# Real-Time FM Streaming with Captions

CSE 490 W Final Project Report

June 11, 2021

Kavel Rao

# Contents

# List of Figures

# 1 Introduction

## 1.1 Background

This is a project writeup for CSE 490 W at the University of Washington, taught by Professor Joshua Smith in Spring of 2021. The class topic is wireless communication, with a software focus. Through the class, we had access to Software Defined Radio (SDR) devices, which can be used to capture signals in frequency ranges from 55 MHz to 2.3 GHz. In class, we explored FM demodulation using the SDR by capturing a short 2.5 second sample, demodulating it, and playing the audio clip through the computer's speakers. This project is a significant extension of that concept.

## 1.2 Goal

The goal of this project is to create a program that streams an FM signal from a software defined radio (SDR), demodulates the signal, plays back the audio, and outputs captions all in real time. The primary goal of the project is the streaming FM radio player, and the captioning is an additional objective.

# 2 Approach

The radio player has a lot of pieces. This section details the complete process, including sampling from the SDR, demodulating the FM signal, and adding captions.

## 2.1 Program Structure and Multiprocessing

To achieve a real-time radio player, many pieces must be computed simultaneously. While processing a sample, the SDR must be concurrently listening to the next sample, all while the speakers and captions output the previous sample.

To solve this problem, I used python's multiprocessing library, which allows a program to create subprocesses that use multiple CPU cores. This approach significantly speeds up computations and enables the creation of a python real-time radio player.

Separate processes are used for each task listed below.

(a) Sample from the SDR

(b) Demodulate the raw signal into audio data

(c) Play audio through the speakers

(d) Generate and print captions

The main process is used to play audio to simplify speaker access, and all the other steps are done in subprocesses.

In addition to separate processes, the captioning is implemented in a separate script, which imports a Radio class that handles everything else.

To transfer data between processes, I used shared queues from the multiprocessing package. These are global variables that all processes have access to, and I use three in total. The SDR sampling process adds complete samples into a queue, which the audio extraction process gets. Once the signal is processed, the audio data is put into another queue, where it is received by the main process. Before playing the audio, the main process copies the final samples into an "output queue", which is exposed to external programs (like the captioning script). Finally, the captioning process takes from this output queue to run its speech-to-text model and output the captions.

See below for an example of the setup for a subprocess.

Figure 1: Subprocess Example

```
class ExtractionProcess(multiprocessing.Process):
  def __init__(self, f_sps, f_offset, f_audiosps):
      multiprocessing.Process.__init__(self)
      self.f_sps = f_sps
      self.f_offset = f_offset
      self.f_audiosps = f_audiosps

  def run(self):
      while not exitFlag.is_set():
          samples = sample_queue.get(block=True)
          filteredsignal = self.filter_samples(samples, self.f_sps,
              self.f_offset)
          audio = self.process_signal(filteredsignal, self.f_sps,
              self.f_audiosps)
          audio_queue.put(audio)
```

This multiprocessing approach is essential to the real-time component of the radio player. However, there are some challenges that come along with it. The main issue I had was in attempting to create a graceful exit for the program, rather than using keyboard interrupts. See section 3.2 for a detailed description of this issue.

## 2.2 Asynchronous SDR Sampling

The simplest way to get samples from the SDR is to use `sdr.read_samples()`, but this function is not suited for continuous sampling. Instead, I use the asynchronous `sdr.stream()` functionality, which must be handled differently on the client side. See below for the code used to stream samples.

Figure 2: SDR Streaming

```
async def stream_samples(self, sdr, N, f_sps, f_c, f_offset):
    sdr.sample_rate = f_sps
    sdr.center_freq = f_c + f_offset
    sdr.gain = -1.0 # increase for receiving weaker signals
    samples = np.array([], dtype=np.complex64)
    async for sample_set in sdr.stream(): # streams 131072 samples at
        a time
        samples = np.concatenate((samples, sample_set))
        if len(samples) >= N:
            sample_queue.put(samples)
            samples = np.array([], dtype=np.complex64)
```

There is a much simpler way to stream N samples: `sdr.stream(N)`. However, with longer buffer times ($> 3$ seconds), this results in an low level error:

```
Failed to allocate zero-copy buffer for transfer 0
Falling back to buffers in userspace
Failed to submit transfer 3
Please increase your allowed usbfs buffer size
```

To get around this error, I simply concatenate samples together until reaching N samples, then send that batch to the signal processing side.

## 2.3 Signal Demodulation

This is the most involved section of the program, so I'll go through it step by step. A brief overview: the program first shifts and filters samples, then calculates the phase, cleans the signal, and computes the derivative of the angle; finally, it downsamples this to the audio sample rate. For many of the steps, there are more conceptually simple ways to accomplish the same goal, but I've implemented optimizations to enable it to run in real-time.

### 2.3.1 Shifting and Filtering

Because the SDR creates an artificial spike at the center sampling frequency, I sample at an offset of 250 KHz from the tuning frequency. Before extracting the audio data, the desired signal must be shifted back to center. To do this, I multiply the signal by a complex exponential with a period of the offset frequency over the sampling rate.

Figure 3: Shifting Samples

```
1  shift = np.exp(1.0j * 2.0 * np.pi * f_offset / f_sps * np.arange(N))
2  shifted_samples = samples * shift
```

See below for example frequency plots of the original and shifted signals, with a listening frequency of 94.9 MHz.

Figure 4: Offset Samples

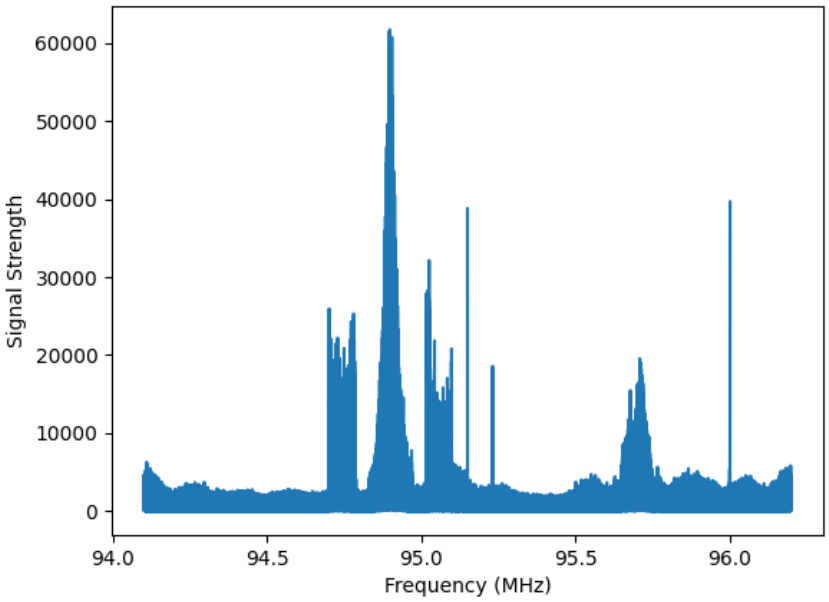
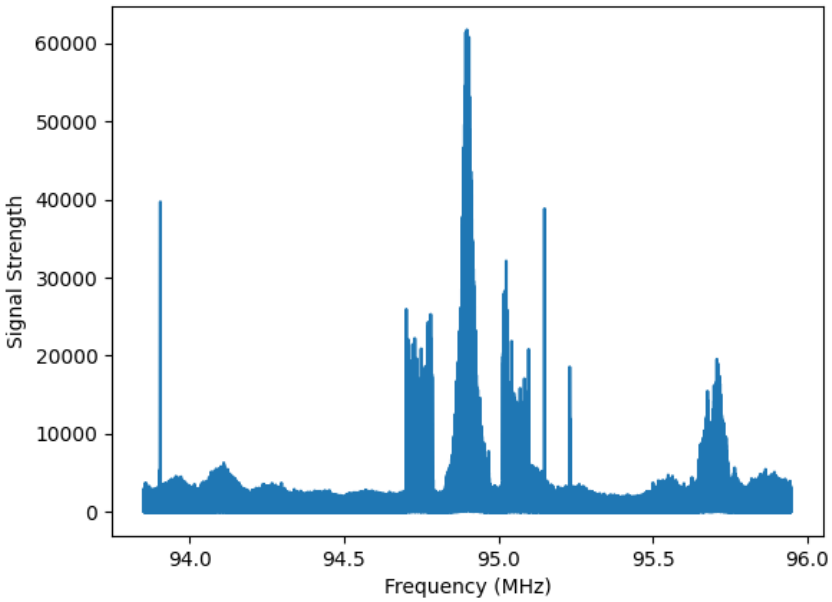
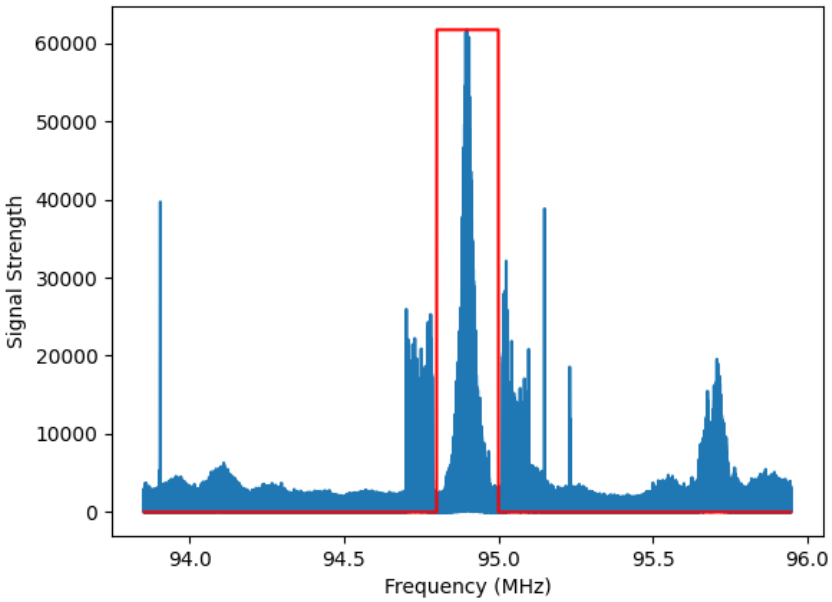


Figure 5: Centered Samples




Once the samples are centered, the next step is to filter out only the signal we want. The actual FM data is in the center spike on the frequency plot, while the outer blocks are in-band on-channel digital sidebands. The easiest filtering step would be multiplying a bandpass mask by the signal in the frequency domain, which is visualized below.

Figure 6: Bandpass Mask Filtering




While this approach is conceptually simple, it requires having a frequency domain representation of the signal. Computing the Fourier Transform is computationally intensive, so for a real-time player it's much better to do all the signal processing in the time domain. Instead, I make use of the convolution theorem, which states that multiplication in the frequency domain equals convolution in the time domain (and vice-versa). Applied to this context, that means that we should be able to filter the signal by convolving it with a sinc function, which is the time domain representation of the bandpass mask (a "top-hat" function). This sinc function would be infinite in the time domain, but we can approximate it by using a finite impulse response filter. This approach and its results are shown below.

Figure 7: Filtering

```
1  k = 201 # number of filter taps - increase this to improve filter
       quality
2  firtaps = signal.firwin(k, cutoff=f_bw, fs=f_sps, window='hamming')
3  filteredsignal = np.convolve(firtaps, shifted_samples, mode='same')
```

Figure 8: Finite Impulse Response Filter

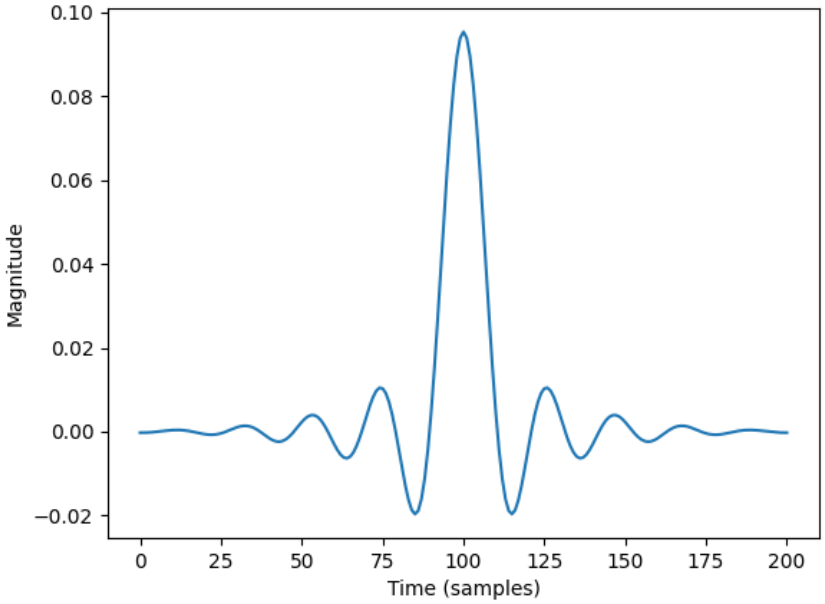
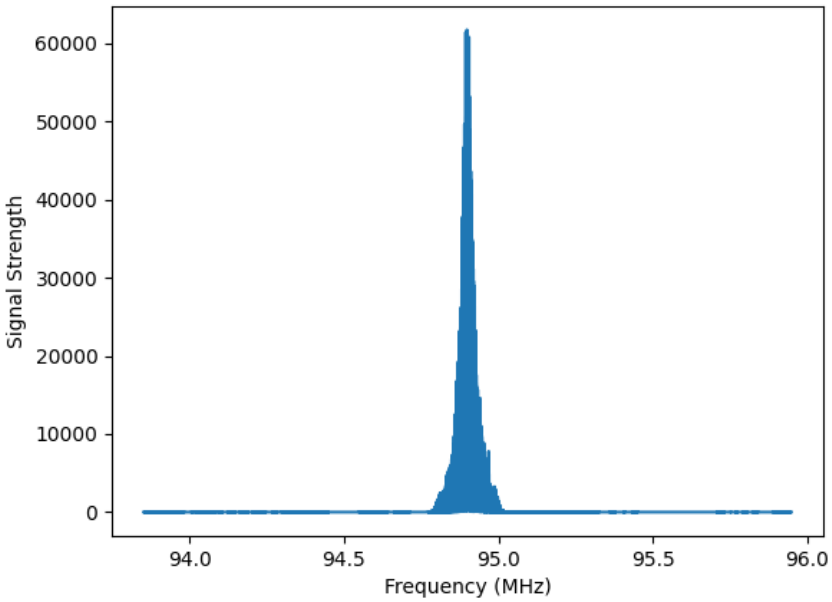
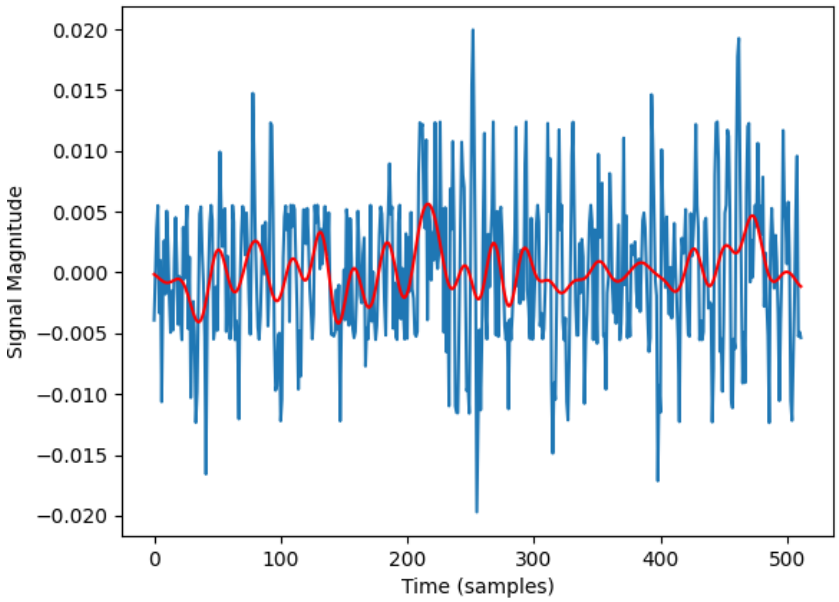


Figure 9: Filtered Signal




The filtering is extremely important, as the signal without the filtering contains a lot of extraneous data. See below for a comparison of the filtered (red) and unfiltered (blue) signal.

Figure 10: Filtered vs Unfiltered Signal




### 2.3.2 Phase Processing

With the filtered signal, the next step is to extract the audio data. In an FM signal, the data is encoded by the instantaneous frequency, which is the derivative of the phase. After computing the phase, I then do a cleaning step called squelching, where low power signal is zeroed out to reduce noise when there's not much data in the signal.

Figure 11: Finding Phase and Squelching

```
1  theta = np.arctan2(filteredsignal.imag, filteredsignal.real)
2
3  # squelch low power signal to remove noise
4  abssignal = np.abs(filteredsignal)
5  meanabssignal = np.mean(abssignal)
6  theta = np.where(abssignal < meanabssignal / 3, 0, theta)
```

See below for a piece of the signal with and without squelching. The squelching

removes high frequency noise from the portion without any data in it.

Figure 12: Non-Squelched Signal

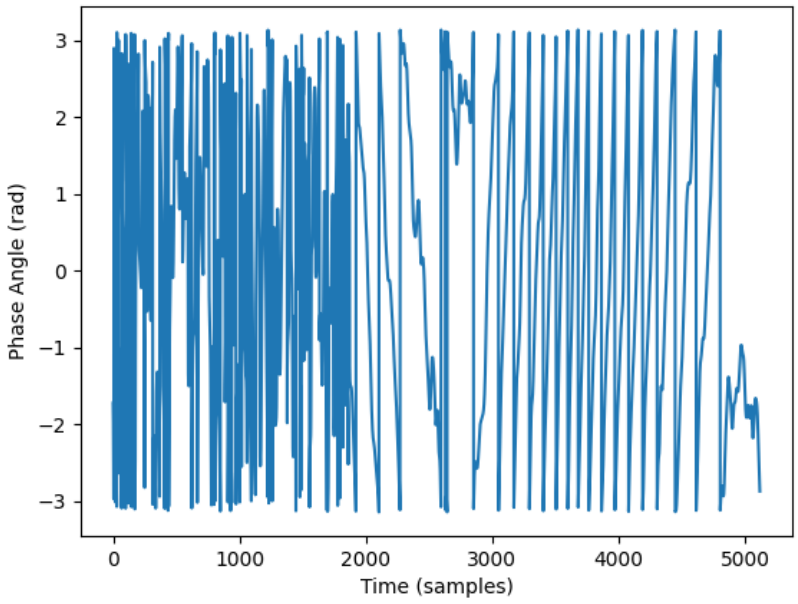
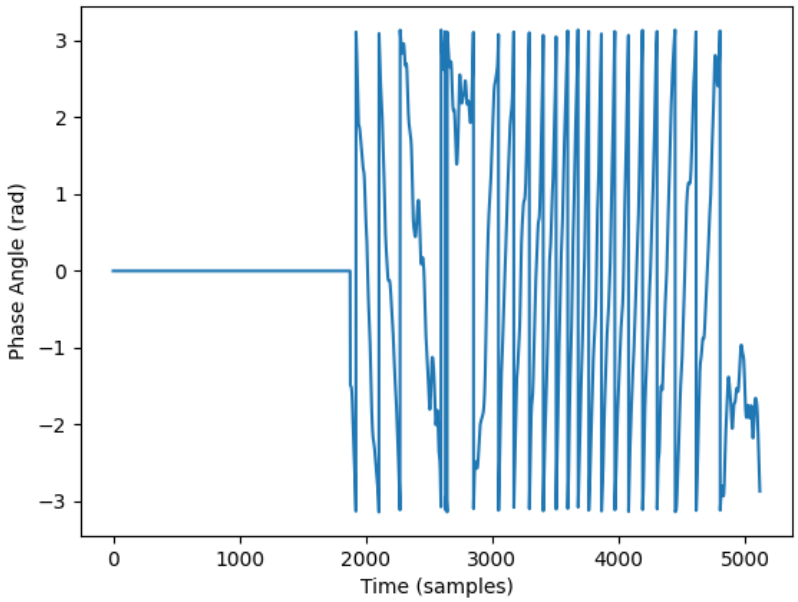


Figure 13: Squelched Signal




After squelching, next I compute the derivative of the phase.

Figure 14: Derivative of Phase

```
# calculate derivative of phase (instantaneous frequency)
# and unwrap phase-wrapping effects
derivtheta = np.diff(np.unwrap(theta) / (2 * np.pi))
```

`np.diff()` computes the derivative, and `np.unwrap()` performs phase unwrapping. Phase unwrapping is necessary because the angle could go from near $2\pi$ in one instant to near 0 in the next, so the derivative of phase across those two samples would incorrectly appear to be very large. Numpy's unwrap function removes these artifacts, and the difference in the plots of the derivative of phase is very clear.

Figure 15: Derivative of Phase Without Unwrapping

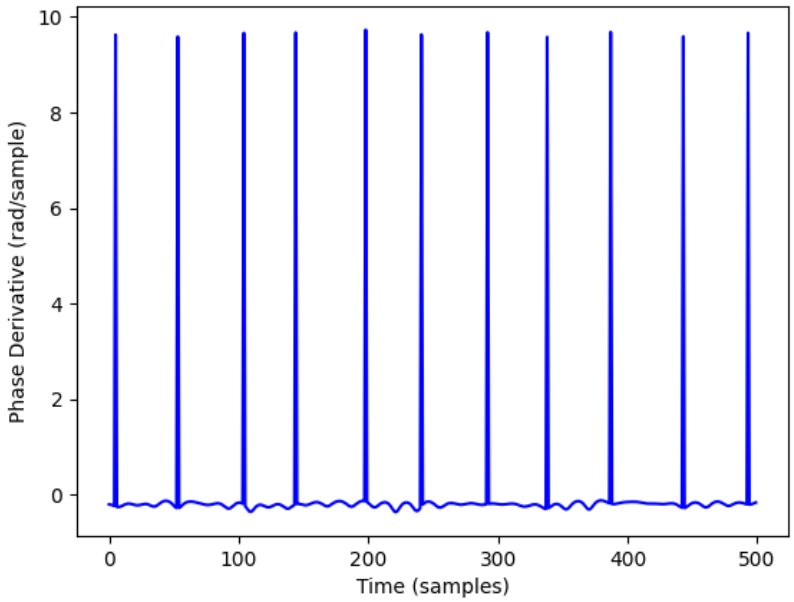
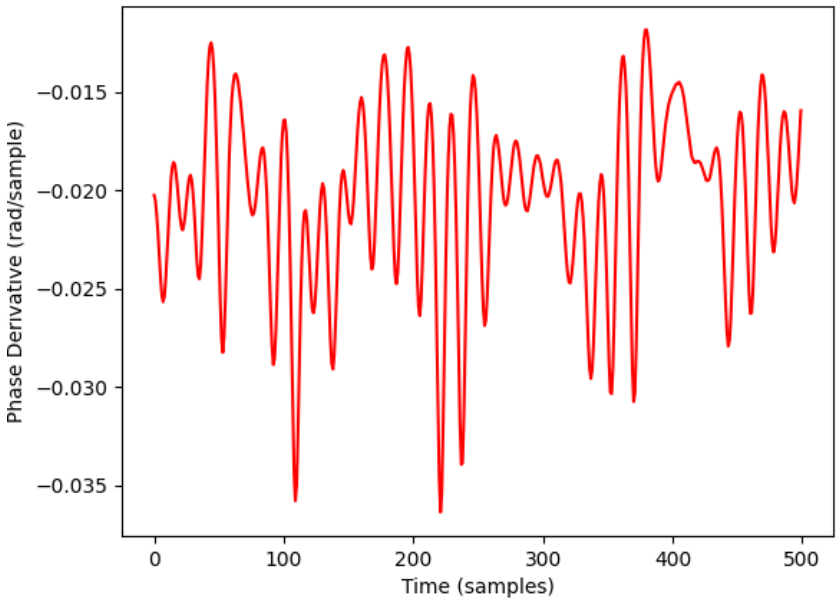


Figure 16: Derivative of Phase With Unwrapping




The same plots in the frequency domain also show the significant improvement gained by phase unwrapping.

Figure 17: Frequency Domain Derivative of Phase Without Unwrapping

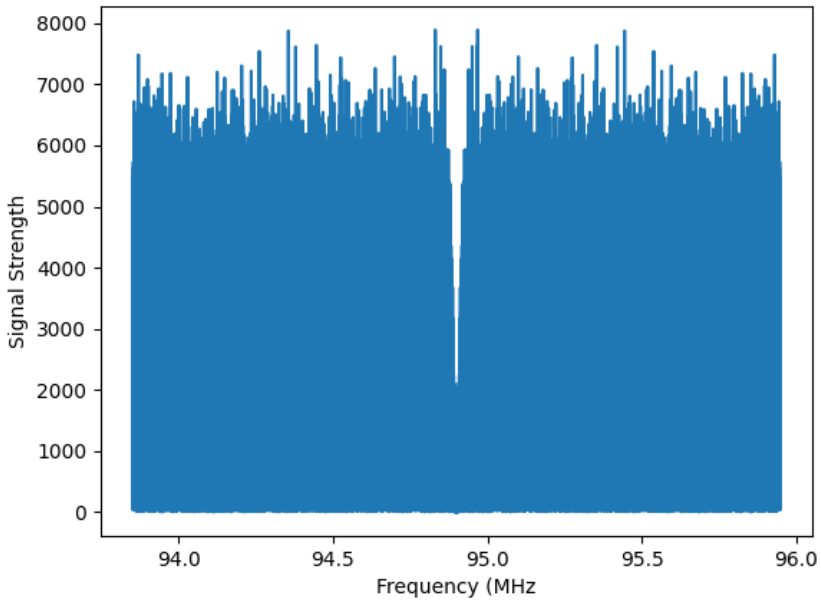
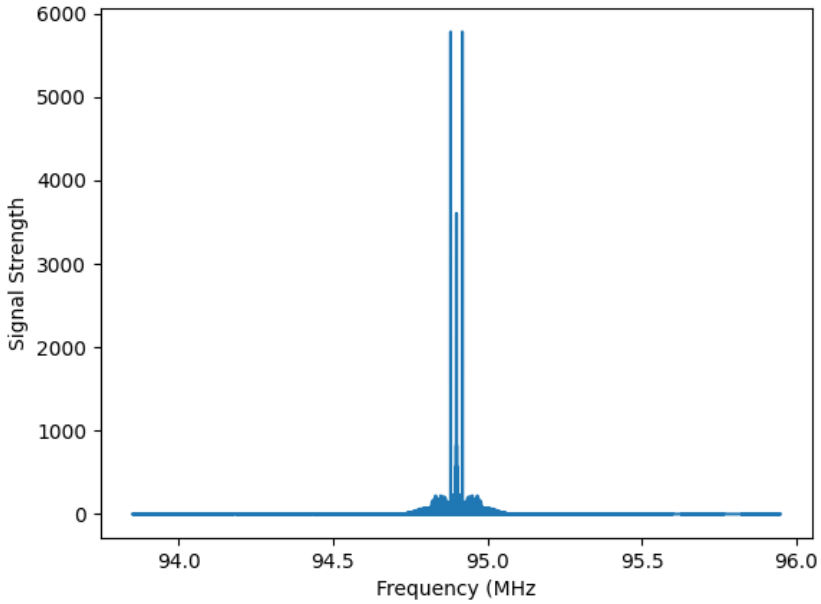


Figure 18: Frequency Domain Derivative of Phase With Unwrapping




### 2.3.3 Downsampling

Finally, the last processing step is to downsample the signal to the audio sampling rate by averaging surrounding values.

Figure 19: Downsampling

```
1  # downsample by taking average of surrounding values
2  dsf = round(f_sps/f_audiosps) # downsampling factor
3  # pad derivtheta with NaN so size is divisible by dsf, then split into
       rows of size dsf and take the mean of each row
4  derivtheta_padded = np.pad(derivtheta.astype(float), (0, dsf -
       derivtheta.size % dsf), mode='constant', constant_values=np.NaN)
5  dsdtheta = np.nanmean(derivtheta_padded.reshape(-1, dsf), axis=1)
```

I used some numpy tricks to speed up this computation. First, I pad the array with NaN's so that its length is divisible by the downsampling factor. Then, I rearrange that array into two dimensions, where the size of each row is the

downsampling factor. Finally, the downsampling is complete by taking the mean of each row, ignoring NaN's. The final audio data in the time and frequency domains are below.

Figure 20: Final Audio (Time Domain)

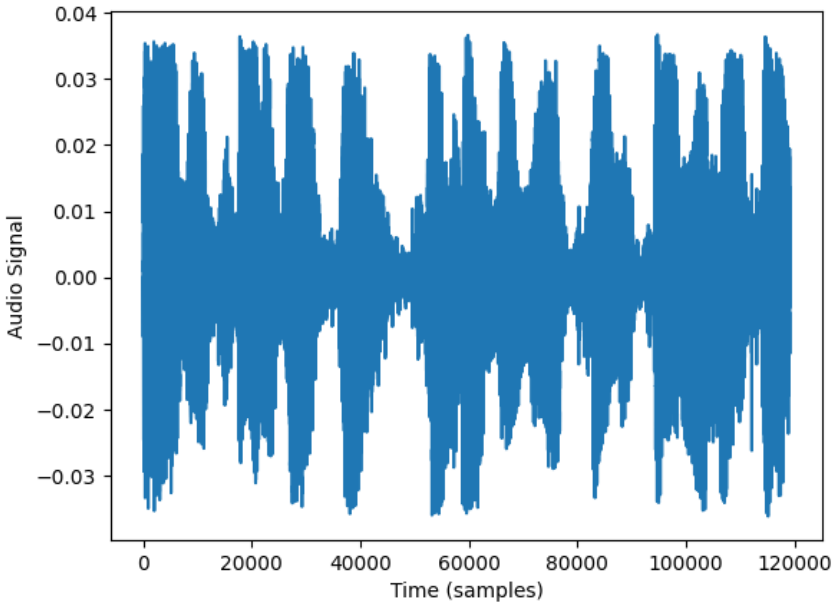
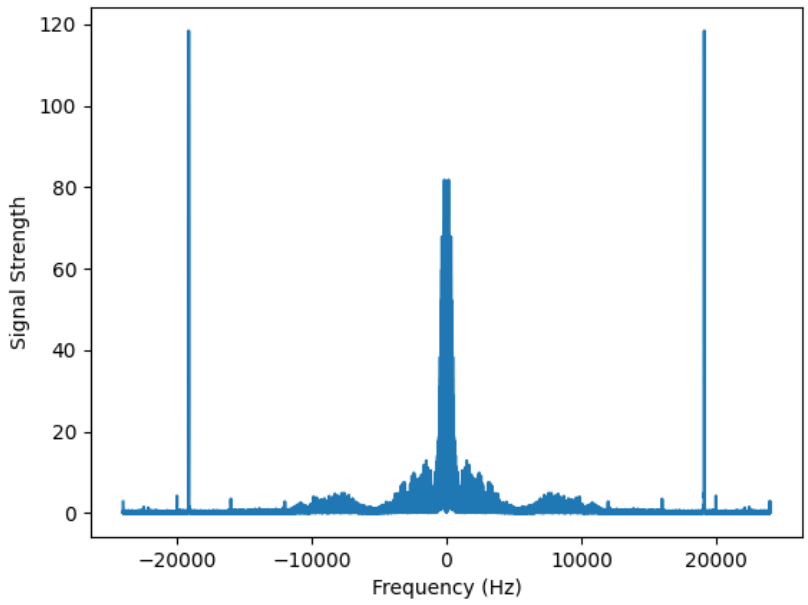


Figure 21: Final Audio (Frequency Domain)




With that, the signal is demodulated into audio data!

## 2.4 Playing Audio

I play the audio using and OutputStream through the sounddevice library. The stream is instantiated as follows.

```
1 self.stream = sd.OutputStream(samplerate=self.f_audiosps,
      blocksize=int(self.N / (self.f_sps / self.f_audiosps)), channels=1)
2 self.stream.start()
```

Writing to the stream is very simple: `self.stream.write(3 * audio)`.

Using a stream instead of calling `sd.play(audio)` allows for much smoother sound between buffers, and it also doesn't block the following code until the audio has been played.

## 2.5 Captioning

I focused mostly on the radio player since this is a project for a wireless programming course, so for captioning, I used a pre-trained speech-to-text model found here. The model runs through pytorch and is very simple to use. Because of words getting clipped between buffers, the caption quality is much better with a longer buffer length (5 seconds) as opposed to a shorter buffer (1 second) that works fine for the audio-only player.

Figure 22: Captioning

```
audio = self.input_queue.get(block=True)
scipy.io.wavfile.write('audio.wav', 48000, audio)
test_files = glob('audio.wav')
batches = split_into_batches(test_files, batch_size=10)
input = prepare_model_input(read_batch(batches[0]),
              device=device)
output = model(input)
for example in output:
    print(decoder(example.cpu()))
```

# 3 Results and Evaluation

## 3.1 Results

The results show this project to be a success. The audio quality output by the radio player is excellent, and there are a variety of easily tunable parameters like the sample rate, buffer time, and the tuning frequency. The audio is clear and continuous, with no gaps between buffers. The caption quality is decent when conditions are favorable, like with only one person speaking clearly with no background noise.

See below for an example of generated captions compared to the actual words. Note that the model does not generate punctuation.

Figure 23: Generated Captions

Generated Text:

```
one has been a child everyone has had parents and the
parent child relationship is relationship marked by
great heaps of delional thinking so parents typically
believe that children are unique and
special miracles beyond our miracles...
```

Actual Text:

```
Everyone has been a child, everyone has had parents, and the
parent-child relationship is a relationship marked by
great heaps of delusional thinking. So parents typically
believe that their children are unique and
special miracles beyond all miracles...
```

One area that could still be improved is the performance. With a sampling rate of 2 Msps, on my computer the radio player can use up to 70% of the CPU. If there are many other programs running, this can sometimes be too much for my computer to handle. To mitigate this, the sample rate can be lowered, but it could be worth it to seek additional performance optimizations in the future. Overall, while audio quality is not quantifiable, the final product works reasonably well in all the goals set out at the beginning.

## 3.2 Challenges

The greatest challenge of this project was achieving seamless playback through the sampling, demodulation, and audio playing processes. I fixed these problems by implementing the optimizations that are detailed in the sections above, but here I'll go into some additional troubles I ran into along the way.

### 3.2.1 Buffer Underruns

Buffer underruns were one issue I got from the audio output stream. These occur when not enough data is being sent to the stream, forcing silent sections that cause the audio to stutter. At first I thought that the output stream required a certain number of samples to be sent in each batch; however, the issue was really just that the demodulation process was not able to keep up to the rate that samples were being read, so the audio player was data starved. I fixed this by adding one more optimization, removing a for loop from the squelching step.

### 3.2.2 Program Exit

Another problem, and one that I haven't yet solved, is getting the program to exit gracefully. I used the pynput library to detect an exit keypress, but the use of multiprocessing makes it very difficult to properly close the program. Because of the multiple subprocesses, each process must clean up, terminate, and close without disrupting any other processes.

Another layer of complication is added because some processes wait for others (for example the audio extraction process waits for the sampling process to send it a sample). This means that there must be some level of synchronization in shutting down the program to make sure no process hangs while waiting for data. While it's not essential to running the radio player, fixing this issue in the future would complete the program nicely.

## 3.3 Future Work

Extensions to this project could include adding a GUI, allowing the listening frequency to be changed while the program is running, and creating a custom speech-to-text model for the captions. However, none of these are particularly involved with the wireless programming side, which is the topic of this class.

I think that this project could be adapted well into a lab for future classes. After going through the conceptually simple FM demodulation steps and with some guidance, I think students would be ready to tackle a streaming FM player. In addition to being a great application of the topics taught in class, the end result is tangible and satisfying to create.

# A Appendix

## A.1 Project Code

The complete code for this project can be found on my github here.

## A.2 External Resources

Along with class resources, the following sources of documentation and information were useful throughout this project.

- PyRTLSDR documentation
- Sound Device documentation
- Multiprocessing documentation
- SciPy Signal documentation
- NumPy documentation
- Silero Models for STT
- SDR product page