University of Glasgow

ENG4052: Digital Communication 4 (2022-23)

Lab2: Carrier Recovery using Costas Loop

Haoshi Huang (2635088H)

Submission Date

8/2/2023

1 Introduction

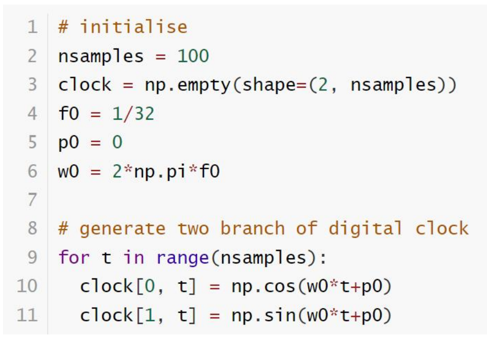
Carrier recovery is a key technique during demodulation, the aim of which is to synchronise the carrier frequency of modulated signal at the receiving end to the local oscillator frequency. Although there are a variety of factors affecting modulated signals during transmission, using carrier recovery can make sure the correctness of demodulation signal partly.

In this lab, I will use Python3.10 in Visual Studio to program a **VCO** firstly, and then implement **Carrier Recovery** with the help of **Costas Loop** using **Cordic** algorithm (Coordinated Rotation Digital Computer) during demodulation. Libraries NumPy1.23, SciPy1.9 are imported to implement advanced math operation. Library matplotlib3.6 is used to display graphics, which can show how signal and parameters changes after processing.

2 Numerically Controlled Oscillator

2.1 Create a Digital Clock

An oscillator is an electronic component to generate a fixed frequency, controlled sine or cosine wave. A digital oscillator can simulate an oscillator operation. The code of NCO is shown in Fig. 2.1 and the digital signal of 100 samples is shown in Fig. 2.2. We can adjust the parameters f0 (frequency), p0 (phase) to control the output wave. When creating wave, the f0 and p0 are constant we set initially.



```python
# initialise
nsamples = 100
clock = np.empty(shape=(2, nsamples))
f0 = 1/32
p0 = 0
w0 = 2*np.pi*f0

# generate two branch of digital clock
for t in range(nsamples):
  clock[0, t] = np.cos(w0*t+p0)
  clock[1, t] = np.sin(w0*t+p0)
```

Figure 2.1 Program NCO

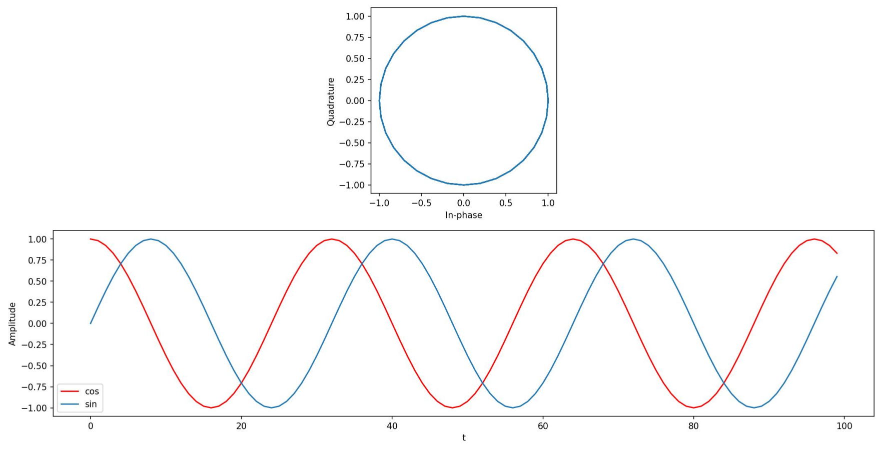


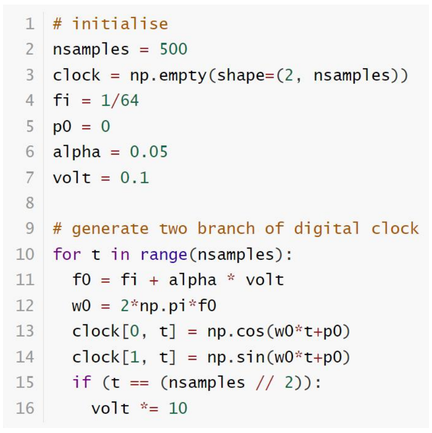
Figure 2.2 Controlled cosine & sine wave

2.2 Voltage Controlled Oscillator

On the basis of the NCO, we can add another parameter volt (voltage) to control frequency with voltage variation which varies with time. The frequency controlled by voltage can be expressed by following formular:

$$f_0 = f_i + \alpha * volt$$

fi is initial frequency. The code for VCO is shown in Fig. 2.3 and the output wave is shown in Fig. 2.4. We increase the voltage 10 times from 250 sample point, and then the wave frequency changed accordingly.



```python
# initialise
nsamples = 500
clock = np.empty(shape=(2, nsamples))
fi = 1/64
p0 = 0
alpha = 0.05
volt = 0.1

# generate two branch of digital clock
for t in range(nsamples):
  f0 = fi + alpha * volt
  w0 = 2*np.pi*f0
  clock[0, t] = np.cos(w0*t+p0)
  clock[1, t] = np.sin(w0*t+p0)
  if (t == (nsamples // 2)):
    volt *= 10
```

Figure 2.3 Program VCO

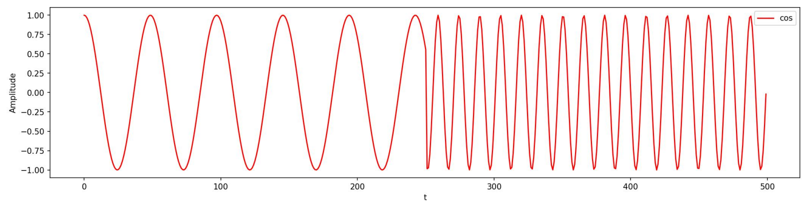


Figure 2.4 Output of VCO

2.3 Cordic Algorithm

Cordic algorithm is an algorithm which can quickly computes trigonometric functions' value, with matrix multiplication under continuous iteration. In fact, the nature of matrix multiplication is angular rotation. According to the flow diagram in the Fig. 2.5, we can program a function called cordic as shown in the Fig. 2.6:

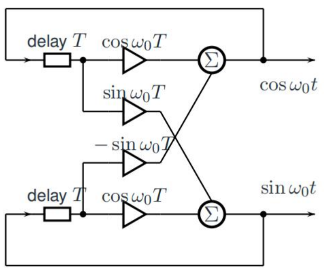
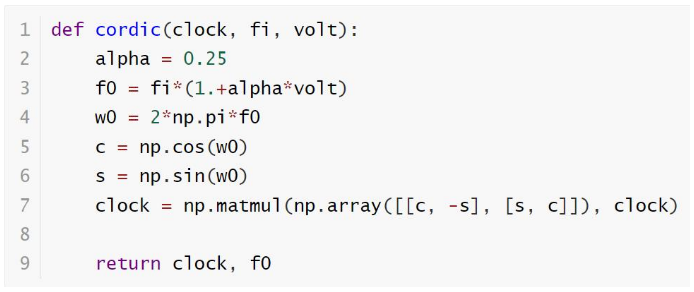


Figure 2.5 Flow diagram of Digital Oscillator with Cordic algorithm



```python
def cordic(clock, fi, volt):
    alpha = 0.25
    f0 = fi*(1.+alpha*volt)
    w0 = 2*np.pi*f0
    c = np.cos(w0)
    s = np.sin(w0)
    clock = np.matmul(np.array([[c, -s], [s, c]]), clock)

    return clock, f0
```

Figure 2.6 Cordic function

3 Carrier Recovery

3.1 Modulation

During transmission, some factors may cause distortions of the modulated signal slightly, such frequency and phase of carrier wave. We use random function to simulate distortion. In this lab, we still use the same original coded information as BPSK project shown as Fig. 3.1. With the BPSK modulation, we can get the modulated signal shown as following Fig. 3.2. The number of samples per bit is 128. And we will sample 64 per period as ideal carrier frequency, meaning 1/64.

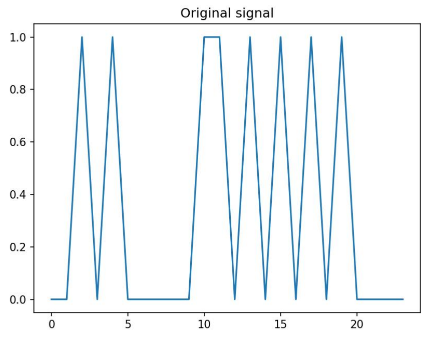


Figure 3.1 Original signal

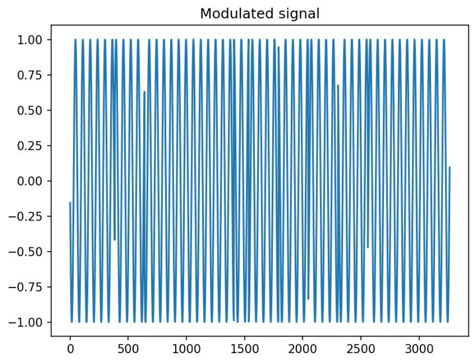


Figure 3.2 Modulated signal by BPSK

3.2 Costas Loop with Cordic Algorithm

In fact, Costas Loop can be considered as an advanced Phase-locked Loop (PLL). PLL just use one branch wave to circle; Costas Loop uses two orthogonal waves to loop as shown in the Fig.

3.3.

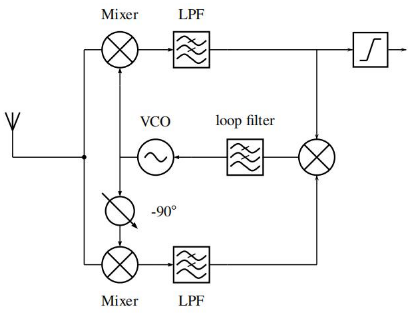


Figure 3.3 Flow diagram of Costas Loop

In the diagram, firstly we mixed I&Q components of modulation signal with VCO output respectively. Secondly, two branches mixed signal are filtered by the same low pass filters respectively. Finally, the two filtered branch signal multiply each other, which get voltage as the error to adjust the frequency of VCO output. In the Fig. 3.4 & 3.5, We can observe the adjustment process of voltage and frequency, and the frequency closes to the real carrier frequency at the end.

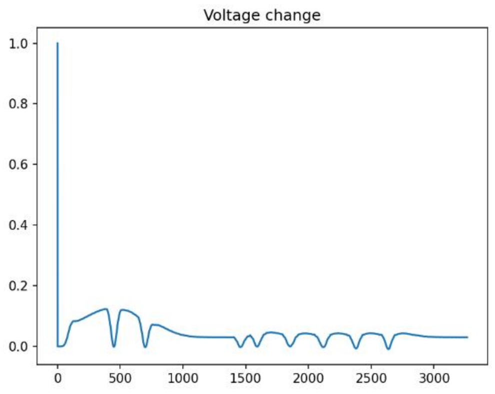


Figure 3.4 Adjustment process of voltage

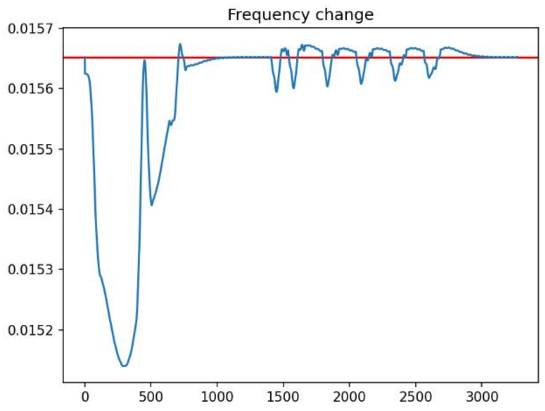


Figure 3.5 Adjustment process *of frequency and fi*

We can draw the reference wave and self-adaption wave in the same plot. In the Fig. 3.6, the red wave and the bule wave become overlapping gradually.

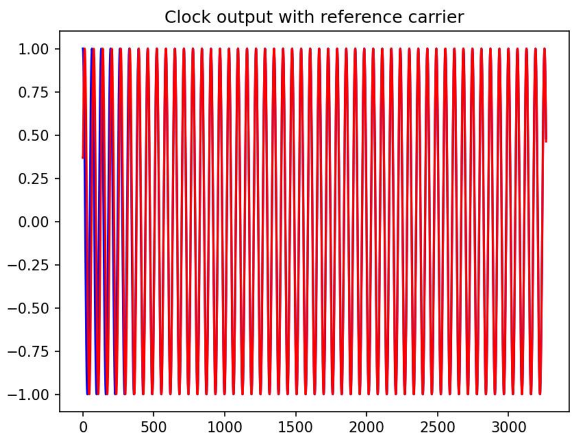


Figure 3.6 Adjustment process of frequency and fi

After carrier recovery, we can get the whole output from the filtered cos branch wave as shown in the Fig. 3.7.

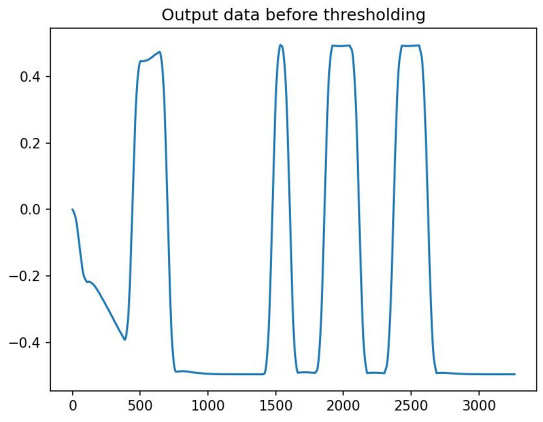


Figure 3.7 Cos branch of Costas Loop after LPF

After thresholding, we can get the demodulated signal, which is the same as the original signal as shown in the Fig. 3.8.

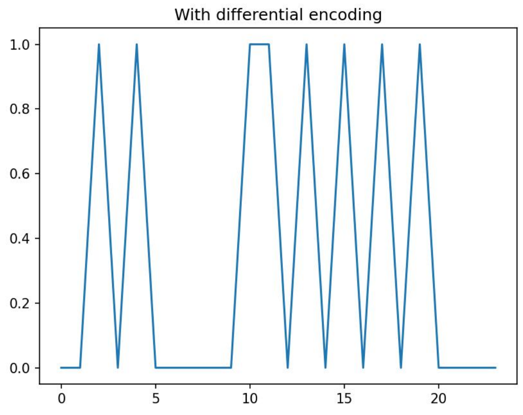


Figure 3.8 Demodulated signal with differential coding

4 Conclusion

Demodulated signal with inverse phase sometimes

During carrier recovery, the phase and frequency need to be adjusted to synchronize the modulated signal. If the phase has an great error, resulting in the inverse phase of demodulated signal after carrier recovery exactly comparing with original signal. To avoiding this situation, we need to use differential coding, meaning **XOR**, before the modulation and after the thresholding.

In the code, for comparison conveniently, we can use a on-off variable to change the partial code with or without differential coding. Setting on-off as “False” and running the code many times, in the Fig. 4.1 & 4.2, we can compare the result. The orange has inverse phase with the original coded information. And if we set on-off variable as “True”, we use differential code. The result of running code many times will only get the same signal as the original signal.

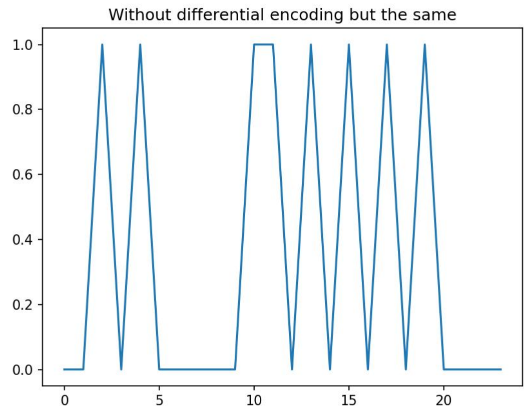


Figure 4.1 Demodulated signal with the same phase

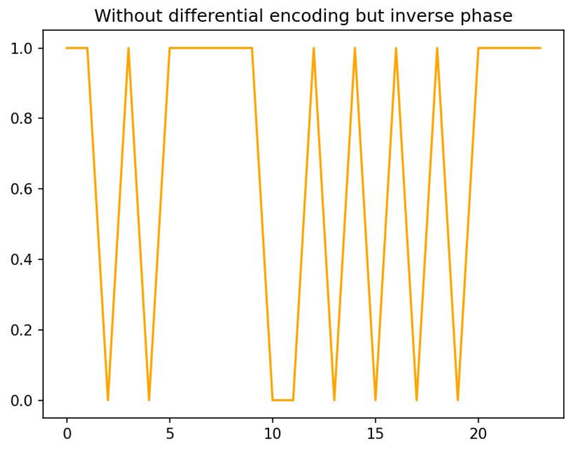


Figure 4.2 Demodulated signal with the inverse phase

Appendix:

NCO.py

```
import numpy as np
from scipy import signal
from matplotlib import pyplot as plt
# initialise
nsamples = 100
clock = np.empty(shape=(2, nsamples))
f0 = 1/32
p0 = 0
w0 = 2*np.pi*f0

# generate two branch of digital clock
for t in range(nsamples):
    clock[0, t] = np.cos(w0*t+p0)
    clock[1, t] = np.sin(w0*t+p0)

fig = plt.figure()
ax1 = fig.add_subplot(211)
ax1.set_aspect(1)
ax1.plot(clock[0], clock[1])
ax1.plot(clock[0][0], clock[1][0], color='r')
ax1.set_xlabel('In-phase')
ax1.set_ylabel('Quadrature')

ax2 = fig.add_subplot(212)
ax2.plot(clock[0], color='r')
ax2.plot(clock[1])
ax2.legend(labels = ('cos', 'sin'))
ax2.set_xlabel('t')
ax2.set_ylabel('Amplitude')
plt.show()
```

VCO.py

```
import numpy as np
from scipy import signal
from matplotlib import pyplot as plt
# initialise
nsamples = 500
clock = np.empty(shape=(2, nsamples))
fi = 1/64
p0 = 0
```

```
alpha = 0.05
volt = 0.1

# generate two branch of digital clock
for t in range(nsamples):
    f0 = fi + alpha * volt
    w0 = 2*np.pi*f0
    clock[0, t] = np.cos(w0*t+p0)
    clock[1, t] = np.sin(w0*t+p0)
    if (t == (nsamples // 2)):
        volt *= 10

fig = plt.figure()
ax = fig.add_subplot(111)
ax.plot(clock[0], color='r')
ax.legend(labels = ('cos', 'sin'))
ax.set_xlabel('t')
ax.set_ylabel('Amplitude')
plt.show()
```

CarrierRecovery.py

```
'''
Author          : Eureke
Date            : 2023-02-08 16:10:41
LastEditors     : Marcus Wong
LastEditTime : 2023-02-09 00:40:27
Description     :
'''
import numpy as np
from matplotlib import pyplot as plt
from scipy import fft
from scipy import signal
from numpy import random

def bin_array(num, m):
        # Convert a positive integer num into an m-bit bit vector
        return np.array(list(np.binary_repr(num).zfill(m))).astype(bool)

id_num = 2635088
Nbits = 24
tx_bin = bin_array(id_num, Nbits)

# Show original signal
```

```
plt.figure()
plt.title('Original signal')
plt.plot(tx_bin)
plt.show()

s = tx_bin
# carrier wave ideal frequency
fi = 1/64
# samples per bit
bit_len = 128 # 64

withDiff = True
if (withDiff):
    #Differential Coding of tx_bin
    tx_diff = np.zeros(1, dtype='bool')
    for i in range(Nbits):
        tx_diff = np.append(tx_diff, tx_diff[i]^s[i])
    Nbits = Nbits+1
else:
    tx_diff = tx_bin

# low-pass filter
numtaps = 128 #64
b1 = np.flip(signal.firwin(numtaps, 0.005))

# initialise
clock = np.array([1.0,0.0])
# carrier radom frequency
f_c =    fi*(1.+0.02*(random.rand()-0.5))
# carrier radom phase of carrier
p_c = 2*np.pi*random.rand()

# Modulation
s_mod = np.empty(0)
for t in range(0, bit_len*Nbits + numtaps//2):
    s_mod = np.append(s_mod, (2*tx_diff[(t//bit_len)%Nbits]-1)*np.cos(p_c+2*np.pi*f_c*t))

plt.figure()
plt.title('Modulated signal')
plt.plot(s_mod)
plt.show()

fout = np.array(f_c)
volt = 1.0
```

```
# volt changes
vout = np.array(volt)
# output of clock cos wave
cout = clock[0]
# output of reference clock
rout = np.cos(p_c)
# demod output
dout = np.empty(0)

def cordic(clock, fi, volt):
        alpha = 0.25
        f0 = fi*(1.+alpha*volt)
        w0 = 2*np.pi*f0
        c = np.cos(w0)
        s = np.sin(w0)
        clock = np.matmul(np.array([[c, -s], [s, c]]), clock)

        return clock, f0

mixed = np.zeros((2,numtaps))
for i in range(0, bit_len*Nbits + numtaps//2):
        # modulated signal mixed with clock
        mixed[0,:] = np.append(mixed[0,1:],clock[0]*s_mod[i])
        mixed[1,:] = np.append(mixed[1,1:],-clock[1]*s_mod[i])
        # lpf
        lpmixed = [np.sum(b1*mixed[j,:]) for j in range(2)]
        volt = lpmixed[0]*lpmixed[1]

        clock, f0 = cordic(clock, fi, volt)

        fout = np.append(fout, f0)
        vout = np.append(vout, volt)
        cout = np.append(cout, clock[0])
        rout = np.append(rout, np.cos(p_c+2*np.pi*f_c*i)) #Reference block
        dout = np.append(dout, lpmixed[0])

plt.figure()
plt.title('Voltage change')
plt.plot(vout)
plt.show()

plt.figure()
plt.title('Frequency change')
plt.axhline(f_c, color='r')
plt.plot(fout)
plt.show()

plt.figure()
plt.title('Clock output with reference carrier')
plt.plot(cout)
plt.plot(rout, color='r')
plt.show()

plt.figure()
plt.title('Output data before thresholding')
plt.plot(dout)
plt.show()

print(f_c)
print(fout[-1])
print(fout[-1] - f_c)

if (withDiff):
    # With differential coding
    rx_diff = np.empty(0)
    for i in range(Nbits):
        #select an appropriate sample point
        k = (2*i+1)*bit_len//2 +numtaps//2
        rx_diff= np.append(rx_diff, np.heaviside(dout[k],0))

    rx_bin = np.empty(0, dtype='bool')
    Nbits = Nbits-1
    for i in range(Nbits):
        rx_bin = np.append(rx_bin, rx_diff[i].astype(bool)^rx_diff[i+1].astype(bool))

    # print(rx_bin)
    plt.figure()
    plt.title('With differential encoding')
    plt.plot(rx_bin)
    plt.show()
else:
    # Without differential coding
    rx_bin = np.empty(0, dtype='bool')
    for i in range(0,Nbits):
        t = (2*i+1)*bit_len//2 +numtaps//2
        rx_bin = np.append(rx_bin, np.heaviside(dout[t],0))

    if ((rx_bin != tx_bin).any()):
```

```
    plt.figure()
    plt.title('Without differential encoding but inverse phase')
    plt.plot(rx_bin, color='orange')
    plt.show()
else:
    plt.figure()
    plt.title('Without differential encoding but the same')
    plt.plot(rx_bin)
    plt.show()
```