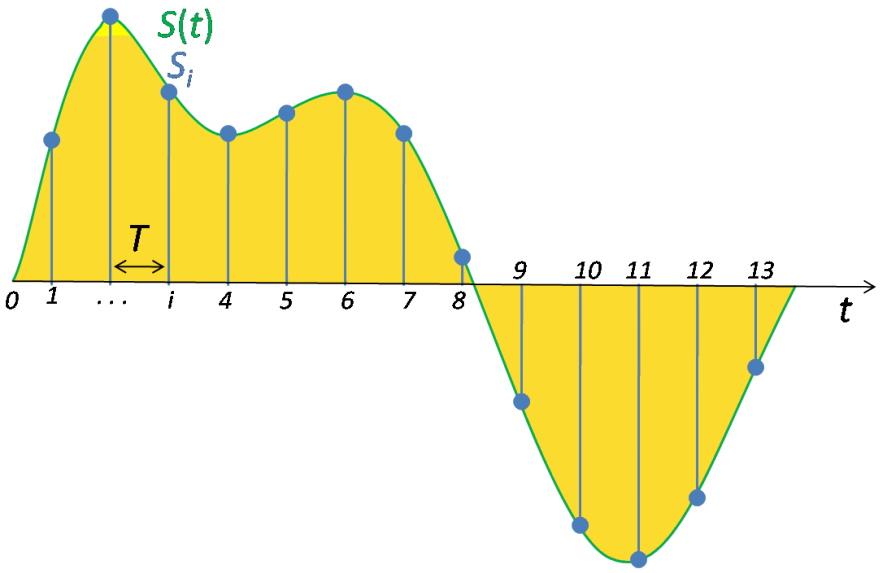

FIGURE 2 – Sampling a continuous signal at discrete times. Source : Wikimedia / Public Domain.

Of course, there are several obstacles to generalize this reasoning to any piece of music. First, we will need many more than 5 cosines to represent any complex sound, but it is unclear how many are needed. Second, any small segment of the music may contain few frequencies, but the entire song can contain a huge variety of sounds, therefore we must split it in many small *chunks* if we hope to decompose it into few components. Lastly, assuming we found a chunk with few different frequencies, how do find them ?

As it turns out, the last point is probably is easiest : decomposing a signal into a sum of sines and cosines is exactly what the DFT does ! As illustrated below, if we run the DFT on our signal with 5 cosines, we will observe [2] five peaks centered on the frequencies, with height proportional to the amplitude.

Tackling the other issues is beyond of the scope of this course, however we will make some simple choices for our experiment : we will split the music into fixed-length chunks, and in each chunk, we will only keep a fixed percentage of the frequencies in the spectrum. In particular, since we are going to "throw away" some components of the music, we will have less data to save (compression) but will also irreversibly change the content (lossy).

3.2 Experiment

Download the `wav.py` file from Moodle : it contains some code to load music in the WAV file format. We also provide you with two samples sample 1.wav and sample 2.wav of music [3]. If you want to experiment with your own music, you will need to convert your MP3/WMA/OGG music to WAV, and probably only keep a small part of it (the above samples last 20 seconds) ; we used Audacity, an open source tool, to do this. The small wav library we provided contains several functions, of which is the most important is

```
process_file_by_chunk(in_file, out_file, compress, decompress, chunk_sz)
```

that performs the following :

1. load `in_file` to memory ;
2. split the music into chunks of size `chunk_sz` ;
3. run the provided function `compress` on each chunk ;

2. Technically, the DFT produces complex numbers, the spectrum graph is generally obtained by taking the norm of the (complex) coefficients.
3. Copyright of music :

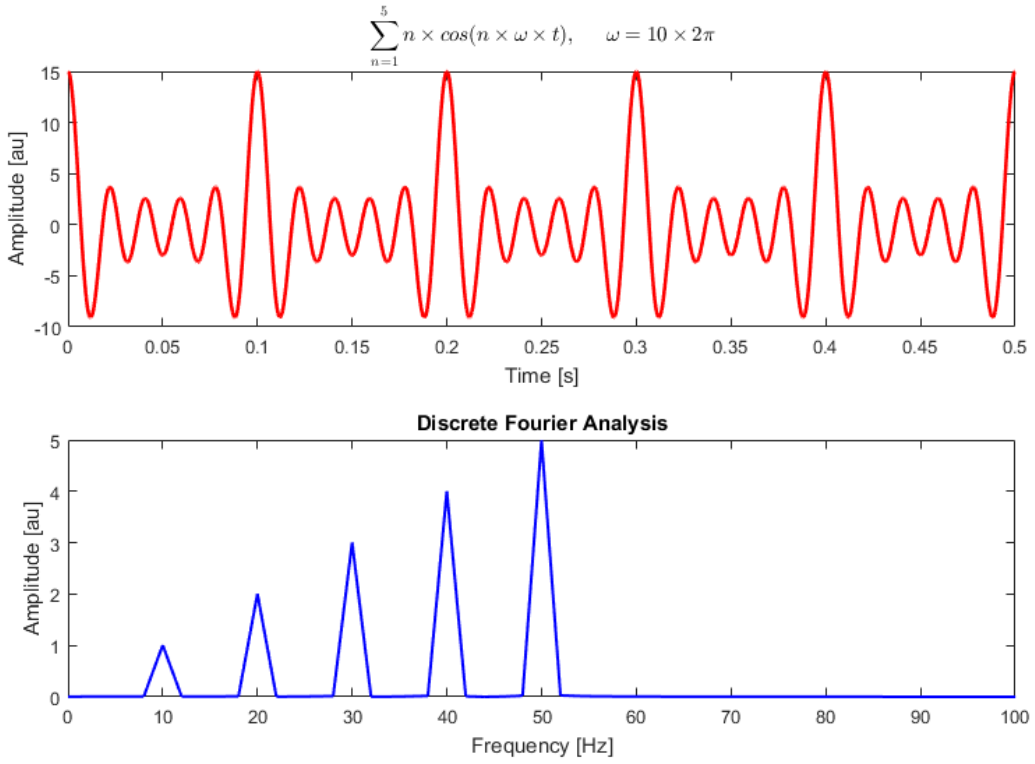

FIGURE 3 – A signal composed of 5 cosines, and its DFT. Source : Wikimedia / CC BY-SA 4.0.

4. run the provided function `decompress` on the output of each chunk ;
5. reconstruct the music and save it to `out_file`.

The following code illustrated how to use this function : our decompress function does nothing, and the compress functions modifies to samples to add a fixed tone to the music.

```
import wav

def add_fixed_tone(samples, position):
    F = 1000 # frequency (Hz)
    A = 2 ** 13 # amplitude (between 0 and 2**15)
    return [samples[i] + A*math.sin(2*(position+i)/44100*math.pi*F)
            for i in range(len(samples))]

def decompress_none(samples, position):
    return samples

print("processing sample 1.wav")
wav.process_file_by_chunk('sample 1.wav', 'sample 1 out.wav',
                          add_fixed_tone, decompress_none, 512)
print("done")
```

Question 6. Run the above example, try changing F to values between 20 and 20000 and listen to how it changes the frequency of the tone. Try lowering the value of A, what do you hear ?

The previous question hopefully convinced you that you can only hear frequencies between 20Hz and 20kHz, but that a low amplitude signal quickly disappears if surrounded by more powerful signals. This justisfies that we can compress a signal by only keeping the signals of highest amplitude : the

difference is almost impossible to hear. The following question shows that it is important to work with frequencies and not samples.

Question 7. Implement a function `compress_decimate2(samples, position)` that returns a list L that contains only half of the samples : $L[i] =$ `samples`$[2i]$, in particular it has half the length of `samples`. Check that it returns `[1,3,5,7]` when run on list `[1,2,3,4,5,6,7,8]`. Implement a function `decompress_decimate2(samples, position)` that does the opposite : $L[i] =$ `samples`$[\lfloor \frac{i}{2} \rfloor]$. Check that it returns `[1,1,3,3,5,5,7,7]` when run on list `[1,3,5,7]`. Try to process a sample file with those (de)compression routines, can you hear the difference ? Implement `compress_decimate4` and `decompress_decimate4` that keep only one sample in four (the size of the list is now a fourth of the original). Can you hear the difference ?

3.3 Discrete Cosine Transform

You are now familiar with the DFT, that allows us to represent a discrete signal of length n by n complex numbers. However, for audio applications, the DFT has several drawbacks : even if the signal is real, it produces complex numbers that are redundant (some coefficients are the conjugate of some others). Furthermore, applying the DFT on chunks and gluing them together introduces some discontinuities that result in audio artifacts. Therefore we prefer to use another Fourier-related transform known as the Discrete Cosine Transform (DCT). Informally [4], the DCT allows us to represent a discrete signal $S_0, \ldots, S_{N-1}$ by a sum of cosines :

$$\frac{1}{2}x_0 + \sum_{n=1}^{N-1} x_n \cos\left(\frac{n\pi}{N}(k+\frac{1}{2})\right)$$

where $x_0, \ldots, x_N$ are the coefficients produced by the DCT. It can be computed in a similar fashion to the DFT, but we are going to use the implementation from `scipy` for speed reasons :

```
import scipy

array = [1,2,3,4,5,6,7,8]
dct = scipy.fftpack.dct(array, norm='ortho') # use norm='ortho'
print('DCT is {}'.format(dct))
res = scipy.fftpack.idct(dct, norm='ortho') # use norm='ortho'
print('IDCT(DCT) is {}'.format(res)) # should be array
```

Question 8. Implement `compress_dct` that applies the DCT on the samples and returns the result. Implement `decompress_idct` that applies the IDCT on the samples and returns the result. Try to process a sample file with those (de)compression routines, and check that you cannot hear the difference (in theory the two files should be exactly the same).

Question 9. Implement a function `simplify` that takes as input a list `L` of tuples `(index, value)` and a percentage `ratio` between 0 and 1. It returns a list `L'` of size `int(ratio*len(L))` that contains the tuples of highest values in L. For example if `L=[(0,42),(2,13),(1,19),(3,25)]` and `ratio=0.5` then `L'=[(0,42),(3,25)]`.

Question 10. Implement `compress_dct_simplify` that applies the DCT on the `samples`, then builds a list of tuples $(i, L[i])$ where L are the coefficients of the DCT. Finally, it calls `simplify` on this list to get S and returns the tuple `(len(samples),S)`. Run it on `[0,1,2,3,3,2,1,0]` with `ratio=0.4` and check that you get approximately `(8, [(0, 4.242), (2, -3.154), (6, -0.224)])`.

Question 11. Implement `decompress_idct_rebuild` that first decomposes its first argument into a tuple `(count, table)` (this is the output of your compress function). It then builds a list `L` of size `count`, and for each tuple `(index,value)` in `table`, sets `L[index]` to `value`. It finally applies the IDCT on `L` and returns the result. Run it on `(8, [(0, 4.242), (2, -3.154), (6, -0.224)])` and check that you get approximately `[-4.50e-05, 0.999, 1.999, 2.999, 2.999, 1.999, 0.999, -4.50e-05]`.

4. There are several types of DCT, here we use the "standard" DCT of type 2, and its inverse of type 3.

Question 12. Try to process a sample file with `compress_dct_simplify` for compress and `decompress_idct_rebuild` for decompress, with `ratio=1`. Can you hear the difference ? (you should not, the file are identical). Try with decreasing values of the ratio : `0.75`, `0.5`, `0.4`, `0.3`, `0.25`, `0.2`, `0.15`, `0.12`, `0.1`. When can you start hearing the difference ? When does the quality degrades significantly ?