ESE 2240 Spring 2025 Lab 4 Group 3

Group members: Celia Tung, Laura Gao, Mohamed Elsheshtawy

https://colab.research.google.com/drive/1vgFXoSaQwuyFrtXPOaMny2EeCGmlZD9N?usp=sharing

[ ]:
```
!pip install -q numpy matplotlib
```

[ ]:
```
import numpy as np
import matplotlib.pyplot as plt

plt.rcParams["figure.dpi"] = 100
```

0. Provided Functions

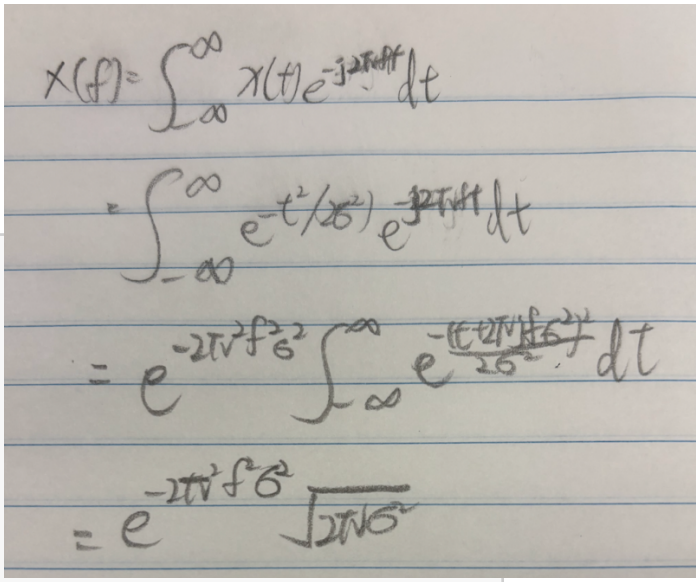
[ ]:
```
def compute_fft(x, fs: int, center_frequencies=True):
    """
    Compute the DFT of x.

    Args:
        x: Signal of length N.
        fs: Sampling frequency.
        center_frequencies: If true then returns frequencies on [-f/2,f/2]. If␣
 ↪false then returns frequencies between [0,f].
    Returns:
        X: DFT of x.
        f: Frequencies
    """
    N = len(x)
    X = np.fft.fft(x, norm="ortho")
    f = np.fft.fftfreq(N, 1/fs)
    if center_frequencies:
        X = np.fft.fftshift(X)
        f = np.fft.fftshift(f)
    return X, f

def compute_ifft(X, fs: int, center_frequencies=True):
    """
    Compute the iDFT of X.
```

```
    Args:
        X: Spectrum  of length N.
        fs: Sampling frequency.
        center_frequencies: If true then X is defined over the frequency range␣
 ↪[-f/2,f/2]. If false then [0,f].
    Returns:
        x: iDFT of X and it should be real.
        t: real time instants in the range [0, T]
    """
    N = len(X)
    if center_frequencies:
      X = np.fft.ifftshift(X)
    x = np.fft.ifft(X, norm="ortho")
    t = np.arange(N)/fs
    return x.real, t

def plot_signal_and_spectrums(x, t, X, f, title=None):
    """
    Plot a pulse x and its dft X.

    Args:
        x: (N,) Signal of length N.
        t: (N,) times
        X: (N,) DFT of x.
        f: (N,) frequencies
    """
    fig, ax = plt.subplots(2, 1, figsize=(6,6))
    fig.suptitle(title)

    ax[0].plot(t, x)
    ax[0].set_xlabel("t (s)")
    ax[0].set_ylabel("x(t)")
    ax[0].grid(True)

    ax[1].plot(f, X)
    ax[1].set_xlabel("f (Hz)")
    ax[1].set_ylabel("$|(\mathcal{F}x)(f)|$")
    ax[1].grid(True)

def cexp(k, N):
    """
    Creates complex exponential with frequency k and length N.

    Args:
        k: discrete frequency
        N: length
```

```
    Returns:
        a numpy array of length N.
    """

    n = np.arange(N)
    return np.exp(2j*np.pi*k*n/N) #/ np.sqrt(N)
```

1. Computation of Fourier Transforms

1.1 Fourier transform of a Gaussian pulse



1.2 Fourier transform of a Gaussian Pulse

[ ]:
```
def special_plot(x, t, X, X2, f, title=None):
    """
    Plot a pulse x and its dft X.

    Args:
        x: (N,) Signal of length N.
        t: (N,) times
        X: (N,) DFT of x.
        f: (N,) frequencies
    """
    fig, ax = plt.subplots(3, 1, figsize=(6,6))
    fig.suptitle(title)

    ax[0].plot(t, x)
    ax[0].set_xlabel("t (s)")
    ax[0].set_ylabel("x(t)")
    ax[0].grid(True)

    ax[1].plot(f, np.abs(X))
    ax[1].set_xlabel("f (Hz)")
    ax[1].set_ylabel("$|(\mathcal{F}x)(f)|$")
    ax[1].grid(True)

    ax[2].plot(f, np.abs(X2))
    ax[2].set_xlabel("f (Hz)")
    ax[2].set_ylabel("$|(\hat{\mathcal{F}}x)(f)|$")
    ax[2].grid(True)
```

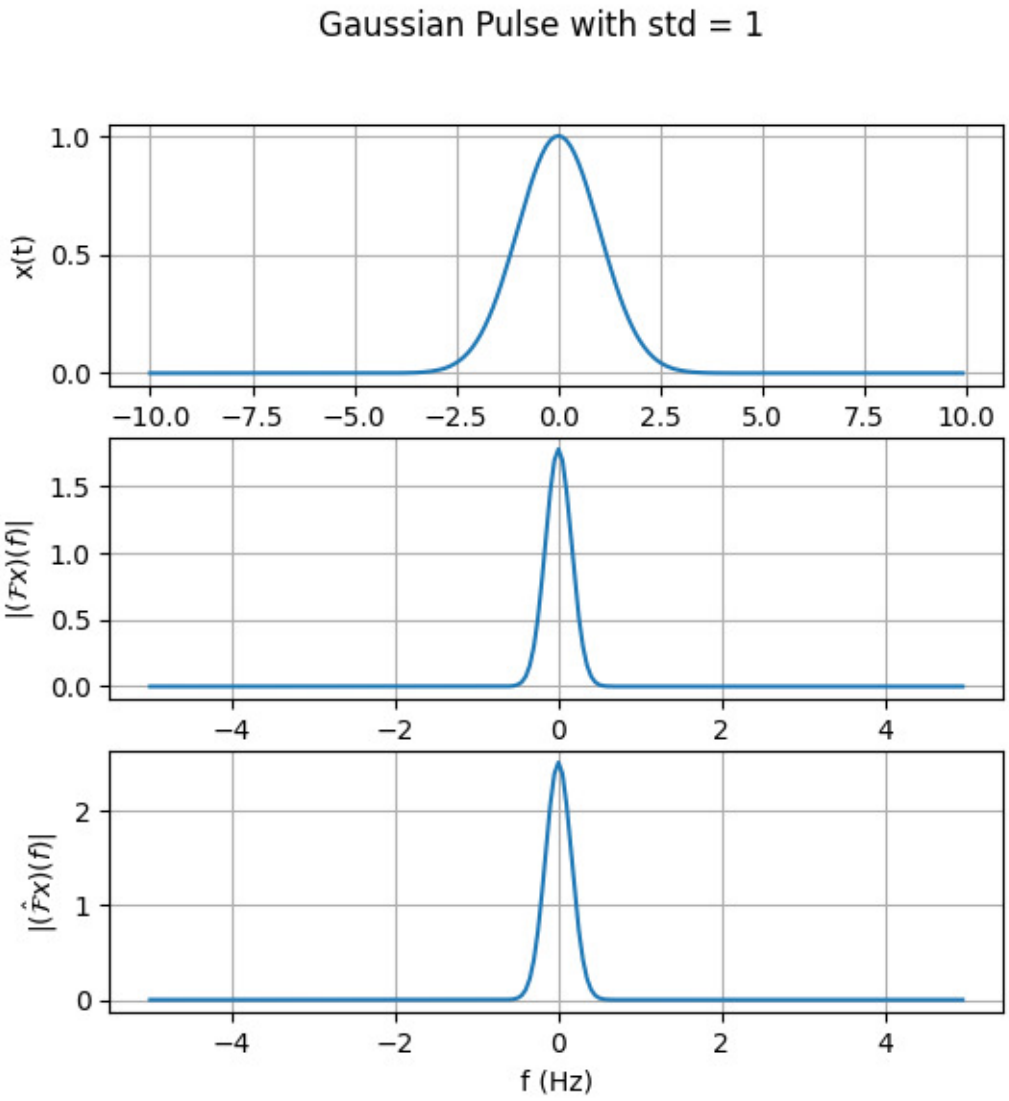
[ ]:
```
def gaussian_pulse(u, s, T, fs):
  """
```

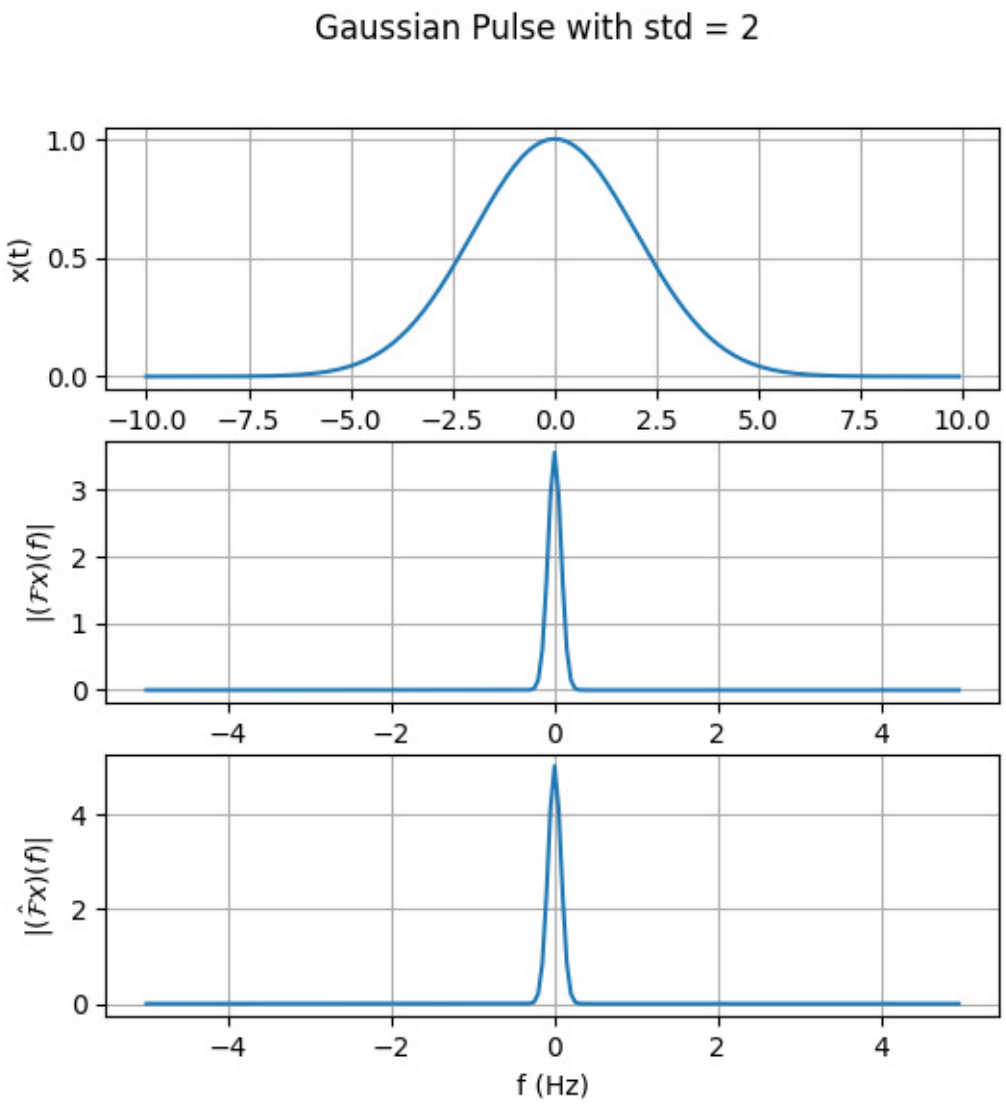
```
    Generate a gaussian pulse with mean mu and std sigma

    Args:
        u: average value mu
        s: standard deviation sigma
        T: duration of the pulse
        fs: sampling frequency
    Returns:
        x: signal
        t: real time instants t = -T, ..., -Ts, 0, Ts, ..., T
    """

    #TODO: implement
    t = np.arange(-T, T, 1/fs)
    x = np.exp(-((t-u)**2)/(2*s**2))

    return x, t
```

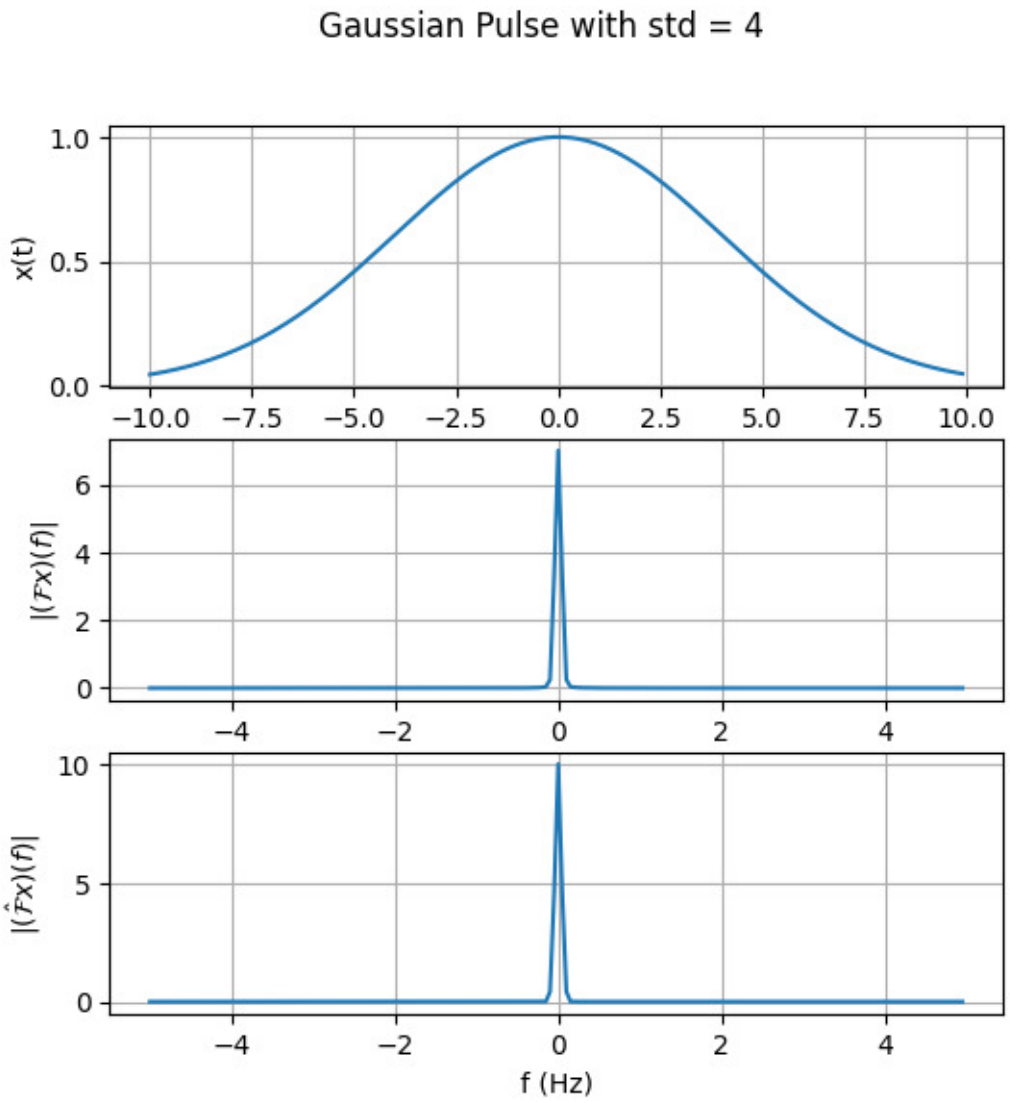
[ ]:
```
def gaussian_FT(u, s, f):
    """
    Evaluate the FT of a gaussian pulse according to your derivation in 1.1

    Args:
        u: Average value mu
        s: standard deviation sigma
        f: frequency range from -fs/2 to fs/2
    Returns:
        X: Spectrum evaluated according to the derivation in 1.1
    """

    #TODO: implement
    X = np.sqrt(2 * np.pi * s**2) * np.exp(-2 * (np.pi**2) * (f**2) * s**2) * np.
    ↪exp(-1j * 2 * np.pi * f * u)


    return X
```

[ ]:
```
# Define Parameters
u = 0
s_list = [1, 2, 4]
T = 10
fs = 10

# Construct the signal and evaluate its spectrum using DFT and FT
for s in s_list:
    x, t = gaussian_pulse(u, s, T, fs)     # TODO: Generate the gaussian pulse
    X, f = compute_fft(x, fs)              # Compute the spectrum using D/FFT
```

```
X_ft = gaussian_FT(u, s, f)            # Compute the spectrum analytically␣
↪(Part 1.1)

# Plotting
title = f'Gaussian Pulse with std = {s}'
special_plot(x, t, X, X_ft, f, title=title)
```



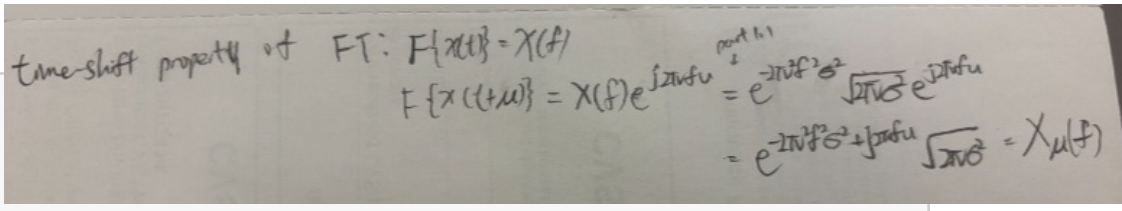

Gaussian Pulse with std = 2
x(t)
1.0
0.5
0.0
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0
|(ℱx)(f)|
3
2
1
0
−4
−2
0
2
4
|(ℱ̂x)(f)|
4
2
0
−4
−2
0
2
4
f (Hz)



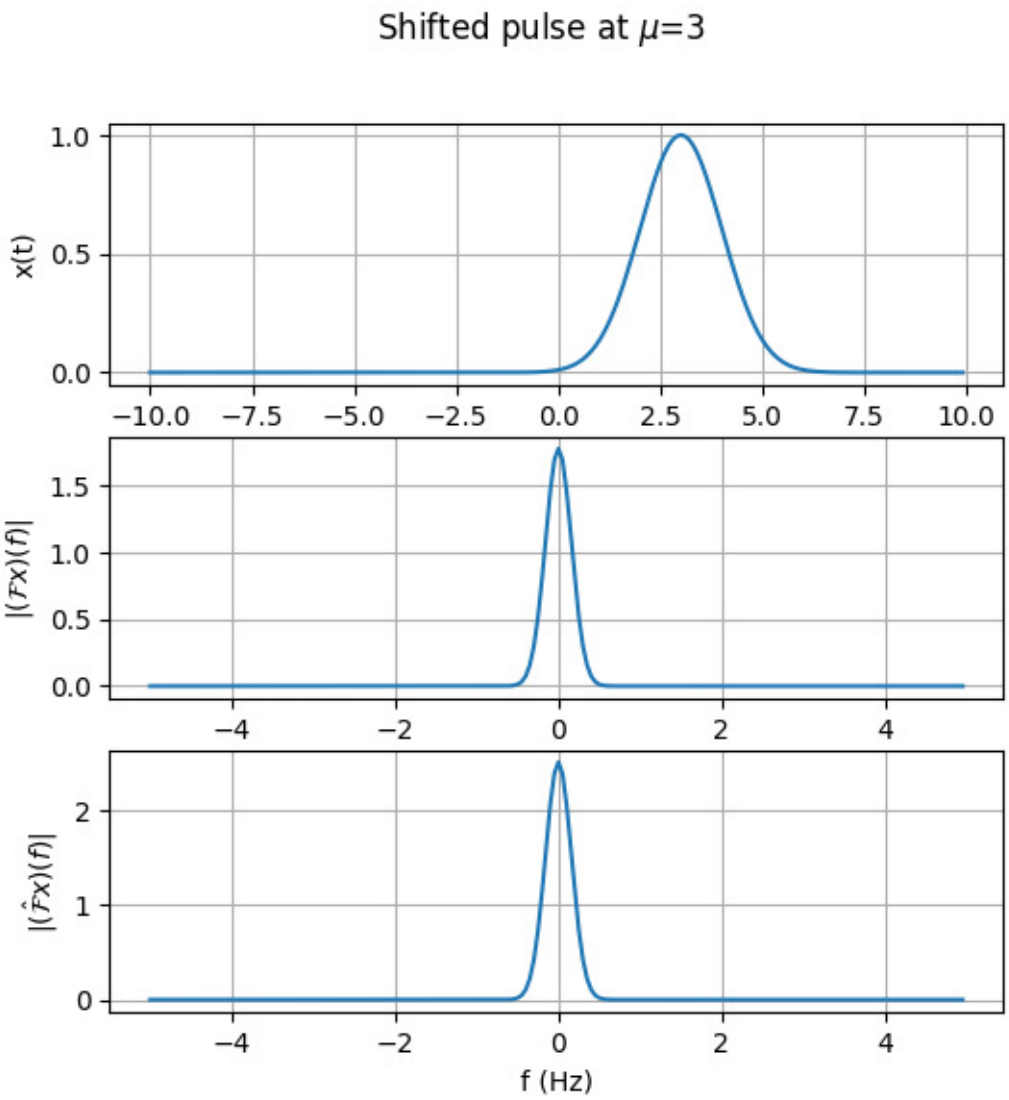
For each std value above, the DFT and scaled FT graphs look identical, allowing us to verify our derivation is correct.

1.3 Shifted Gaussian pulse

[ ]:
```
# Define Parameters
u = 3
s = 1
T = 10
fs = 10
```

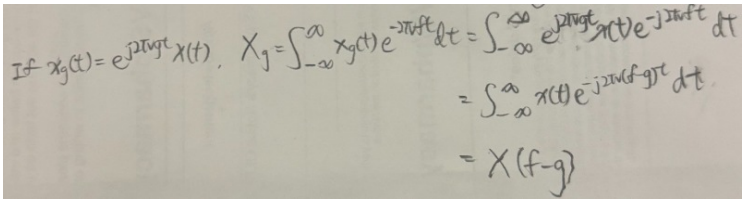


```python
# Construct the signal and evaluate its spectrum using DFT and FT
x, t = gaussian_pulse(u, s, T, fs)      # TODO: Generate the gaussian pulse
X, f = compute_fft(x, fs)                 # Compute the spectrum using D/FFT
X_ft = gaussian_FT(u, s, f)               # Compute the spectrum analytically

# Plotting
title = f'Shifted pulse at $\mu$={u}'
special_plot(x, t, X, X_ft, f, title=title)
```



2. Modulation and Demodulation

3.1 Theorem 2



3.2 2.0 Recording and Playing your voice

[ ]:
```
!pip -q install pydub
```

[ ]:
```
from IPython.display import HTML, Audio
from google.colab.output import eval_js
from base64 import b64decode
from io import BytesIO
from pydub import AudioSegment


AUDIO_HTML = """
<script>
var my_div = document.createElement("DIV");
var my_p = document.createElement("P");
var my_btn = document.createElement("BUTTON");
var t = document.createTextNode("Start 3s Recording");

my_btn.appendChild(t);
my_div.appendChild(my_btn);
document.body.appendChild(my_div);

var base64data = 0;
var reader;
var recorder, gumStream;
var recordButton = my_btn;

var handleSuccess = function(stream) {
  gumStream = stream;
  recorder = new MediaRecorder(stream);

  recorder.ondataavailable = function(e) {
    var url = URL.createObjectURL(e.data);
    var preview = document.createElement('audio');
    preview.controls = true;
    preview.src = url;
    document.body.appendChild(preview);

    reader = new FileReader();
    reader.readAsDataURL(e.data);
    reader.onloadend = function() {
```

```
      base64data = reader.result;
    }
  };

  recorder.start();
  recordButton.innerText = "Recording...";

  // Automatically stop after 3 seconds
  setTimeout(function() {
    recorder.stop();
    gumStream.getAudioTracks()[0].stop();
    recordButton.innerText = "Recording Finished!";
  }, 3400);
};

recordButton.onclick = function() {
  navigator.mediaDevices.getUserMedia({audio: true}).then(handleSuccess);
};

async function getData() {
  while (!base64data) {
    await new Promise(resolve => setTimeout(resolve, 500));
  }
  return base64data;
}

var data = getData();
</script>
"""

def record(fs):

  display(HTML(AUDIO_HTML))
  data = eval_js("data")
  binary = b64decode(data.split(',')[1])

  audio = AudioSegment.from_file(BytesIO(binary)).set_frame_rate(fs).
 ↪get_array_of_samples()
  audio = np.array(audio, dtype=np.float32)

  return audio
```

3.3 2.1 Bandlimited Signals

[ ]:
```python
def bandlimitter(X, f, B):
  """
  Generate bandlimited signals by nulling the frequency components outside the␣
 ↪range [-B, B]

  Args:
      X: Specrum
      f: frequency range from -fs/2 to fs/2
      B: Bandwidth of the bandlimited signal and B <= fs/2
  Returns:
      X_BL: Spectrum of the bandlimited signal
  """
  assert X.shape == f.shape
  assert B < np.max(np.abs(f))

  X_BL = X.copy()
  X_BL[np.abs(f) > B] = 0

  return X_BL
```

1. Record a signal

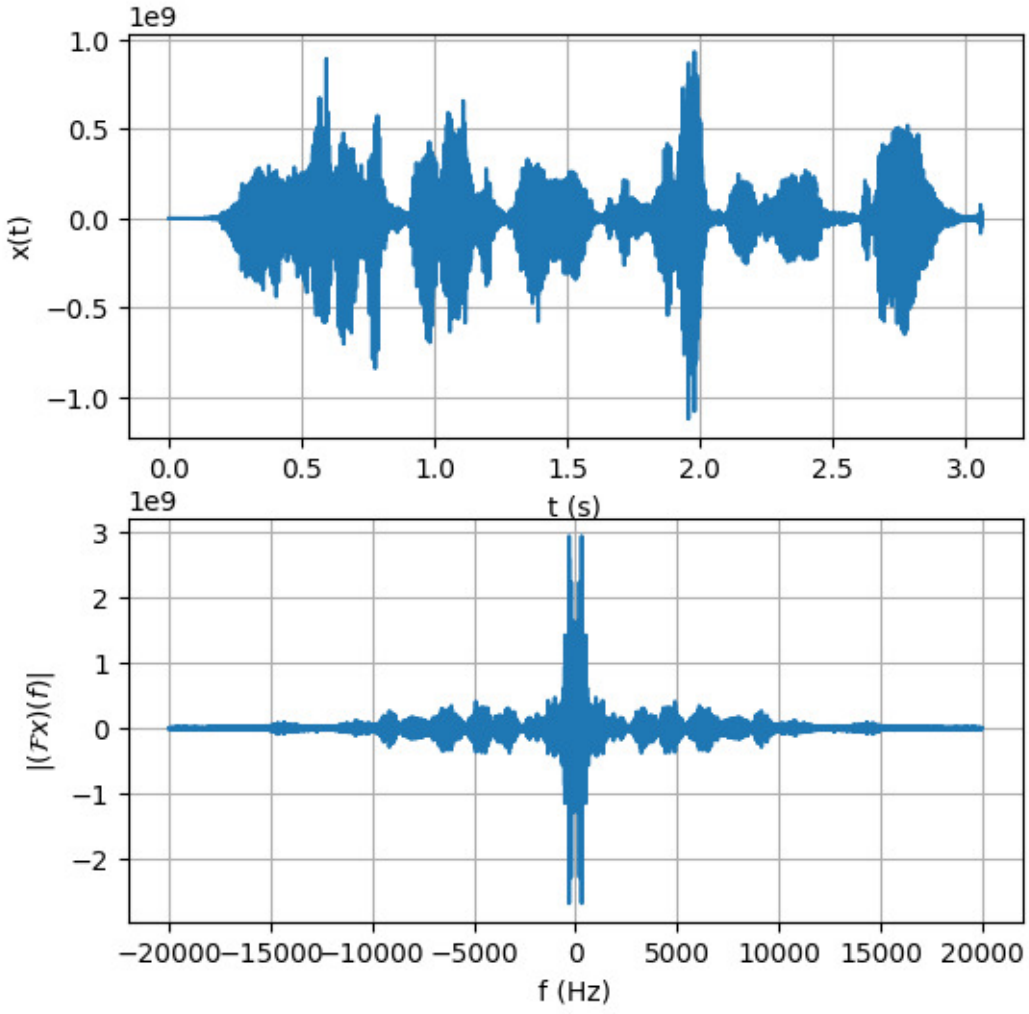
[ ]:
```python
# This part is only for recording your voice. You don't need to modify it.
fs = 40_000
B = 4_000

# Record a signal
x = record(fs)
t = np.arange(len(x))/fs

# Compute Spectrums
X,f = compute_fft(x, fs)            # Compute the DFT of x
```

```
<IPython.core.display.HTML object>
```

[ ]:
```python
# Plotting
print(t)
plot_signal_and_spectrums(x, t, X, f)
```

```
[0.000000e+00 2.500000e-05 5.000000e-05 … 3.059925e+00 3.059950e+00
 3.059975e+00]

/usr/local/lib/python3.11/dist-packages/matplotlib/cbook.py:1709:
ComplexWarning: Casting complex values to real discards the imaginary part
  return math.isfinite(val)
/usr/local/lib/python3.11/dist-packages/matplotlib/cbook.py:1345:
ComplexWarning: Casting complex values to real discards the imaginary part
```

```
return np.asarray(x, float)
```



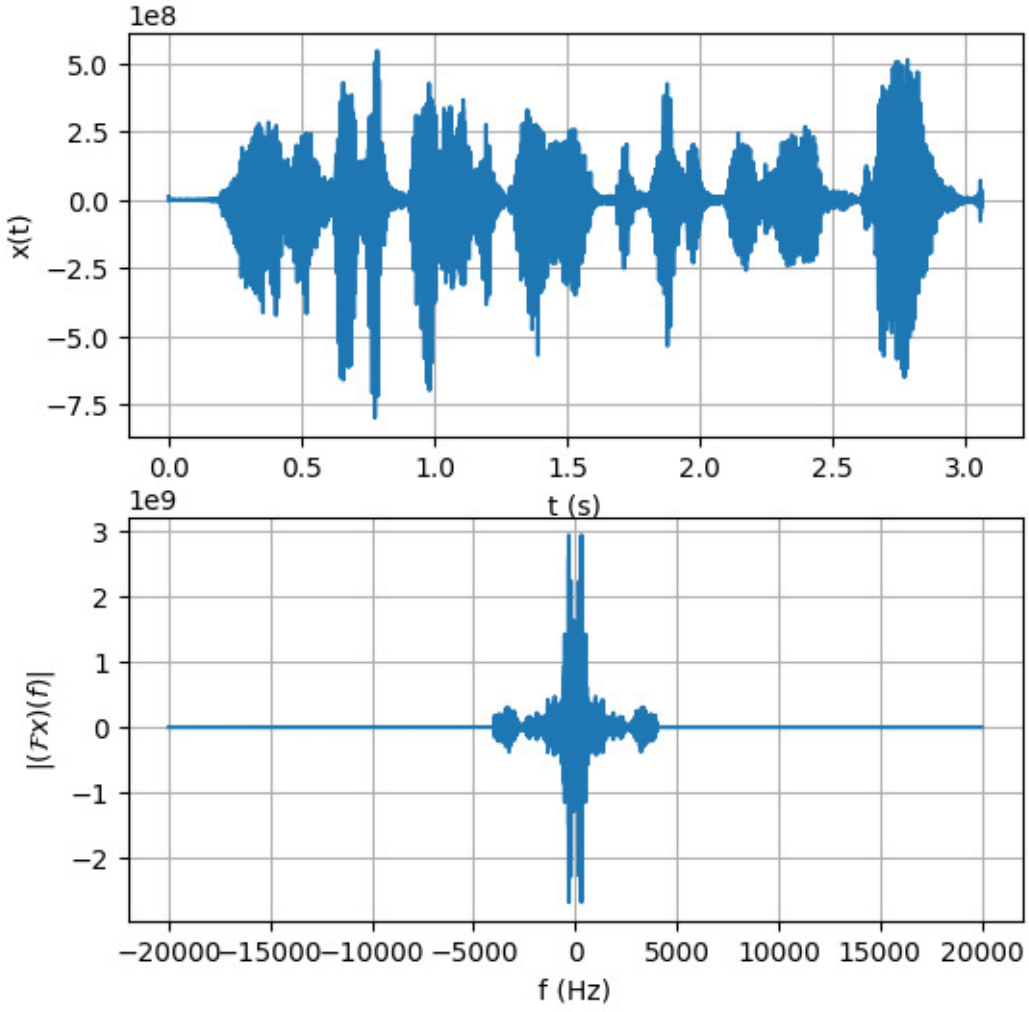
2. Generate a bandlimited version of your voice

[ ]:
```
# Reconstruct the bandlimitted signal in time domain
X_BL = bandlimitter(X, f, B)
x_BL, t_BL = compute_ifft(X_BL, f)

# Plotting
plot_signal_and_spectrums(x_BL, t, X_BL, f)
print(t)

# Play the signal
Audio(data=np.array(x_BL, dtype=np.float32), rate=fs, autoplay=True)
```

```
[0.000000e+00 2.500000e-05 5.000000e-05 … 3.059925e+00 3.059950e+00
```

```
 3.059975e+00]
<ipython-input-3-c56dbcbfd063>:37: RuntimeWarning: divide by zero encountered in
divide
  t = np.arange(N)/fs
```

[ ]:
```
<IPython.lib.display.Audio object>
```

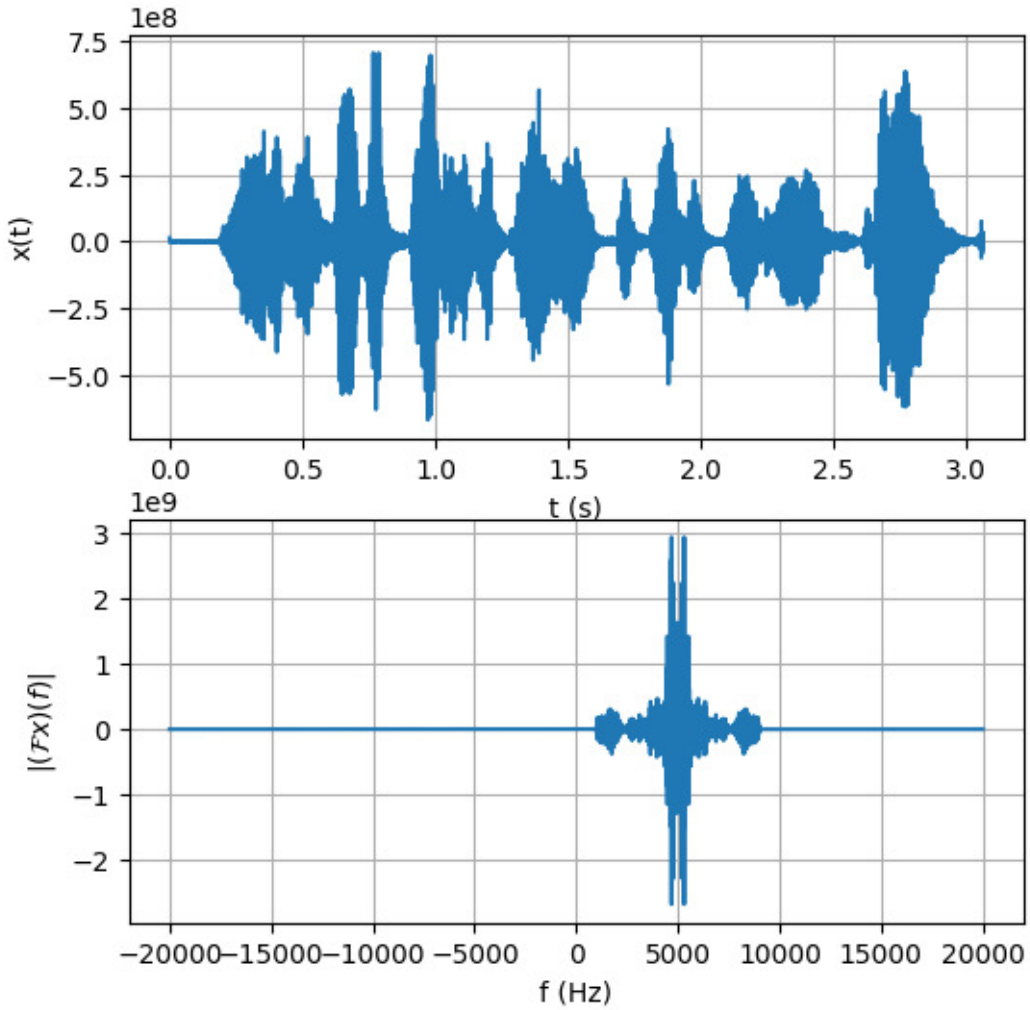


2.2 Voice Modulation

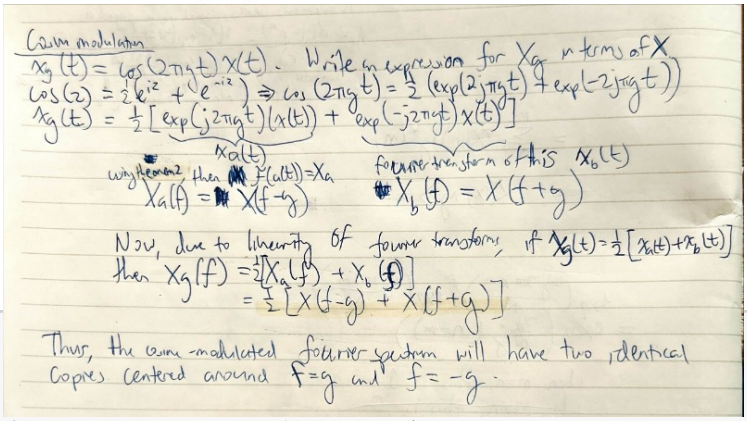
[ ]:
```
def Modulation(x, fc, fs, carrier='exp'):
  """
  Generate a modulated signal. The carrier could be an exponential or a cosine␣
  ↪signal

  Args:
```

```
    x: signal of length N
    fc: carrier frequency
    carrier: is either 'exp' or 'cos'.
  Returns:
    x_mod : modulated signal in time
    t: time instances
    X_mod: modulated sigal in freq
    f: frequency range
  """

  N = len(x)
  t = np.arange(N)/fs

  if carrier == "exp":
    carrier_signal = np.exp(2j * np.pi * fc * t)
  elif carrier == 'cos':
    carrier_signal = np.cos(2 * np.pi * fc * t)
  else:
    raise ValueError("Carrier type must be either 'exp or 'cos")

  x_mod = x * carrier_signal # element-wise multiplication
  X_mod, f = compute_fft(x_mod, fs)

  # Note the function returns both the time and frequency representations of␣
 ↪the modulated signal
  return x_mod, t, X_mod, f
```

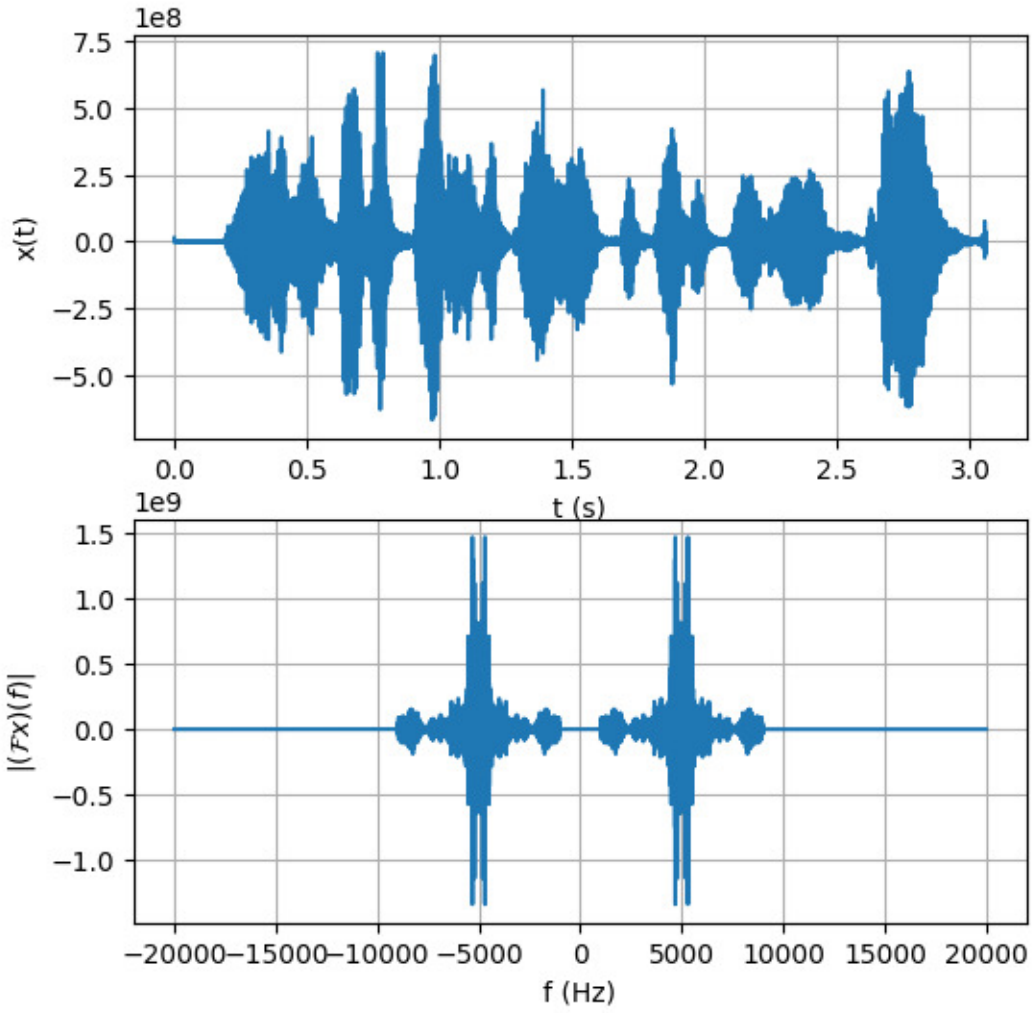
[ ]:
```
# Modulation
# we want to center it on 5kHz which means that we're shifting it by that much␣
 ↪in the
# frequency domain. so in the time domain we need to multiply by a complex␣
 ↪exponential
#
fc = 5_000
x_mod, t, X_mod, f = Modulation(x_BL, fc, fs)

# Plotting
plot_signal_and_spectrums(x_mod, t, X_mod, f)
```



2.3 Cosine Modulation




```
[ ]: # Modulation
     fc = 5_000

     x_mod, t, X_mod, f  = Modulation(x_BL, fc, fs, carrier="cos")

     # Plot the spectrums
     plot_signal_and_spectrums(x_mod, t, X_mod, f)
```



These results indeed match our theoretical predictions. In the plot above, the Fourier spectrum is split in half, with the two halves centered around ± 5000. The magnitudes in frequency space have also all been scaled down by a factor of two, with the largest peaks at around 9e8, as compared to the plot in 2.2, with the largest peaks being around 1.8e9

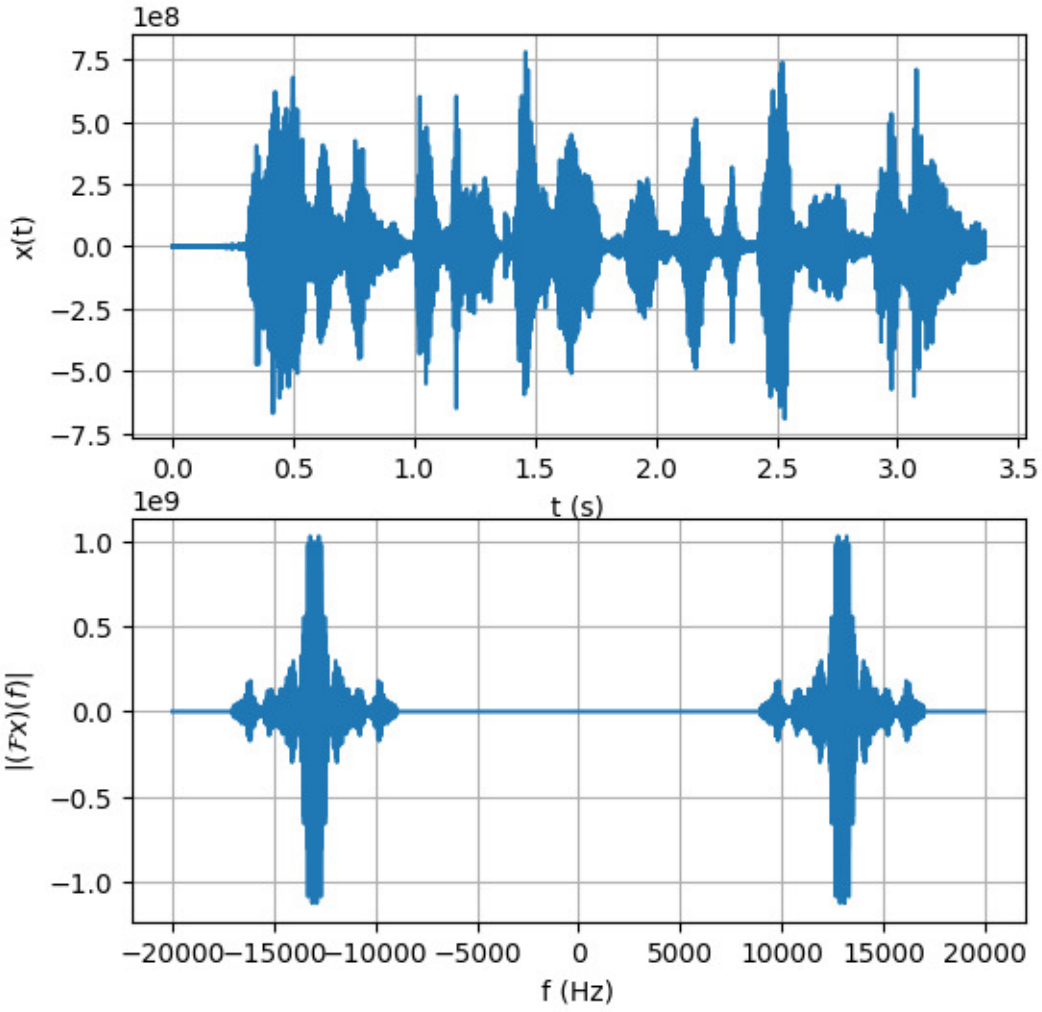
2.4 Repeat

[ ]:
```
# Record another signal (Your friend's voice) and Compute its FFT
x2 = record(fs)                    # Recording
X2,f2 = compute_fft(x2,fs)      # TODO: Compute the FFT

# Bandlimiting version
X_BL2 = bandlimitter(X2,f2,B)          # TODO: Generate a bandlimited version
x_BL2, t2 = compute_ifft(X_BL2,f2)       # TODO: Compute its iFFT
```

```
# Cosine modulation
fc2 = 13_000
x_mod2, t2, X_mod2, f2 =  Modulation(x_BL2, fc2, fs, carrier="cos")    #TODO:␣
 ↪implement cosine modulation

plot_signal_and_spectrums(x_mod2, t2, X_mod2, f2)
```

```
<IPython.core.display.HTML object>
```

```
<ipython-input-3-c56dbcbfd063>:37: RuntimeWarning: divide by zero encountered in
divide
  t = np.arange(N)/fs
/usr/local/lib/python3.11/dist-packages/matplotlib/cbook.py:1709:
ComplexWarning: Casting complex values to real discards the imaginary part
  return math.isfinite(val)
/usr/local/lib/python3.11/dist-packages/matplotlib/cbook.py:1345:
ComplexWarning: Casting complex values to real discards the imaginary part
  return np.asarray(x, float)
```



2.5 Recover individual voices

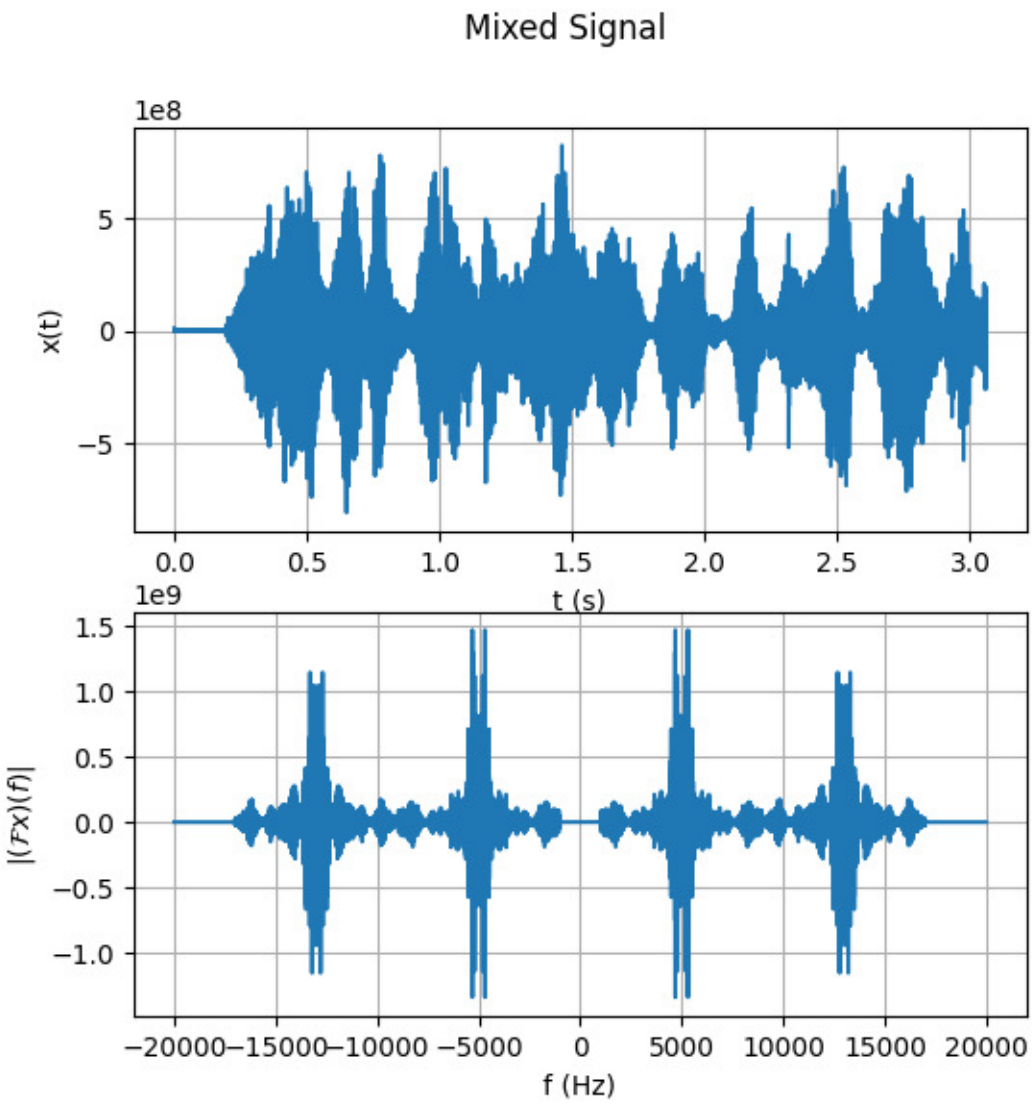
To recover it we will just use take the signals around fc for each signal. for example, if we modulated some signal with fc = 10k cosine then this signal has been shifted to the left and the right by 10k and it was bandlimited so we know the barrier of the signal from each side so we can cut it out of a mixed signal.

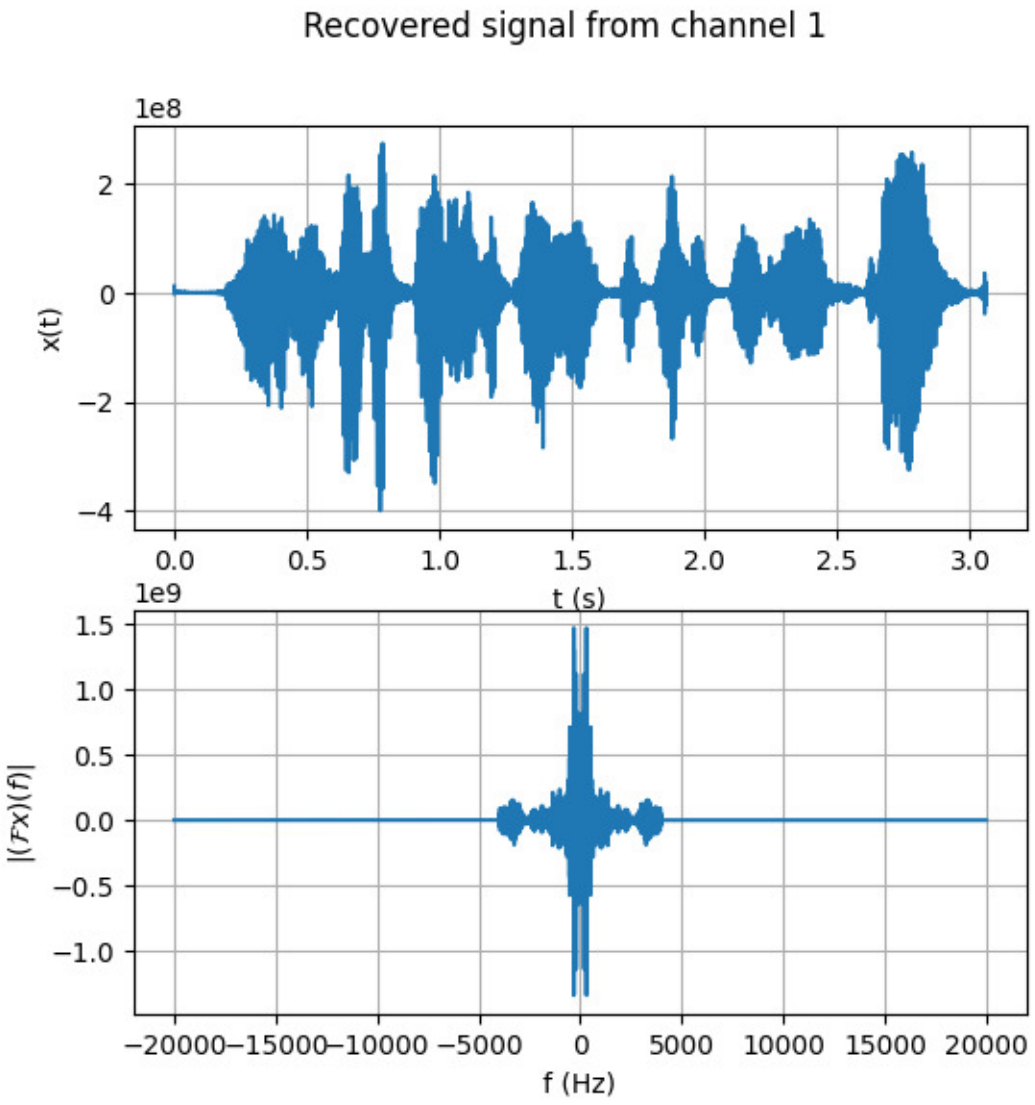
[ ]:
```python
def filtering(X, f, fc, B):
  """
  Filter out a bandlimited signal, centered around fc, from a mix of signals.

  Args:
      X: spectrum
      f: range of frequencies
      fc: carrier frequency
      B: bandwidth
  Returns:
      X_flt : filtered spectrum
  """

  # TODO: implemet
  mask = ((fc - B <= f) & (f <= fc + B)) | ((-fc - B <= f) & (f <= -fc + B))

  # Apply the mask to the spectrum
  X_flt = np.zeros_like(X)
  X_flt[mask] = X[mask]

  return X_flt
```

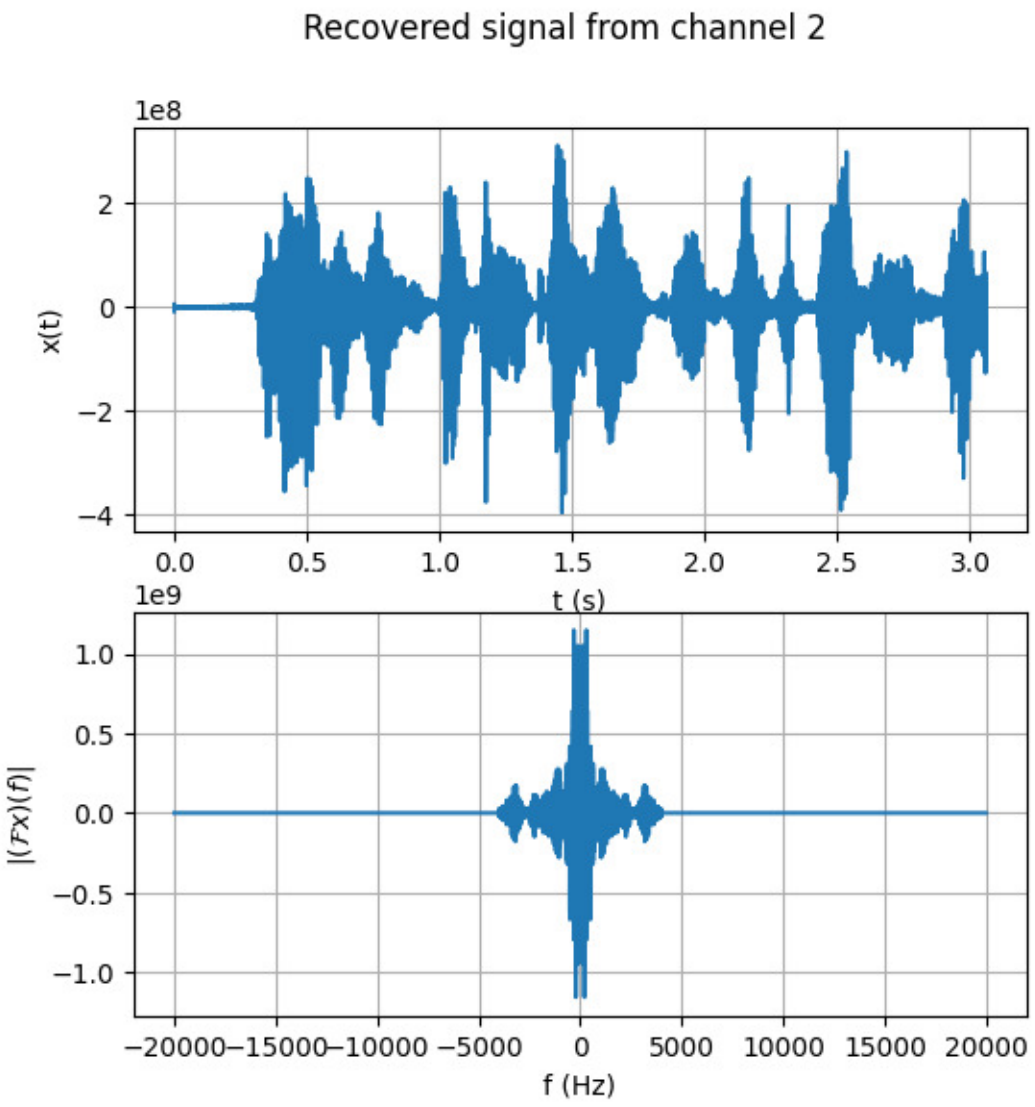
[ ]:
```python
def deModulation(X, fc, fs, B, carrier='exp'):
  """
  Return demodulated signal centered around 0 with bandwidth B

  Args:
      X: spectrum
      fc: carrier frequency
      fs: sampling rate
      B: bandwidth
  Returns:
      x_recon : reconstructed signal in time
      t: time instances
      X_recon: reconstructed signal in freq
      f: frequencies
  """
```

```
  # TODO: implement. You might use the compute_ifft and Modulation functions.
  N = len(X)
  f = np.linspace(-N/2 * (fs/N), N/2 * (fs/N), N)
  shift = int((fc*N)/fs)
  X_recon = np.zeros_like(X)  # Create an array of zeros
  X_recon[:-shift] = X[shift:]  # Shift elements to the left
  X_recon = filtering(X_recon, f, 0, B)
  x_recon, t = compute_ifft(X_recon, f)
  # Note the function returns both the time and frequency representations of␣
 ↪the reconstructed signal
  return x_recon, t, X_recon, f
```

[ ]:
```
# Combine the two signals
if len(x_mod2) < len(x_mod):
  mix = x_mod2 + x_mod[:len(x_mod2)]
  t = t2
  f = f2
else:
  mix = x_mod + x_mod2[:len(x_mod)]

MIX, f = compute_fft(mix,fs)          # TODO: Compute the FFT of the mixed␣
 ↪signal
plot_signal_and_spectrums(mix, t, MIX, f, title='Mixed Signal')

fc_list = [fc, fc2]
recovered=[]
for i, f_tune in enumerate(fc_list):
  X_ch = filtering(MIX, f, f_tune, B)          # TODO: Filter the signal on␣
 ↪channel i  (BPF)
  x_recon, _, X_recon, f = deModulation(X_ch, f_tune, fs, B, carrier='cos')  #␣
 ↪TODO: Demodulate the signal and recover the original one
  plot_signal_and_spectrums(x_recon, t, X_recon, f, title=f'Recovered signal␣
 ↪from channel {i+1}')

  # Save signals
  recovered.append(x_recon)
```


Mixed Signal
1e8
5
0
−5
x(t)
0.0
0.5
1.0
1.5
2.0
2.5
3.0
t (s)
1e9
1.5
1.0
0.5
0.0
−0.5
−1.0
|(Fx)(f)|
−20000
−15000
−10000
−5000
0
5000
10000
15000
20000
f (Hz)


Recovered signal from channel 1
1e8
2
0
−2
−4
x(t)
0.0
0.5
1.0
1.5
2.0
2.5
3.0
t (s)
1e9
1.5
1.0
0.5
0.0
−0.5
−1.0
|(ℱx)(f)|
−20000
−15000
−10000
−5000
0
5000
10000
15000
20000
f (Hz)



Play the recovered signals

[ ]:
```
# Signal #1
Audio(data=np.array(recovered[0], dtype=np.float32), rate=fs, autoplay=True)
```

[ ]:
```
<IPython.lib.display.Audio object>
```

[ ]:
```
# Signal #2
Audio(data=np.array(recovered[1], dtype=np.float32), rate=fs, autoplay=True)
```

[ ]:
```
<IPython.lib.display.Audio object>
```