Module 8: Design of a Simple Communication System Part 2

© 2019 Christoph Studer (studer@cornell.edu); Version 0.1

This module will teach you how to model a simple communication channel and how to demodulate the digital AM signal generated in the previous module. Note that we are using test channels and we are not performing any transmissions over-the-air yet. ***Remember: Please ask us in case you have questions!***

9.5 Transmission Over a Simple Test Channel

Communication system engineers always test their transmission schemes using test channels that model some of the most common effects occurring in a real system. Transmission over a real, physical channel is often significantly more difficult and is done at later stages during the design. We will pursue the same approach. We will now explain how to design and use a simple MATLAB test channel that does neither use the loudspeakers nor the microphone. We will transmit our digital AM signal through that channel and look at the output of this channel. Simulated test channels have the main advantage that one has full control over all distortions, whereas in real channels, the distortions are determined by the scenario (location of transmitter and receiver, objects in the area, humidity, imperfections in the hardware, etc.).

One of the most common communication channel used for simulations is the so-called *additive white Gaussian noise* (AWGN, for short) channel. In words, AWGN channels add noise to the transmitted waveform (basically random changes to the amplitude of the transmit waveform). The amount of noise (relative to the power of the information signal) determines the quality of transmission. Additive noise appears in real-world communication systems and is mainly caused by (i) analog circuits and (ii) interference from other communication systems that use the same or nearby parts of the spectrum.

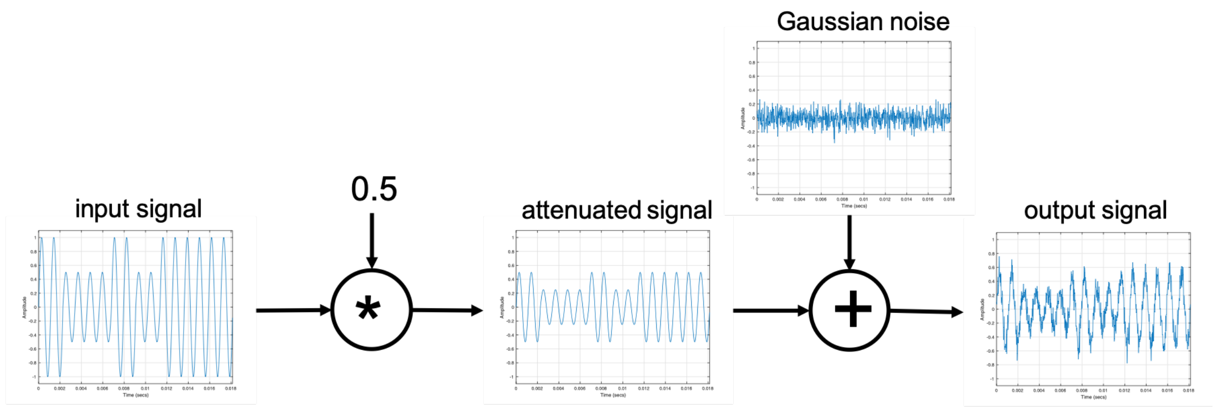


Figure 22: Illustration of an additive white Gaussian noise (AWGN) channel. Left: the signal is attenuated by a factor 0.5; middle: random Gaussian noise is added; right: output signal of the channel.

Figure 22 illustrates the key components of an AWGN system. The input signal (the amplitude-modulated waveform) is first attenuated. In this example, we attenuate the signal by a factor of 0.5 to model the fact that a microphone would not record a full-amplitude signal—the transmit signal is radiated from the loudspeaker in all directions and the microphone would only pick up a small portion of the transmitted signal. Then, we add Gaussian noise to the signal. Note that Gaussian noise is a randomly generated with specific statistics (hence, the terms *white* and *Gaussian* in AWGN). If you are not familiar with Gaussian noise and you would like to know more about it, feel free to ask us. However, the details are not important for this project. In this example, we picked the noise level (also known as the noise power) so that the received signal is clearly distorted but the main components of the transmitted waveform are still visible. The result of adding noise (hence, the term *additive* in AWGN) is the distorted output signal.

Activity 32: Listen to the output of an AWGN channel

To gain intuition on AWGN channels, load a waveform from the `examples` folder (for example, load the female or male speech signal) and pass it through the function we provide called `channel_AWGN`. The function has one input and one output argument. If your waveform is stored in the vector `y`, then use the following command:

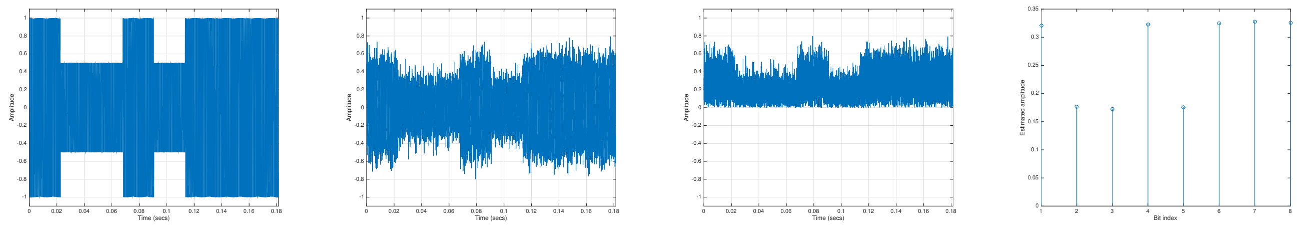
```
y_noise = channel_AWGN(y);
```

which creates a new vector `y_noise` that contains the output of the AWGN channel. Listen to the audio file and explain what you are hearing. Can you still understand the speech signal? Can you imagine that transmission errors may occur after a digital communication signal passes through such a channel? Try to listen to different audio signals.

Important: **Noise** ***is the main source of errors in digital communication systems and practical communication receivers have to be designed to mitigate such effects.***

9.6 Building a Receiver for Digital AM Signals

Now, take the *digital* AM signal you generated in the last module and pass it through the AWGN channel. You should see that the output looks similar as the one on the left side of Figure 23. Our goal is now to recover the digital input signal from the noisy output with as few errors as possible. The principle of a digital receiver is very similar to that of demodulating the analog AM signal (the one we designed in the previous module). First, we take the absolute value of the received signal. Second, we compute the envelope of the signal. Third, we have to build a *detector* that decides whether the envelope belonged to a digital one or a digital zero. We will now describe how to implement these three tasks in MATLAB.



(a) Transmit signal. (b) Received signal. (c) Rectified signal. (d) Estimated amplitudes.

Figure 23: Illustration of the main steps of a digital AM receiver. Digital AM signal at the transmitter (a); noisy received signal (b); rectified signal (c); and estimated amplitudes (d).

As in the previous activity, assume that the received digital communication signal from the AWGN channel is stored in the vector `y_noise`. The left side of Figure 23 shows the transmit signal and the second figure on the left shows the received noisy signal. Our goal is now to extract the envelope of this signal (i.e., the magnitude levels; remember that we encoded the digital information in two distinct magnitudes of the sine wave signal). To this end, we first rectify the noisy receive signal using

```
y_rect = abs(y_noise);
```

where the MATLAB command `abs` computes the absolute value of each entry of the vector. The third signal on the left in Figure 23 shows the output of the rectifier—one can already see the bit sequence $[1, 0, 0, 1, 0, 1, 1, 1]$ that is encoded in the amplitudes—you see portions of the signal with large amplitude and portions with smaller amplitude.

The second task is to extract the magnitude information. Remember that each sine wave was of length `tx_length=1000`. We also know that we transmitted eight bits. To estimate the magnitude of the first sine wave that encoded the first bit, we can compute

```
mean(y_rect(1:tx_length))
```

which calculates the average (or mean) of the rectified signal for the first `tx_length=1000` samples. The result of this averaging procedure is the mean amplitude for the duration of one sine wave whose amplitude was modulated depending on whether the transmitted bit was one or zero. We now wish to repeat the same procedure for the remaining seven bits that we transmitted. Here, we can use `for`-loops to implement this repeating task. One solution would be to execute the following commands:

```
q = [];
for kk=1:length(bits)
    indices = (1+(kk-1)*tx_len:kk*tx_len);
    q(kk) = mean(yr_rect(indices));
end
```

This loop is repeated for as many times as number of bits stored in the vector `bits` (this should be eight times for the example we are using). For each bit, we first compute the range of indices over which we want to compute an average. The range of indices is obtained by the command `(1+(kk-1)*tx_len:kk*tx_len)`, which creates an index vector. If, for example, `kk=1`, then the index vector contains the indices from 1 to 1000 which contains all samples that belong to the first modulated bit; if, for example, `kk=2`, then the index vector contains the indices from 1001 to 2000 which contains all samples that belong to the second modulation bit, etc. Finally, the command `q(kk) = mean(yr_rect(indices));` extracts the indices contained in the vector `indices` from the rectified receive vector `yr_rect` and computes the average amplitude of the result. The average amplitude is then stored in the `kk`th entry of the vector `q`. After executing the `for` loop, the vector `q` contains the estimated magnitudes associated with each of the transmitted sine waves. The right most figure in Figure 23 shows the estimated amplitude. We can clearly see that bit 1 mapped to higher magnitudes that bit 0. However, the magnitudes seem rather arbitrary, which is due to the fact that the AWGN channel was attenuating our transmitted signal. In practice, we do not know the attenuation of a wireless channel and we have to learn this information somehow.

The third task is to automatically convert the obtained magnitudes back into bits—we are very good in using our eyes to look at the magnitudes and make that decision, but we want MATLAB to make this decision for us. As just mentioned, large magnitudes map to bit 1 and smaller magnitudes to 0. However, the key question is how do we know what large and small mean? We reiterate that the receiver does, in general, not know the channel's attenuation. To automatically make the distinction between large and smaller amplitude, we first find the largest entry in the vector q and claim that this is a typical amplitude value of a bit that was one. We then assume that 0.5 times that largest value is a typical value of a bit that was zero. With this assumption, we can now distinguish between large and smaller amplitude by assuming that everything that is larger than 0.75 times the largest value was a large value (a bit 1) and everything that was smaller than 0.75 times the largest value was a small value (a bit 0). Basically we put a *threshold* right in-between the largest and 0.5 times the largest value. This process is known as *detection* and is used to make final decisions whether a magnitude belonged to either a bit 1 or 0. Note that this approach is very similar to what we do when we decide which bit was transmitted when looking at the signals with our eyes: we compare a bunch of them and decide which ones are noticeably larger than other ones and determine that those correspond to the transmitted bits that were 1. In words, we automatically use a good decision threshold in our brain. In MATLAB, detection is not too hard. Assume that the vector q contains the estimated magnitudes. We can now use the following two commands:

```
max_magnitude = max(q);
bits_detected = q>(max_magnitude*0.75)
```

The first command extracts the largest value from the vector q. The second command compares each entry of that vector to 0.75 of the maximum magnitude. If an entry in the vector q is larger than `max_magnitude*0.75`, then the entry is replaced with a binary 1 (which also stands for "true" in MATLAB); if an entry is smaller than that threshold, then it is replaced with a binary 0 (which also stands for "false" in MATLAB). Hence, these two commands perform exactly the tasks we wanted to do.

Finally, you can compare the original bit sequence stored in the vector bits with the new vector bits_detected and see whether any errors happened. Note that the transmitted bits in bits should perfectly match with the detected ones in bits_detected, otherwise transmission errors would have occurred. If you want to automatically count the number of errors, you can use the command:

```
errors = sum(bits~=bits_detected)
```

which counts the number of entries in which the two vectors differ.

Activity 33: Build the receiver

Design a script that generates an AM modulated digital signal as in the last module, passes the transmit signal through the provided AWGN channel, and performs detection by following the steps described above. If everything works correctly, then you should have zero transmission errors. If you have transmission errors, then carefully go through all the steps in your script and visualize the signals as done in Figure 23. In case you cannot find your mistake (you should have zero transmission errors), ask us—some errors can be hidden at innocent places in your MATLAB code and are very hard to find. Debugging is a key skill every engineer has to learn!

Important: If a communication system is corrupted by a lot of noise (the noise level is high), then large and smaller magnitudes are harder to distinguish at the receiver. In such cases, the detector may decide for a bit 1 but the the transmitted bit was 0 or vice versa, which causes transmission errors. In general, it is important that the noise is significantly smaller than the information signal to avoid such situations. This is known as high "signal-to-noise-ratio." By increasing the transmission power, one can often make sure that the signal is large compared to the noise. In practice, however, regulations often prevent one from "blasting" signals with too much power—think about everyone in this room would be using their loudspeakers on maximum level. Similarly, for electromagnetic waves the transmit power is also limited due to health reasons (e.g., to not heat up living tissue by too much). In addition to that,

transmitting signals at high power would drain the battery of your cellphone faster, which is clearly not desirable as we already have to charge our phones once per day.

Activity 34: Transmit more bits

If you achieved zero transmission errors, try to transmit more bits over the AWGN channel. In principle you can transmit as many as you want—you just need to change the definition of the vector `bits`. Even if you transmit 1000 bits or more, you should have zero transmission errors with the current channel model. In case you are too lazy to manually define the vector `bits`, you can generate a vector of bits with randomly generated zeros and ones. To do so, you can write

```
bits = [ 1, randn(1,99)>0 ]
```

which generates a single 1 followed by 99 random zeros and ones. Note that every time you execute this line of code, MATLAB will generate a different set of bits at random. If you want, feel free to transmit more bits (one thousand?)—you should not have any transmission errors as we designed an extremely robust communication system for the provided AWGN channel.

Remember: You have designed your first digital communication system for a simple AWGN channel. AWGN channels are pretty accurate models for systems that communicate over cables (but not necessarily for over-the-air communication). A USB or Ethernet cable, for example, can be modeled as an AWGN channel. Wireless systems are more complicated as the channel depends on the physical properties between the transmitter and receiver, which can cause the channel to change over time. You will see such effects (which are called* fading*) in the last module!