2 Frame Synchronization

In the first lab, the received signal samples corresponded exactly to the transmitted symbols, i.e. the beginning and the end of the data was inherently given. In reality, we receive a signal continuously, and we have to detect the beginning of the data frames within the received signal. This operation is called *Frame Synchronization*.

A common way to achieve frame synchronization is to prepend the data blocks with a special signal that is known to the receiver, called a *preamble*. The receiver can then search for the preamble using a correlation filter. If the preamble is long enough and has a random-like structure, it is very unlikely that the data signal (or the received noise) has a similar shape as the preamble, and therefore a peak in the correlator output indicates with high probability the beginning of a data frame.

2.1 The Need for a Random-Like Structure

In the introduction above, we mentioned that the preamble should have a random-like structure. Why is this the case? Could we not instead use, e.g., a series of ones as the known data? After all, this would simplify the correlator, since all multiplications could be omitted and the correlator could simply be implemented as a summation over N_p consecutive received symbols, where N_p is the length of the preamble.

First of all, it is likely that such a data sequence also occurs in the payload data. Consider for instance a black and white bitmap file, where a white pixel is stored as the symbol 0, and a black pixel as 1. Since there are many consecutive white or black pixels, the correlator would not be able to differentiate between the preamble and the actual data. By using a random-like data sequence as preamble, we can circumvent this problem.

A second problem is illustrated in Figure 2.1. It shows the autocorrelation functions for two sequences, both of length $N_p = 20$. In the left figure, the sequence consists of all ones, and in the right figure, each symbol is randomly either $+1$ or -1. We see that in the left figure, the correlator output rises while the sequence is shifted into the correlator. The peak at the center, where the sequence is completely inside the correlator, is not very distinct. In the right figure, however, we have a very sharp peak at the center. This happens because, due to the random-like structure of the sequence, the terms in the sum that gives us the correlation tend to cancel each other out. Now imagine that the graphs in Figure 2.1 are superimposed by noise. It is easily seen that in the first case, it will be difficult to determine the exact peak location, while in the latter case, it will still be possible to identify the peak, as long as the SNR is not too low.

It should be clear now that a random-like data sequence should be used as preamble. Why random-*like*? Obviously, the receiver needs to know the sequence in order to look for it, so

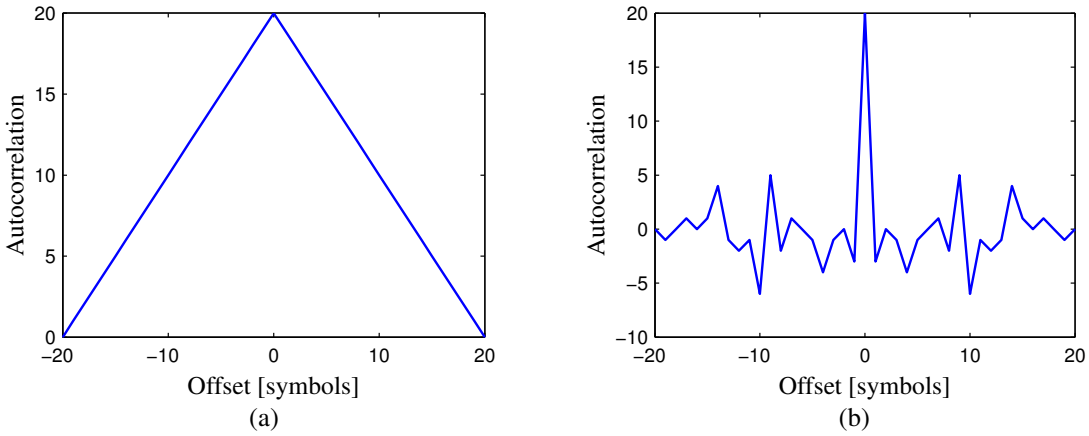

Figure 2.1: Autocorrelation of the all-ones sequence (a) and a random sequence (b) of length $N_p = 20$.

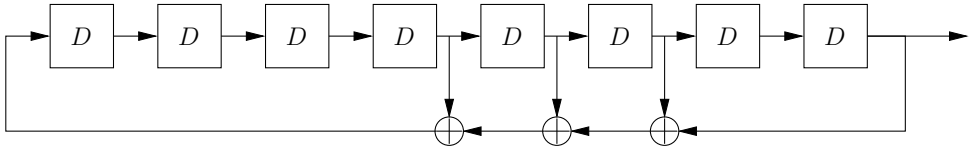

Figure 2.2: LFSR used for creating the frame synchronization sequence

the transmitter cannot use really random data. Hence we need a sequence which has statistical properties similar to random noise, but which is still reproducible in the receiver. One possibility would be to create a random sequence once, and then store it in the transmitter and receiver. But this would be very memory consuming if the sequence is long.

Another possibility is to use *pseudo-noise* (PN) sequences. These are sequences with random-like statistical properties, but which can still be reproduced deterministically. A common way to generate PN sequences is by means of a *linear feedback shift register* (LFSR).The output of the LFSR is periodic and it can be shown that the statistical properties of the LFSR output closely resemble those of a really random sequence generated by flipping a fair coin N_p times. In order to reproduce the sequence, the receiver only needs to the position of the LFSR feedback taps, the initial state of the LFSR, and the length of the sequence.

In the lab, we use the LFSR depicted in Figure 2.2, initialized with all ones. The length of the preamble is $N_p = 100$. The bits of the preamble are modulated using *binary phase shift keying* (BPSK), where the bit 0 is mapped to the symbol $+1$, and the bit 1 is mapped to the symbol -1.

2.2 The Detection Algorithm

Our goal is to detect the presence of the known preamble sequence $p[i]$, $i = 0, 1, \ldots, N_p - 1$ in the received signal $r[n] = a[n] + w[n]$. The main idea, as already mentioned, is to feed the received signal through a correlator, which correlates the received signal with the known preamble. As soon as the portion of the received signal that is inside the correlator closely

resembles the preamble, the magnitude[1] of the correlator output exhibits a distinct peak.

How do we detect this peak? We could directly take the magnitude of the correlator output and compare it to a certain threshold value. However, the problem is that the absolute strength of the correlator output depends on the unknown SNR value. In order to use a fixed threshold for peak detection, the correlator output must be normalized with respect to the received signal power.

The correlator output at discrete time n can be written as

$$c[n] = \sum_{i=0}^{N_p-1} \tilde{p}^*[i] r[n-i], \tag{2.1}$$

where $\tilde{p}[i] = p[N_p - 1 - i]$ is the reverted preamble.[2] We decide on the presence of a peak if the following condition is fulfilled:

$$\frac{|c[n]|^2}{\sum_{i=0}^{N_p-1} |r[n-i]|^2} > \gamma. \tag{2.2}$$

The denominator in (2.2) is the energy of the received signal that is currently inside the correlation filter, and γ is the decision threshold.

2.3 Using the Profiler in MATLAB

In the previous assignment, you used the commands `tic` and `toc` to measure the overall execution time of your code. For a more detailed analysis, you can use the MATLAB profiler, which gives you a detailed report including the execution of each *line* that was executed and how often it was called. Thep rofiler can be accessed by using the following commands:

- `profile on` starts the profiler, clearing previously recorded profiling statistics.
- `profile off` stops the profiler.
- `profile viewer` stops the profiler and displays the results in the profiler window.
- `profsave` saves the results in HTML format. The HTML files are stored in a subfolder of the current folder named `profile_results`.

2.4 Your Tasks

1. Implement the LFSR shown in Figure 2.2 as a MATLAB function and generate the preamble sequence $p[i]$.
2. On the lecture homepage you can download the `task2.mat` file which contains a signal where the start of the transmission occurs at an unknown time step (see Figure 2.3).

   i.) Load the file and add Gaussian noise for a given SNR value.

[1] Remember that the received signal is complex-valued, and hence the correlator output is also complex.

[2] The complex conjugation of $\tilde{p}[i]$ is in our case not necessary because the preamble only consists of ± 1 anyway.

Figure 2.3: Signal with unknown number of time steps where no transmission occurs, followed by the preamble (BPSK) and the image data (QPSK).

ii.) Implement the correlator as a MATLAB function and plot the normalized correlator output given in (2.2) for several SNR values ranging fromm -5 dB to 10 dB. You should notice a distinct peak within the signal.

iii.) Determine a reasonable peak detection threshold γ. If the threshold is too high, it can happen that we miss the presence of the preamble (this kind of error is called a *detection miss*). If, on the other hand, the threshold is too low, it may happen that we wrongly decide on the presence of a peak (this kind of error is called a *false alarm*). Explain the reasoning behind your choice of detection threshold γ (note that the threshold that your detector uses must be independent of the SNR).

3. On the lecture homepage you can download two MATLAB functions:

   - `demapper.m`: converts the QPSK symbols to bits.
   - `image_decoder.m`: draws the demapped image (you already used this function in the 1st assignment).

   Implement a receiver using the above two functions, along with the preamble generation (Task 1) and frame synchronizer (Task 2). Your receiver should run the frame synchronization algorithm on the (noiseless) signal contained in `task2.mat` until the preamble is detected. Then, the transmitted image should be received and displayed.

   Since the signal is noiseless, if your choice of peak detection threshold γ is correct, the image should always be displayed perfectly. In order to highlight the importance of frame synchronization, pick a threshold that will result in false alarms (a very *low* threshold will do this). What do you observe when frame synchronization fails?

4. Use the MATLAB profiler on your code. Which part of your code takes the longest time to run? Include the HTML file with the report in your submitted solution.