Why Did the Fourier Transform the Mode?
To Get to the Other Spike!

Ankush Kumar

October 19, 2024

1 Motivation

In mathematics, a transformation is a process of converting one quantity into another. There exists all kinds of transformation like basis transformation, similarity transformation, Legendre transformation, etc.
One simple example is in the case of shapes. We know that there are transformations like translation, rotation, reflection, scaling, etc.
In the following sections, we will be looking into one specific type of transformation, the Fourier transform (FT).

FT is heavily used in physics, some common examples are listed below:

- In quantum mechanics, we represent a particle in either the position space or the momentum space. If we are given the position space representation of a particle, how do we convert it into a momentum space representation? FT.
- In solid state physics, we describe a crystal's lattice either in the real space or in the reciprocal space. Given the real lattice of a crystal, how do we convert it into the reciprocal lattice? FT.
- In classical mechanics, we describe waves either in the time domain or in the frequency domain. Given a time domain description of a wave, how do we convert it into a frequency domain description? FT.

Now, one might wonder if we can do this in reverse. For example, if we are given a frequency domain description of a wave can we convert it into a time domain one? The answer is yes and it is done via an inverse Fourier transform (IFT).
When we perform FT (or IFT) on paper/by hand, we use continuous variables. For example, if we are converting a wave from time domain to frequency domain, then we assume that time and frequency increases continuously.
However, a computer cannot do that. If we wish to perform FT using a computer program, then we must consider finite increments in the variables.
This is where, discrete Fourier transform (DFT) comes in.

2 Discrete Fourier Transform (DFT)

The idea of DFT is to take a discrete set of numbers, for example amplitudes of a wave at finite time intervals, and transform them into a new discrete set of numbers, which would represent amplitudes of a wave at discrete frequency intervals in this case.
The recipe for DFT is as follows:

1. Let $\mathbf{x}$ be a set of N numbers given as
$$\mathbf{x} = \{x_0, x_1, x_2, \dots, x_{N-1}\}.$$
We wish to find a new set $\mathbf{X}$ of N numbers given as
$$\mathbf{X} = \{X_0, X_1, X_2, \dots, X_{N-1}\}$$
such that, $\mathbf{X}$ is the DFT of $\mathbf{x}$.

2. This is done by calculating each element of $\mathbf{X}$ as the following:

$$X_k = \sum_{n=0}^{N-1} x_n \cdot e^{-i2\pi \frac{k}{N} n} \quad \text{for each value of } k \tag{1}$$

The set $\mathbf{X}$ obtained from this is defined as the DFT of the set $\mathbf{x}$.

Now, similar to how IFT reverses the effect of FT, we have inverse discrete Fourier transform (IDFT) that reverses DFT.
The recipe for IDFT is described below:

1. Let $\mathbf{X}$ be a set of N numbers, which we have already performed DFT on, given as

$$\mathbf{X} = \{X_0, X_1, X_2, \ldots, X_{N-1}\}.$$

We wish to get back the original set $\mathbf{x}$ of N numbers given as

$$\mathbf{x} = \{x_0, x_1, x_2, \ldots, x_{N-1}\}$$

such that, $\mathbf{x}$ is the IDFT of $\mathbf{X}$.

2. This is done by calculating each element of $\mathbf{x}$ as the following:

$$x_n = \frac{1}{N} \sum_{k=0}^{N-1} X_k \cdot e^{i2\pi \frac{k}{N} n} \quad \text{for each value of } n \tag{2}$$

The set $\mathbf{x}$ obtained from this is defined as the IDFT of the set $\mathbf{X}$.

Notice how the formulas for DFT and IDFT are so similar. This goes to show that the underlying idea is the same in both cases.

2.1 Problem 1

Using Python, implement a function for DFT and one for IDFT. Verify whether or not your functions do the following transformations:

1. $\{1, 2-i, -i, -1+2i\} \xrightarrow{\text{DFT}} \{2, -2-2i, -2i, 4+4i\}$
2. $\{2, -2-2i, -2i, 4+4i\} \xrightarrow{\text{IDFT}} \{1, 2-i, -i, -1+2i\}$

3 Toy Example

Here, we will go through the process of performing DFT on a time domain wave/signal. You are encouraged to code each step as you go through it. Please note the following:

- We will be required to use Python libraries like NumPy and Matplotlib. If you are unfamiliar with them, you are free to seek help online.
- In this section, there will be a few situations where we will do something without an explanation. Please bear with it, if you still want the explanation then you are free to research on your own.

Let's begin.

1. We start by defining what is known as "sampling frequency (`fs`)". It is defined as the number of sample points on a one second long time domain signal.
For example, if we have a signal that is 10 seconds long and we have taken 20,000 sample points on it, then `fs` is 2000Hz.
A general rule of thumb is to take `fs` to be twice as much as the highest frequency present in the signal. This is a statement from the "Nyquist–Shannon sampling theorem."

So, if we are working with sound waves, then we should take `fs` to be about 40kHz, as the highest frequency that humans can hear is about 20kHz.
However, for this toy example, let us take `fs` to be 2000Hz, as we will be working with low frequencies.

2. Next we define two quantities, length of the signal (`T`) and the total number of sampling points taken on the signal (`N`).
   `T` is for our choosing. Let us take `T` to be 0.01 seconds.
   As for `N` ...

3.1 Problem 2

Find an expression for `N` in terms of `fs` and `T`, using the definition of `fs` given in step 1.

3. Next we define two more quantities, time steps for time domain (`time_steps`) and frequency steps for frequency domain (`freq_steps`).
   Here, `time_steps` or `freq_steps` mean the finite increment in time or frequency. (In case of FT or IFT, these steps would have been approximately 0, because we assume continuous increase in time and frequency.)
   For example, if `time_steps` = 0.01s, then that means that time increases like so: 0.00s, 0.01s, 0.02s, and so on. And if `freq_steps` = 0.2Hz, then that means that frequency increases like so: 0.0Hz, 0.2Hz, 0.4Hz, and so on.

   So now, using the definition of `fs`, it is easy to see that `time_steps = 1 / fs`.
   As for `freq_steps` ...

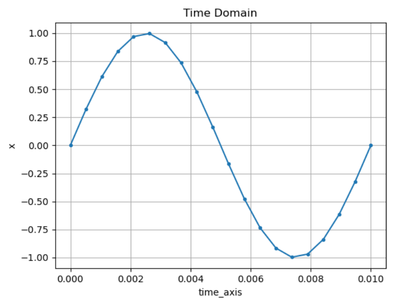
3.2 Problem 3

Find an expression for `freq_steps` in terms of `fs` and `N`.
(Note: You can get a very simple expression for `freq_steps` in terms of `T` but it is better if we define it in terms of `fs` and `N`.)

4. Now comes the part of making use of NumPy.
   We define two arrays `time_axis` and `freq_axis`. They will be used to construct the time domain signal and the frequency domain signal, and to help plot them as well.
   Use the function `linspace` of the NumPy library to construct `time_axis` with a start point of `0`, end point of `N * time_steps`, and `N` number of points.
   This will produce an array of length `N` which contains time values that differ by `time_steps` s, going from 0s to `T` s.
   Do the exact same procedure for `freq_axis`, except switch the `time_steps` to `freq_steps` to produce an array of length `N` which contains frequency values that differ by `freq_steps` s, going from 0Hz to `fs` Hz.

5. Next, we define `x` using the function `sin` of the NumPy library to produce a sine wave time domain function.
   The argument of the function should be of the form `2 * pi * f * time_axis`, where `pi` is π and `f` is the frequency of the sine wave signal. Let us take `f` to be 100Hz for this example.
   So, `x` is an array of length `N` that contains all the amplitudes of the sine wave at different times.

6. With `time_axis` and `x` obtained, we can plot it using the function `pyplot.plot` of Matplotlib. The output should look like the following:



7. Now let us perform DFT on `x` and save it in a new array say `X`.
   Plot `X` w.r.t. `freq_axis` and observe what happens.
   You should observe two things:

   (a) Firstly, the plot does not look like what we would expect. As our input to the DFT function was a simple sine wave of frequency 100Hz, we expected to get a peak/spike at 100Hz alone. But it seems we have got two peaks.

   (b) Secondly, there seems to be a warning message in the terminal.

   Let us tackle these problems one by one:

   (a) According to the Nyquist–Shannon sampling theorem mentioned in step 1, we understand that the `fs` we have chosen would only yield useful result up to `fs / 2`. So in this case, as we have chosen `fs` to be 2000Hz, the results that DFT produces is only useful up to 1000Hz.
   So, we can fix the problem of getting two peaks in the output by simply plotting only the first half of `freq_axis` and `X`.

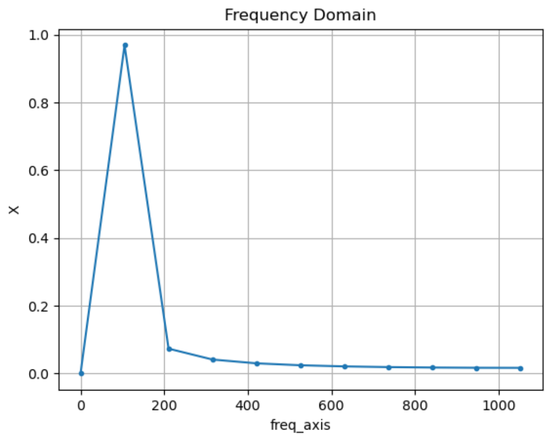
   (b) Even though we are feeding amplitudes of a sine-wave in DFT which are completely real, the outputs seems to be complex. We wish to plot amplitudes in the frequency domain as well which are also completely real.
   So, a simple fix to this problem is to convert all values of `X` into their absolute values/magnitudes. This can be done by using the `abs` function of the NumPy library.

8. Plot `X` w.r.t. `freq_axis` again.
   This time, the general form of the plot will look correct. However, the scaling of `X` axis seems off ...

3.3 Problem 4

Find a correct scaling factor for the `X` axis.
That is, the peak at 100Hz should have an amplitude equal to the amplitude of the sine-wave that we produced, which is 1 for this example.

9. After fixing the scaling of `X`, plot the frequency domain graph again and verify that it looks like the following:



Congratulations, you have successfully performed DFT on a time domain signal!
Now that we know the basics, we can start playing around with our code to discover more things.
The following set of problems helps with it.

3.4 Problems 5-10

1. Currently in our toy example, the peak at 100Hz is not looking very sharp. Let's fix that. Play with the values of `fs` and `T`, and figure out which of them effects the sharpness of the peak and explain why.

2. As we know, sounds in the real world almost always contain more than one frequency in it. So let's add another sine wave in our time domain signal of a different frequency, let's say 500Hz. Perform DFT on this new wave and plot the frequency domain graph. Explain what you see in it.
   Also change the amplitude of this additional wave to a different value, let's say 2, and explain the changes that you get in the frequency domain graph.

3. **Special case**: Add a 0Hz signal (a.k.a. DC component) of a different amplitude than the rest, let's say 3, in the signal and plot the frequency domain graph. Do you observe something wrong in it? If yes, describe the issue and fix it.

4. Now let's make use of that dormant function sitting in our code.
   Perform IDFT on the frequency domain signal that we got in the toy example. Verify that you get the same time domain signal back that we started our toy example with.
   (Note: We are supposed to perform IDFT on the `X` that we get immediately after performing DFT. If we do it after manipulations with scaling factors, we will get wrong results.)

5. Repeat the previous problem with more complicated frequency domain signals, i.e., frequency domain signals that we got after performing DFT on a signal with multiple frequencies. Verify that you get the correct results.

6. Finally, it is time to actually make use of the true power of Fourier transform.
   Create a time domain signal with two amplitudes and two frequencies of your choosing and perform DFT on it.
   Alter the frequency domain signal that you get by removing any one of the frequencies present in it. (Here removable means setting its amplitude to 0.)
   Now perform IDFT on this new frequency domain signal.
   Plot the resulting time domain signal and explain your observations.

4 Main Problem

The normal hearing frequency range of humans is between 20Hz and 20kHz. In music theory, we divide this range into smaller chunks which describe different types of sounds as the following:

Sound Type	**Frequency Range**
Sub-bass	20 - 60 Hz
Bass	60 - 250 Hz
Low midrange	250 - 500 Hz
Midrange	500 - 2000 Hz
Upper midrange	2000 - 4000 Hz
Presence	4000 - 6000 Hz
Brilliance	6000 - 20000 Hz

The problem statement is the following:

Given an audio clip, separate out the 7 sound types from it.

For this problem, use the following audio clip which is taken from the song "theDOGS" by Hiroyuki Sawano: Download Audio File

Please note the following for this problem:

- Other than NumPy and Matplotlib, you will also need the SciPy library for this problem. More specifically, you will need the `io.wavfile.read` and `io.wavfile.write` functions of the SciPy library. Please go through their documentations to understand how to use them.
- When you read the audio file in Python, you will find that it contains amplitudes for two channels. These two channels represent the left and right speakers.
  This is to say that, you will have two time domain waves for one audio file, one for left speaker and the other for the right.
  So, make sure to perform DFT on both and similarly, make sure that your output audio files are also stereo.
- While attempting this problem you will come to realise that the DFT and IDFT algorithms implemented earlier are not good enough, i.e., they are very slow.
  To perform FT on a signal of such a long length and large sampling frequency, we must make use of a faster algorithm for DFT. This algorithm is known as the fast Fourier transform (FFT).
  If you wish to learn FFT and implement it yourself, you are free to do so. (FFT has not been explained here because it requires a good understanding of complex numbers.)
  Otherwise, just substitute your DFT and IDFT functions with the `fft.fft` and `fft.ifft` functions of the NumPy library.