1 Mapping and Demapping

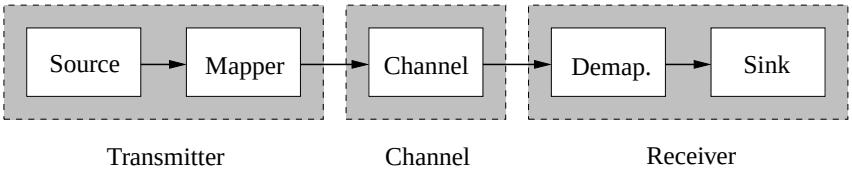

Figure 1.1: Universal System model for Communication Systems

We start our **guided project** with a review of the very basic communication system model shown in Fig. 1.1. In the transmitter, a source emits a bit stream (representing the information) which is mapped onto a set of symbols. The symbols are then fed into a channel. The output of the channel (which does not necessarily have to be from the same set of symbols) is then demapped back to bits. The set of symbols and the channel model are chosen to replicate (or at least approximate) the physical behavior of the used channel.

1.1 The BSC channel

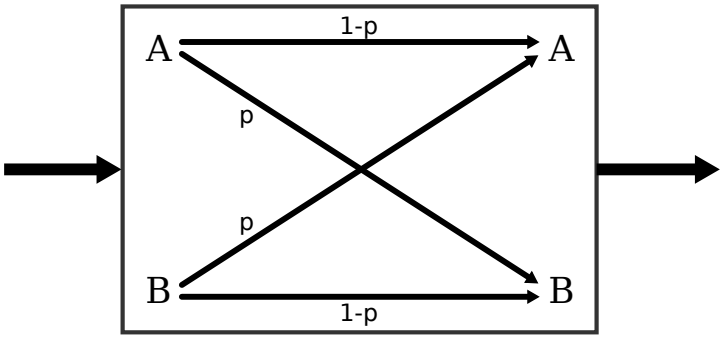

Figure 1.2: The binary symmetric channel (BSC) model

The binary symmetric channel (BSC) shown in Fig. 1.2 is one of the easiest to understand channel models in communication theory. The BSC has a discrete binary (and identical) set of input and output symbols. With probability p the channel causes a flip of the transmitted symbol value, otherwise the symbol is transmitted unchanged. Despite its very simplistic nature there are real communication systems (e.g. photons in quantum communication) which can be modeled by it.

Based on student submissions from previous years we have assembled a MATLAB implementation of a BSC simulation. It generates a random bitstream, applies the BSC and calculates the

resulting bit error rate (BER).

Listing 1.1: bscpoor.m - Bad BSC Example Code

```
% Start time measurement
tic();

% Source: Generate random bits
txbits = randi([0 1],1000000,1);

% Mapping: Bits to symbols
tx = {};
for i=1:1000000
    if txbits(i) == 0
        tx{i} = 'A';
    else
        tx{i} = 'B';
    end
end

% Channel: Apply BSC
rx = {};
i = 1000000;
while i >= 1
    randval = rand(1);

    if randval < p
        switch tx{i}
            case 'A'
               rx{i} = 'B';
            case 'B'
               rx{i} = 'A';
        end
    else
        rx{i} = tx{i};
    end
    i = i - 1;
end

% Demapping: Symbols to bits
rxbits = [];
for i=1:1000000
    if rx{i} == 'A'
        rxbits(i) = 0;
    else
        rxbits(i) = 1;
    end
end
```

```
45
46 % BER: Count errors
47 errors = 0;
48 for i=1:1000000
49     if rxbits(i) ~= txbits(i)
50         errors = errors + 1;
51     end
52 end
53
54 % Output result
55 disp(['BER: ' num2str(errors/1000000*100) '%'])
56
57 % Stop time measurement
58 toc()
```

Please note that this is exceptionally **POOR** code and combines many of the mistakes most commonly done in MATLAB programming. You shpuld never hand in such code. Nevertheless the script is valid MATLAB code and you can use it as a reference for the language syntax.

1.2 Your Tasks - Part I

1. Download the `bscpoor.m` MATLAB script from the course moodle and measure how long it takes to run. The code already contains the tic and toc functions for measuring the execution time. Reduce the number of bits to 1000 and measure the execution time again.

2. Implement a better version of the BSC simulation which is smaller and faster. It should **NOT** contain any loops, cell-arrays, duplicated constants or if statements. Compare the improvement in speed for 1000000 and for 1000 bits. You should be at least 20x faster for the first case.

1.2.1 Some General MATLAB Advice

MATLAB (MATrix LABoratory) is a very powerful tool for numeric evaluation. It will be our main tool during the course. While the program is very flexible, it is also very slow for general programming tasks. In order to MATLAB offers a lot of highly optimized built-in routines for vector and matrix operations. Writing your MATLAB code in the right way can save you a lot of time and trouble. The second important goal of good MATLAB code is readability. In many cases simpler MATLAB-aware code also leads to faster execution speed. Unfortunately this is not always the case and sometimes you have to find a trade-off between speed and readability.

We have assembled some general rules which can help you to achieve good code:

- Use vector or even matrix operations instead of slow loops.
- Write **short** comments in your code in order to document its behaviour. But keep in mind that code which is well structured and easy to read is even the better documentation.
- Use variables for parameters in order to keep the code reconfigurable.

- Give all your variables meaningful names. It can help to stick to a unified naming pattern.
- Use indention and newlines to keep your code readable. MATLAB offers a built-in `smart indent` function which provides good results.
- Optimize only performance critical parts, for all other parts readability is more important.

1.3 The AWGN channel

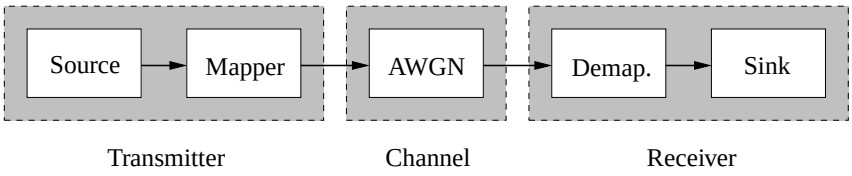

Figure 1.3: System model for Lab 1

We now replace the BSC by another very common but a bit more sophisticated channel model. In the additive white gaussian noise (AWGN) channel the symbols are complex numbers which represent the amplitude and phase of the transmitted signal and carry the digital data. In Lab 3, we will have a closer look into the synthesis of the transmitted signal from the data symbols. After transmission over a channel which disturbs the symbols with additive noise, the demapper converts the received noisy symbols back to a bit stream.

In the following, the transmitter and channel blocks are explained in more detail.

- Source: Throughout the lab, we use grayscale images as payload data. The pixels are transmitted rowwise, starting with the upper left pixel. The brightness of each pixel is represented by an unsigned 8-bit value, and the *least significant bit* (LSB) is transmitted first.
- Mapper: The modulation scheme used by our system is *quadrature phase shift keying* (QPSK), which means that the bits are pairwise mapped onto symbols $a = (\pm 1 \pm j)/\sqrt{2}$, as shown in Figure 1.4. With this modulation scheme, the information lies solely in the phase, since all symbols have the same amplitude $|a| = 1$, hence the name *Phase Shift Keying*.
- AWGN channel: In the channel, the transmitted symbols $a[n]$ are disturbed by complex-valued *additive white Gaussian noise* (AWGN) $w[n]$ with zero mean and variance σ_w^2. (A complex-valued Gaussian distributed random variable w is obtained by taking two independent real-valued Gaussian random variables with half the variance, $w_1, w_2 \sim \mathcal{N}(0, \sigma_w^2/2)$[1], and combining them to a complex number $w = w_1 + jw_2$.) "White" in this context means that the noise samples are uncorrelated, i.e. $E\{w[m]w^*[n]\} = \sigma_w^2 \delta[m-n]$.

[1]The notation $x \sim \mathcal{N}(\mu, \sigma^2)$ means that the random variable x is Gaussian distributed with mean μ and variance σ^2, i.e. the *probability density function* (PDF) is $p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$.

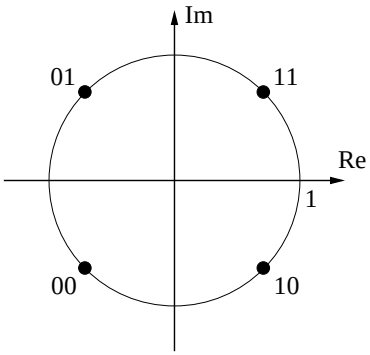

Figure 1.4: Mapping of bit pairs $b_0 b_1$ to QPSK symbols

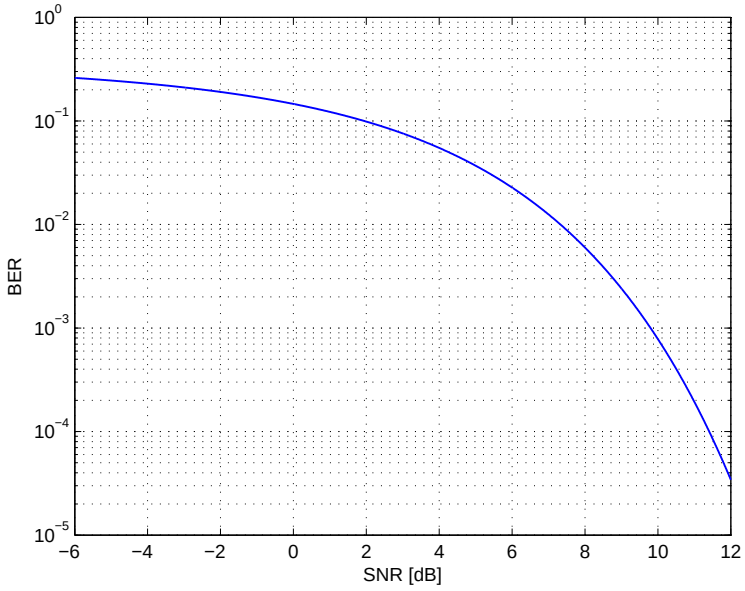

Figure 1.5: BER for uncoded QPSK modulation, AWGN channel

The received symbols are then given as

$$r[n] = a[n] + w[n]. \tag{1.1}$$

An important measure in communication systems is the ratio of average signal power to average noise power (*signal-to-noise ratio*, SNR), usually expressed in decibel:

$$\mathrm{SNR}\,[\mathrm{dB}] = 10\,\log_{10}\left(\frac{E\{|a[n]|^2\}}{E\{|w[n]|^2\}}\right). \tag{1.2}$$

Since the power of the data symbols is normalized to 1 ($|a[n]|^2 \equiv 1$), the SNR of our system is $1/\sigma_w^2$. The best achievable *bit error rate* (BER) for an uncoded QPSK modulation is plotted in Figure 1.5 as a function of the SNR.

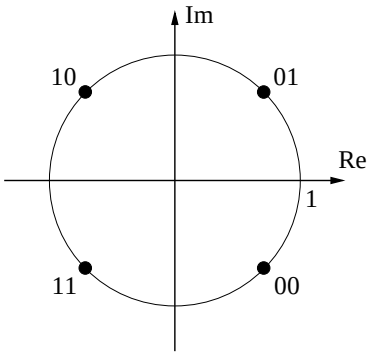

Figure 1.6: Alternative Mapping of bit pairs $b_0 b_1$ to QPSK symbols

1.4 Your Tasks - Part II

1. You are given a transmitted signal, as well as the width and height of the transmitted image. The signal can be loaded into the Matlab workspace with the command `load <filename>`.

   Your task is to generate a noisy received signal by adding, dependent on a SNR value in dB, correctly scaled noise to the loaded signal.[2] In a second step demap the received symbols into bits and display the received image with the provided function `image_decoder`.

   Display the images using different SNR values, so that you get an impression of how badly the pictures are disturbed under different channel conditions.

2. Adapt your code in order to plot the received symbols as clouds in the complex plane. Compare plots of the noisy constellations of the received signal for different channel conditions.

3. Now write a function that experimentally reproduces the BER plot from Figure 1.5. Create a random bit sequence, map it onto symbols according to Figure 1.4, and add scaled noise. Demap the noisy symbols to bits and determine the bit error rate. Repeat these steps for different SNR values and then create a BER plot with the function `semilogy` (in order to get a logarithmic scale on the y-axis).

4. Finally, create another BER plot where you use the symbol mapping as shown in Figure 1.6. Compare the two plots. Which mapping scheme is better, and why?

5. On the moodle you can download a MATLAB function `compressed_decoder` which is an alternative image_decoder based on lossy image compression. Unfortunately the script was programmed by a lazy PhD student and is therefor full of errors. Your mission is to hunt down all bugs.
   Hint: Don't try to understand the functionality of the code but use the debugger to track development and shape of the variables to identify odd behaviour.

[2] In later chapters, the signals that you get will already contain the impairments introduced by the channel, e.g., a timing or phase error. However, you will always have to add the noise yourself, so that you can simulate your system at arbitrary SNR values.

1.4.1 Plotting in MATLAB

Guidelines for plotting graphs:

- **Plot the BER in a logarithmic scale (`semilogy`).**
- Make sure that the ranges of the axis are meaningful.
- If you have multiple curves in one plot, assign different colors and or markers to it.
- Always label the axis (`xlabel/ylabel`).
- Assure that each plot has a meaningful legend. (`legend`)
- Use the MATLAB function for export of pictures (`print`) NOT a screenshot.