Mandatory project IN3190

Nikolai Kemi Engstad

Task 1: Convolution and frequency spectra

1a)

Will create a function that creates a convolution between two given signals.

In [37]:
```
#importing libraries

import numpy as np
import matplotlib.pyplot as plt
import scipy.io
from scipy.signal import find_peaks
from numba import jit, prange
```

In [126…
```
@jit(nopython=True, parallel=True)
def convin3190(x: np.ndarray, h: np.ndarray, ylen: int) -> np.ndarray:
    # Convolution of x and h
    # x: input signal
    # h: FIR filter
    # ylen: determine the length of the output signal, if ylen = 0, the length
of the output signal is len(x)
    # if ylen is 1, the length of the output signal is len(x) + len(h) - 1
    # return: convolution of x and h

    if ylen == 0:
        y = np.zeros(len(x))
    elif ylen == 1:
        y = np.zeros(len(x) + len(h) - 1)
    else:
        raise ValueError ('ylen must be 0 or 1') # raise an error if ylen is
not 0 or 1
    for i in prange(len(y)):
        for j in prange(len(h)):
            if i - j >= 0 and i - j < len(x): # check if the index is within
the range of x
                y[i] += x[i - j] * h[j] # compute the convolution
    return y
```

1b)

The number of points on the unit circle is N, we also know that the difference between each point is $\Delta f = \frac{f_s}{N}$ and that the maximum frequency is given by $f_{max} = \frac{f_s}{2}$. Where f_s is the samplings frequency.

In [39]:
```
def freqspecin3190(x: np.ndarray, N: int, fs: float):
    # Compute the frequency spectrum of x
    # x: input signal
    # N: number of samples of the frequency spectrum
    # fs: sampling frequency
```

```
    # return: frequency spectrum X and frequency axis f
    n = len(x)
    f_max = fs / 2 # maximum frequency
    f = np.linspace(0, f_max, N // 2) # frequency axis, only keep the first half because the frequency spectrum is symmetric
    X = np.zeros(N, dtype = complex) # frequency spectrum
    for i in range(N):
        for j in range(n):
            X[i] += x[j] * np.exp(-2j * np.pi * i * j / N) # compute the frequency spectrum
    X= X[:N//2] # only keep the first half because the frequency spectrum is symmetric
    return X, f
```

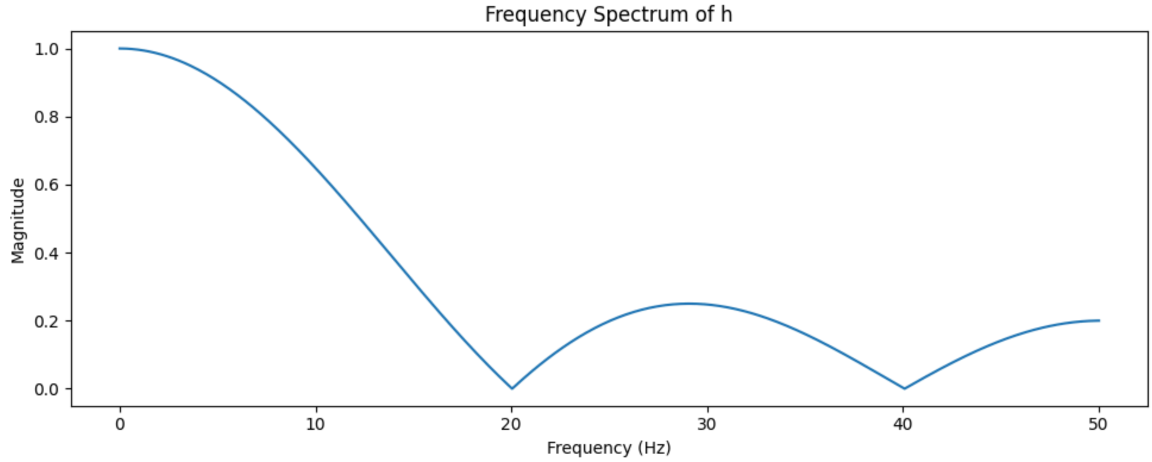
1c)

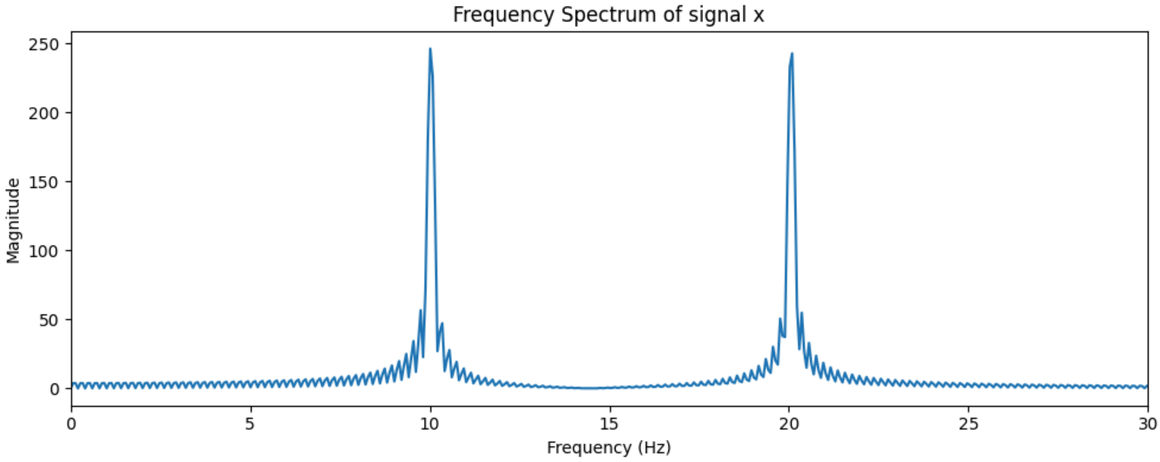
We will now test out funcions with given signal x(n) and a given FIR filter h(n)

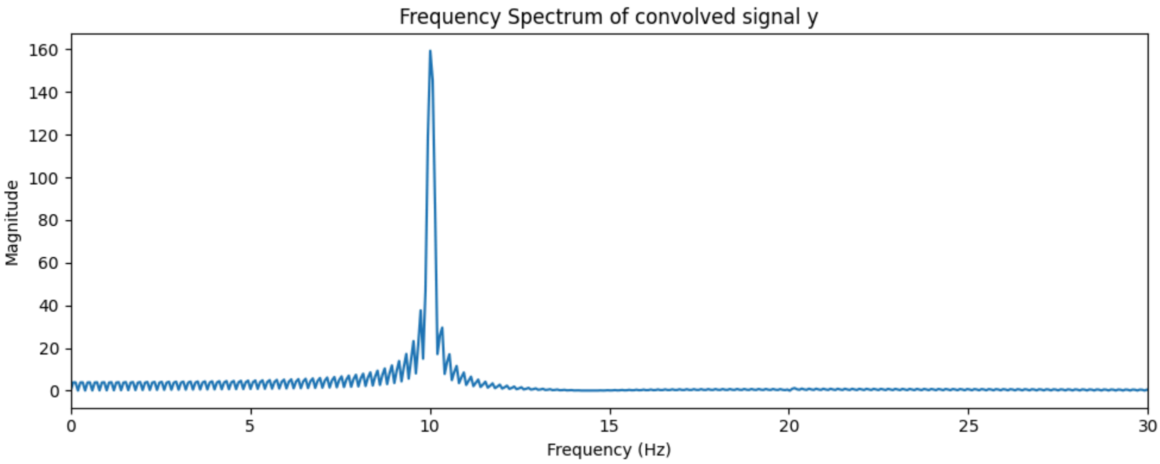
In [40]:

```
# Constants
f1 = 10
f2 = 20
fs = 100
t = np.linspace(0, 5, 5 * fs)
x = np.sin(2 * np.pi * f1 *t ) + np.sin(2 * np.pi * f2 * t) # generate a signal with two frequencies

def h(n: int): # generate the given FIR filter
    h = np.zeros(n)
    h[:5] = 1 / 5
    return h


H, fh = freqspecin3190(h(5), 1000, fs) # compute the frequency spectrum of h
X, fx = freqspecin3190(x, 1500, fs) # compute the frequency spectrum of x
Y, fy = freqspecin3190(convin3190(x, h(5), 1), 1500, fs) # compute the frequency spectrum of the convolved signal

fig, (ax1, ax2, ax3) = plt.subplots(3, 1, figsize=(10, 12))

# Plot the FIR filter's frequency spectrum
ax1.plot(fh, np.abs(H))
ax1.set_title('Frequency Spectrum of h')
ax1.set_xlabel('Frequency (Hz)')
ax1.set_ylabel('Magnitude')

# Plot the original signal's frequency spectrum
ax2.plot(fx, np.abs(X))
ax2.set_xlim(0, 30)
ax2.set_title('Frequency Spectrum of signal x')
ax2.set_xlabel('Frequency (Hz)')
ax2.set_ylabel('Magnitude')

# Plot the convolved signal's frequency spectrum
ax3.plot(fy, np.abs(Y))
ax3.set_xlim(0, 30)
ax3.set_title('Frequency Spectrum of convolved signal y')
ax3.set_xlabel('Frequency (Hz)')
ax3.set_ylabel('Magnitude')
```

```
plt.tight_layout()
plt.show()
```







We can see that for the frequency spectrum of h, the magnitude of 20 Hz is 0. The peaks of the frequecy spectrum of x is 10 Hz and 20 Hz, that matches the signal x. When we convolve the signals the magnitude of 20 is also 0. This means that we filtered out the higher frequencies of the signal. This means that the FIR filter h works as a lowpass filter.

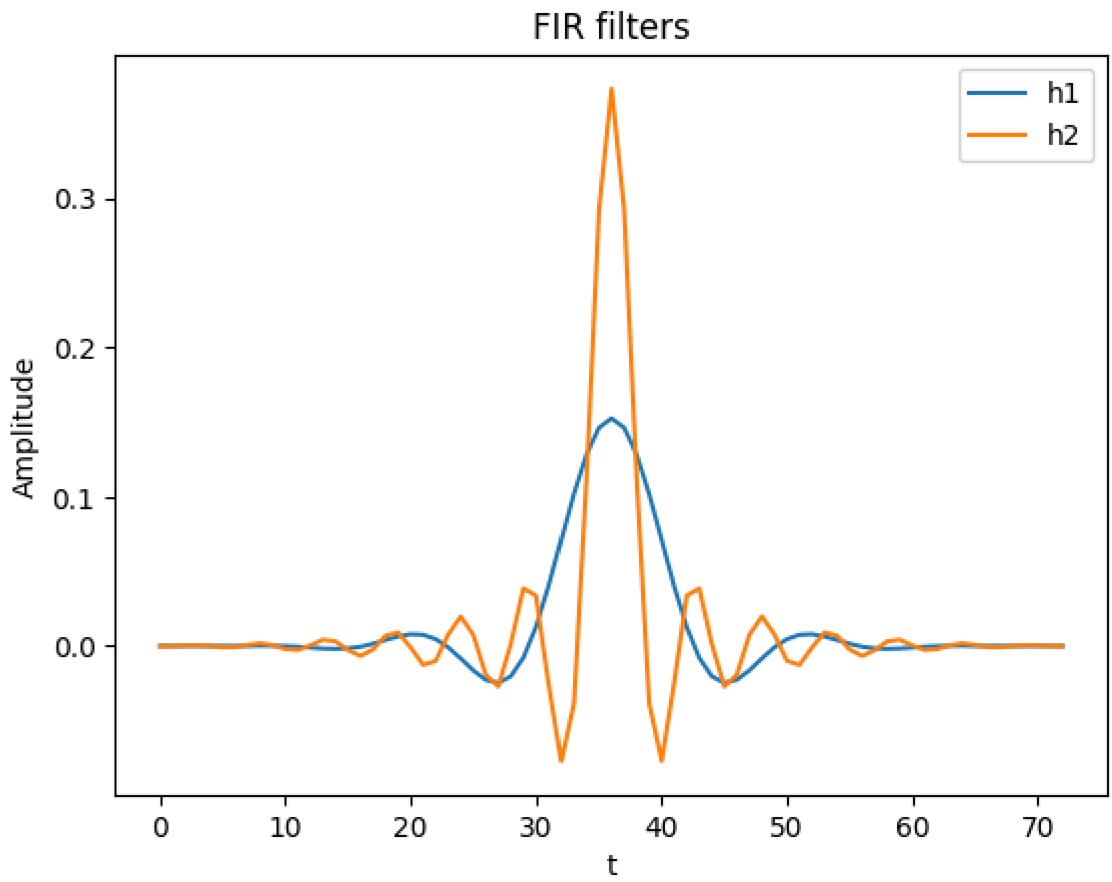
Task 2 - Noise removal

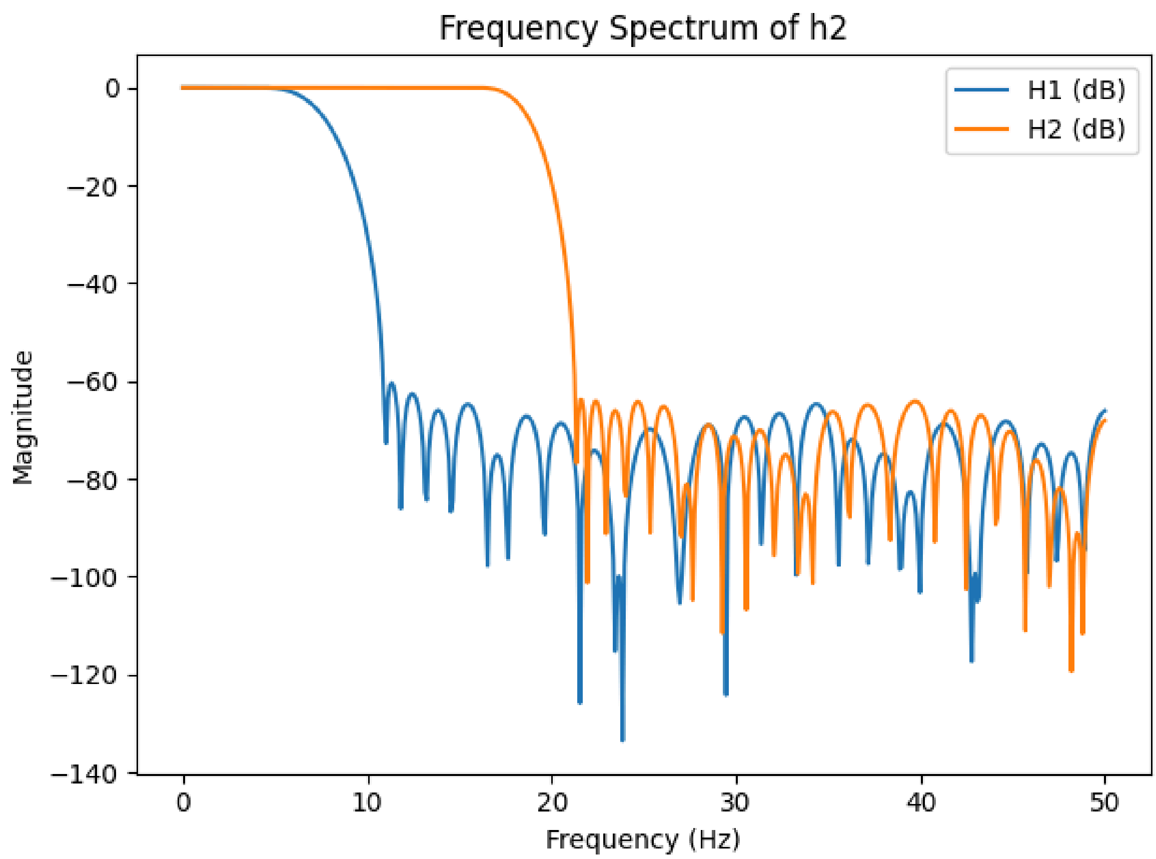
2a)

We will now test two different FIR filters (h1, h2). They will first be plotted as a function of time and then in a frequency spectrum to see the difference in the filters.

In [41]:
```
# Generate two given FIR filters
h1 = np.array([0.0002, 0.0001, -0.0001, -0.0005, -0.0011, -0.0017, -0.0019,
-0.0016, -0.0005, 0.0015,
               0.0040, 0.0064, 0.0079, 0.0075, 0.0046, -0.0009, -0.0084,
-0.0164, -0.0227, -0.0248,
               -0.0203, -0.0079, 0.0127, 0.0400, 0.0712, 0.1021, 0.1284,
0.1461, 0.1523, 0.1461,
               0.1284, 0.1021, 0.0712, 0.0400, 0.0127, -0.0079, -0.0203,
-0.0248, -0.0227, -0.0164,
               -0.0084, -0.0009, 0.0046, 0.0075, 0.0079, 0.0064, 0.0040,
0.0015, -0.0005, -0.0016,
               -0.0019, -0.0017, -0.0011, -0.0005, -0.0001, 0.0001, 0.0002])

h2 = np.array([-0.0002, -0.0001, 0.0003, 0.0005, -0.0001, -0.0009, -0.0007,
0.0007, 0.0018, 0.0005,
               -0.0021, -0.0027, 0.0004, 0.0042, 0.0031, -0.0028, -0.0067,
-0.0023, 0.0069, 0.0091,
               -0.0010, -0.0127, -0.0100, 0.0077, 0.0198, 0.0075, -0.0193,
-0.0272, 0.0014, 0.0386,
               0.0338, -0.0246, -0.0771, -0.0384, 0.1128, 0.2929, 0.3734,
0.2929, 0.1128, -0.0384,
               -0.0771, -0.0246, 0.0338, 0.0386, 0.0014, -0.0272, -0.0193,
0.0075, 0.0198, 0.0077,
               -0.0100, -0.0127, -0.0010, 0.0091, 0.0069, -0.0023, -0.0067,
-0.0028, 0.0031, 0.0042,
               0.0004, -0.0027, -0.0021, 0.0005, 0.0018, 0.0007, -0.0007,
-0.0009, -0.0001, 0.0005,
               0.0003, -0.0001, -0.0002])


h1 = np.pad(h1, (int((len(h2) - len(h1)) / 2), int((len(h2) - len(h1)) / 2)),
'constant') # pad h1 with zeros to make it the them on top of each other

# Plot the FIR filters in a time domain
plt.plot(h1, label='h1')
plt.plot(h2, label='h2')
plt.title('FIR filters')
plt.xlabel('t')
plt.ylabel('Amplitude')
plt.legend()
plt.show()

# Compute the frequency spectrum of the FIR filters
H1, fh1 = freqspecin3190(h1, 1000, fs)
H2, fh2 = freqspecin3190(h2, 1000, fs)

# Plot the frequency spectrum of the FIR filters
plt.plot(fh1, 20*np.log10(np.abs(H1)), label='H1 (dB)')
plt.title('Frequency Spectrum of h1')
plt.xlabel('Frequency (Hz)')
plt.ylabel('Magnitude')

plt.plot(fh2, 20*np.log10(np.abs(H2)), label='H2 (dB)')
plt.title('Frequency Spectrum of h2')
plt.xlabel('Frequency (Hz)')
plt.ylabel('Magnitude')

plt.legend()
```

```
plt.tight_layout()
plt.show()
```





Based on the frequency spectrum of the two filters, we see that h1 filters out lower frequencies than h2.

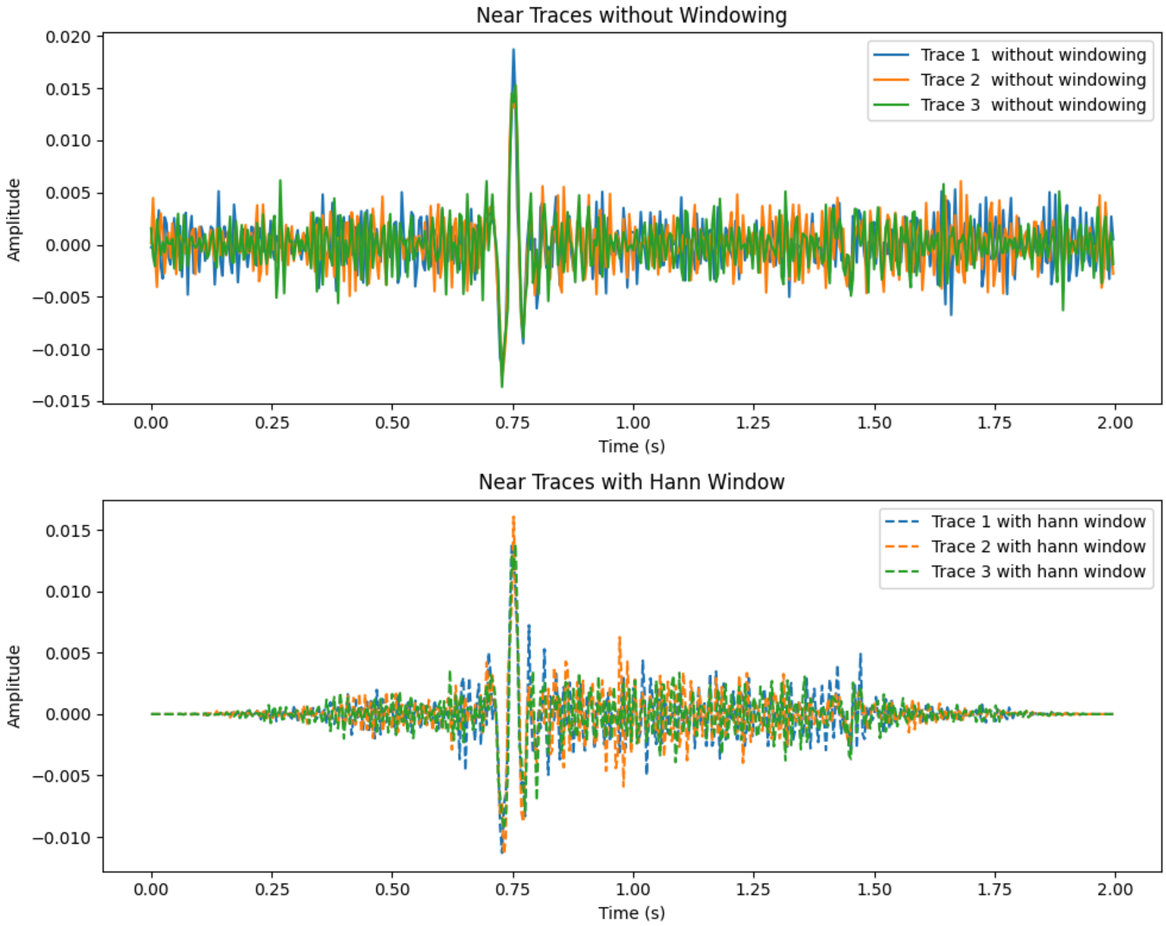
2b)

We will now load in marine seismic data. The data plotted will be the near traces with and without a window funcion. The chosen window funcion is the hanning window. It works by tapering the signal towards zero at the edges, it basically smoothly reduces the amplitude of the signal at the beginning and end of the window. I use the numpy funcion for the window, and its parameters is only the length of the array.

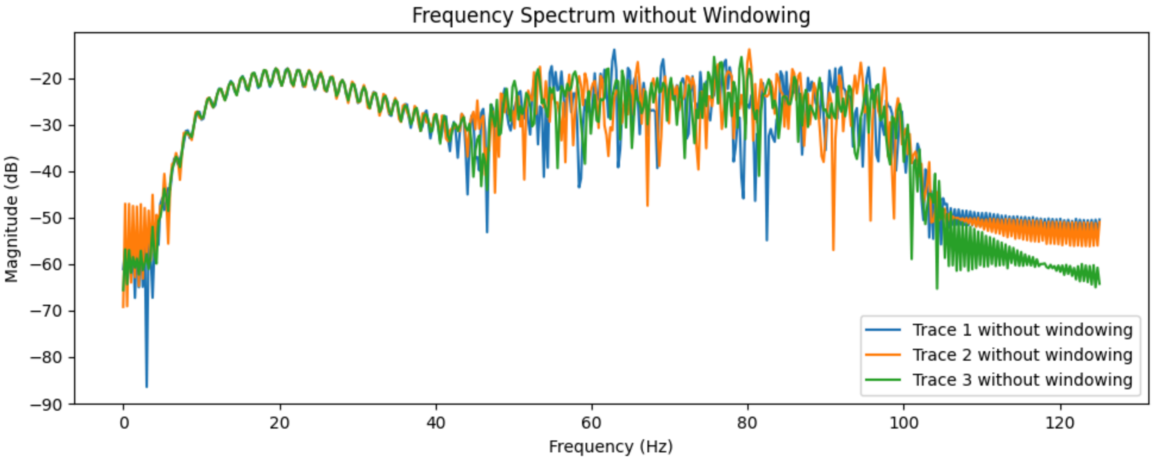
In [42]:

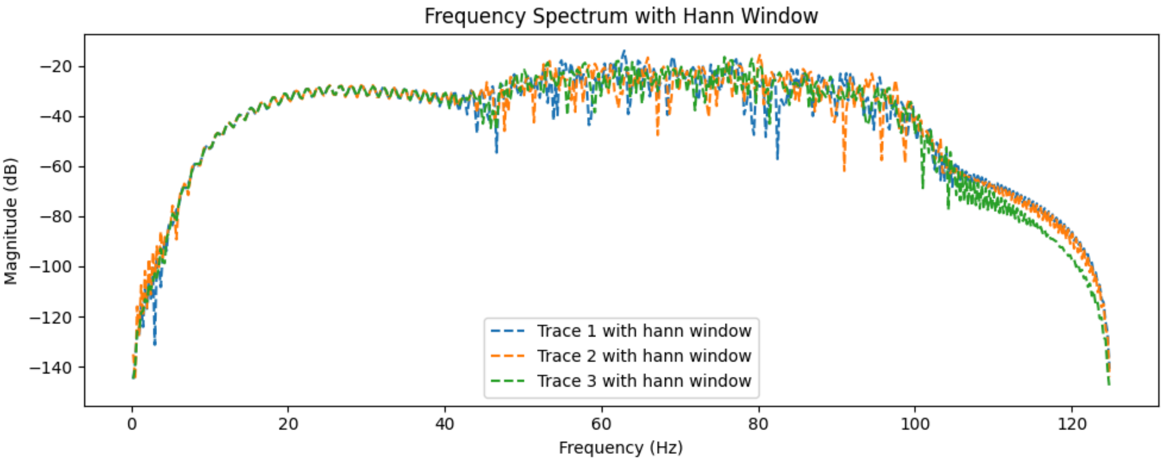
```
hann = np.hanning(500) # chosen window function
# Load in .mat-file
mat_data = scipy.io.loadmat('31.mat')

# Get the data from the .mat-file
offset1 = mat_data['offset1'].flatten()
offset2 = mat_data['offset2'].flatten()
seismogram1 = mat_data['seismogram1']
seismogram2 = mat_data['seismogram2']
t = mat_data['t'].flatten()

fs = len(t) / t[-1] # sampling frequency of the seismograms


# Apply Hann window to the first 500 samples of each trace in seismogram1 and
seismogram2
# Use only the first 500 samples for the shallow traces
seismogram1_windowed = seismogram1[:500, :] * hann[:, np.newaxis]
seismogram2_windowed = seismogram2[:500, :] * hann[:, np.newaxis]

fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(10, 8))
for i in range(3):
    ax1.plot(t[:500], seismogram1[:500, i], label=f"Trace {i+1}  without
windowing")
    ax2.plot(t[:500], seismogram2[:500, i]*hann, label=f"Trace {i+1} with hann
window", linestyle='--')


# Set titles and axis labels for ax1 (without windowing)
ax1.set_title('Near Traces without Windowing')
ax1.set_xlabel('Time (s)')
ax1.set_ylabel('Amplitude')
ax1.legend()

# Set titles and axis labels for ax2 (with windowing)
ax2.set_title('Near Traces with Hann Window')
ax2.set_xlabel('Time (s)')
ax2.set_ylabel('Amplitude')
ax2.legend()

# Adjust layout and show the plot
plt.tight_layout()
plt.show()

fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(10, 8))
# Compute the frequency spectrum of the seismograms
for i in range(3):
    Seismogram1, fSeismogram1 = freqspecin3190(seismogram1[:500, i], 1000, fs)
    ax1.plot(fSeismogram1, 20*np.log10(np.abs(Seismogram1)), label=f"Trace
```

```
{i+1} without windowing")
    ax2.plot(fSeismogram1, 20*np.log10(np.abs(Seismogram1*hann)), label=f"Trace
{i+1} with hann window", linestyle='--')
# Set titles and axis labels for ax1 (without windowing)
ax1.set_title('Frequency Spectrum without Windowing')
ax1.set_xlabel('Frequency (Hz)')
ax1.set_ylabel('Magnitude (dB)')
ax1.legend()

# Set titles and axis labels for ax2 (with windowing)
ax2.set_title('Frequency Spectrum with Hann Window')
ax2.set_xlabel('Frequency (Hz)')
ax2.set_ylabel('Magnitude (dB)')
ax2.legend()

# Adjust layout and show the plot
plt.tight_layout()
plt.show()
```




```
C:\Users\nikol\AppData\Local\Temp\ipykernel_8068\1782125051.py:47: RuntimeWarnin
g: divide by zero encountered in log10
  ax2.plot(fSeismogram1, 20*np.log10(np.abs(Seismogram1*hann)), label=f"Trace {i+
1} with hann window", linestyle='--')
```





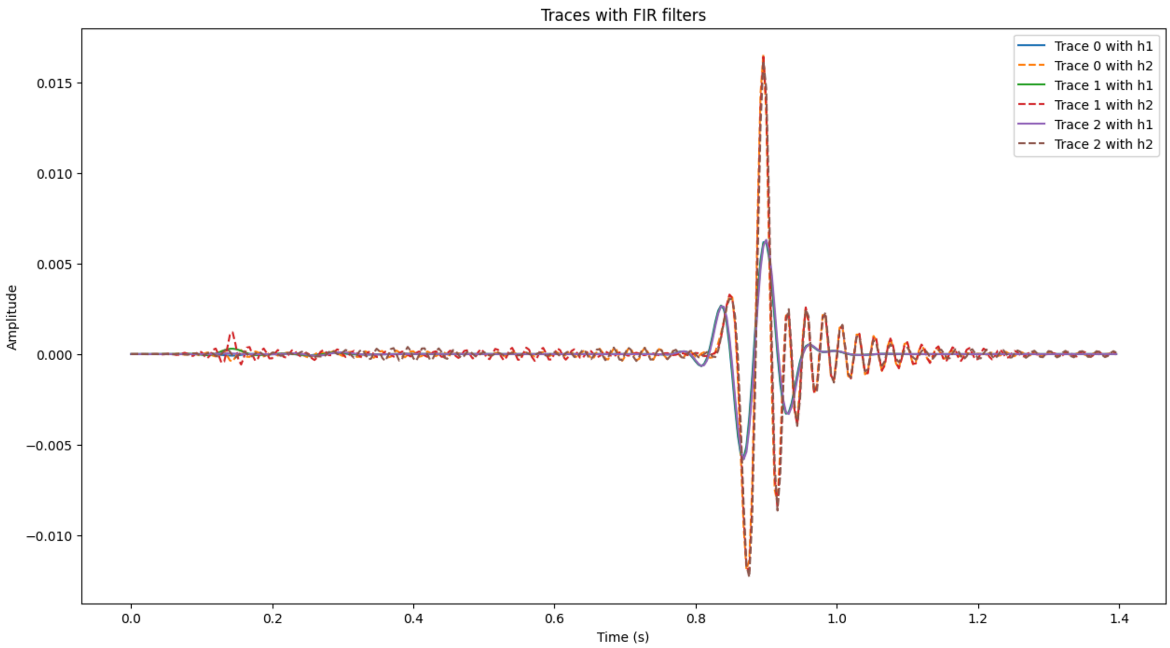
Alot of the noise comes from the higher frequencies > 40 Hz.

2c)

Will now convolve all of the date in seismogram1 with h1 and h2 to see the difference in the filters.

In [43]:
```
y1 = np.zeros_like(seismogram1)
y2 = np.zeros_like(seismogram1)
for i in range(len(seismogram1[0])):
    y1[:, i] = convin3190(seismogram1[:,i], h1, 0)
    y2[:, i] = convin3190(seismogram1[:,i], h2, 0)
```

In [44]:
```
plt.figure(figsize=(15, 8))  # Change the size of the plot
for i in range(3):
    plt.plot(t[:350], y1[:350, i], label=f"Trace {i} with h1")
    plt.plot(t[:350], y2[:350, i], label=f"Trace {i} with h2", linestyle='--')

plt.xlabel('Time (s)')
plt.ylabel('Amplitude')
plt.title('Traces with FIR filters')
plt.legend()
plt.show()
```



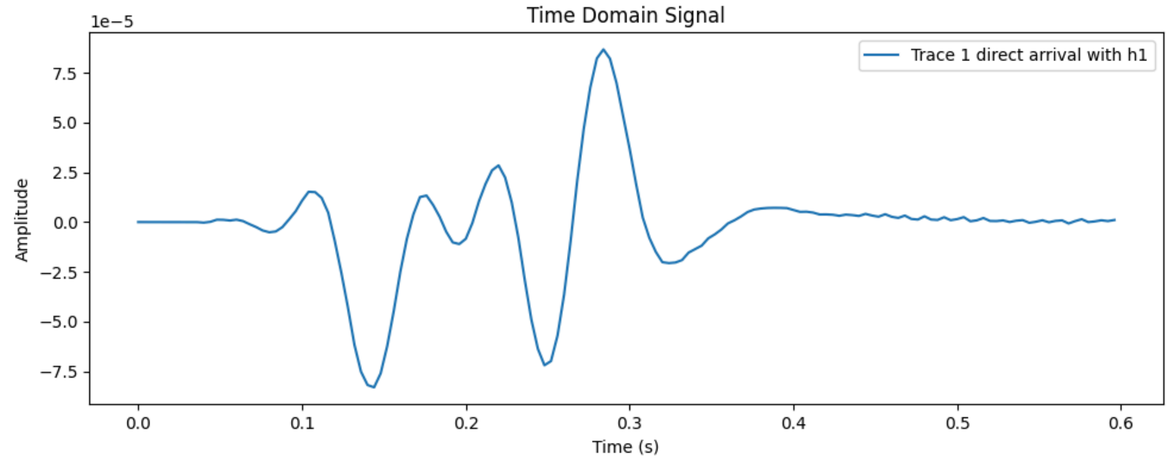
Based on what we have done i would argue that h1 is the better filter. Since it only keeps the most prevelent frequencies in the signal and gets rid of more of the noise than h2. If we look at the near trace plots for both filters we see that for h2 the signal propagetes longer than for h1.

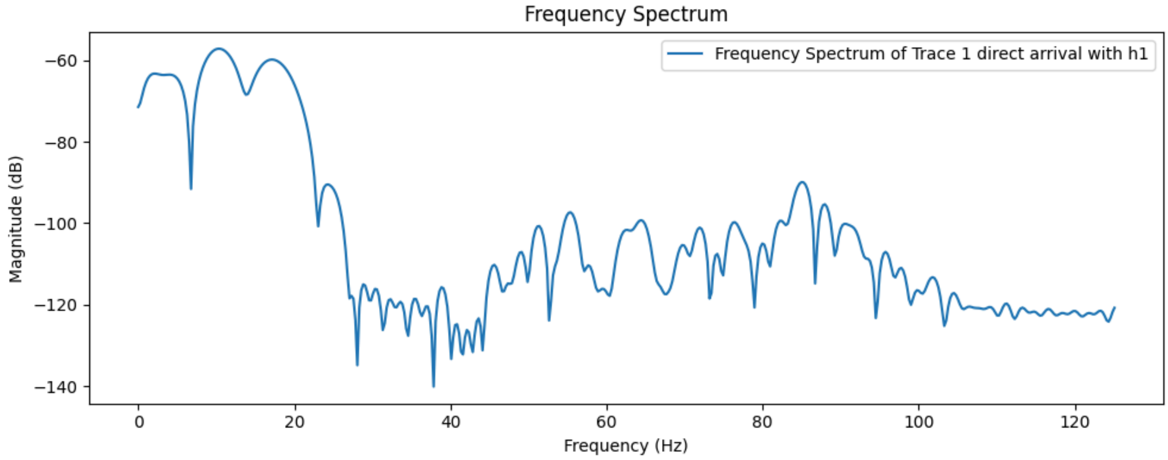
Task 3 - Far field signature

3a)

We will now plot the pulse for the air gun. We will use the the direct arrival of the wavefield through the water.

In [45]:
```
dir_arr = y1[:150, 0] # First trace, direct arrival
dir_arr_fft, d_a_f = freqspecin3190(dir_arr, 1000, fs) # Compute the frequency spectrum of the direct arrival
fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(10, 8))

# Plot the time domain signal
ax1.plot(t[:150], dir_arr, label='Trace 1 direct arrival with h1')
ax1.set_title('Time Domain Signal')
ax1.set_xlabel('Time (s)')
ax1.set_ylabel('Amplitude')
ax1.legend()

# Plot the frequency spectrum
ax2.plot(d_a_f, 20*np.log10(np.abs(dir_arr_fft)), label='Frequency Spectrum of Trace 1 direct arrival with h1')
ax2.set_title('Frequency Spectrum')
ax2.set_xlabel('Frequency (Hz)')
ax2.set_ylabel('Magnitude (dB)')
ax2.legend()

plt.tight_layout()
plt.show()
```





3b)

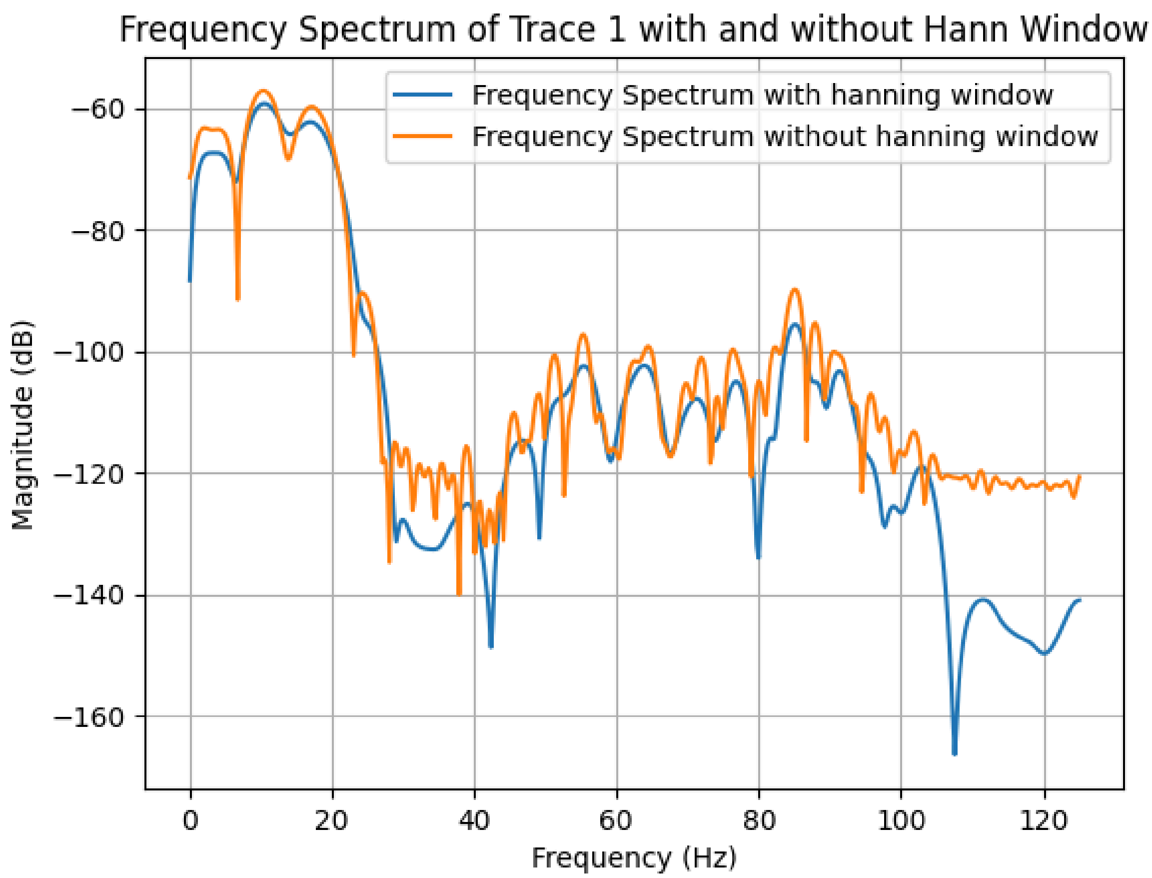
Will now add the same window function and plot the frequency spectrum with and without this window function.

In [46]:

```
hann = np.hanning(150) # Hann window of length 150
window_arr = dir_arr * hann # Apply the Hann window to the direct arrival
window_arr_fft, w_a_f = freqspecin3190(window_arr, 1000, fs) # Compute the 
frequency spectrum of the direct arrival with the Hann window

plt.plot(w_a_f, 20*np.log10(window_arr_fft), label='Frequency Spectrum with 
hanning window')
plt.plot(d_a_f, 20*np.log10(dir_arr_fft), label='Frequency Spectrum without 
hanning window')
plt.title('Frequency Spectrum of Trace 1 with and without Hann Window')
plt.xlabel('Frequency (Hz)')
plt.ylabel('Magnitude (dB)')
plt.grid(True)
plt.legend()

dominant_freq = w_a_f[np.argmax(window_arr_fft)]
print(f'Dominant frequency: {dominant_freq} Hz')
```

```
Dominant frequency: 10.277388109552437 Hz
```

```
c:\Users\nikol\AppData\Local\Programs\Python\Python310\lib\site-packages\matplotl
ib\cbook\__init__.py:1369: ComplexWarning: Casting complex values to real discard
s the imaginary part
  return np.asarray(x, float)
```



3c)

We are now trying to estimate the vertical resolution of imagery. We will be using the formulas $f = c/\lambda$ and $h = \lambda/8$. Where f is the dominant frequency we found earlier, c is speed of sound, λ is the wavelength and h is vertical resolution.

In [47]:
```
c = 3000 # speed of sound in m/s
wave_length = c / dominant_freq
vertical_resolution = wave_length / 8
print(f"Vertical resolution: {vertical_resolution} m")
```

```
Vertical resolution: 36.48786987536764 m
```

Task 4 - Reflections and multiples

4a)

We will now find the primary reflection and its multiples aswell as finding the time difference between them.

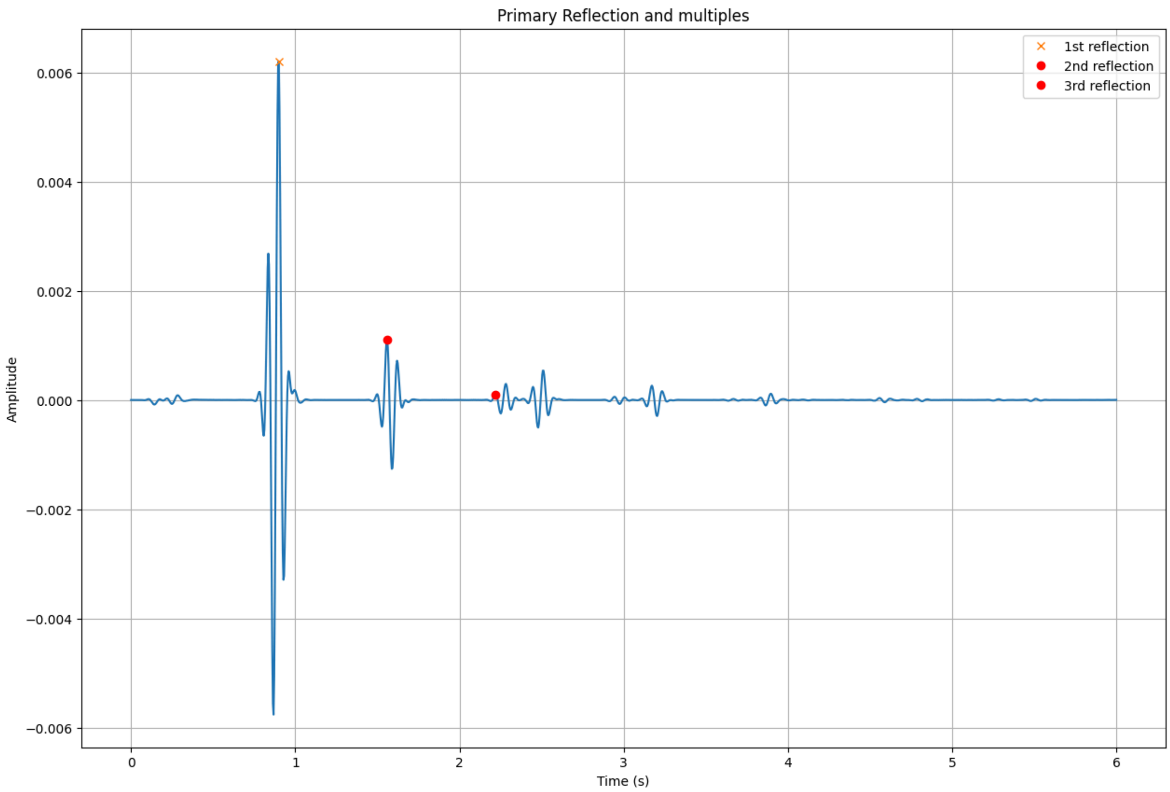
In [81]:
```
trace = y1[:, 0] # First trace
peaks = find_peaks(trace, distance= 50, height= 0.0006)[0] # Find the peaks to determine the reflections

difference = np.diff(peaks)  # Find the difference between the first and second peak indexes
difference_t = t[difference] # Find t_w
print(f"t_w is {difference_t[0]} s")
```

```
t_w is 0.66 s
```

Since we know the difference between the reflections and we only have the first two reflections we can use the difference to find the next multiples since the time difference is constant.

In [82]:
```python
plt.figure(figsize=(15, 10))
plt.plot(t, trace)
plt.plot(t[peaks[0]], y1[peaks[0], 0], 'x', label='1st reflection')
plt.plot(t[peaks[1]], y1[peaks[1], 0], 'ro', label='2nd reflection')
plt.plot(t[peaks[1] + difference], y1[peaks[1] + difference, 0], 'ro',
label='3rd reflection') # Use difference to find the 3rd reflection since the
difference is the same
plt.xlabel('Time (s)')
plt.ylabel('Amplitude')
plt.title('Primary Reflection and multiples')
plt.grid(True)
plt.legend()
```

Out[82]: `<matplotlib.legend.Legend at 0x1a1bf4c8580>`

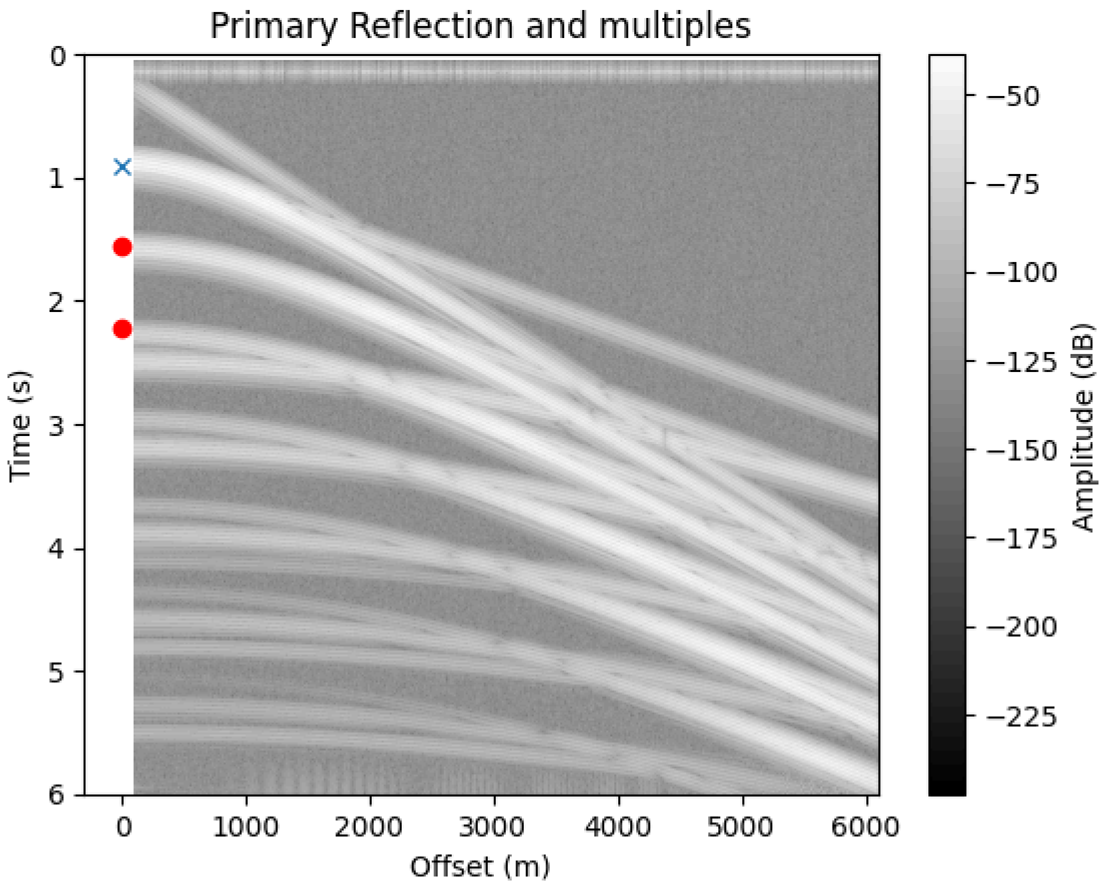


In [83]:
```python
extent = [offset1.min(), offset1.max(), t.max(), t.min()]

plt.imshow(20*np.log10(np.abs(y1)), extent = extent, aspect='auto', cmap =
"gray")
plt.colorbar(label = 'Amplitude (dB)')
plt.plot(0, t[peaks[0]], 'x', label='1st reflection')
plt.plot(0, t[peaks[1]], 'ro', label='2nd reflection')
plt.plot(0, t[peaks[1] + difference], 'ro', label='3rd reflection')
plt.xlabel('Offset (m)')
plt.ylabel('Time (s)')
plt.title('Primary Reflection and multiples')
```

```
C:\Users\nikol\AppData\Local\Temp\ipykernel_8068\447880957.py:3: RuntimeWarning: divide by zero encountered in log10
  plt.imshow(20*np.log10(np.abs(y1)), extent = extent, aspect='auto', cmap = "gray")
```

Out[83]:

```
Text(0.5, 1.0, 'Primary Reflection and multiples')
```



Here you can see the first 3 reflections marked.

4b)

From the plot above we can see that there are reflection other than the multiples we found, we know this since these dont follow the t_w that we have found. Based on the plot above this starts after the third reflection. These uncounted for reflections are reflections from the deepest sedimentary layer.

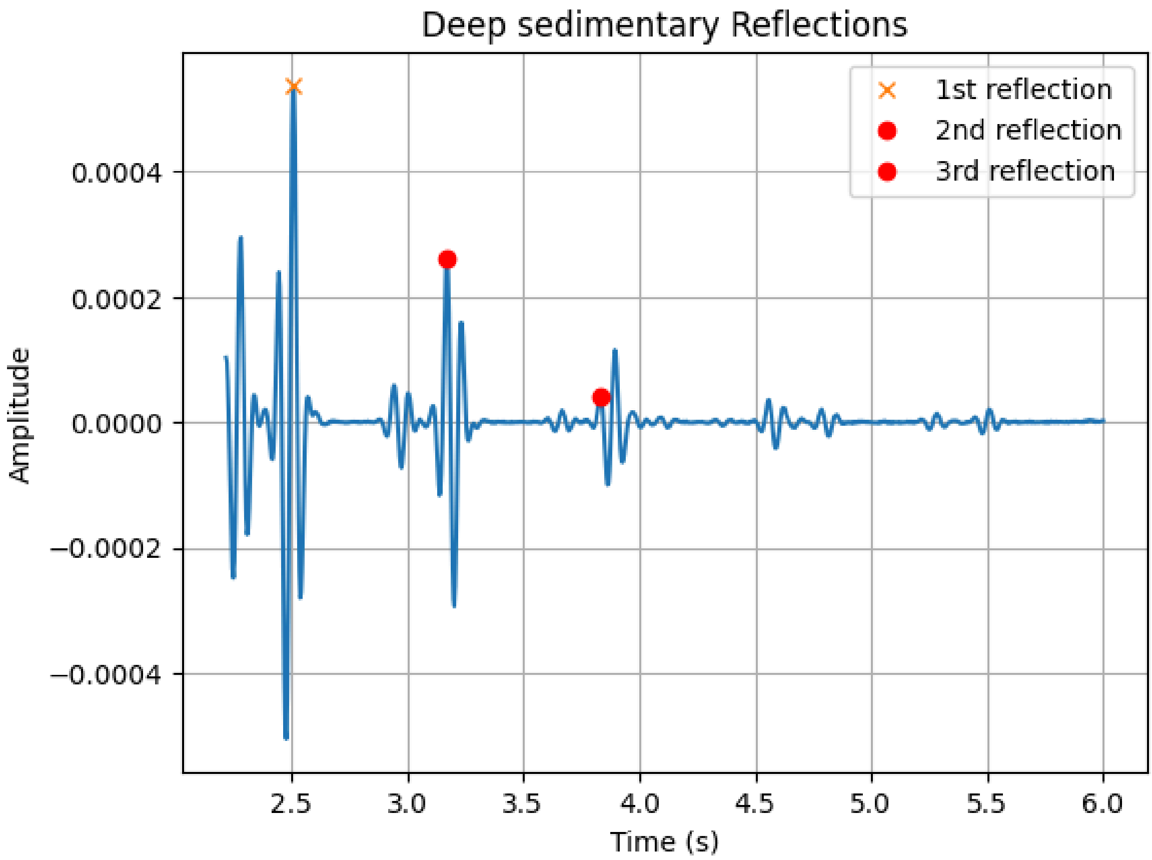
In [51]:

```
print(peaks[1] + difference) # The index of the 3rd reflection
deep_peaks = find_peaks(trace[555:], distance= 100, height= 0.00015)[0]
difference_deep = np.diff(deep_peaks) # Find the difference between the first and second peak indexes
difference_t_deep = t[555:][difference_deep] # Find t_w for the deep reflections

plt.plot(t[555:], trace[555:]) # Start from 555 to avoid the direct arrival, based on the index of the 3rd reflection
plt.plot(t[555:][deep_peaks[0]], trace[555:][deep_peaks[0]], 'x', label='1st reflection')
plt.plot(t[555:][deep_peaks[1]], trace[555:][deep_peaks[1]], 'ro', label='2nd reflection')
plt.plot(t[555:][deep_peaks[1] + difference_deep], trace[555:][deep_peaks[1] + difference_deep], 'ro', label='3rd reflection')
```

```
plt.title("Deep sedimentary Reflections")
plt.xlabel('Time (s)')
plt.ylabel('Amplitude')
plt.grid(True)
plt.legend()
```

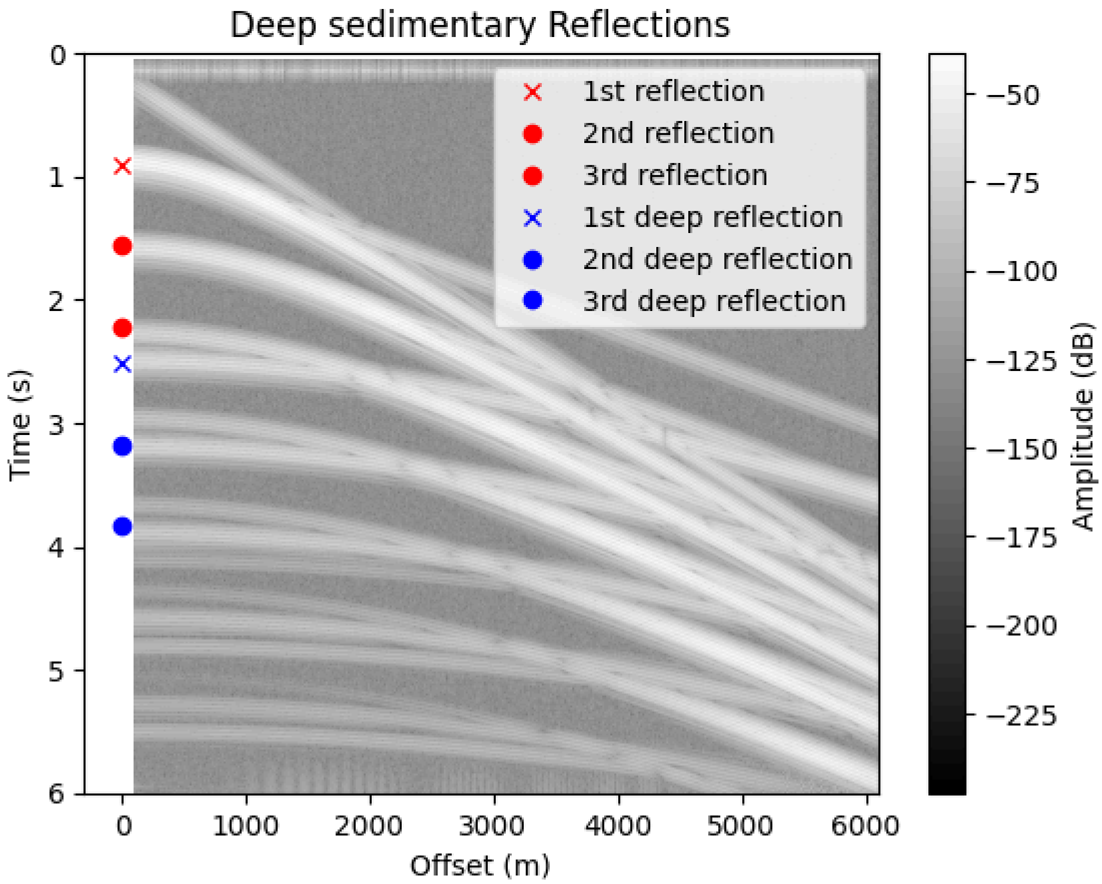
```
[-45]
```

Out[51]: <matplotlib.legend.Legend at 0x1a1b62aef20>



In [87]:
```
plt.imshow(20*np.log10(np.abs(y1)), extent = extent, aspect='auto', cmap =
"gray")
plt.colorbar(label = 'Amplitude (dB)')
plt.plot(0, t[peaks[0]], 'rx', label='1st reflection')
plt.plot(0, t[peaks[1]], 'ro', label='2nd reflection')
plt.plot(0, t[peaks[1] + difference], 'ro', label='3rd reflection')
plt.plot(0, t[deep_peaks[0]+555], 'bx', label='1st deep reflection') # Add 555
to compensate starting from 555
plt.plot(0, t[deep_peaks[1]+555], 'bo', label='2nd deep reflection') # Add 555
to compensate starting from 555
plt.plot(0, t[deep_peaks[1] + difference_deep +555], 'bo', label='3rd deep
reflection') # Add 555 to compensate starting from 555
plt.xlabel('Offset (m)')
plt.ylabel('Time (s)')
plt.title("Deep sedimentary Reflections")
plt.legend()
```

```
C:\Users\nikol\AppData\Local\Temp\ipykernel_8068\3062386820.py:1: RuntimeWarning:
divide by zero encountered in log10
  plt.imshow(20*np.log10(np.abs(y1)), extent = extent, aspect='auto', cmap = "gra
y")
2.508
```



Here we see the deep reflections marked.

Task 5 - Water velocity

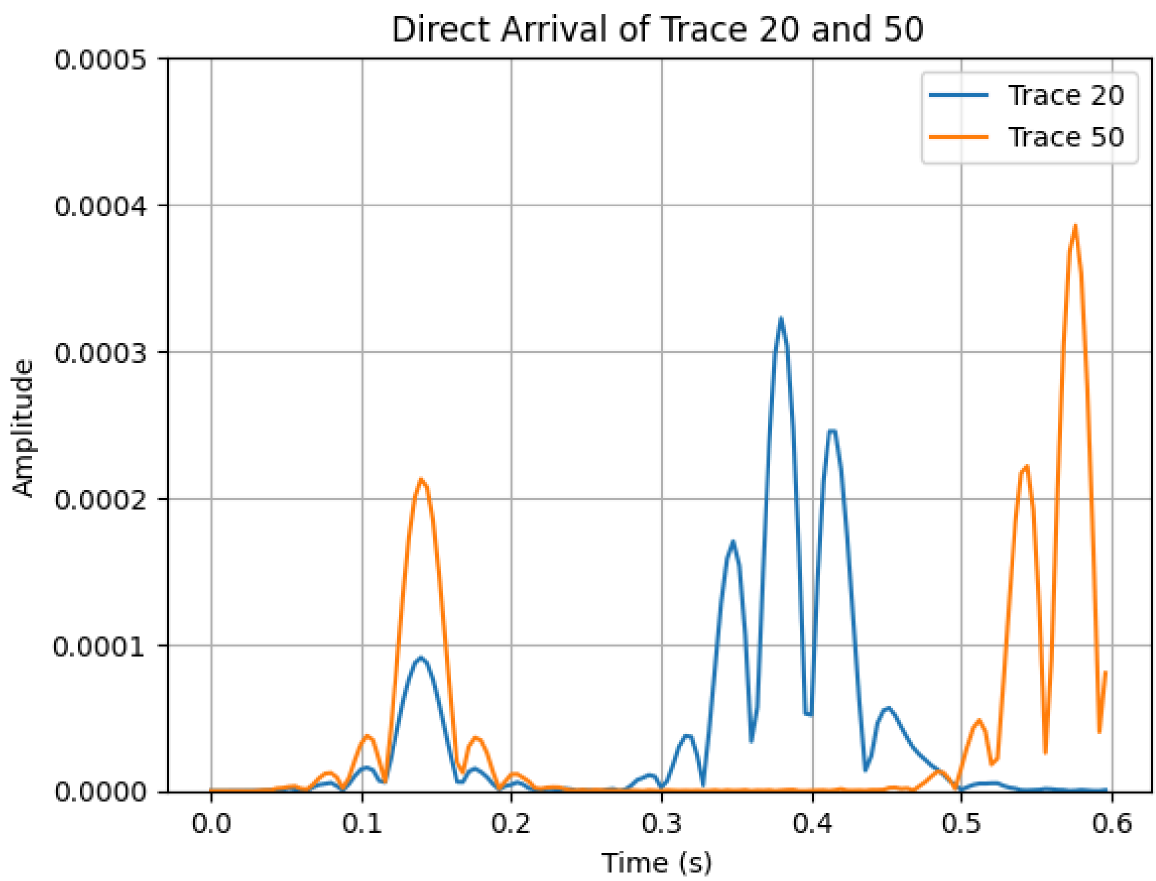
To find the water velocity we will be using two traces at two different offsets. We will specifically be looking at the direct arrival (The second spike for both traces). We will also find the distance to the offset using the data in offset1. We estimate the time by finding when the direct arrival arrives. Will then be using the formula $v = s/t$

In [53]:

```
trace_20 = y1[:150, 20] # Trace 20 direct arrival
trace_50 = y1[:150, 50] # Trace 50 direct arrival

plt.plot(t[:150], np.abs(trace_20), label='Trace 20')
plt.plot(t[:150], np.abs(trace_50), label='Trace 50')
plt.grid(True)
plt.ylim(0, 0.0005) # Set the y-axis limit to 0.0005 so that the plot is more
clear
plt.xlabel('Time (s)')
plt.ylabel('Amplitude')
plt.title('Direct Arrival of Trace 20 and 50')
plt.legend()
```

Out[53]:

```
<matplotlib.legend.Legend at 0x1a1b07f8040>
```



In [54]:
```
peak_20 = find_peaks(np.abs(trace_20), height= (0.0003, 0.00035))[0] # Find the index of direct arrival for trace 20
peak_50 = find_peaks(np.abs(trace_50)  ,height= (0.0003, 0.0004))[0] # Find the index of direct arrival for trace 50


direct_arrival_20 = t[peak_20[0]] # Find the direct arrival time for trace 20
direct_arrival_50 = t[peak_50[0]] # Find the direct arrival time for trace 50


offset20 = offset1[20] # Find the offset for trace 20
offset50 = offset1[50] # Find the offset for trace 50

velocity = (offset50 - offset20) / (direct_arrival_50 - direct_arrival_20) # Compute the velocity
print(f"Velocity of sound in water: {velocity:.5} m/s")
```

```
Velocity of sound in water: 1530.6 m/s
```

This result is a good estimate of the velocity of sound in water.

Task 6 - Sediment velocity 1

6a)

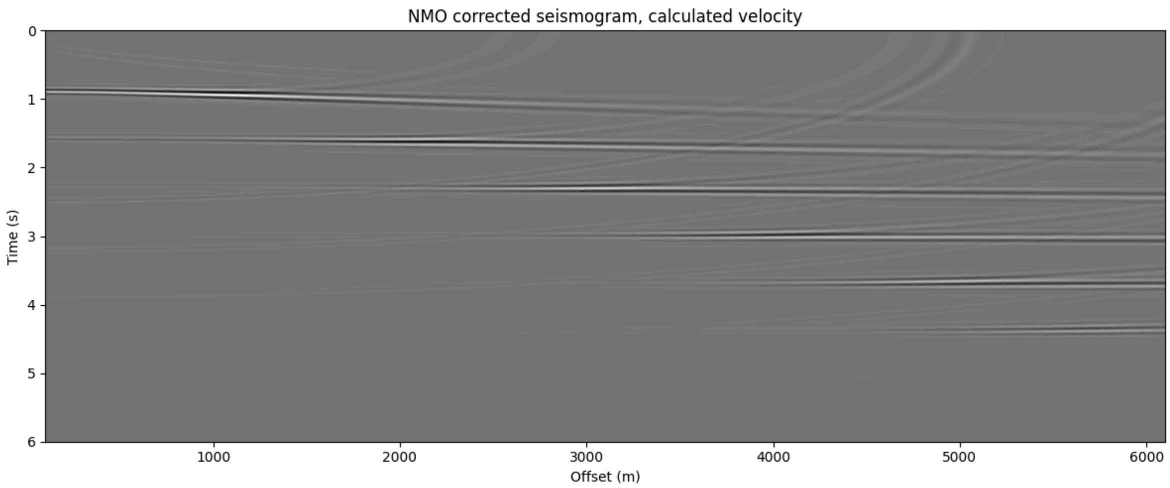
We will be using a given funcion nmo_correction that will flatten the seabed reflections. Will also try three different velocities to see how this affects the flatness. The velocity used will be the calculated water velocity 1530 m/s and where we add 100 m/s and 200 m/s

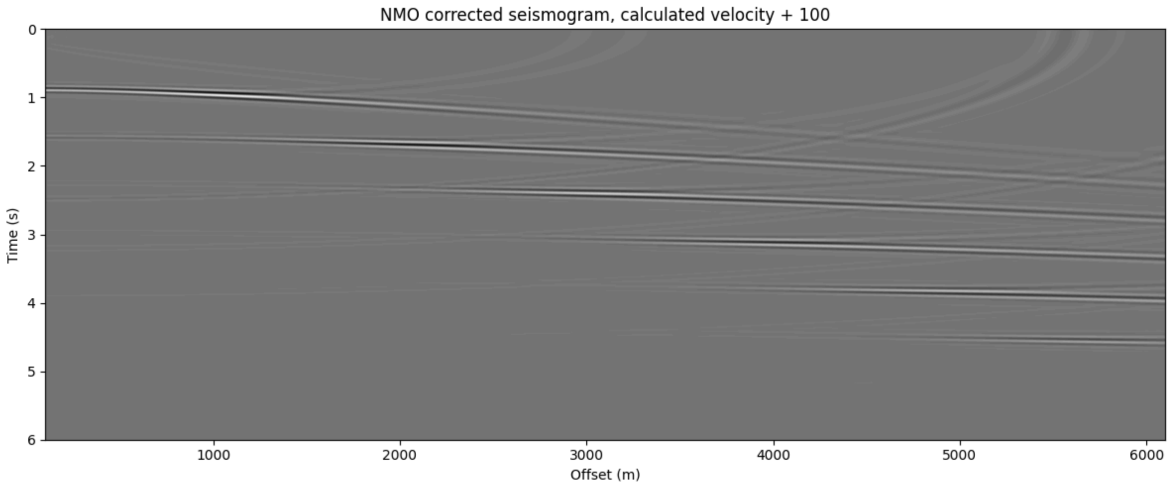
In [88]:
```
from nmo_correction import nmo_correction

v = np.ones(1501) * velocity # Create an array of the same length as the trace
with the velocity value
dt = t[1] - t[0] # Find the time difference between two samples

y1_nmo = nmo_correction(y1, dt ,offset1, v) # Apply NMO with calculated
velocity
```

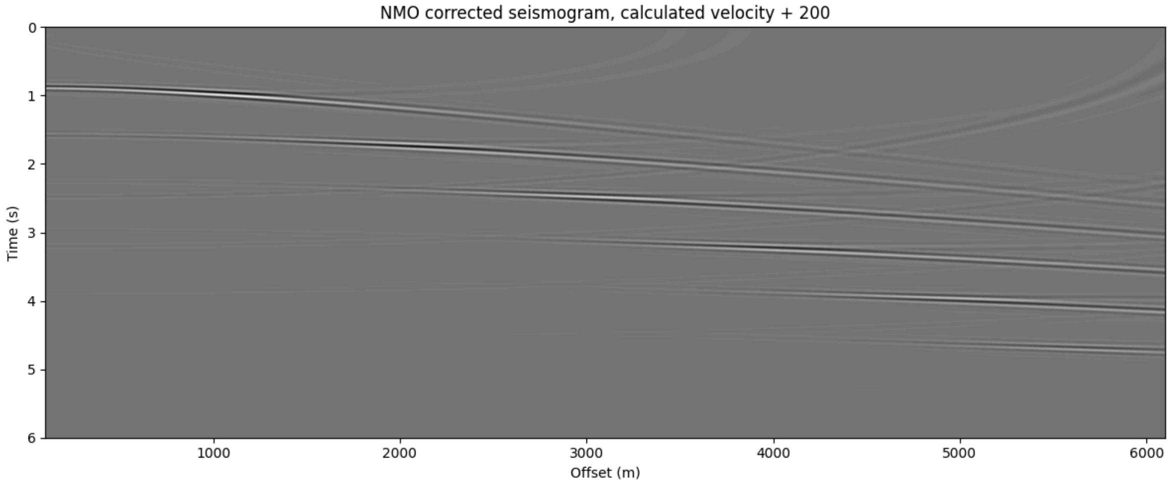
In [89]:
```
y1_nmo_2 = nmo_correction(y1, dt ,offset1, v + 100) # Apply NMO with velocity +
100 m/s
```

In [90]:
```
y1_nmo_3 = nmo_correction(y1, dt ,offset1, v + 200) # Apply NMO with velocity +
200 m/s
```

In [91]:
```
fig, ax = plt.subplots(3, 1, figsize=(12, 15))
ax[0].imshow(y1_nmo, extent = extent, aspect='auto', cmap = "gray",
label="Calculated velocity")
ax[1].imshow(y1_nmo_2, extent = extent, aspect='auto', cmap = "gray",
label="Calculated velocity + 100 m/s")
ax[2].imshow(y1_nmo_3, extent = extent, aspect='auto', cmap = "gray",
label="Calculated velocity + 200 m/s")
for i in range(3):
    ax[i].set_xlabel('Offset (m)')
    ax[i].set_ylabel('Time (s)')
ax[0].set_title('NMO corrected seismogram, calculated velocity')
ax[1].set_title('NMO corrected seismogram, calculated velocity + 100')
ax[2].set_title('NMO corrected seismogram, calculated velocity + 200')
plt.tight_layout()
```







Based on the plots we can see that there is some difference in flatness when we increase the velocity. Its not extremely significant but we can see a difference.

6b)

Will now plot the the NMO-corrected gather with constant velocity in water and sedimentary layers and one plot with smooth transition.

In [122…

```
v1 = np.ones(500) * 1530
v2 = np.ones(1001) * 2800

sudden_velocity = np.concatenate((v1 , v2))
```

```
v_1 = np.ones(500) * 1530 # the water layer is constant
vs = np.linspace(1530, 2800, 250) # transition between water and sedimentary layer
v_2 = np.ones(751) * 2800 # sedimentary layer
smooth_velocity = np.concatenate((v_1, vs, v_2)) # The full smooth velocity layer
```

In [123…

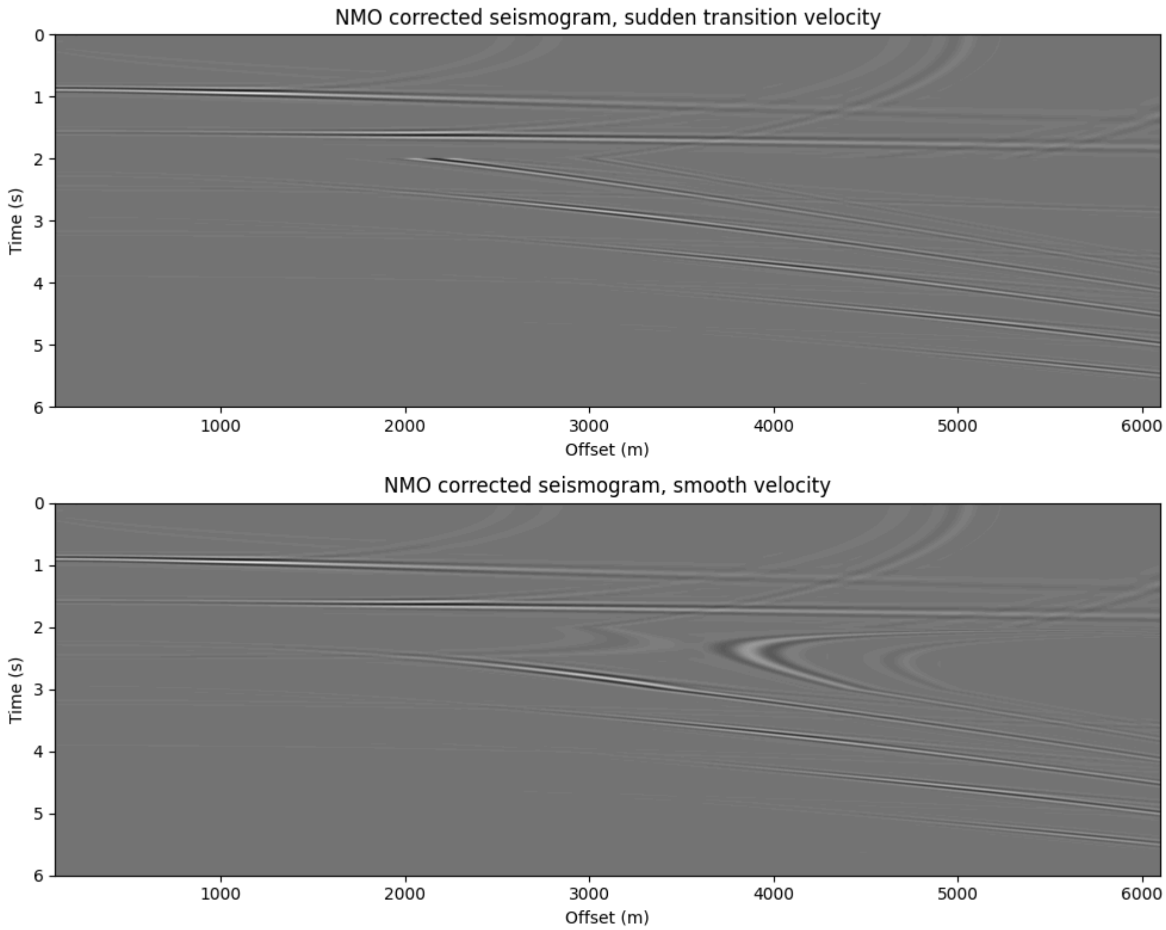
```
y1_smooth_nmo = nmo_correction(y1 , dt , offset1 , smooth_velocity)
```

In [124…

```
y1_real_nmo = nmo_correction(y1 , dt , offset1 , sudden_velocity)
```

In [125…

```
fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(10, 8))
ax1.imshow(y1_real_nmo, extent = extent, aspect='auto', cmap = "gray")
ax1.set_xlabel('Offset (m)')
ax1.set_ylabel('Time (s)')
ax1.set_title('NMO corrected seismogram, sudden transition velocity')

ax2.imshow(y1_smooth_nmo, extent = extent, aspect='auto', cmap = "gray")
ax2.set_xlabel('Offset (m)')
ax2.set_ylabel('Time (s)')
ax2.set_title('NMO corrected seismogram, smooth velocity')

plt.tight_layout()
```



Task 7 - Sediment velocity 2

7a)

To find the speed of refraction we can look at the linear line from the primary reflection in the gather plot in task 4b, since this is the layer just below the water. If we look at how long it will take for that line to reach 6000 m, we can calculate the speed using the formula from task 5. By looking at the plot we can see that this is roughly in 2.1 s. The velocity is then $v = 2860 m/s$

7b)

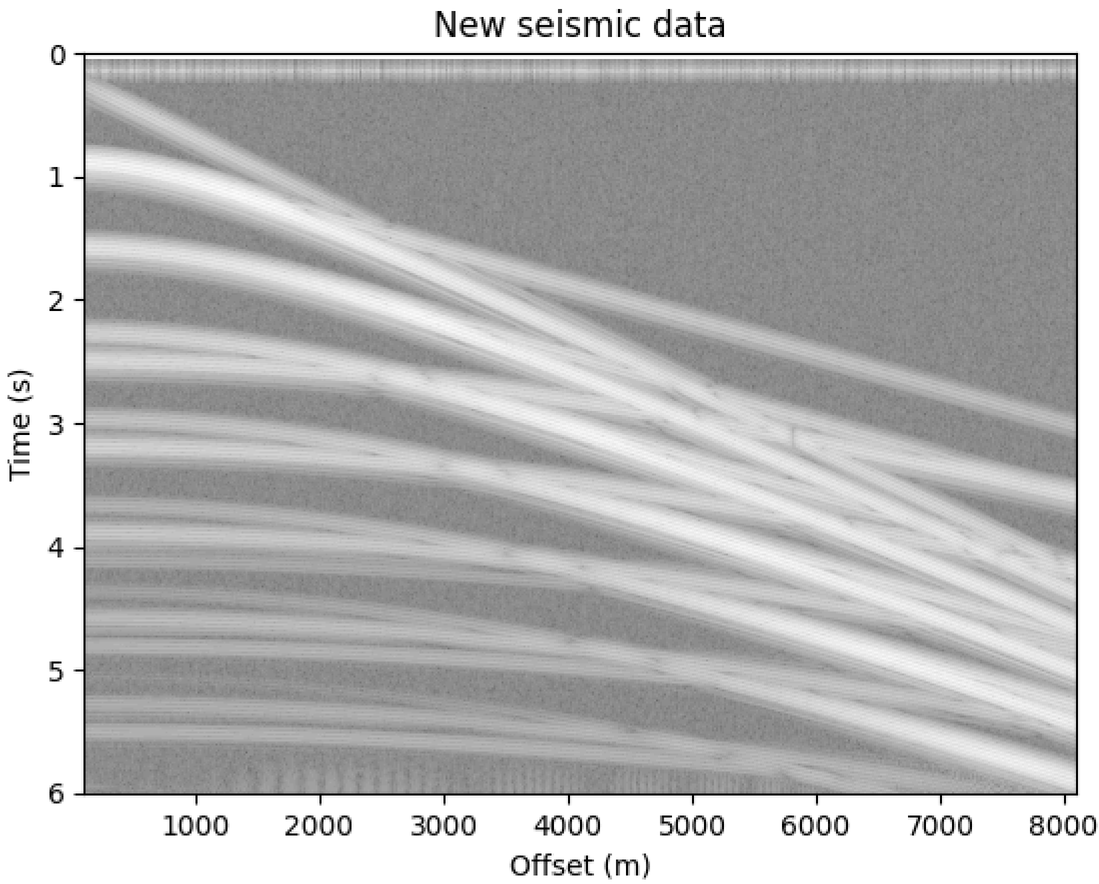
We will use the same method here but look at the second reflection, it takes roughly 1.4 seconds the velocity is then $v = 4285 m/s$

7c)

In [27]:
```
new_y1 = np.zeros_like(y1)
for i in range(len(y1[0])):
    new_y1[:, i] = convin3190(seismogram2[:,i], h1, 0)

extent = [offset2.min(), offset2.max(), t.max(), t.min()]
plt.imshow(20*np.log10(np.abs(new_y1)), extent = extent, aspect='auto', cmap = "gray")
plt.xlabel('Offset (m)')
plt.ylabel('Time (s)')
plt.title("New seismic data")
```

```
C:\Users\nikol\AppData\Local\Temp\ipykernel_8068\2444247562.py:6: RuntimeWarning: divide by zero encountered in log10
  plt.imshow(20*np.log10(np.abs(new_y1)), extent = extent, aspect='auto', cmap = "gray")
```

Out[27]:
```
Text(0.5, 1.0, 'New seismic data')
```



By using the same method as earlier, we can see that it now takes rougly 2.5s to go from 0 to 8000m, this means that the velocity is rougly $v = 3200m/s$. Which i find more reasonable. The difference comes from the fact that there are more sensors and there will be less variation so that the average velocity over this distance will be more correct.

7d)

To find the depth of the layers we need the time it takes to reach them. Since we have $t_w = 0.66s$ from ealier tasks, which is the time it takes for the reflections to the sensor we find the depth.

$$d_{water} = v_w \cdot t_w/2 = 1530m/s \cdot 0.66s/2 = 990m$$

To find the deeper sediment layers we can use t_w and the time for the refraction to the senson which is 2.508s.

$$t_{deep} = t_w + t_sediment \rightarrow t_{sediment} = t_{deep} - t_w = 2.508s - 0.66s = 1.848s$$

$$d_{deep} = d_{water} + v_{sediment} \cdot t_{sediment}/2 = 990m + 2860m/s \cdot 1.848s/2 = 3632m$$

Description	Value
Depth sedimentary layer 1	990 m
Depth sedimentary layer 2	3632 m
Velocity of water layer	1530 m/s
Velocity of sedimentary layer 1	2860 m/s

Description	Value
Velocity of sedimentary layer 1	3200 m/s