Stanford University Department of Electrical Engineering

Professor Dusty Schroeder

EE 102B: Signal Processing and Linear Systems II

June 1, 2023

Stanford University

EE 102B Project Report

Filtering Noise from Ice Sheets using Dynamic Bandstop Filters

Anabella Hernandez
Didi Kamalova
Caleb Matthews
Naomi Mo

Leland Stanford Junior University
Stanford, California

Table of Contents

Segmenting Our Approach

From the original project write up, we were told that the signal we wanted to retrieve lived within the 0 - 15M Hz range. Generally, the approach we took can be sectioned off into the following segments: revealing where the the most lied through the use of bandstop filters, filtering out the "eye patches" of noise in the spectrogram, filtering out the "smile" of the spectrogram, and merging both previously mentioned filters to clean up the entire data set. The reason that we targeted these particular areas on the spectrogram is because they exhibited the highest power spectral density, as indicated by the spectrogram as ~20dB or greater. This was easy to understand upon visual inspection, as these areas were bright yellow regions on the spectrogram. As noise is often correlated with higher power densities due to the addition of additional frequency components, we knew we had to target these areas in our filtering.

Finding The Noise

When modifying the bandpass filters to discover the noisiest parts of the radargram, we used varying frequency ranges. Our first attempts were scanning the probable range of frequencies where the signal might've been seen. A bandpass filter allowed us to do this scanning since (ignoring Gibbs phenomena) it only passed through signals within its limited bandwidth. We changed the bandpass frequency range from 12-14MHz first, noticing that the SNR ratio bumped up to 28.09dB. Although the SNR ratio increased from the unfiltered radargram, there were some prominent gray streaks of noise that were not filtered out. We would later find out that these likely corresponded to the patches of noise seen in the spectrogram, what we later refer to as "eye patches".

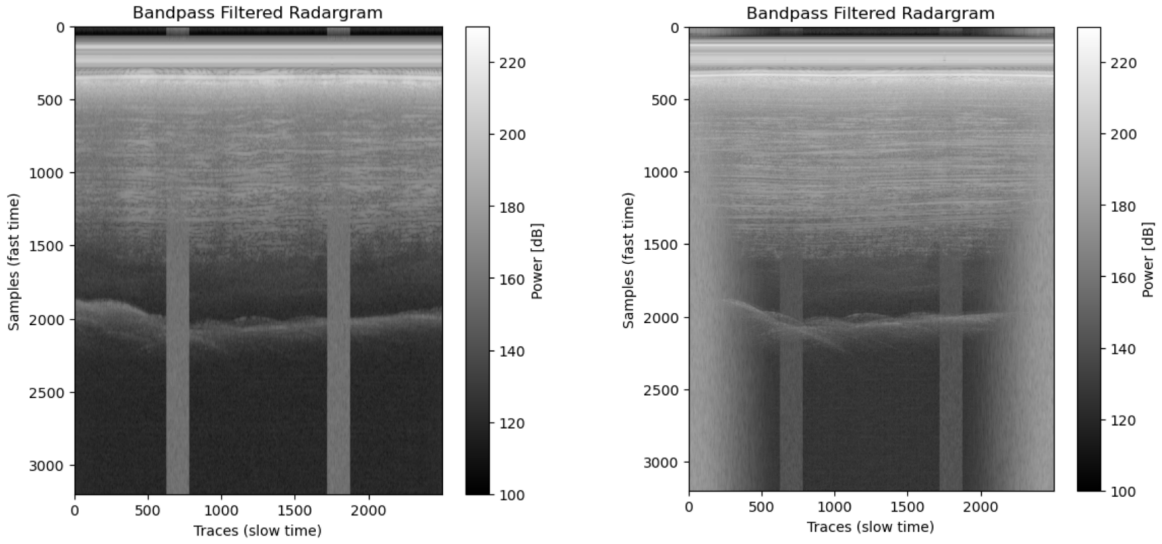


Fig.1 (left to right) Radargrams for bandpass filters with SNRs 28.09dB and 28.86dB.

Playing with the frequencies some more, we changed the bandwidth of the bandpass filter to be between 9 and 12M Hz, further increasing the SNR to 28.09 dB. This indicated to us that when the filter was applied at this range compared to 12-14MHz, there was more signal than noise. Lowering the bottom end of the frequency range by another 2 MHz (now running from 7-12 MHz), yielded an SNR increase to 28.86dB.

Comparing our resulting SNR ratios and bandwidths to the spectrogram, we see that the reason we noticed lower signal levels between 9 and 12MHz is because the spectrogram shows less noise at those frequencies. Picking apart the graph, we see two square patches and a "smile" with relatively low power levels, indicating that they were added noise. The smile lived within the 0 - 9M Hz range, occupying the most portion of the spectrogram. Therefore, when we set the bandwidth to be between 9 and 12MHz, it made sense that our SNR was slightly lower than when we set it to higher bandwidths: the smile appeared for a portion of the bandwidth!

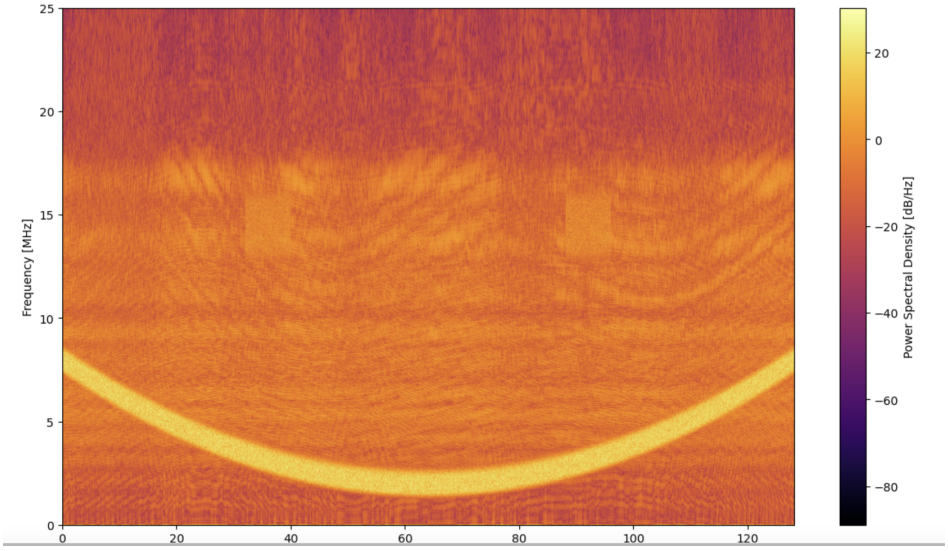


Fig.2 Spectrogram of unfiltered signal.

After probing around to find noisy sections of our radargram, we decided to work on 1) filtering out the "eye patches" and 2) the smile.

However, before discussing smile filtering, there was one interesting discovery that we made when playing around with bandstop/bandpass filters. When band pass filtering the data set between 13 and 16 MHz (essentially cutting out the majority of our signal data), we observed a relatively large spike in SNR. We were confused as to how this could be possible, as we know the majority of the signal lives between 1 and 15 MHz. We then came to the conclusion that the high SNR was the result of the majority of the egregious noise being cut out alongside the signal. Although we were cutting out essential data for the strength of our signal with this band pass technique, we were cutting out a greater magnitude of noise. Thus,

we experienced a positive boost in SNR rather than the decrease that we were expecting. This experiment taught us that boosting SNR can't just rely on preserving the signal we want, but that we must do everything in reference to the noise also being attenuated. If we are cutting out signal but cutting out even more noise, then we can consider that a successful approach in raising the SNR.

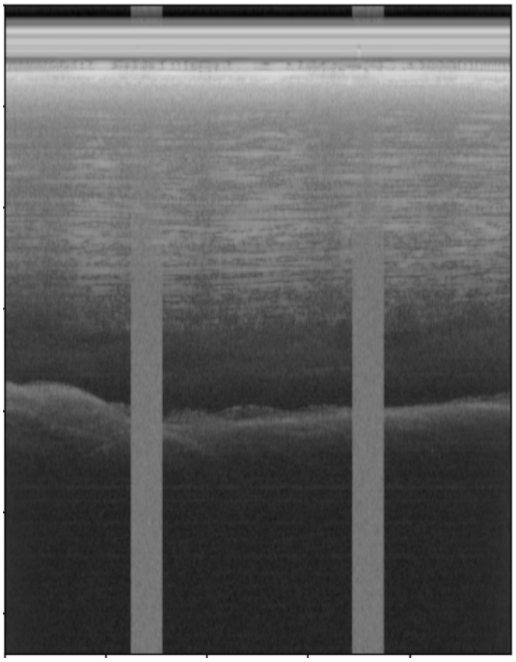


Fig.3 Bandpass filtered (13-16 MHz) radargram, 31.99 dB SNR.

With this caveat in mind, we started our approach in filtering out the smile with greater emphasis on targeting the most egregious of noise frequencies, rather than creating the most precise filter possible. We decided it would be more important to get a brute force noise attenuator working before we tailored the precision of our filter further.

Filtering The Smile

In order to filter out the smile, we wanted to implement a dynamic bandstop filter that would change which frequencies it stopped depending on which moment of slow time it was acting upon. This is because the "smile" did not stay constant in frequency across slow time. We designed two different dynamic bandstop filters with different approaches in accomplishing this goal.

Filtering by Max Power Spectral Density

One approach that we utilized to filter out the smile was to filter out the frequencies (at each moment in slow time) that exhibited the greatest magnitude of power spectral

density. This is because the smile exhibited the greatest magnitude of power spectral density on the unfiltered data's spectrogram (around 20dB, indicated by bright yellow).

We implemented this approach by accessing the unfiltered spectrogram's values for spectral density, computed in the starter code and stored in Sxx_test:

*f, t, Sxx_test = scipy.signal.spectrogram(final_trace.flatten(), **spectrogram_options)*

We then converted these values to dB/Hz with scaling and a log conversion, storing the result in an array called chirp_data:

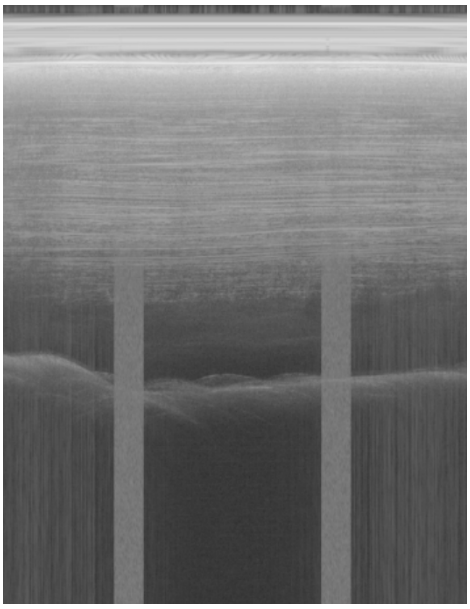
*chirp_data = 10*np.log10(np.abs(Sxx_test))*

This array provided us with the power spectral densities in dB/Hz for each frequency in the signal's frequency spectrum for a given moment in slow time. We were then able to iterate through each moment in slow time (the loop given to us in the starter code), and then within this loop access the signal's power densities at that moment in slow time. By looping through the densities at that particular moment in slow time, we were able to find the index at which the maximum power spectral density occurred, as shown in the code snippet below. The index at which the highest power spectral density occurred was stored in a variable named top_val_freq_idx. This index was then used to access the corresponding frequency at which the highest power spectral density occurred. This frequency was then used as a "center frequency" by which to create the upper and lower bounds of our bandstop. In the end, it wasn't exactly in the center (as you can see from the code), because we found that certain regions needed greater upper bound attenuation than lower bound attenuation and vice versa. However, the grabbed frequency did serve as a reliable reference point for which to tailor these bounds.

In the end, this approach worked reliably well. As we expected, this approach only targeted the "smile" present in the spectrogram of the dataset, as evidenced by the attenuation shown in our new spectrogram (also below). Upon further inspection of the spectrogram, one can see how the bandstop attenuates different frequencies along the smile—the relationship between adjacent bandstops isn't particularly smooth, but the bandstop filters reliably adhered to the curvature of the smile over all instances in slowtime. This approach gave us a relatively high SNR of 31 dB, indicating that a substantial amount of noise was successfully attenuated from the data set.

However, despite the approach largely working, we observed some areas that the approach failed: namely, it didn't *smoothly* attenuate frequencies along the smile, and rather hopped around between the frequency bounds of the smile's curvature. Thus, some noises at some frequencies were passed in certain moments of slow time and then stopped in immediately adjacent periods of slow time. We wanted to improve upon this approach and

create a bandstop filter that more smoothly attenuated the curvature of the smile, which then led us to our next approach.



```
# access chirp powers
top_val = 0
top_val_freq_idx = 0
for i in range(len(chirp_data)):
    if chirp_data[i][idx] > top_val:
        top_val = chirp_data[i][idx]
        top_val_freq_idx = i

# access value at same index in the frequencies
freq_mpd = f[top_val_freq_idx]

# create bounds for the bandstop filter
bandwidth = 1.56e6 #adjust bandwidth as needed
lower_bound = freq_mpd - bandwidth
upper_bound = freq_mpd + 1.55 * bandwidth
if idx < 500 or idx > 2150:
    lower_bound = lower_bound - .5e6
else:
    upper_bound = upper_bound + .3e6
```

Fig.4 The radargram of the power-density filtered smile and the code utilized to target frequencies with high power densities.

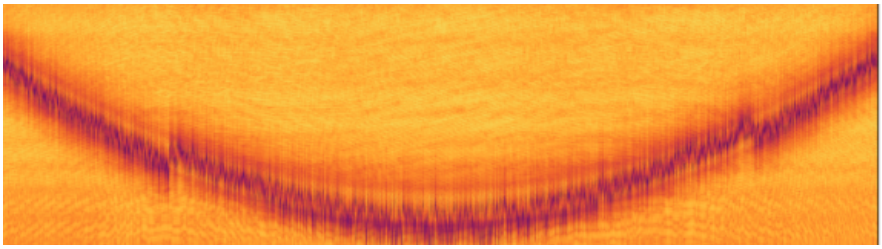
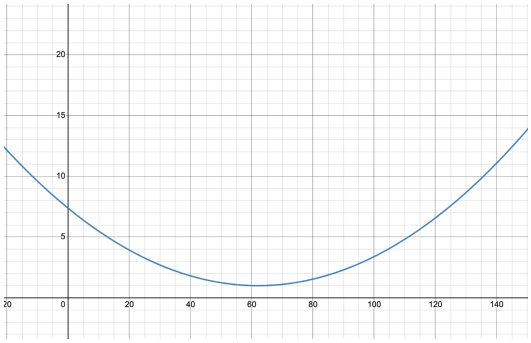


Fig.5 The spectrogram of the power-density filtered smile. Note the bandstop filter changes frequency ranges with each instance in slow time.

Filtering by Parabolic Fit

Another approach implemented by our team works by varying bandwidth values for each slice of time slice according to a parabolic shape we estimated by inspecting the spectrogram. Our initial visual estimations yielded the graph and equation below.



$$y = \left[\left(\frac{x}{24.57} - 2.523\right)^2 + 1\right] \times 10^6$$

Fig.6 Parabolic representation of the "smile" noise in the spectrogram.

Upon implementation, we discovered that the smile is not a perfect parabola, and therefore cannot be well mapped by one. To account for this, we developed a second implementation that utilizes 6 bounding parabolas. We divided time into three segments, ranging from 0-50, 50-80 and 80-120 seconds, each of which corresponds to an upper and lower-bounding parabola that are picked to match the curve of the smile. As the iterator steps through time, it applies a bandstop across the frequencies defined by its upper and lower bounding curves which correspond to Equations 1-6.

$$a = \left[\left(\frac{x}{24.57} - 2.1\right)^2 + 1\right] \times 10^6 \qquad (1)$$

$$b = \left[\left(\frac{x}{24.57} - 2.523\right)^2 + 5\right] \times 10^6 \qquad (2)$$

$$c = \left[\left(\frac{x}{30.1} - 2\right)^2 + 0.7\right] \times 10^6 \qquad (3)$$

$$d = \left[\left(\frac{x}{24.57} - 2\right)^2 + 5\right] \times 10^6 \qquad (4)$$

$$e = \left[\left(\frac{x}{25} - 2.4\right)^2 + 0.5\right] \times 10^6 \qquad (5)$$

$$f = \left[\left(\frac{x}{25} - 2.523\right)^2 + 4.8\right] \times 10^6 \qquad (6)$$

Overall, we are utilizing 6 different curves, each of which have been created through generating an array of 2500 equally spaced points between 0 and 120 and to 6 functions outlined below.

```
# create 2500 equally spaced points between 0 and 120
x = np.linspace(0, 120, 2500)

a = (((x/24.57) - 2.1)**2 + 1) * 1e6
b = (((x/24.57) - 2.523)**2 + 5) * 1e6

c = (((x/30.1) - 2)**2 + 0.7) * 1e6
d = (((x/30.1) - 2)**2 + 5) * 1e6

e = (((x/25) - 2.4)**2 + 0.5) * 1e6
f = (((x/25) - 2.523)**2 + 4.8) * 1e6
```

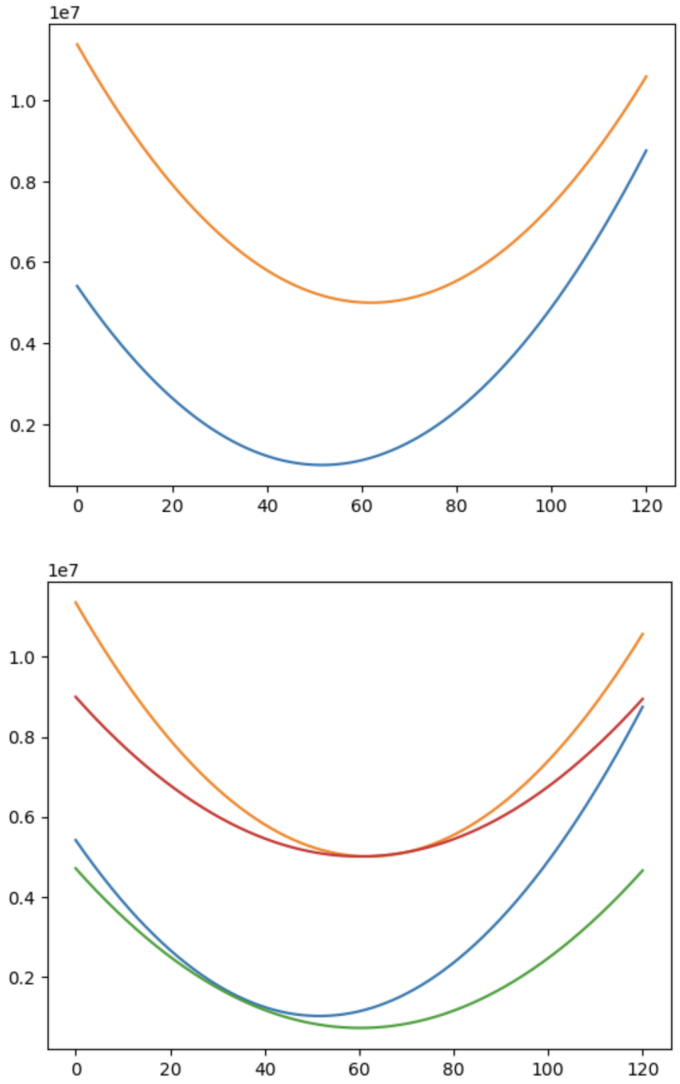


Fig.7 (top to bottom) Curves applied for 0-50 seconds and all 6 curve pairs.

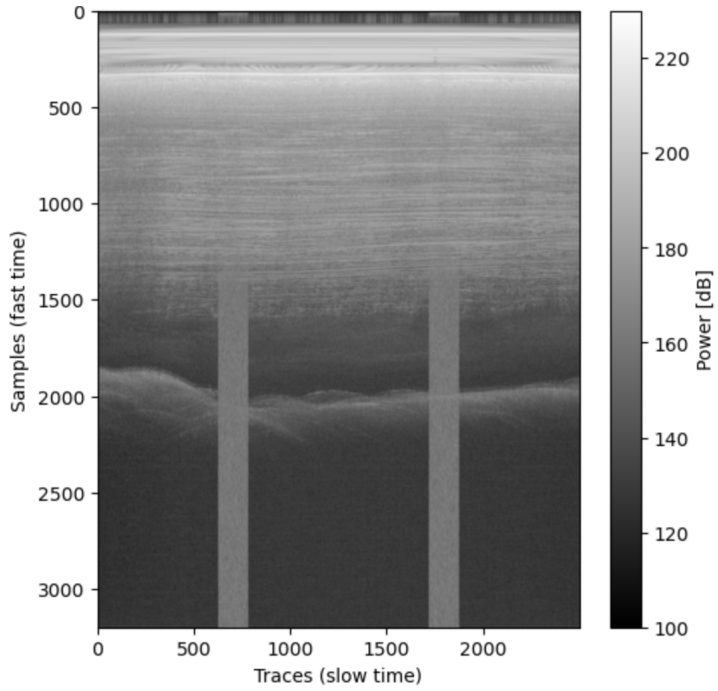


Fig.8 Filtered radargram for the parabolically fitted dynamic bandstop filter.

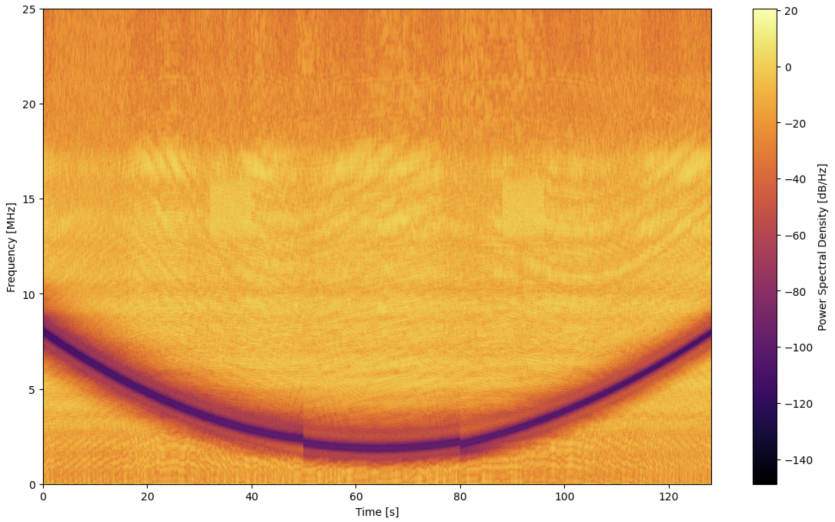


Fig.9 Spectrogram the parabolically fitted dynamic bandstop filter.

This approach worked pretty well and gave an SNR 35.72 dB standalone. It was able to follow a smooth curve along the line of the smile, which is exactly what we wanted to achieve. This approach, however, is time-consuming and tedious to implement as it requires a number of manual adjustments in order to recreate the curve of the smile.

Filtering The Eye Patches

To address the eye patches, our approach was to first create a bandstop filter that could attenuate the frequency range where the patches appeared and second, only apply the filter at the time intervals where the patches occur.

The radargram shown in Figure 10 below shows the result of applying the bandstop filter at the frequency where the patches appear (~10MHz - 18MHz) and at particular time intervals. While the spectrogram looks more "fuzzy" than the unfiltered radagram, it is apparent that the eye patch segments have been attenuated, since they are not as clearly defined in the intervals of time we'd previously seen them appear in.

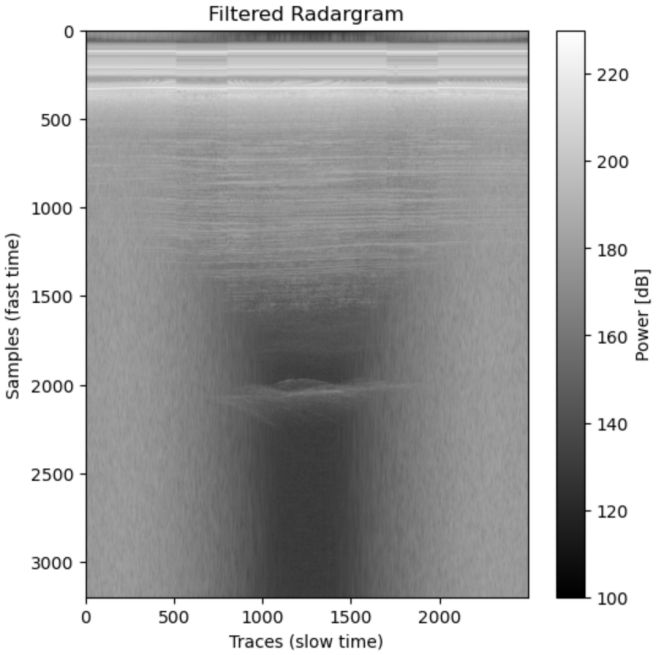


Fig.10 Radargram of filtered eye patch section.

From the unfiltered spectrogram, we observed the patches appearing around 510 - 880 seconds and 1700 - 1990 seconds, which is where we "turned on" the bandpass filter. The spectrogram below shows that once we were able to identify that the patches appeared within the 10-18MHz range and at the aforementioned intervals of time, we were able to attenuate that noise, indicated by the purple.

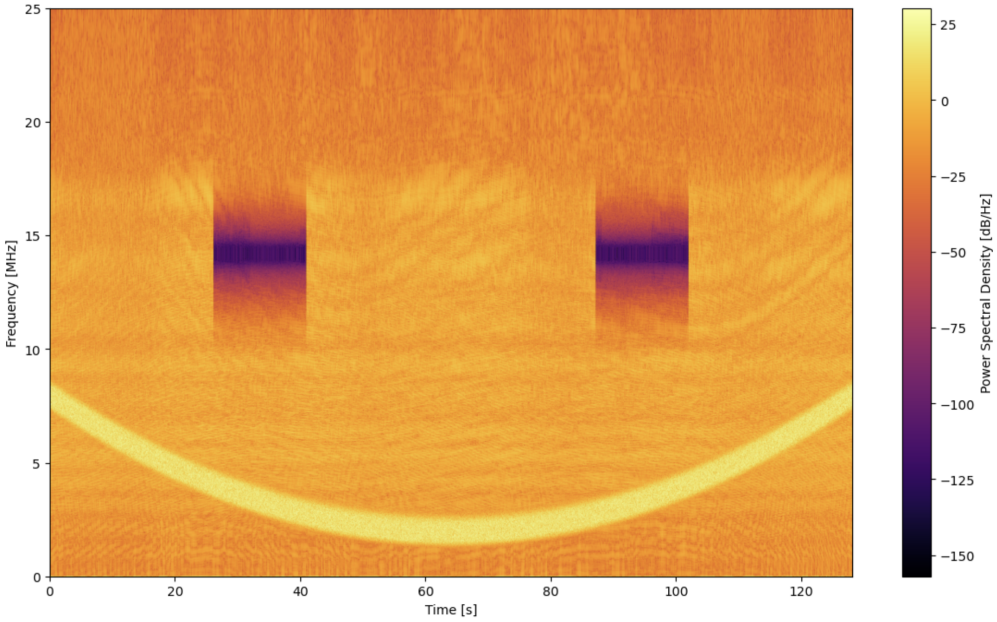


Fig.11 Spectrogram of filtered eye patches.

Merging Filters

Finally, in order to maximize our SNR, we combined our most successful filters, in order to eliminate the parabola-esque curve, as well as the eye patches. To achieve this, we added our three-segment parabolic-arc bandstop filter and our time-triggered eyepatch bandstop filter together in the iterator. We ran each segment of data first through the parabolic bandstop and then through the eyepatch filter (if active) before moving to the next time slice. This methodology proved very effective, yielding an SNR of 36.20 dB.

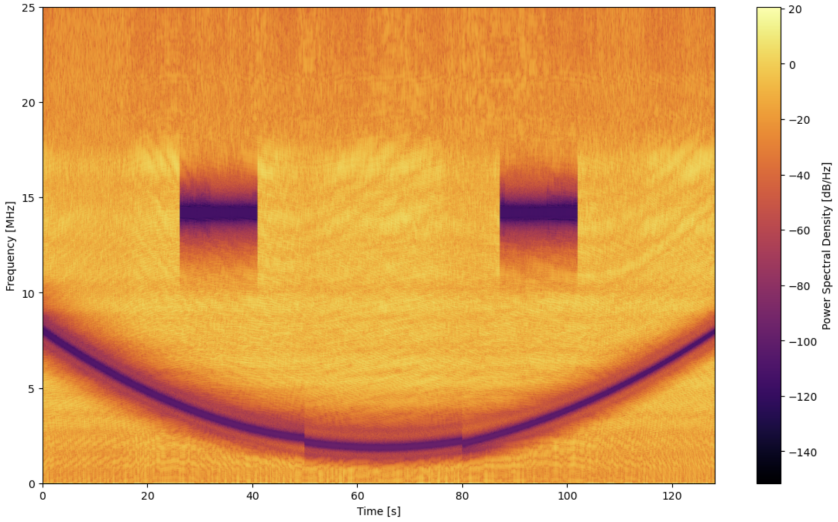


Fig.12 Spectrogram of the parabolic approach.

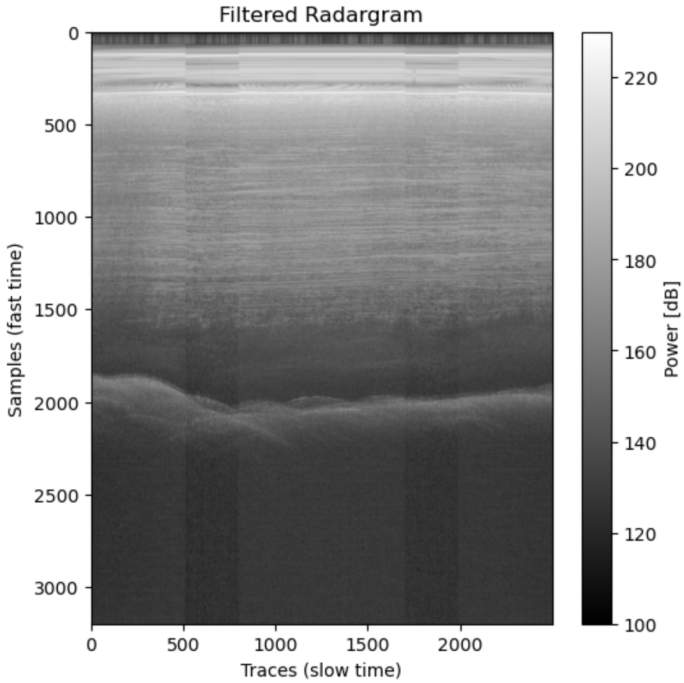


Fig.13 Radargram of the parabolic approach.

By adding further optimizations, such as additional bandstop filters across certain frequencies with higher spectral density (an example if shown in figure 14), as well as experimenting with elliptic filters with extremely high stop band attenuation, we achieved an even higher SNR: 36.36 dB.

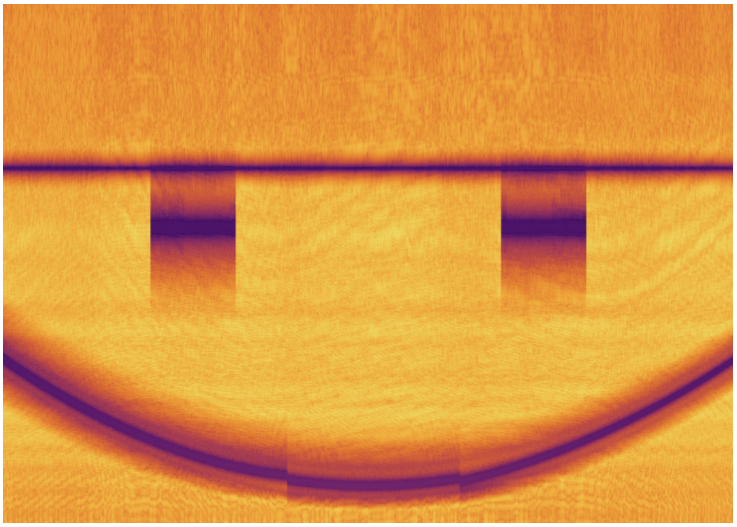


Fig.14 Spectrogram showing additional filters added to attenuate regions of high power spectral density.

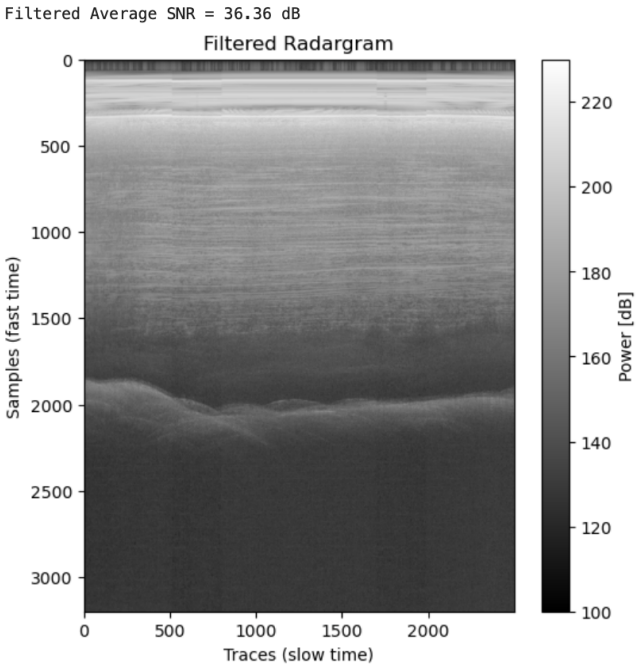


Fig.15 Final radargram showcasing 36.36 dB SNR.

Conclusion

The implementation of a dynamic bandstop filter to attenuate the "smile" showcased in the original data's spectral density proved to be an effective technique in raising our SNR. The parabolic approach was particularly effective in attenuating the smile precisely. Given more time, we think we could increase our SNR even further by filtering out patches of noise (areas of high power spectral density) beyond the smile/eyes with more precision, rather than using brute force methods such as the time-spanning bandstop in figure 14.

Although our group would have additionally liked to experiment more with using different types of filters, we believe we did the best that we could with the approach we ideated in our early stages of planning. We are overall satisfied with the filters that we have created, and feel that we have learned much over the course of this project.

GitHub Repo to our project can be accessed via this link.

Acknowledgements

We would like to extend our gratitude to Professors Joseph Kahn and Dustin Schroeder for aiding our learning these past two quarters, and for opening up the world of signal processing for us to enjoy and explore. We would also like to thank our TA Shreya Shankar for her kindness, encouragement, and patience in dealing with our many questions, and for making difficult concepts seem easy the moment we step into office hours.

Appendix

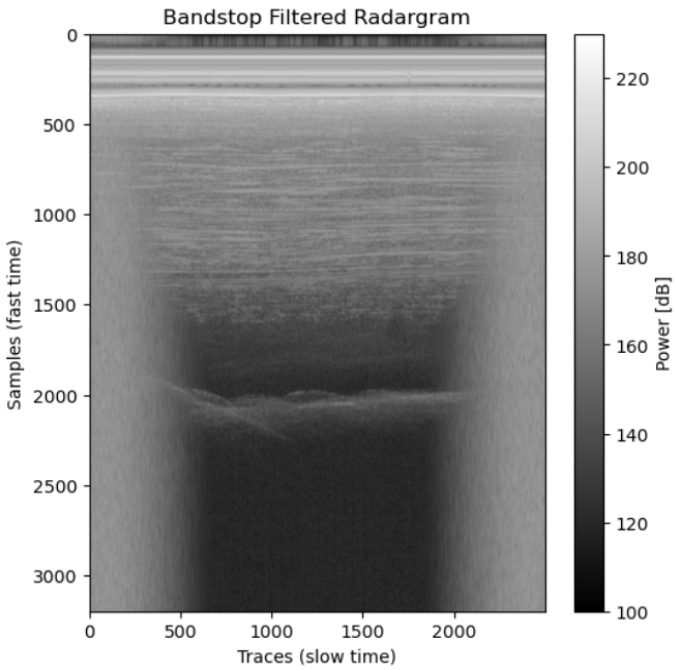
Implement Series of Bandstop Filters

In [7]:
```
# look at noise characteristics (e.g. compute spectrogram) fill in TODOs in framework below
# next step is plotting filtered data?
```

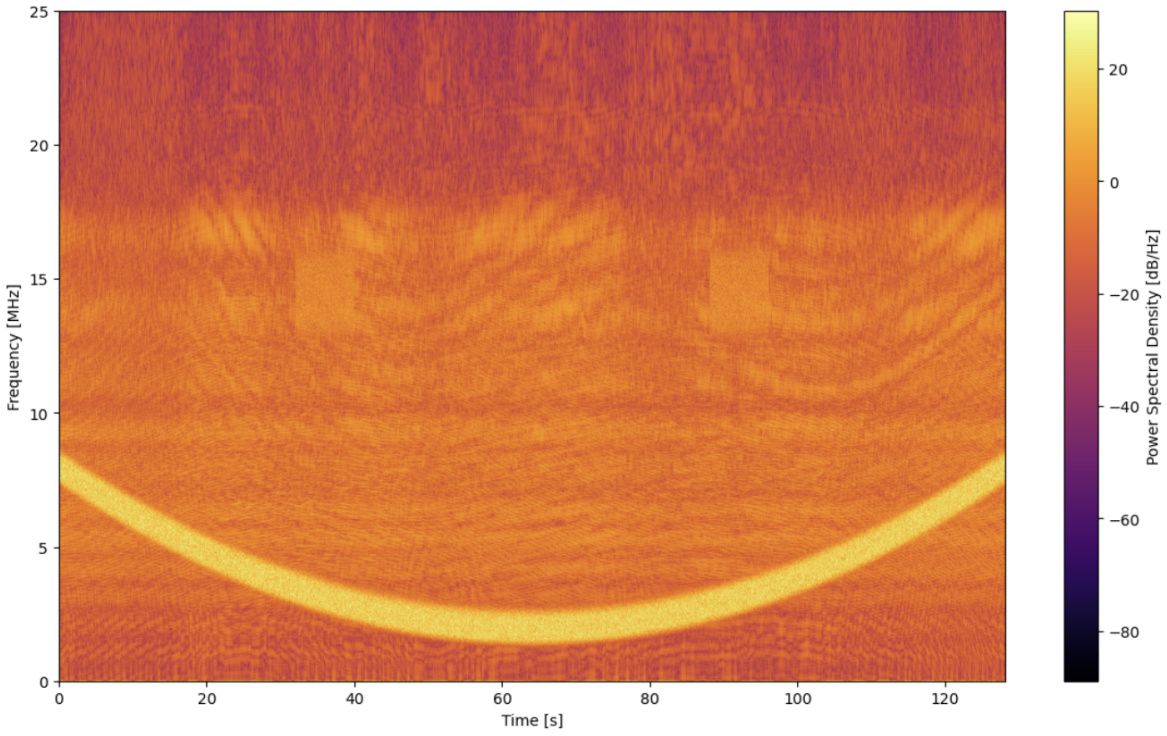
In [8]:
```
# First try bandstop filter:
#bandpass_filter = scipy.signal.butter(4, (1e6,21e6), btype='bandpass', output='sos', fs=ds.fs);
bandstop_filter1 = scipy.signal.butter(4, (1e6,7e6), btype='bandstop', output='sos', fs=ds.fs)
bandstop_filter2 = scipy.signal.butter(4, (8e6, 20e6), btype='bandstop', output='sos', fs=ds.fs)
bandstop_filter3 = scipy.signal.butter(4, (7.5e6, 7.6e6), btype='bandstop', output='sos', fs=ds.fs)

data_out = np.zeros((len(ds["slow_time"]), len(ds["fast_time"]))).astype(float)

ref_chirp = ds["reference_chirp_real"] + 1j * ds["reference_chirp_imag"]
```

In [9]:
```
for idx, t in enumerate(ds["slow_time"]):
 trace_unfiltered = ds["data"][idx,:]

    # first bandpass
    #trace_filtered_bp = scipy.signal.sosfilt(bandpass_filter, trace_unfiltered)
    #pulse_compressed = pulse_compress_one_trace(trace_filtered_bp, ref_chirp_time_domain=ref_chirp)

    # first bandstop
    trace_filtered_1 = scipy.signal.sosfilt(bandstop_filter1, trace_unfiltered)

    # second bandstop
    trace_filtered_2 = scipy.signal.sosfilt(bandstop_filter2, trace_filtered_1)
    pulse_compressed = pulse_compress_one_trace(trace_filtered_2, ref_chirp_time_domain=ref_chirp) #
    data_out[idx, :] = (20*np.log10(np.abs(pulse_compressed)))

    # third bandstop
    #trace_filtered_3 = scipy.signal.sosfilt(bandstop_filter3, trace_filtered_2)
    #pulse_compressed = pulse_compress_one_trace(trace_filtered_3, ref_chirp_time_domain=ref_chirp)
    #data_out[idx, :] = (20*np.log10(np.abs(pulse_compressed)))

plot_radargram(data_out, title="Bandstop Filtered Radargram")

snr_bandstop, snr_avg_bandstop = estimate_snr(data_out, ds["bed_pick"])
snr_avg_bandstop = 10*np.log10(snr_avg_bandstop)
print('Bandstop Filtered Average SNR = %.2f dB' % snr_avg_bandstop)
```

```
Bandstop Filtered Average SNR = 30.48 dB
```



In [10]:
```
window = 'flattop' #TODO
spectrogram_options = {
    "fs": ds.fs,
    "window": window,
    "nperseg": len(ds.fast_time),
    "noverlap": 0, #TODO
    "mode": "psd",
    "scaling": "density",
    "return_onesided": True
}
f, t, Sxx_test = scipy.signal.spectrogram(ds["data"].to_numpy().flatten(), **spectrogram_options)

fig, ax = plt.subplots(1,1, figsize=(14, 8), facecolor='white')
p = plt.pcolormesh(ds["slow_time"], f/1e6, 10*np.log10(np.abs(Sxx_test)), shading='gouraud', cmap='inferno')#, vmin=-40, vmax=10)
clb = fig.colorbar(p, ax=ax)
clb.set_label('Power Spectral Density [dB/Hz]')
ax.set_xlabel('Time [s]')
ax.set_ylabel('Frequency [MHz]')
plt.show()
```

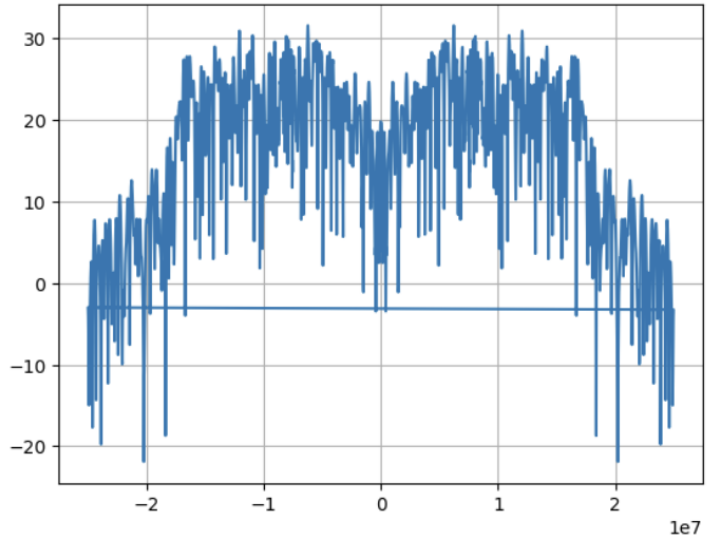


Try Time-Dependent Bandstop Filter - Buggy Attempt

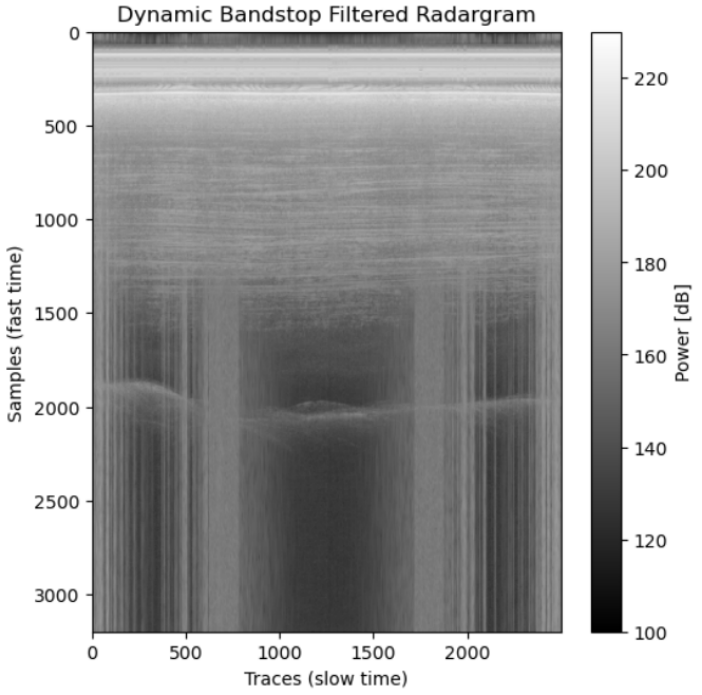
In [12]:
```
# build bandstop filters over different ranges
bandpass = scipy.signal.butter(4, (1e6,20e6), btype='bandpass', output='sos', fs=ds.fs)
seven_filter = scipy.signal.butter(4, (7.5e6,8e6), btype='bandstop', output='sos', fs=ds.fs)
two_filter = scipy.signal.butter(4, (2e6,3e6), btype='bandstop', output='sos', fs=ds.fs)

data_out = np.zeros((len(ds["slow_time"]), len(ds["fast_time"]))).astype(float)

ref_chirp = ds["reference_chirp_real"] + 1j * ds["reference_chirp_imag"]
```

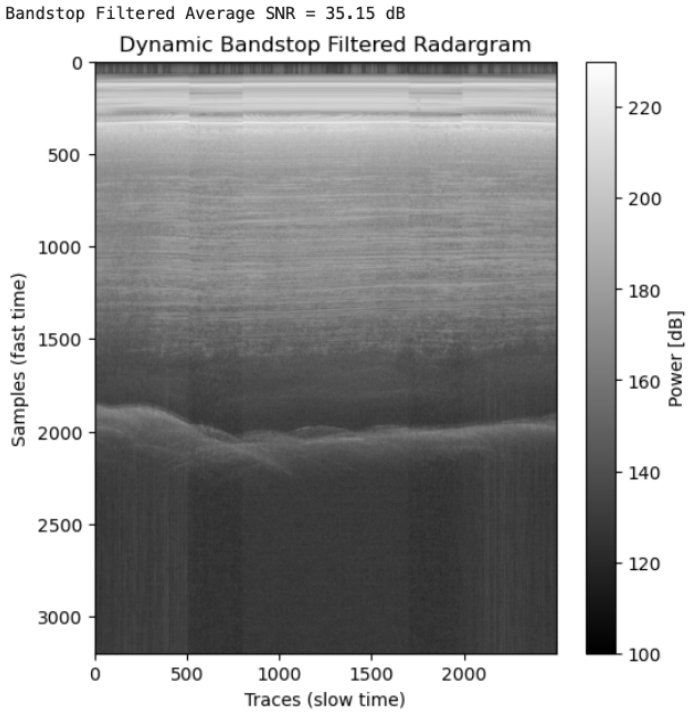
In [13]:
```
for idx, t in enumerate(ds["slow_time"]):
    trace_unfiltered = ds["data"][idx,:]

    # convert signal amplitude data into np array
    amplitudes = np.array(trace_unfiltered)

    # compute FFT of signal amplitudes
    amplitudes_fft = np.fft.fft(amplitudes)

    # compute spectral power
    psd_2 = np.abs(amplitudes_fft) ** 2

    # compute one-sided power spectral density
    psd = (1/len(amplitudes)) * np.abs(amplitudes_fft) ** 2

    # compute frequency values associated with each PSD coefficient
    n = len(amplitudes)
    sampling_freq = ds.fs
    freqs = np.fft.fftfreq(n,d=1/sampling_freq)

    # convert PSD to dB/hZ
    psd_dbhz = 10 * np.log10(psd/sampling_freq)
    max_power_density = np.max(psd_dbhz)
    max_pd_index = np.argmax(psd_dbhz) # finds index at which max power density occurs
    pd_threshold = 20;

    # grab frequency at which max power density occurs
    freq_mpd = np.abs(freqs[max_pd_index])

    # code to print power densities/frequency index for given slow time
    if idx == 0:
        print(max_power_density)
        print(max_pd_index)
        print(freqs[#max_pd_index])
        print(psd_dbhz)
        print(freqs)

    # create bounds for the bandstop filter
    bandwidth = 2.15e6 #adjust bandwidth as needed
    lower_bound = freq_mpd - bandwidth
    upper_bound = freq_mpd + bandwidth

    # code to print bounds of filter
    # print(str(lower_bound) + ", " + str(upper_bound))

    # code to print spectral density for a given slow time
    #if idx == 5:
        #plt.plot(freqs, psd_dbhz)
    #    plt.xlabel('Frequency (Hz)')
    #    plt.ylabel('Power Spectral Density (dB/Hz)')
    #    plt.title('Power Spectral Density')
        #plt.grid(True)
        #plt.show()
    #    print('lower bound: ' + str(lower_bound) + ', upper bound: ' + str(upper_bound))

    # apply bandstop filter
    filter = scipy.signal.butter(4, (lower_bound,upper_bound), btype='bandstop', output='sos', fs=ds.fs)
    trace_filtered = scipy.signal.sosfilt(filter, trace_unfiltered)
    pulse_compressed = pulse_compress_one_trace(trace_filtered, ref_chirp_time_domain=ref_chirp)
    data_out[idx, :] = (20*np.log10(np.abs(pulse_compressed)))

plot_radargram(data_out, title="Dynamic Bandstop Filtered Radargram")

snr_lowpass, snr_avg_lowpass = estimate_snr(data_out, ds["bed_pick"])
snr_avg_lowpass = 10*np.log10(snr_avg_lowpass)
print('Bandstop Filtered Average SNR = %.2f dB' % snr_avg_lowpass)
```

```
30.71912983453271
518
8093750.0
[ 5.59444508 16.01239675 15.83335456 ... 15.4001951  15.83335456
 16.01239675]
[     0.  15625.  31250. ... -46875. -31250. -15625.]
```



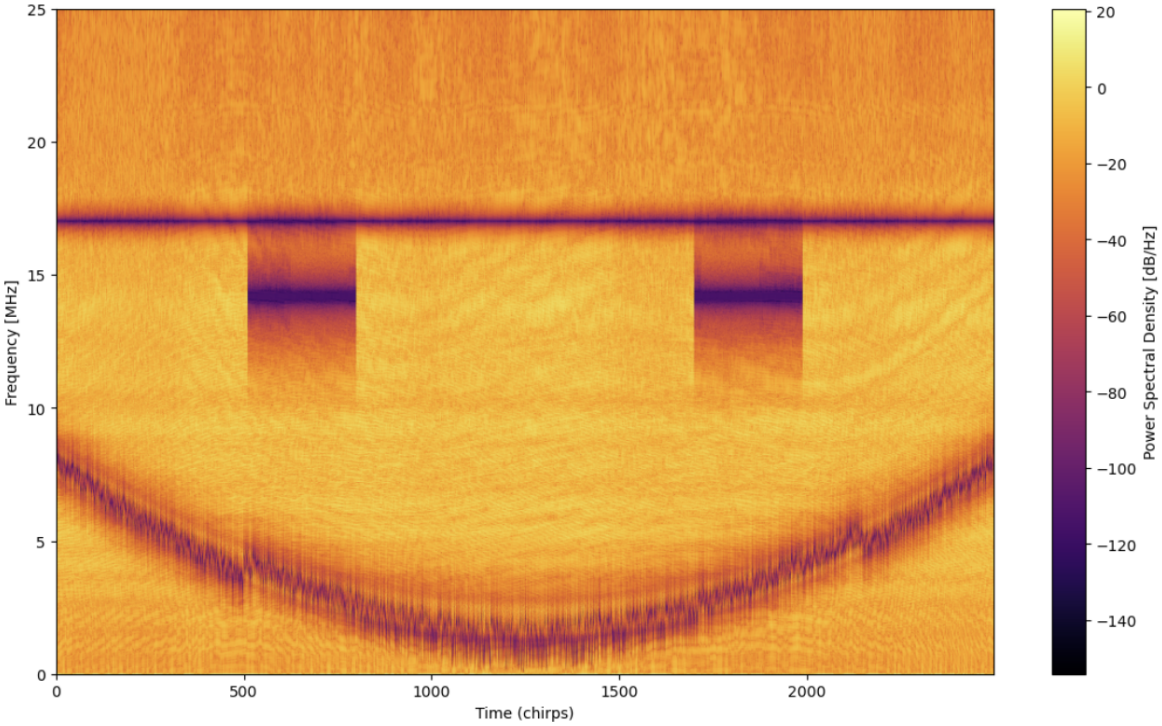

```
Bandstop Filtered Average SNR = 29.89 dB
```





In [231…

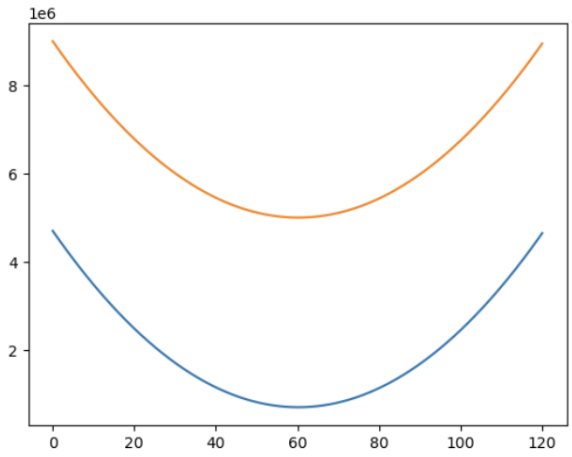
```
window = "flattop" #TODO
spectrogram_options = {
    "fs": ds.fs,
    #"window": scipy.signal.windows.gaussian(len(ds.fast_time),20),
    "window": window,
    "nperseg": len(ds.fast_time),
    "noverlap": 0, #TODO
    "mode": "psd",
    "scaling": "density",
    "return_onesided": True
}
f, t, Sxx_test = scipy.signal.spectrogram(final_trace.flatten(), **spectrogram_options)

fig, ax = plt.subplots(1,1, figsize=(14, 8), facecolor='white')
p = plt.pcolormesh(np.arange(len(ds["slow_time"])), f/1e6, 10*np.log10(np.abs(Sxx_test)), shading='gouraud', cmap='inferno')#, vmin=-40, vmax=10)
clb = fig.colorbar(p, ax=ax)
clb.set_label('Power Spectral Density [dB/Hz]')
ax.set_xlabel('Time (chirps)')
ax.set_ylabel('Frequency [MHz]')
plt.show()

#OTHER NOTES/SCRIBBLES
#compute the frequency values associated with each FFT coefficient
    # slow time samples: 2500 -> timestep = 1/2500
    #timestep = .0004
    #frequencies = np.fft.fftfreq(len(trace_unfiltered), d=timestep)

    # find the index of the maximum spectral power/density
    #max_power = np.argmax(np.abs(frequencies))

    # get corresponding frequency values at maximum index
    #max_frequency = frequencies[max_index}

    # frequencies = take FFT to get spectrum
    # threshold - higher power than certain amount is noise
    # take peak power
```


Frequency [MHz]
Time (chirps)
Power Spectral Density [dB/Hz]
0
5
10
15
20
25
0
500
1000
1500
2000
20
0
−20
−40
−60
−80
−100
−120
−140

Didi's Parabolic Bandstop - Final Implementation With Optimized Filtering

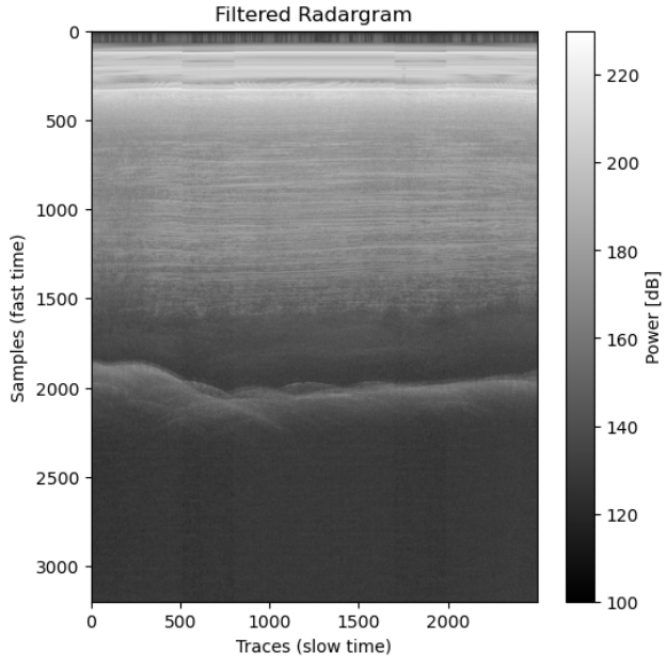
In [318…
```
# Creating a scuffed bandstop

# create 2500 equally spaced points between 0 and 120
x = np.linspace(0, 120, 2500)

a = (((x/24.57) - 2.1)**2 + 1) * 1e6
b = (((x/24.57) - 2.523)**2 + 5) * 1e6

c = (((x/30.1) - 2)**2 + 0.7) * 1e6
d = (((x/30.1) - 2)**2 + 5) * 1e6

e = (((x/25) - 2.4)**2 + 0.5) * 1e6
f = (((x/25) - 2.523)**2 + 4.8) * 1e6

fig, ax = plt.subplots()
ax.plot(x, c)
ax.plot(x, d)
```

Out[318]: `[<matplotlib.lines.Line2D at 0x14fd41d90>]`



In [331…
```
data_out = np.zeros((len(ds["slow_time"]), len(ds["fast_time"]))).astype(float)
ref_chirp = ds["reference_chirp_real"] + 1j * ds["reference_chirp_imag"]

final_trace = np.zeros_like(data_out)
for idx, t in enumerate(ds["slow_time"]):
    trace_unfiltered = ds["data"][idx,:]
    #padding = 100
    attenuation = 200
    ripple = 0.001

    # originally butterworth filters, now elliptic
    smile_filter1 = scipy.signal.ellip(5, ripple, attenuation, (a[idx], b[idx]), btype='bandstop', output='sos', fs=ds.fs)
    smile_filter2 = scipy.signal.ellip(5, ripple, attenuation, (c[idx], d[idx]), btype='bandstop', output='sos', fs=ds.fs)
    smile_filter3 = scipy.signal.ellip(5, ripple, attenuation, (e[idx], f[idx]), btype='bandstop', output='sos', fs=ds.fs)
    eyes_filter = scipy.signal.ellip(5, ripple, attenuation, (10e6,18.5e6), btype='bandstop', output='sos', fs=ds.fs)
    strip_filter = scipy.signal.ellip(5, ripple, attenuation, (16e6,18.1e6), btype='bandstop', output='sos', fs=ds.fs)

    trace_last = trace_unfiltered
    if t < 50:
        trace_filtered = scipy.signal.sosfilt(smile_filter1, trace_last)
    elif t > 80:
        trace_filtered = scipy.signal.sosfilt(smile_filter3, trace_last)
    else:
        trace_filtered = scipy.signal.sosfilt(smile_filter2, trace_last)

    trace_last = trace_filtered

    if idx > 510 and idx < 800:
        trace_filtered = scipy.signal.sosfilt(eyes_filter, trace_last)
    elif idx > 1700 and idx < 1990:
        trace_filtered = scipy.signal.sosfilt(eyes_filter, trace_last)

    trace_last = trace_filtered

    trace_filtered = scipy.signal.sosfilt(strip_filter, trace_last)

    final_trace[idx, :] = trace_filtered
    pulse_compressed = pulse_compress_one_trace(trace_filtered, ref_chirp_time_domain=ref_chirp)
    data_out[idx, :] = (20*np.log10(np.abs(pulse_compressed)))

  #  if idx > 510 and idx < 800:
   #     filter = scipy
    #    trace_filtered = scipy.signal.sosfilt(

plot_radargram(data_out, title="Filtered Radargram")

snr_lowpass, snr_avg_lowpass = estimate_snr(data_out, ds["bed_pick"])
snr_avg_lowpass = 10*np.log10(snr_avg_lowpass)
print('Filtered Average SNR = %.2f dB' % snr_avg_lowpass)
```

```
Filtered Average SNR = 36.36 dB
```



In [316…

```
window = 'flattop'
spectrogram_options = {
    "fs": ds.fs,
    "window": window,
    "nperseg": len(ds.fast_time),
    "noverlap": 0,
    "mode": "psd",
    "scaling": "density",
    "return_onesided": True
}

f, t, Sxx_test = scipy.signal.spectrogram(final_trace.flatten(), **spectrogram_options)

fig, ax = plt.subplots(1,1, figsize=(14, 8), facecolor='white')
p = plt.pcolormesh(ds["slow_time"], f/1e6, 10*np.log10(np.abs(Sxx_test)), shading='gouraud', cmap='inferno')#, vmin=-40, vmax=10)
clb = fig.colorbar(p, ax=ax)
clb.set_label('Power Spectral Density [dB/Hz]')
ax.set_xlabel('Time [s]')
ax.set_ylabel('Frequency [MHz]')
plt.show()
```