Module 9: Synchronization

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

This module will teach you how to synchronize the receiver to the transmitter, which means that the receiver learns when the transmitter decided to transmit information. So far we have ignored this important aspect which is critical to virtually all existing communication systems. Note that we will be using test channels and are not performing any transmissions over-the-air yet—the next module will perform the first truly wireless communication. ***Remember: Please ask us in case you have questions!***

9.7 Synchronization

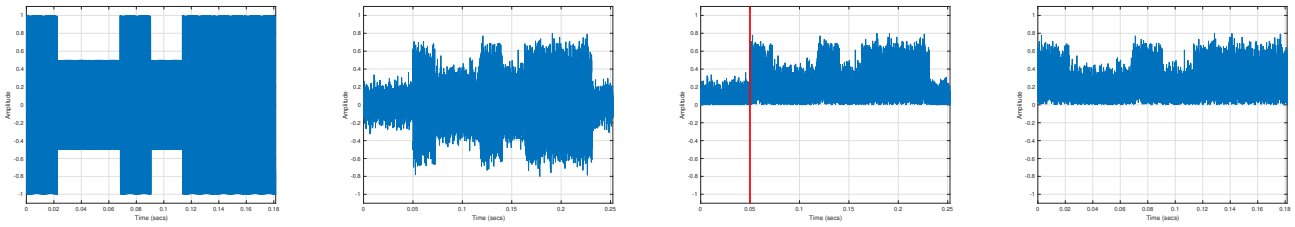
In real-world communication systems, the receiver does not know when the transmitter decides to send information. In general, the receiver has to "listen" to the incoming signals and decide when transmission occurred. Your cell-phone, for example, has to continuously sense the signals coming from the antenna and decide when a transmission occurs, e.g., when someone tries to call you. The process of detecting data transmission at the receiver side is known as *synchronization*. Synchronization was not needed in the AWGN channel we used previously as everything was synchronized perfectly (transmission always started at exactly the first sample, which never happens in practical systems). We now design a simple synchronization scheme so that the receiver can detect when data transmission starts. We will then use this synchronization method in the next module when communicating over the air.

To test our synchronization scheme, we will be using a new channel model that adds a variable delay to transmission over the AWGN channel of the previous module. This variable delay emulates the fact that transmission can occur at any time and the receiver has to identify the exact moment where transmission occurred. To this end, we have created the following MATLAB function

```
y_noise = channel_AWGN_delay(y);
```

which adds a random delay before and after the transmission of the vector `y` . These delays model the fact that the receiver does, in general, not know when exactly data transmission occurs.

Consider, for example, transmission of the digital AM signal from the previous modules. Figure 24 shows the effect of the channel model. The first figure shows the signal that we wish to transmit. The second figure shows the output of this channel model. As you can see, this channel attenuates the transmit signal, adds noise, and also a random delay at the beginning and the end of transmission. Our goal is now to develop a method that detects the start of transmission (see the red vertical line in third figure of Figure 24). Finally, we will crop (or trim) the signal to only keep what has been transmitted. Note that in order to perform cropping, we need to know the start and the end of transmission. The start is detected by the synchronization method; the end of transmission is easy to detect if one knows how many bits will be transmitted and how many samples are used to transmit each bit.



(a) Transmit signal. (b) Received signal with delay. (c) Start of rectified signal. (d) Cropped signal.

Figure 24: Illustration of the main steps of synchronizing the receiver to the transmitter. Digital AM signal at the transmitter (a); noisy received signal with a random delay caused by the channel (b); rectified signal with detected transmission start (c); and cropped signal (d).

To perform synchronization, we will be working with the rectified signal `yr_rect`. The principle of our synchronization method is rather simple: We will detect the time instant of the first "strong" sample we observe at the receiver. In the third figure of Figure 24 you can see that we detect transmission at exactly this point—note that this is also the reason why we decided to always transmit a binary 1 as our first symbol! (Remember that a binary 1 was mapped to a magnitude of 1.) The key question is what we mean by "strong" sample. We can say that a sample is "strong" if it is about 0.5 times the magnitude of the strongest signal we observe. Note that the threshold of 0.5 is rather arbitrary and may have to be adapted later. To find the strongest signal in the rectified vector, simply use the command

```
yr_rect = abs(y_noise);
max_val = max(yr_rect);
```

In what follows, we claim that "strong" signals are larger than `max_val*0.5`. We now have to find the first occurrence of such a "strong" signal. There are many ways to implement this in MATLAB and we are using an extremely compact sequence of commands:

```
start_idx = find((yr_rect>max_val*0.5),1,'first');
```

Let us break this sequence down into separate pieces. The command `(yr_rect>max_val*0.5)` creates a vector that contains zeros and ones. Zeros for the entries of `yr_rect` that are smaller than `max_val*0.5` and ones for those that are larger than this threshold. The command `find` with the argument `'first'` finds the index of the first entry in the input vector that is 1. Note that this is exactly the first index of the rectified receive vector `yr_rect` that exceeds the threshold `max_val*0.5`. Hence, the scalar `start_idx` contains the start index of our transmission. Note that you can visualize this synchronization method as shown in the third figure of Figure 24 using the following plot commands:

```
plot_signal(yr_rect,FS)
hold on
plot(start_idx/FS*[1 1],[-1.1,+1.1],'r-','LineWidth',3)
hold off
```

This overlays a vertical line at the extracted start index `start_idx` with the rectified signal `yr_rect`. If you think that your start index is not correct, then you may have to play with the threshold parameter 0.5. Note that with this test channel, a value of 0.5 works very well but for over-the-air experiments, this threshold may not be working well anymore—trial-and-error may be required.

The final piece of our synchronization method is to crop the signal, i.e., to keep only the transmit signal and remove the rest (the pre and post pauses). Since we know how many bits have been transmitted (by using the command `length(bits)`) and we know that we are using sine waves of length `tx_len` samples for each bit, the transmit signal is of length `tx_len*length(bits)`. As a consequence, cropping is straightforward. Simply use the command

```
yr_rect = yr_rect(start_idx:start_idx+length(bits)*tx_len);
```

which keeps only the signal that is between the start index `start_idx` and the start index plus the length of the transmit signal `start_idx+length(bits)*tx_len`. A typical output of this cropping procedure is shown in the fourth figure of Figure 24. If synchronization succeeds, you should end up with a noisy version of the transmit signal but without the random pauses at the start and end of the transmit signal.

Activity 29: Build the receiver

Write a MATLAB script that generates an AM modulated digital signal as in the last module, passes the transmit signal through the provided AWGN channel that adds random delays, and performs detection by following the steps described above. Include the synchronization method we just described. 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 24. In case you cannot find your mistake (you should easily find the start of transmission and have zero transmission errors), do not hesitate to ask us!

Important: The above synchronization method does not always work. First, it can be that there is a noisy sample that occurs before data transmission and exceeds the threshold—in this case synchronization would fail in a sense that our method thinks the transmission starts earlier than it actually does. Second, it can be that the largest entry of `yr_rect` is extremely large (again due to noise). In that case, `max_val*0.5` would be very large as well which could cause the synchronization method to* miss *the start of transmission. In both cases, data transmission would fail spectacularly as we do not even know when transmission starts. Hence, real-world communication systems are using very powerful synchronization methods that are much more complicated than our simple threshold detector. If you want to know more about such advanced methods, we can explain some of these methods and ideas!

Activity 30: Play with the threshold

Change the threshold `max_val*0.5` in your synchronization algorithm. For example, use `max_val*0.1`. You should see that the synchronization method "believes" that transmission started much earlier than the actual transmit signal. You may even observe some transmission errors which are a result due to synchronization errors. Change the threshold to `max_val*0.99`. You should see that the synchronization "believes" that transmission started much later than the actual transmit signal. Again, detection may fail completely—it may also happen that MATLAB throws an error as the start point is so late that we cannot extract a signal of appropriate length...

Please keep in mind that in many cases if transmission over wireless channels fails, then it can be due to an error in synchronization (and not necessarily due to noise or other reasons). Hence, we urge you to always visualize the synchronization time instant as shown above!