A look into SDR - Part II

Y.Ramgulam

March 2019

In the previous article (*A look into SDR - Part I*) we talked about acquiring the IQ samples of an OOK-modulated signal, and we also talked about the subject of Decimation. Let's continue our journey into decoding an OOK-modulated signal with the next step: demodulation.

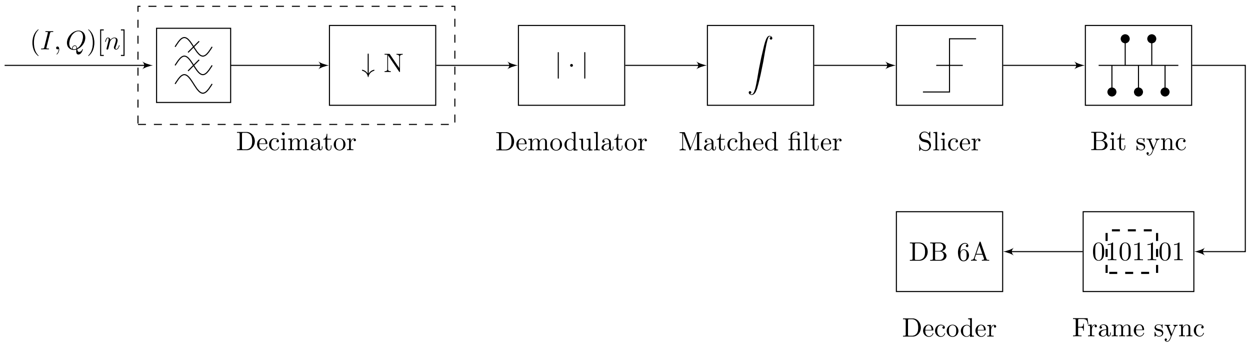


Figure 1: Process flow block diagram

1 Demodulation

A complex signal x with I and Q components can be written as:

$$x = I + jQ \tag{1}$$

The amplitude of x will simply be the magnitude of that complex number:

$$A = \sqrt{I^2 + Q^2} \tag{2}$$

Fortunately Python does a lot of things for us. The Numpy package[1], which contains an impressive set of functions dedicated to numerical analysis and scientific computing, has a function to calculate the modulus of a number:

```
signal_amplitude = numpy.abs(x)
```

The result of this operation is visible in Figure 2:

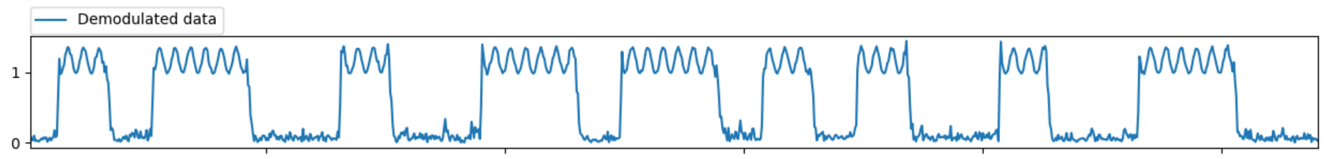


Figure 2: demodulated signal

It's starting to look nice, we can clearly see the bits although there is still a significant amount of noise. Granted the bandwidth we used is too large (and we can see an effect of saturation as I was too close to the dongle) so we could improve things by decimating more, but that's not the point. The signal was recorded close to the dongle so the SNR was ideal. In a real-world scenario where the SNR will be poorer, the signal might be a lot worse than this one.

We can really clean up a signal thanks to an operation which is very common in radio signal processing: matched-filtering.

2 Matched-filtering

A matched a filter is a filter designed to maximize SNR in the presence of random noise. The way it works is by correlating the signal of interest with a reference signal. The correlation will measure the degree of resemblance between the signal of interest and the reference signal. The higher the resemblance, the higher the output of the correlation. Mathematically speaking, the correlation between two signals x and y is given by:

$$r[n] = \sum_{k=-\infty}^{\infty} x[k]y[k-n] \tag{3}$$

Again, don't worry, no need to implement this ourselves: the Scipy[2] package will come to the rescue.

In our case we define our reference signal as a rectangular pulse of length=1bit, since we are looking for pulses that are integer multiples of a bit. Our bitrate is 5000bps and our sample rate is 200ksps, hence we have 40 samples/bit: we create a reference pulse of length=40samples. We then correlate using the following code:

```
matched_data = scipy.signal.correlate(data,ref_pulse,mode='same')/samples_per_bit
# The division at the end is to normalize the result
```

This gives us the result in Figure 3.

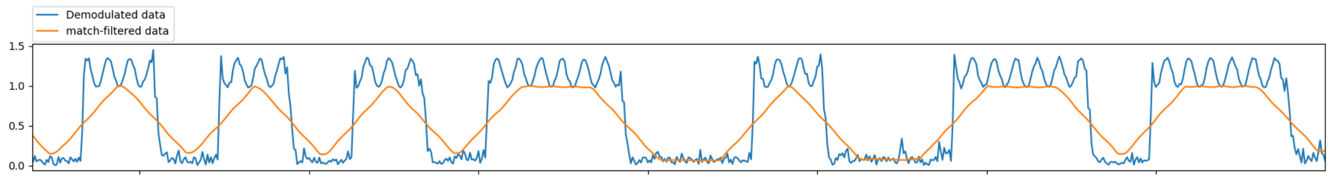


Figure 3: Match-filtered data

We can see that the signal has really been cleaned up and is now triangular rather than square shaped. Visually, locating the bits is trivial. If we want to decode the message, we can stop here, manually recover the 1's and 0's and decode the message. But we're lazy (yes we are) and we want an automated process for that, which brings us to the next step: bit synchronisation.

NOTE: If we refer to Figure 1, I've switched the "Slicer" and "Bit sync" steps, as it was more interesting this way.

3 Bit synchronization

Bit synchronization will be done with a certain type of algorithm also known as "Timing Error Detector" or TED. We're going to use a very simple and famous algorithm called the Gardner algorithm[3], which is part of a broader family of so-called "Early-Late" algorithms. The Gardner algorithm is depicted in Figure 4. If we define Tb as the duration of a bit, the timing error using Gardner is calculated as:

$$e[n] = (x[n] - x[n - Tb]) \times x[n - Tb/2] \tag{4}$$

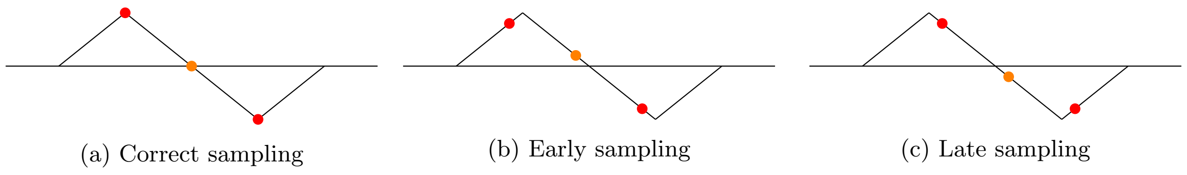


Figure 4: Gardner algorithm

In the example of Figure 4, we have:

- Figure 4a: $e[n] = (-1 - 1) \times 0 = 0$. We have no error
- Figure 4b: $e[n] = (-0.8 - 0.8) \times 0.2 = -0.32$. A timing advance yields a negative error
- Figure 4c: $e[n] = (-0.8 - 0.8) \times -0.2 = +0.32$. A timing delay yields a positive error

Hence we are able to adjust the timing advance or delay to ensure that our bit is sampled at the right time. For the purpose of this exercise I developed a very simplistic implementation of the algorithm with degraded performance. This way we can really see how it behaves. If implemented the right way, Gardner will actually provide much better performance. I leave it to you to explore the python code to see how it was implemented. The result of its execution is shown in Figure 5

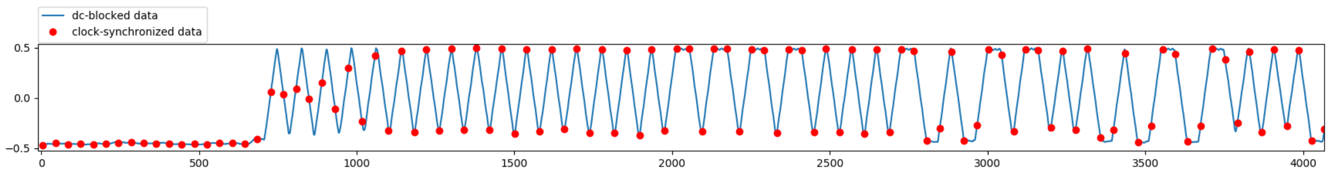


Figure 5: Match-filtered data

First the signal is centered around 0 (No DC component). In this example this was done in a very crude way. There is no equalization so the positive and negative peaks don't necessarily have the same amplitude, which will tend to throw the algorithm off.

We can see that we start in the noise and as soon as a signal appears the algorithm starts to track it (~750 mark) until it synchronises with it (~1100 mark). Then we can see at ~2800 mark that the TED starts to lose it synchronisation as sampling is done too late. It then readjusts its timing advance to recuperate and at ~3100 mark it's back on track.

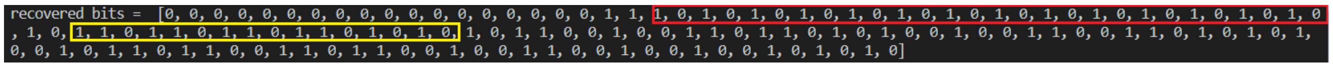
Now that our TED has done the job of identifying the position of the bits for us, we can go to the next step: Slicing.

4 Slicing

This operation has nothing really exciting to it: it's simply transforming our pseudo "analog" signal in digital 1's and 0's. At the previous step we have identified the position of our bits (or where we should sample them).

We then take these samples and apply a simple decision process:

```
if sample > 0:
        bit = 1
else :
        bit = 0
```



```
recovered bits =  [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0
, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 1, 0, 1, 0, 0, 1, 0, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0, 1,
0, 0, 1, 0, 1, 1, 0, 1, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0]
```

Figure 6: bit array

From that we obtain an array of bits visible in Figure 6. If we inspect them visually, we can spot what looks like a preamble (sequence of 101010.., in red). We can also spot our synchronisation word. Referring to the previous article, the sync word is DB6A in hexadecimal, or 1101 1011 0110 1010. This sequence can be identified in Figure 6, in the yellow rectangle.

Again, at this stage we could decode the message and reconstitute our bytes manually. But since we're definitely very lazy, we need something to identify the sync word for us, and then decode the payload. This brings us to the next step, Frame synchronization.

5 Frame synchronization

So far we have obtained a raw stream of bits. The problem is that the payload is lost in it and we don't know where it begins. This is the purpose of the synchronisation word, a specific sequence that will serve as a marker for the beginning of the payload. In this section we will find a way to detect the synchronisation word so that we can learn when the payload begins.

A trivial way of doing this would be to simply parse our bit array and find the matching pattern using an XOR. The problem with that method is that it leaves no room for errors. But why would we want to leave room for errors?

In "real life", the magic of wave propagation could result in one or more erroneous bits in the synchronisation word. But it doesn't mean the payload would be corrupted; or even if it was, protocols that integrate error coding could allow us to recover those bits. So why throw away a potentially good message by being too restrictive?

To tackle this problem we're going to re-use **correlation** (which comes down to doing an XOR but in a different way). Correlation measures the level of resemblance between two signals (which is what we want here), and it allows us to easily set a threshold for that level of resemblance; thus we can choose to be more or less restrictive. The way of implementing the threshold is left to you. We correlate our bit array with our sync word and show the result in Figure 7.

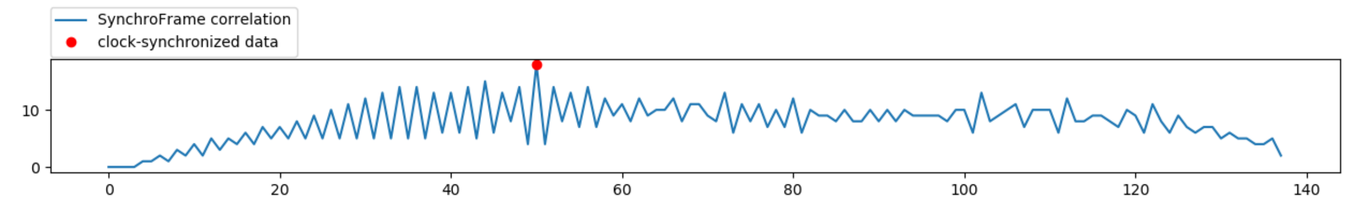


Figure 7: Correlation of sync word with bit array

I created a function performing the correlation and decision process and I chose an arbitrary threshold of 12: the process returned a positive match at index 50. By positive match, it only means that we are confident to a certain level that a sync word was found at that location. The peak indicates the middle of the sync word so the payload starts a bit later, in this case at index 66. The decision process I implemented is really basic:

```
for i in range(1,len(data)-1):

  #if we have a peak above the threshold
  if (data[i] > threshold):

    #We check that the neighbouring peaks are sufficiently lower
    if (data[i]-data[i-1]>threshold) and (data[i]-data[i+1]>threshold):
          sync_index = i
          payload_start = i + len(synchronisation_sequence)/2)
    else:
      pass
  else:
    pass
```

NOTE: When I wrote this code about a year ago, I didn't know about a Scipy function that would do the same thing, only much better: **scipy.signal.find_peaks**. I advise you to check it out, I've since then learned to use it and it's quite powerful if correctly configurated.

So now we know where our payload starts. We also know (See Part 1) that we are expecting 9 bytes. the only thing left to do is to create a function to transform the remaining bits in bytes.

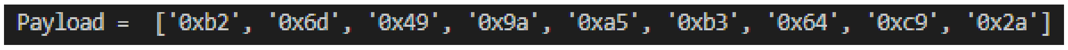
6 Payload decoding

The payload decoder simply converts the bits that come after the sync word into bytes:

```
for j in range(0, payload_size):
        payload_data = 0
        k = 7

    for i in range(start_index + j*8, start_index + j*8 + 8):
        payload_data += data[i] << k
        k -= 1
```

We get the following payload bytes:



```
Payload =  ['0xb2', '0x6d', '0x49', '0x9a', '0xa5', '0xb3', '0x64', '0xc9', '0x2a']
```

Figure 8: payload

Which is what we were expecting!

Conclusion

Throughout these articles we have gone through a complete chain of demodulation, from the acquisition of IQ samples to a decoded payload expressed in hexadecimal. There were no "manual" steps in the decoding process and all of this was realised using Python and some of its most famous packages. I really hope it can spark some interest in people who are curious about Software Defined Radio. All the code that was used for writing these articles is available on github at `https://github.com/yramgu/SDRarticle.git`. Have fun!

Y.Ramgulam

References

[1] Numpy. `http://www.numpy.org/`.

[2] Scipy. `https://www.scipy.org/`.

[3] Floyd M. Gardner. A bpsk/qpsk timing-error detector for sampled receivers. *Communications, IEEE Transactions on*, 34:423 – 429, 06 1986.