

# NARI PPS BITS ON AIR – BERICHT

Céline Schönenberger – [sceline@student.ethz.ch](mailto:sceline@student.ethz.ch)

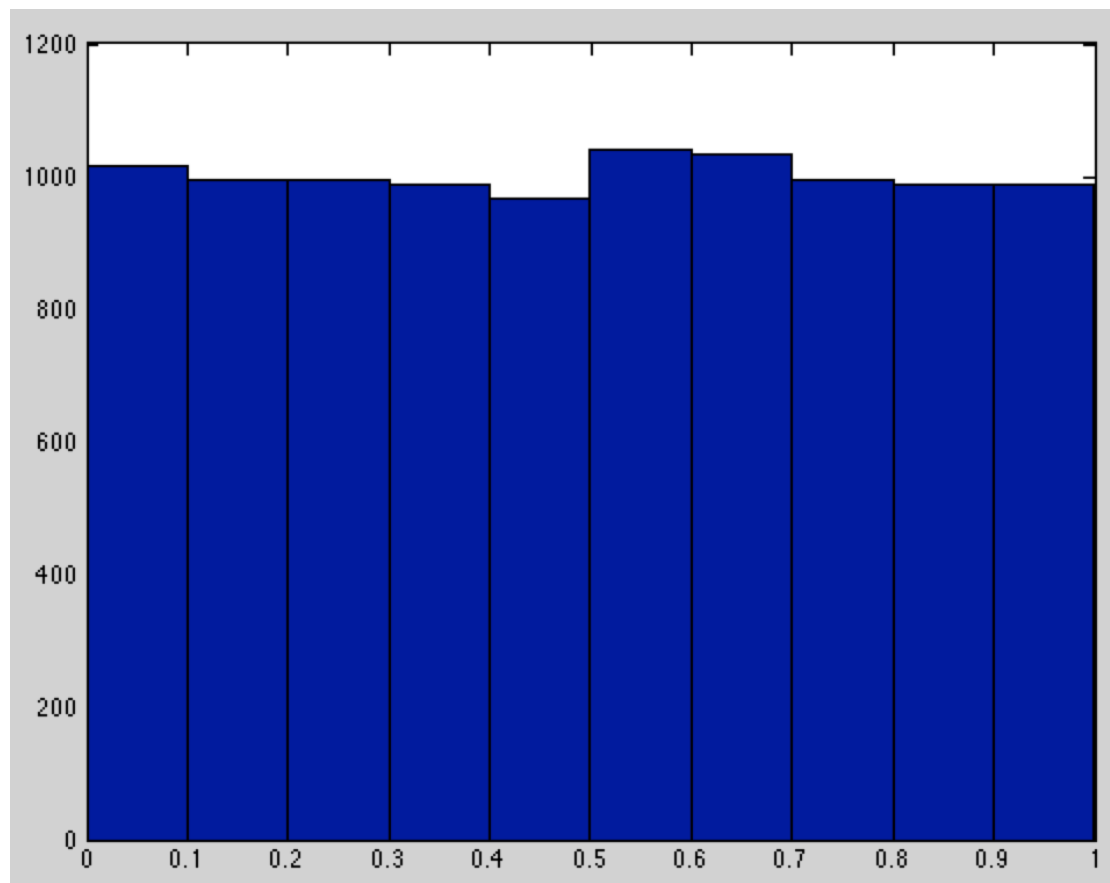
Jonathan Müller – [jo@student.ethz.ch](mailto:jo@student.ethz.ch)

## NACHMITTAG 1 – PROGRAMMIERUNG EINES EINFACHEN KOMMUNIKATIONSKANALS

### ZUFALLSGENERATOR

```
hist(rand(10000, 1))
```

Produziert folgenden Graphen:



Man kann daher davon ausgehen, dass die Zahlen gleichmässig verteilt sind.

## SENDER

Der Sender generiert eine zufällige Bitsequenz, wobei die 0 mit Wahrscheinlichkeit  $p$  auftritt.

```
function bitsequence = source( sequence_length, p )
    bitsequence = rand(1, sequence_length) > p;
end
```

## EMPFÄNGER/DRAIN

Die Funktion Drain berechnet die Standardabweichung und den Mittelwert einer Bitsequenz.

```
function [ mittelwert, sigma ] = drain( bitsequenz )
    mittelwert = mean(bitsequenz);
    sigma = std(bitsequenz);
end
```

## MITTELWERT UND STANDARDABWEICHUNG MESSEN

In einer Schleife soll jetzt `source.m` und `drain.m` mehrere male ausgeführt werden, um ein Histogramm der Standardabweichung auszugeben.

```
function [sigma_total] = loop( n, loopsize )
    p = 0.5;

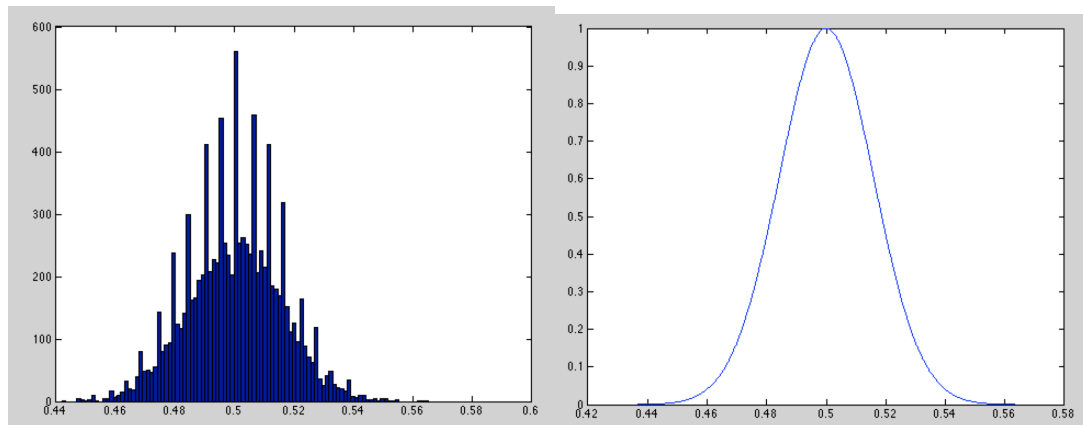
    mittelwert_v = zeros(loopsize, 1);
    sigma_v = zeros(loopsize, 1);
    for k = [1:loopsize]
        [ mittelwert, sigma ] = drain(source(n,p));
        mittelwert_v(k) = mittelwert;
        sigma_v(k) = sigma;
    end

    figure(1);
    hist(mittelwert_v, 100)
    figure(2);
    hist(sigma_v, 100)

    gauss(1-p, std(mittelwert_v))

    sigma_total = std(mittelwert_v);
end
```

Der Mittelwert konvergiert gegen  $1 - p$  für grosse Werte von  $n$ .



Histogramm des Mittelwerts, mit der Gausskurve daneben.

Unser Programm zur Erzeugung der Gausskurve:

```
function [ output_args ] = gauss( u, sigma )
    x = (-sigma * 4 + u:sigma * 8 / 1000:sigma * 4 + u)';
    y = gaussmf(x, [sigma u]);
    figure(3);
    plot(x, y)
    ylim([0 1]);
end
```

$\sigma$  und  $p$  hängen voneinander ab:

$n$ :Sequenzlänge

$q$ :Wahrscheinlichkeit für  $x_n = 0$

$\mu_n$ :Mittelwert =  $1 - q$

Anzahl Einsen:  $n(1 - q)$

Anzahl Nullen:  $nq$

$$\Rightarrow \sigma(q) = \frac{1}{n} \sqrt{n^2 q(1-q)^2 + n^2 (1-q)(1-(1-q))^2} =$$

$$\sigma(q) = \sqrt{p - p^3}$$

## ÜBERTRAGUNGSKANAL

Nun soll ein fehlerhafter Übertragungskanal simuliert werden, der mit der Wahrscheinlichkeit  $q$  Übertragungsfehler (Bit-Flips) verursacht. Wir verwenden dazu `xor`.

```
function [ channel_bit_sequence ] = BSC_channel( bit_sequence, q )
    channel_bit_sequence = xor( source(length(bit_sequence), 1-q), bit_sequence);
end
```

Die Bit-Error-Rate gibt an, wie viele Bits sich im Vergleich zum Ursprungssignal verändert haben.

```
function [ ber ] = calcBER(bit_sequence, channel_bit_sequence)
    ber = mean(xor(bit_sequence, channel_bit_sequence))
end
```

## FEHLERKORREKTUR MIT REPETITIONSCODE

Mit einem einfachen Code sollen Übertragungsfehler korrigiert werden können. Dazu wird jedes bit einfach  $n$  mal wiederholt.

### Encodierung:

```
function [ bitsequence_encoded ] = repencode(bitsequence, n )
    bitsequence_encoded = repmat(bitsequence, n, 1);
    bitsequence_encoded = bitsequence_encoded(:)';
end
```

### Decodierung:

```
function [ bitsequence_decoded ] = repdecode( bitsequence, n )
    bitsequence_decoded =
        reshape(bitsequence', n, length(bitsequence) / n);
    bitsequence_decoded = mean(bitsequence_decoded, 1) > 0.5;
end
```

Wenn die Fehlerrate  $q$  kleiner ist, funktioniert der Code natürlich besser, da es weniger Fehler zu korrigieren gibt. Wenn per Zufall mehr wie die Hälfte der  $n$  Bits einer Wiederholung geflipt werden, kommt das Bit am Ende falsch heraus.

## NACHMITTAG 2 – MODULATION UND DEMODULATION

### MODULATION

$p(t - n\tau_s)$  ist die Sprungfunktion, die vor dem Zeitpunkt  $n\tau_s$  immer 0 und danach immer 1 ist.

Die gezeigte Formel gibt die modulierte Version einer Bitsequenz wieder, d.h. jeweils Abschnitte von  $\cos(2\pi f_n)$  mit den beiden Übertragungsfrequenzen  $f_1, f_2$ .

Bei unsere Implementierung werden die Parameter nicht aus einem .mat-File geladen, sondern direkt als Parameter der Funktion übergeben.

```
function y = modulate( seq, ts, t0, t1 )
    seq1 = repencode(seq, ts);
    f0 = repmat(cos([0:ts - 1] / t0 * 2 * pi), 1, length(seq));
    f1 = repmat(cos([0:ts - 1] / t1 * 2 * pi), 1, length(seq));

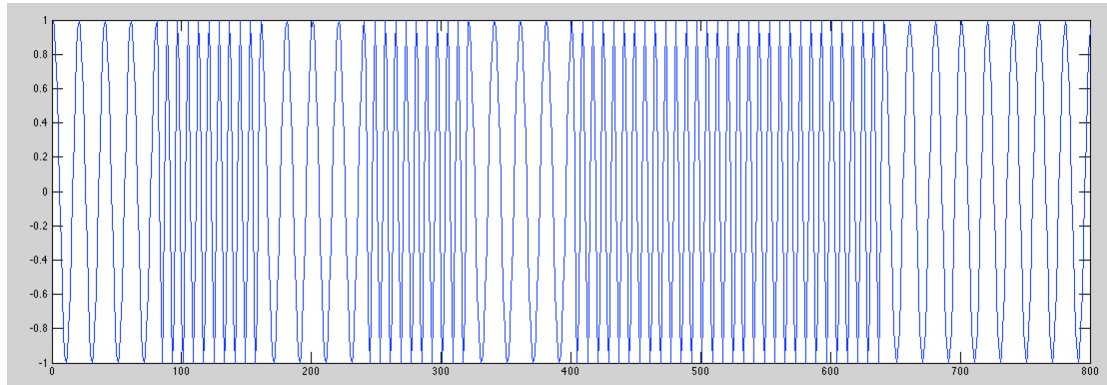
    y = (seq1 .* f1) + (~seq1 .* f0);

    plot(y)
end
```

Unser Programm `test_modulate.m`:

```
modulate([0 1 0 1 0 1 1 0 0 ], 80, 20, 8)
```

Dies erzeugt folgenden Plot:



## DEMODULATION

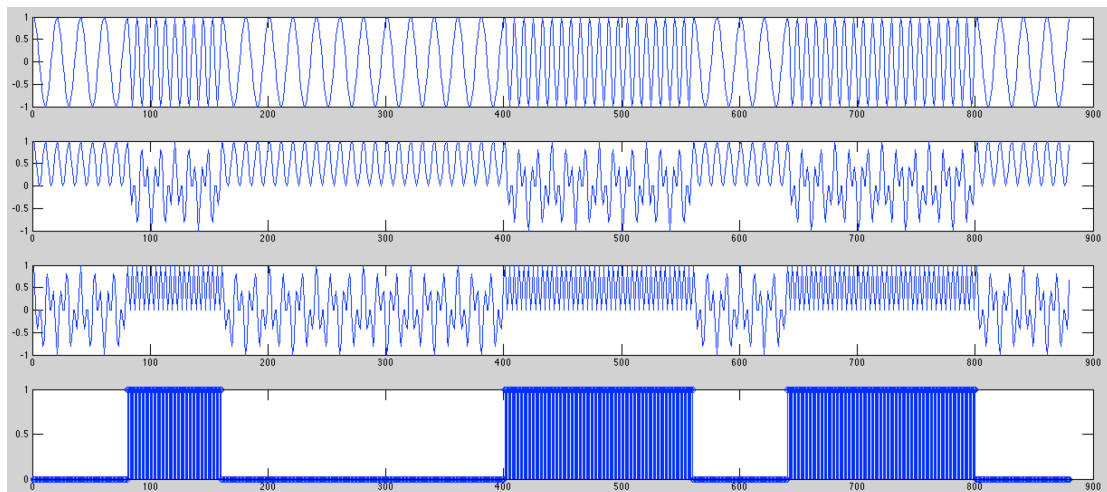
Aus den beiden korrelierten Sequenzen soll die originale Bitsequenz wiederhergestellt werden. Das empfangene Signal wird aufgeteilt in Abschnitte, deren Länge jeweils  $t_s$  entsprechen. Jeder Abschnitt wird separat mit den Trägerfrequenzen  $f_1$  und  $f_2$  korreliert. Um herauszufinden, ob nun eine 0 oder eine 1 empfangen wurde, wird das Integral des Betrags der beiden Korrelationen verglichen (Grösser als), diskret betrachtet also `sum(abs(abschnitt))`. Diese Schreibweise findet sich jedoch nicht direkt so in unserem Code, da wir mit `reshape` und Matrizen arbeiten.

```
function seq = demodulate( y, ts, t0, t1 )
    x0 = repmat(cos([0:ts - 1] / t0 * 2 * pi), 1, length(y) / ts);
    x1 = repmat(cos([0:ts - 1] / t1 * 2 * pi), 1, length(y) / ts);

    plot(x0 .* y)

    seq0 = sum(reshape(x0 .* y, ts, length(y) / ts));
    seq1 = sum(reshape(x1 .* y, ts, length(y) / ts));

    seq = abs(seq0) < abs(seq1);
end
```



Plots für das Decoding der Bitsequenz [ 0 1 0 0 0 1 1 0 1 1 0 ]:

1. Moduliertes Signal
2. Korrelation mit  $f_1$
3. Korrelation mit  $f_2$
4. Berechnete Bitsequenz

**Durchsatz erhöhen:** Dies kann man über mehrere Wege erreichen:

- Die beiden Trägerfrequenzen erhöhen, und gleichzeitig die Zeitabschnitte  $t_s$  verkleinern
- Nur  $t_s$  verkleinern
- Ein besserer Fehlerkorrekturalgorithmus wie „Repencode“

### ADDITIVE WHITE GAUSSIAN NOISE CHANNEL (AWGN)

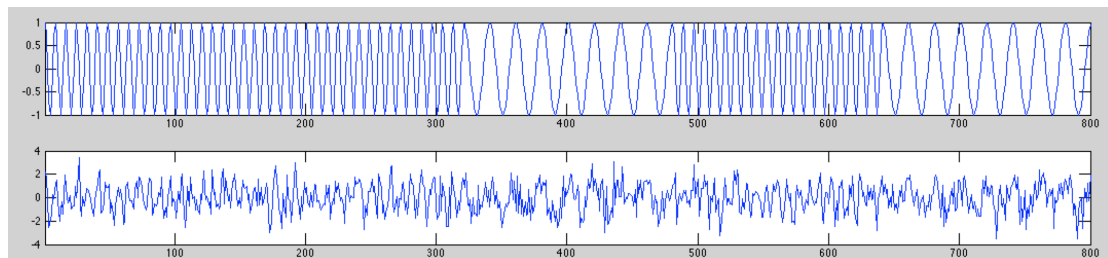
Um die Modulation und Demodulation zu testen, wird bei der Simulation des Übertragungskanals nun White Noise auf das Signal addiert. Unser Code dazu:

```
clear
repenc = 10

seq = source(1000, 0.5);
enc = repencode(seq, repenc);
y = modulate(enc, 80, 20, 8);
r = awgn(y, 0)

dec = repdecode(demodulate(r, 80, 20, 8), repenc);

calcBER(seq, dec)
```



- Oben: Originalsignal
- Unten: Signal mit Noise

Wir haben einen Repetitionscode mit  $n = 10$  verwendet, und eine Signallänge von 10000 Bits.

SNR [dB]	0	-10	-20	-25	-30	-50	-100
BER	0	0.0043	0.3527	0.4584	0.4937	0.5053	0.4923

Ab ca.  $SNR = -30dB$  beginnt die BER sich immer um 0.5 zu bewegen.

### NACHMITTAG 3 – SYNCHRONISATION

#### KREUZKORRELATION VON HAND

Wir korrelieren den Vektor  $v_1 = [-1 \ -1 \ 1 \ 1]$  mit dem Vektor  $v_2 = [1 \ -1]$ .

	-1	-1	1	-1	1	Summe
$n = 0$	-1	1	0	0	0	0
$n = 1$	0	-1	-1	0	0	-2
$n = 2$	0	0	1	1	0	2
$n = 3$	0	0	0	-1	-1	-2
$n = 4$	0	0	0	0	1	1
$n = 5$	0	0	0	0	0	0

Diese Korrelation zeigt uns, dass der Vektor  $v_2$  mit dem Offset  $n = 2$  im Vektor  $v_1$  vorkommt.

## SYNCHRONISATION

Dies ist unsere allgemeine Sync-Funktion, die den Offset eines Signals  $b$  im Signal  $a$  angibt. Dabei sollte  $b$  kürzer wie  $a$  sein.

```
function offset = sync( a, b )
    [c, lag] = xcorr(a, b);
    [m, i] = max(c);

    offset = lag(i) + 1;
end
```

Es werden anstatt Werte  $\{0, 1\}$  die Werte  $\{1, -1\}$  zur Repräsentation verwendet. Weil das Signal zur Kreuzkorrelation mit Nullen aufgefüllt wird, würde sich sonst das Signal nicht mehr vom Null-Signal abheben, und dieses könnte somit einen störenden Einfluss haben.

Wenn die Sequenz  $b$  mehrere Male in  $a$  vorkommt, wird das letzte Vorkommen davon zurückgegeben. Dies liegt daran, dass  $\max()$  bei mehreren Maxima den Index des letzten Maximums zurückgibt.

Die Vereinfachung  $\text{length}(a) \gg \text{length}(b)$  ist nicht nötig,  $\text{length}(a) \geq \text{length}(b)$  genügt.

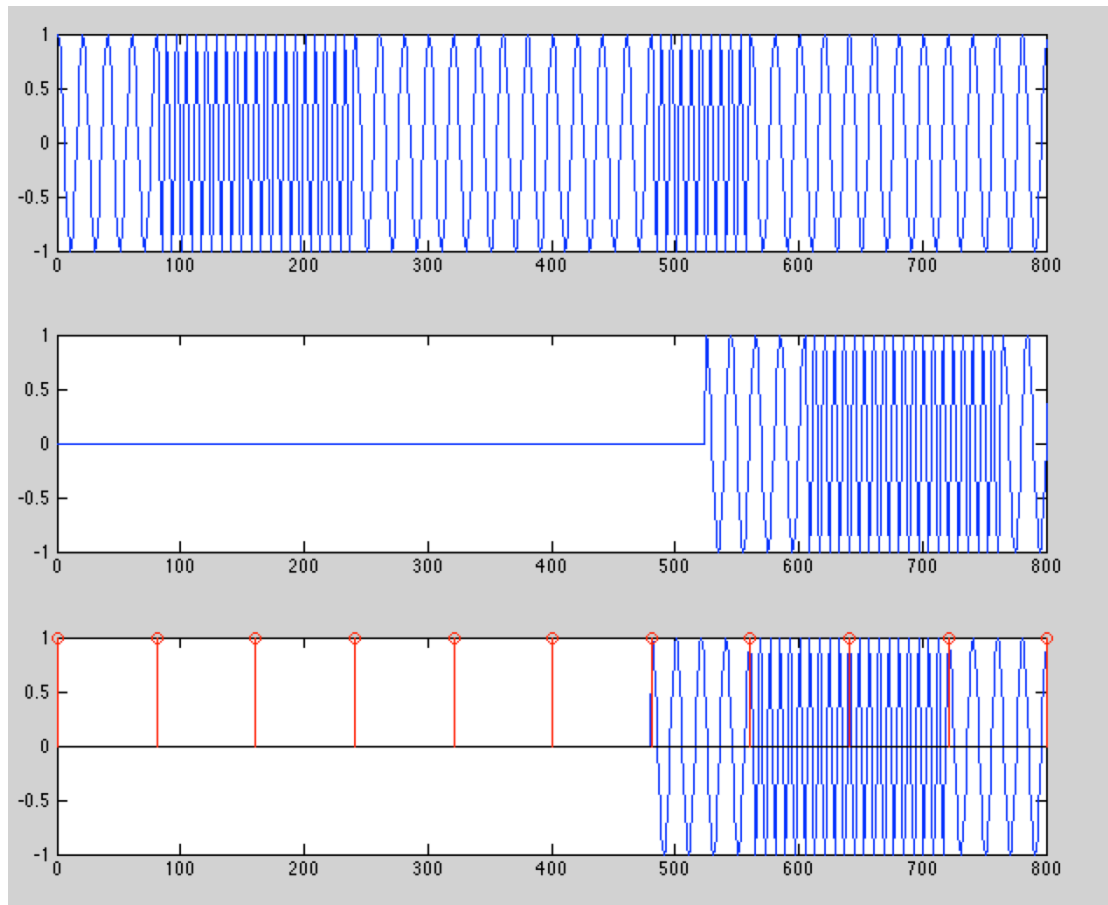
## SYMBOLSYNCHRONISATION

Bei der Symbolsynchronisation geht es nun darum, den Beginn der Zeitabschnitte der Länge  $t_s$  zu synchronisieren. Dafür wird zuerst ein  $[0\ 1]$ -Übergang gesucht, und anschliessend die nötige Anzahl Nullen vorne angehängt.

```
function [ output ] = symbolsync( a, ts, t0, t1 )
    b = modulate([0 1], ts, t0, t1);

    offset = sync(a, b);

    output = a(mod(offset, ts):length(a));
    output = [output, zeros(1, (ts - mod(length(output), ts)))];
end
```



In diesem Plot wird illustriert, wie Symbolsync arbeitet.

1. Das Originalsignal
2. Das Signal, mit einer zufälligen Anzahl Nullen vorangesetzt. Dies simuliert die Desynchronisation bei der Übertragung
3. Das synchronisierte Signal (Ende abgeschnitten auf dem Plot), mit den eingezeichneten Zeitschlitten (rot)

## RAHMENSYNCHRONISATION

Um den Beginn und das Ende einer Übertragung zu finden, muss nun eine Rahmensynchronisation durchgeführt werden. Dabei wird bei jeder gesendeten Nachricht eine Prä- und Postabel angehängt, die auch dem Empfänger bekannt ist, welcher dann danach sucht.



Unsere Prä- und Postambel sind jeweils 50 Bit lang, und wurden nach völlig unwissenschaftlichen Kriterien zufällig ausgewählt:

```
function [pre, post] = preamble( )
    pre = [
        0 0 1 0 1 1 0 1 0 1
        1 0 0 1 1 0 1 1 0 0
        1 0 1 0 1 1 1 1 0 0
        1 0 1 0 1 0 1 0 1 1
        1 0 1 0 0 1 1 1 0 0
    ];
    post = [
        1 0 1 1 0 0 1 1 0 0
        1 1 1 0 0 1 1 0 1 1
        0 0 1 1 1 1 0 0 1 1
        0 0 1 1 0 0 0 1 0 1
        0 1 1 0 1 0 0 1 1 0
    ];
end
```

Anhängen ans Signal:

```
function signal = addkeys( source )
    [pre, post] = preamble();
    signal = [pre, source, post];
end
```

Die `framesync` Funktion sucht nun nach der Prä- und Postambel und schneidet das Signal entsprechend zu.

```
function output = framesync( signal )
    [pre, post] = preamble();
    pre = pre * 2 - 1;
    post = post * 2 - 1;

    signal_ = signal * 2 - 1;

    offset = sync(signal_, pre);
    offset_end = sync(signal_, post);

    output = signal(offset+length(pre):offset_end - 1);
end
```

Unser Test-Code für die Synchronisation:

```
repenc = 2;
signal = source(10, 0.5);

seq = addkeys(signal);
enc = repencode(seq, repenc);
y = modulate(enc, 80, 20, 8);
shifted = awgn([zeros(1, 100 + floor(rand(1, 1) * 1000)), y, zeros(1, floor(rand(1, 1) * 1000))], 10);
synced = symbolsync(shifted, 80 * repenc, 20, 8);

demodulated = demodulate(synced, 80, 20, 8);
fsynced = framesync(repdecode(demodulated, repenc));
```

Und ein Beispieldurchlauf:

```
>> test_framesync  
  
signal =  
  
    1    0    1    0    0    1    1    0    1    1  
  
fsynced =  
  
    1    0    1    0    0    1    1    0    1    1
```

## NACHMITTAG 4 – SENDEN UND EMPFANGEN

### AUDIOPLAYER- UND REKORDER

```
function [] = playsound( snd )  
    samplerate = 22044;  
    seconds = length(snd) / samplerate;  
  
    ap = audioplayer(snd', samplerate, 16, -1);  
    play(ap);  
    pause(seconds);  
    stop(ap);  
end
```

```
function snd = recordsound( seconds )  
    ar = audiorecorder(22044, 16, 1, -1);  
    record(ar);  
    pause(seconds);  
    stop(ar);  
  
    snd = getaudiodata(ar);  
    snd = snd';  
end
```

**SEND**

```
function modulated = send( data, repenc )
    seq = addkeys(data);
    enc = repencode(seq, repenc);

    modulated = modulate(enc, 30, 10, 3);
end
```

Unser Test-Programm `test_send.m`:

```
data = source(50, 0.5);
repenc = 2;

tosend = send(data, repenc);
tosend = [zeros(1, 800) tosend zeros(1, 8000)];

samplerate = 22044;

ap = audioplayer(tosend', samplerate, 16, -1);
playblocking(ap);
pause(2)
```

**RECEIVE**

```
function data = receive( modulated, repenc )
    synced = symbolsync(modulated, 30 * repenc, 10, 3);

    demodulated = demodulate(synced, 30, 10, 3);

    decoded = repdecode(demodulated, repenc);

    data = framesync(decoded);
end
```

`test_receive.m`:

```
seconds = 10

ar = audiorecorder(samplerate, 16, 1, -1);
record(ar);
pause(seconds)
stop(ar);

recv = getaudiodata(ar);
recv = recv';

dataout = receive(recv, repenc);

playsound(recv)
calcBER(data, dataout);
```

**KOMBINATION VON SEND UND RECEIVE**

Wir sind nun dazu in der Lage, eine Nachricht in ein Audiosignal zu verwandeln, dieses abzuspielen und gleichzeitig über ein Mikrofon aufzunehmen, und es dann zu decodieren.

Testprogramm zur Übermittlung einer zufälligen 50Bit-Sequenz:

```
data = source(50, 0.5);
repenc = 2;

tosend = send(data, repenc);
tosend = [zeros(1, 800) tosend zeros(1, 8000)];

samplerate = 22044;
seconds = length(tosend) / samplerate;

ar = audiorecorder(samplerate, 16, 1, -1);
record(ar);
pause(2)
ap = audioplayer(tosend', samplerate, 16, -1);
playblocking(ap);
pause(2)
stop(ar);

recv = getaudiodata(ar);
recv = recv';

length(tosend)
length(recv)

dataout = receive(recv, repenc);

calcBER(data, dataout);
```

Die Übertragung funktioniert so sehr zuverlässig. Wir haben auch die Übertragung über mehrere Meter versucht, was auch funktioniert hat.

Störungen wie Klatschen oder ins Mikrofon blasen bleiben meist erfolglos, ausser die Prä- oder Postambel werden gestört.

Was auffällt ist, wie langsam die Übertragung stattfindet.

## ÜBERTRAGUNG VON BILDERN

Um Übertragungsfehler zu visualisieren, haben wir die Übertragung von monochromen (Schwarz/Weiss) Bitmaps versucht. Entsprechender Code:

```
fid = fopen('image.bmp', 'r');
data = fread(fid, 'ubit1');
fclose(fid)

repenc = 1;

tosend = send(data, repenc);
tosend = [zeros(1, 800) tosend zeros(1, 8000)];

samplerate = 22044;
seconds = length(tosend) / samplerate;

ar = audiorecorder(samplerate, 16, 1, -1);
record(ar);
pause(2)
ap = audioplayer(tosend', samplerate, 16, -1);
playblocking(ap);
pause(2)
stop(ar);

recv = getaudiodata(ar);
recv = recv';

dataout = receive(recv, repenc);




fid = fopen('image-recv.bmp', 'w');
fwrite(fid, dataout, 'ubit1');
fclose(fid);
```

Ein Monochrom-Bitmap mit 50x50 Pixeln ist 463 Bytes gross, d.h. 3.7kbit.

Unser original-Bild, erzeugt mit MS Paint:



Am besten lassen sich visualisierbare Störungen erzeugen, wenn man `repencode` auf 1 setzt. Jedoch wird dann manchmal auch die Präambel oder der Bitmap-Header falsch übertragen, was zum vollständigen Verlust des Signals führt.

		
<p>Beinahe fehlerfreie Übertragung</p>	<p>Viele Fehler</p>	<p>Vermutlich kaputter Header</p>