Hard Band-Pass Filter Using FFT

Programming and Synthesis
Roberto Becerra

December 29, 2025

1 Introduction

In this tutorial and assignment, we will code a simple proof-of-concept filter that removes energy from selected frequencies using the **Fast Fourier Transform (FFT)** in Python in real time. This code builds upon the concepts of the Fourier Transform and frequency-domain manipulation.

Just to be explicit, the FFT is an algorithm by which time-domain information in a buffer is transformed into the frequency domain and deposited into a buffer of the same size as the input time-domain signal.

In this text we are dealing with audio signals, so the terms *signal*, *audio*, and *information* may be used interchangeably.

2 A Bit of Context and History (Optional reading)

Most modern implementations of the FFT use an algorithm called **Cooley–Tukey**, named after J. W. Cooley and John Tukey, who proposed its current version around 1965.

In short, this algorithm exploits the internal structure of the *Discrete Fourier Transform* (discrete because it is digital, as opposed to a continuous Fourier Transform). Because of this structure, the algorithm eliminates redundant computations, speeding up the process and making it *Fast*.

2.1 The Discrete Fourier Transform

For the beauty of it, here is the equation of the Discrete Fourier Transform (DFT). You can ignore it for this tutorial if you want, but it may be useful for a deeper understanding.

$$X[f] = \sum_{n=0}^{N-1} x[n]\, e^{-j2\pi \frac{fn}{N}} \tag{1}$$

As stated before:

- $x[n]$ is the signal in the time domain, indexed by sample n
- $X[f]$ is the Fourier transform of $x[n]$, representing the amplitude and phase of frequency f.
- f represents the discrete frequency index

We can say that $x[n]$ is the amplitude of the signal at each moment in time, whereas $X[f]$ represents the amplitude and phase of each frequency component present in the signal.

In summary, this equation scans through sine and cosine waves of all N discrete frequencies $e^{-j2\pi fn/N}$. where f is sequentially incremented corresponding to the f-th position in the X buffer.

This results in all samples $x[n]$ to be multiplied by the corresponding sine and cosine of frequency indexed by f to measure how strongly the signal correlates with that frequency.

Note that the actual frequency is not f (which is just the index), but in fact:

$$frequency = f * \frac{SR}{FFTsize} \tag{2}$$

This process is repeated from $n = 0$ and $f = 0$ to $n = N - 1$ and $f = N - 1$, where N is both:

- the size of the input array x
- the size of the frequency-domain array X

In this context, N is also referred to as the **FFT size**.

3 Back to Our Filter: Your Task

You are given a Python script that computes FFTs on successive chunks of audio in real time. The filter should process every FFT window, select the frequencies we want to keep, and zero out the rest. Then, it will put togetehr all the chunks (buffers) and convert them back to the time domain and output the filtered sound signal.

To do this in real time, the script uses the `sounddevice` library and allows you to select input and output devices.

The code is mostly complete. You only need to modify the section indicated between:

```
########################
# START YOUR CODE HERE #
########################
```

and

```
########################
# END YOUR CODE HERE #
########################
```

You are encouraged to explore the rest of the code to understand how it works.

4 Description of the Code

This chapter describes the code step by step, following the order of execution rather than the top-down structure of the script.

Remember that to run the script you need to first create a virtual environment, install the requirements, and then run it like this:

```
macOS / Linux:
  python -m venv venv
  source venv/bin/activate
  pip install -r requirements.txt
  python spectral_filter.py

Windows:
  python -m venv venv
  venv\Scripts\activate
  pip install -r requirements.txt
  python spectral_filter.py
```

For some Window's installs you may have to use *py* instead of *python*. Stop execution with `Cmd+C` (macOS) or `Ctrl+C` (Windows).

4.1 User Parameters

```
# ========================================================
# USER PARAMETERS
# ========================================================
SAMPLE_RATE = 48000
BLOCK_SIZE  = 1024
FFT_SIZE    = 1024
LOW_CUT     = 2000.0
HIGH_CUT    = 4000.0
GAIN        = 1.0
```

This section initializes the constants used throughout the script. Note that `BLOCK_SIZE` and `FFT_SIZE` are equal here, but do not have to be.

In other applications, the FFT size may be larger than the block size, and the block size may act as a *hop size*. More on that *hop* later.

The values `LOW_CUT` and `HIGH_CUT` define a **band-pass filter**. Everything outside those frequencies should be removed (filtered out).

The Sampling Rate should match that of your audio devices or interfaces. This could be programmed to automatically detect them and be adjusted dynamically. Feel free to explore this.

4.2 Entry Point

```python
# ======================================================
# Entry point
# ======================================================
if __name__ == "__main__":
    try:
        main()
    except KeyboardInterrupt:
        print("\nStopped.")
```

This is the entry point of the script. It executes as entry right after the imports and the top level code (that is, the code without indentation, for example the one from the previous section).

This Entry point only calls one function. The *main* function. The `try/except` structure allows us to gracefully handle interruptions or runtime errors.

4.3 Main Function

```python
# ======================================================
# MAIN
# ======================================================
def main():
    print("\nExplicit FFT Band Filter")
    print(f"FFT size: {FFT_SIZE}")
    print(f"Sample rate: {SAMPLE_RATE} Hz")
    print(f"Pass band: {LOW_CUT} { {HIGH_CUT} Hz\n")

    in_dev, out_dev, in_ch, out_ch = choose_device()

    with sd.Stream(
        samplerate = SAMPLE_RATE,
        blocksize  = BLOCK_SIZE,
        device     = (in_dev, out_dev),
        channels   = (in_ch, out_ch),
        dtype      = "float32",
        callback   = audio_callback,
    ):
        print("\nRunning... Ctrl+C to stop.")
        while True:
            time.sleep(0.5)
```

The `main()` function prints informative messages, allows the user to select audio devices, and starts a real-time audio stream. It obtains information about the input and output audio interfaces with user interaction using the function *choose_device()*. More on that function later. For now suffice to say that we only need the index number of such devices and the number of channels we are using.

Then, a real-time audio stream is started with the parameters defined by the constants that we defined earlier. This audio stream is in fact started in a separate thread that causes its execution to be *non-blocking*. This means that while the audio stream is executed, it is done in parallel, so that it will not *block* the main execution line of the main script process.

This parallel thread will continuously execute the function stated by the parameter *callback*. A *callback* function is a function that is executed when a certain condition is met. The condition in this case is the arrival of a new incoming audio input buffer, which causes the audio thread to *call back* the function that was specified. In this case, the function is called *audio_callback*.

The *while True* part is simply a way to run forever while the audio thread is working.

4.4 Audio Callback: The main DSP part

```python
# ========================================================
# AUDIO CALLBACK (THE WHOLE DSP HAPPENS HERE)
# ========================================================
def audio_callback(indata, outdata, frames, time_info, status):
    """
    This function is called repeatedly by the audio engine.
    Each call processes one block of audio.
    """
```

This function takes as arguments the input buffer *indata*, the buffer *outdata* into which we need to write the audio output, the size in samples of the input data frames, information about the timing of the current frame, and a status flag informative of potential errors in the execution. This is the main part of this script, so we are describing it by parts:

4.4.1 Mix Input to Mono

```python
# --------------------------------------------------
# 1. MIX INPUT TO MONO
# --------------------------------------------------
# indata shape: (frames, channels)
x = np.zeros(frames)

for i in range(frames):
    s = 0.0
    for ch in range(indata.shape[1]):
        s += indata[i, ch]
    x[i] = s / indata.shape[1]
```

All input channels are averaged into a single mono signal.

4.4.2 FFT

```python
# ---------------------------------------------------
# 2. FFT (time domain -> frequency domain)
# ---------------------------------------------------
X = np.fft.rfft(x)
```

The FFT converts the signal from the time-domain to the frequency-domain. Notice the convention of using capital X to denote frequency domain vectors.

4.4.3 Hard Band Filter

```python
# ---------------------------------------------------
# 3. HARD BAND FILTER
# ---------------------------------------------------
X_filtered = np.zeros_like(X)
bin_width = SAMPLE_RATE / FFT_SIZE

for f in range(len(X)):
    freq = f * bin_width

    ########################
    # START YOUR CODE HERE #
    ########################

    ########################
    # END YOUR CODE HERE #
    ########################
```

Here is where the magic happens, and it is in fact where you perform it.

First, a new auxiliary array *X_filtered* is created with the same dimensions as X.

Then, we can deduce the frequency range of every bin (position in the X vector). Because we know that the X buffer can only represent up to the frequency of the sampling rate, which means that the X vector starts at 0hz and ends at SR hz. This means that to know how many Hz fit in every position of X we divide as such:

$$bin\_width = \frac{SAMPLE\_RATE}{FFT\_SIZE} \tag{3}$$

However, we also know that because of aliasing, the second half of the buffer contains redundant information. So, only the first half of the vector is actually useful. This has no consequences in this particular script, but is good to know for future reference.

Now, knowing the “width” of every bin (position in X), we can scan through all the positions of X using a *for* loop, and we can know at which frequency we are at with this equation:

$$freq = f * bin\_width \quad (4)$$

Notice that the for loop uses the letter f as index to scan through the positions *X[f]* of the input buffer, and we will use the same syntax to set the value into the new *X_filtered[f]* position.

Here, you will have to write some code to decide what to do with *X_filtered[k]* **IF** the *freq* is within or without the range specified by LO_CUT and HI_CUT.

Each FFT bin represents a frequency range.

For this task you must:

- Check **if** `freq` lies between `LOW_CUT` and `HIGH_CUT`
- If yes, copy $X[f]$ into $X_{\text{filtered}}[f]$
- Otherwise, assign zero

4.4.4 Inverse FFT

```python
# --------------------------------------------------
# 4. INVERSE FFT (frequency -> time)
# --------------------------------------------------
y = np.fft.irfft(X_filtered)
```

This section performs an Inverse Fast Fourier Transform to the filtered vector, turning it back to a time domain signal, and assigning it to vector y, which is a letter commonly used to denote outputs.

4.4.5 Apply Gain

```python
# --------------------------------------------------
# 5. APPLY GAIN
# --------------------------------------------------
for i in range(len(y)):
    y[i] *= GAIN
```

4.4.6 Write to Output

```python
# ------------------------------------------------
# 6. WRITE TO OUTPUT
# ------------------------------------------------
if outdata.shape[1] == 1:
    for i in range(frames):
        outdata[i, 0] = y[i]
else:
    for i in range(frames):
        outdata[i, 0] = y[i]
        outdata[i, 1] = y[i]
```

Finally, we need to dump the content of y into the *outdata* vector that is expected by the library callback function. This section also duplicates the resulting data to a stereo output if that is what was set from the start.

Note that this is a simple reduction to mono and then duplication to stereo. Feel free to explore how you could handle real stereo or multichannel signals.

4.5 Audio Device Selection

```python
# ====================================================
# AUDIO DEVICE SELECTION
# ====================================================
def choose_device():
    devices = sd.query_devices()

    print("\nAvailable audio devices:")
    for i, d in enumerate(devices):
        print(f"[{i}] {d['name']} | in:{d['max_input_channels']}
         ↪  out:{d['max_output_channels']}")

    in_dev  = int(input("\nInput device index: "))
    out_dev = int(input("Output device index: "))

    in_ch  = int(input("Input channels (1 or 2 recommended): "))
    out_ch = int(input("Output channels (1 or 2 recommended): "))

    return in_dev, out_dev, in_ch, out_ch
```

This helper function allows interactive selection of audio devices for real-time processing.

5 What is that noise I hear?

When you finish implementing this filter you will notice that there is a very clear noise, a sort of constant granulation.

This is the result of the simplistic approach we have coded in here. We are only chopping the audio in consecutive chunks, doing an FFT, removing unwanted frequencies, and then putting those chunks together. The problem is that there is a discontinuity in between those chunks: the end of one buffer does not perfectly blend into the start of the next one.

This is fixed by using so called *windows*, and overlapping them. In other words, instead of gluing together one buffer after the other, we cross-fade between subsequent buffers. Although the cross-fading is not completely straightforward, and we have to apply a certain curve to fade-in and fade-out every buffer before overlapping them. The shape of those fades is the shape of the window. This alone will solve the noise you hear. And the amount by which we overlap them is the so called *hop* size.

We will deal with these *windows* at a later time. But for now feel free and encouraged to implement this solution on your own for an excellent grade.