Code Division Multiple Access report

Radmer van der Heyde

I. Introduction

One of the challenges in communication is how to deal with multiple people transmitting on the same frequency, so that their messages each go through, but do not interfere with others. There are many solutions to this problem: have each user send signals at a specific time, divide the channel and have each user send signals on that frequency, or have each user use a different code. Each of these has different drawbacks and strengths. This report will focus on the last strategy.

In code division multiple access(CDMA), each user is assigned a code, which when applied to a signal turns it into noise to all others except for the receiver with the same code. This only works if the codes have a low correlation and works best if the codes are completely orthogonal.

II. Synchronous CDMA

A. Description

In synchronous CDMA, a signal is encoded with an orthogonal code specific to each user. These codes contain only the values -1 and 1, and are known as Walsh Codes. The figure below shows the set of size 4 Walsh codes.

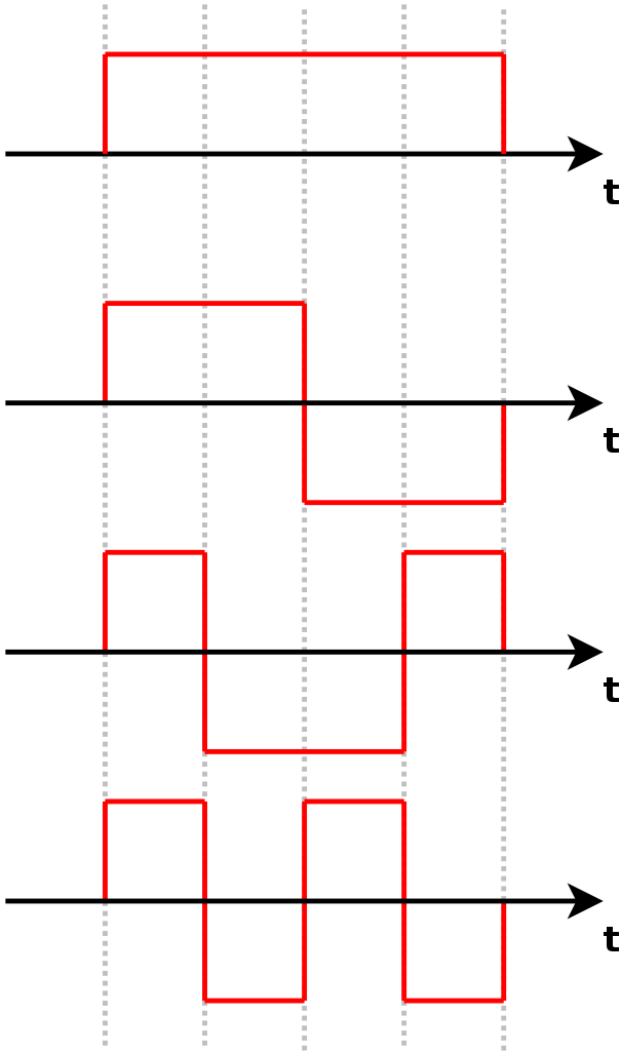

Fig. 1: Size 4 walsh codes

The signal is encoded by taking the outer product of the specific Walsh code for the users signal and the signal centered around zero(a value of 1 is 1 and 0 is -1). This produces a signal where each bit is either the positive or the negative of the

code. A carrier wave is then applied to the encoded signal and sent through the channel. The receiving channel extracts the encoded signal and decodes it by splitting the extracted signal into pieces the length of the original code and taking the dot product of the code with the each piece of the signal.

Synchronous CDMA is best used when one transmitter is sending a different signal to multiple users. On the transmitter side, each separate code can be applied to the different signals, and the sum can be sent through the channel. Each individual receiver will then receive the sum of all the different signals being transmitted, but its code will only decode the information sent with the same code.

Synchronous CDMA also spreads the signal over a wider band of frequencies. This is beneficial because if the channel has some narrow band interference majority of the signal should still be able to be received, making CDMA more practical as real channels have varying levels of interference. However, with synchronous CDMA, it is impossible to recover all signals and use the code space fully if they are sent at arbitrary starting points.

B. Demonstration

Here I will demonstrate how Synchronous CDMA works step by step and provide code that accomplishes this. The packages I use are numpy, pyaudio, matplotlib, and random. I also use the functions Siddhartan wrote for the Acoustic modem problem set for the same reasons we used them then. The simulation and the synchronous transmitter code can also be found at this ipython notebook http://nbviewer.ipython.org/github/rvanderheyde/ThinkDSP/blob/master/Simulation_of_CDMA.ipynb

1) Walsh Code Generation: The first step is to generate the orthogonal codes to encode the desired signal. Walsh codes can be obtained from the rows or columns of a Hadamard matrix. This matrix can be generated recursively from the following equation:

$$H(2^k) = \begin{bmatrix} H(2^{k-1}) & H(2^{k-1}) \\ H(2^{k-1}) & -H(2^{k-1}) \end{bmatrix}$$

The code below implements this equation.

```
def walsh(x, sign=1):
    """Generates a array of Walsh codes, x is the length of the code, sign is the sign"""
    if x == 1 and sign == 1:
        return [[1,1],[1,-1]]
    if x == 1 and sign == -1:
        return [[-1, -1],[-1, 1]]
    a = walsh(x-1,1)
    b = walsh(x-1,-1)
    if sign == 1:
        out = []
        for i in range(len(a)*2):
            if i<len(a):
                out.append(a[i]+a[i])
            else:
                out.append(a[i-len(a)]+b[i-len(a)])
        return out
    if sign == -1:
        out = []
        for i in range(len(b)*2):
            if i<len(b):
                out.append(b[i]+b[i])
            else:
                out.append(b[i-len(b)]+a[i-len(b)])
        return out
```

2) Encoding with Walsh Codes: To encode the signal, we use the equation

$$y = \sum_{m=1}^{k} x_m c_i$$

, where k is the length of the message x and c_i is the vector we are encoding the signal with. This is just the elements of each row in the matrix produced by the outer product of the two vectors in one vector. Let's say the first user is assigned the code $[1,-1]$, and the second user is assigned the code $[1,1]$. The first user wants to send over the sequence of bits $[1,0,0,1]$, and the second user wants to send over the sequence of bits $[1,0,1,0]$. The first user's signal is encoded which gives

$$s \otimes c = [1,-1] \otimes [1,-1,-1,1] = [1,-1,-1,1,-1,1,1,-1]$$

. The encoded second user's signal is

$$[1,1] \otimes [1,-1,1,-1] = [1,1,-1,-1,1,1,-1,-1]$$

. When we add these two together, we get

$$[2, 0, -2, 0, 0, 2, 0, -2]$$

.

3) Decoding with Walsh Codes: To decode the combined signal we separate the signal into chunks the size of the user's code. For our signal this is

$$[(2, 0), (-2, 0), (0, 2), (0, -2)]$$

. Each chunk is then dotted with the user's code and divided by the length of the code; for user one this produces

$$[2, -2, -2, 2] \times \frac{1}{N} = [1, -1, -1, 1]$$

. This is the same as the equation

$$x = \frac{1}{N} c_i^T y$$

. For the second user this gives us

$$[2, -2, 2, -2] \times \frac{1}{N} = [1, -1, 1, -1]$$

. These were the signals that were originally sent, demonstrating that by encoding each signal with orthogonal codes we can add them and successfully extract each one.

However this is a trivial example, so I have provided code below that uses the Walsh generation function in the section above in an example with longer codes and more users transmitting. I have also included code to play the signal over audio and recover it.

III. Asynchronous CDMA

A. Description

Asynchronous CDMA is a solution to synchronous CDMA's inability to handle signals with arbitrary starting points. This makes it a useful protocol for cellular devices to transmit to a central tower, as each mobile station has no idea when and where other devices are transmitting. In asynchronous CDMA, instead of using orthogonal codes, the codes are a randomly generated sequence of 1 and -1. A randomly generated sequence of 1s and -1s will have low correlation between another randomly generated sequence, and so the dot product should be close to zero between each randomly generated sequence. Because of this we can demonstrate that signals with different codes or sent at different times do not interfere with our ability to extract the desired signal if each signal has similar power levels, and that the steps to encode and decode the signal are similar to that in synchronous CDMA.

If we have a signal x we want to send using asynchronous CDMA, we can encode it using

$$y = \sum_{m=1}^{k} x_m c_i$$

where k is the length of the message and c_i is the random sequence we want to encode the message with, which is the same way we encoded signals in synchronous CDMA. We can also decode the signal with the same equation used in synchronous CDMA,

$$x \approx \frac{1}{N} c_i^T y$$

, where y is the incoming signal and N is the length of the code c_i. This is only approximately equal because when we substitute in the sum of all the n different messages on the channel, $\sum_{l=1}^{n} x_l c_l$ for y, the incoming signal, we get

$$x_i = \frac{1}{N} \sum_{l=1}^{n} c_i^T x_l c_l$$

. This can be rearranged to

$$\frac{1}{N} \sum_{l=1}^{n} x_l \sum_{j=1}^{N} c_{ij} c_{lj}$$

and we can split the outer sum so that

$$\frac{1}{N} \sum_{l=1 l \neq i}^{k} x_l \sum_{j=1}^{N} c_{ij} c_{lj} + \frac{1}{N} x_i \sum_{j=1}^{N} c_{ij} c_{ij}$$

. The term, $\frac{1}{N}x_i \sum_{j=1}^{N} c_{ij}c_{ij}$ is equal to x_i, so we can rewrite the equation as

$$\frac{1}{N}\sum_{l=1 l\neq i}^{k} x_l \sum_{j=1}^{N} c_{ij}c_{lj} + x_i$$

. Because for any element in any pseudo random code a 1 or a -1 is equally likely, if N is large, $\sum_{j=1}^{N} c_{ij}c_{lj}$ is close to zero so the original message,x_i, was extracted successfully.

B. Demonstration

Transmitter Ipython notebook:
http://nbviewer.ipython.org/github/rvanderheyde/ThinkDSP/blob/master/Asynchronous_CDMA_transmit.ipynb
Receiver Ipython notebook:
http://nbviewer.ipython.org/github/rvanderheyde/ThinkDSP/blob/master/CDMA%20receivers.ipynb

IV. Appendix 1: Synchronous CDMA code

Transmission code:

```
def outer1D(a, b):
    """computes the outer product in a 1D list"""
    output = []
    for i in a:
        for j in b:
            output.append(i*j)
    return output

def fakeTransmit(a,b):
    """adds the signals as if they were in the air at the same time"""
    output = []
    for i in range(len(a)):
        output.append(a[i]+b[i])
    return output

users = 3
a = walsh(2)
code1 = a[0]
code2 = a[1]
code3 = a[2]

cBits = [i*2-1 for i in bits]
cBits2 = [i*2-1 for i in bits2]
cBits3 = [i*2-1 for i in bits3]

syncSig = outer1D(cBits, code1)
syncSig2 = outer1D(cBits2, code2)
syncSig3 = outer1D(cBits3, code3)

sum1 = fakeTransmit(syncSig, syncSig2)
sum2 = fakeTransmit(sum1, syncSig3)
print cBits
print sum2

#transmit over speaker
tempSum = [[i]*250 for i in sum2]
elogSum = []
for i in tempSum:
    elogSum += i
# x = [-1.0 if i== 0.0 else i for i in elogSum]
outSum = [1.0]*250+elogSum
# print outSum
x = [i/users if abs(i)>=1.0 else i for i in outSum]
xt = np.array(x)
ts = np.arange(0, len(xt)/float(8820), 1/float(8820))
mplib.plot(ts,xt)
mplib.show()
ps = 2*np.pi*1000*ts
cs = 8000*np.cos(ps)

ms = np.multiply(xt,cs)
for i in range(250):
```

```
    np.insert(ms,0,0.0)

play_samples(ms, rate=8820, chunk_size = 1000)
```

Receiving code:

```
x = get_samples_from_mic(sample_rate = 8820, threshold = 3000, chunk_size = 1000)

i, j = find_start_and_end(x, 4000)
xt = x[i:j]
ts = np.arange(0, len(xt)/float(8820), 1/float(8820))
ps = 2*np.pi*1000*ts
cs = np.cos(ps)
ms = np.multiply(xt,cs)

fs = np.fft.fft(ms)
n = ms.size
timestep = 1.0/8820
freq = np.fft.fftfreq(n, d=timestep)

#low pass filter
for i,val in enumerate(freq):
    if val>1000:
        fs[i] = 0
    if val<-1000:
        fs[i] = 0
ys = np.fft.ifft(fs)

bitsOut = []
rectified = np.abs(ys)

#convert back to values
maximum = np.amax(ys)
for i in range(50, len(ys),250):
    if ys[50] >0:
        if ys[i] >maximum-3000:
            bitsOut.append(3.)
        elif ys[i]>2*maximum/3-200:
            bitsOut.append(2.)
        elif ys[i]>maximum/3-400:
            bitsOut.append(1.)
        elif abs(ys[i])<100:
            bitsOut.append(0.)
        elif ys[i] <-1*maximum+3000:
            bitsOut.append(-3.)
        elif ys[i] <-2*maximum/3+200:
            bitsOut.append(-2.)
        else:
            bitsOut.append(-1.)
    else:
        if ys[i] >maximum-3000:
            bitsOut.append(-3.)
        elif ys[i]>2*maximum/3-200:
            bitsOut.append(-2.)
        elif ys[i]>maximum/3-400:
            bitsOut.append(-1.)
        elif abs(ys[i])<100:
            bitsOut.append(0.)
        elif ys[i] <-1*maximum+3000:
            bitsOut.append(3.)
        elif ys[i] <-2*maximum/3+200:
            bitsOut.append(2.)
        else:
            bitsOut.append(1.)

a = walsh(2)
code1 = a[0]
code2 = a[1]
code3 = a[2]

def chunks(l, n):
    n = max(1, n)
    return [l[i:i + n] for i in range(0, len(l), n)]
# break into segements same size as codes
chunkList = chunks(bitsOut, size)
```

```
print bitsOut
out = []
out2 = []
out3 = []
for i in chunkList:
    out.append(dotProduct(i,code1))
    out2.append(dotProduct(i,code2))
    out3.append(dotProduct(i,code3))
print out
# Turn into 1 or 0
strBits = []
for i in out:
    if i >0:
        strBits.append(1)
    else:
        strBits.append(0)

strBits2 = []
for i in out2:
    if i >0:
        strBits2.append(1)
    else:
        strBits2.append(0)

strBits3 = []
for i in out3:
    if i >0:
        strBits3.append(1)
    else:
        strBits3.append(0)
# Convert back to string
npbits = np.array(strBits)
npbits2 = np.array(strBits2)
npbits3 = np.array(strBits3)
```

V. Appendix 2: Asynchronous CDMA code

Transmitter code:

```
def generateCodeAsync(N):
    """generate the code for asynchronous CDMA"""
    output = []
    for i in range(N):
        num = random.random()
        if num<.5:
            output.append(-1)
        else:
            output.append(1)
    return output

bits = string2NPArray('Hello')
user1Code = generateCodeAsync(8)

#Multiply by 2 and subtract 1 to center around 0.
cBits = [i*2-1 for i in bits]

sig = outer1D(cBits, user1Code)

tempSum = [[i]*250 for i in sig]
elogSum = []
for i in tempSum:
    elogSum += i
# x = [-1.0 if i== 0.0 else i for i in elogSum]
outSum = [1.0]*250+elogSum
# print outSum
x = [i/users if abs(i)>=1.0 else i for i in outSum]
xt = np.array(x)
ts = np.arange(0, len(xt)/float(8820), 1/float(8820))
mplib.plot(ts,xt)
mplib.show()
ps = 2*np.pi*1000*ts
cs = 8000*np.cos(ps)

ms = np.multiply(xt,cs)
```

```
for i in range(250):
    np.insert(ms,0,0.0)

play_samples(ms, rate=8820, chunk_size = 1000)
```

Receiver code:

```
xA = get_samples_from_mic(sample_rate = 8820, threshold = 3000, chunk_size = 1000)

iA, jA = find_start_and_end(xA, 4000)
xtA = xA[iA:jA]
tsA = np.arange(0, len(xtA)/float(8820), 1/float(8820))
psA = 2*np.pi*1000*tsA
csA = np.cos(psA)

msA = np.multiply(xtA,csA)

fsA = np.fft.fft(msA)
nA = msA.size
timestep = 1.0/8820
freqA = np.fft.fftfreq(nA, d=timestep)

for i,val in enumerate(freqA):
    if val>1000:
        fsA[i] = 0
    if val<-1000:
        fsA[i] = 0

ysA = np.fft.ifft(fsA)
bitsOut = []

for i in range(50, len(ysA),250):
    if ysA[50] >0:
        if ysA[i]>1500:
            bitsOut.append(1.)
        elif ysA[i]<-1500:
            bitsOut.append(-1.)
        else:
            bitsOut.append(0.)
    else:
        if ysA[i]>1500:
            bitsOut.append(-1.)
        elif ysA[i]<-1500:
            bitsOut.append(1.)
        else:
            bitsOut.append(0.)

#generate all possible pseudo noise codes
def pn_codes(N):
    """returns a list of all pseudo noise codes of size N"""
    if N == 0:
        return [[]]
    a = pn_codes(N-1)
    out = []
    for i in a:
        out.append(i+[1])
        out.append(i+[-1])
    return out

PN = pn_codes(8)
#test all of the codes on the input signal
chunkList = chunks(bitsOut, len(PN[0]))
out = {}
for i,val in enumerate(PN):
    for j in chunkList:
        if i in out:
            out[i].append(dotProduct(j,val))
        else:
            out[i] = [dotProduct(j,val)]
print len(out.keys())
out2 = {}
for k in out.keys():
    out2[k] = []
    for l in out[k]:
        if l>=0:
```

```
            out2[k].append(1.)
        else:
            out2[k].append(0.)
strings = {}
count = 0
for n in out2.keys():
    temp = np.array(out2[n])
    strings[n] = NPbits2String(temp)
    if strings[n] == 'Hello':
        print True
        count +=1
```