Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Mesterséges Intelligencia és Rendszertervezés Tanszék

Intelligens beágyazott rendszerek laboratórium
(BMEVIMIMA21)

2. mérés

Laboratóriumi jegyzőkönyv

Készítette:
Gazdag László (HL1YQ4)
Jedla Martin (DEC4F6)

2025. november 10.

Tartalomjegyzék

1. Bevezetés **2**

2. Elméleti háttér **2**
2.1. A hangmagasság-eltolás problémája 2
2.2. Az alkalmazott kétlépcsős módszer 2
2.3. A SOLA (Synchronous Overlap-Add) algoritmus részletei[1] 3
2.4. Az újramintavételezés (resampling) elve 5

3. Megvalósítás **5**
3.1. Hardveres környezet és valós idejű adatfolyam 5
3.2. A SOLA algoritmus C implementációja (Fixpontos) 6
3.3. Az Újramintavételezés (Resampling) implementációja 7

4. Mérési eredmények **7**
4.1. Hangmagasság csökkentése (Down-pitch) 8
4.2. Hangmagasság növelése (Up-pitch) 8

5. Összefoglalás **9**

1. Bevezetés

A laboratóriumi gyakorlat célja egy valós idejű hangmagasság-eltoló (pitch shifter) algoritmus tervezése és megvalósítása C nyelven. A célplatform az Analog Devices *ADSP-BF537 (Blackfin)* processzora, egy 16-bites, fixpontos DSP. A feladat a bemeneti audiojel hangmagasságának tetszőleges, p faktorral történő skálázása, anélkül, hogy a jel időbeli hossza megváltozna.

A választott módszer egy kétlépcsős, időtartománybeli eljárás, amely a *SOLA (Synchronous Overlap-Add)* algoritmuson alapul[1].

1. Első lépésben a SOLA algoritmus segítségével elvégeztük a jel időbeli nyújtását (time stretching) egy α faktorral, megváltoztatva ezzel a jel hosszát, de megőrizve annak eredeti hangmagasságát.

2. Második lépésben a megnyújtott jelet újramintavételeztük (resampling). Ez a lépés skálázta a hangmagasságot, és egyúttal (megfelelő választott faktorok esetén) visszaállította a jel eredeti időtartamát.

A rendszer az ADSP-BF537 DSP panel audio CODEC-ét használta a jel be- és kimenetére, a feldolgozás pedig valós időben, a SPORT porton keresztül, DMA-vezérelt megszakítási rutinokban történt.

2. Elméleti háttér

2.1. A hangmagasság-eltolás problémája

A hangmagasság-eltolás (pitch shifting) alapvető célja, hogy egy $x(n)$ bemeneti jel f_0 alaphangját (pitch) egy p faktorral $f_{new} = p \cdot f_0$ értékre módosítsuk, miközben a jel teljes T időtartama változatlan marad ($T_{new} = T_{old}$).

Egy egyszerű újramintavételezés (vagyis a jel "lejátszási sebességének" megváltoztatása) önmagában nem megoldás. Ha egy jelet p faktorral "felgyorsítunk", a hangmagassága valóban p-szeresére nő, de az időtartama $1/p$-szeresére csökken.

2.2. Az alkalmazott kétlépcsős módszer

Az alkalmazott algoritmus a következő két lépésben valósítja meg a pitch shift-et:

1. lépés - Időbeli nyújtás (Time Stretching) SOLA-val: Az első lépés célja a jel időtartamának megváltoztatása a hangmagasság megőrzése mellett. A SOLA algoritmust egy α nyújtási faktorral alkalmazzuk.

- Bemenet: $x(n)$ (hossz: T_{old}, hangmagasság: f_0)
- Művelet: SOLA(α)
- Kimenet: $y(m)$ (hossz: $T_{mid} = \alpha \cdot T_{old}$, hangmagasság: f_0)

A SOLA algoritmus ezt úgy éri el, hogy a bemeneti jelből vett blokkokat korreláció alapján illeszt össze és adja átlapolva a kimenetre, de a blokkokat α-val skálázott időközönként veszi. Mivel az egyes blokkok belső szerkezete (periódusideje) nem változik, a hangmagasság megmarad.

2. lépés - Újramintavételezés (Resampling): A második lépés célja a $y(m)$ köztes jel újramintavételezése egy p_{resamp} faktorral. Ez egy "lejátszási sebesség" változtatás, ami egyszerre módosítja a hangmagasságot és a hosszt.

- Bemenet: $y(m)$ (hossz: T_{mid}, hangmagasság: f_0)
- Művelet: Resample(p_{resamp})
- Kimenet: $z(k)$ (hossz: $T_{new} = T_{mid}/p_{resamp}$, hangmagasság: $f_{new} = f_0 \cdot p_{resamp}$)

A faktorok kapcsolata A cél az, hogy a végső hangmagasság $f_{new} = p \cdot f_0$ legyen, a végső hossz pedig $T_{new} = T_{old}$.

1. A hangmagasságból: $f_{new} = f_0 \cdot p_{resamp}$. Ahhoz, hogy $f_{new} = p \cdot f_0$ legyen, teljesülnie kell:
$$p_{resamp} = p$$
2. Az időtartamból: $T_{new} = T_{mid}/p_{resamp} = (\alpha \cdot T_{old})/p_{resamp}$. Ahhoz, hogy $T_{new} = T_{old}$ legyen, teljesülnie kell:
$$(\alpha \cdot T_{old})/p_{resamp} = T_{old} \implies \alpha/p_{resamp} = 1 \implies \alpha = p_{resamp}$$

Konklúzió: Ahhoz, hogy a teljes rendszer p faktorú hangmagasság-eltolást végezzen, mind a SOLA nyújtási faktorát (α), mind az újramintavételezési faktort (p_{resamp}) a kívánt p értékre kell állítani. Az algoritmust a 1. ábra mutatja.

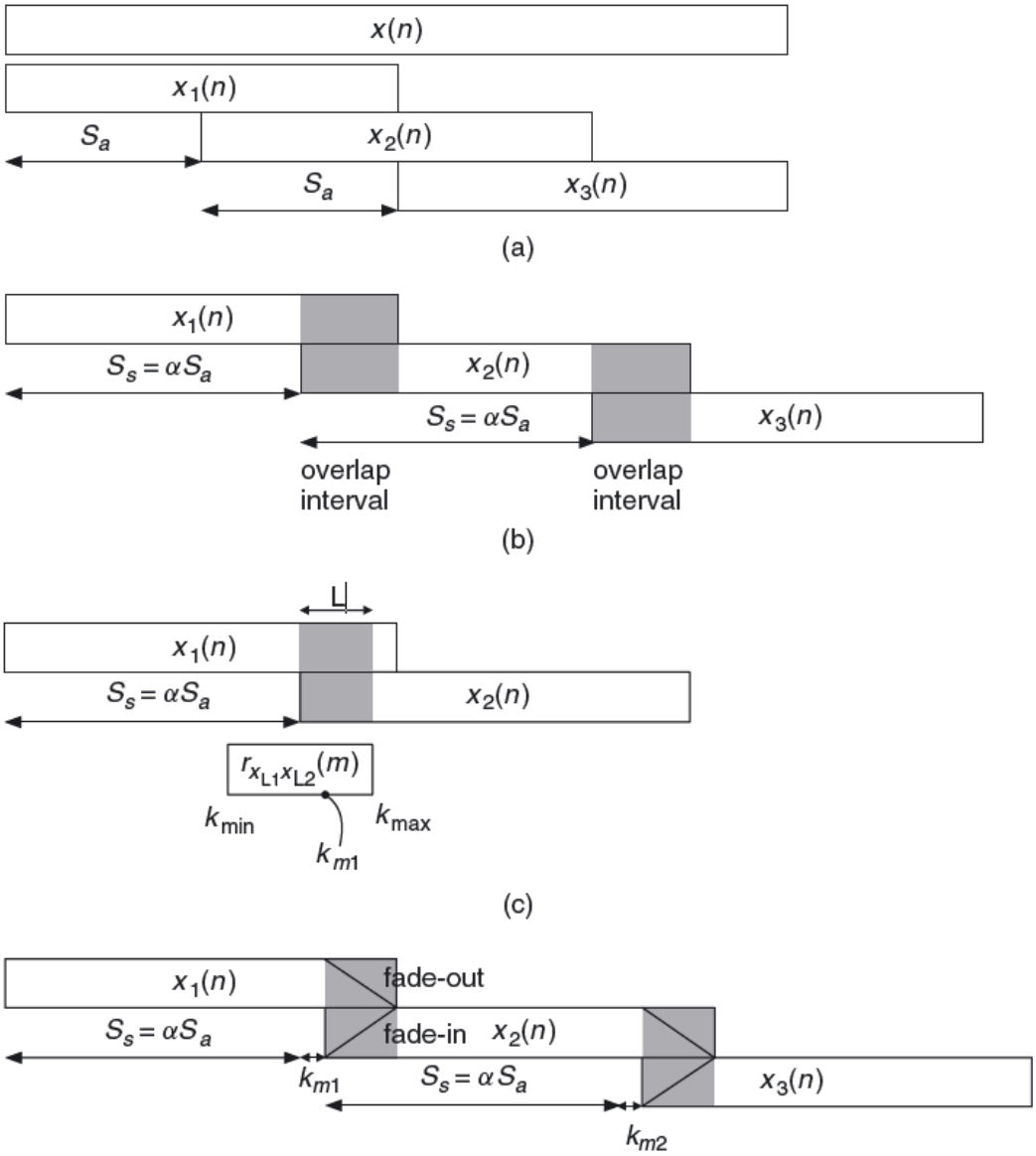
$$\alpha = p \quad \text{és} \quad p_{resamp} = p$$

Például egy 20%-os magasításhoz ($p = 1.2$):

1. A SOLA 1.2-es faktorral 1.2x hosszúságú jelet generál (megtartva a hangmagasságot).
2. Az újramintavételező 1.2-es faktorral "felgyorsítja" ezt a 1.2x hosszabb jelet. A gyorsítás 1.2-szeresére emeli a hangmagasságot (ezt akartuk), és $1/1.2$-szeresére csökkenti a hosszát, ami ($1.2 \cdot 1/1.2 = 1.0$) visszaállítja az eredeti hosszt.

2.3. A SOLA (Synchronous Overlap-Add) algoritmus részletei[1]

1. **Szegmentálás:** A bemeneti $x(n)$ jelet N hosszúságú, átlapolódó blokkokra osztjuk. A blokkok kezdőpontjai közötti távolság S_a.
2. **Áthelyezés:** A kimeneten a blokkokat átlagosan $S_s = \alpha \cdot S_a$ lépésközzel helyeznénk el. Ez biztosítja a α faktorú nyújtást.
3. **Szinkronizálás (keresztkorreláció):** Ha a blokkokat csak mereven S_s távolságra helyeznénk el egymás mellé és adnánk össze (Overlap-Add, OLA), a periodikus jeleknél fázisugrások (kattanások, "glitch"-ek) keletkeznének. A SOLA ezt kerüli el. Amikor egy új blokkot (x_j) kell elhelyezni a kimeneten, azt nem fixen S_s-re helyezi az előzőtől (x_i), hanem megkeresi a legjobb illeszkedési pontot. Vesszük az előző blokk végét (x_{L1}) és az új blokk elejét (x_{L2}) egy L hosszúságú átlapolási



1. Ábra. Az algoritmus lépései ábrázolva[1]

zónában. Kiszámítjuk a két szegmens közötti keresztkorrelációt egy adott k eltolási tartományban:

$$r_{x_{L1}x_{L2}}[k] = \sum_n x_{L1}[n] \cdot x_{L2}[n+k]$$

Megkeressük azt a k_m eltolást (lag), amelyre ez a korreláció maximális. Ez a k_m jelöli azt a pozíciót, ahol a két jel a legjobban "hasonlít" egymásra, azaz ahol a fázisillesztés optimális.

4. **Korrekció és szintézis:** Az új blokkot az optimális k_m eltolással korrigált pozícióba helyezzük. Végül a két, immár szinkronizált blokk átlapolódó részét egy ablakfüggvénnyel (jellemzően lineáris fade-out az előző blokkon és fade-in az újon, ami egy háromszögablaknak felel meg) súlyozva összeadjuk (Overlap-Add). Ez a szinkronizált, finom átmenet biztosítja az artefaktum-mentes kimenetet.

2.4. Az újramintavételezés (resampling) elve

A SOLA által előállított $y(m)$ jel újramintavételezése (a p faktorral) a jel "lejátszási sebességének" megváltoztatását jelenti. Mivel a p faktor (pl. 1.2 vagy 0.8) általában nem egész szám, a kimeneti mintákat a bemeneti minták *közé* eső időpontokban kell kiszámítani.

Az ideális megoldás egy precíz aluláteresztő szűrést végző *polifázisú szűrő* lenne; azonban ennek valós idejű implementálása a laborgyakorlat szűkös időkeretei miatt nem volt lehetséges. A legegyszerűbb lineáris interpoláció (amely csak 2 pontot használ) bár számításilag gyors, de gyenge aliasing-elnyomást biztosít, ami hallható magasfrekvenciás komponenseket okozhat a kimeneten.

A laboratóriumi gyakorlaton ezért egy jobb minőség-költség kompromisszumot adó *harmadfokú (köbös) spline interpolációt* (cubic spline interpolation) valósítottunk meg. Ez a módszer a lineárisnál egy sokkal simább görbét illeszt a mintákra, ami hatékonyabb aluláteresztő szűrésnek felel meg. Míg a lineáris interpoláció két pontot, a köbös spline négy szomszédos mintapontot használ a köztes minta értékének becsléséhez.

A kimeneti buffer k-adik mintájának kiszámításához először meghatározzuk a köztes olvasási pozíciót a bemeneti $y(m)$ bufferben: $pos = k \cdot p$. Legyen $m = \lfloor pos \rfloor$ a pozíció egész része (az index), $f = \text{frac}(pos)$ pedig a törtrésze ($0 \leq f < 1$). A számításhoz szükséges négy szomszédos pont:

- $p_0 = y(m-1)$
- $p_1 = y(m)$
- $p_2 = y(m+1)$
- $p_3 = y(m+2)$

Ezekből a $z(k)$ kimeneti mintát a Catmull-Rom spline (egy elterjedt köbös interpolációs forma) harmadfokú polinomjával számítottuk:

$$\begin{aligned} z(k) = p_1 &+ f \cdot \frac{1}{2}(p_2 - p_0) \\ &+ f^2 \cdot \frac{1}{2}(2p_0 - 5p_1 + 4p_2 - p_3) \\ &+ f^3 \cdot \frac{1}{2}(-p_0 + 3p_1 - 3p_2 + p_3) \end{aligned}$$

Bár ennek a harmadfokú polinomnak a kiértékelése (több szorzás és összeadás) láthatóan számításigényesebb, mint az egyetlen szorzást és összeadást igénylő lineáris módszer, az ADSP-BF537 processzor hardveres MAC (Multiply-Accumulate) egységével még így is hatékonyan, valós időben implementálható volt. Cserébe a kimeneti jel minősége (különösen magasabb frekvenciákon és nagyobb p faktorok esetén) jelentősen jobb lett, és kevesebb, kisebb amplitúdójú belapolódott komponenst tartalmazott.

3. Megvalósítás

3.1. Hardveres környezet és valós idejű adatfolyam

- **SPORT és DMA:** Az audio minták a SPORT0 (Serial Port) interfészen keresztül érkeztek, és DMA (Direct Memory Access) vezérléssel kerültek a memória és a port között mozgatásra. Ez tehermentesítette a CPU-t a mintánkénti adatmozgatás alól.

- **Kettős bufferelés (Ping-Pong Buffering):** A folyamatos, megszakításmentes működés érdekében a `process_data.c` fájlban látható kettős bufferelési sémát alkalmaztuk. Négy fő buffert definiáltunk: `input1`, `input2`, `output1`, `output2`. Mindegyik buffer mérete $N = 1024$ minta (`functions.h`). Ez a nagy méret ($4 \times N$ minta) komoly kihívást jelentett, mivel a bufferek és a program futásidejű verme (stack) ugyanazon a szűkös L1 memórián osztoztak. Különös figyelmet kellett fordítani a stack méretére; a választott $N = 1024$-es méret mellett a rendszer éppen stabilan tudott működni a CrossCore Embedded Studio által alapértelmezetten definiált veremméret mellett. Nagyobb N érték már veremtúlcsordulást (stack overflow) okozott.

- **L1 memória optimalizálás:** A kritikus sebességű, nagy méretű buffereket a `section("L1_data_a")` és `section("L1_data_b")` direktívákkal a processzor leggyorsabb, L1 belső SRAM memóriájába helyeztük (a stack helyett a globális adatmemóriába), elkerülve a lassú külső SDRAM elérést és a verem túlhasználatát.

- **Megszakítás-vezérelt adatkezelés:** A `Process_Data(void)` függvény maga a DMA megszakítási kiszolgáló rutin (ISR). Ennek feladata nem a fő feldolgozás, hanem kizárólag az adatkezelés:

  1. A beérkező mintákat (`iChannel0RightIn`) beírja az éppen inaktív `input_next` bufferbe.
  2. A már feldolgozott `output_current` bufferből kiolvassa a mintákat, és kiküldi azokat a DAC-ra (`iChannel0RightOut`).
  3. Amikor a buffer (1024 minta) megtelik, elvégzi a buffer-cserét (pointer-swap), és beállítja a `process_start = true` globális flag-et.

- **Fő feldolgozó ciklus:** A számításigényes fő algoritmus (a `process(void)` függvény) a főprogram `while(1)` ciklusában fut. Amikor az ISR jelzi a `process_start` flag-en keresztül, hogy egy teljes, 1024-es buffer rendelkezésre áll (`input_current`), a fő ciklus elvégzi rajta a SOLA és resampling műveleteket, majd az eredményt az `output_current` bufferbe írja.

3.2. A SOLA algoritmus C implementációja (Fixpontos)

A `process()` függvény hívta meg a SOLA algoritmus építőelemeit, melyek paramétereit a `functions.h` definiálja. A két legfontosabb művelet a `corr()` és az `apply_fade()` volt.

- **Keresztkorreláció számítása (`corr()`):** A SOLA legszámításigényesebb művelete a `corr()` függvény, amely megkeresi a legjobb illeszkedési pontot (k_m) a régi blokk vége (`arr1`) és az új blokk eleje (`arr2`) között. A `functions.c` fájlban látható implementáció két fontos optimalizálást tartalmaz:

  1. A korrelációt csak egy szűk, $L = 64$ mintás ablakon belül kerestük.
  2. A kód nem egyszerűen a legnagyobb szorzatösszeget kereste. A `functions.c` implementációja normalizálta a korrelációs összeget. Minden k eltoláshoz kiszámította az aktuális átlapolási ablak hosszát (`window_len = L - k`), majd az összeget (`accu`) elosztotta ezzel a hosszal: `accum normalized_accu = accu`

`/ window_len;`. Ez a normalizálás biztosítja, hogy a rövidebb átlapolások (nagyobb k eltolások) ne kerüljenek hátrányba. A rendszer azt a k_m indexet adja vissza, ahol az átlagos hasonlóság a legnagyobb.

- **Átlapolás és ablakozás (`apply_fade()`)** Az optimális k_m eltolás megtalálása után a blokkok szinkronizált összeadását az `apply_fade()` függvény készítette elő. Ez a funkció valósította meg a "crossfade"-et, vagyis az ablakozást. A `functions.c` kódja alapján ez egy fixpontos lineáris rámpát generál. Kiszámítja a lépésközt (`dx = FRACT_MAX / fade_length`), majd egy ciklusban a régi blokk végét (`arr1`) 0-ra fade-eli (fade-out), az új blokk elejét (`arr2`) pedig 1-re fade-eli (fade-in). A tényleges összeadás (Overlap-Add) ezután a fő `process()` függvényben történik, a már előkészített (ablakozott) blokkokkal.

3.3. Az Újramintavételezés (Resampling) implementációja

Az újramintavételezést a `resample_spline()` és az általa hívott `cubic_interp()` függvények valósították meg C nyelven, fixpontos aritmetikával.

- **Harmadfokú Spline Interpoláció (`cubic_interp()`)** A laboron a Catmull-Rom spline egy specifikus, fixpontos megvalósítását használtuk. A függvény a négy szomszédos mintapont ($y0, y1, y2, y3$) és a tört-eltolás (x) alapján számolta ki a harmadfokú polinom együtthatóit ($c0, c1, c2, c3$). A `functions.c` kódja alapján a polinom kiértékelése egy optimalizált, beágyazott formában, az ún. *Horner-módszerrel* történt, ami csökkenti a szükséges szorzások számát: `accum out = ((c3 * x + c2) * x + c1) * x + c0;`. Ez a számítási hatékonyság kulcsfontosságú volt a DSP valós idejű működéséhez.

- **Újramintavételező vezérlő (`resample_spline()`)** Ez a függvény vezérelte az interpolációt. A `functions.c` kódja alapján a működése a következő:

  1. Egy `accum` típusú, 40-bites, fixpontos változót (`source_index`) használt ún. *fázisakkumulátorként*, ami a köztes olvasási pozíciót követte nyomon a bemeneti bufferben.
  2. Minden kimeneti minta generálása előtt az `source_index`-ből kinyerte az egész részt (i) és a törtrészt (x).
  3. Boundary check-et végzett: `if(i < 1) i = 1;` és `if(i > N_in - 3) i = N_in - 3;`. Ez biztosította, hogy a `cubic_interp()` a 4-pontos ablakával ($i-1$-től $i+2$-ig) soha ne olvasson a bemeneti buffer érvényes határain kívülre, elkerülve ezzel a memóriahibákat.
  4. A ciklus végén az akkumulátort növelte a fixpontos `ratio` értékkel.
  5. Külön lekezelte az utolsó mintát (`output_buffer[N_out - 1] = ...`), hogy a buffer végét pontosan másolja és elkerülje az interpolációs hibákat a jel végén.

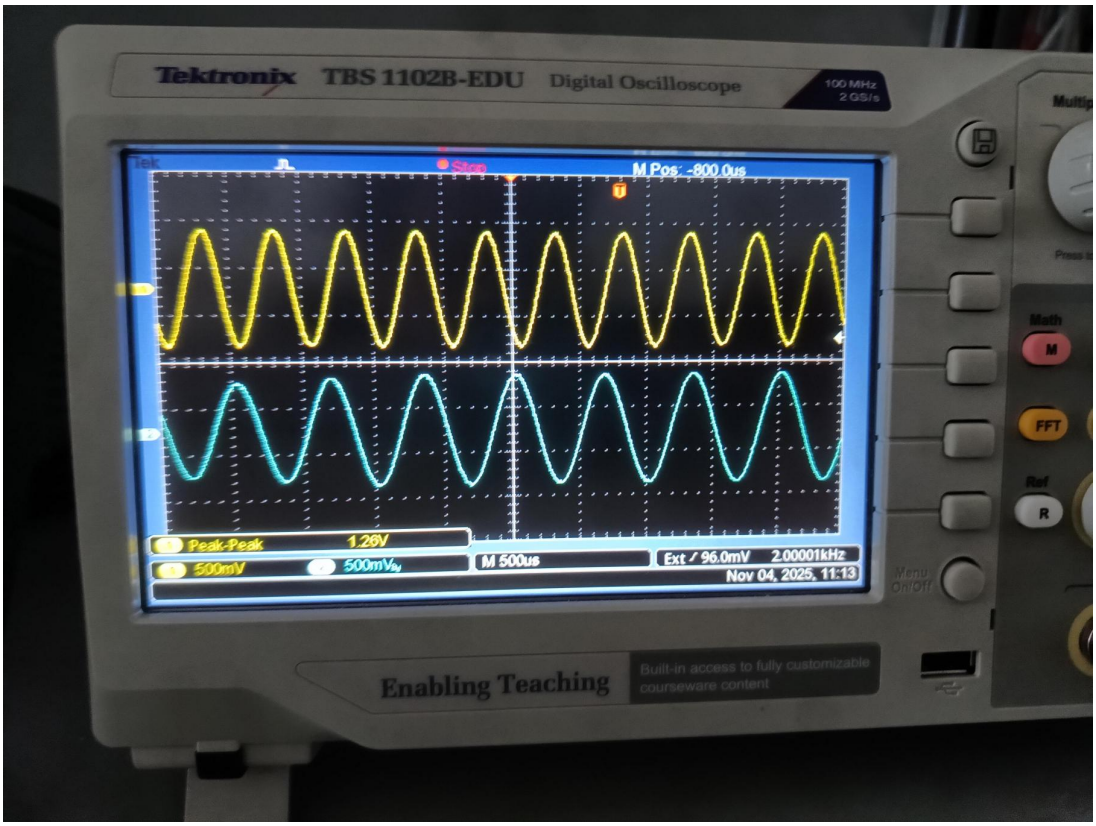
4. Mérési eredmények

A rendszer működését egy jelgenerátorral és oszcilloszkóppal teszteltük. A jelgenerátor szinuszos jelet adott a DSP bemenetére, míg az oszcilloszkóp egyszerre mutatta a bemeneti jelet (CH1, sárga) és a feldolgozott kimeneti jelet (CH2, kék). Az összes mérésnél az

időalap (Timebase) 500 μs/div volt. A bemeneti jel (CH1) frekvenciája konstans 2 kHz volt.

4.1. Hangmagasság csökkentése (Down-pitch)

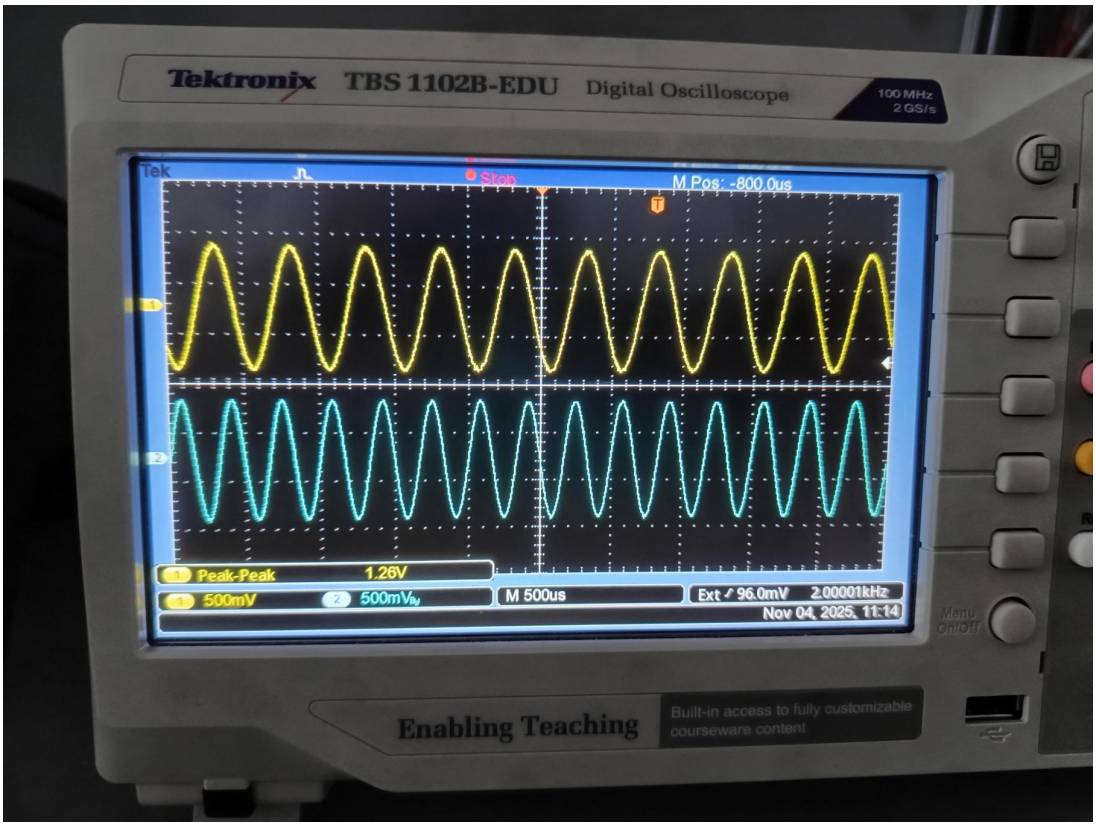
Az első mérés során a pitch shift faktort $p = 0.75$ értékre állítottuk. Ennek megfelelően a SOLA $\alpha = 0.75$ faktorral "rövidítette" a jelet, majd az újramintavételező $p_{resamp} = 0.75$ faktorral "lelassította" azt, visszaállítva az eredeti hosszt. A 2 ábra mutatja az eredményt. Jól látható, hogy a bemeneti 2 kHz-es jel (sárga) periódusidejéhez ($T_{in} = 500\mu$s) képest a kimeneti jel (kék) periódusideje a megnövekedett ($T_{out} = 667\mu$s), ami $f_{out} = 1500$ Hz-es frekvenciát jelent ($f_{out} = f_{in} \cdot p$). Ez pontosan megfelel a hang mélyítésének. A kimeneti jel szinuszos maradt, ami a SOLA algoritmus korrekt fázisillesztését és a spline interpoláció helyes működését igazolja.



2. Ábra. Hangmagasság csökkentése ($p = 0.75$) szinuszos jelen. A kimeneti jel (kék) periódusideje kétszerese a bemeneti jelének (sárga).

4.2. Hangmagasság növelése (Up-pitch)

A második mérés során a shift faktort $p = 1.5$ értékre állítottuk. A SOLA $\alpha = 1.5$ faktorral megnyújtotta a jelet, majd az újramintavételező $p_{resamp} = 1.5$ faktorral "felgyorsította" azt. A 3 ábrán látható, hogy az 2 kHz-es bemeneti jelhez (CH1) képest a kimeneti jel (CH2) periódusideje észrevehetően lecsökkent. Míg a bemenetből 10 periódus látható, a kimenetből 15 periódust számlálhatunk meg ugyanazon időtartam (5 ms) alatt. Ez $f_{out} = 3$ kHz-es kimeneti frekvenciát jelent, ami pontosan megfelel a $p = 1.5$ faktornak. A kimenet itt is stabil és tiszta szinuszos, ami igazolja az algoritmus helyes működését.



3. Ábra. Hangmagasság növelése ($p = 1.5$) szinuszos jelen. A kimeneti jel (kék) frekvenciája 1.5-szerese a bemeneti jelének (sárga).

5. Összefoglalás

A laboratóriumi gyakorlat során sikeresen implementáltunk egy valós idejű, SOLA algoritmuson és újramintavételezésen alapuló pitch shifter-t. A megvalósítás C nyelven, ADSP-BF537 fixpontos DSP fejlesztőkártyán történt. A projekt rávilágított a valós idejű, beágyazott jelfeldolgozás alapvető kihívásaira:

- A számításigényes algoritmusok (normalizált keresztkorreláció) optimalizálása hardverspecifikus utasítások (MAC) használatával.
- A korlátozott pontosságú fixpontos aritmetika kezelése, különös tekintettel a túlcsordulás elkerülésére (pl. 40-bites akkumulátorok) és a pontosságvesztés minimalizálására.
- A folyamatos adatfeldolgozás biztosítása DMA-vezérelt, megszakításos rendszerben, kettős bufferelléssel (ping-pong buffering).

Az oszcilloszkópos mérések igazolták, hogy a megvalósított kétlépcsős (SOLA $\rightarrow$ Resample) algoritmus szinuszos jeleken a vártnak megfelelően működik, elvégezve a hangmagasságeltolást a jel időtartamának megváltoztatása nélkül.

A laboratóriumi gyakorlat szűkös időkeretei miatt azonban az algoritmus kvalitatív tesztelésére összetettebb, tranzienseket is tartalmazó audiojeleken (pl. beszéd, zene) már nem maradt idő. A projekt során elkészült teljes C nyelvű forráskód a projekthez tartozó GitHub repository-ban nyilvánosan elérhető.

Hivatkozások

[1] P. Dutilleux, G. De Poli, A. von dem Knesebeck, and Udo Zölzer. Time-segment processing. In Udo Zölzer, editor, *DAFX: Digital Audio Effects*, chapter 6, pages 191–193. John Wiley & Sons, 2nd edition, 2011. Section 6.3.2.