Bericht PPS – Bits on Air

Nachmittag 1 - (15-03-2011)

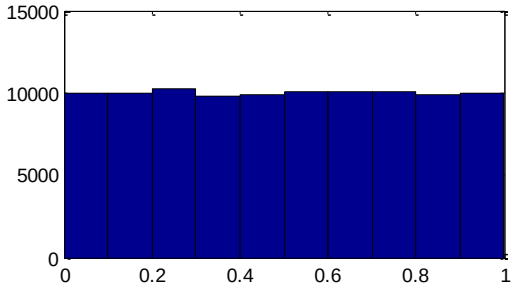
Programmierung eines einfachen Kommunikationskanal.

Test des Random-Generators

Für die folgenden Aufgaben brauchen wir einen Zufallsgenerator, nun wollen wir testen, ob dieser auch gleichverteilte Zahlen liefert, oder ob eine andere Verteilung verwendet wird.

```
z = rand(10^5,1);
hist(z)
```

Die Zahlen sind offenbar gleichverteilt.



Sender

Als erstes implementieren wir einen Sender, der eine zufällige Bitsequenz erzeugt. Die 0 und 1 Bits sind Bernoulli-Verteilt, mit Wahrscheinlichkeit p für eine 0.

```
function bitsequence = source( sequence_length, p )
%SOURCE Signal Source
%   Generates a bitsequence of sequence_length elements with a probability
%   of p for each bit to be 0
    bitsequence = ceil(rand(1,sequence_length)-p);
end
```

Drain

Nun brauchen wir eine Hilfsfunktion, die unsere Bitsequenz analysieren kann. Als erstens sind wir am Mittelwert und an der Standardabweichung interessiert.

```
function [ mu, std_dev] = drain( bitsequence )
%DRAIN Calculates mean and standard deviation of bitsequence
    mu = mean(bitsequence);
    std_dev = std(bitsequence);
end
```

Messung des Mittelwertes und der Standardabweichung

Jetzt machen wir ein paar Messungen um herauszufinden, wie der Mittelwert und die Standardabweichung unseres generierten Signals aussehen:

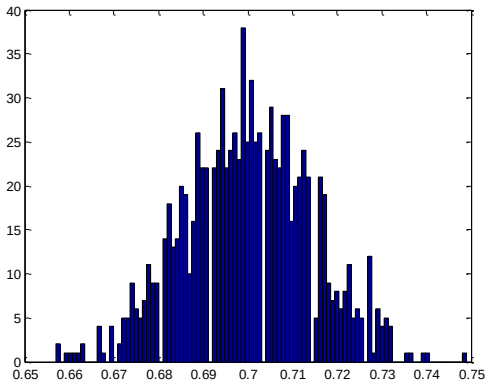
```
loopsize = 10;
N = 100;

values = zeros(loopsize,2);
for i = 1:loopsize
    [values(i,1),values(i,2)] = drain(source(N,0.3));
end
```

```
figure(1)
hist(values(:,1),100);
figure(2)
gauss(mean(values(:,1)),mean(values(:,2)));
```

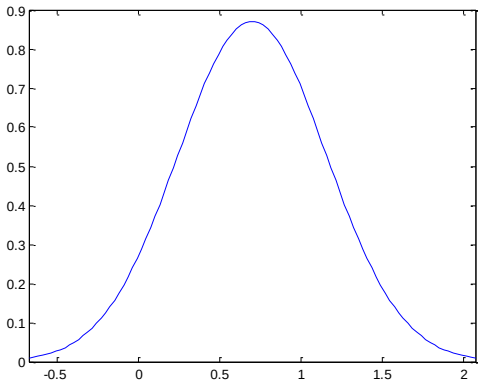
Gaussverteilung in MatLab

Unsere gemessenen Werte wollen wir mit der Gaussverteilung vergleichen:

```
function [  ] = gauss( average, std_dev )
    points = linspace(average - 3*std_dev, average + 3*std_dev,100);
    values = 1/(sqrt(2*pi).*std_dev)* exp(-(points- ...
                  average).^2/(2*std_dev^2));
    plot(points,values);
    xlim([average-3*std_dev, average+3*std_dev])
end
```



Histogramm der Mittelwerte der Messungen



Zum Vergleich: Die Gaussverteilung

Wir stellen fest, dass wir eine Gaussverteilung für den Mittelwert erhalten und dass der Mittelwert zu $1 - q$ konvergiert.

Der Übertragungskanal

Weiter brauchen wir einen fehlerhaften Übertragungskanal. In unserem Fall haben wir einen Binary-Symmetric-Channel, der mit einer symmetrischen Wahrscheinlichkeit 0 und 1 Bits vertauscht.

```
function channel_bit_sequence = BSC_channel( bit_sequence, q )
%BSC_CHANNEL Binary-Symmetric-Channel Model with errorprobability q
    noise = source(length(bit_sequence),1-q);
    channel_bit_sequence = xor(bit_sequence,noise);
end
```

Berechungen der Bit-Error-Rate(BER)

Um die Fehlerrate quantifizieren zu können, erweitern wir unsere Hilfsfunktion mit einer BER-Berechnung, die die Anzahl Fehler pro Länge zurückgibt:

```
function [ mu, std_dev, BER] = drain2( bitsequence, q)
%DRAIN2 Calculates mean and standard deviation of bitsequence and the BER
    mu = mean(bitsequence);
    std_dev = std(bitsequence);

    BER = sum(mod(bitsequence-BSC_channel(bitsequence,q),2)) …
            /length(bitsequence);
End
```

Unsere Messungen haben ergeben, dass der BER ungefähr der Fehlerwahrscheinlichkeit q des Binary-Symmetric-Channel entspricht (was zu erwarten war):

```
[ mu, std_dev, BER ]= drain2(source(N,0.3),0.3)
BER = 0.2946
```

Reptitionskode

Nun versuchen wir mit einer Repetitionscodierung (jedes Bit wird N Mal nacheinander gesendet) die Fehler zu erkennen und zu korrigieren um so den BER zu senken:

Encodierung

```
function [ bitseq_out ] = encode_repetitioncode( bitseq_in, num_reps )
%ENCODE_REPETITIONCODE Encodes the bitsequence with num_reps repetitions
    bitseq_out = repmat(bitseq_in,num_reps,1);
    bitseq_out = bitseq_out(:)';
end
```

Decodierung

```
function [ decodedSignal ] = decode_repetitioncode( input_sig, num_reps )
%DECODE_REPETITIONCODE Decode Repetitioncode with num_reps repetitions
    decodedSignal = ((mean(vec2mat(input_sig,num_reps),2))>0.5)';
end
```

Drain3

Wir erweitern wieder unsere Hilfsfunktion, so dass wir den BER mit und ohne die fehlerkorrigierende Kodierung vergleichen können.

```
function [ mu, std_dev, BER, BER_COD ] = drain3( bitsequence, q, num_rep )
%DRAIN3 Calculates mean and standard deviation of bitsequence and the BER
with and without encoding
    mu = mean(bitsequence);
    std_dev = std(bitsequence);

    signalToSend = encode_repetitioncode(bitsequence,num_rep);
    signalReceived = BSC_Channel(signalToSend,q);
    decodedSignal = decode_repetitioncode(signalReceived,num_rep);

    BER_COD = sum(mod(bitsequence-decodedSignal,2))/length(bitsequence);
    BER = sum(mod(bitsequence-BSC_channel(bitsequence,q),2))
            /length(bitsequence);
End
```

Die Codierung kann die Fehler im Übertragungskanal korrigieren, solange man in einem Zeichen nicht mehr als $\frac{N}{2}$ Fehler hat. Deshalb führt ein grösseres N zu einer besseren Fehlerkorrektur:

```
[ mu, std_dev, BER, BER_COD ]  = drain(source(N,0.3),0.3,20)

BER = 0.2800
BER_COD = 0.0380
```

Nachmittag 2 - (22-03-2011)

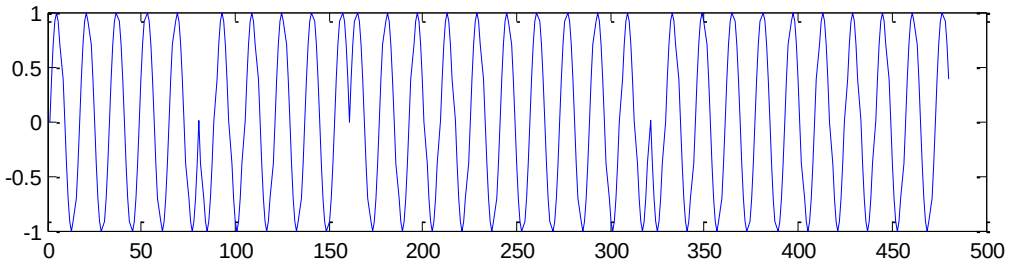
Phasenmodulation

Modulation

Wir haben ein Trägersignal erstellt und darauf ein Binärsignal mit binärer Phasenumtastung modelliert.

```
function [ y ] = amod_modulate_rect( b, tau_c, tau_s )
    t = 0:(length(b)*tau_s-1);
    x = sin(2*pi*t/tau_c);
    m = repmat(1-2*b,tau_s,1);
    m = m(:);
    y = x.*m';
end
```

```
plot(amod_modulate_rect([0 1 0 0 1 1],16,80))
```



Mit Phasenumtastung Moduliertes Trägersignal

Demodulation

Für die Demodulation des Signals haben wir es mit dem Trägersignal multipliziert und Symbolweise integriert/aufsummiert. Mit der folgenden Schwellwertfunktion lässt sich das Ursprungssignal rekonstruieren:

$$d(z_n) = \begin{cases} 1, & z_n < 0 \\ 0, & z_n \geq 0 \end{cases}$$

```
function z = amod_demodulate_rect( r, tau_c, tau_s )
    t = 0:(length(r)-1);
    x = sin(2*pi*t/tau_c);
    m = vec2mat(r.*x,tau_s);
    z = sum(m,2); z = z < 0; z = z';
end
```

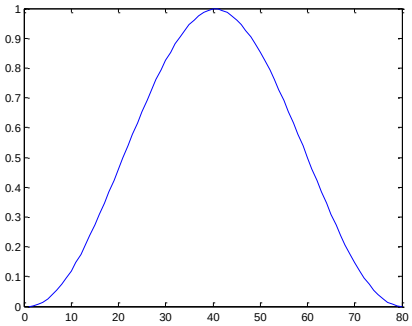
```
amod_demodulate_rect(amod_modulate_rect([0 1 0 0 1 1],16,80),16,80)

ans =    0     1     0     0     1     1
```

Rolloff

Um Unstetigkeit im Signal zu reduzieren und damit das Signal schmalbandig zu halten, wird das Symbol mit einem Rolloff an seinen Rändern gedämpft.

Rolloff

```
function [ p_beta ] = p_beta_( beta , tau_s )
    tp = 0:(tau_s*beta/2-1);
    p_beta_part = (1-cos(2*pi*tp/(beta*tau_s)))/2;
    p_beta = ones(1,tau_s);
    p_beta(1:length(p_beta_part)) = p_beta_part;
    p_beta(end:-1:(end-length(p_beta_part)+1)) = p_beta_part;
end

plot(p_beta_(1,80))
```



Rolloff Fenster

Modulate

Modulation unter Berücksichtigung des Rolloffs

```
function [ y ] = amod_modulate( b, tau_c, tau_s, beta )
    t = 0:(length(b)*tau_s-1);
    x = sin(2*pi*t/tau_c);
    m = repmat(1-2*b,tau_s,1);
    q = repmat(p_beta_(beta,tau_s),length(b),1)';

    m = m(:);
    q = q(:);
    m = m.*q;
    y = x.*m';
end
```

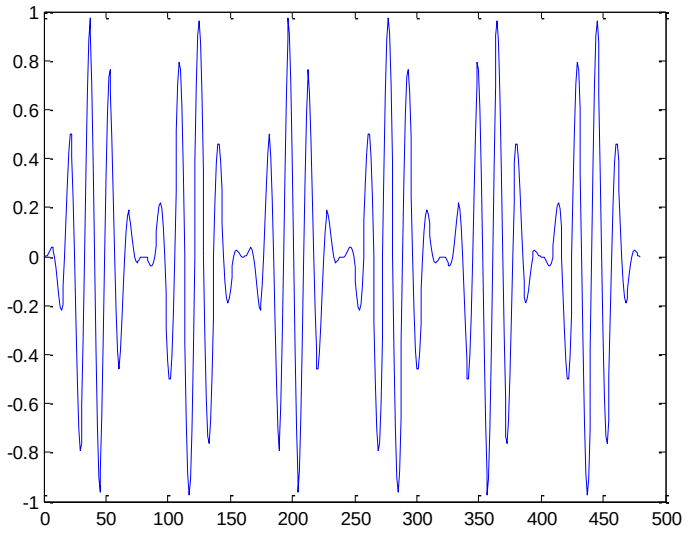
Demodulate

Demodulation unter Berücksichtigung des Rolloffs

```
function z = amod_demodulate( r, tau_c, tau_s, beta )
    t = 1:(length(r));
    x = sin(2*pi*t/tau_c);
    m = vec2mat(r.*x,tau_s);
    r = repmat(p_beta_(beta,tau_s),floor(length(r)/tau_s),1);

    m = m.*r;
    z = sum(m,2);
    z = z < 0; z = z';
end
```

Moduliertes Signal mit Rolloff

```
plot(amod_modulate([0 1 0 0 1 1],16,80,1))
```



Moduliertes Signal mit Rolloff

```
amod_demodulate(amod_modulate([0 1 0 0 1 1],16,80,1),16,80,1)

ans =     0     1     0     0     1     1
```

Nachmittag 3 - (29-03-2011)

Synchronisation

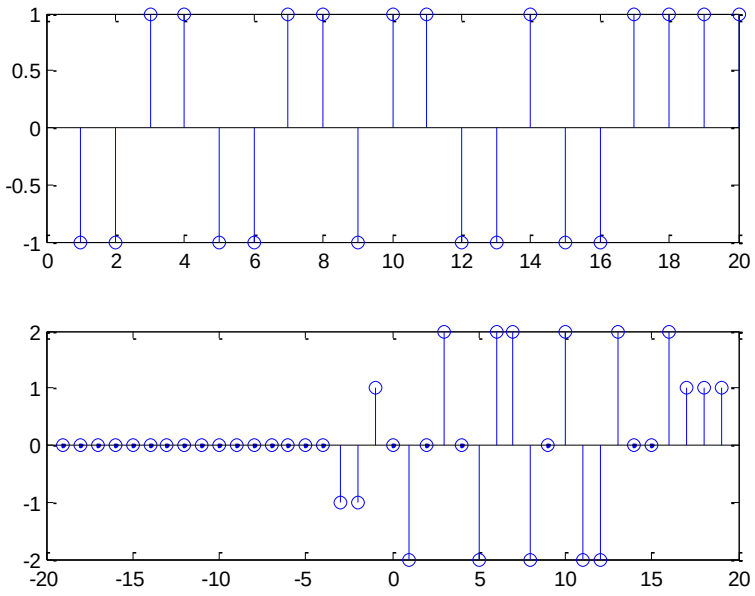
Korrelation – Einführungsbeispiel

Erzeugt Zufallssignal und sucht die Bitfolge v darin.

```
function z = xcorr_bsp( n, v )
    u = rand(1,n)>0.5;
    u = u*2 - 1;
    [c,l] = xcorr(u,v);
    figure(1)
    subplot(2,1,1)
    stem(u);
    subplot(2,1,2)
    stem(l,c)

    z = l(c==max(c));
end
```

Die Bitfolge [1 0 0 1] wurde an mehreren Stellen gefunden:

```
xcorr_bsp(20,[1 0 0 1])

ans =     3     6    10    16
```



oben Bitsequenz, unten Korrelation

Symbolsynchronisation

Zuerst machen wir die Symbolsynchronisation, indem wir im Signal nach 0-1 Übergängen suchen. Dadurch lässt sich die Verschiebung zu einem Symbolintervall bestimmen und korrigieren. Anschliessend lässt sich das Signal demodulieren und wir erhalten eine 0-1 Sequenz.

Modulator mit Symbolsynchronisation

```
function s = amod_demod_sync( r, tau_c, tau_s, beta )
    mod_sig = amod_modulate([1 0],tau_c, tau_s, beta);
    [c,l] = xcorr(r, mod_sig);
    z = l(max(abs(c))==abs(c));
    z = mod(z(1),tau_s);
    r = r(z+1:end);
    r = r(1:floor(length(r)/tau_s)*tau_s);
    s = amod_demodulate(r,tau_c,tau_s,beta);
end
```

Testen

```
amod_demod_sync([zeros(1,55) amod_modulate([0 1 0 0 1 1],16,80,1)],16,80,1)

ans =     0     1     0     0     1     1
```

Rahmensynchronisation

In einen nächsten Schritt müssen wir den Anfang unseres Signals in der Bitfolge finden. Dazu wird am Anfang des Signals eine Präambel gesendet. Diese suchen wir jetzt in der Bitfolge und finden so den Anfang der Daten. Unmittelbar nach der Präambel schicken wir noch die Länge der Daten, damit wir wissen wo die Daten zu Ende sind.

Konfiguration

Zentrale Konfiguration für die folgenden Aufgaben:

```
function [ k ] = config_default(  )
    k = struct();
    k.tau_c = 16;
    k.tau_s = 32;
    k.beta = 0.5;
    k.preamble = [1 0 1 1 0 0 0 1 1 1 1 1 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1];
    k.maxlenbase2 = 64;
end
```

Sender

Der Sender stellt das Signal zusammen: Präambel, Codierte Signallänge und die zu übertragende Bitfolge und moduliert das Ganze:

```
function [ z ] = send( r , k )
     z = amod_modulate([k.preamble de2bi(length(r),k.maxlenbase2) r],
             k.tau_c, k.tau_s, k.beta);
end
```

Empfänger

Der Empfänger demoduliert das Signal mit eingebauter Symbolsynchronisation. Anschliessend wir in der Bitfolge die Präambel gesucht und die Länge der Daten ausgelesen. Zum Schluss werden die Daten aus der Bitfolge ausgeschnitten und zurückgegeben.

```
function [ r ] = receive( z, k )
    s = amod_demod_sync(z,k.tau_c, k.tau_s, k.beta);
    [c l] = xcorr(s*2-1,k.preamble*2-1);
    [v i] = max(c);

    preamb_start = l(i);
    seq_start = preamb_start + length(k.preamble);
    r = s((seq_start+1):end);
    len = bi2de(r(1:k.maxlenbase2));
    r = r((1+k.maxlenbase2):(len+k.maxlenbase2));
end
```

Testen

Der Test in Matlab ist erfolgreich.

```
k = config_default;
receive([zeros(1,55) send([0 1 0 0 1 1],k)],k)

ans =     0     1     0     0     1     1
```

Nachmittag 4 – (05-04-2011)

Bits on Air!! – Nun wird es Zeit das Signal über die Luft zu übertragen.

Sendeprozedur

Prozedur zum Einlesen von Daten und Bereitstellen des modulierten Signals als Wavefile.

```
filetosend = '1.bmp';
fs = 48000;

fid = fopen(filetosend,'r');
bitsequence = fread(fid, 'ubit1');
fclose(fid);

k = config_default;
z = send( bitsequence' , k );

wavwrite(z, fs, 'wavout.wav');
```

Empfangsprozedur

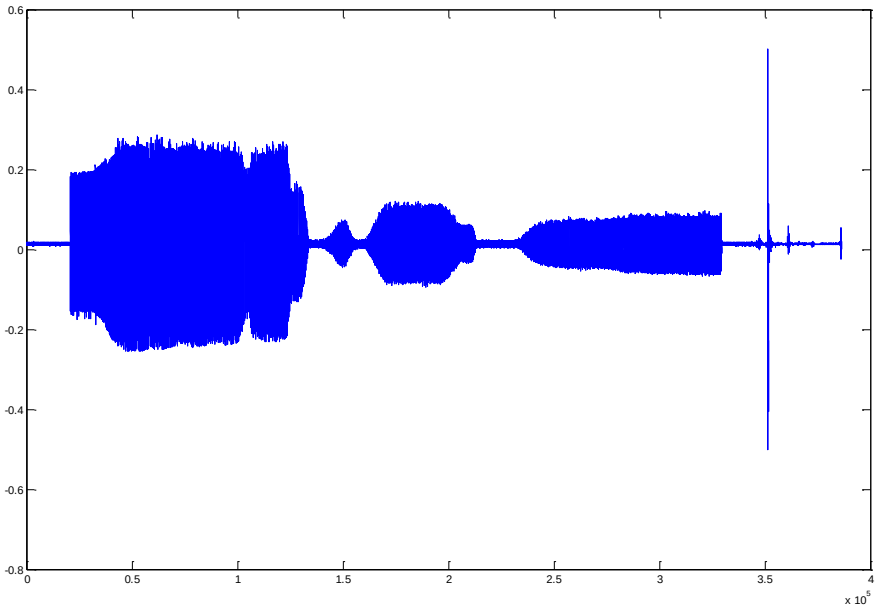
Einlesen des aufgenommenen Signals und rekonstruieren der Daten.

```
k = config_default;
filereceived = 'messageR.bmp';

[y fs] = wavread('wavin.wav');

y = -y;
bitsequenceOUT = receive(y',k);

fid = fopen(filereceived,'w');
fwrite(fid, bitsequenceOUT, 'ubit1');
fclose(fid);
```

Tests

Wir haben einige Tests gemacht. Zuerst haben wir es mit Textdateien versucht. Das hat wunderbar geklappt. Dann haben wir es mit Bitmaps versucht. Das hat auch sehr gut funktioniert. Wir hatten sogar etwas Probleme unser Signal zu stören, da die Übertragung so robust war.

In folgenden Beispiel haben wir unser Signal extrem gestört (u.a. lautstärkevariiert) und die Auswirkungen auf die übertragenen Daten gezeigt:



Gestörtes Übertragungssignal

Empfangen wurde dabei:

Man sieht aus diesem Beispiel wie extrem robust unsere Übertragung war. Obwohl wir keine Fehlerkorrigierende Kodierung verwendet haben.