Bartosz Golis

Acoustic Equalizer

Introduction
The subject of the project we chose for the digital signal processing course was the design of an acoustic equalizer. An equalizer, also known as a "correction device" or "equalizer," is a set of filters used to boost or attenuate specific frequency ranges, thus altering the sound's tonal quality.

The application of an equalizer is an integral part of the sound processing and mixing process during the recording of live musicians, acoustic tracks, synthetic sounds, or samples. Equalizers are used to shape the overall sound, correct specific frequency issues, and blend sounds from different sources or instruments. There are several types of equalizers, with graphic and parametric equalizers being the most common. In this case, the acoustic equalizer project aimed to filter the signal or process music for the user's needs.

Most typical equalizers use active circuits that actively boost or attenuate the sound in different frequency bands by employing electronic feedback techniques. These active circuits can introduce audible "ringing" due to inevitable phase shifts in such situations. Passive equalizers, on the other hand, work differently. The frequency response correction is achieved using passive elements (operating without power), such as resistors, capacitors, and inductors. A single amplifier stage with a simple design placed after this circuit compensates for the signal level decrease, resulting in a flat frequency response when all controls are in their central positions.
The equalizer's purpose was to set appropriate signal amplitudes in different frequency bands. The project consisted of three parts:

- Acoustic equalizer implemented in the Code Composer Studio using the C language.

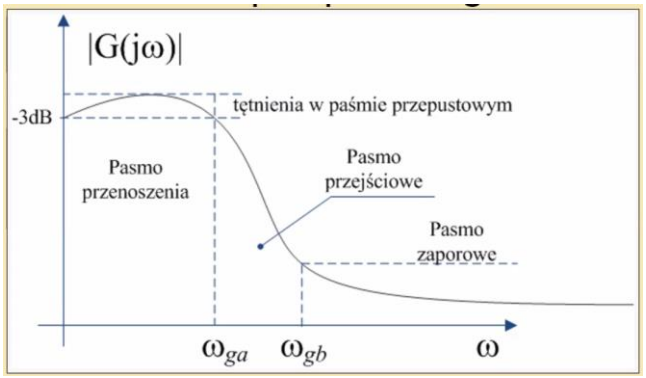
- Configuration files for the equalizer written in MatLab. For the project, three separate scripts were designed, allowing the user to easily change the settings of each of the three used filters.

- GUI (Graphic User Interface), a menu created using Windows Forms, incorporating elements from C# and C++. This additional element enables user-friendly control of the entire project.

Project Description:

I)
The equalizer designed as part of the project consisted of three filters responsible for attenuating specific frequency bands.

1. The first filter was a low-pass filter aimed at attenuating the bass frequencies, i.e., frequencies from zero to 200 Hz. Low-pass filters find many applications in signal processing. For example, they are used to remove aliasing effects, which occur when samples overlap, by employing a low-pass filter.



Rys.1 Charakterystyka amplitudowa filtru dolnoprzepustowego.

The quantity characterizing such a system is the transfer function, which is defined as the ratio of the output voltage to the input voltage and is often expressed in operator form:

$$G(s) = \frac{K}{Ts + 1}$$

Where:

K - gain in the passband
T - time constant
S - complex pulsation (angular frequency), $s = j2\pi f$.

2. The second filter is a bandpass filter, also known as a mid-pass filter. This filter allows frequencies in the range from 200 Hz to 2000 Hz to pass through the equalizer. A bandpass filter with a wide bandwidth is often realized by constructing a circuit with a low-pass filter up to a certain central frequency value and a high-pass filter that allows frequencies above the central cutoff frequency to pass through, extending to the desired end of the filter's passband.
In the project, filters were implemented using an algorithm, which greatly facilitates the attenuation of signals, eliminating the need for using other filter approximations. If the filter settings are not optimal, the project allows for the possibility of changing parameters. In the Matlab code, there are two lines responsible for generating the values on which the filters are based. These values can be modified as needed, keeping in mind that both arrays must have the same number of elements.

```
fHz=[0 50 100 150 200 250 300 1250 1350 1950 2050 3000 fNq]; %wektor f
amp=[1 1 1 1 1 0.5 0 0 0 0 0 0 0];% i odpowiadające im pożądane amplitudy
```

Fig. 2. A snippet of code from the script of the low-pass filter.

As can be observed, the values in the "amp" array, responsible for attenuation, are set to one only at frequencies up to 200 Hz. This establishes the passband and attenuation bands of the low-pass filter. Wherever the coefficients are set to 0, the signal is attenuated, and the filter does not pass it through. This is the operating range of the bandpass filter, and it is responsible for passing (or attenuating) signal values within this range. As can be noticed, there is also one value set to 0.5. This is done to enhance the quality of our equalizer. The filters slightly overlap with each other to avoid creating a rigid boundary, allowing sounds on the border, for example, between bass

and midrange, to be gently attenuated.

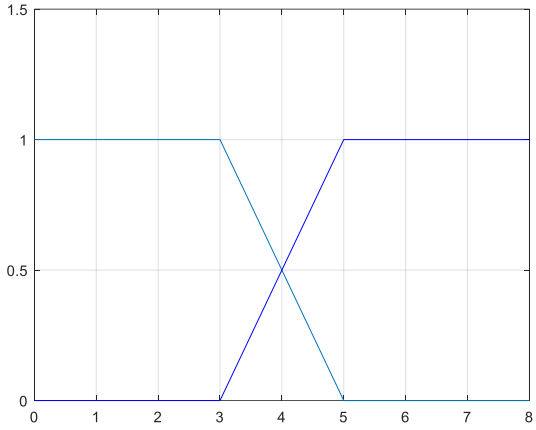


Fig. 3. Schematic diagram illustrating the overlapping passbands of two filters.

3. The third and final filter is a high-pass filter that allows only values above a certain cutoff frequency, which in our case is the range from 2000 Hz to 4000 Hz, corresponding to half of our sampling frequency. The equalizer was designed for a sampling frequency of 8 kHz. In an ideal filter, the attenuation coefficient in the passband should be zero, while in the stopband, it should be significant. Understanding the frequency response characteristic of the phase coefficient allows us to determine the change in voltage and current phase when the signal passes through the filter. Similarly to the previous filters, in the program files, we also have a script responsible for the filter settings and its attenuation coefficients.

```
fHz=[0 50 100 150 200 250 300 1250 1350 1950 2050 2500 2700 3000 fNq]; %wektor
amp=[0 0 0 0 0 0 0 0 0 0 1 1 1 1 1];% i odpowiadające im pożądane amplitudy
```

Fig. 4. Lines of code with settings for the high-pass filter.

As can be seen in this case, there is no attenuation value of 0.5 at the passband boundary. This is because the values at the boundary frequencies are close to each other (1950 Hz and 2050 Hz). Therefore, if we set the value to 0 for the first frequency and 1 for the second frequency, the response curve will increase from 0 to 1 between them, creating a transition band in the shape of a rising function. This configuration results in the response curve resembling a trapezoid shape.

After uploading each file, the equalizer library generates configuration files with the specified settings for each filter.

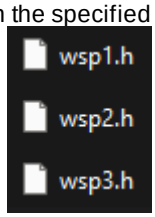


Fig. 5. Libraries with settings for all three filters:

II)

The next element of the project is the actual equalizer code, which combines the filters and utilizes their operations.

```
//Fir.c
#include "dsk6713_aic23.h"  //plik narz

Uint32 fs=DSK6713_AIC23_FREQ_8KHZ;  //u


#include "wsp1.h" //wspó³czynniki filtr
#include "wsp2.h" //wspó³czynniki filtr
#include "wsp3.h" //wspó³czynniki filtr
int yn = 0,yn1=0,yn2=0,yn3=0;
float bass_gain,mid_gain,treble_gain;
short x[N];              // tablica prób
```

Fig. 6. The beginning of the equalizer code in Code Composer Studio.

Analyzing the code, we can observe that the first lines consist of the implemented libraries, both for the processor drivers and the coefficients of all three filters mentioned earlier. The last three lines are for adding necessary variables. In the first of these lines, we see variables initialized to zero. These variables will be incremented later, and they will hold the value of each subsequent sample. The first one, yn, corresponds to the final value of the sample after filtering. The remaining three variables store the sample after filtering through each of the filters, one at a time.

```
interrupt void c_int11()        //funkcja obs³ugi przerwania
{
    short i;

    x[0] = input_sample();    //czytanie próbki z przetwornika A
        yn = 0;

    for (i = 0; i< N; i++)
    {
        yn1 += (h1[i] * x[i]);
        yn2 += (h2[i] * x[i]);
        yn3 += (h3[i] * x[i]);  // tutaj robimy sumê iloczynów(
    }
    for (i = N-1; i > 0; i--)
       x[i] = x[i-1];       //przesuniêcie o jedn¹ próbkê zawart

    yn=bass_gain*yn1*1/3+mid_gain*yn2*1/3+treble_gain*yn3*1/3;
    output_sample(yn);    //wys³anie próbki wyjœciowej do przetw
    return;
}
```

Fig. 7. Further part of the equalizer code.

The code shown in the above image illustrates the implementation of the library "x[0]," responsible for storing input signal samples. Next, there is a "for" loop that processes each sample through all three filters. The incoming sample is multiplied by the coefficient of each filter from the library. After filtering, the next loop returns to the initial value of the "x[i]" array.

As can be noticed, all samples are now multiplied by the appropriate slider coefficient and divided by three. The division is done because the implemented program assumes the parallel connection of filters, not in series. This allows the filters to operate independently from each other. The second set of coefficients, which we multiply the samples by, corresponds to the settings of the sliders.

III)

The sliders were added to the program to allow adjusting the operation of each individual filter. The .gel file has been implemented into the program.

```
slider Bass(0,10,1,1,bass_slider) {       /* From 0 to 10
      bass_gain = ((float)bass_slider)/10;  }   /* ->From

slider Middle(0,10,1,1,mid_slider) {            /* From 0
      mid_gain = ((float)mid_slider)/10;  }     /* ->From

slider Treble(0,10,1,1,treble_slider) {         /* From 0
      treble_gain = ((float)treble_slider)/10;  }   /* ->F
```

Fig. 8. Presentation of the code responsible for adding sliders.

This solution allows manipulating the attenuation level of each filter individually. Depending on the slider settings, the real-time value of a variable is adjusted, which is multiplied with the final sample values in the main program.

The slider can be adjusted from 0 to 1 in increments of 0.1. Setting the slider to the bottom (value of zero) fully attenuates the signal and does not allow it to pass. This enables us to alter the sound, for example, to only pass through frequencies in the middle.

IV)

The last element of the project is a menu that facilitates navigation through the files. To ensure smooth functionality, the equalizer files have been placed in the menu files. This approach allows dynamic paths, making it possible for anyone who downloads the project to run it. However, the menu was created using newer Visual Studio solutions and the internal structure of Windows Forms, which makes it difficult to run on older devices.

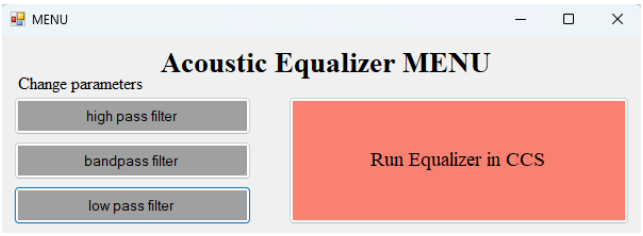


Fig. 8. View of the designed menu.

The menu has four buttons essential for the project's operation. Three gray buttons on the left side, labeled with the names of the filters, execute the Matlab scripts responsible for the attenuation coefficients of each file, as described earlier. The menu operates in the background, making it active and waiting for the Matlab scripts to complete their work. The last button is responsible for launching Code Composer Studio, where we have the entire equalizer project along with the sliders. This solution makes the usage easier and eliminates the need to delve into the project files.

User Manual

To run the project, you need to open the file named "Project_EQ," then locate and open "GUI_EQ." After that, navigate to the "Debug" folder and run the application from within the "Debug" folder to launch the equalizer.

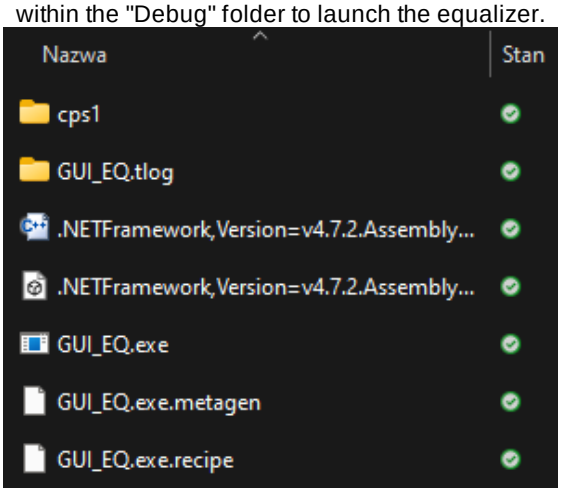


Fig. 8. View of the Debug folder.

The menu of the entire project is an application located in the "Debug" folder and named "GUI_EQ." Once the menu is launched, you can easily navigate through the project. Running the scripts has been described above (Menu Description IV). However, running the equalizer itself is more complex. To do this, follow these steps: Click on the "Run Equalizer in CCS" button in the menu. Choose the "load project" option and select the .pjt file in the project files. Compile the program. In the "Debug" folder of the equalizer (not the one in the GUI project folder), select the .out file. Click on the "connect" option in the toolbar at the top of the Code Composer Studio window. Then, click the "run" button to start the program. To add buttons, follow these steps: Before compiling, add a .gel file (it should be added at the beginning as it is in the project files). In the toolbar, select the "GEL" option and enable all sliders by clicking on their names in the extension of the "graphicEQ" button.

Summary

The program has been written to be clear and user-friendly. The knowledge gained during the digital signal processing classes, introductory sessions, and lectures provided insights into using the Matlab environment and its functions essential for determining data transmission, samples, filter design, checking their aspects, and many other useful things, all of which were incorporated into this project.

At the beginning, an equalizer amplifier was designed, consisting of three filters: low-pass, high-pass, and bandpass filters. Each filter had its specific role in the program.

- The low-pass filter aimed at attenuating bass frequencies up to 200Hz. The filter's performance in attenuating bass frequencies was tested and verified in the laboratory. A similar filter was also used to suppress unwanted spectral overlays called aliasing (described in the project description).

- The second bandpass filter passed frequencies from 200Hz to 2000Hz. Its purpose was to attenuate or pass signals at mid-range frequencies, and its parameters could be adjusted freely by the user.

- The final filter briefly described in the summary is the high-pass filter. Its cutoff frequency ranged from 2000Hz to 4000Hz, which is half the sampling frequency value. By

applying knowledge about filters, appropriate filter types and cutoff frequencies were chosen to meet the requirements of creating the equalizer amplifier. Each filter was implemented in Matlab code, and samples were generated for them using special libraries.

Once the filters and the Equalizer code written in C were designed, adjustments were made to ensure perfect sound, and each filter type fully fulfilled its purpose. The code was developed based on code provided in a small project on filter design, information provided by the instructor, and supplementary materials found on the internet. The Equalizer (specifically the filters comprising it) was designed in parallel, which was much easier as it did not require filtering each sample individually through all three filters. Instead, each sample was filtered by one filter with the appropriate frequency, and then the output sample component was multiplied by 1/3. The code was thoroughly documented, explaining the purpose of each line of code in the instructions.
Thanks to this project, we familiarized ourselves with the concept of sliders and how to add them to the project using the gel extension, as demonstrated by the instructor. We understood their purpose, how to find and design them for easy manipulation of operations, such as adjusting the bass (i.e., using the low-pass filter). An important element of the project was also creating the menu window. The menu window was designed to facilitate navigation through the Equalizer files. We grasped the principles of creating menus in Visual Studio, specifically using WindowsForms. As we were eager to expand the project further, we not only focused on optimizing and streamlining the code written in C, and improving our programming skills and understanding of digital signal processing, but also paid attention to the aesthetics. We aimed to make this project stand out from others (done by other groups) executed on boards in the laboratory. We wanted it to be more than just source code to analyze, but also to have a graphical interface that provides clear instructions for using the equalizer amplifier. Each button in our menu (the home window) was briefly classified in the program description, where everything was detailed and clearly explained. The project is functional; however, a lesson learned for the future is the unexpected file compatibility issue between the new and old operating systems. Unfortunately, the Equalizer menu is not supported on older computers used during the laboratory, resulting in a lack of convenient and clear graphical interface, which makes navigating the project problematic, contrary to the intended goal. On the other hand, newer computers encounter issues with the Code Composer Studio, the main program used throughout the project. The newer operating system refuses to run the application, which led to some parts of the project requiring remote work, while other parts involved writing the code in the laboratory. To achieve optimal performance, it would be necessary to create the menu in older programs or design the GUI in a different language allowing smoother interaction with other systems (e.g., Python or JavaSwing). Designing the Equalizer allows us to familiarize ourselves with the basic operation of the processor, programming it to utilize sounds and other signal samples. This knowledge is essential in digital signal processing as it provides the foundations for creating commercial projects sold on the market by companies that eventually reach consumers. Potential buyers of related projects could be bars, clubs, or concert stages that always demand good sound processing equipment.

Literature

[1] *https://pl.wikipedia.org/wiki/Korektor_graficzny*
[2] *https://www.youtube.com/watch?v=_zSZQo7FY84&ab_channel=ONTMATLAB*
[3] *https://upload.wikimedia.org/wikibooks/pl/6/6a/C.pdf*
[4] *https://pl.wikipedia.org/wiki/Filtr_dolnoprzepustowy*
[5] *https://pl.wikipedia.org/wiki/Filtr_g%C3%B3rnoprzepustowy*
[6] *https://pl.wikipedia.org/wiki/Filtr_%C5%9Brodkowozaporowy*
[7] *https://pl.wikipedia.org/wiki/Filtr_%C5%9Brodkowoprzepustowy*
[8] *https://visualstudio.microsoft.com/pl/*
[9] *https://www.youtube.com/watch?v=v40x3EeLUZE&ab_channel=Obud%C5%BAwsobieGeeka-GeekON*
[10] *https://www.youtube.com/watch?v=_JBGV8jVcbw&ab_channel=MMCSchool*
[11] Chassaing Rulph *Digital Signal Processing and Applcatios with the C6713 and C6416 DSK*
[12] *Wyklad_01_CPS_analizawczasie.pdf*
[13] *Wyklad_02_CPS_probkowanie pdf*
[14] *Wyklad_03_CPS_transformataZ.pdf*
[15] *Wyklad_04_CPS_Analizawidmowa.pdf*
[16] *Wyklad_05_CPS_filtry.pdf*
[17] *Wyklad_06_CPS_algorytmy.pdf*
[18] Bejmert Daniel *Wykład z MIASC_filtry.pdf ~*
[19] *https://pl.wikipedia.org/wiki/Aliasing_(przetwarzanie_sygna%C5%82%C3%B3w)*
[20] *https://edu.pjwstk.edu.pl/wyklady/poj/scb/PrgGui1/PrgGui1.html*
[21] *https://pl.wikipedia.org/wiki/Cyfrowe_przetwarzanie_sygna%C5%82%C3%B3w*
[22] Zieliński T., *Cyfrowe przetwarzanie sygnałów. Od teorii do zastosowań*
[23] *https://www.ti.com/tool/CCSTUDIO*
[24] *https://www.youtube.com/watch?v=zMKX5Ci3vRg&ab_channel=drselim*
[25] *https://pl.wikipedia.org/wiki/Transmitancja_operatorowa*
[26] *https://sound.eti.pg.gda.pl/~greg/dsp/07-SpecjalneFiltry.html*
[27] *https://estradaistudio.pl/technologia/30777-wszystko-o-filtrach*

Autors:
Bartosz Golis, Politechnika Wrocławska

Jakub Janiszewski, Politechnika Wrocławska