ESE_224_Spring_2025_Lab_3_Group3

February 19, 2025

https://drive.google.com/file/d/1QVvXyDfqLTtSs9lrZDg8L8Hxg9-yOfr5/view?usp=sharing

Group members: Laura Gao, Mohamed Elsheshtawy, Celia Tung

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

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

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

1 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 plot_signal_and_dft(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=(6,4))
    fig.suptitle(title)

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

def inner_product(x, y):
    """
    Computes the inner product between two numpy arrays x and y.

    Args:
        x: numpy array.
        y: numpy array.
    """
    return x @ np.conj(y.T)

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)

def make_square_pulse(T0: float, T: float, fs: int):
    N = int(T * fs)
    M = int(T0 * fs)
```

$$\tilde{x}(\tilde{n}) = \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} X(k) e^{j2\pi k\tilde{n}/N}$$

$$= \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} \left(\frac{1}{\sqrt{N}} \sum_{n=0}^{N-1} x(n) e^{-j2\pi kn/N} \right) e^{j2\pi k\tilde{n}/N}$$

$$= \sum_{n=0}^{N-1} x(n) \left(\sum_{k=0}^{N-1} \frac{1}{\sqrt{N}} e^{-j2\pi kn/N} \frac{1}{\sqrt{N}} e^{j2\pi k\tilde{n}/N} \right)$$

orthonormal

$$= \sum_{n=0}^{N-1} x(n) \delta(n-\hat{n}) = x(n)$$

```
    x = np.zeros(N)
    x[:M] = 1 / np.sqrt(M)
    return x

def make_triangle_pulse(T0: float, T: float, fs: int):
    N = int(T * fs)
    M = int(T0 * fs)
    n = np.arange(N)
    x = np.zeros(N)
    x[n < M] = M - 1 - n[n < M]
    x[n < M/2] = n[n < M/2]
    return x / np.linalg.norm(x)
```

2 Proof of Theorem 1

3 1 Signal reconstruction and compression

3.1 1.1 Computation of the iDFT

[ ]:
```
def compute_idft(X, N, fs: int):
    """
    Compute the iDFT of X using first N/2 DFT coefficients.
    Assume that X is a DFT of a real signal.

    Args:
        X: DFT coefficients corresponding to k=[0, 1, 2, ..., N/2]
        N: the original length of the signal
        fs: Sampling frequency
    Returns:
        x: iDFT of X (should be **real**)
        t: real time instants n * Ts associated with x, n = 0, 1, ..., N-1
    """
    N = int(N)
    x = np.zeros(N).astype(np.complex128)
    for k in range(1, len(X)):
      x += X[k] * cexp(k, N) + np.conj(X[k]) * cexp(-k, N)
    x += X[0] * cexp(0, N)
    T = N*fs
    t = np.linspace(0, T, N)
    return x, t
```

3.2 1.2 Signal reconstruction

[ ]:
```python
def reconstruct_signal(X, N: int, fs: int):
    """
    Reconstruct a real signal using the first K+1 DFT coefficients.
    Specifically, compute the iDFT of X using first K+1 DFT coefficients.

    Args:
    X: DFT coeffs [0, 1, 2, ..., K-1], 0 <= K <= N/2
    N: signal duration
    fs: Sampling frequency
    Returns:
    x: iDFT of X
    t: real time instants n * Ts associated with x, n = 0, 1, ..., N-1
    """
    # TODO: implement
    # You can use compute_idft to do so.
    x,t = compute_idft(X,N,fs)
    return x, t
```

3.3 1.3 Reconstruction of a square pulse

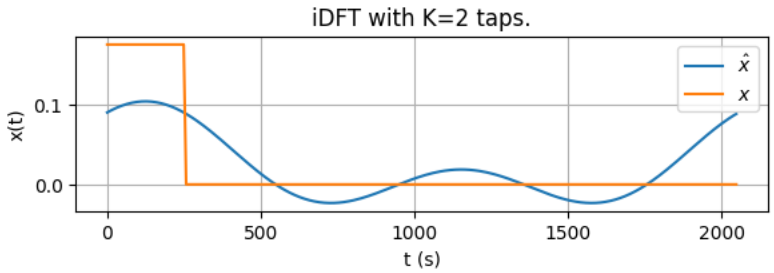
[ ]:
```python
def plot_reconstruction(ax, t, x, xhat, title):
    """
    Plots the signal and the reconstructed version on an axis.
    This function does **not** call plt.show() or plt.figure().

    Args:
        ax: The matplotlib axis to plot on.
        t: An array of times at which the signals are sampled.
        x: the original signal.
        xhat: the reconsructed signal. The same length as x and t.
        title: A string that gives the title of the plot.
    """
    ax.plot(t, xhat, label = '$\hat{x}$')
    ax.plot(t, x, label = '$x$')
    ax.set_xlabel("t (s)")
    ax.set_ylabel("x(t)")
    ax.legend()
    ax.grid(True)
    ax.set_title(title)
```

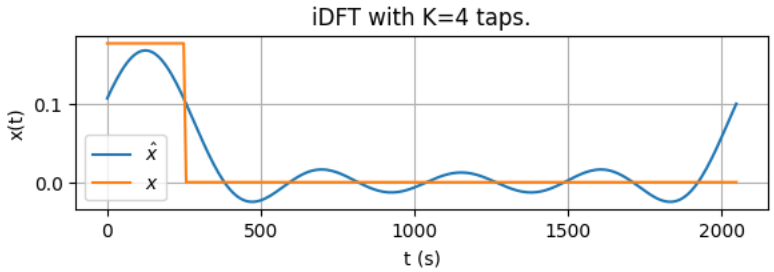
[ ]:
```python
T = 32
T0 = 4
fs = 8
Ks = [2, 4, 8, 16, 32,64,120]
```

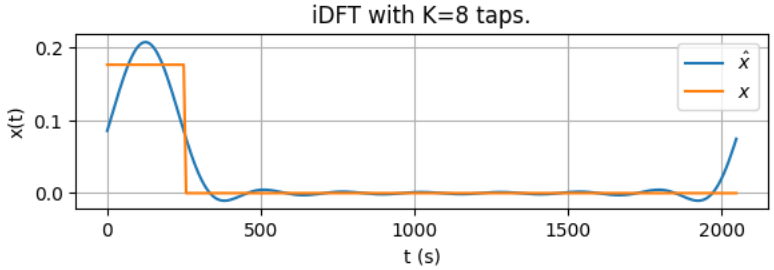
```
x = make_square_pulse(T0, T, fs)
N = int(T*fs)
X, f = compute_fft(x, fs, center_frequencies=False)

fig, ax = plt.subplots(len(Ks), 1, figsize=(6, 2*len(Ks)))
fig.tight_layout(h_pad=3, w_pad=0)

for i, K in enumerate(Ks):
    X_K = X[:K+1]  # X_K has the first K+1 coefficients.
    xhat, t = reconstruct_signal(X_K, N, fs)
    title = f"iDFT with {K=} taps."
    plot_reconstruction(ax[i], t, x, xhat, title)

    energy_diff = abs(inner_product(x, x) - inner_product(xhat, xhat))
    diff = abs(x - xhat)
    energy_diff_spectral = inner_product(diff, diff)
    print(f"{K=:<2} | {energy_diff:.4f} | {energy_diff_spectral:.4f}")
```

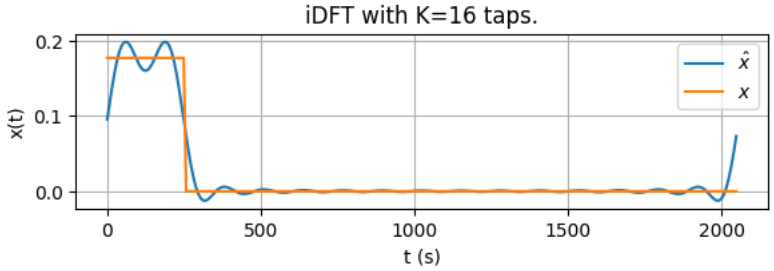
```
/usr/local/lib/python3.11/dist-packages/matplotlib/cbook.py:1709:
ComplexWarning: Casting complex values to real discards the imaginary part
  return math.isfinite(val)
/usr/local/lib/python3.11/dist-packages/matplotlib/cbook.py:1345:
ComplexWarning: Casting complex values to real discards the imaginary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

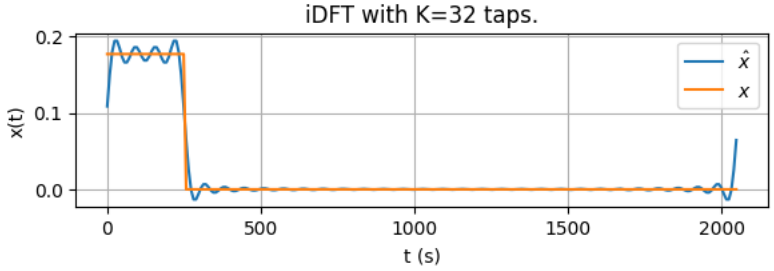
```
K=2  | 0.4349 | 0.4349
K=4  | 0.1797 | 0.1797
K=8  | 0.0968 | 0.0968
K=16 | 0.0494 | 0.0494
K=32 | 0.0239 | 0.0239
K=64 | 0.0099 | 0.0099
K=120 | 0.0010 | 0.0010
```

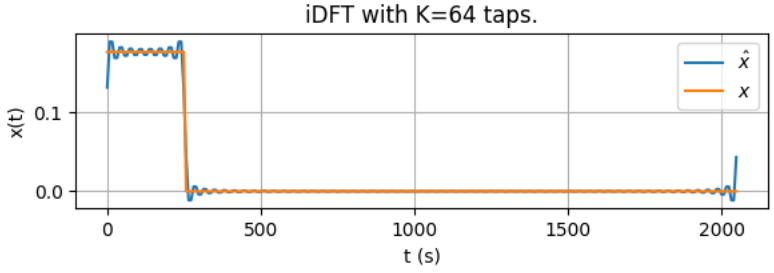

iDFT with K=2 taps.
x(t)
0.1
0.0
0
500
1000
1500
2000
t (s)
$\hat{x}$
x

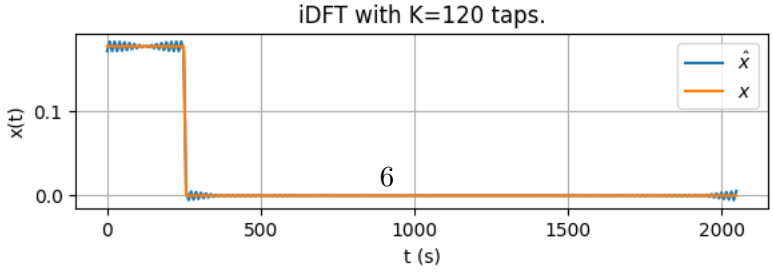

iDFT with K=4 taps.
x(t)
0.1
0.0
0
500
1000
1500
2000
t (s)
$\hat{x}$
x


iDFT with K=8 taps.
x(t)
0.2
0.1
0.0
0
500
1000
1500
2000
t (s)
$\hat{x}$
x


iDFT with K=16 taps.
x(t)
0.2
0.1
0.0
0
500
1000
1500
2000
t (s)
$\hat{x}$
x


iDFT with K=32 taps.
x(t)
0.2
0.1
0.0
0
500
1000
1500
2000
t (s)
$\hat{x}$
x

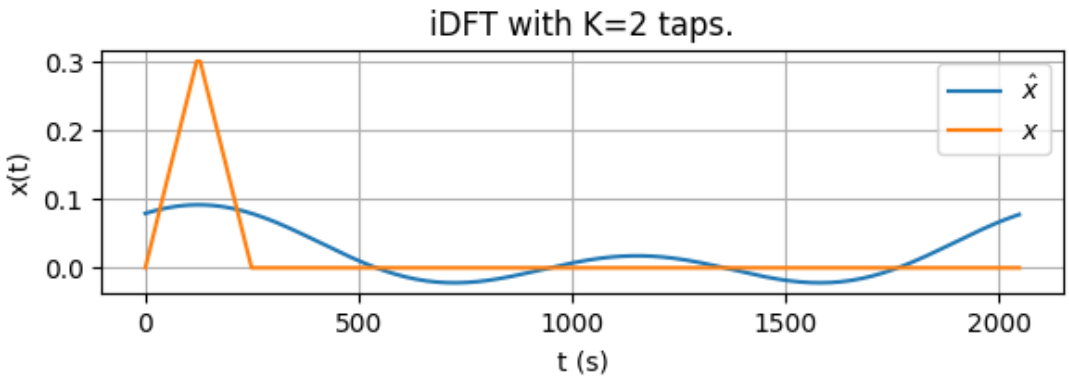

iDFT with K=64 taps.
x(t)
0.1
0.0
0
500
1000
1500
2000
t (s)
$\hat{x}$
x

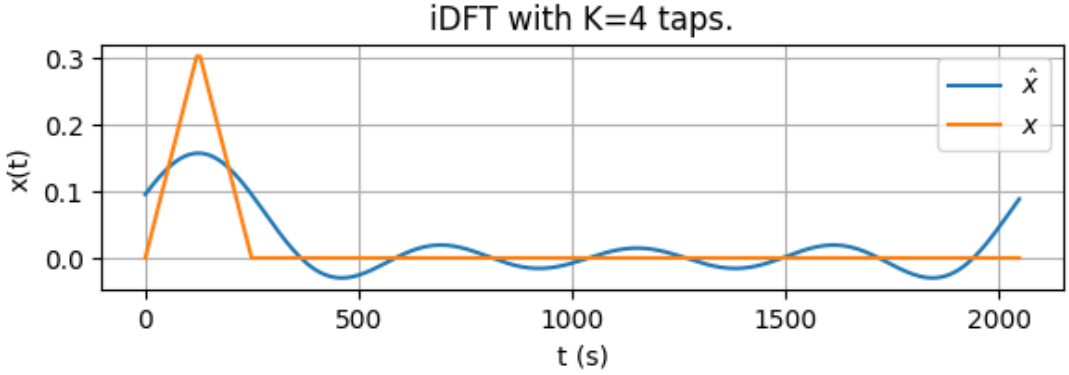

iDFT with K=120 taps.
x(t)
0.1
0.0
0
500
1000
1500
2000
t (s)
$\hat{x}$
x

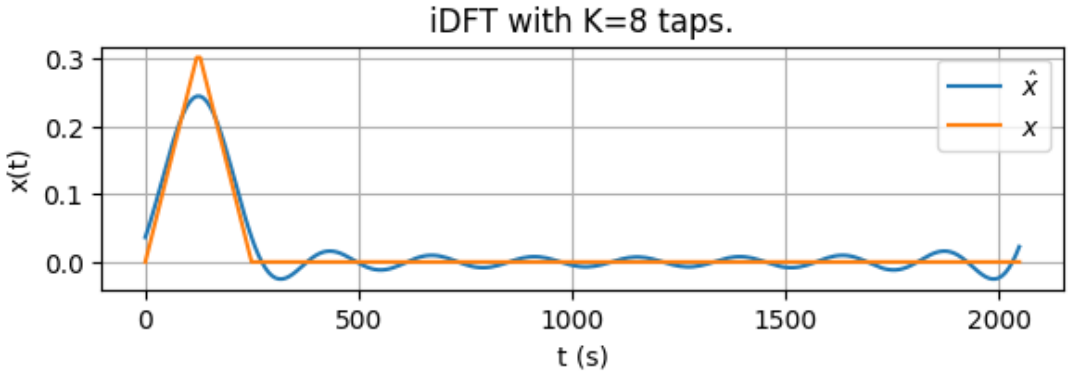
3.4 1.4 Reconstruction of a triangular pulse

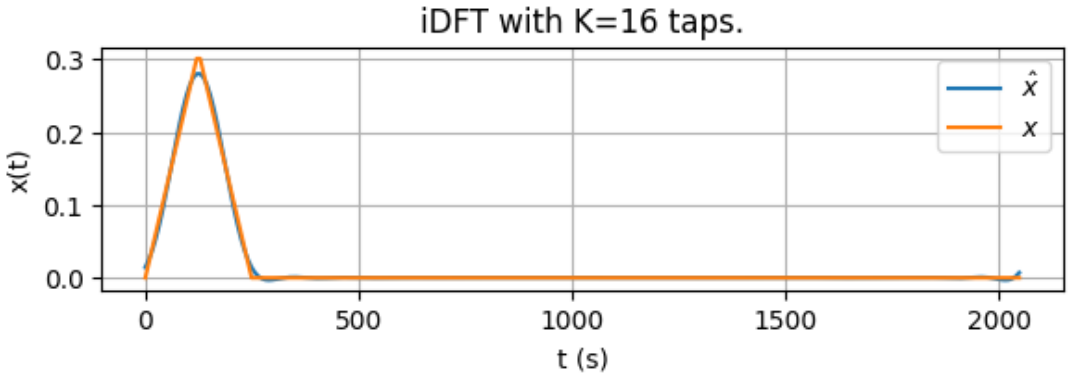
[ ]:
```
T = 32
T0 = 4
fs = 8
Ks = [2, 4, 8, 16, 32]

x = make_triangle_pulse(T0, T, fs)
X, f = compute_fft(x, fs, center_frequencies=False)

fig, ax = plt.subplots(len(Ks), 1, figsize=(6, 2*len(Ks)))
fig.tight_layout(h_pad=3, w_pad=0)

N = int(T*fs)

for i, K in enumerate(Ks):
    X_K = X[:K+1]  # X_K has the first K+1 coefficients.
    xhat, t = reconstruct_signal(X_K, N, fs)
    title = f"iDFT with {K=} taps."
    plot_reconstruction(ax[i], t, x, xhat, title)

    energy_diff = abs(inner_product(x, x) - inner_product(xhat, xhat))
    diff = abs(x - xhat)
    energy_diff_spectral = inner_product(diff, diff)

    print(f"{K=:<2} | {energy_diff:.4f} | {energy_diff_spectral:.4f}")
plt.show()
```

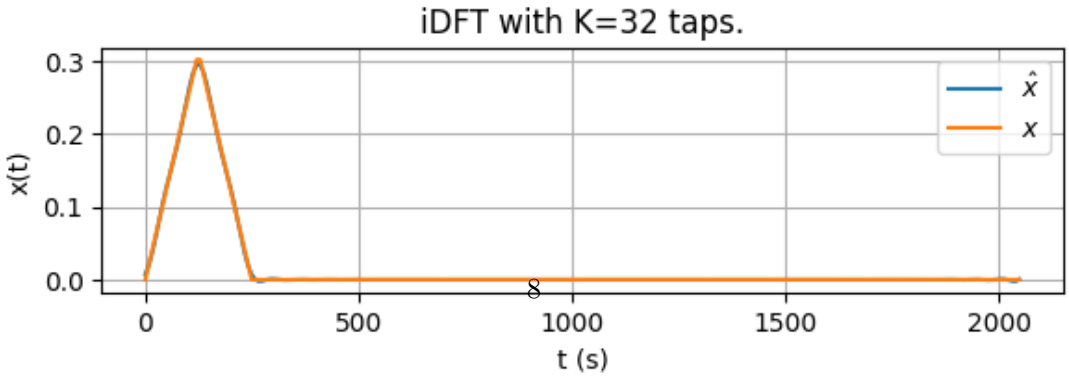
```
K=2  | 0.5674 | 0.5674
K=4  | 0.2991 | 0.2991
K=8  | 0.0429 | 0.0429
K=16 | 0.0029 | 0.0029
K=32 | 0.0004 | 0.0004
```


iDFT with K=2 taps.
x(t)
t (s)
$\hat{x}$
x


iDFT with K=4 taps.
x(t)
t (s)
$\hat{x}$
x


iDFT with K=8 taps.
x(t)
t (s)
$\hat{x}$
x


iDFT with K=16 taps.
x(t)
t (s)
$\hat{x}$
x


iDFT with K=32 taps.
x(t)
t (s)
$\hat{x}$
x

4 1.5 The energy of the difference signal

Our numerical results above show that energy difference calculated by the two methods coincide, for both square and triangle pulses.

4.1 1.6 Signal compression.

[ ]:
```python
def compress_signal(x, fs, K):
    """
    Takes a real signal x of length N with sampling frequency fs.
    Finds the K largest (by absolute value) DFT coefficients for frequencies (f␣
 ↪>= 0).

    Args:
        x: real signal.
        fs: sampling frequency.
        K: the number of DFT coefficients to keep.
    Returns:
        X_K: The K largest coefficients with frequencies f >= 0.
        f_K: The corresponding frequencies.
    """
    X, f = compute_fft(x, fs,center_frequencies=True)

    # Ensure only non-negative frequencies are selected
    mask = f >= 0
    X = X[mask]
    f = f[mask]

    # Ensure K does not exceed available elements
    K = min(K, len(f))

    # Sort by absolute magnitude of DFT coefficients
    indices = np.argsort(-np.abs(X))  # Sort descending by absolute value
    X_K = X[indices[:K]]  # Select the top K values
    f_K = f[indices[:K]]  # Corresponding frequencies
    freq_sort_indices = np.argsort(f_K)
    X_K = X_K[freq_sort_indices]
    f_K = f_K[freq_sort_indices]
    return X_K, f_K
```

4.2 1.7 The why of signal compression

According to Parseval's theorem, the energy of signals in the time domain comes from energy of DFT coefficients from the frequency domain, meaning for signal compression, we can approximate

the signal with minimal losses (the errors would just be proporitional to the small DFT coefficients) using the largest DFT coefficients as they contribute to the majority of the signal.

4.3 1.8 Signal reconstruction

[ ]:
```python
def decompress_signal(X, f, N, fs):
    """

    Args:
        X: An array of length K DFT coefficients.
        f: The frequencies corresponding to X. All the frequencies are assumed to be within [0,fs/2].
        N: the length of the original signal.
        fs: The sampling rate.
    Returns:
        x: a reconstruction of the original real signal.
        t: the sampling times in seconds corresponding to x.
    """
    assert (0 <= f).all() and (f <= fs/2).all()

    # we need to sort X based on f and add zeros in between where there are missing ones.
    f/=fs
    f*=N
    X_new = np.zeros(int(N/2)+1,dtype=np.complex128)
    j= 0
    for i in f:
       X_new[int(i)] = X[j]
       j+=1
    # we need to pass this through the idft.
    x,t = reconstruct_signal(X_new, N, fs)
    # TODO: implement, don't forget to set dtype=np.complex128
    return x,t
```

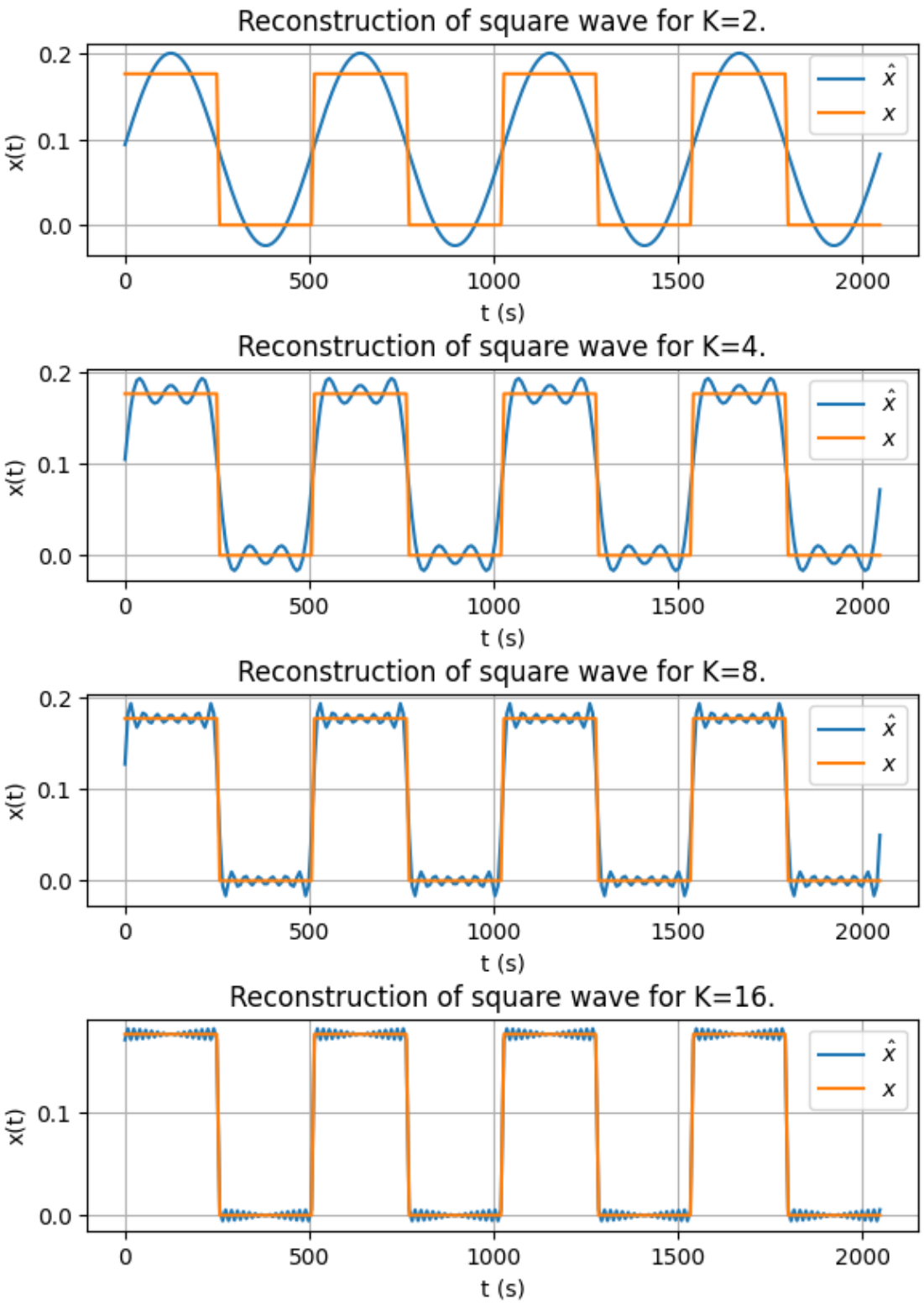
4.4 1.9 Compression and reconstruction of a square wave.

[ ]:
```python
def make_square_wave(f0: float, T:float, fs: int):
    """
    Args:
        f0: The frequency of the wave.
        T: The duration of the wave.
        fs: The sampling frequency.
    Returns:
        x: A square wave as defined in equation (9).
    """
    N = int(T * fs) # number of samples in the total wave
    T0 = 1/f0 # T0 is the length of one period
```

```
    M = int(T0 * fs) # number of samples in one period.
    x = np.zeros(N)

    n = N / M # number of periods in the range
    for i in range(int(n/2)):
        # don't forget to normalize the signal x
        x[i*2*M:(i*2+1)*M] = 1 / np.sqrt(M)
    return x
```

[ ]:
```
T = 32.0
f0 = 0.25
fs = 8
Ks = [2, 4, 8, 16]

x = make_square_wave(f0, T, fs)

fig, ax = plt.subplots(len(Ks), 1, figsize=(6, 2*len(Ks)))
fig.tight_layout(h_pad=3, w_pad=0)

for i, K in enumerate(Ks):
    X, f = compress_signal(x, fs, K)# TODO: compress the signal
    xhat, t = decompress_signal(X, f, T*fs, fs) # TODO: decompress the signal

    title = f"Reconstruction of square wave for {K=}."
    plot_reconstruction(ax[i], t, x, xhat, title)

    energy_diff = abs(inner_product(x, x) - inner_product(xhat, xhat))
    print(f"{K=:<2} | {energy_diff:.4f}")
plt.show()
```

```
K=2  | 0.3776
K=4  | 0.1300
K=8  | 0.0484
K=16 | 0.0039
```



The square wave was easier to reconstruct than the square pulse because it is periodic.

Across the board, with all K values, the square wave's energy difference is much less than the square pulse's. At K=16, for instance, the square wave had an energy difference of 0.0039, while the square pulse's energy difference was 0.0010: a 4x difference. This showcases that the reconstruction of the square wave converges much more quickly.

The square wave has a period of, in this case, T/4, which means that its period repeats 4 times in the domain. (The square pulse does not repeat itself at all.) Since sine/cosine waves are periodic, the DFT can take advantage of this periodicity to create a more accurate representation. All DFT coefficients with f less than 4 can be discarded, allowing for the emergence of higher frequency components (and thus higher accuracy) more quickly. If you look at the plotted reconstructions, you can see that at K=16 for the square pulse, even the highest frequency sine wave component is not very high frequency: we can clearly make out a bunch of loopy structures. In the square wave, in contrast, at the same K=16, contains much higher frequency components: there is an angular fast-moving squiggly line on top of the square wave, and we can't really see any loopy structures. These high frequency components do not have a high amplitude, leading them to be cut off at lower Ks, but they do contribute to shaping the reconstruction to be more accurate, allowing for angular structures instead of curvy structures.

From the get go, even with K=2, the reconstructed function somewhat flowed along the general path of the square wave, sharing ups and down. Further additions only served to refine this path. On the contrary, in a square pulse, the wave starts high and finishes low. This is contradictory to the nature of sinusoidal waves which do not have global changes in size, leading to the necessity of many more waves in order to approximate this unnatural phenomenon.

5 2. Speech processing

For this part of the lab you are asked to record audio. The following 2 cells implement a `record(T,fs)` function which you can use to do this, if you are using colab. **NB:** Recording works in firefox, but not in safari. Tested on windows and mac.

If you are running on your computer (not colab), then feel free to use the class provided on the website.

Finally, if neither of this works you can make an audio recording on your phone on laptop, save it as a file and upload to colab. Ask your TAs for help with this.

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

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

RECORD = """
const sleep  = time => new Promise(resolve => setTimeout(resolve, time))
const b2text = blob => new Promise(resolve => {
    const reader = new FileReader()
    reader.onloadend = e => resolve(e.srcElement.result)
```

```
    reader.readAsDataURL(blob)
})
var text = document.createTextNode("TEST")
document.querySelector("#output-area").appendChild(text)
var record = time => new Promise(async resolve => {
    chunks = []
    stream = await navigator.mediaDevices.getUserMedia({ audio: true })
    recorder = new MediaRecorder(stream)
    recorder.ondataavailable = e => chunks.push(e.data)
    recorder.onstop = async ()=>{
        blob = new Blob(chunks)
        text = await b2text(blob)
        resolve(text)
    }
    recorder.start()
    text.data = "Start talking"
    await sleep(time)
    text.data = "Stop talking"
    recorder.stop()
})
"""

def record(T, fs):
    """
    Record audio from microphone in colab.
    Args:
        T: Duration in seconds.
        fs: Sampling rate.
    Returns:
        A numpy array of length approximately T*fs.
    """
    display(Javascript(RECORD))
    s = output.eval_js(f"record({T*1000})")
    b = b64decode(s.split(',')[1])
    samples = (
        sum(AudioSegment.from_file(BytesIO(b)).split_to_mono())
        .set_frame_rate(fs)
        .get_array_of_samples()
    )
    fp_arr = np.array(samples).T.astype(np.float32)
    fp_arr /= np.iinfo(samples.typecode).max
    return fp_arr
```

Example how to record audio.

[ ]:
```
T = 3.0
fs = 20_000
```

```
x = record(T, fs)
print(f"Recording duration was {len(x) / fs} seconds.")
```

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

```
Recording duration was 2.94 seconds.
```

Example how to play back audio. Note that `autoplay=True` will play audio automatically.

[ ]:
```
Audio(data=x, rate=20_000, autoplay=True)
```

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

5.1 2.1 Record, graph, and play your voice

We leave the rest of the lab up to you.

To complete this question, you might need to do idft of large signals (high f_s). To avoid runtime issues, you can use the ifft function instead. The following is an example of how we can implement it. Notice that this might take arguments different from your **compute_idft** function and might work in a different manner. You can find after the code a few examples that show how it works.

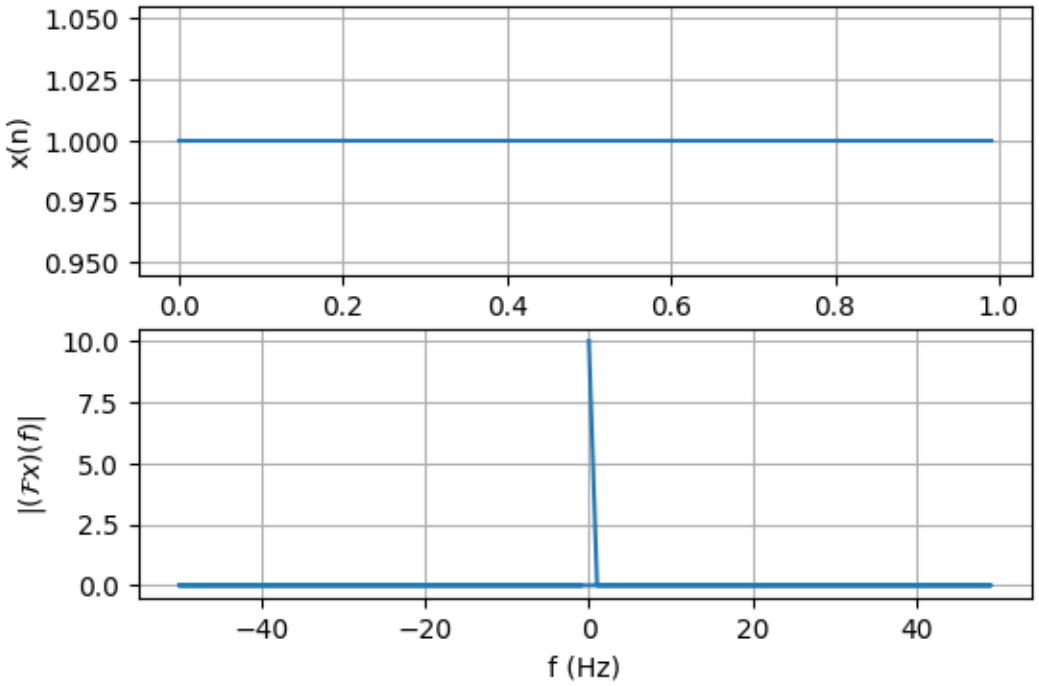
[ ]:
```
def compute_ifft(X, N, fs: int, truncated=True):
    """
    Compute the iDFT of X.

    Args:
        X: Spectrum.
        N: length of the produced time signal
        fs: Sampling frequency.
        truncated: If true then X is defined over the frequency range [0,int(N/
 ↪2)]. If false then [0,N-1].
    Returns:
        x: iDFT of X and it should be real.
        t: real time instants in the range [0, T]
    """
    if truncated:
      if N%2 == 0:
        assert 2*(len(X)-1) == N
        X_new = np.append(X, np.flip(np.conjugate(X[1:-1])))
      else:
        assert 2*(len(X)-1) == N-1
        X_new = np.append(X, np.flip(np.conjugate(X[1:])))
    else:
      X_new = X

    x = np.fft.ifft(X_new, norm="ortho")
    assert len(x) == N
    t = np.arange(N)/fs
```

```
    return x.real, t
```

The following shows how you can use it.

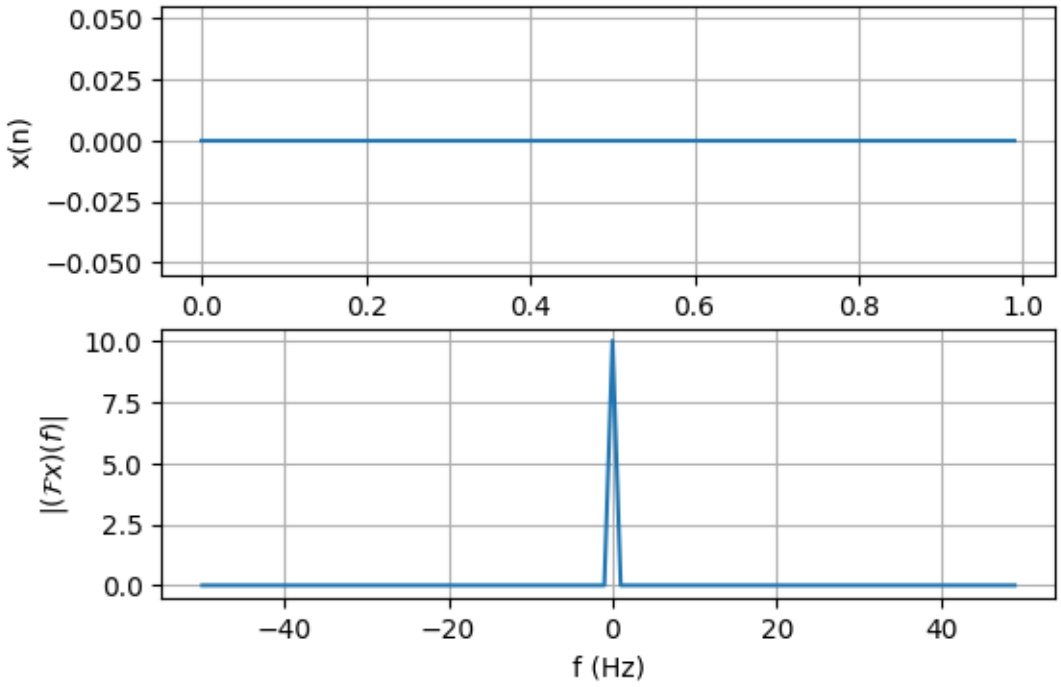
Important: These are just examples and your actual implementations of this in 2.2 might look slighlty different.

[ ]:
```
fs = 100
T = 1
N = int(T*fs)

x = np.ones(N)
X, f = compute_fft(x, N, center_frequencies=False)
xhat, t = compute_ifft(X[:int(N/2)+1], N, fs, truncated=True)
plot_signal_and_dft(xhat, t, X, f)
```



[ ]:
```
fs = 100
T = 1
N = int(T*fs)

x = np.ones(N)
X, f = compute_fft(x, N, center_frequencies=True)
# Compressed
X_K = X[0:30]
X_half = np.append(X_K, np.zeros(int(N/2)+1-len(X_K), dtype=complex))
```

```
xhat, t = compute_ifft(X_half, N, fs, truncated=True)
plot_signal_and_dft(xhat, t, X, f)
```

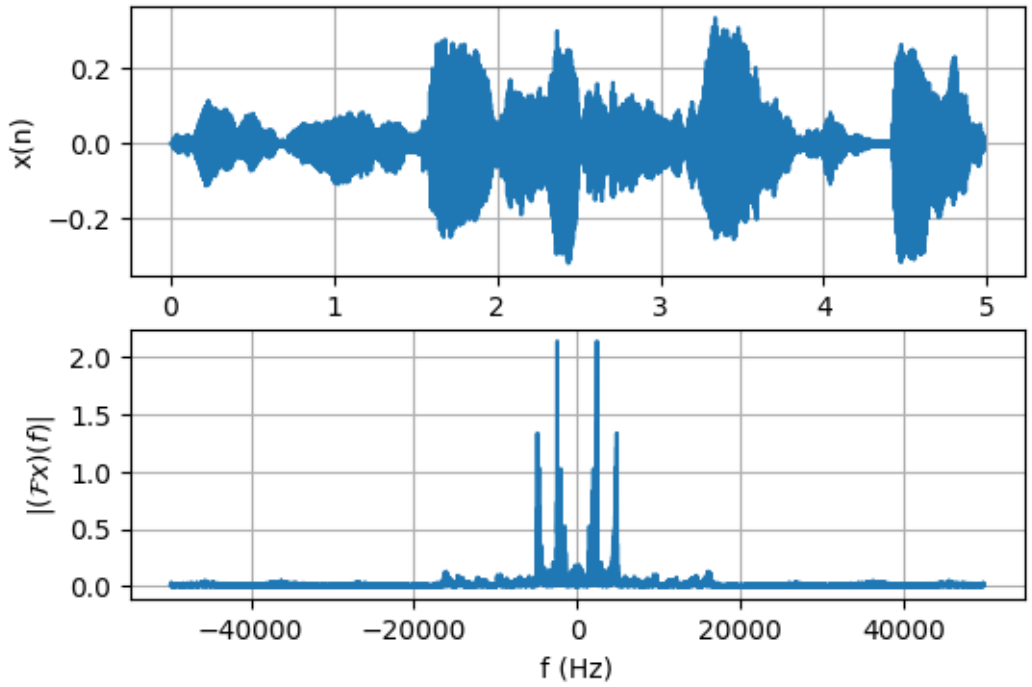


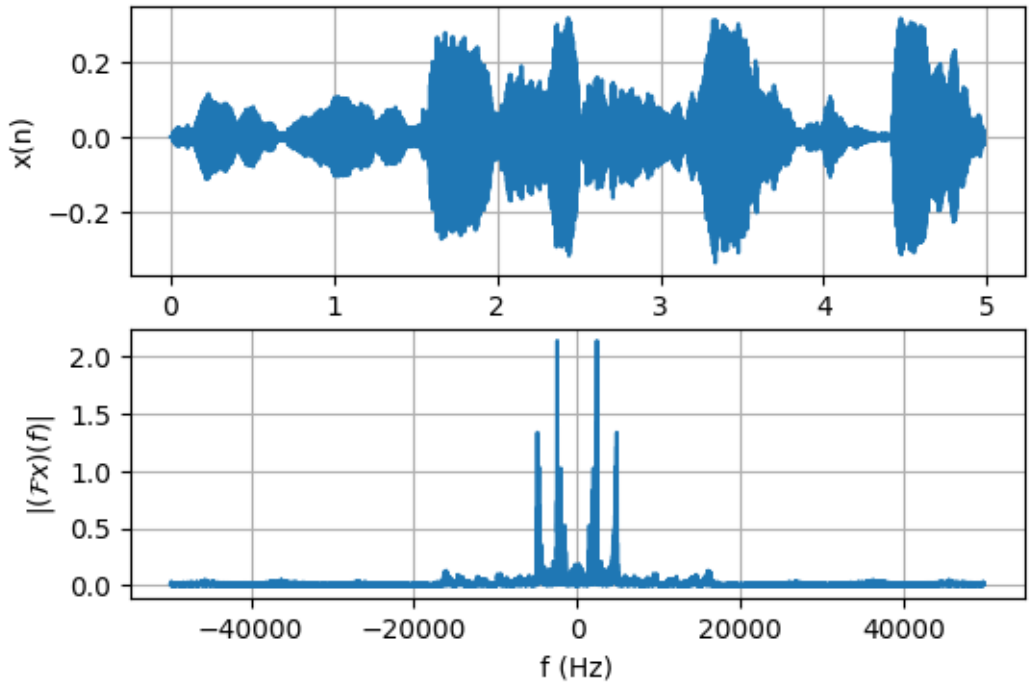
[ ]:
```
T = 5.0
fs = 20_000
x = record(T, fs)
print(f"Recording duration was {len(x) / fs} seconds.")
```

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

```
Recording duration was 4.98 seconds.
```

[ ]:
```
# TODO: Plot your voice, compute its DFT and plot its magnitude.
N = int(len(x))
t = np.linspace(0, T, N)
X, f = compute_fft(x, N, center_frequencies=True)
xhat, t = compute_ifft(X, N, fs, truncated=False)
plot_signal_and_dft(x, t, X, f)
plot_signal_and_dft(xhat, t, X, f)
```





[ ]:
```
Audio(data=x, rate=20_000, autoplay=True)
```

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

```
[ ]: Audio(data=xhat, rate=20_000, autoplay=True)
```

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

5.2 2.2 Voice compression

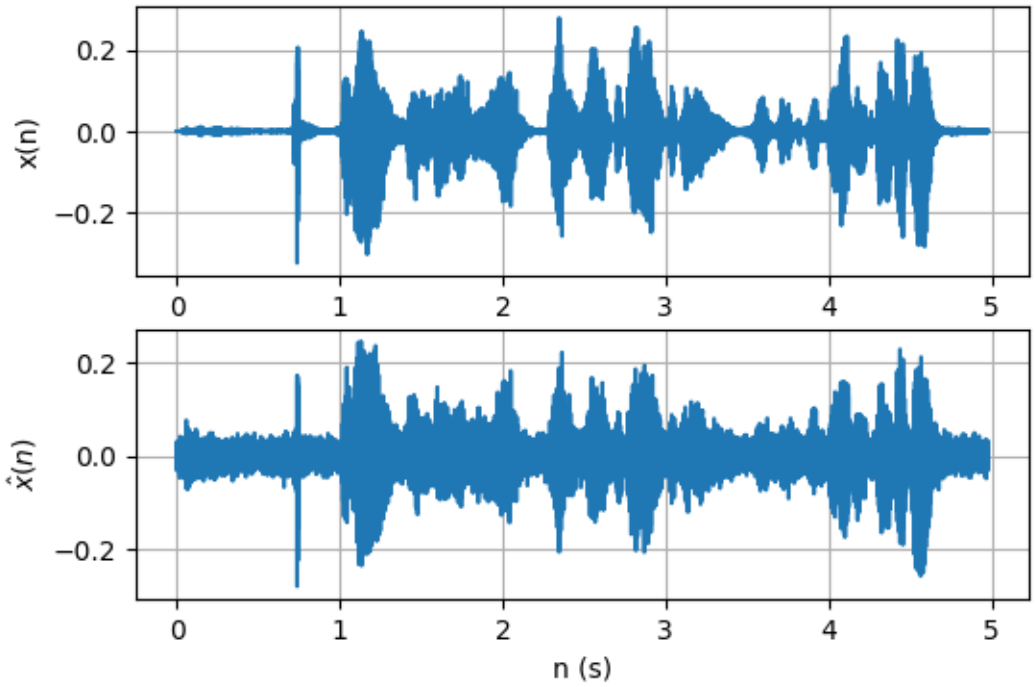
```
[ ]: # Helper function to plot the reconstruction of the voice signal depending on␣
      ↪the compression factor
def plot_voice(t, x, xhat, title=None, dft=False):
  fig, ax = plt.subplots(2, 1, figsize=(6,4))
  fig.suptitle(title)

  ax[0].plot(t, x)
  ax[0].set_xlabel("f (Hz)" if dft else "n (s)")
  ax[0].set_ylabel("|X(f)|" if dft else "x(n)")
  ax[0].grid(True)

  ax[1].plot(t, xhat)
  ax[1].set_xlabel("f (Hz)" if dft else "n (s)")
  ax[1].set_ylabel("$|\hat{X}(f)|$" if dft else "$\hat{x}(n)$")
  ax[1].grid(True)
```

```
[ ]: #this is fast decompression
def fast_reconstruct_signal(X, N: int, fs: int):
    x,t = compute_ifft(X,N,fs,truncated=True)
    return x, t
def fast_decompress_signal(X, f, N, fs):
    assert (0 <= f).all() and (f <= fs/2).all()
    f/=fs
    f*=N
    X_new = np.zeros(int(N/2)+1,dtype=np.complex128)
    j= 0
    for i in f:
       X_new[int(i)] = X[j]
       j+=1
    x,t = fast_reconstruct_signal(X_new, N, fs)
    return x,t
def edited_plot_reconstruction(ax, t,x,xhat, title):
    """
    Plots the signal and the reconstructed version on an axis.
    This function does **not** call plt.show() or plt.figure().

    Args:
        ax: The matplotlib axis to plot on.
        t: An array of times at which the signals are sampled.
```

```
        x: the original signal.
        xhat: the reconsructed signal. The same length as x and t.
        title: A string that gives the title of the plot.
    """
    ax.plot(t, x, label = '$x$')
    ax.plot(t, xhat, label = '$\hat{x}$')
    ax.set_xlabel("t (s)")
    ax.set_ylabel("x(t)")
    ax.legend()
    ax.grid(True)
    ax.set_title(title)
def compress_compare_play(Ks,x,fs,N):
    X, f = compress_signal(x, fs, Ks)
    xhat, t = fast_decompress_signal(X, f, N, fs)
    #plot_signal_and_dft(x, t, X, f)
    plot_voice(t, x, xhat, title=None, dft=False)
    return xhat
```

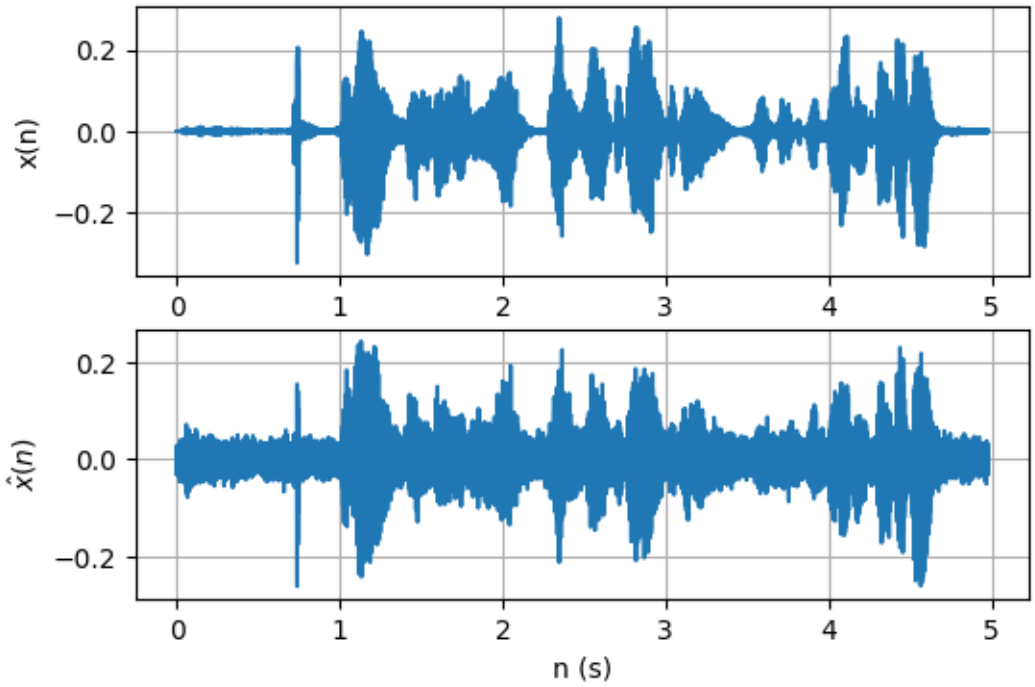
[ ]:
```
# TODO: Compress signal and use ifft to reconstruct it. Plot it using the␣
 ↪function above and then play the audio to notice the effects fo the␣
 ↪compression
# there is nearly no noise here and it is very clear (the ifft has a problem␣
 ↪converting truncated signals into normal ones)
xhat = compress_compare_play(int(N/2),x,fs,N)
Audio(data=xhat, rate=20_000, autoplay=True)
```

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



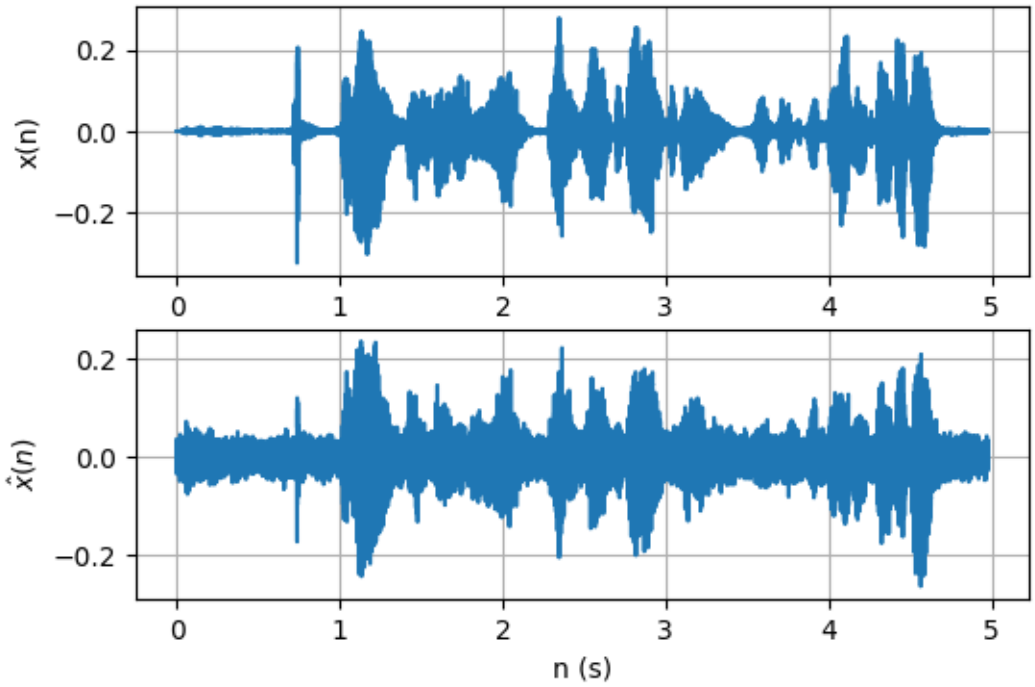
[ ]:
```
# TODO: Repeat for various compression factors and comment on the difference
#here this is compression of 4 and it is the same as compression 2 quality
xhat = compress_compare_play(int(N/4),x,fs,N)
Audio(data=xhat, rate=20_000, autoplay=True)
```

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



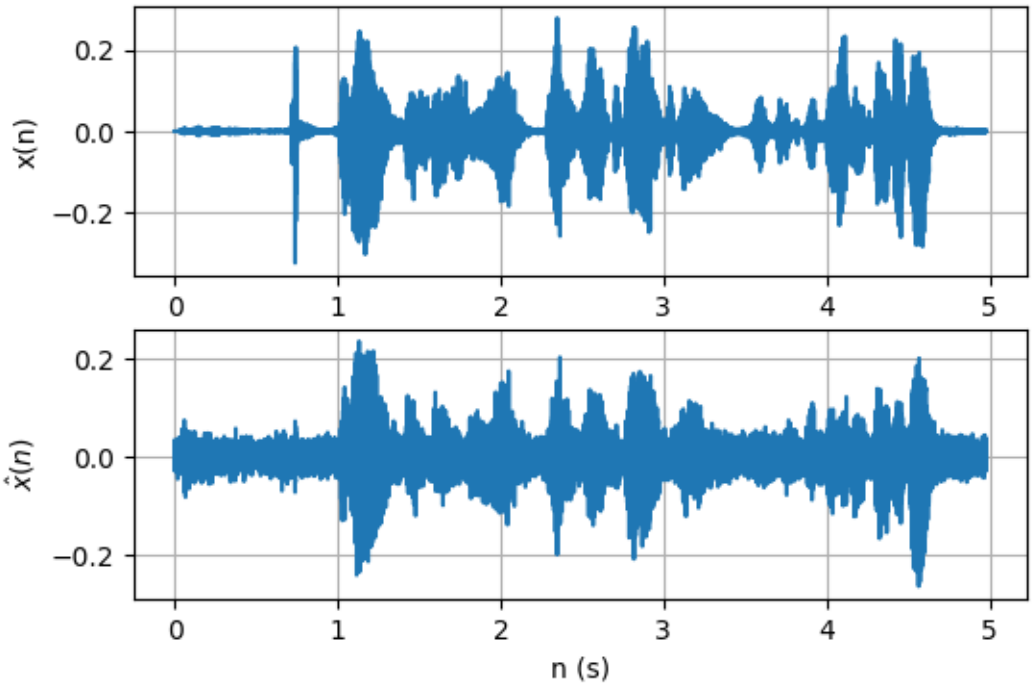
[ ]:
```
#here the distortion is mild with a word only not easily understandable
xhat = compress_compare_play(int(N/8),x,fs,N)
Audio(data=xhat, rate=20_000, autoplay=True)
```

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



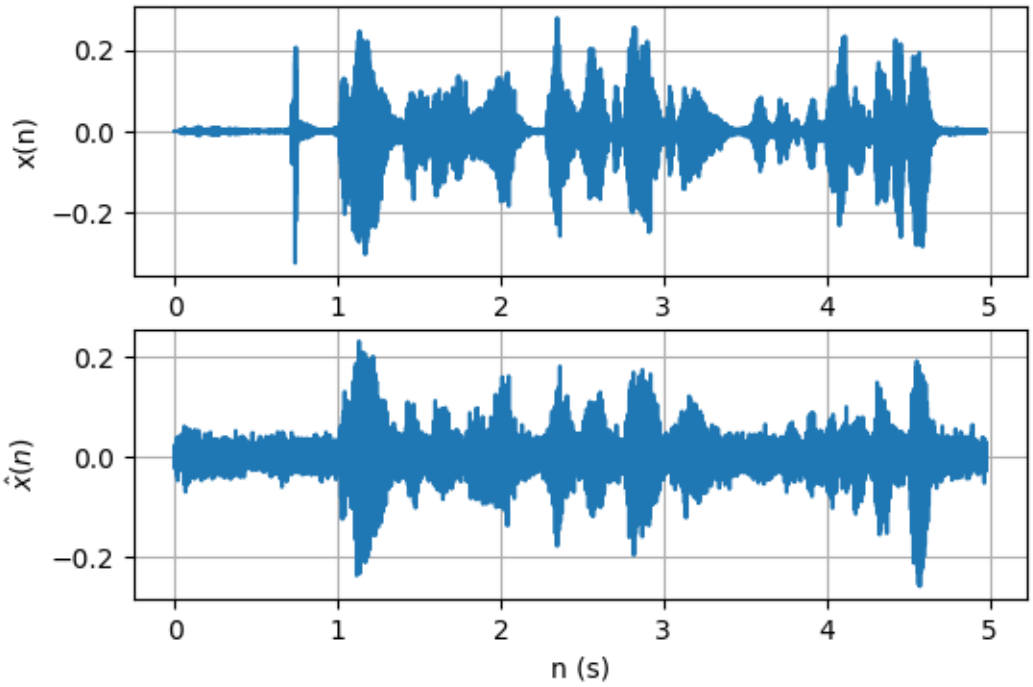
[ ]:
```
# Here the voice started to get distorted a little bit but the voice is still␣
↪easy to understand
xhat = compress_compare_play(int(N/16),x,fs,N)
Audio(data=xhat, rate=20_000, autoplay=True)
```

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



[ ]:
```
# Here the voice is so distorted due to high compression
xhat = compress_compare_play(int(N/32),x,fs,N)
Audio(data=xhat, rate=20_000, autoplay=True)
```

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



5.3 2.3 Voice masking

We leave this part up to you.

[ ]:
```
# 1. record voice
T = 5.0
fs = 20_000
x = record(T, fs)
print(f"Recording duration was {len(x) / fs} seconds.")
```

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

```
Recording duration was 4.98 seconds.
```

[ ]:
```
print("original voice:")
Audio(data=x, rate=20_000, autoplay=True)
```

```
original voice:
```

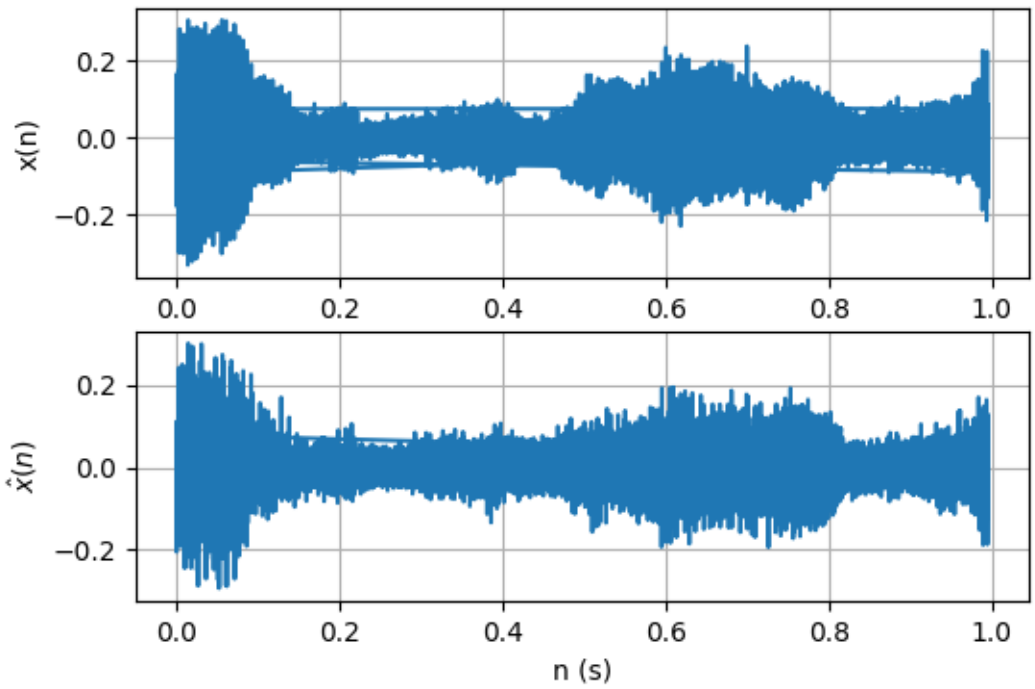
[ ]:
```
<IPython.lib.display.Audio object>
```

[ ]:
```
# 2. Do DFT
# split x into 5 chunks
x1, x2, x3, x4, x5 = np.array_split(x, 5)
xs = [x1, x2, x3, x4, x5]
```

```
xhats = []
ts = []
# do dft on each of them seperately
for x_part in xs:
  N = int(len(x_part))
  X, f = compress_signal(x_part, fs, N)

  # 3. modify the spectra in fourier space
  # mid frequencies between 200 and 2000 Hz is where most of the voice is.
  # shift mid frequencies down.
  lower_mid = np.where((f > 165) & (f < 500))[0]
  middle_mid = np.where((f > 500) & (f < 1000))[0]
  upper_mid = np.where((f >= 1000) & (f < 2000))[0]
  shift1 = 150  # Hz
  shift2 = 200  # Hz
  shift_samples1 = int(abs(shift1) / (fs/N))
  shift_samples2 = int(abs(shift2) / (fs/N))

  X[lower_mid] = np.roll(X[lower_mid], +shift_samples1)
  X[middle_mid] = np.roll(X[middle_mid], -shift_samples1)
  X[upper_mid] = np.roll(X[upper_mid], +shift_samples2)

  # add some random phase to high frequencies
  high_freq = np.where(f > 2000)[0]
  X[high_freq] *= np.exp(1j * np.random.uniform(0, np.pi, len(high_freq)))
  X[high_freq] *= np.random.uniform(0, 2, len(high_freq))

  # 4. convert back
  xhat_part, t_part = fast_decompress_signal(X, f, N, fs)
  xhats.append(xhat_part)
  ts.append(t_part)

xhat = np.concatenate(xhats)
t = np.concatenate(ts)

# plot x and xhat side by side
plot_voice(t, x, xhat, title=None, dft=False)
print("Masked voice:")
Audio(data=xhat, rate=20_000, autoplay=True)
```

```
Masked voice:
```

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



5.4 2.4 Better voice compression

We leave this part up to you.

[ ]:
```python
def chunk_and_compress(x, fs, g):
    chunk_size = int(0.1 * fs)
    compressed_chunks = []

    for i in range(0, len(x), chunk_size):
        chunk = x[i:i + chunk_size]
        if len(chunk) < chunk_size:
            chunk = np.pad(chunk, (0, chunk_size - len(chunk)))

        K = int(len(chunk) * g)
        X_K, f_K = compress_signal(chunk, fs, K)
        compressed_chunks.append((X_K, f_K))

    return compressed_chunks, chunk_size

def decompress_chunks(compressed_chunks, chunk_size, fs):
    reconstructed_signal = []

    for X_K, f_K in compressed_chunks:
```

```
        x_chunk_reconstructed, _ = fast_decompress_signal(X_K, f_K, chunk_size,␣
 ↪fs)
        reconstructed_signal.append(x_chunk_reconstructed)

    x_reconstructed = np.concatenate(reconstructed_signal)
    return x_reconstructed

T = 5.0
fs = 20_000
g = 0.2

x = record(T, fs)
compressed_chunks, chunk_size = chunk_and_compress(x, fs, g)
x_reconstructed = decompress_chunks(compressed_chunks, chunk_size, fs)



N = len(x)
t = np.linspace(0, T, N)
X, f = compute_fft(x, N, center_frequencies=True)
X_reconstructed, f_reconstructed = compute_fft(x_reconstructed, N,␣
 ↪center_frequencies=True)

plot_signal_and_dft(x, t, X, f, title="Original Signal DFT")
plot_signal_and_dft(x_reconstructed, np.linspace(0, T, len(x_reconstructed)),␣
 ↪X_reconstructed, f_reconstructed, title="Reconstructed Signal DFT")
print("original audio:")
display(Audio(data=x, rate=fs))

print("reconstructed audio:")
display(Audio(data=x_reconstructed, rate=fs))
```

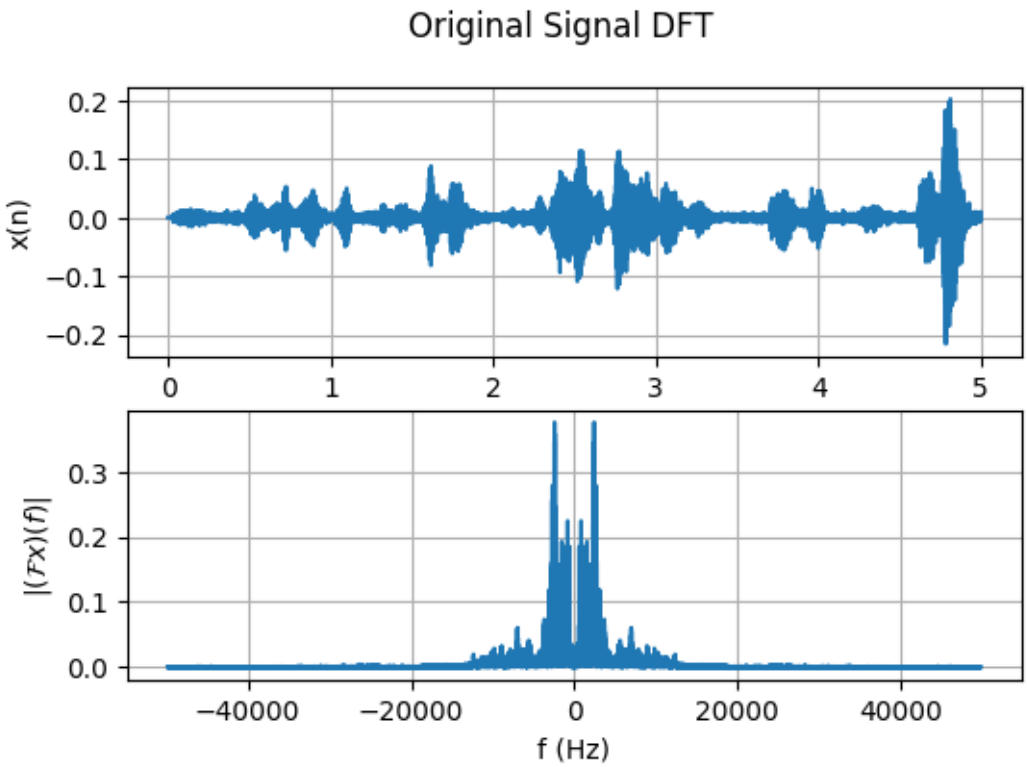
```
<IPython.core.display.Javascript object>
```

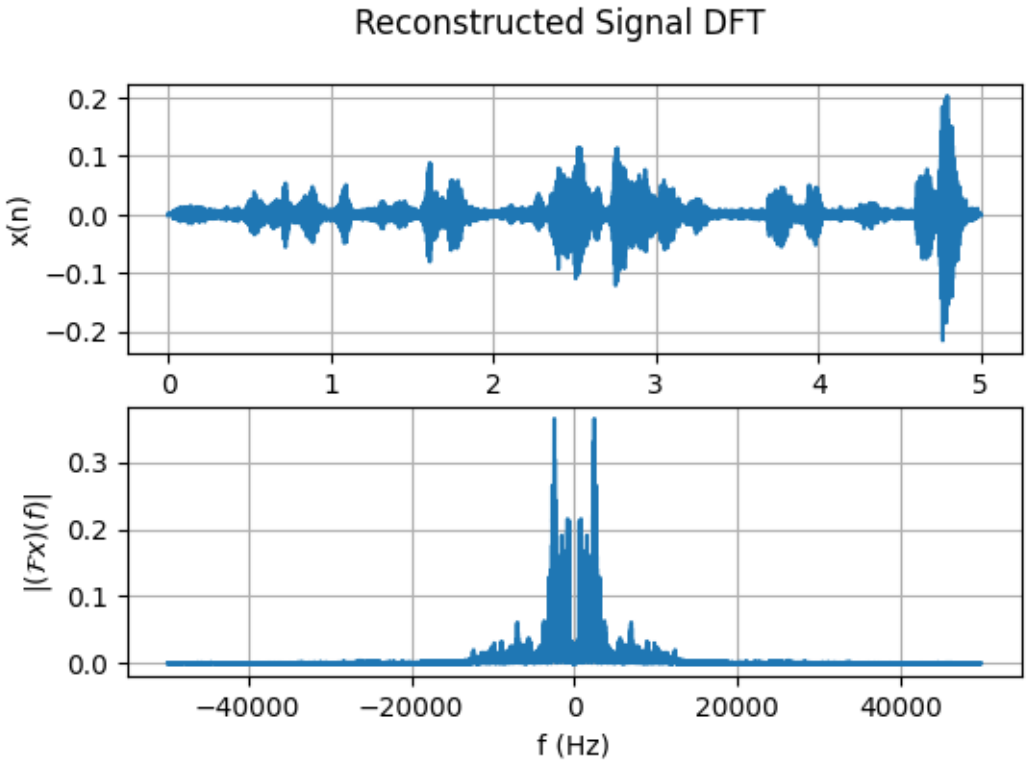
```
original audio:
```

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

```
reconstructed audio:
```

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


Original Signal DFT
x(n)
0.2
0.1
0.0
−0.1
−0.2
0
1
2
3
4
5
|(Fx)(f)|
0.3
0.2
0.1
0.0
−40000
−20000
0
20000
40000
f (Hz)


Reconstructed Signal DFT
x(n)
0.2
0.1
0.0
−0.1
−0.2
0
1
2
3
4
5
|(Fx)(f)|
0.3
0.2
0.1
0.0
−40000
−20000
0
20000
40000
f (Hz)