

# Final Project

## Introduction

### Group Members:

David Garcia Gonzalez (dgarcia23@nd.edu) || Alex Kaup (akaup@nd.edu)

## Project Overview

This project is focused on improving three key aspects of the system we use in lab. Those aspects are improving the accuracy of the Carrier Frequency Offset (CFO) estimation without increasing too much the computational complexity, increasing the Carrier Phase Offset (CPO) by moving to a decision-based CPO estimation for QPSK signals and finally changing the packet preamble from a OOK preamble to a complex-valued preamble using the Zadoff-Chu sequence.

Each aspect has a dedicated section in which we cover the motivation behind our work, what we did to achieve that improvement and what are some things we could do to make it even better as well as addressing some of the limitations of our new system.

## Carrier Frequency Offset (CFO) Estimation

### Motivation

On Lab 08, we used the FFT of a known preamble to estimate the Carrier Frequency Offset (CFO) in our system. The frequency offset is a small difference between the receiver and transmitter carrier frequencies that introduces distortion to our system and decreases our detection ability.

The accuracy of our estimation depends on the length of the preamble and the length of the FFT we perform at the receiver. To increase accuracy, we could increase the length of the FFT to get more resolution across the entire spectrum. This method is computationally expensive, since we only care about the resolution of the FFT in a determined band (around the peak frequencies) and increasing the length of the FFT spends time and resources computing the FFT in frequency bands we are not interested in. In order to increase accuracy without sacrificing too much computation time, we can do the following two step method.

First, we will take a coarse FFT (with a length of around 200). This will give us an estimate on where the peak frequency is located in our band. With this frequency, we perform a Zoom FFT in the surrounding frequencies. Using this we increase the accuracy, but only on the frequency set we are interested in since the Zoom FFT focuses on a sub-band of the entire frequency spectrum.

```
In [3]: # Import Libraries
import numpy as np
from scipy import signal
import pyfftw
import scipy.fftpack
import matplotlib.pyplot as plt
import pulse_shaping
import symbol_mod
import time
from scipy.signal import ZoomFFT
```

## Parameters

```
In [4]: #Number of data bits
N = 10000
#Sampling frequency
fs = 1000000
Ts = 1/fs
#Samples per symbol
M = 8
# Length of FFT
full_L = 1000
# Length of zoom FFT approach [length of full FFT, length of zoom FFT]
comb_L = [100, 100]
```

## Creating the signal

We will use the preamble\_generator below that was used in Lab 08 to generate the preamble. It will be a sequence of 200 bits. For the signal of interest, we create a random binary signal and add the preamble to the front of it. We use QPSK modulation and root raised cosine for the modulation. This gives us our signal of interest at baseband.

```
In [5]: # This preamble generator was provided in Lab 08
def preamble_generator():

    preamble = np.array([1,0,1,0])
    preamble = np.append(preamble, np.ones(6))
    preamble = np.append(preamble, np.zeros(10))
    preamble = np.append(preamble, np.ones(180))

    return preamble
```

```
In [6]: # Generate ideal baseband signal
def generate_baseband(N, fs, M):
    Bits = np.random.randint(0,2,N) #random data
    preamble = preamble_generator()
    packet_bits = np.append(preamble, Bits)
    preamble_length = len(preamble)
    baseband_symbols = symbol_mod.symbol_mod(packet_bits, 'QPSK', preamble_length)
    pulse_shape = 'rrc'
    baseband = pulse_shaping.pulse_shaping(baseband_symbols, M, fs, pulse_shape, 0.9, 8)

    return baseband, preamble
```

```
In [7]: baseband, preamble = generate_baseband(N, fs, M)
```

## Frequency offset generation

To simulate the effects of the channel, we treat the frequency offset as a uniform random variable from -0.01 to 0.01 of the sampling frequency. We create the offset by multiplying our signal by a complex exponential.

```
In [8]: def generate_baseband_offset(baseband, fs=fs):
# Generate the frequency offset
frequency_offset = np.random.uniform(-0.01*fs,0.01*fs)
Ts = 1/fs
```

```

t = np.arange(0, len(baseband)*Ts, Ts)
nonideal_term = np.exp(1j*2*np.pi*frequency_offset*t)
baseband_with_frequency_offset = np.multiply(baseband, nonideal_term)

return baseband_with_frequency_offset, frequency_offset

```

```

In [9]: baseband_offset, offset = generate_baseband_offset(baseband, fs)
preamble_offset = baseband_offset[0:len(preamble)]

print("Frequency offset: {:.0f} Hz".format(offset))

```

Frequency offset: 6415 Hz

## Frequency Estimation: Full FFT and Zoom FFT

### Full FFT Method

The Full FFT method was covered in Lab 08, and it consists of taking the FFT of the preamble with the frequency offset. We then find the frequency corresponding to the peak of the FFT and that is the frequency offset.

```

In [10]: def fft_baseband_offset(baseband, fs=fs, L=full_L):
# FFT
spectrum = np.abs(np.fft.fft(baseband, L))
spectrum = np.fft.fftshift(spectrum)
# Return max point
peak = np.argmax(spectrum)
offset = (peak - L/2)/L * fs

return spectrum, offset

```

```

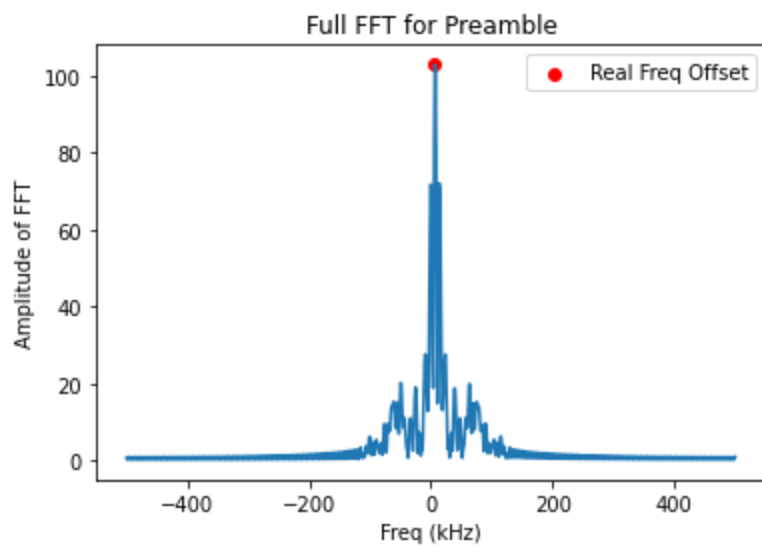
In [11]: spectrum_offset_full, offset_full = fft_baseband_offset(preamble_offset, fs, full_L)

print("Real frequency offset: {:.0f} Hz".format(offset))
print("Calculated frequency offset with full FFT: {:.0f} Hz".format(offset_full))

plt.plot(np.linspace(-fs/2000, fs/2000, full_L), np.abs(spectrum_offset_full))
plt.scatter([offset_full/1000], [np.max(spectrum_offset_full)], c='r', label="Real Freq (")
plt.xlabel("Freq (kHz)")
plt.ylabel("Amplitude of FFT")
plt.title("Full FFT for Preamble")
plt.legend(loc="upper right")
plt.show()

```

Real frequency offset: 6415 Hz  
Calculated frequency offset with full FFT: 6000 Hz



## Combined Zoom FFT Method

The combined Zoom FFT Method uses both a full FFT and a Zoom FFT. After the full FFT identifies the peak point, the Zoom FFT does a FFT around that frequency. The range of frequencies for the Zoom FFT depends on the length of the full FFT.

```
In [12]: def zoomfft_baseband_offset(baseband, fs, M, L):
transform = ZoomFFT(len(baseband), [-fs/(2*M), fs/(2*M)], L, fs=fs)
spectrum = np.abs(transform(baseband))

peak = np.argmax(spectrum)
offset = (peak*fs) / ((L-1)*M) - fs/(2*M)

return spectrum, offset

def zoomfft_baseband_offset_bounds(baseband, fs, bottom, top, L):
transform = ZoomFFT(len(baseband), [bottom, top], L, fs=fs)
spectrum = np.abs(transform(baseband))

peak = np.argmax(spectrum)
offset = (top - bottom)*peak/(L-1) + bottom

return spectrum, offset
```

```
In [13]: def combined_baseband_offset(baseband, fs, L):
_, freq_est = fft_baseband_offset(baseband, fs, L[0])
spectrum_offset_zoom, offset_zoom = zoomfft_baseband_offset_bounds(baseband, fs, freq_est, freq_est+fs/2, L)

return spectrum_offset_zoom, offset_zoom, freq_est
```

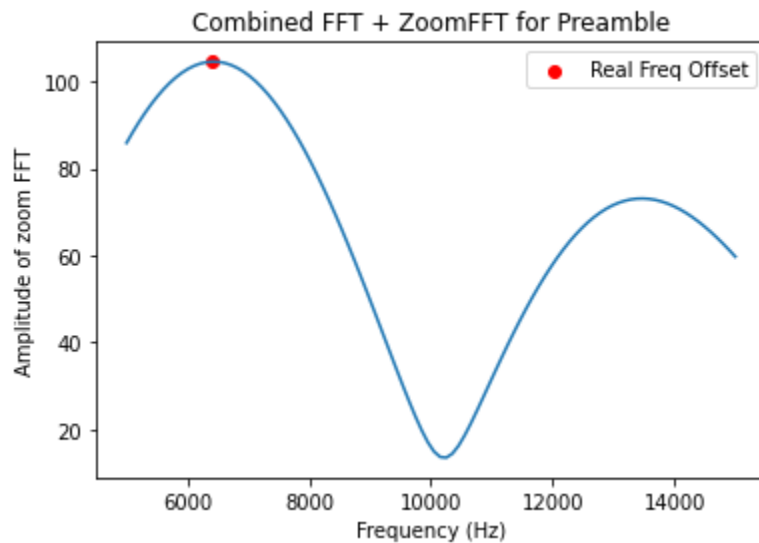
```
In [14]: spectrum_offset_zoom, offset_zoom, freq_est_full = combined_baseband_offset(preamble_offset, fs, L)

print("Real frequency offset: {:.0f} Hz".format(offset))
print("Calculated offset with Zoom FFT: {:.0f} Hz".format(offset_zoom))

plt.plot(np.linspace(freq_est_full-(0.5)*fs/comb_L[0], freq_est_full+(0.5)*fs/comb_L[0], comb_L[0]),
plt.scatter([offset_zoom], [np.max(spectrum_offset_zoom)], c='r', label="Real Freq Offset")
plt.xlabel("Frequency (Hz)")
plt.ylabel("Amplitude of zoom FFT")
plt.title("Combined FFT + ZoomFFT for Preamble")
plt.legend()
plt.show()
```

Real frequency offset: 6415 Hz

Calculated offset with Zoom FFT: 6414 Hz



## Error Analysis and Execution Time

Now we are interested in analyzing the accuracy of the two approaches and the execution time of both. We examine the "amount of frequency" that we miss in the CFO estimations and the error percentage compared to the actual CFO.

In [19]:

```
error_full = 0
error_zoom = 0

time_full = 0
time_zoom = 0

comb_len = [500, 500]
full_len = 8000

num_full_better = 0

tests = 100 # Do not change this parameter

for i in range(tests):

    baseband, preamble = generate_baseband(N, fs, M)
    baseband_offset, offset = generate_baseband_offset(baseband, fs)
    preamble_offset = baseband_offset[0:200]

    # Full FFT
    start_time = time.time()
    spectrum_offset_full, offset_full = fft_baseband_offset(preamble_offset, fs, full_len)
    new_time_full = time.time() - start_time
    time_full += new_time_full

    # Zoom FFT
    start_time = time.time()
    spectrum_offset_zoom, offset_zoom, _ = combined_baseband_offset(preamble_offset, fs,
                                                                    full_len, comb_len)
    new_time_zoom = time.time() - start_time
    time_zoom += new_time_zoom

    # Errors
    #new_error_full = np.abs((offset - offset_full)*100/(offset))
    new_error_full = np.abs(offset - offset_full)
    error_full += new_error_full
    #new_error_zoom = np.abs((offset - offset_zoom)*100/(offset))
    new_error_zoom = np.abs(offset - offset_zoom)
```

```

error_zoom += new_error_zoom

if(new_error_full < new_error_zoom):
    num_full_better +=1

error_full /= tests
time_full /= tests
error_zoom /= tests
time_zoom /= tests

print("Average error full FFT:", error_full, "Hz")
print("Average time full FFT: ", time_full*1000, "ms")
print("Average error zoom FFT:", error_zoom, "Hz")
print("Average time zoom FFT: ", time_zoom*1000, "ms")
print("\nNumber of time zoom FFT was less accurate:", num_full_better, "/", tests, "tests")

```

```

Average error full FFT: 30.1298783007762 Hz
Average time full FFT: 0.5107283592224121 ms
Average error zoom FFT: 2.2173653672885343 Hz
Average time zoom FFT: 0.5660462379455566 ms

```

```

Number of time zoom FFT was less accurate: 4 / 100 tests

```

As we see above, the execution time of both approaches is around 0.5 ms, but the Zoom FFT method outperforms the regular full FFT method in calculating the CFO. We ran a lot of tests and found that the Zoom FFT outperformed constantly the full FFT, but there is also space for a more robust evaluation of the execution time vs accuracy trade-offs in both methods.

## Characterizing full FFT

The following two sections are about characterizing the performance of both the full FFT and zoom FFT. We wanted to see if the execution time of the Zoom FFT increased in a similar way to the full FFT as the FFT window increased. Although there are some interesting insights on how both they perform, it is not too relevant to the project.

In [32]:

```

full_len = 10
tests = 1000
timing = []
lengths = []

while (full_len < 10000):
    time_full = 0
    for i in range(tests):
        # Full FFT
        start_time = time.time()
        spectrum_offset_full, offset_full = fft_baseband_offset(preamble_offset, fs, full_len)
        new_time_full = time.time() - start_time
        time_full += new_time_full

    time_full /= tests

    timing.append(time_full)
    lengths.append(full_len)
    full_len += 50

plt.scatter(lengths, timing)
plt.title("Execution Time for full FFT")
plt.xlabel("Length of FFT")
plt.ylabel("Execution time (s)")
plt.show()

```



## Characterizing combined Zoom + Full FFT

In [34]:

```
comb_len = [10, 10]
tests = 1000
timing = []
lengths_zoom = []
lengths_full = []

while (comb_len[0] < 5000 and comb_len[1] < 5000):
    time_zoom = 0
    for i in range(tests):
        # Zoom FFT
        start_time = time.time()
        spectrum_offset_zoom, offset_zoom, _ = combined_baseband_offset(preamble_offset,
                                new_time_zoom = time.time() - start_time
        time_zoom += new_time_zoom

    time_zoom /= tests

    timing.append(time_zoom)
    lengths_full.append(comb_len[0])
    comb_len[0] += 50

timing = []
comb_len = [10, 10]
while (comb_len[0] < 5000 and comb_len[1] < 5000):
    time_zoom = 0
    for i in range(tests):
        # Zoom FFT
        start_time = time.time()
        spectrum_offset_zoom, offset_zoom, _ = combined_baseband_offset(preamble_offset,
                                new_time_zoom = time.time() - start_time
        time_zoom += new_time_zoom

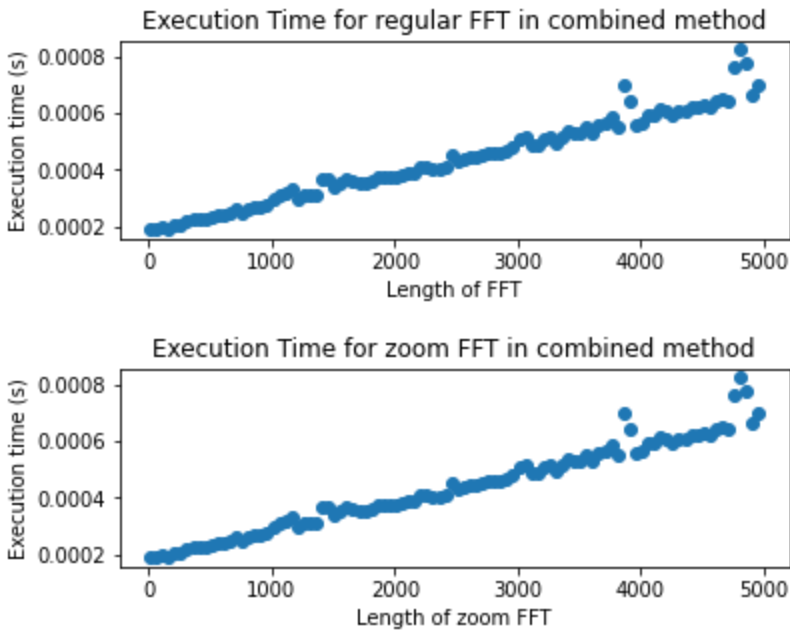
    time_zoom /= tests

    lengths_zoom.append(comb_len[1])
    timing.append(time_zoom)
    comb_len[1] += 50
```

In [33]:

```
plt.subplot(2,1,1)
plt.scatter(lengths_full, timing)
```

```
plt.title("Execution Time for regular FFT in combined method")
plt.xlabel("Length of FFT")
plt.ylabel("Execution time (s)")
plt.show()
plt.subplot(2,1,2)
plt.scatter(lengths_zoom, timing)
plt.title("Execution Time for zoom FFT in combined method")
plt.xlabel("Length of zoom FFT")
plt.ylabel("Execution time (s)")
plt.show()
```



## Carrier Phase Offset (CPO) Estimation

In Lab 08 we used a OOK preamble signal in order to estimate the phase offset in the channel. We sent a OOK preamble and measured the angle between the real and imaginary parts of the received signal. The new method for CPO estimation relies on QPSK preambles and it allows for constant evaluation of the phase offset.

### General framework for phase detection

The code below shows a generic example of how we can calculate the CPO for a QPSK preamble. The difference between the two methods lies in the computational efficiency of the approach. For Method #1, we need to take two arctan operations (computationally expensive), while Method #2 only requires simpler arithmetic computations.

In [23]:

```
# Generate random QPSK symbol
x = np.random.choice([-1, 1])*np.sqrt(2)/2
y = np.random.choice([-1, 1])*np.sqrt(2)/2
s = x + 1j*y

# Generate random phase offset
phase_offset = np.random.uniform(-np.pi, np.pi)
print("Phase offset (°): {:.2f}°".format(phase_offset*180/np.pi))

# Add phase offset
s_offset = s*np.exp(1j*phase_offset)

# First method for error estimation
e_k = np.angle(s_offset) - np.arctan(np.sign(np.imag(s_offset))/np.sign(np.real(s_offset)))
```



```

print("Phase estimation #1: {:.2f}°".format(e_k*180/np.pi))

# Second method for error estimation
e_k = (np.imag(s_offset)*np.sign(np.real(s_offset)) - np.real(s_offset)*np.sign(np.imag(s_offset)))/(np.real(s_offset)**2 + np.imag(s_offset)**2)
print("Phase estimation #2: {:.2f}°".format(np.angle(1j*e_k + np.sqrt(1 - e_k**2))*180/np.pi))

# Correct for phase offset
s_corrected = s_offset*(1j*e_k + np.sqrt(1 - e_k**2))

Phase offset (°): -16.82°
Phase estimation #1: -196.82°
Phase estimation #2: -16.82°

```

## Ambiguity and CPO

For QPSK, our phase offset detector has a  $\pi/2$  ambiguity. This means that when we correct our signal we might get a rotated version of our original symbols by 0, 90, 180 or 270 degrees. To compensate for the rotation, we can send a known preamble in order to figure out the rotation and compensate for it.

In [181]...

```

# Estimates the CPO with a pi/2 ambiguity
def cpo_estimation(preamble, known_preamble):

    e_k = (np.imag(preamble)*np.sign(np.real(preamble)) - np.real(preamble)*np.sign(np.imag(preamble)))/(np.real(preamble)**2 + np.imag(preamble)**2)
    e_k = np.arcsin(e_k)

    return np.mean(e_k)

# Estimates the CPO with the rotation caused by the ambiguity
def cpo_rotation_angle(preamble, known_preamble):

    e_k = cpo_estimation(preamble, known_preamble)

    preamble_corrected = preamble*np.exp(-1j*e_k)

    angle_rotation = np.mean(np.angle(preamble_corrected) - np.angle(known_preamble))

    return angle_rotation + e_k

# Adjusts for CPO and rotation of given sequence with known preamble
def cpo_rotate(sequence, preamble_length, known_preamble):

    angle = cpo_rotation_angle(sequence[0:preamble_length], known_preamble)

    sequence_corrected = sequence*np.exp(-1j*angle)

    return sequence_corrected

```

This code below does some testing on a randomly generated QPSK signal with a known preamble. It then confirms that we were able to get the signal back with the phase and ambiguity corrected.

In [190]...

```

N = 10 # Length of signal
x = np.random.choice([-1, 1], N)*np.sqrt(2)/2
y = np.random.choice([-1, 1], N)*np.sqrt(2)/2
payload = x + 1j*y
preamble = [np.sqrt(2)/2 + 1j*np.sqrt(2)/2] # Known preamble
packet = np.append(preamble, payload) # Add payload and preamble
phase_offset = np.random.uniform(-np.pi, np.pi)
packet_offset = packet*np.exp(1j*phase_offset)
preamble_offset = packet_offset[0]

print("Phase offset: {:.2f}°".format(np.angle(preamble_offset)*180/np.pi))

```

```
print("Phase offset estimation: {:.2f}°".format(np remainder(cpo_rotation_
print("Equality test after phase correction: {}".format(np.isclose(cpo_rotate(packet_of
```

```
Phase offset: 157.93°
Phase offset estimation: 157.93°
Equality test after phase correction: True
```

## Preamble and Timing Offset: Zadoff-Chu Sequence

### Generating the Zadoff-Chu sequence

In [15]:

```
# N: Length of ZC sequence
# u: Less than N, coprime with N
def generateZC(N, u):
    n = np.linspace(0, N-1, N)

    ZC = np.exp(-1j*np.pi*(n+1)*n*u/N)

    return ZC
```

### Zadoff-Chu properties

As we can see below, ZC sequences exhibit some interesting correlation properties. The auto correlation of a Zadoff-Chu sequence with a cyclically shifted version of itself is zero, i.e., it is non-zero only at one instant which corresponds to the cyclic shift. This property is useful for frame detection.

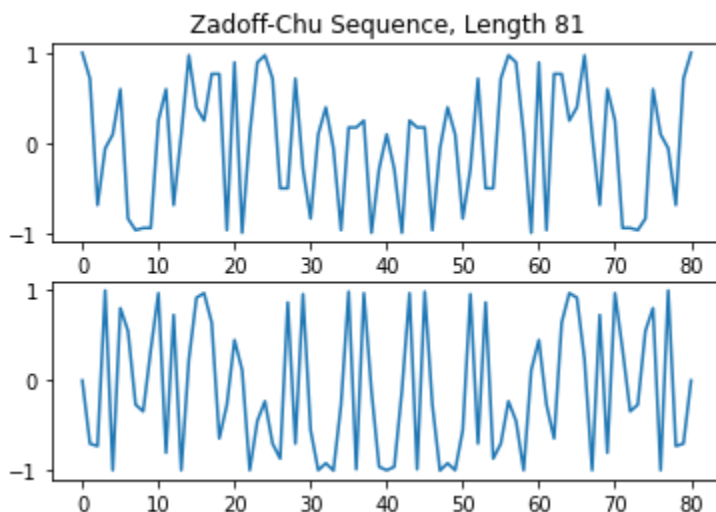
In [206...]

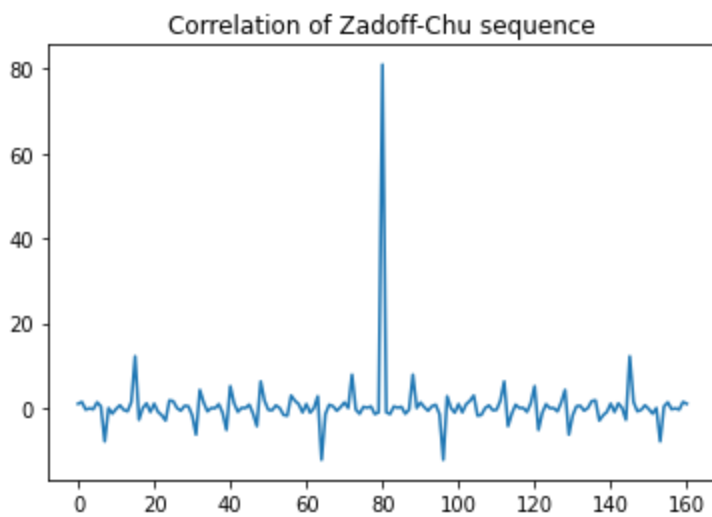
```
ZC = generateZC(81, 10)

X = np.correlate(ZC, ZC, mode="full")

ax1 = plt.subplot(2,1,1)
plt.plot(np.real(ZC))
plt.title("Zadoff-Chu Sequence, Length 81")
ax2 = plt.subplot(2,1,2)
plt.plot(np.imag(ZC))
plt.show()

plt.title("Correlation of Zadoff-Chu sequence")
plt.plot(np.real(X))
plt.show()
```





## Comparing the ZC sequence with the regular OOK preamble

In Lab 07 we studied frame detection using a OOK preamble. For the OOK preamble, we did a fft convolution between the entire data frame (payload and preamble) and return the point where the fft convolution is highest. That is the detected start frame. For the Zadoff-Chu sequence, we can do the correlation between the entire frame (payload and preamble) with the known ZC preamble. We want to compare the performance of the OOK preamble frame detection and QPSK frame detection compared to the ZC preamble detection.

### OOK Frame Detection

```
In [17]: def frame_sync(data_bb_ac, known_preamble_ac):

    matched_filter_coef = np.flip(known_preamble_ac, 0)
    crosscorr = signal.fftconvolve(data_bb_ac, matched_filter_coef)

    return crosscorr

count = 0 # This counts the amount of errors

for i in range(10000):
    #preamble = np.random.choice([-1, 1], 10)*np.sqrt(2)/2
    preamble = np.random.randint(0, 2, 10)

    #payload = np.random.choice([-1, 1], 128)*np.sqrt(2)/2
    payload = np.random.randint(0, 2, 128)
    start_index = np.random.randint(0, 128)

    final_payload = np.insert(payload, start_index, preamble)

    crosscorr = frame_sync(final_payload, preamble)

    peak = np.argmax(crosscorr) - len(preamble) + 1

    if (peak != start_index):
        count += 1

print("Percentage of tests with frame start mismatch in OOK: {}".format(count*100/10000))
```

Percentage of tests with frame start mismatch in OOK: 65.01%

### QPSK Frame Detection

```
In [18]: def frame_sync(data_bb_ac, known_preamble_ac):
```

```

        matched_filter_coef = np.flip(known_preamble_ac, 0)
        crosscorr = signal.fftconvolve(data_bb_ac, matched_filter_coef)

        return crosscorr

count = 0 # This counts the amount of errors

for i in range(10000):
    preamble = np.random.choice([-1, 1], 10)*np.sqrt(2)/2

    payload = np.random.choice([-1, 1], 128)*np.sqrt(2)/2

    start_index = np.random.randint(0, 128)

    final_payload = np.insert(payload, start_index, preamble)

    crosscorr = frame_sync(final_payload, preamble)

    peak = np.argmax(crosscorr) - len(preamble) + 1

    if (peak != start_index):
        count += 1

print("Percentage of tests with frame start mismatch in QPSK: {}".format(count*100/10000))

```

Percentage of tests with frame start mismatch in QPSK: 5.94%

In [19]:

```

def frame_sync(data_bb_ac, known_preamble_ac):

    crosscorr = np.correlate(data_bb_ac, known_preamble_ac, mode="full")

    return crosscorr

count = 0

frequency_offset = np.random.uniform(-0.01*fs, 0.01*fs)
Ts = 1/fs
t = np.arange(0, len(baseband)*Ts, 41593)
nonideal_term = np.exp(1j*2*np.pi*frequency_offset*t)

for i in range(10000):
    N = 10
    u = 7

    n = np.linspace(0, N-1, N)

    preamble = np.exp(-1j*np.pi*(n+1)*n*u/N)

    payload = np.random.choice([-1, 1], 128)*np.sqrt(2)/2

    start_index = np.random.randint(0, 128)

    final_payload = np.multiply(np.insert(payload, start_index, preamble), nonideal_term)

    crosscorr = frame_sync(final_payload, preamble)

    peak = np.argmax(np.real(crosscorr)) - len(preamble) + 1

    if (peak != start_index):
        count += 1

print("Percentage of tests with frame start mismatch in ZC: {}".format(count*100/10000))

```


```
/home/dgarcia23/.local/lib/python3.8/site-packages/numpy/lib/function_base.py:5287: Compl
exWarning: Casting complex values to real discards the imaginary part
  values = array(values, copy=False, ndmin=arr.ndim, dtype=arr.dtype)
Percentage of tests with frame start mismatch in ZC: 0.0%
```

As we can see, for a preamble of only 10 symbols, we do not get any errors for the Zadoff-Chu sequence as the preamble, while we get 65% of tests with a mismatch for OOK and around 6% for QPSK preambles. We recognize that errors might occur if continued testing is done, we have runned the previous script multiple times and we haven't had any errors yet.

## Combining Zadoff-Chu with CFO and CPO estimation

The system below is an example of how to combine the ZC sequence for frame detection with CFO and CPO estimation. We tried to use our current methods for CPO and CFO estimation with the Zadoff-Chu sequence but we were ultimately unsuccessful. Our system uses a simple QPSK configuration preamble (sent at the start of the transmission to match the two PLLs) and then use the Zadoff-Chu sequence for frame detection once the PLLs are matched.

There are some limitations in this design. We recognize that the best solution would be to use the ZC sequence for the phase and frequency offset. In the current state, there needs to be some synchronization between the transmitter and the receiver for the system to match the two PLLs. We believe that although this limitation exists, it is still an improvement over the system developed in lab. Once the PLL matching is done (which we have also improved), the ZC sequence provides really good performance for the frame start detection (we haven't detected any errors yet). If the PLLs phase and frequency stays relatively constant over time after the first matching is done, our system is way more reliable than other systems with similar preamble length designed with OOK or even QPSK.

 Full system

```
In [ ]: # Creates a frequency and phase offset
def generate_offset(baseband, fs=fs):
    # Generate the frequency offset
    frequency_offset = np.random.uniform(-0.01*fs, 0.01*fs)
    # Generate the phase offset
    phase_offset = np.random.uniform(-np.pi, np.pi)
    Ts = 1/fs
    t = np.arange(0, len(baseband)*Ts, Ts)
    nonideal_term = np.exp(1j*(2*np.pi*frequency_offset*t + phase_offset))
    baseband_with_frequency_offset = np.multiply(baseband, nonideal_term)

    return baseband_with_frequency_offset, frequency_offset, phase_offset

# Frame start correlation
def frame_sync(data_bb_ac, known_preamble_ac):

    crosscorr = np.correlate(data_bb_ac, known_preamble_ac, mode="full")

    return crosscorr
```

```
In [ ]: ZC = generateZC(10, 1) # Generate a ZC sequence of length 10
preamble = ZC

payload = (np.random.choice([1, -1], 300) + 1j*np.random.choice([1, -1], 300))*np.sqrt(2)
config = np.array(20*[1 + 1j])/np.sqrt(2) # Configuration preamble for phase and frequen
```

```

# Create the packet by placing configuration preamble at the start, and then insert the
start_index = np.random.randint(0, 128)
packet = np.insert(payload, start_index, preamble)
packet = np.append(config, packet)

packet_mod = pulse_shaping.pulse_shaping(packet, M, fs, "rect", None, None) # Modulate i
packet_mod_offset, freq_offset, phase_offset = generate_offset(packet_mod, fs=fs) # Freq

# Compensate for frequency offset
packet_zoom, freq_offset_est, _ = combined_baseband_offset(packet_mod_offset[0:10], fs,
Ts = 1/fs
t = np.arange(0, len(packet_mod)*Ts, Ts)
Digital_L0 = np.exp(1j*(-2*np.pi*freq_offset_est*t))
packet_freq_corrected = np.multiply(packet_mod_offset, Digital_L0)

# Compensate for phase offset
phase_offset_est = cpo_rotation_angle(packet_freq_corrected[0:10], config[0:10])
packet_corrected = packet_freq_corrected*np.exp(-1j*phase_offset_est)

# Demodulate
packet_demod = packet_corrected[:,M]

# Frame start detection
crosscorr = frame_sync(packet_demod, ZC)
peak = np.argmax(np.real(crosscorr)) - len(ZC) + 1 - len(config)

```

In [233...

```

# Count the amount of errors
count_frame = 0
count_phase = 0
count_freq = 0
count_demod = 0

for i in range(1000):
    ZC = generateZC(10, 1) # Generate a ZC sequence of length 10
    preamble = ZC

    payload = (np.random.choice([1, -1], 300) + 1j*np.random.choice([1, -1], 300))*np.sc

    config = np.array(20*[1 + 1j])/np.sqrt(2) # Configuration preamble for phase and fre

    # Create the packet by placing configuration preamble at the start, and then insert
    start_index = np.random.randint(0, 200)
    packet = np.insert(payload, start_index, preamble)
    packet = np.append(config, packet)

    packet_mod = pulse_shaping.pulse_shaping(packet, M, fs, "rect", None, None) # Modula
    packet_mod_offset, freq_offset, phase_offset = generate_offset(packet_mod, fs=fs) #

    # Compensate for frequency offset
    packet_zoom, freq_offset_est, _ = combined_baseband_offset(packet_mod_offset[0:10],
Ts = 1/fs
t = np.arange(0, len(packet_mod)*Ts, Ts)
Digital_L0 = np.exp(1j*(-2*np.pi*freq_offset_est*t))
packet_freq_corrected = np.multiply(packet_mod_offset, Digital_L0)

    # Compensate for phase offset
    phase_offset_est = cpo_rotation_angle(packet_freq_corrected[0:10], config[0:10])
    packet_corrected = packet_freq_corrected*np.exp(-1j*phase_offset_est)

    # Demodulate
    packet_demod = packet_corrected[:,M]

    # Frame start detection
    crosscorr = frame_sync(packet_demod, ZC)

```

```

peak = np.argmax(np.real(crosscorr)) - len(ZC) + 1 - len(config)

# Error checking
if (not(np.isclose(phase_offset, phase_offset_est, rtol=0.1))):
    count_phase += 1
if not(np.isclose(freq_offset, freq_offset_est, rtol=50)):
    count_freq += 1
if (peak != start_index):
    count_frame += 1
if not(np.isclose(packet_demod[peak+10:], packet[peak+10:], rtol=0.1).all()):
    count_demod += 1

print("Frame start errors:", count_frame)
print("Frequency offset estimation errors:", count_freq)
print("Phase offset estimation errors:", count_phase)
print("Packets with errors:", count_demod)

```

```

Frame start errors: 0
Frequency offset estimation errors: 0
Phase offset estimation errors: 125
Packets with errors: 0

```

As we can see above, for the 1000 test we ran, we always detected the start of the frame correctly. We were also able to recover the original payload without any errors for all the tests.

## Appendix

### Storing data for Error and Execution Time

In [9]:

```

def store_data(data):
    with open("data.mem", "a") as file:
        data_text = " ".join([str(i) for i in data]) + "\n"
        file.write(data_text)

def read_data():
    with open("data.mem", "r") as file:
        data = file.readline()

```