# Analyzing "The Distance" by Cake: Signal Processing Approach for Enhanced Audio Experience

**Shaun Pies**

**Jul 17, 2023**

# CONTENTS

Signal processing has become an indispensable tool in the world of audio analysis and manipulation. By extracting and manipulating various features of audio signals, it is possible to enhance sound quality, identify specific components, and even create new audio experiences. In this paper, we explore the application of signal processing techniques on the song "The Distance" by Cake, with a focus on the first 30 seconds of the track.

"The Distance" is a highly recognizable and popular alternative rock song that exemplifies Cake's unique musical style. By subjecting this composition to signal processing techniques, we aim to uncover hidden details, enhance audio clarity, and provide a fresh perspective on the song's acoustic properties.

Our approach involves a series of steps, starting with the application of the Fast Fourier Transform (FFT) to transform the time-domain representation of the audio signal into the frequency domain. This transformation enables us to analyze the distribution of frequencies and identify any underlying patterns that may influence the overall sound. By focusing on the first 30 seconds of the song, we can extract and analyze a specific segment that captures the essence of the composition.

To further refine the audio quality and mitigate potential artifacts introduced during the frequency domain analysis, we apply a raised cosine filter. This filter serves to suppress undesired components and emphasize the important frequencies, leading to a cleaner and more focused sound output. The raised cosine filter is selected for its ability to effectively balance between attenuating unwanted frequencies and preserving the desired ones.

After applying the raised cosine filter, we perform the inverse Fourier Transform to convert the filtered frequency-domain signal back to the time domain. This step reconstructs the processed audio, preserving the key characteristics while incorporating the improvements obtained through the frequency domain manipulation.

Finally, we evaluate the effectiveness of our signal processing approach by playing the processed audio and comparing it to the original version of "The Distance." By conducting a comprehensive subjective and objective analysis, we aim to demonstrate the enhanced audio experience resulting from our signal processing techniques.

Overall, this paper showcases a novel application of signal processing on the song "The Distance" by Cake. By employing the Fourier Transform, raised cosine filtering, and inverse Fourier Transform, we aim to unlock hidden details and improve the audio quality of the first 30 seconds of the composition. Through our analysis, we seek to provide a fresh perspective on the song's acoustic properties and offer a valuable contribution to the field of audio signal processing.

In addition to its cultural significance, "The Distance" by Cake holds relevance to scientific research within the field of audio analysis. The selection of this particular song for our signal processing study is not arbitrary; it represents an opportunity to explore the potential of applying advanced techniques to a well-known and widely appreciated musical composition.

Scientific research often benefits from real-world examples that resonate with a broad audience. By utilizing a popular song like "The Distance," we can engage both scientific and non-scientific communities in the exploration of signal processing methodologies. This approach allows us to bridge the gap between scientific analysis and the general public's familiarity with the chosen musical piece, fostering greater interest and understanding.

Furthermore, "The Distance" exhibits various sonic elements that make it an intriguing candidate for signal processing analysis. Its composition encompasses a blend of vocals, instrumental tracks, and complex arrangements that challenge conventional audio processing techniques. By focusing on this song, we can address the unique characteristics and challenges present in its audio representation, ultimately contributing to the development of more robust and adaptable signal processing methodologies.

Moreover, the application of signal processing techniques to popular music opens avenues for interdisciplinary collaborations. Researchers from fields such as musicology, psychology, and computer science can converge to explore how signal processing can influence the subjective experience of music. This multidisciplinary approach not only expands the possibilities within audio analysis but also encourages cross-pollination of ideas and methodologies, leading to novel discoveries and advancements in scientific research.

By acknowledging the importance of "The Distance" in scientific research, we recognize the potential impact of our signal processing study. Through this exploration, we aim to inspire further investigations into the application of advanced audio analysis techniques on well-known musical compositions. Ultimately, our findings may contribute to the development of innovative approaches in signal processing and shape the future of audio analysis in diverse scientific disciplines.

**Disclaimer**

This indroduction was written by ChatGPT for a bit of fun. The real introduction is on the next page.

CHAPTER

ONE

# SIGNAL PROCESSING OF THE SONG "THE DISTANCE" BY CAKE

## 1.1 Introduction

In this project we are going to take the song "The Distance" by the band "Cake" and modify the song with the use of a couple different signal processing techniques. We need to import the song data into python so that it is workable. We then will shorten the song to just the first 30 seconds, take the Fourier Transform of those 30 seconds, apply a filter, take the inverse transform, then plot and play the filtered song.

The first steps in analyzing "The Distance" with Python are to import the necessary packages. Those packages are `numpy`, `matplotlib`, `scipy`, and for playing the audio back to us inside the Jupyter Notebook `IPython`. We will also use the `copy` package from the standard library.

```
# Import necessary modules
import numpy as np
from matplotlib import pyplot as plt
from IPython.display import Audio
from scipy.io import wavfile
from copy import deepcopy
```

## 1.2 Gather Data

First we need to gather the data with the `scipy` module. We will use `scipy.io.wavfile` to read the wave file and sort it into 3 sections. The sampling rate, left channel, and right channel. Then we used the `IPython.display.Audio` class to play the first thirty seconds of the song.

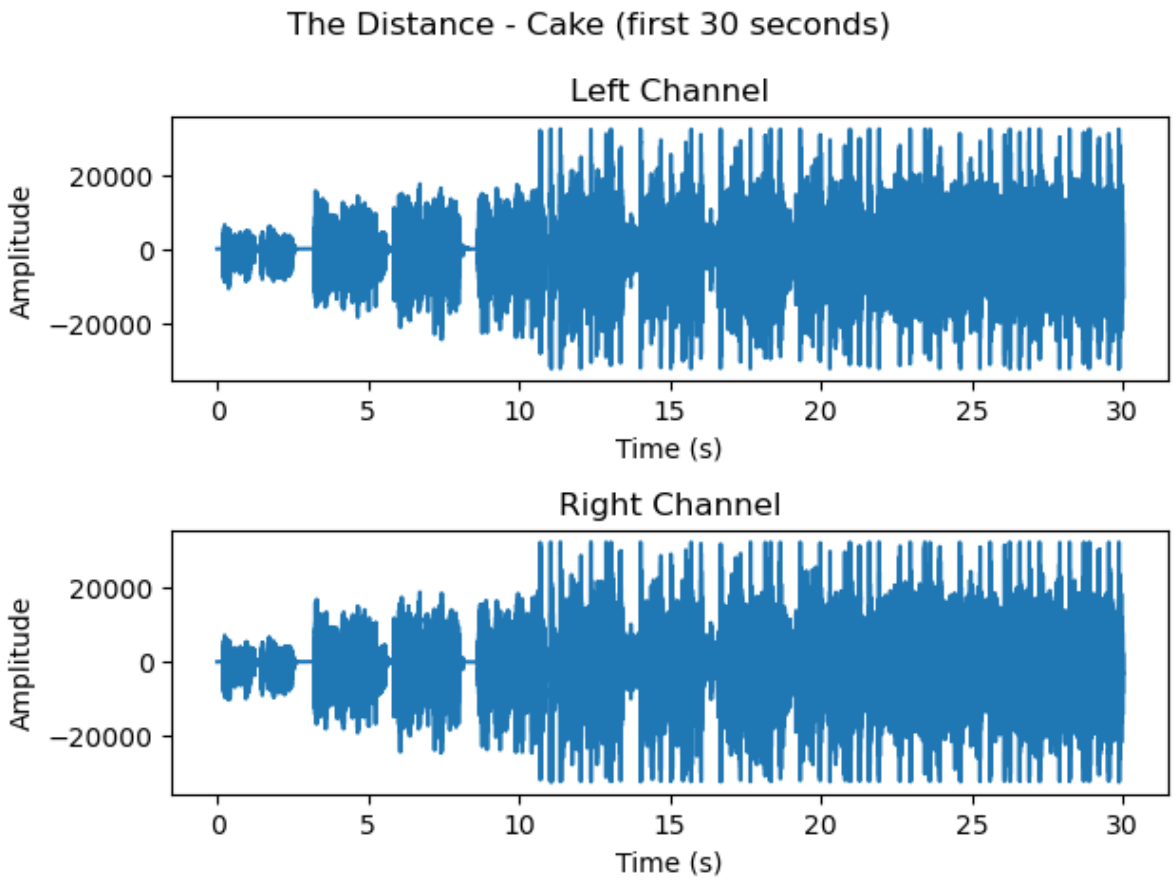
```
# Gather data and play a 30 second segment
song = wavfile.read("TheDistance.wav")
hz = song[0]
dt = 1 / hz
time = np.arange(0, 30, dt)
N = np.size(time)
data = song[1].T
left_channel = data[0][:N]
right_channel = data[1][:N]
channels = [left_channel, right_channel]
Audio(channels, rate=hz)
```

```
<IPython.lib.display.Audio object>
```

## 1.3 Plot the Data

To plot the data we used `matplotlib.pyplot`. The first 30 seconds of each channel are plotted below

```
# Plot the data
fig, ax = plt.subplots(2, 1)
for i, v in enumerate(ax):
    v.plot(time, channels[i])
    v.set_xlabel("Time (s)")
    v.set_ylabel("Amplitude")
ax[0].set_title("Left Channel")
ax[1].set_title("Right Channel")
fig.suptitle("The Distance - Cake (first 30 seconds)")
fig.tight_layout()
plt.show()
```



## 1.4 Fourier Transform

The discrete Fourier Transform is defined as

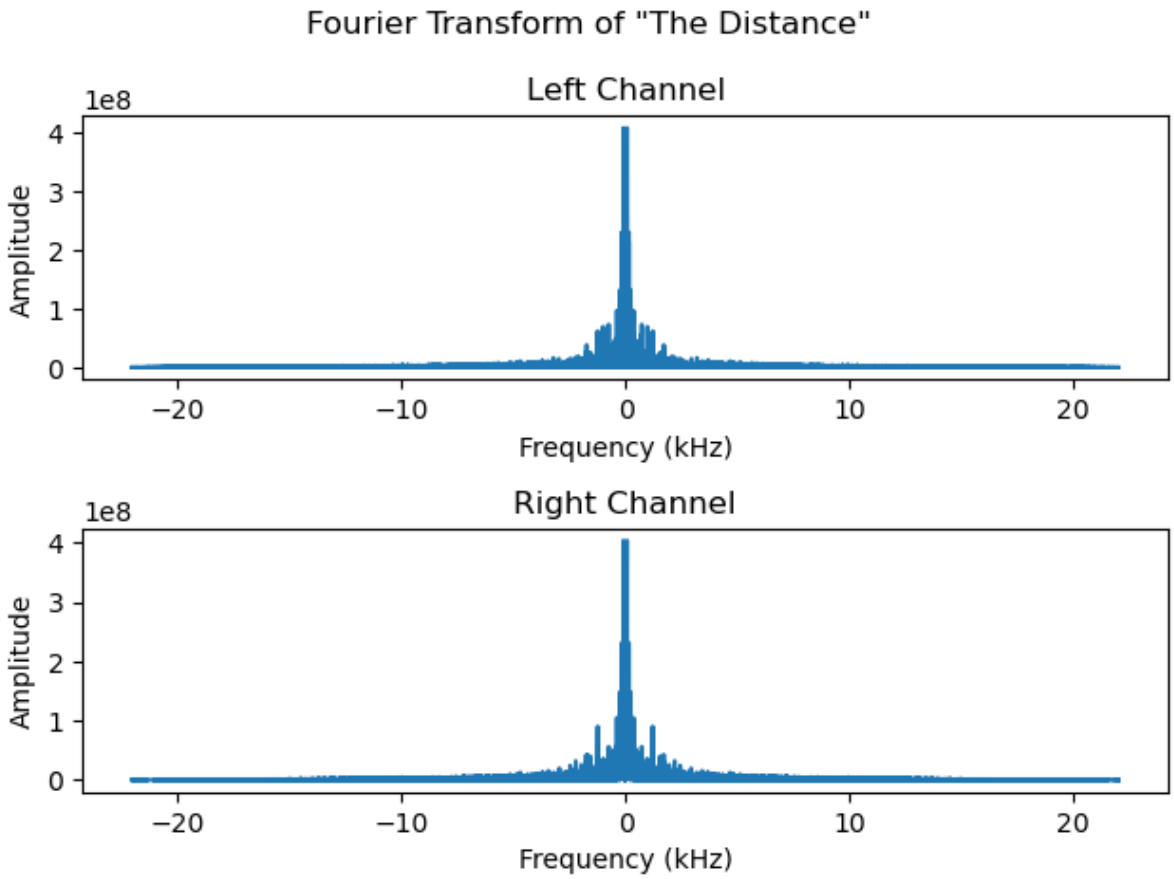
$$F(k) = \sum_{0}^{N-1} f(n) e^{-i2\pi \frac{kn}{N}}$$

An example of the discrete Fourier Transform is shown below. This is a discrete four point Fourier Transform

$$[yay, yay, yay, yay] \supset [4yay, 0, 0, 0]$$

In Python there are several ways to take the Fourier Transform of a set of data. Here we have used the `numpy` package and its various `fft` algorithms. We have plotted the absolute value of the Fourier Transform of each channel below.

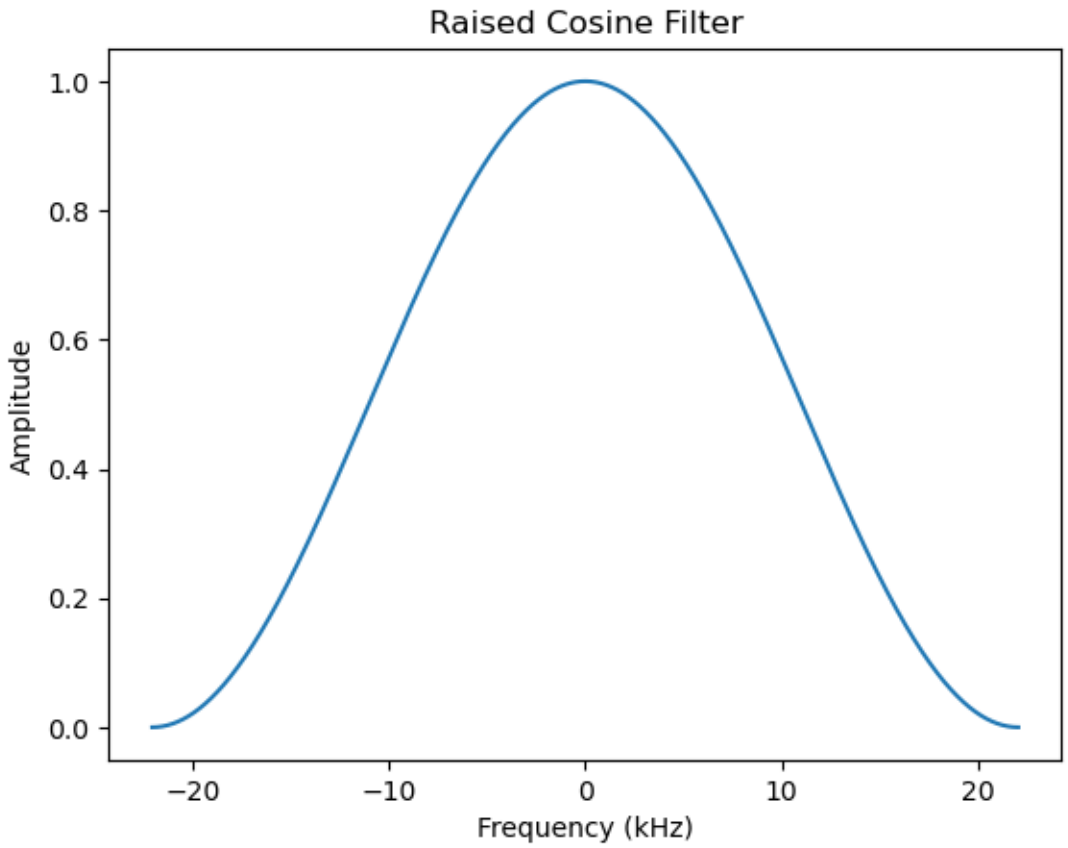
```
# Take the Fourier Transform of each channel and plot it
freqs = np.linspace(-hz / 2000, hz / 2000, N)
ffts = []
for i in channels:
    ffts.append(np.fft.fftshift(np.fft.fft(i)))

fig, ax = plt.subplots(2, 1)
for i, v in enumerate(ax):
    v.plot(freqs, np.abs(ffts[i]))
    v.set_xlabel("Frequency (kHz)")
    v.set_ylabel("Amplitude")
ax[0].set_title("Left Channel")
ax[1].set_title("Right Channel")
fig.suptitle('Fourier Transform of "The Distance"')
fig.tight_layout()
plt.show()
```




## 1.5 Design the Filter

There are several different filters that we could choose to use. A standard rectangle, a Gaussian filter, a Wiener filter and many more. The filter that I chose was a raised cosine filter or also known as a cosine squared filter. This filter has some nice properties. For one it starts at zero and ends at zero which means that there are no sharp edges that can cause some undesired artifacts in the data. It gradually increases in value and at the central value has a maximum amplitude of one then gradually goes back down to zero before the end of the data.

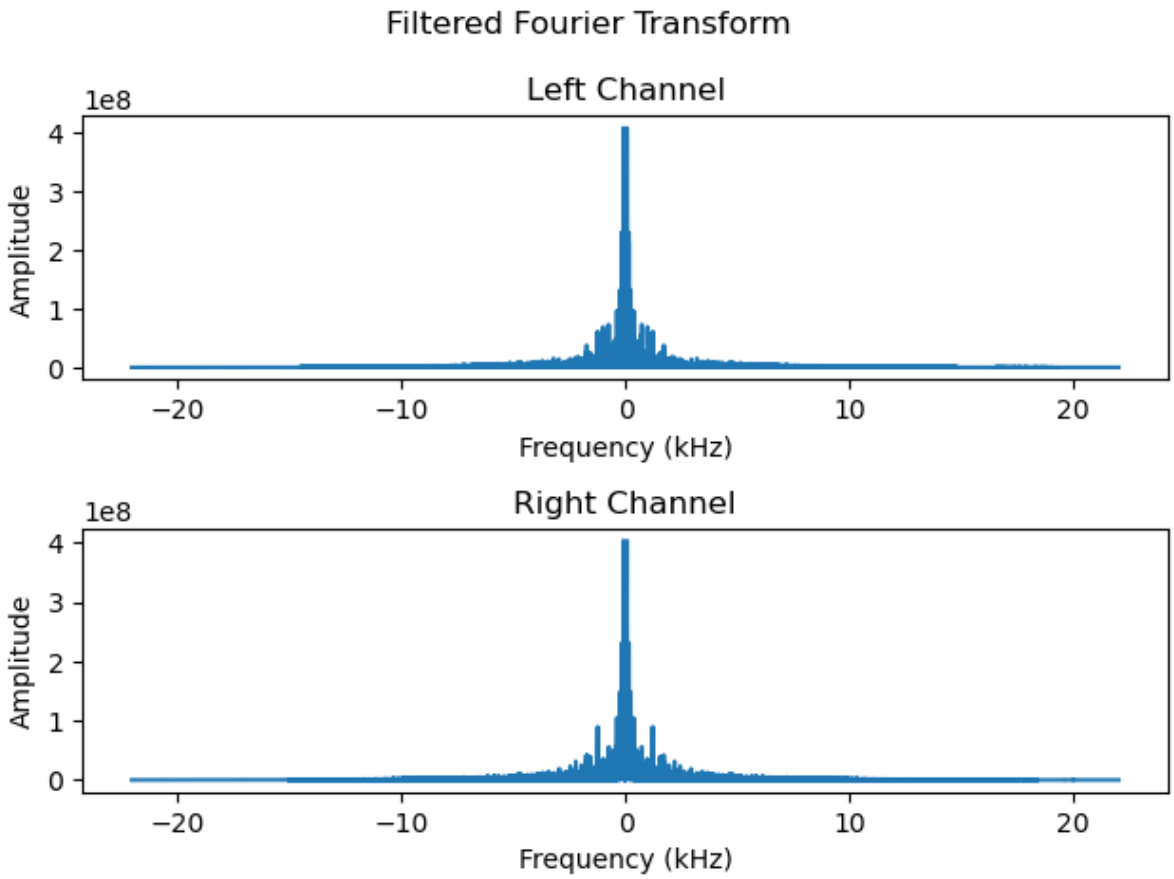
```
# Design the filter
x = np.linspace(-np.pi / 2, np.pi / 2, N)
cos_filt = np.cos(x) ** 2

# Plot the filter
fig, ax = plt.subplots()
ax.plot(freqs, cos_filt)
ax.set_xlabel("Frequency (kHz)")
ax.set_ylabel("Amplitude")
ax.set_title("Raised Cosine Filter")
plt.show()
```




## 1.6 Apply the Filter

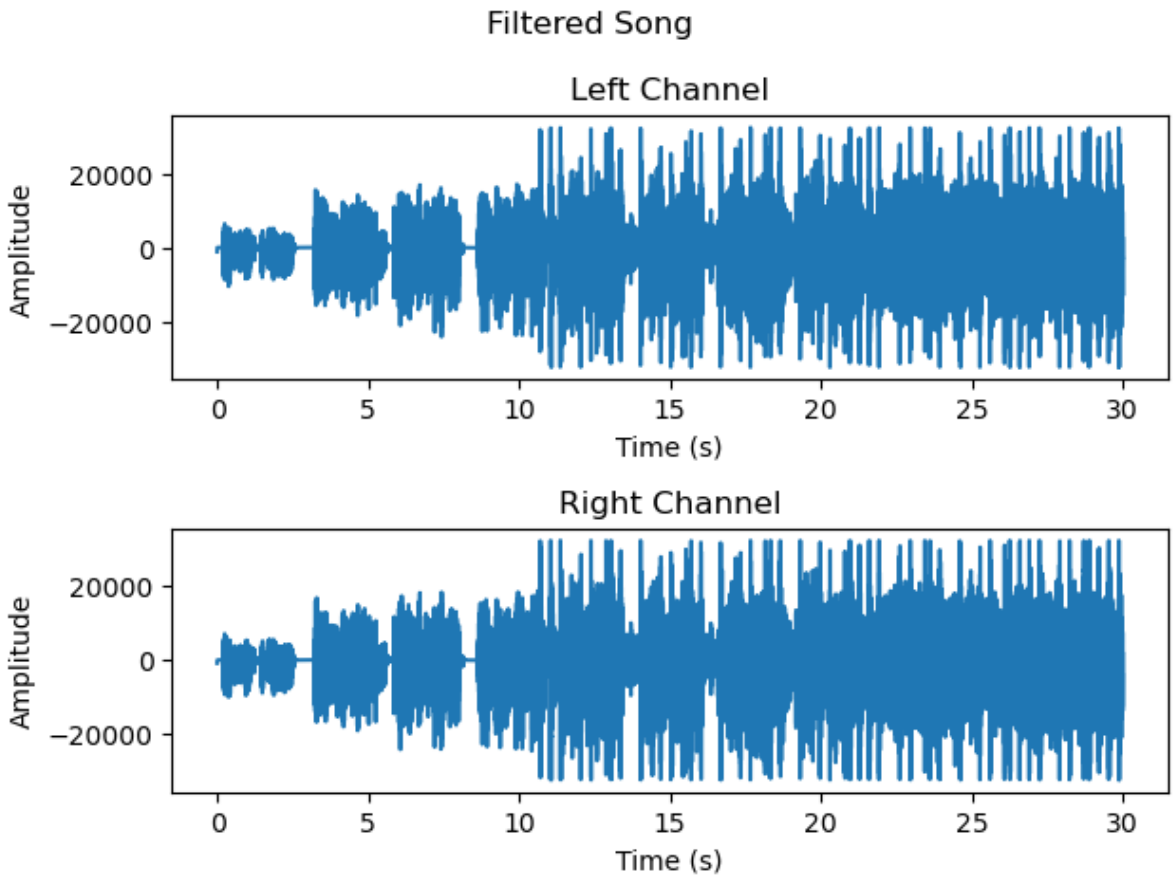
To apply the filter to the transformed data we must do term by term multiplication. Python's default array multiplication is term by term so all that there is to do is multiply the two arrays together (making sure that they are the same length of course).

```
# Apply filter to data
filtered_ffts = deepcopy(ffts)
for i in filtered_ffts:
    i *= cos_filt

# Plot filtered ffts
fig, ax = plt.subplots(2, 1)
for i, v in enumerate(ax):
    v.plot(freqs, np.abs(filtered_ffts[i]))
    v.set_xlabel("Frequency (kHz)")
    v.set_ylabel("Amplitude")
ax[0].set_title("Left Channel")
ax[1].set_title("Right Channel")
fig.suptitle("Filtered Fourier Transform")
fig.tight_layout()
plt.show()
```




Once the filter has been applied to the data we then take the inverse Fourier Transform of the filtered data then we can plot and play the filtered data

```
# Inverse Fourier Transform. Plot. Play.
filtered_song = []
for i in filtered_ffts:
    filtered_song.append(np.fft.ifft(np.fft.fftshift(i)).real)

fig, ax = plt.subplots(2, 1)
for i, v in enumerate(ax):
    v.plot(time, filtered_song[i])
    v.set_xlabel("Time (s)")
    v.set_ylabel("Amplitude")
ax[0].set_title("Left Channel")
ax[1].set_title("Right Channel")
fig.suptitle("Filtered Song")
fig.tight_layout()
plt.show()

Audio(filtered_song, rate=hz)
```




```
<IPython.lib.display.Audio object>
```

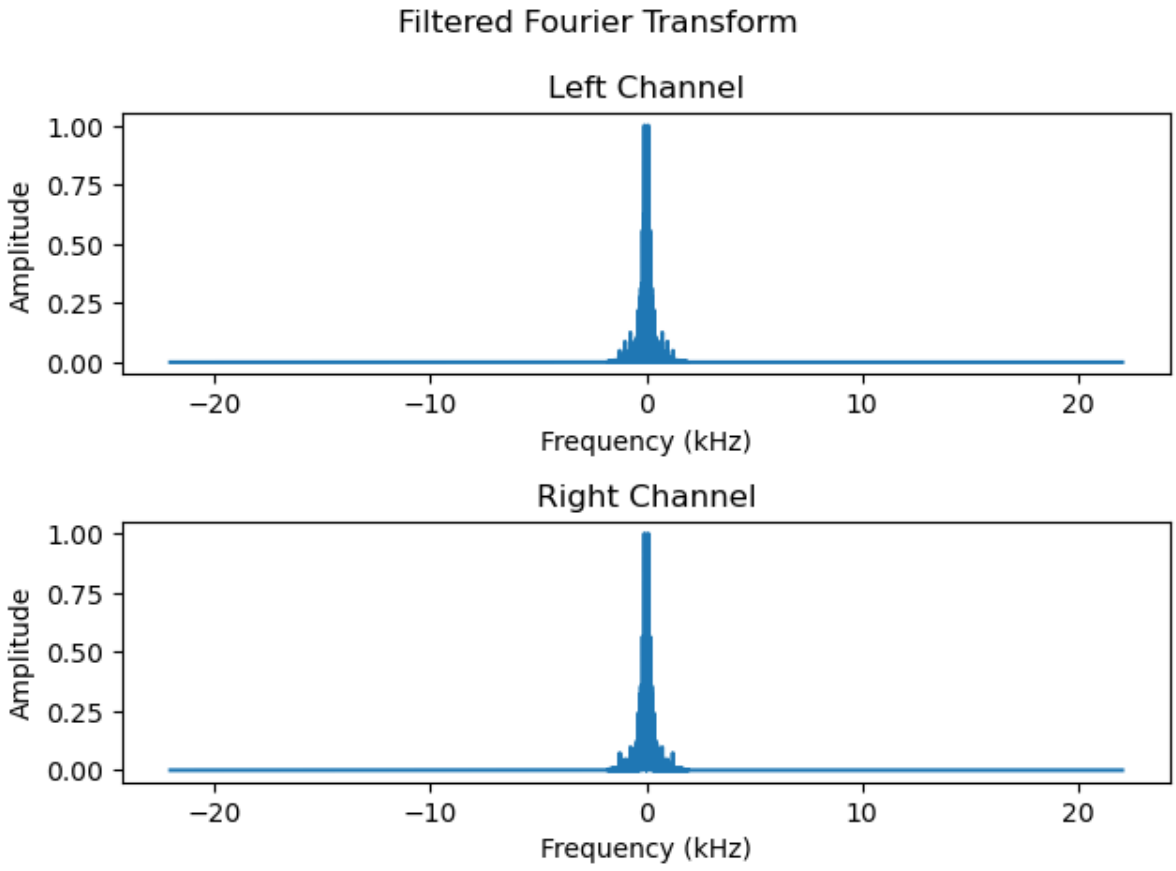
## 1.7 Modified Filter

The raised cosine filter did not seem to affect the data very much since most of the frequency information is within about 10 kHz. So in order to see (and hear) what the filter does I modified the filter so that the filter removes all frequency information above 2 kHz.

```
# Make a smaller filter to remove more data and see what happens
# Find subset of indices where the frequency is between -2000 and 2000
cos_filt = np.zeros(N)
start = int(np.where(freqs >= -2)[0][0])
end = int(np.where(freqs >= 2)[0][0])

size = len(cos_filt[start:end])
x = np.linspace(-np.pi / 2, np.pi / 2, size)
cos_filt[start:end] = (np.cos(x)) ** 2
```

In this case I start with the same size filter. I then find where the frequencies are above -2 kHz and below 2 kHz and use that information to generate the raised cosine filter. Once the filter has been created we can then apply it to the Fourier Transform of the data in the same manner that we did for the first filter.

```
filtered_ffts_2 = deepcopy(ffts)
for i in filtered_ffts_2:
    i *= cos_filt

# Plot filtered ffts
fig, ax = plt.subplots(2, 1)
for i, v in enumerate(ax):
    v.plot(freqs, np.abs(filtered_ffts_2[i]) / np.max(np.abs(filtered_ffts_2[i])))
    # v.plot(freqs, cos_filt)
    v.set_xlabel("Frequency (kHz)")
    v.set_ylabel("Amplitude")
ax[0].set_title("Left Channel")
ax[1].set_title("Right Channel")
fig.suptitle("Filtered Fourier Transform")
fig.tight_layout()
plt.show()
```

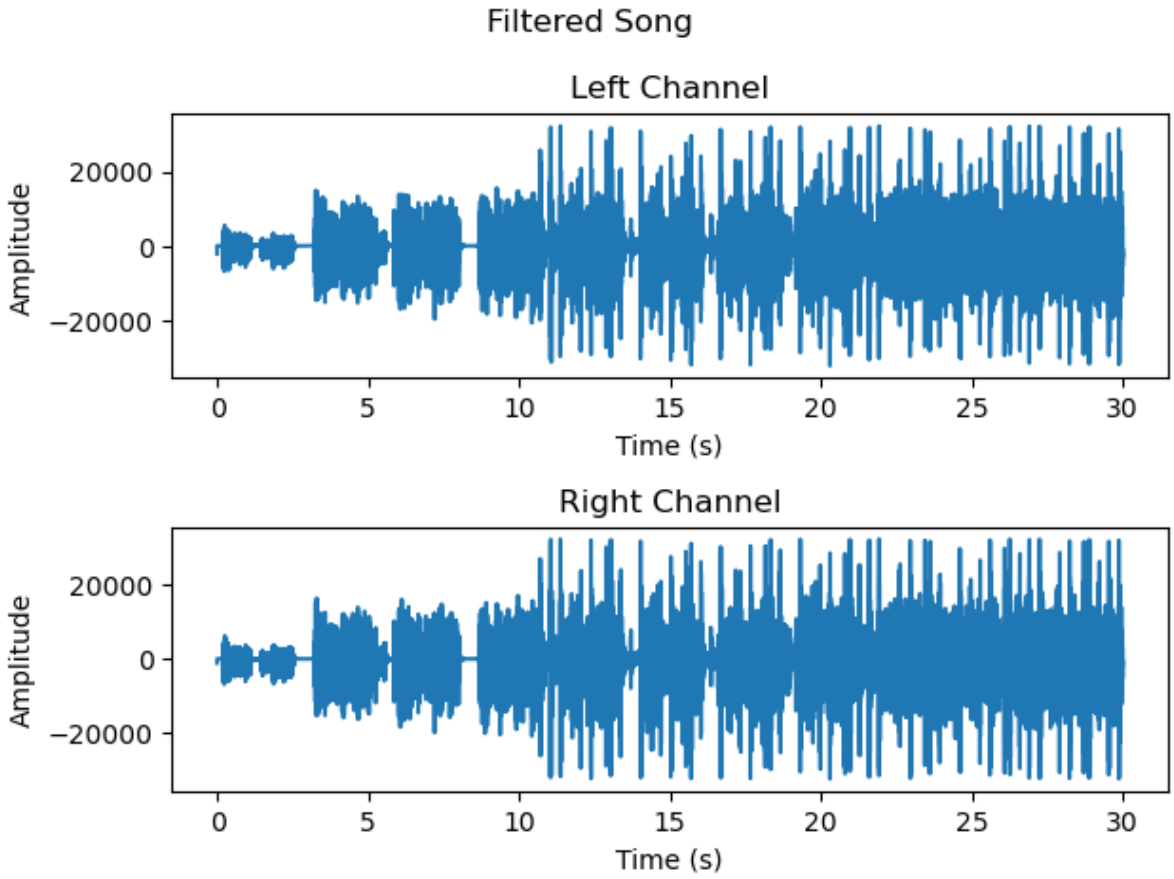



```
# Inverse Fourier Transform. Plot. Play.
filtered_song_2 = []
for i in filtered_ffts_2:
    filtered_song_2.append(np.fft.ifft(np.fft.fftshift(i)).real)

fig, ax = plt.subplots(2, 1)
for i, v in enumerate(ax):
    v.plot(time, filtered_song_2[i])
```

(continues on next page)

(continued from previous page)

```
    v.set_xlabel("Time (s)")
    v.set_ylabel("Amplitude")
ax[0].set_title("Left Channel")
ax[1].set_title("Right Channel")
fig.suptitle("Filtered Song")
fig.tight_layout()
plt.show()

Audio(filtered_song_2, rate=hz)
```




```
<IPython.lib.display.Audio object>
```

Now we can play all three tracks back to back to hear the difference.

```
Audio(channels, rate=hz)
```

```
<IPython.lib.display.Audio object>
```

```
Audio(filtered_song, rate=hz)
```

```
<IPython.lib.display.Audio object>
```

```
Audio(filtered_song_2, rate=hz)
```

```
<IPython.lib.display.Audio object>
```

CHAPTER

# TWO

# CONCLUSION

As can be seen and heard from the three versions of the audio files the regular raised cosine does not alter the song by very much. The modified raised cosine however does modify the song quite a bit. It sounds much more muffled as all the high frequency information has been discarded and all but the DC value have been reduced by the raised cosine filter. Similar results could be assumed if the filter was a Gaussian filter depending on the value of sigma.

Overall this was a really fun project and class in general. I truly feel like I have grown and learned a ton this semester. I have a much better understanding of how to write code and good programming practices as well as a better intuitive understanding of many of the physics concepts that we dealt with in this class. I've also learned to use fewer jokes while naming my variables. I opted to just rewrite everything rather than trying to navigate all my different Portal references in the project part 1 file.