Proiect 3
Efecte audio digitale bazate pe linii de întârziere

Predună Tudor-Gabriel
Enache George-Vlad
Grupa 441G

Cuprins

Note generale despre implementare

Codul nostru se bazează pe următoarea structura:

```
 9 typedef struct buf{
10     short BUFFER[4000];
11     short indiceBuffer;
12     short delaySamples;
13 } bufferObject;
14
15
16 bufferObject buffer;
17
18 void append(short a, bufferObject *buffer)
19 {
20     buffer->BUFFER[buffer->indiceBuffer % buffer->delaySamples] = a; //din cauza asta crapa
21     buffer->indiceBuffer++;
22 }
23
24 short dequeue(bufferObject *buffer)
25 {
26     return buffer->BUFFER[buffer->indiceBuffer % buffer->delaySamples];
27 }
28
29
30
31 short reverberatingDelay(short x, short dry, short wet, short g, short delay_ms, bufferObject *buffer)
32 {
33     buffer->delaySamples = 44.1 * delay_ms;
34     short s1 = sub(WORD16(0.999), g);
35     buffer->indiceBuffer %= buffer->delaySamples;
36     short popat = dequeue(buffer);
37     append(add(mult(mult(x, wet), s1), mult(popat, g)), buffer);
38     return add(mult(x, dry), popat);
39 }
40
41
```

Avem o structura numita bufferObject al cărei unic scop este sa tina eșantioanele alocate unei structuri recursive, fie ea filtru All Pass sau Reverberating Delay. Sigur, puteam sa optimizam codul mai tarziu sa folosim același buffer, pentru ca operațiile se fac secvențial pană la urma, dar nu am mai ajuns acolo.

Acest bufferObject are ca membrii vectorul BUFFER de lungime fixă 4000, unde ne vom tine eșantioanele întârziate, indiceBuffer, care arata “coada” lui BUFFER și delaySamples, care este un număr < 4000 și care este numărul de eșantioane cu care vrem sa intarziem semnalul. Numărul de 4000 este ales arbitrar, dar l-am ales de așa natura sa ne permita o intarziere maximă de aproape 90ms, la un Fs = 44.1Khz. Sigur, acesta poate fi modificat fără probleme, însă pentru experimentele noastre e suficient ca sa “auzim” ceva.

De asemenea, acest bufferObject are și două metode, append, care adaugă la coada BUFFER cel mai recent eșantion din semnalul de intrare, și deque, care întoarce semnalul din “varful” cozii. Practic, bufferObject și metodele aferente sunt doar o coada.

Din cauza blocajelor intelectuale pe care le-am avut în a implementa filtrul Schroeder și filtrul Moorer, am decis să le implementăm fără a face corect convoluția. Adică, unde la celelalte filtre recursive semnalul de ieșire era de lungime N+M, unde N e lungimea inițială si M intarzierea totala, aici toate semnalele sunt de lungime N.

De asemenea, am uitat funcția main; funcția main va fi aproximativ la fel al toate programele. Arată așa:

```
44
45 int main()
46 {
47     FILE *input = fopen("intrare.dat", "r+b");
48     FILE *outputMoorer = fopen("iesireReverberating2.dat", "w+b"); //experimentam cu coada
49     short x, temp;
50     printf("befor while\n");
51     int i = 0;
52
53     while(fscanf(input, "%hd", &x) != EOF)
54     {
55         fprintf(outputMoorer, "%hd ", reverberatingDelay(x, WORD16(0.2), WORD16(0.7), WORD16(0.8), 90, &buffer));
56
57     }
58
59     fclose(input);
60     fclose(outputMoorer);
61     return 0;
62
63 }
```

În afara de cateva lucruri de debug rămase, ideea e simpla; citește eșantion cu eșantion pana se termina numerele din fisier si printeaza esantion cu esantion rezultatul funcției apelate.

La efectele simple, ca sa iasa convolutia corect, doar am padat cu 0-uri la intrare semnalul; adică. am hrănit funcției unde e implementat filtrul un număr de 0-uri egal cu intarzierea totala, ca sa se goleasca bufferul filtrului.

1. Efecte audio folosind linii de întârziere cu structură nerecursivă (FIR)

Efectele audio bazate pe linii de întârziere cu structură nerecursivă reprezintă un bun punct de start pentru acest proiect, deoarece oferă o implementare destul de ușoară. Astfel de efecte pot fi realizate utilizând o linii de întârziere constante și factori multiplicativi pentru a altera amplitudinea semnalului.

1.1 Ecoul simplu

Este cea mai simpla implementare a unui efect audio utilizând linii de întârziere. Pentru aceasta, am utilizat direct configurația ce permite scalarea semnalului, pentru a preveni depășirile:

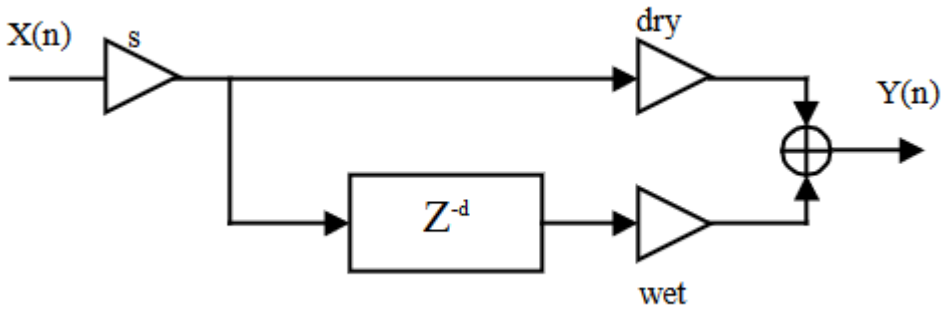


Fig.1 Ecou Simplu cu scalare

Implementare:

```
Word16 simpleDelay(Word16 x, Word16 dry, Word16 wet, Word16 scale ,Word16 delay_ms, bufferObject *buffer){
    buffer->delaySamples = 44.1*delay_ms;

    buffer->indiceBuffer %= buffer->delaySamples;
    short popat = dequeue(buffer);

    append(mult(x, scale), buffer);
    return add(mult(mult(x, scale), dry), mult(popat, wet));
}
```

Funcția simple_delay() primește ca parametrii un sample al semnalului, coeficienții multiplicativi (dry, wet și scale), valoarea întârzierii în secunde și o structură de tip buffer. Implementarea schemei bloc se face astfel:

A. Când buffer-ul a fost umplut, se va returna valoarea semnalului original înmulțit cu coeficienții dry și scale, adunat cu ultimul element din coadă, înmulțit cu coeficientul wet;
B. Cât timp buffer-ul nu a fost încă umplut, se va returna doar valoarea semnalului original înmulțit cu coeficienții dry și scale, din moment ce elementul popped din buffer va fi 0, deoarece acesta este declarat global.

Obținem o suprapunere perfectă a semnalului de control generat în MATLAB și acelui ieșit din SC140:

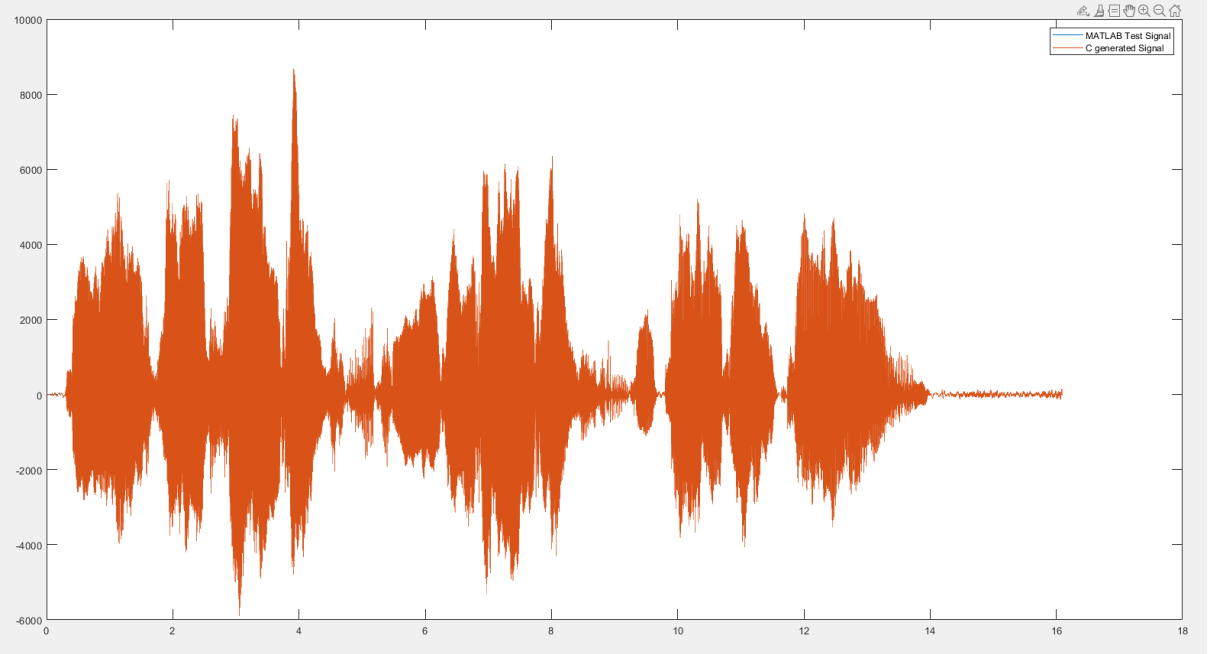


Fig.2 Semnal de ieșire suprapus cu simularea MATLAB

1.2 Early Echoes

O utilizare mai complexă a unui efect de tipul simple delay poate fi ecoul multiplu (Early Echoes). Aceasta pornește de la simularea reflexiilor inițiale. Pentru aceasta, am utilizat direct configurația optimizată pentru consum minim de memorie, ce permite scalarea semnalului, pentru a preveni depășirile:

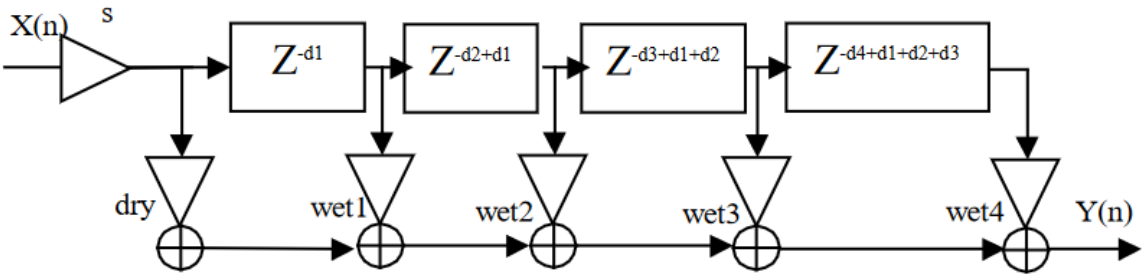


Fig.3 Schema bloc Early Echoes

Unde s, coeficientul de scalare, a fost calculat după formula:

$$|y(n)| \leq |dry| + \sum_{i=1}^{4} |wet_i| = k$$

$$s = \frac{1}{k} = \frac{1}{|dry| + \sum_{i=1}^{4} |wet_i|}$$

Implementare:

```
Word16 early_echoes(Word16 x, bufferObject *buffer){

    buffer->delaySamples = intarzieri[3];
    buffer->indiceBuffer %= intarzieri[3];
    append(mult(x,scale), buffer);

    int i, suma = 0;
    for (i = 0; i < 4; i++){
        if (buffer->indiceBuffer - intarzieri[i]<0){
            suma = add(suma, mult(buffer->BUFFER[buffer->indiceBuffer  + intarzieri[3] - intarzieri[i]], wet[i]));
        }
        else {
            suma = add(suma, mult(buffer->BUFFER[buffer->indiceBuffer - intarzieri[i]], wet[i]));
        }
    }

    Word16 out = add(mult(mult(dry, x),scale), suma);
    return out;
    }
```

Similar cu funcția simple_delay(), early_echoes primește ca parametrii un sample al semnalului și o structură de tip buffer. Coeficienții multiplicativi (dry, wet și scale), cât și valorile întârzierii în samples sunt variabile declarate global. În loc să se utilizeze 4 buffers înseriate, se va utiliza unul singur, însă fiecare dintre cele 4 buffers va avea un coeficient wet și un delay individual. Implementarea schemei bloc începe cu adăugarea elementului x înmulțit cu coeficientul de scalare în buffer. După aceasta, urmează punerea în cod a următoarei formule:

$$y(n) = s \cdot dry \cdot x(n) + s \cdot \sum_{i=1}^{4} wet_i \cdot x(n - d_i)$$

Cât timp poziția indexului curent este mai mică ca 0, practic rezultatul sumei va reprezenta doar semnalul de intrare înmulțit. Odată ce se epuizează lungimea asociată “primului buffer”, la rezultatul final se va adăuga suma sample ului întârziat cu wet-ul corespunzător.

Funcția Early Echoes este o adaptare particulară a funcției reflexii inițiale ce este utilizată mai târziu în modelul Moorer. Diferențele constau în coeficienții wet introduși, scalările diferite aplicate și numărul redus de buffers utilizat, dar principiul de funcționare este același. Inițial am încercat o implementare ce utilizează un buffer individual pentru fiecare buffer descris în diagrama bloc, însă această implementare s-a dovedit a fi deosebit de ineficientă și greoaie. Astfel, am trecut la metoda curentă, ce implementează un singur buffer de lungime maximă.

Obținem singura suprapunere aproape perfectă din acest proiect, a semnalului de control generat în MATLAB și acelui ieșit din SC140:

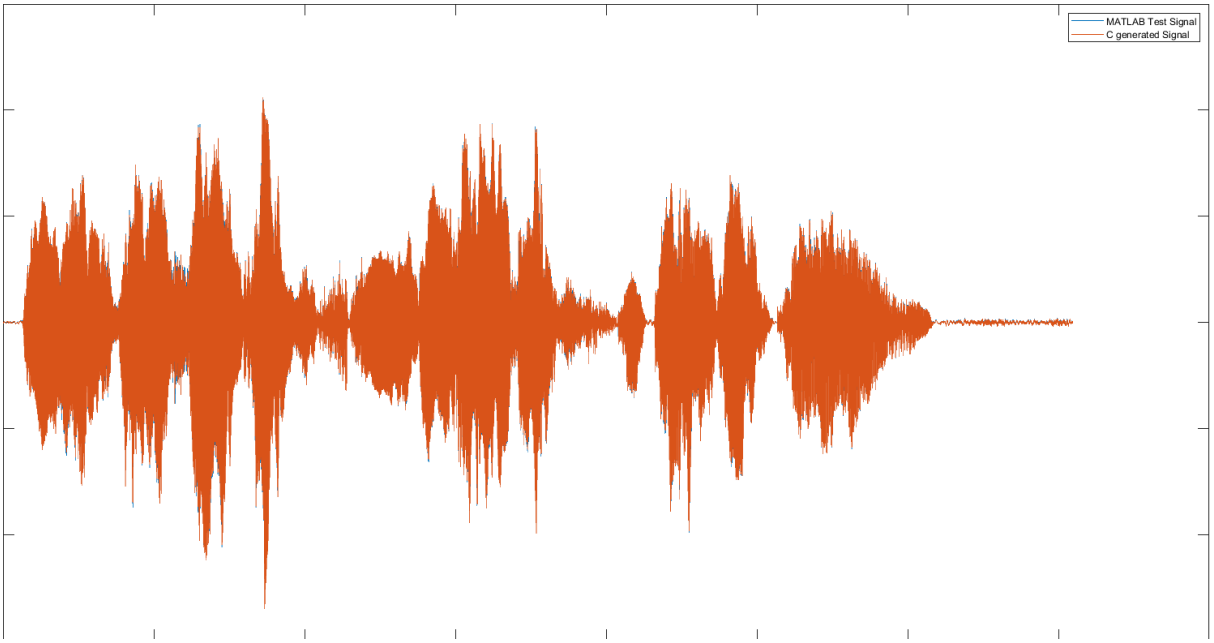


Fig.4 Semnal de ieșire suprapus cu simularea MATLAB

Pentru a vă face o idee despre eroare, am atașat și o poza cu zoom in:

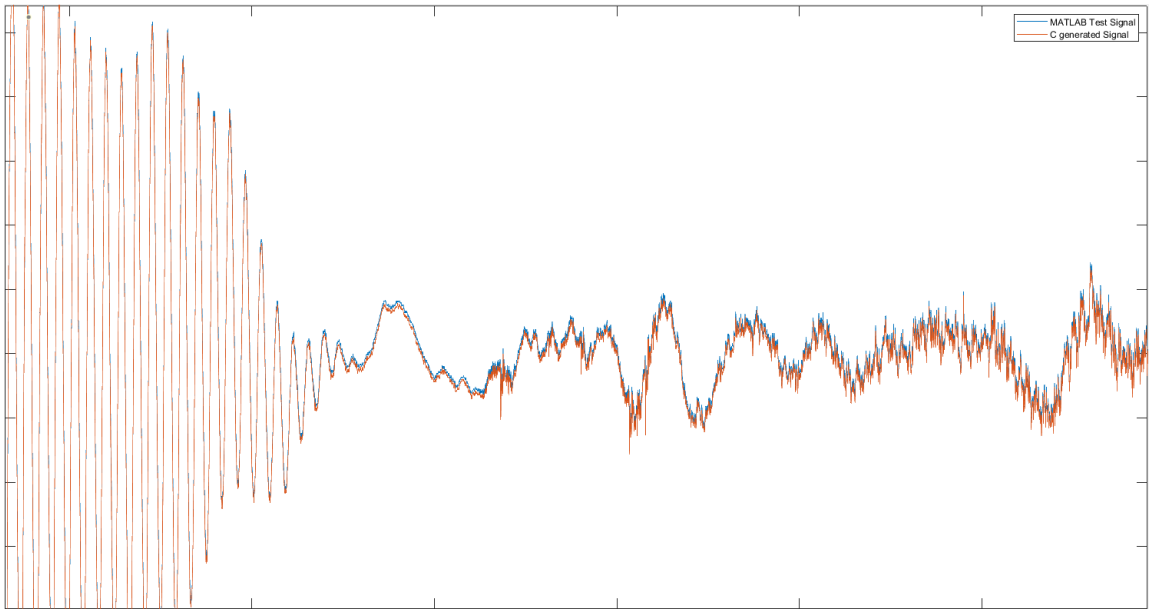


Fig.5 Suprapunere zoomed in

2.Efecte audio folosind linii de întârziere cu structură recursivă (IIR)

2.1 Reverberating delay

Am folosit această structură:

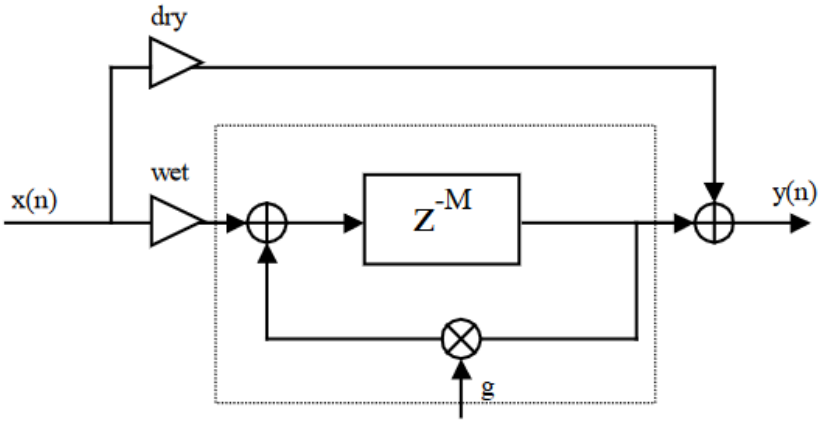


Fig.6 Schema bloc Reverberating Delay

Am observat de asemenea că filtrul pieptene (Comb), poate fi implementat cu un Reverberating Delay, unde dry = 0.
Funcția reverberating delay, implementată mai jos, nu face decat sa implementeze formula :

$$y(n) = dry \cdot x(n) + wet \cdot s_1 \cdot x(n) * h(n) \qquad (3.29)$$

```c
short reverberatingDelay(short x, short dry, short wet, short g, short delay_ms, bufferObject *buffer)
{
    buffer->delaySamples = 44.1 * delay_ms;
    short s1 = sub(WORD16(0.999), g);
    buffer->indiceBuffer %= buffer->delaySamples;
    short popat = dequeue(buffer);
    append(add(mult(mult(x, wet), s1), mult(popat, g)), buffer);
    return add(mult(x, dry), popat);
}




int main()
{
    FILE *input = fopen("intrare.dat", "r+b");
    FILE *outputMoorer = fopen("iesireReverberating2.dat", "w+b"); //experimentam cu coada
    short x, temp, delay = 90;
    printf("befor while\n");
    int i = 0;

    while(fscanf(input, "%hd", &x) != EOF)
    {
        fprintf(outputMoorer, "%hd ", reverberatingDelay(x, WORD16(0.2), WORD16(0.7), WORD16(0.8), delay, &buffer));

    }
    while(i < 44.1 * delay)
    {
        fprintf(outputMoorer, "%hd ", reverberatingDelay(0, WORD16(0.2), WORD16(0.7), WORD16(0.8), delay, &buffer));
        i++;

    }

    fclose(input);
    fclose(outputMoorer);
    return 0;

}

 ( reverberatingDelay.c  )  ~\Desktop\p3 cmpilat\functii udor\reverberatingDelay.c
```

Parametrii pe care îi are funcția sunt botezați exact ca în formula, cu aditia parametrului delay_ms, care reprezinta delay-ul în milisecunde, și adresa unui bufferObject, pe care îl va folosi funcția să țină minte eșantioanele întârziate.

Funcția calculeaza cate eșantioane sunt necesare pentru intarzierea in ms dorită, actualizează aceasta valoare în bufferObject. Este apoi inițializat s1, care va fi folosit în formulele ce urmează; motivul pentru care folosim WORD16(0.999) la inițializare în loc de 1, este ca WORD16(1) este -32768, așa ca 0.999 e cea mai rezonabilă aproximare pe care o avem.

După inițializarea lui s1, puteți observa ca de fiecare data actualizam valoarea indicelui din buffer cu % delay samples, ca sa nu ajungă indiceBuffer foarte mare, căci daca-l lasam sa creasca la infinit (să spunem ca dădeam la intrare un semnal mai lung de 32768, ceea ce nu e foarte mult, indiceBuffer crește peste short), aveam probleme.

Apoi, destul de evident, luăm din coada elementul cu prioritate, botezat “popat”, adaugă ,cu metoda append, în coada, elementul x * wet *s1 + popat * g , pentru ca acesta este semnalul care intra în celulă de intarziere, si la final, funcția întoarce x * dry + popat, semnalul care rezulta la nodul de ieșire al structurii.

La final, în main, după ce terminăm de citit eșantioanele semnalului de intrare, hranim funcția cu 0-uri cat lungimea bufferului de intarziere, ca sa golim bufferul de eșantioanele întârziate. Mai jos am verificat dacă lungimile sunt corecte după convolutie:

```
In [85]: len(tehno)
Out[85]: 705600

In [86]: len(semnalControl)
Out[86]: 709570

In [87]: len(a)
Out[87]: 709570

In [88]: 44.1 * 90
Out[88]: 3969.0
```

În figura de mai sus, “tehno” este semnalul inițial (am uitat sa schimbăm numele), de o lungime de 705600 eșantioane, a este semnalul scos de SC140, semnalDeControl este semnalul generat în python, de lungime 709570 iar 44.1*90 este lungimea de intarziere în eșantioane. Am mai adaugat un eșantion de 0 ca sa putem asculta in python. În final, obținem o suprapunere perfectă a semnalului de control generat în Python și a celui ieșit din SC140:

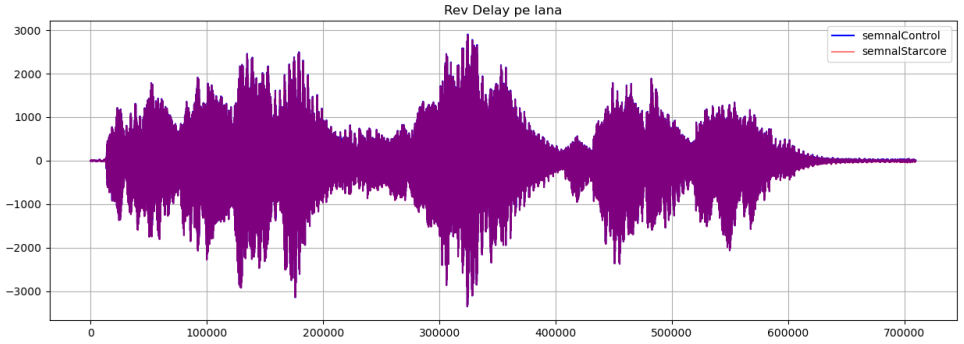


Fig. 7 Semnal de ieșire suprapus cu simularea Python

Și ca sa va faceți o idee despre eroare, am atașat, de asemenea, o poza cu zoom in:

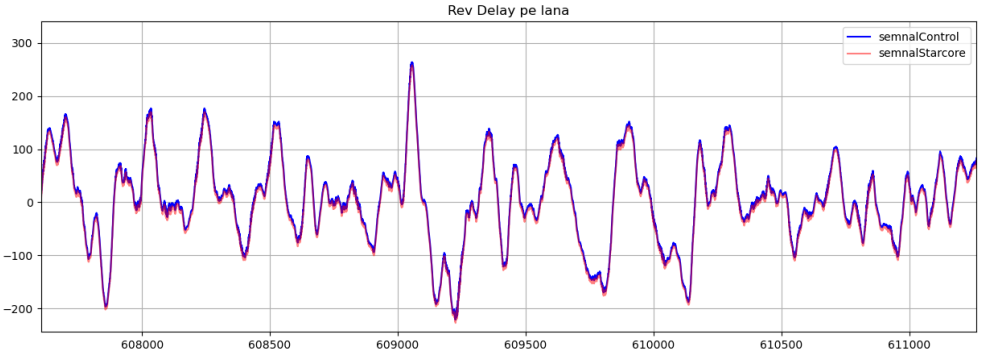


Fig. 8 Fig.5 Suprapunere zoomed in

O observație importantă, este că, datorită faptului că am declarat global bufferObject-ul, C-ul a inițializat totul cu 0, deci este complet legal sa facem operațiile pe care le facem chiar dacă nu am inițializat pe nicăieri nimic.

2.2 Filtrul Trece-Tot

Filtrul Trece-Tot este una din componentele de bază ale filtrelor mai complexe de reverberație.

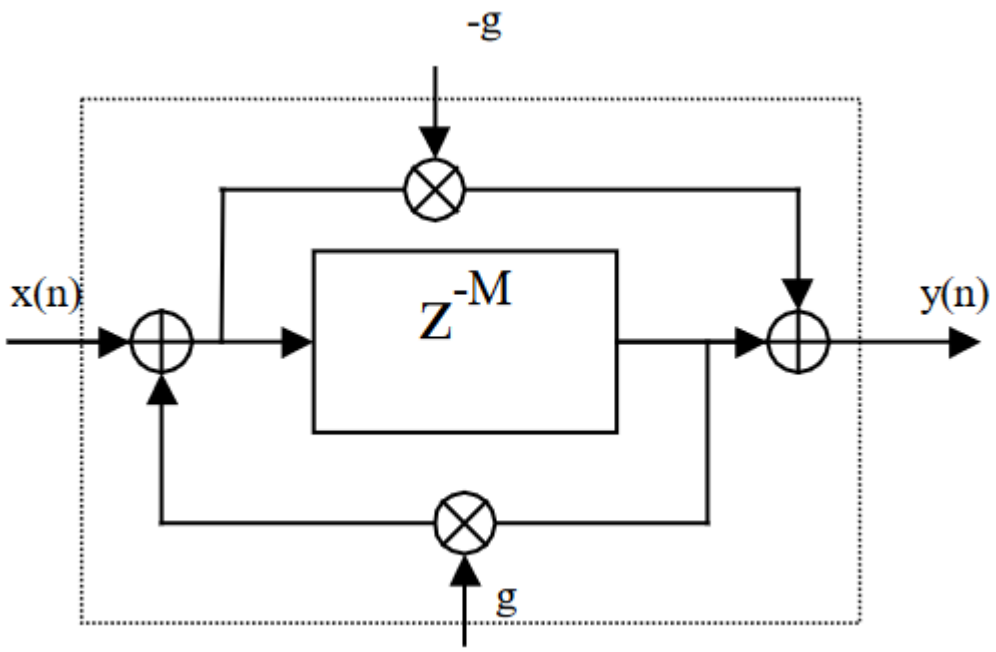


Fig.9 Schema bloc filtru trece-tot

Implementare:

```
short AllpassFilter(Word16 x, Word16 g, Word16 delay_ms, bufferObject *buffer){
    buffer->delaySamples = 44.1 * delay_ms;
    buffer->indiceBuffer %= buffer->delaySamples;
    short popat = dequeue(buffer);
    append(add(x, mult(popat,g)), buffer);
    return add(mult(add(x, mult(popat, g)), -g) , popat);
}
```

Funcția filtrului trece-tot primește ca parametrii un sample al semnalului, coeficientul multiplicativ g, valoarea întârzierii în secunde și o structură de tip buffer. Implementarea schemei bloc se face astfel; se seteaza parametrii structurii buffer în funcție de ce parametrii primește funcția, apoi, similar cu reverberatingDelay, doar implementăm formula funcției de transfer, pe care am determinat-o ca fiind y[n] = (x[n] + buff[n-M] * g)*(-g) + buf[n-M]

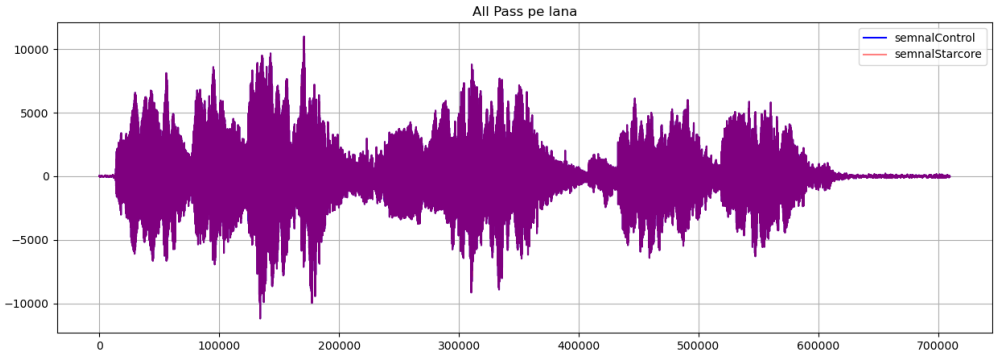


Fig.10 Semnal de ieșire suprapus cu simularea Python

Mai sus, puteți observa din nou, o suprapunere perfectă între semnalul de test și cel din sc140.

2.3 Schroeder

Filtrul schroder construiește pe cele doua filtre de mai devreme, cu excepția ca de data asta nu mai hranim cu zero la intrare după ce se termina semnalul, pentru ca nu ne-am prins cum sa facem asta corect. Inițial am zis ca “nu are cum sa fie greu”, însă a fost prima data cand ne-am dat seama ca trebuie sa fim extraordinar de atenți la scalarea semnalului ca sa fie toata lumea în regula. Eu și colegul meu avem tendința să abuzăm de tipuri de date mari, amândoi fiind obișnuiți cu Matlab si Python care sunt limbaje interpretate. De asemenea, când am implementat filtrul Schroeder, nu înțelesesem exact cum funcționează scalarea L1 și tot tacâmul, așa ca am dezbătut până am hotarat sa o luăm cât putem noi de logic. A rezultat funcția următoare:

```
58
59 short schroeder(short x, Word16 dry){
60     //injumatatire sa nu depasim
61     x = shr(x, 1);
62     short copieX = shr(x, 1); // pentru cele 4 linii
63     short temp = 0;
64     temp = add(temp, reverberatingDelay(copieX, WORD16(0), WORD16(0.999), WORD16(0.5), 35, &buffer1));
65     temp = add(temp, reverberatingDelay(copieX, WORD16(0), WORD16(0.999), WORD16(0.5), 40, &buffer2));
66     temp = add(temp, reverberatingDelay(copieX, WORD16(0), WORD16(0.999), WORD16(0.5), 45, &buffer3));
67     temp = add(temp, reverberatingDelay(copieX, WORD16(0), WORD16(0.999), WORD16(0.5), 50, &buffer4));
68     temp = AllpassFilter(temp, WORD16(0.7), 5, &buffer5);
69     // fa functie de resetat bufere pentru reutilizare si eficientizare memorie
70
71     temp = AllpassFilter(temp, WORD16(0.7), 2, &buffer6);
72     temp = add(temp, mult(dry, x));
73     return temp;
74 }
75
```

Pentru început, se observa ca împărțim x-ul la 2, apoi hranim funcțiile de reverberatingDelay cu o copie a lui x, inca odata impartita la 2. De ce? Ne vom lămuri repede aruncând un ochi la schema filtrului Schroeder:

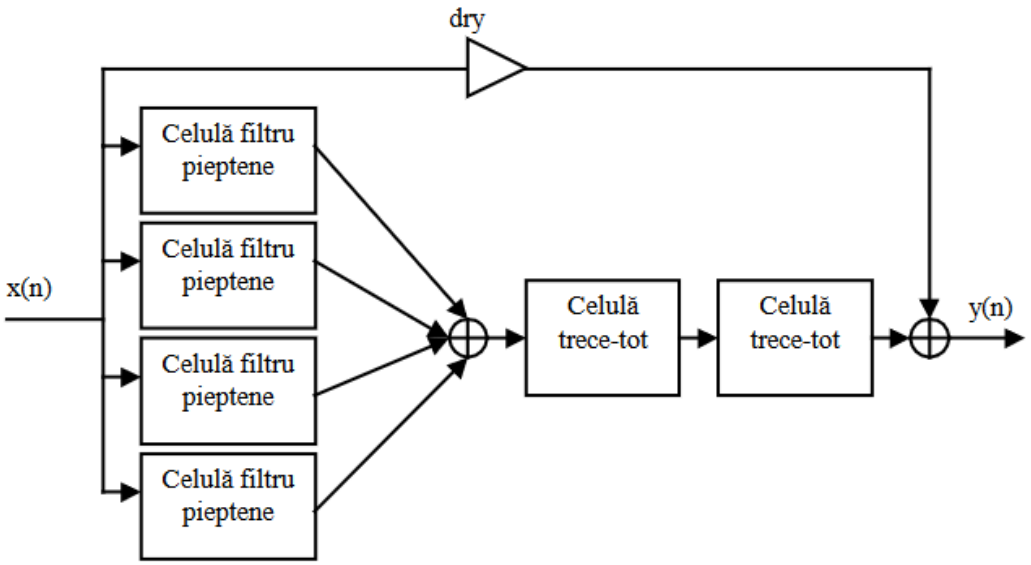


Figura 3.30 Modelul de reverberaţie a lui Schroeder

Putem spune ca semnalul inițial se împarte întâi din doua bucati: prima va trece prin scalarul “dry” iar a doua prin cele 4 filtre comb. De aici prima injumatatire.

Cat despre filtrele comb, sunt 4 filtre, iar în ele intra jumătate din X, în acest moment. Mai este nevoie deci de o injumatatire a lui X, ca să între în fiecare filtru câte un sfert, ca sa nu avem overflow. Așa deci am ajuns sa facem cele doua shiftari la dreapta și să evităm overflow.

Restul funcției doar implementează schema eșantion cu eșantion. Variabila temp e folosită pentru a tine tot ce trebuie la ieșiri. Variabila copieX e folosită de asemenea și ca sa avem o valoarea inițială a lui x pe care sa o folosim la sumatorul final.

O ultima observatie: se vede ca am apelat funcțiile reverberating delay cu timp in milisecunde în loc de eșantioane; am luat valorile în eșantioane din document și le-am împărțit la 44.1 (Fs) si am obtinut acei timpi. Amandoi credem ca nu e un efect așa spectaculos, dar știm ca acei timpi pot fi ușor ajustati.

În final, după cum v-am obișnuit, o suprapunere perfectă între semnalul de control și cel din sc140:

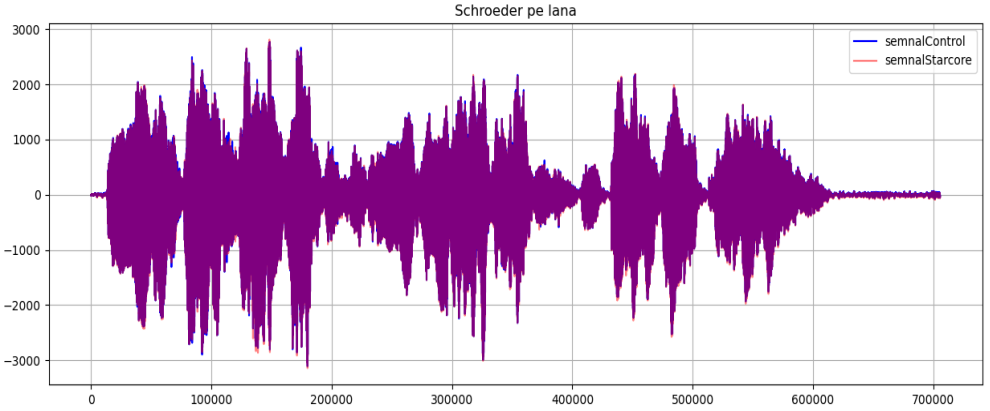


Fig.12 Semnal de ieșire suprapus cu simularea Python

2.4 Moorer

Reflexii inițiale

Dificil la aceasta parte a fost sa gandim un algoritm cat de cat eficient ca timp și memorie, pentru ca nu voiam sa facem un buffer pentru fiecare dintre cele 19 tap-uri ale filtrului, apoi sa le configuram manual. De menționat ca implementarea de mai jos e posibila doar datorită faptului ca filtrele sunt în serie, spre deosebire de comb-uri unde chiar trebuie sa alocam buffers individuale, ca sunt filtre în paralel.

```
77
78 short ri(short x, bufferObject *buffer){
79     buffer->delaySamples = intarzieri[17]; //ar trebui executat o singura data, dar nush unde altundeva sa l pun
80
81
82     buffer->indiceBuffer %= intarzieri[17]; //resetam indicele si ca sa nu depaseasca si ca sa lista circulara
83     append(x, buffer);    //DEBUG, pe aici pe undeva se ascunde un bug de un esantion
84
85
86     int i, suma = 0;
87     for (i = 0; i < 18; i++){
88         if (buffer->indiceBuffer - intarzieri[i] < 0 ){ //trebe mai mic sau egal ca defapt 190 e 189
89             suma = add(suma, mult(buffer->BUFFER[buffer->indiceBuffer  + intarzieri[17] - intarzieri[i]], ponderi[i]));
90         } //de asemenea trebe compensat cu + 1 din acelasi motiv
91         else { //pe aici pe undeva se pierde un esantion, cred ca trebe compensat cu -1 aici sau adaugat in python
92             suma = add(suma, mult(buffer->BUFFER[buffer->indiceBuffer - intarzieri[i]], ponderi[i])); //m am gandit foar
93         }
94
95     }
96
97
98     return suma;
99
100
```

Ideea e sa alocam un buffer doar, de lungime maximă și să-l folosim pentru toate filtrele; algoritmul suna așa: “ pentru primul tap, se folosește buffer-ul ca să ne acumulam toate eșantioanele necesare pentru întârziere. Apoi, deoarece indicele este resetat doar când ajungem la lungimea maximă a bufferului, o sa ne mutam totul la dreapta cu 1, până ajungem la lungimea maximă a bufferului. Abia cand ajungem la lungimea maximă a bufferului (cea de 4000), se reseteaza indicele și începem să suprascriem valorile de la început, deci ca să accesăm buffer[n-M], trebuie sa adaugam un offset egal cu lungimea bufferului ”. Este tot o lista circulara, la finalul zilei, implementată puțin mai complex, dar super eficientă din punct de vedere al utilizării memoriei.

Am adaugat o poză, cu un desen despre cum merge bufferul la reflexiile inițiale. Mai bine de atat nu putem explica. Mai jos, este o poza cu filtrul Moorer întreg:

```
106 short milisecundeComburi[] = {40, 44, 48, 52, 56, 60};
107
108 short moorer(short x, short reglSemnal, short reglReflexii, short reglReverb){
109     short reflexiiInit, copieReflexiiInit, iesireComburi = WORD16(0), reverbFinal, y;
110     float gainComburi = 0.8;
111     reflexiiInit = ri(x, &bufferRI);
112     int i;
113     copieReflexiiInit = shr(reflexiiInit, 3); //scalare for good measure
114     for (i = 0; i < 6; i++) {
115         iesireComburi = add(iesireComburi, reverberatingDelay(copieReflexiiInit, WORD16(0), WORD16(0.999), WORD16(gainComburi), milisecundeComburi[i], &buffereComburi[i]));
116     }
117     reverbFinal = AllpassFilter(iesireComburi, WORD16(0.5), 7,&buffer7);
118     y = add( add( mult(x, reglSemnal), mult(reglReflexii, reflexiiInit)), mult(reglReverb, reverbFinal));
119     return y;
120 }
121
```

Și, după cum v-am obișnuit, o suprapunere aproape perfectă între semnalul de test și cel din starcore:

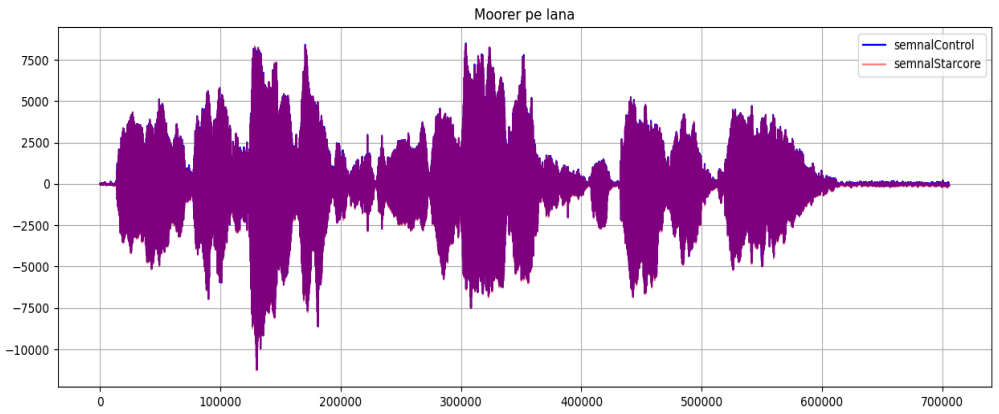


Fig.13 Semnal de ieșire suprapus cu simularea Python

Singura chestie de menționat în plus e ca nu mai știm de ce am ales sa scalam semnalele cu 1 / 2^3, este într-adevăr o valoare aleasă empiric.

Explicație Buffer

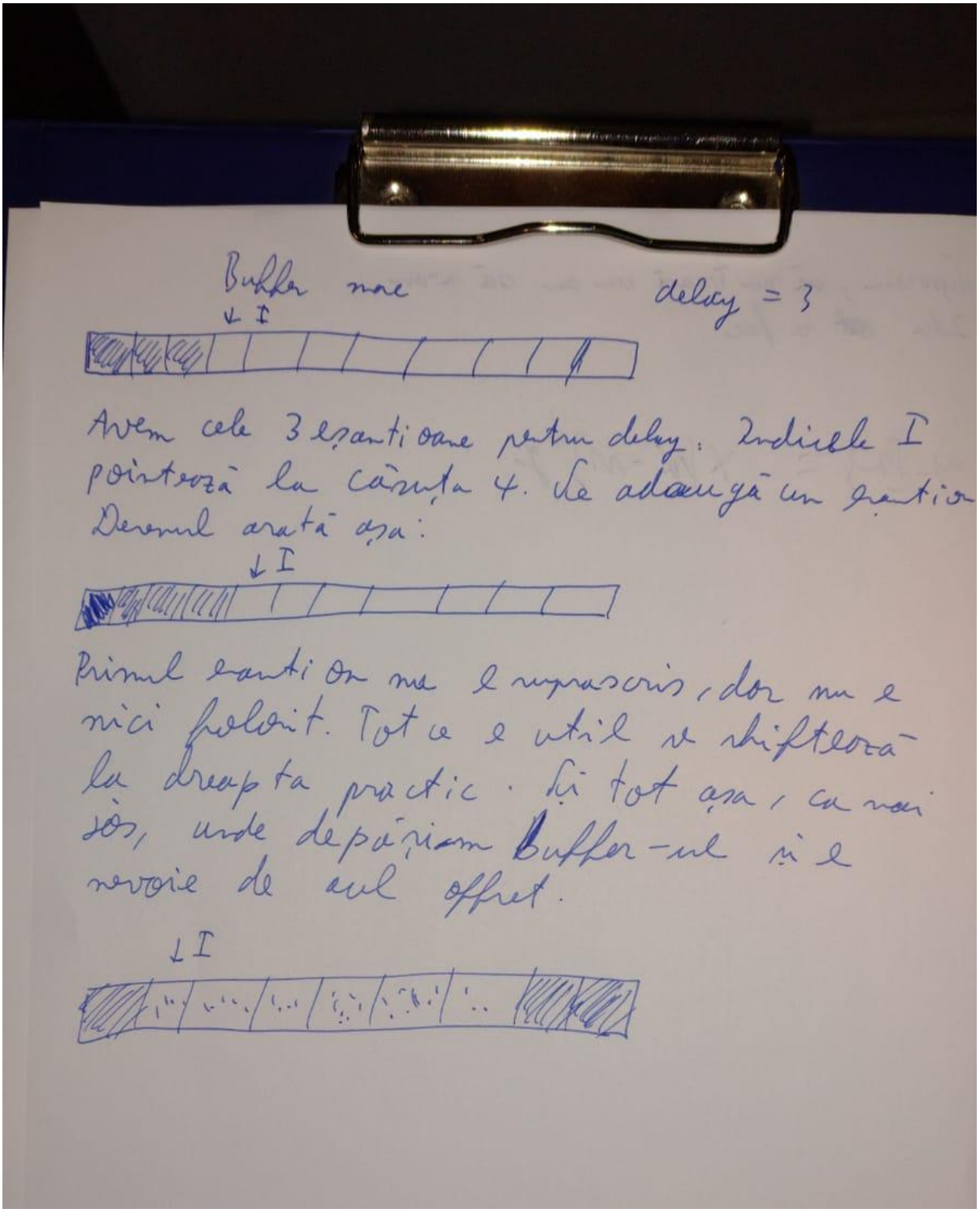
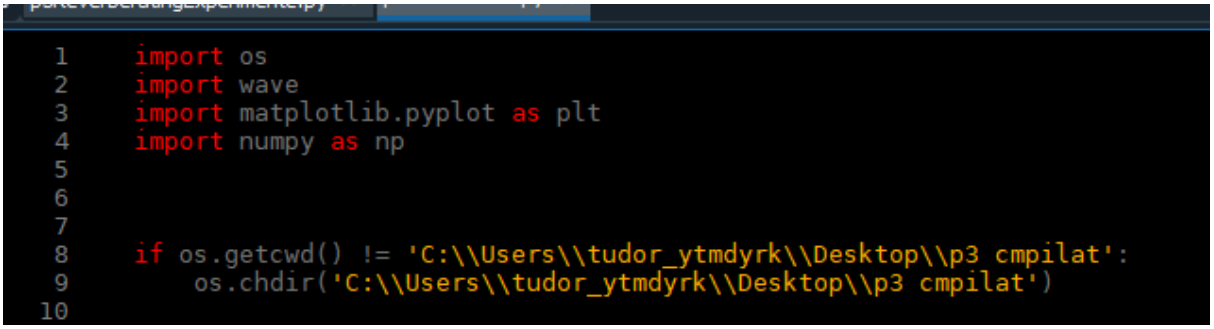


Fig.14 Detalierea utilizării buffer-ului din RI

Anexa: Testarea în Python și MATLAB

Daca vreti sa rulati testele pe care le-am rulat în Python și sa obtineti aceleași grafice că noi, va trebui sa va instalati distributia de anaconda https://www.anaconda.com/products/individual. lansați “Spyder” din search bar-ul de la windows, deschideți fișierul nostru grafice.py și schimbati Path-ul catre pathul absolut al dumneavoastra:



```
1   import os
2   import wave
3   import matplotlib.pyplot as plt
4   import numpy as np
5
6
7
8   if os.getcwd() != 'C:\\Users\\tudor_ytmdyrk\\Desktop\\p3 cmpilat':
9       os.chdir('C:\\Users\\tudor_ytmdyrk\\Desktop\\p3 cmpilat')
10
```

Apoi, cu butonul “play” de sus, rulați fiecare celulă pe rand și așteptați.

Python-ul a fost folosit pentru testarea tuturor funcțiilor recursive.

Pentru puținele teste rulate în MATLAB, am încărcat scripturile pe GitHub, de unde pot fi rulate direct, cât timp se află în același folder cu fișierele de intrare și ieșire necesare.

Simulările au fost realizate în limbajul de programare în care implementarea funcțiilor s-a dovedit a fi mai convenabilă.

Întregul proiect poate fi accesat aici:
https://github.com/tdr999/reverbs/