Implementation of a hearing-impairment simulator based on TEENSY

Special Course

Supervisor:

Bastian Epp

bepp@elektro.dtu.dk

Author:

Vassili Cruchet, s172226

vassili.cruchet@gmail.com

DTU Electrical Engineering
Technical University of Denmark
Spring, 2018

Contents

1 Introduction

Baer and Moore [2] described that one of the consequences of hearing impairment is a decrease in the ability to discriminate frequency. This can be modeled by smearing the audio spectrum. They tested an algorithm on pre-recorded audio samples and were able to show a correlation between speech intelligibility and frequency smearing. The idea and goal of this project is to implement this algorithm on an embedded platform to process audio in real-time. This could then be used as a demonstrator in different events where people could wear headphones and experience this aspect of hearing impairment and the deficit in communication that comes with it.
In practice, the algorithm was first implemented in MATLAB to test it off-line with different parameters. Secondly, the algorithm was implemented on a microcontroller-based platform (TEENSY 3.6) which integrate a DSP co-processor with which complete libraries can be used.

2 Frequency smearing theory

2.1 Basic principle

As explained in [4], the ear identifies frequencies using a frequency-to-place transformation in the cochlea that could be modeled as a bank of asymmetric band-pass filters. The width of these so called auditory filters depends on the center frequency and can be modelled using a $roex(p)$ function.

$$W(g) = (1+pg)e^{-pg},\ g = \frac{|f-f_0|}{f_0}, \tag{1}$$

where g is the normalized frequency, f_0 the center frequency and p determines the tuning of the filter. Note that this model assumes a symmetric auditory filter. Moreover, one can define the equivalent rectangular bandwidth (ERB) as follows [4] :

$$\mathrm{ERB}_{roex} = \frac{4f_0}{p} \underset{\substack{\uparrow \\ \text{normal hearing}}}{\simeq} 24.7(0.000437f_0+1) \tag{2}$$

Equation 2 can then be used to estimate the parameter p and frequency smearing can be simulated by broadening of the auditory filter. This is done by dividing p by a broadening factor b. Figure 1 shows auditory filters for different levels of broadening.

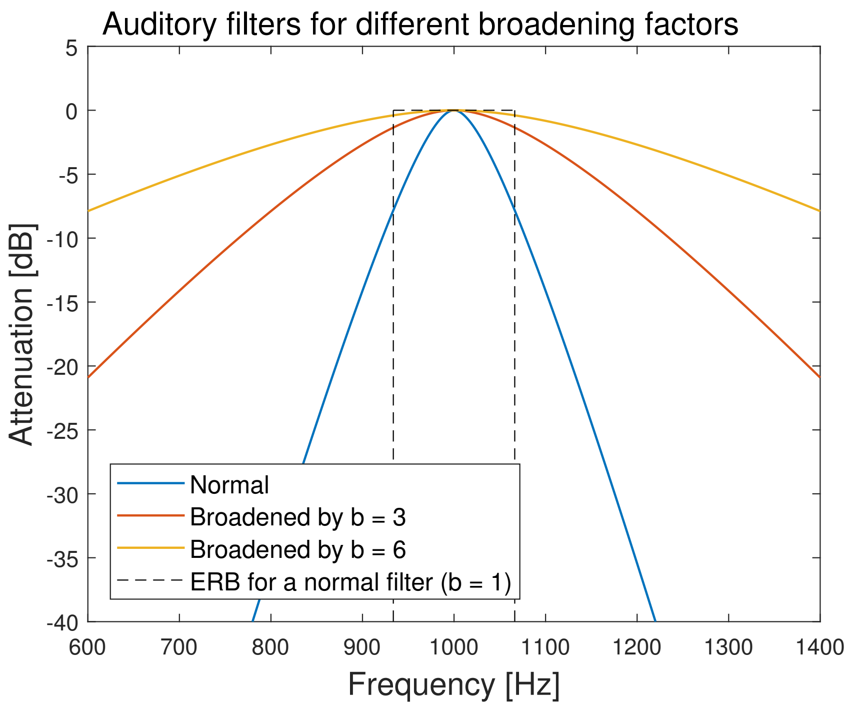

Figure 1: Auditory filter centered on $f_0 = 1kHz$

2.2 Algorithm implementation

HERE YOU MIGHT WANT TO ADD SOME MORE GLOBAL DESCRIPTION OF HOW IT WORK IN PRINCIPLE BEFORE GOING DIRECTLY INTO THE ALGORITHM. The broadening explained above is accomplished using a matrix multiplication. The power spectrum is represented in a vector where the n-th component is the power at n times the frequency resolution. When this vector is multiplied with the smearing matrix as in equation 2.2, this component is replaced by the convolution of itself with the broadened auditory filter centered on the corresponding frequency.

$$Y(n) = \sum_i A_s(n,i)X(i) \iff \boldsymbol{Y} = \boldsymbol{A_s X}, \tag{3}$$

where $\boldsymbol{X}$ and $\boldsymbol{Y}$ are the input and output spectrum vectors and $\boldsymbol{A_s}$ is the matrix containing the normalized smeared auditory filter. $A(n,i)$ is then the value of the filter centered on the n-th frequency at the i-th frequency. It is calculated as follow:

$$\boldsymbol{A_s} = \boldsymbol{A_N^{-1} A_W}, \tag{4}$$

with

$$A_N(n,i) = \frac{1}{ERB}(1+pg)e^{-pg}, \text{ and } A_W(n,i) = (1+\frac{pg}{b})e^{-\frac{pg}{b}}, \tag{5}$$

using the normalized frequency

$$g(n,i) = \frac{|f_i - f_n|}{f_n}. \tag{6}$$

As seen in equation 4, it is needed to multiply the widened filter $\boldsymbol{A_W}$ with the inverse of the normal auditory filter $\boldsymbol{A_N}$ to compensate for the listener's own auditory filter. To prevent unwanted spikes in the output spectrum, the auditory filter is also normalized by the normal hearing ERB.

Figure 2 shows the signal path when processed to be smeared. The window functions used before FFT and after IFFT are square root of hann window. The square root is needed as windowing occurs to times. This will achieve exact reconstruction with an overlap of 50%. As seen on figure 3, the smearing only affects the magnitude and the original phase is kept to preserve the time domain waveform, as suggested by Baer and Moore in [2].

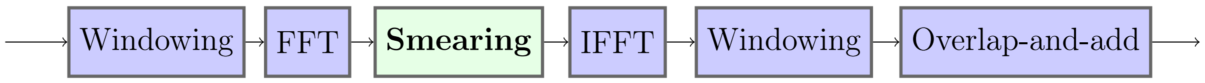

Figure 2: Diagram of the operations to perform frequency smearing. The time-domain frame is first windowed with a square-root of hann window. After FFT, the power spectrum is processed and converted back to time-domain. It is windowed again before overlap-and-add with 50% overlap.

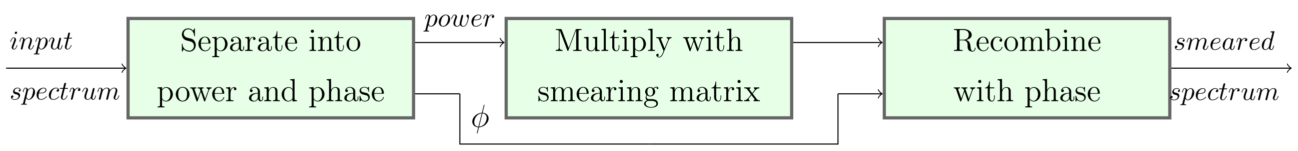

Figure 3: Diagram of the smearing algorithm. Only the power spectrum is processed so that the input and output spectrum have the same phase in order to preserve the waveform and limit noise [2].

3 *MATLAB* implementation

3.1 Functions description

3.1.1 `A_s = calc_smear_matrix(fs, N, b)`

Calculates the smearing matrix A_s according to Baer and Moore article [2]:

$$A_s = A_n^{-1} \cdot A_w$$

where A_n and A_w are normal and broadened auditory filters calculated with `calc_audit_filt.m`.

Input arguments

- f_s: Sampling frequency
- N: Length of the signal. When block processing is used, this corresponds to a frame length.
- b: Broadening factor: $b > 1$, $b = 1$ does not affect the spectrum.

Output arguments

- A_s: Smearing matrix of size $(\frac{N}{2} \times \frac{N}{2})$ to be used in `smearing.m`.

3.1.2 `A = calc_audit_filt(fs, N, b)`

Calculates the broadened auditory filter A according to Baer and Moore article [2]. See section 2 page 1 for more details.

$$A(n,i) = \frac{1}{ERB} = \left(1 + \frac{pg}{b}\right) e^{-\frac{pg}{b}}$$

Input arguments

- f_s: Sampling frequency
- N: Length of the signal. When block processing is used, this corresponds to a frame length.
- b: Broadening factor: $b > 1$, $b = 1$ does not affect the spectrum. This argument is optional, if not given, the default value is 1 (no broadening).

Output arguments

- A: Auditory filter matrix of size $(\frac{N}{2} \times \frac{N}{2})$ to be used in `calc_smear_matrix.m`.

3.1.3 `Y = smearing(X, A_s)`

Smears the power spectrum X (column vector) using the smearing matrix calculated with `calc_smear_matrix.m`. It assumes a real-valued time-domain signal and therefore that spectral component for negative frequencies are the complex conjugate of the positive ones.

Input arguments

- X: Input power spectrum to be smeared given as a column vector of length N.
- A_s: Smearing matrix calculated with `calc_smear_matrix.m` whose dimensions are $(\frac{N}{2} \times \frac{N}{2})$.[1]

Output arguments

- Y: Smeared power spectrum given as a column vector of length N.

3.2 Test scripts

Several scripts were used to test different functionalities of the program, separately or combined.

3.2.1 `test_frequency_smearing.m`

This script tests the frequency smearing algorithm for one single frame. To keep computation time low, the signal's length should not exceed 10000 samples. The second part test computing time by calculating a smearing matrix a multiplying it with a spectrum for a large number of growing different frame lengths. The time required to do these operations is measured with the function `tic toc` and plotted versus the frame length. First a test signal is synthesized.

```
fs = 16000;        % sampling frequency
T = 0.5;          % signal duration
f = 1000;         % signal fundammental frequency
t = [0:1/fs:T-1/fs];  % time vector
x = (sin(2*pi*f*t))' +  0.75*(sin(2*pi*2*f*t))' ...
  +  0.5*(sin(2*pi*4*f*t))';
```

Then smearing matrix is calculated and the power spectrum is smeared.

```
% calculate smearing matrix and output power spectrum
A_s = calc_smear_matrix(fs, 1, b);
Y = smearing(abs(X), A_s);
```

[1]The size is $(\frac{N}{2} \times \frac{N}{2})$ and not $(N \times N)$ because only the positive frequencies have to be processed. The negative ones are their complex conjugate.

IFFT is used to recover a time-domain signal than can be analysed as shown in figures 4.

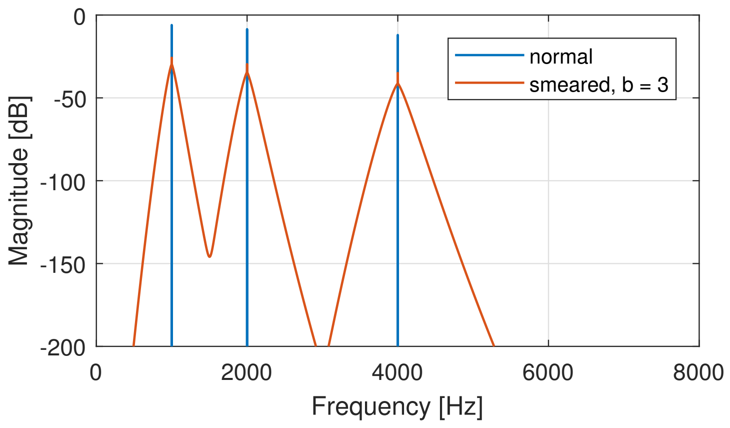

Figure 4: Spectrum of the smeared signal. It is clear that each spikes of the input spectrum is smeared with the shape of the auditory filter. One can notice that the effect is more important at higher frequencies as auditory filter is wider.

3.2.2 `frequency_smearing.m`

This script loads an audio file and apply frequency smearing to its spectrum. Input signal is processed frame by frame and reconstructed using overlap-and-add method. The overlap is 50% what should achieve perfect recombination with two square-root of hann windows. Note that the `'periodic'` argument is used for correct overlap-and-add. Time- and frequency-domain outputs are shown in figures 5 and 6, and one can see that overlap-and-add method adds some noise in the spectrum. It will be discussed more in section 3.3.

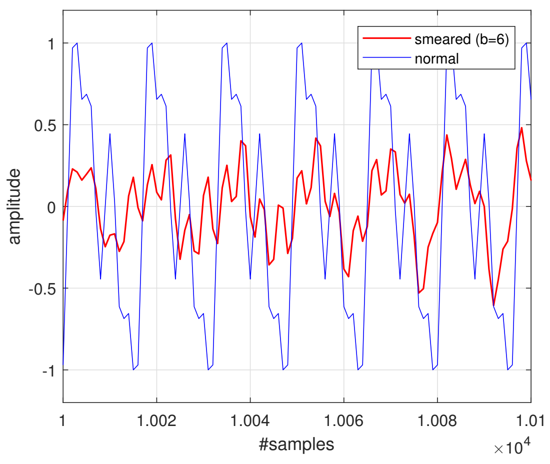

Figure 5: Time domain smeared signal.

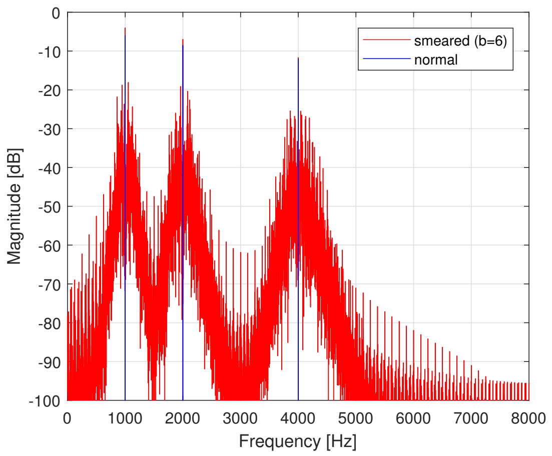

Figure 6: Spectrum of the smeared signal.

3.3 Comparison with Bear and Moore results [2]

To replicate the test presented in FIG.2 and FIG.4 of Baer and Moore's article, the following procedure was implemented:

- A test signal containing the word *"now"* (the vowel /æ/ was used in the article) was sampled at $f_s = 16kHz$ and low-pass filtered at $7kHz$ with the function `FIR_eq`.
- Frame length is 128 samples and a Hamming window is used.
- Each frame is zero-padded with 64 zeros at both ends to increase spectral resolution.

Figures 7 and 8 are replicas of FIG.2 and FIG.4 of the article. It is interesting to notice that the overlap-and-add brings some details back in the spectrum and thus potentially decrease the effect of the spectrum smearing.
In general those results are similar to the ones presented in [2]. Each frame is correctly smoothened and the effect of overlap-and-add described above is also present. This suggest that the implemented algorithm is operating correctly.

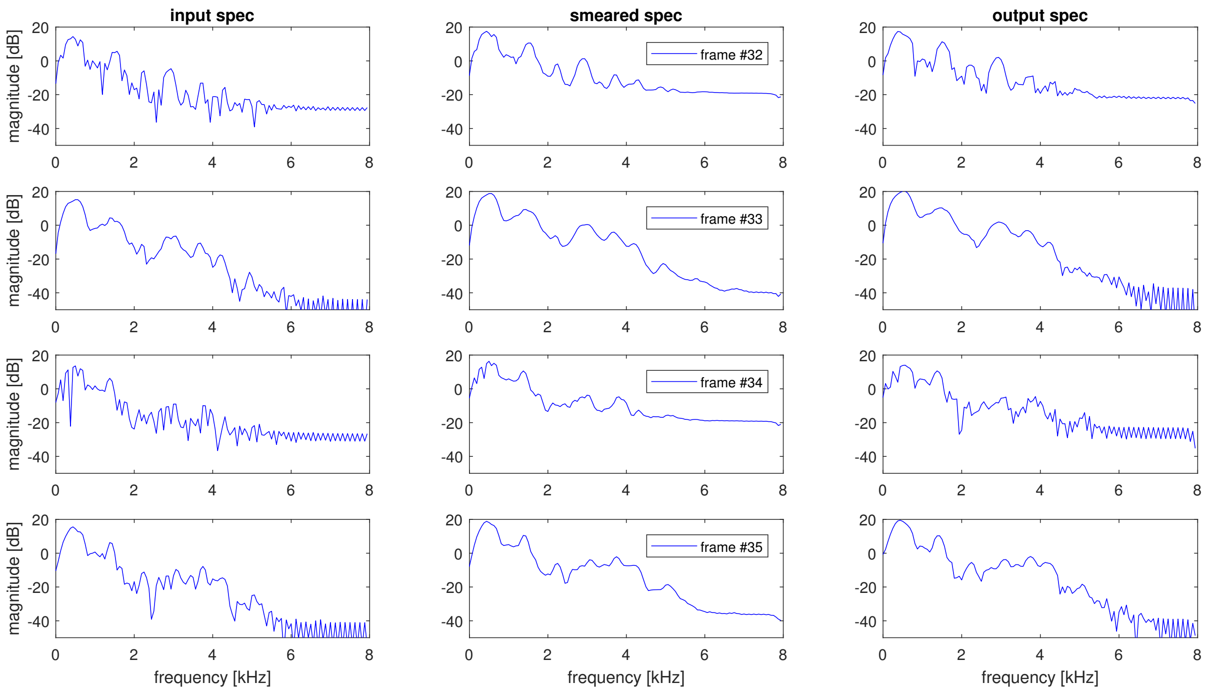

Figure 7: Spectrum of a few frame before frequency smearing (left column), after frequency smearing (middle column) and after overlap-and-add (right column).

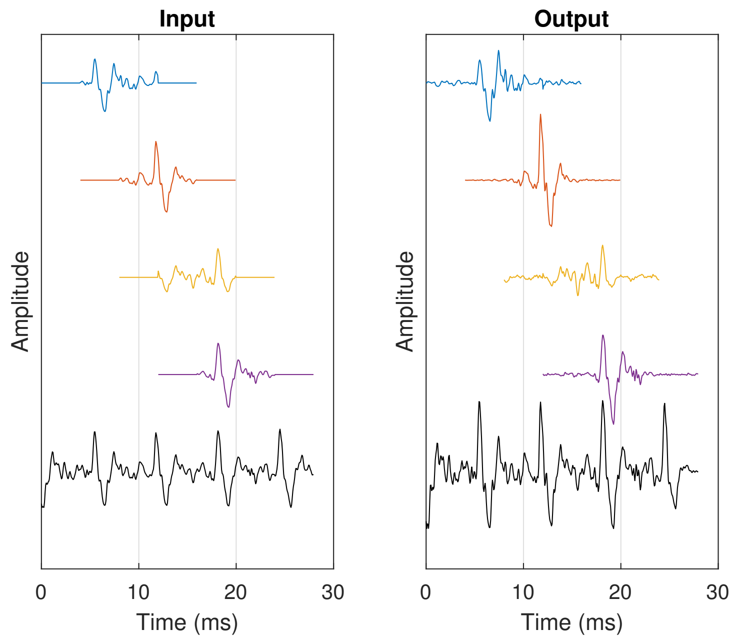

Figure 8: Time-domain representation of the zero-padded frames before and after frequency smearing. The black signal is the input signal (on the left) and the overlapped signal (on the right).

3.4 Scripts and functions used for C-code generation

As the smearing matrix is unique for given parameters, it is first calculated in MATLAB and hard coded in the memory of the TEENSY by simply defining arrays. As memory usage is crucial in an embedded system, the row-indexed sparse storage method was used to store the matrix's coefficient in a more judicious way.

3.4.1 Row-indexed sparse storage

The smearing matrix as calculated with `calc_smear_matrix.m` reach very considerable size, depending of the frame length used. For instance, if $N = 256$ samples the smearing matrix has $\left(\frac{N}{2}\right)^2 = 128^2$ elements which correspond to 65.5kB of memory if stored in floating point format. This occupies already a quarter of the TEENSY 3.6 RAM memory (256kB), leaving less space for buffers and other variables. However due to the exponential shape of the auditory filters, most of the matrix's elements are very small relative to the peak (figure 9) resulting in a band diagonal shaped sparse matrix. It is then possible to use the *row-indexed sparse storage mode* described in chapter 2 of [1] that only stores the non-zero elements. This compression is done with the script `compress_matrix.m`, adapted from [1]. The gain of memory depends on the smearing factor b and the threshold under which a element is assumed to be null.

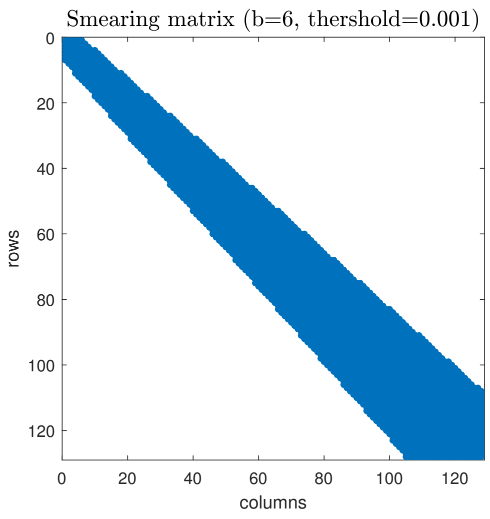

Figure 9: Non-zero elements for a 128 by 128 smearing matrix (all values smaller than the threshold are set to 0). With the row-indexed sparse storage, only 42.2% of memory is used compared to storing the whole matrix.

3.4.2 `generate_smear_matrix.m`

This scripts prints the coefficient in a text file using either row-indexed sparse storage or normal storage using two-dimensional arrays. The content of this file can then be copy-pasted in the C-code (in `smear_mat.h`) to declare the arrays containing the smearing coefficient corresponding to the desired parameters. For instance the following parameters:

```
fs = 16e3;      % sampling frequency
N  = 128;       % frame length
b  = 6;       % smearing coefficient
tol = 1e-3;     % thershold used for compressed storage
compressed = 1;   % 1 -> row-indexed sparse storage
          % 0 -> classical two-dimensional array
```

will output this line of code for normal storage:

```
float smear_mat_b6[2][2] = {{1.000000, 0.000000},
{0.000000, 1.000000}};
```

and those two lines for row-indexed sparse storage:

```
unsigned int ija_b6[3] = {3, 3, 3};
float sa_b6[2] = {1.000000, 1.000000};
```

In this particular example the length is so small that the classical storage is preferable, nonetheless it does not correspond to an real situation.

4 Implementation on *TEENSY* board

4.1 Hardware description

Platform used is a TEENSY 3.6 board[1], that uses an ARM Cortex-M4F is a 32 bit processor, clocked at 180MHz.
It is combined to the TEENSY audio shield[2] is used to provide audio interface in a high 16 bits quality. In addition to this, peripheral components as a small electret microphone[3] or a jack cable and push-button are used. Figure 10 illustrates how they should be connected to the board.

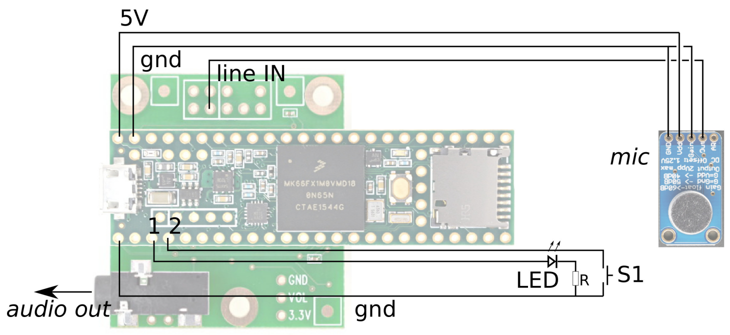

Figure 10: Wiring schematic to use the microphone. A mono jack cable can also be used by connecting it between *lineIN* and *gnd*. R should set the maximal current to 10mA.

4.2 Software description

The code can be divided according to the different tasks that need to be executed:

- **Real-time block processing**. This take part of the whole data flow, from fetching data from the audio shield, managing input and output circular buffers and sending the data back to the audio shield. The `Audio.h` library [7] and the `CircularBuffer.h` library [5] are used in this part.
- **FFT and IFFT transforms**. The CMSIS DSP library [6] is used for all transforms functions as well as for vector operations. Note that for compatibility with the audio library an older version (1.1.0) must be used. Therefore it is not needed to install the full library, one can simply include `arm_math.h` that comes with the installation of TEENSY.

[1]`https://www.pjrc.com/store/teensy36.html`
[2]`https://www.pjrc.com/store/teensy3_audio.html`
[3]`https://www.adafruit.com/product/1713`

- **Smearing algorithm**. The smearing matrices are calculated in MATLAB and hard-coded in the memory. Therefore the smearing function is simply a matrix multiplication. However to save space in the limited memory, the *row-index sparse storage method* is used.

4.2.1 Real-time block processing

During the execution, audio data pass through several buffer under different formats as illustrated in figure 11. Most important functions are in the files `circ_buff_util.h/cpp`.

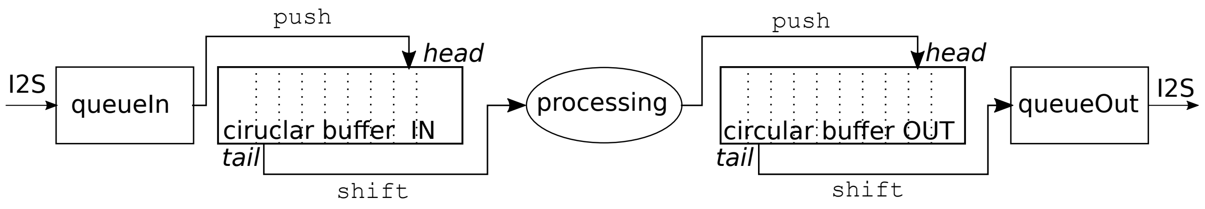

Figure 11: Data flow through the different buffers.

1. Audio signal is sampled by the audio shield and theses data are fetched with the `AudioRecordQueue` block from the Teensy audio library. It gives a pointer on a `QUEUE_LEN` samples array of 16 bits integers.

2. It is converted into a floating point with the function `read_array_from_queue()` and pushed at the head of the input circular buffer.

```
if(read_array_form_queue(arrayIn, QUEUE_LEN, &queueIn)) {
  // copy input signal in arrayIn in blocks
  // copy elements in buffer
  for(i=0; i<QUEUE_LEN; i++) {
    buffIn.push(arrayIn[i]);
  }
}
```

3. A frame of `FFT_LEN` samples is read at the tail of the circular buffer with 50% overlap using `read_array_form_buffer()` and converted into a complex array according to the CMSIS convention: `frameC[] = {real[0], imag[0], real[1], imag[1], ...}` and windowed. It is now ready for FFT and processing.

4. The real part of the processed complex frame is windowed and pushed to the head of the output circular buffer, overlapping the last half of the previous frame using the function `overlap_add()`.

5. Blocks of `QUEUE_LEN` samples are read at the tail of the buffer, converted back to 16 bits integer and outputted to the audio shield's headphone line via the *AudioPlayQueue* block.

4.2.2 Smearing algorithm

As explained in section 2 the smearing consist in calculating the power spectrum, multiply it with the pre-calculated matrix and convert it back to real and imaginary part. This is done with the functions written in `smearing.h/cpp`. The following code extract illustrate it for classical storage.

```
    for(i=0; i<N/2; i++) {
      row = smear_mat_b6[i];
      arm_dot_prod_f32 (row, spec_power, N/2, &tmp);
      frame[2*i]    = tmp * cos(spec_phase[i]);          // real part
      frame[2*i+1]  = tmp * sin(spec_phase[i]);          // imaginary part
    }
```

The particularity of these files is that not everything is compiled, depending in the constant parameters defined in `global.h`. This is needed because only the necessary smearing matrix is declared in `smear_mat.h`.

Table 1 summarizes elapsed time for execution of different tasks. With the same parameters, the duration of FFT/IFFT and smearing process should not excess **7.2**μs to avoid audible cuts in the sound.

task	approximative duration
in-to-out buffer path	200 ns
FFT/IFFT	500 ns
smearing	5.8 μs
total	**6.5** μs

Table 1: Example of processing times for $f_s = 16kHz$ and a frame length of `FFT_LEN = 256`.

4.3 Functions description

4.3.1 `read_array_form_queue`

Arguments

- `float arrayIn[]`. Pointer to the array to be filled with samples.
- `int array_length`. Length of the array. It must be a multiple of QUEUE_LEN. If a smaller size is wanted, QUEUE_LEN can be changed in `AudioStream.h`.
- `AudioRecordQueue* queueIn`. Pointer to the input queue that provide data form the audio shield.

Description
This function converts and copies input signal fetched form the input queue into a floating point array. Returns 0 if the dimensions mismatch or if the queue does not have enough samples.

4.3.2 `read_frame_from_buffer`

Arguments

- `float frame[]`. Pointer to the array to be filled in with elements form the input circular buffer `buffIn`.
- `int frame_l`. Length of the frame.

Description
This function copies `frame_l` samples form the input circular buffer and leaves the last half of them in the buffer for the next frame. It results in 50% overlap.

4.3.3 `overlap_add`

Arguments

- `float frame[]`. Array of elements (real-valued) to be added in the output circular buffer `buffOut`.
- `int array_length`. Length of the frame.

Description
This function pushes the elements of the frame to the head of the output circular buffer, overlapping the last half of the previous frame.

4.3.4 `smearing_uncomp` and `smearing_comp`

Arguments

- `float32_t* frame`. Pointer on the *complex* vector to be smeared.
- `int N`. Number of complex elements in the vector.

Description
Both functions process frequency smearing according to the parameters defined in `global.h`. `smearing_uncomp` uses a classic matrix multiplication and `smearing_comp` uses row-indexed sparse storage multiplication done in `sprsax`.
Note 1: Only one of these two functions is compiled, depending if the symbol `COMPRESSED` is defined.

Note 2: They are written to work with $B = 3$ and $B = 6$. If different options are wanted, new compiling directive must be written with the corresponding smearing matrix written in `smear_mat.h`:

```
#elif B == NEWVALUE
      for(i=0; i<N/2; i++) {
        row = smear_mat_bNEWVALUE[i];
        arm_dot_prod_f32 (row, spec_power, N/2, &tmp);
        frame[2*i]    = tmp * cos(spec_phase[i]);          // real part
        frame[2*i+1]  = tmp * sin(spec_phase[i]);          // imaginary part
      }


waveform1.frequency(AUDIO_SAMPLE_RATE_EXACT/FS*freq); // where freq is the desired
    frequency in Hz
```

4.3.5 `sprsax`

Arguments

- `float sa[]` and `unsigned int ija[]`. Arrays containing the non-zero coefficient of the smearing matrix and indexes needed for the row-indexed sparse storage. The first element of `ija` must be equal to n+1.
- `float x[]`. Vector to be multiplied.
- `float b[]`. Output vector.
- `unsigned int n`. Length of the vector $\boldsymbol{x}$ and $\boldsymbol{b}$.

Description

This functions process the vector-by-matrix multiplication in equation 7 where $\boldsymbol{A}$ is stored with row-indexed sparse storage. It is directly adapted from [1].

$$\boldsymbol{b} = \boldsymbol{A}\boldsymbol{x} \tag{7}$$

4.3.6 `neg_freq`

Arguments

- `float32_t* frame`. Input and output complex spectrum where frequencies are sorted as:

$$[0Hz, \Delta_f, ..., (f_{max} - \Delta_f), f_{max}, -f_{max}, (-f_{max} + \Delta_f), ..., -\Delta_f]$$

- `int N`. Number of complex elements in frame.

Description

This function reconstructs spectrum's negative frequencies with complex conjugate of the positive one.

4.3.7 `fft_init`

Arguments

- `arm_cfft_radix2_instance_f32* fftInst`. Pointer to the instance of the floating-point CFFT/CIFFT structure to be initialized.
- `uint16_t fftLen`. Length of the FFT/IFFT in number of complex elements.
- `uint8_t fftFlag`. Flag that selects forward (fftFlag=0) or inverse (fftFlag=1) transform.
- `uint8_t bitReverseFlag`. Flag that enables (bitReverseFlag=1) or disables (bitReverseFlag=0) bit reversal of output. It is always enabled in this program.

Descritption
This function initializes the FFT and IFFT instances using the CMSIS function `arm_cfft_radix2_init_f32`[1] and checks for errors.
Note 1: The CMSIS documentation specifies that the supported length are `fftLen` = 16, 64, 256, 1024. However on one thread of the TEENSY forum[2] one can read:

> "The radix2 ones offer 16, 32, 64, 128, 256, 512, 1024, 2048 sizes. On Cortex-M4 for Q15, the raxid4 versions is slightly faster, which is the reason it was used with the audio library."

what would suggest more flexibility towards possible length. It was successfully tested for `FFT_LEN` = 128 and 512.
Note 2: `FFT_LEN` should always be larger than `QUEUE_LEN`. If a smaller value than 128 samples is wanted, `AUDIO_BLOCK_SAMPLES` has to be changed in AudioStream.h using multiples of 16 samples.

4.3.8 `set_periph`

Description
This function initialize the SD card, the buttons, the audio shield and set the I2S sampling frequency. Its only purpose is encapsulation to improve code readability. To simplify inclusions and parameters it is directly written in the `.ino` file and not in `util.h/cpp`.

[1]`https://www.keil.com/pack/doc/CMSIS/DSP/html/group__ComplexFFT.html`
[2]`https://forum.pjrc.com/threads/35277-arm_math-h-and-the-FFT-audio-blocks`

4.3.9 `setI2SFreq`

Arguments

- `int freq`. Wanted sampling frequency, possible values are (in Hertz): 8000, 11025, 16000, 22050, 32000, 44100, 44117 , 48000, 88200, 88234, 96000, 176400, 176468 and 192000.

Description

This function adapts the audio shield's I2S in- and output sampling frequency. It is copied from the TEENSY forum[1].

Note: The sampling frequency is only adapted for the I2S. The rest of the audio library still works on $44.1kHz$. Therefore, if other modules as the waveform generator are used their frequency must be scaled:

```
waveform1.frequency(AUDIO_SAMPLE_RATE_EXACT/FS*freq); // where freq is the desired
    frequency in Hz
```

4.3.10 `read_button`

Description

This function check if the button connected to BUTTON1_PIN has been pushed and (des-)activates frequency smearing. If more buttons are needed some code can easily be added following the same 3.

4.3.11 `create_hann_window` and `create_sqrthann_window`

Arguments

- `float win[]`. Pointer to the window vector to be initialized.
- `int win_l`. Length of the window.

Description

Those two function create a *hann* or $\sqrt{hann}$ window respectively. It uses the AudioWindowHanning256 defined in Audio.h, therefore 256 is the only possible length. More flexibility could be added by defining local variables with window coefficients for other lengths.

[1]`https://forum.pjrc.com/threads/38753-Discussion-about-a-simple-way-to-change-the-sample-rate`

5 Limitations, improvements and future work

Even if the whole simulator is working, some features need to be improved or added to it. Important points are, among others, available processing time, memory usage, user interface and flexibility.

5.1 Processing time

The biggest challenge regarding processing time is the ability to run the program in real-time with a continuous output. As presented in section 4.2.2 on page 12, transforming the frame and processing frequency smearing takes approximately $6.3\mu s$ for a frame length of 256 samples using row-indexed storage. This time would obviously be decreased by reducing the frame length, however this leaves less time for processing between two consecutive frames. This is why the sampling frequency is reduced from $44.1kHz$ to $16kHz$, with the consequence to loose the high frequencies of the processed audio. Nevertheless one could discuss the importance of this inconvenient as hearing impairment also induces losses of high frequencies, even to a lower level.

5.2 Memory usage, operation flexibility and characterization

With *only* 256 kbytes of RAM, storing huge smearing matrices and long buffers is not possible on the embedded platform. This is why the approach of compiling only the useful code using directive was taken. Unfortunately this reduces a lot the possibilities of changing parameters while the program is executing like smearing coefficient or even frame length. Moreover the implementation of row-indexed sparse storage would allow to have a few different matrices stored at the same time. Implementing different coefficients and the ability to switch between them on-line would be a big improvement but it implies a important restructuring of the code in `smear_mat.h` and the smearing function.
It would also be very interesting to be able to vary the frame length, even if it is not on-line, in order to investigate the performance of the algorithm for different frequency resolution (the tests on MATLAB showed that the smearing is more noticeable with high frequency resolution, thus longer frames). To do so, some more hann window have to be implemented as explained in section 4.3.11.
A more detailed and rigorous characterization of the embedded algorithm needs to be done by recording a processed audio file and analyze it to compare with the same file

processed in MATLAB. In order to do so, the audio shield's line out has to be used instead of the headphone output.

5.3 Hardware improvement

Following the objective of a demonstration kit, a better and more "finished" hardware implementation is needed, including adjustable volume and/or gain, buttons to activate smearing and change smearing coefficient. This last point could even be done using wireless command or some timer so that it is not directly commanded by the user. An ideal implementation would be integrated in some headphones.
Adding a low-pass filter with adjustable cutoff frequency would approach real hearing impairment more realistically as would stereo processing.

5.4 Algorithm improvement

In their article [2], Baer and Moore point that asymmetrical broadening around the center frequency has a stronger effect on speech ineligibility. It is also closer to the actual broadening that happens in hearing impaired subject. To implement this, the MATLAB functions have to be modified using different factor for each part of the spectrum.

5.5 TODO list

The points described above can be summarized in a task list:

- Possibility to change smearing coefficient in-line.
- Implement more different window lengths to investigate effect of frequency resolution.
- Record proper audio samples for further analysis.
- Add adjustable low-pass filter.
- Implement stereo processing.
- Implement asymmetrical broadening.
- Improve hardware packaging and user interface to have an operational demonstration kit.

6 Conclusion

References

[1] William H. Press et al. *Numerical Recipes in C: The Art of Scientific Computing, Second Edition*. English. 2 edition. Cambridge ; New York: Cambridge University Press, Oct. 1992. ISBN: 978-0-521-43108-8.

[2] Thomas Baer and Brian C. J. Moore. "Effects of Spectral Smearing on the Intelligibility of Sentences in Noise". en. In: *The Journal of the Acoustical Society of America* 94.3 (Sept. 1993), pp. 1229–1241. ISSN: 0001-4966. DOI: `10.1121/1.408176`.

[3] Dirk Mauler and Rainer Martin. "A Low Delay, Variable Resolution, Perfect Reconstruction Spectral Analysis-Synthesis System for Speech Enhancement". en. In: (2007), p. 5.

[4] Andreas Spanias, Ted Painter, and Venkatraman Atti. *Audio Signal Processing and Coding*. en. Hoboken, NJ, USA: John Wiley & Sons, Inc., Jan. 2007. ISBN: 978-0-470-04197-0 978-0-471-79147-8. DOI: `10.1002/0470041978`.

[5] Roberto Lo Giacco. *CircularBuffer: Arduino Circular Buffer Library*. `https://github.com/rlogiacco/CircularBuffer`. Apr. 2018.

[6] *CMSIS DSP Software Library*. `https://www.keil.com/pack/doc/CMSIS/DSP/html/index.html`.

[7] *Teensy Audio Library, High Quality Sound Processing in Arduino Sketches on Teensy 3.1*. `https://www.pjrc.com/teensy/td_libs_Audio.html`.