STONE ANALYSIS

BOOTSTRAP

STONE ANALYSIS

Exercise 0

Before starting, let's create a wav file with a small sample of a regular sine wave at 440Hz for a duration of 0.2s. You can either use a software for sound editing like Audacity, or generate it yourself using the following formula ($x(t)$ representing each sample):

$$x(t) = M.\sin(2\pi f t)$$

with M the magnitude of the signal (betwenn 0 and 1), f the frequency (here *440Hz*) and t the time in seconds.

The WAV format file is widely documented

Don't forget that a sampling of 48kHz means that we have 48,000 samples of data per second!

The Fourier Analysis

The Discrete Fourier Transform is a mathematical formula that transforms some points from the complex plane into the same amount of other points also on the complex plane.
If we have a sequence x of N samples $(x(0), x(1), ..., x(N-1))$, then the DFT of x is the sequence:

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

This equation can be rewritten by using cos and sin thanks to *Euler's formula*:

$$e^{ix} = \cos(x) + i.\sin(x)$$

Exercise 1

Implement the DFT using this formula with either the exponential or the cos/sin.

Be careful with floating-point precision!

Exercise 2

Now that we applied the DFT on the signal, we have a spectral discrete representation of the sample in the frequency domain.

We can plot the data by having on the x-axis the frequencies (from 0Hz to 24kHz, or narrower) and the y-axis the magnitude (you can see the complex number as a vector, and simply compute its length).
You should see a peak at around 440Hz.

If you try to display a wider range of frequencies by going to the negative, you should see that the graph is symmetric to the y-axis. Can you try to understand why?

Exercise 3

Finally, let's reconstruct back our signal using the Inverse Discrete Fourier Transform!

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

Try to write the data that you just computed into a wav file, and compare it to the original file. You should have exactly the same 440Hz sine wave!

Steganography

Now that we can decompose our signal into a sum of basic sine waves, we can tweak the signal by adding or removing some of the frequencies that compose it.

Exercise 4

The 440Hz frequency represents the A musical note in standard tuning. We are going to tweak our sine wave in order to make an A major chord, which will be composed by the tonic A (440Hz), the mediant C# (554Hz) and the dominant E (660Hz).
We already have the tonic, let's add the mediant and the dominant!

We could simply add the three waves together, compute the signal, and directly write it into the audio file, just like in Exercise 0.
However, this would not work if the original sound is more complex than a simple sine wave of which we already know the frequency.

After doing the DFT on our signal, adding a sine wave is as simple as adding $M \exp(i.\phi)$ to $X(f)$, where f is the frequency, M the magnitude and ϕ the phase

Don't forget that the sequence X is symmetric!

Finally when it is done, you can simply call the IDFT to generate the new signal and voilà! A beautiful A major chord.

Exercise 5

Now that we can shape the signal any way we want, let's focus on the steganography. Because the human ear can only hear sounds that are in the range 20Hz-20kHz, we are going to use *ultrasounds* (sounds with a frequency greater than 20kHz) to cypher our message.

If we try to change the whole signal directly by adding some frequencies, it would be hard to tell in which order they were added.

This is why we may want to decompose our signals into short time windows (50ms for example), each window encoding one letter.

Be careful not to take a too short window with a too low magnitude, otherwise the message may not be decyphered!

How could we encode the number of letters that needs to be read when decyphering?

Going further

You may have noticed, but the DFT gets really slow really fast when the number of data samples grows. Depending on your implementation, it can take up to several minutes to treat a 5 seconds sound.

This is where the Fast Fourier Transform enters the game, as a way to optimize the computations. Indeed, the regular DFT algorithm has an algorithmic complexity of $O(n^2)$, whereas the FFT is considered to be in $O(n \log n)$.

Try to implement an FFT and an IFFT instead, and see the difference!

Another way of improvement may be accomplished by the clever use of *multithreading*.

v1

{EPITECH}