Selected Applications of Digital Signal Processing

V 3.0, November 18, 2016
Dmitriy Shutin, dshutin@tugraz.at
last updated by Michael Müller, michael.g.mueller@student.tugraz.at
Signal Processing and Speech Communication Laboratory, Graz University of Technology
Inffeldgasse 16c
http://spsc.tugraz.at

Required equipment:

- PC with Code Composer Studio & MATLAB installed
- Texas Instruments DSP board
- Oscilloscope Agilent 54622D
- Signal generator Agilent 33120A
- Cables

1 Theoretical part

1.1 Fast convolution with pre- and post-windowing

The goal of this experiment is to demonstrate how signal processing can benefit form the specific structure of the C-code that is used to process the data.

In this laboratory we have extensively used the block-processing environment that has been designed to process data on a block-by-block basis. In the heart of the block processing environment lie two circular buffers, which are used as a storage for the input and output data samples. This structure become particularly useful when it is necessary to implement Fast Convolution algorithms with the Overlap-Add technique when, for example, a non-interrupting processing of the input signal is desired.

The block processing environments reads and writes hops consisting of `HOPSZ` samples each time they are provided from the input by the audio codec. They are written to the working buffer `workbuf` which by default can fit `BLKSZ=2*HOPSZ` samples. This is were processing takes place. We can implement convolutions with filters of length up to `HOPSZ+1` using this buffer, since the length of the convolution of two signals $(a * b)[n]$ is $L_a + L_b - 1$.

We have set the `OVERLAP` constant to `1` in `blocks.h`, therefore we have 50% overlap between two consecutive buffers. This means that each hop of input samples will be present in the

working buffer for two processing iterations. The first half of `workbuf` stores the old samples, the second half the newer ones. When processing is done, the samples stored in the first half of `workbuf` will be written to the audio output by the environment. Samples present in the second half will be saved and added to the next output (Overlap-Add).

Since we only need to process each hop only once, we can set half of the working buffer to zero: either the first half (the old samples), or the second half (see Figure 1):

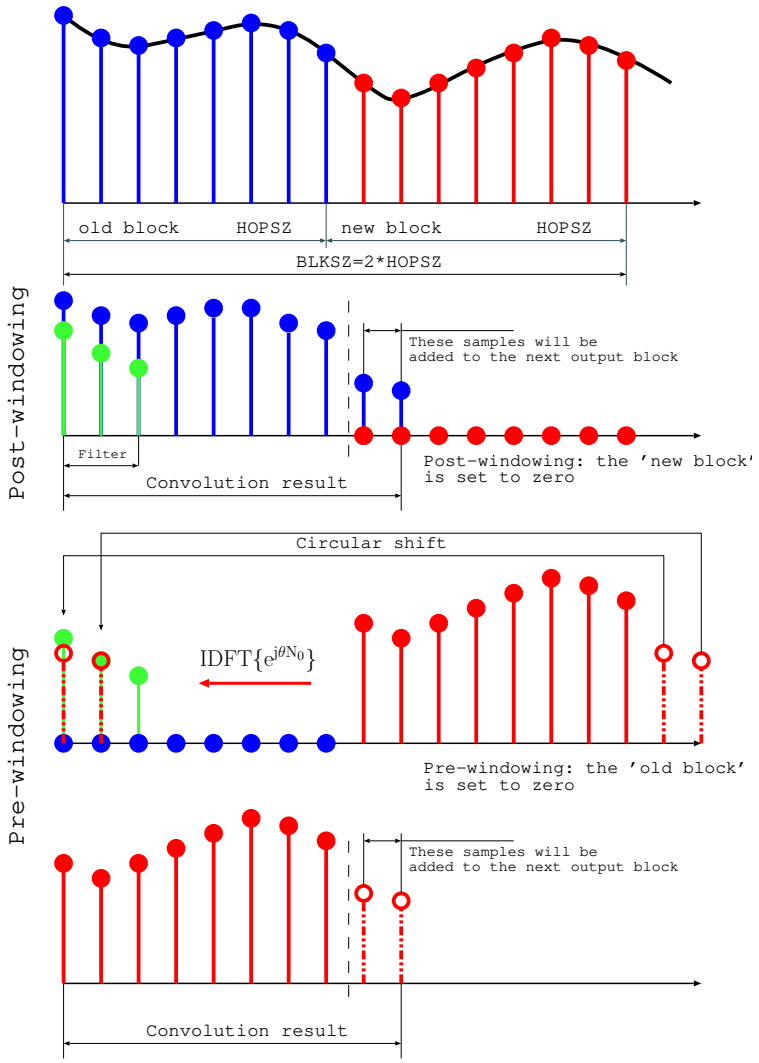

Figure 1: *Implementing Overlap-Add with pre- and post-windowing.*

- **Post-windowing.** The new samples are set to zero. After the convolution takes place in the buffer, the convolution output lies at the correct position in the buffer as expected by the block processing environment.
- **Pre-windowing.** The old samples are set to zero. The convolution will thus lead to the samples lying in the wrong position. First and second half of the buffer need to swapped before the processing iteration is finished.

These two schemes are computationally completely equivalent and produce the same output. However, there is a difference since we are working in a real-time environment: when using

pre-windowing, each input hop is processed immediately, thus, the delay from input to output is smaller.

The shift operation required by the pre-windowing scheme can be performed in the Fourier domain by multiplying the spectrum with $\mathrm{e}^{j\theta N_0}$. The shift width is $N_0 = \texttt{HOPSZ}$ since this is half of the length of `workbuf`. Since the FFT is computed over `BLKSZ` bins, we can expand this term and obtain the spectrum

$$\mathrm{e}^{j\theta_k N_0} = \mathrm{e}^{j\frac{2\pi}{\texttt{BLKSZ}}kN_0} = \mathrm{e}^{j\frac{2\pi}{\texttt{BLKSZ}}k\texttt{HOPSZ}} = \mathrm{e}^{j\pi k} = \cos(\pi k) = (-1)^k, \quad k = 0 : \texttt{BLKSZ} - 1.$$

Thus, the shift can be performed simply by changing the sign of all samples at odd frequency bin positions. Of course, we can also apply this shift directly to the DFT of the filter we wish to apply, so the operation only needs to be performed once and not at every processing iteration.

1.2 QAM modulator

Quadrature Amplitude Modulation (QAM) is a common digital modulation technique. It is used in standard telephone modems, FAX modems, wireless LAN standards and many other digital communication systems.

Modulation. The block diagram of a QAM modulator is shown in Figure 2. It consists of four parts:

- *Processing of the bit stream.* The input data bits d arrive at the rate of $F_{data} = 1/T_{data}$ (in bits/s). They are converted into binary words of length J bits each by simply concatenating consecutive bits. Each J-bit word is mapped to a channel symbol from the QAM alphabet, which consists of 2^J symbols.

  Each symbol is represented by a complex number $c_n = a_n + jb_n$ selected from the QAM constellation. For example, for $J = 2$, the possible binary words are:

binary word	00	01	10	11
signal point	$1+j$	$-1+j$	$1-j$	$-1-j$

  It is customary to call the real part of the symbol a_n the in-phase component or I component, and the imaginary part b_n the quadrature or Q component.

- *Impulse Modulators.* The I and Q components are then passed through the impulse modulators

$$\check{a}(t) = \sum_{k=-\infty}^{\infty} a_k \delta(t - kT_{symb})$$
$$\check{b}(t) = \sum_{k=-\infty}^{\infty} b_k \delta(t - kT_{symb}) \ ,$$

  resulting in the pulse-modulated representation of the transmitted symbols.

- *Baseband Transmit Filters.* Next, the signals are passed through identical baseband transmit filters with impulse response $g(t)$. These are typically a lowpass filters approximating a raised cosine or the square-root of a cosine response. The role of the filters is to restrict the bandwidth of the transmitted signal. The filter output is given by

$$a(t) = \sum_{k=-\infty}^{\infty} a_k g(t - kT_{symb})$$
$$b(t) = \sum_{k=-\infty}^{\infty} b_k g(t - kT_{symb}) \ .$$

- *Amplitude Modulation.* In order to transmit the signal, the in-phase and quadrature components $a(t)$ and $b(t)$ are *double-sideband suppressed-carrier amplitude modulated* by the carriers $\cos(2\pi f_c t)$ and $\sin(2\pi f_c t)$. The resulting pass-band signal $s(t)$ is given by

$$s(t) = a(t)\cos(2\pi f_c t) - b(t)\sin(2\pi f_c t) \ .$$

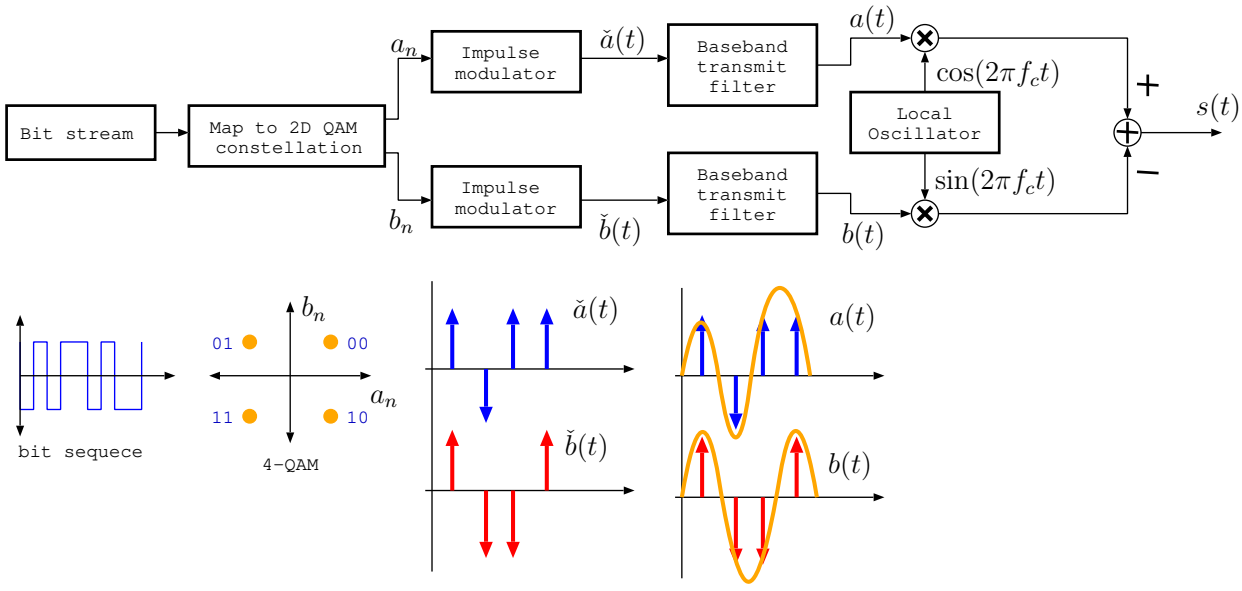

Figure 2: *A Basic QAM modulator.*

This signal $s(t)$ is then passed through some channel, after which it needs to be demodulated to obtain the original bit sequence.

Obviously, different QAM constellations can be used. In practice, $J = 2$ (4-QAM), $J = 4$ (16-QAM), $J = 6$ (64-QAM), $J = 8$ (256-QAM), and $J = 10$ (1024-QAM) are common (see Figure 3). We will use 4-QAM in this experiment. The number of bits J directly leads to the symbol rate (also known as baud rate) of the signal, which is given by

$$F_{symb} = \frac{F_{data}}{J} \ .$$

Each symbol is typically transmitted by some number of digital samples N.

Demodulation. The receiver can recover the quadrature and in-phase components from the modulated signal by using a Hilbert transformer. In this lab unit, we will focus only on the modulation and on influences by non-optimal transmission channels.

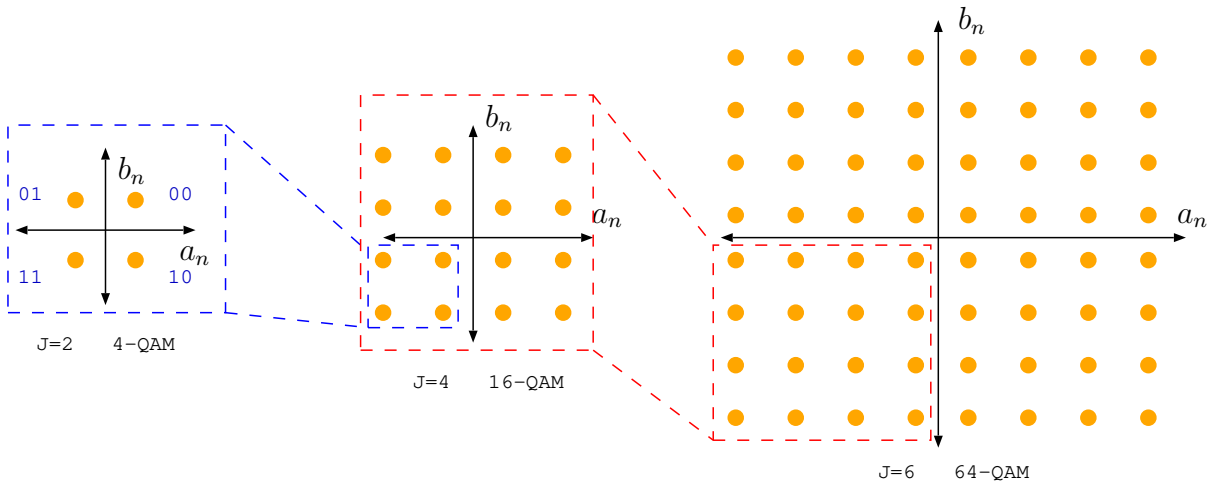

Figure 3: *Example mappings for 4-QAM, 16-QAM, and 64-QAM.*

2 Practical part

Experiment 1
Fast convolution with pre- and post-windowing

1. Connect the signal generator to the input of the DSP board and set up the signal generator to produce a sine wave with frequency 1 kHz and amplitude 0.5 V_{pp}. Connect the DSP output channel to the oscilloscope input.

2. Start CCS and load the `fundamentals/fastconv/fastconv.pjt` project. The loaded program implements the post-windowing approach. Build the program and run it. You should be able to see the output signal once you start the program. You can use averaging to obtain a cleaner signal.

3. Now, on the signal generator press **Shift** → **Burst** to switch on burst mode.

4. Press **Shift** → **Menu** to activate internal menu and enter ***Mod Menu*** by pressing ∨ . Set ***Burst CNT*** (i.e. the number of burst) to 1 and ***Burst RATE*** to 10 Hz.

5. Compress the horizontal axis to 10 ms. This corresponds to the total observation window of 1/10 Hz = 100 ms. Make sure that only two pulses are visible on the screen: one from the signal generator — the reference pulse — and another one from the output of the DSP board. The DSP pulse is be delayed due to the processing time of the board.

6. Using oscilloscope cursors, measure the delay between the pulses. Express the measured value in samples assuming a sampling rate of 32 kHz. Note: increasing the oscilloscope resolution can lead to a more accurate result.

7. Similarly, measure the processing delay for some (e.g. 3) different filter orders `N`.

   - You can easily design simple FIR filters in MATLAB using the `fir1` command. Enter the coefficients and the filter order `N` in `processing.c` .
   - You may need change the `HOPSZ` parameter in `blocks.h` according to the filter order. For example, for a filter with `N`=96 coefficients, `HOPSZ` has to be set to 128 (it must always be a power of two for the computation of the FFT to be possible). Similarly, for `N`=10, `HOPSZ`=16 would suffice.

8. Modify the program to implement the pre-windowing approach.

   - Note that all buffers store complex numbers: in `workbuf`, the k-th sample is stored in `buf[2*k]` (real part) and `buf[2*k+1]` (imaginary part). Accordingly, each buffer has length `2*BLKSZ`. In the beginning, only the real parts are nonzero.
   - You must change two things: the initialization of the spectrum of the filter `H` in the `init()` function, and the `processing()` function.
   - If you use preprocessor instructions like `#define`, you can easily switch between pre- and post-windowing afterwards.

9. Again, measure the delay between two pulses. How much do we gain?

10. Fill the following table with at least three filters of different lengths for both post- and prewindowing and explain the results.

pre / post	filter order	Δt	Δt	processing	codec	τ_{gr}	error
		ms	samples	samples	samples	samples	samples
...	...	...	...	...	...	...	...

- *Processing* refers to the theoretical delay which follows from the windowing scheme and is in general `k * HOPSZ`.
- The delay from the audio codec is 27 samples.
- The delays introduced by processing, audio codec, and filter should add up to the measured delay Δt. If they do not, enter the missing number of samples in the *error* column.

Experiment 2
QAM modulator

1. Disconnect the DSP from the signal generator, and also disconnect the ***SYNC*** cable from the signal generator (or simply switch the signal generator off, since it will not be used in this experiment). Connect the left and right DSP output channels to the oscilloscope.

2. If necessary, start CCS and load the `fundamentals/qam/tx/tx.pjt` project. Open `processing.c` and verify that the settings are as follows:

```
MODULATE 0
SINGLECHANNEL 0
INC_TRANS_CHANNEL 0
```

   This will disable the carrier modulation, and will result in the in-phase and quadrature components outputed separetly over the left and right channels. The last definition will switch off the simulation of the communication channel influence.

3. Now, let us start with the 4-QAM scheme. Set `#define QAM 4`, build the program and run it. Describe the output visible on the oscilloscope.

4. On the oscilloscope horizontal control panel press the **Main/Delayed** button and select ***XY*** mode. What can you see?

5. By default the program will only perform simple impulse modulation without the usage of transmit filters. Implement the baseband transmit filters (see Figure 2) in `processing.c`.

- Use the `fir()` function, which implements FIR filtering.
- The impulse response of a raised cosine filter is stored in the variable `G[]`.

- The positions at which you need to add code are indicated by comments in the `qam()` function.

6. Again inspect the oscilloscope output (normal view and XY-plot). Describe the changes introduced by the filters. Why are they used?

7. Now, include the simulation of a channel. First, set `INC_TRANS_CHANNEL 1`. This will simulate a simple communication channel with an impulse response $h[n] = (c + jd)\delta[n]$. What has changed in the XY-plot? Find the function `channel()` in `processing.c` which simulates the channel and set other values for the channel coefficients `c` and `d`. Describe the influence of each parameter.

8. Next, set `INC_TRANS_CHANNEL 2` to simulate the influence of a time-varying channel with the impulse response $h_t[n] = (c(t) + jd(t))\delta[n]$. Describe the consequences visible in the XY-plot. This kind of channel requires adaptive equalization which cancels the time-varying nature of the channel.

9. In the source code set `INC_TRANS_CHANNEL 1` and `SINGLECHANNEL 1`. In this mode, one of the channels will be used to transmit the sync signal that changes its state at the symbol boundary, while the other channel outputs the difference between the in-phase and the quadrature components. The latter is real valued and it is exactly the signal that is then transmitted over the channel.

10. On this oscilloscope, press **Math** → ***FFT*** → ***Setting*** and set ***Source*** to the channel that corresponds to the information signal, ***Span*** to 20 kHz, and ***Center*** to 10 kHz. Measure the zero-to-zero (i.e. double-sided) signal bandwidth B from the FFT spectrum.

11. In our case, each symbol consist of `N` samples (see `processing.c`), and the sampling rate is $F_s = 32$ kHz. Compute the symbol period T_{symb} and the corresponding symbol rate (or baud rate) F_{symb} (in symbols/s) for this system. What is the corresponding data rate (or bit rate) F_{data} (in bits/s)? Compare these numbers to the bandwidth of the transmitted signal.

12. Now, enable the modulation by setting `MODULATE 1` in the source code. Measure the center frequency F_c and the bandwidth of the resulting channel. What is the connection between the pass-band bandwidth and the zero-to-zero bandwidth measured above? Also, compute the roll-off factor α of the raised cosine filter: $B = F_{symb}(1 + \alpha)$.

13. Try other modulation schemes, e.g., 16-QAM, 64-QAM, 256-QAM, and 1024-QAM. Compute the the bandwidth of the modulated signal and the corresponding symbol and data rates in these cases as a function on J (the number of bits per symbol). What is the advantage of using a large J? What problems may occur?