Aktiv Støj Udligning i Headphone Applikation

Indlejret Signal Behandling - Projektoplæg

Gruppe 1

Navn	Studienummer	Retning
Mathias Ørnstrup Hvidberg	201905706	E
Niels Højrup Pedersen	201604812	E
Jakob Saugbjerg Lange	201907544	E

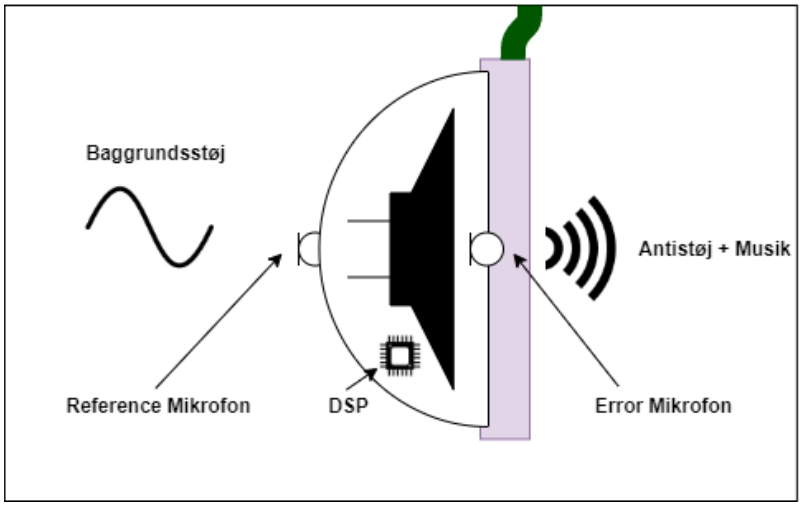
Indhold

1 Introduktion og Problembeskrivelse ... 3
2 Teori ... 4
2.1 LMS ... 4
2.2 Fx-LMS ... 4
2.3 Hastighed ... 6
2.3.1 Hastighedsberegninger ... 6
3 Kravspecifikation ... 8
3.1 Funktionelle krav ... 8
3.2 Ikke-funktionelle krav ... 8
3.3 Afledte krav ... 8
4 MATLAB Løsning ... 9
4.1 Path-estimation ... 9
4.2 ANC – Primary path estimation ... 12
4.2.1 Et mindre forudsigeligt signal ... 14
5 Design og Implementering ... 17
5.1 Systemdesign ... 17
5.2 Softwaredesign ... 17
5.3 Implementering ... 19
5.3.1 Datatyper ... 20
5.4 Cykleforbrug og Optimering ... 20
5.4.1 System Load ... 23
5.5 Memoryforbrug ... 23
5.5.1 Datamemory ... 24
5.5.2 Code memory ... 24
6 Test af implementering ... 25
6.1 MATLAB versus Implementering ... 25
6.1.1 Dæmpning af simple sinus-signal ... 26
6.1.2 Dæmpning af passagerfly-støj ... 27
6.2 Dæmpning af forskellige signaler ... 30
6.3 Test og opbygning af prototype ... 32
6.3.1 Prototype ... 32
6.4 Test af krav ... 35
7 Diskussion ... 36
8 Konklusion ... 37
Referencer ... 38

Appendix 39
Appendix: MATLAB løsning 39
Appendix: Test af implementering 44
Appendix: Test af prototype 49

1 Introduktion og Problembeskrivelse

Dette projekt går ud på at implementere aktiv støjudligning (ANC) til brug i en høretelefonsapplikation. ANC-høretelefoner er efterhånden blevet meget populære på consumer audiomarkedet, og kan ses implementeret i både in-ear og over-ear produkter med lang batterilevetid og imponerende dæmpning. De fleste kommercielle produkter leverer mellem 25-35 dB i det relevante frekvensområde (typisk 20-2000 Hz). De højere frekvenser håndteres med god isolering vha. skum og lignende materialer. Et typisk ANC-system ser ud som illustreret nedenfor på Figur 1.



Figur 1 - Illustration af et ANC-system anvendt i høretelefoner

I et over-ear system fastmonteres der et sæt mikrofoner på selve højtalerkoppen. Den første, benævnt "reference mikrofon" anvendes til at optage baggrundsstøjen. Dette signal føres ind i en DSP, som på baggrund af støjsignalet genererer et "antistøjssignal", der spilles ud af højtaleren. Dette signal er ideelt set bare støjsignalet inverteret, hvilket giver anledning til destruktiv interferens mellem støj og antistøj, heraf navnet "støjudligning". En række yderligere faktorer spiller også ind i genereringen af signalet, hvorfor der også sidder en "error mikrofon", men mere herom senere.

Projektet vil primært fokusere på selve implementeringen af den type filtrer og det software, som skal til for at løse problemet så optimalt som overhovedet muligt, da selve den mekaniske udformning af headsettet samt den dertilhørende akustiske analyse ligger uden for fagets fokusområde. Dog vil der blive lavet et "mock-up" setup til at foretage funktionelle tests, foruden teoretiske verificeringer i MATLAB. Koden implementeres på Analog Devices platformen BF533 EZ-kit. [1]

2 Teori

I dette afsnit gennemgås den teori, der skal anvendes for at kunne implementere et ANC-system.

2.1 LMS

Hele ANC-systemet bygger på LMS-algoritmen [2] (Least-mean-square), som tilpasser filtervægte således at differencen mellem filterets output-signal, $y(n)$, og et ønsket signal, $d(n)$, går mod nul. Denne difference kaldes error-signalet, $e(n)$.

$$e(n) = d(n) - y(n) \tag{1}$$

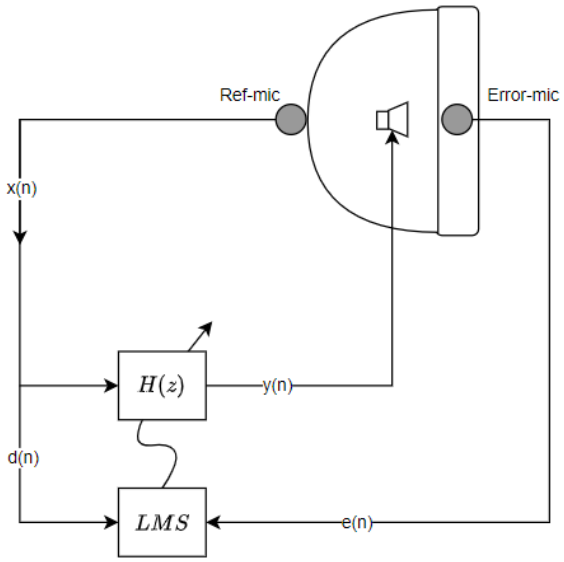
Filteret er et simpelt M-tap FIR-filter, se ligning *(2)*.

$$y(n) = \sum_{m=0}^{M} w_m(n)x(n-m) \tag{2}$$

Selve tilpasningen af vægtene sker ved ligning *(3)*.

$$\boldsymbol{w_m}(n+1) = \boldsymbol{w_m}(n) + 2\mu e(n)\boldsymbol{x}(n) \tag{3}$$

Hvor μ repræsenterer en step-size, dvs. en faktor der afgør hvor hurtigt vægtenes størrelse justerer sig mod deres konvergensniveau. Den kan dog også blive så stor, at fejlen vil begynde at oscillere fra en yderlighed til en anden, så derfor skal den bestemmes nøje. Signalgrafen for et typisk ANC headset system ser ud som set på Figur 2 (Simpel form).



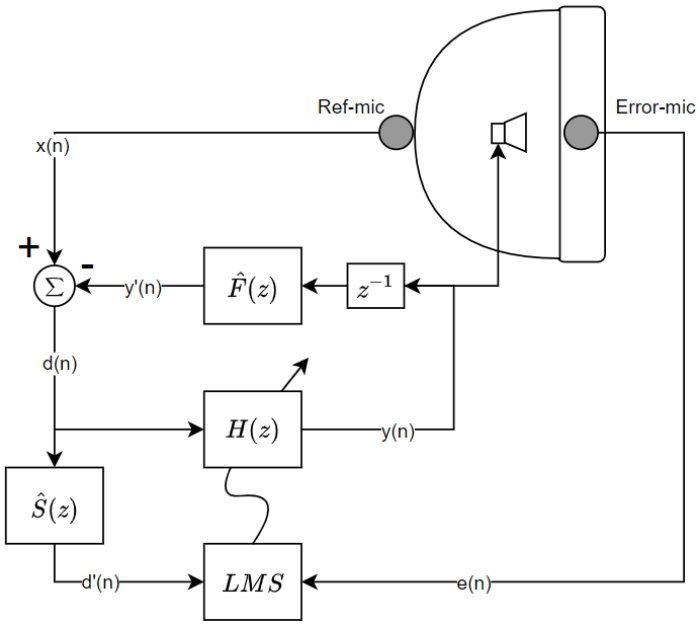
Figur 2 - Signalgraf for simplificeret ANC-system

LMS-algoritmen justerer filtervægte for filteret $H(z)$ for at generere et antistøj-signal ud af højtaleren. Som input tager den lyd fra både reference og error mikrofonen. Lyden fra referencemikrofonen repræsenterer signalet $d(n)$ fra Eq. 1, altså støjsignalet, der skal udlignes med et signal $y(n)$, som outputtets fra filteret. Error-mikrofonen levere signalet $e(n)$, som skal tilnærme sig et minimum. Denne fremstilling repræsenterer dog ikke helt virkeligheden, hvorfor der introduceres Fx-LMS.

2.2 Fx-LMS

Den omkringliggende støjs rejse ind til reference-mikrofonen kaldes primary-path. Målet med ANC-systemet er at danne et antistøjssignal ud fra denne primary-path. Dog skal antistøjs-

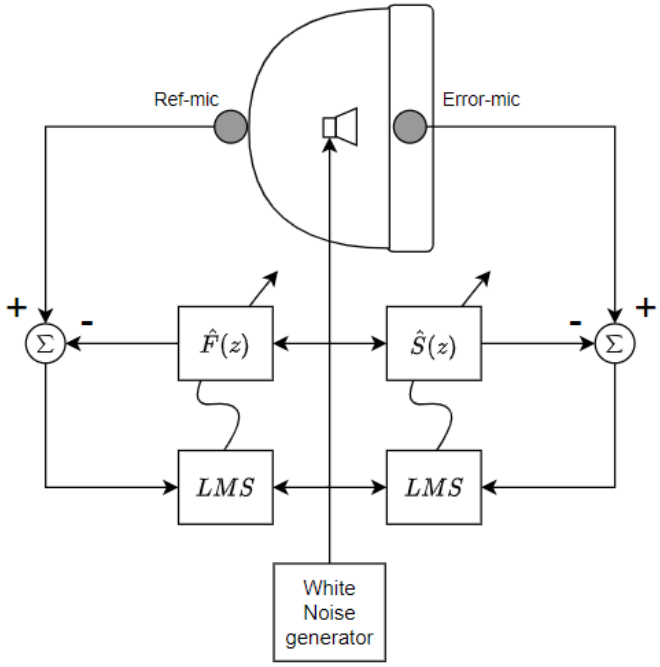
signalet rejse fra højtaleren og ind til error-mikrofronen, hvilket kaldes for secondary-path. Desuden rejser antistøjs-signalet også fra højtaleren og ud i reference-mikrofonen, dette kaldes for feedback-path. Overføringskarakteristikkerne for feedback-path og secondary-path kaldes hhv. $F(s)$ og $S(s)$. Med Fx-LMS [3] tages der højde for disse paths. Dette er nødvendigt for at undgå tilfælde, hvor et feedback forsager ustabile oscilleringer og for at sikre synkroniseringen af antisignalet med støjsignalet i error-mikrofonen. Der skal identificeres nogle estimater for disse paths, $S(s)$ og $F(s)$, som kaldes hhv. $\hat{S}(z)$ og $\hat{F}(z)$, som kan påføres signalerne i systemet og dermed kompenserer for deres karakteristikker. På nedenstående signal-graf (inspireret af [4]) ses det hvordan de estimerede overføringskarakteristikker påføres ANC-systemet, således der kompenseres for disse:



Figur 3 - Signalgraf for ANC-systemet

Overføringsfunktionen $\hat{S}(z)$ er et estimat af overføringsfunktionen $S(s)$, som kan implementeres som et simpelt FIR-filter, der påføres inputsignalet til LMS-algoritmen, hvilket navnet Fx-LMS (Filtered x LMS) kommer fra. Feedback-path estimeringen påføres ANC-systemets filtrerede signal og fratrækkes støjsignalet inden dette føres videre til LMS-algoritmen og filteret $H(z)$. Secondary-path estimeringen påføres støjsignalet fra reference-mikrofonen inden dette føres ind i LMS-filteret. På den måde vil ANC-systemets LMS-algoritme tilpasse vægtene i $H(z)$, med de to paths for øje.

Overføringsfunktionerne $\hat{S}(z)$ og $\hat{F}(z)$ kan estimeres ved brug af en slags kalibrerings-opsætning, hvor LMS-algoritmer tilpasser vægtene i to FIR-filtre, se Figur 4. Der afspilles et hvidstøjssignal ud af højtaleren, som optages med de to mikrofoner i systemet. Hvidstøjssignalet har altså rejst gennem de to forskellige paths og optages, hvorefter LMS-algoritmen tilpasser filter-vægtene i $\hat{S}(z)$ og $\hat{F}(z)$, således at fejlen mellem det filtrerede hvidstøj og den optagne hvidstøj går mod 0. Hermed vil de to FIR-filtre $\hat{S}(z)$ og $\hat{F}(z)$ danne et estimat af de to paths.



Figur 4 - Signalgraf for Path-estimering

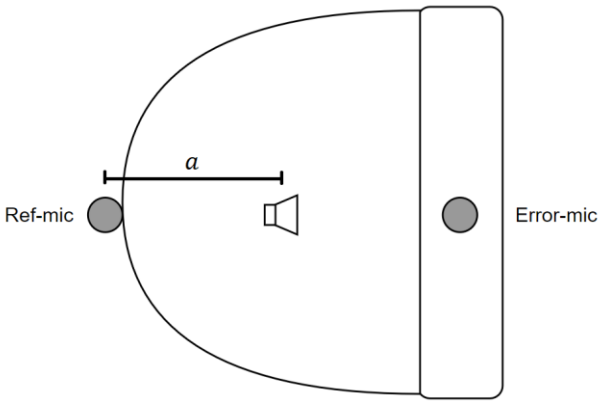
2.3 Hastighed

For at systemet kan fungere optimalt, er der nogle forskellige hastighedskrav der skal overholdes.

2.3.1 Hastighedsberegninger

Afstanden mellem reference-mikrofonen og højtaleren, som vist på Figur 5, kaldes a og er målt til:

$$a = 5 \text{ cm}$$



Figur 5 - Afstanden mellem reference-mikronen og højtaleren

Lydens hastighed regnes som:

$$v_{lyd} = 343 \ \frac{\text{m}}{\text{s}}$$

Dvs. at tiden det tager lyden at rejse fra reference-mikrofonen til højtaleren kan regnes som:

$$t_{lyd} = \frac{a}{v_{lyd}} = 0.146 \text{ ms}$$

DSP'en BF533 har en clock-frekvens på 600MHz. Antaget 90% belastning fås en clock-frekvens på:

$$f_{BF} = 600 \text{ MHz} \cdot 0.9 = 540 \text{ MHz}$$

Det svarer til en periodetid på:

$$T_{BF} = \frac{1}{f_{BF}} = 1.852 \text{ ns}$$

Hvilket er tiden det tager at gennemføre en clock-cycle. Dermed kan processoren nå at udføre:

$$Clock_{lyd-max} = \frac{t_{lyd}}{T_{BF}} = 78717 \text{ cycles}$$

Dvs. dette er det maksimale antal clock-cycles processoren må udføre fra at et sample optages i reference-mikrofonen til at det filtrerede sample outputtes i højtaleren.

Da sample-frekvensen er valgt til:

$$f_s = 48 \text{ kHz}$$

Kan det beregnes hvor mange clock-cycles der maksimalt kan benyttes pr. sample:

$$Clock_{s-max} = \frac{f_{BF}}{f_s} = 11250 \text{ cycles}$$

Dette er tydeligvis lavere end det maksimale antal clock-cycles processoren kan nå, hvormed der godt kan behandles flere samples ad gangen. Det vælges derfor at benytte en input- og output-buffer med størrelsen 4:

$$N = 4$$

Hvorfor der maksimalt må bruges:

$$Clock_{4-max} = Clock_{s-max} \cdot N = 45000 \text{ cycles}$$

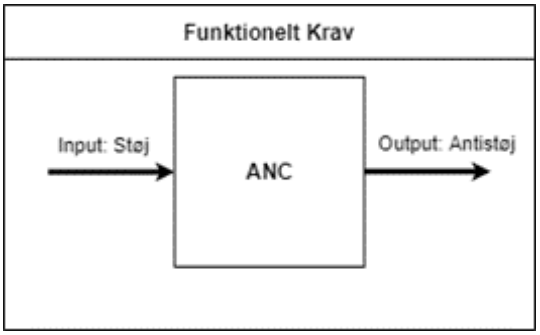
Hvis der benyttes mere end 45000 clock-cycles kan de 4 samples ikke nås at processeres før 4 nye samples er indsamlet, hvorfor dette antal af clock-cycles er den absolutte grænse med denne blocksize på 4.

3 Kravspecifikation

I dette afsnit gennemgås de overordnede krav til systemet, dels funktionelle og ikke-funktionelle.

3.1 Funktionelle krav

Da systemet har minimal brugergrænseflade, er det valgt ikke at specificere use-cases. Funktionaliteten kan illustreres helt enkelt som set nedenfor:



Figur 6 - Funktionaliteten for systemet

På den baggrund kan man opstille følgende funktionelle krav:

- Systemet skal dæmpe baggrundsstøj vha. aktiv støj udligning.
- Systemet skal kunne håndtere forskellige typer støjkilder i frekvensområdet 100-2000 Hz.
- Systemet skal adaptivt regulere sit antistøj-signal, og kunne håndtere ændringer i det akustiske miljø.

3.2 Ikke-funktionelle krav

Nedenfor fremgår systemets ikke-funktionelle krav, hvilket dækker over de objektive kriterier som systemet skal overholde.

Performance

- R1: Systemet skal sikre minimum 20 dB dæmpning af baggrundsstøj målt fra brugerens øre.
- R2: Systemet skal kunne dæmpe baggrundsstøj i frekvensområdet 100 Hz - 2000 Hz.
- R3: Systemet skal have en latenstid fra input til output på maksimalt 0.146 ms pr. sample.
- R4: Systemet skal have et dynamikområde på 96 dB (16-bit opløsning).
- R5: Systemet må maksimalt bruge 90% af BF533's ydeevne

Effektivitet

- R6: Systemet skal kunne adapteres til batteridrevne applikationer og må maksimalt trække 70 mW (20mA, 3.3V) i filtermode.
- R7: Systemets softwareapplikation skal maksimalt fylde 80% instruktionshukommelse.
- R8: Systemets filtrer må maksimalt anvende 80% af ram-hukommelsen.
- R9: Systemets algoritme skal realiseres med fixed point aritmetik.

3.3 Afledte krav

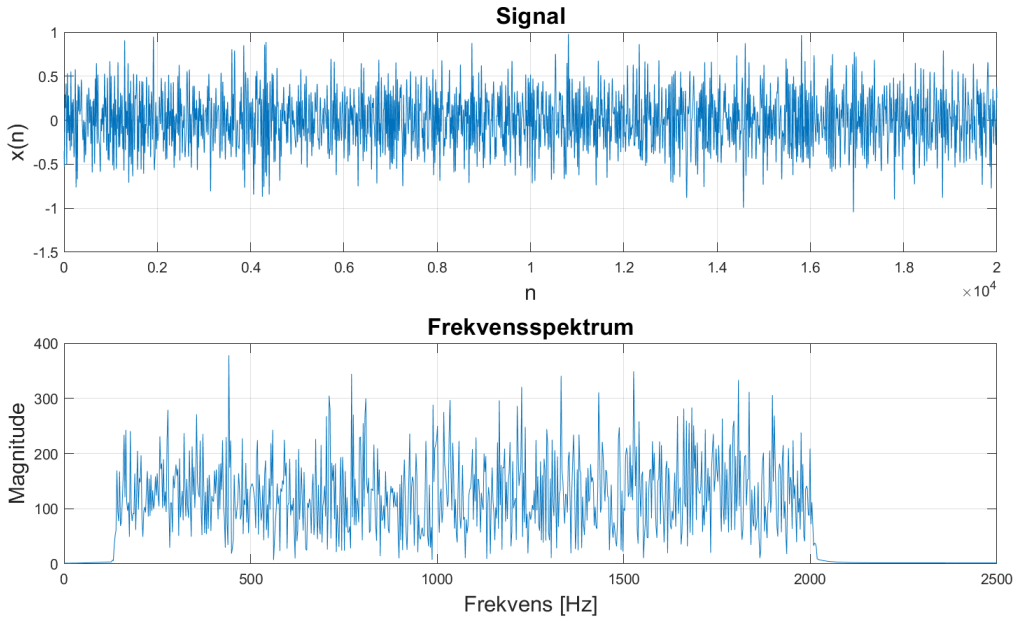
- DR1: Systemet skal sample med 48 kHz.
- DR2: Systemets algoritme må maksimalt bruge 45000 cycles pr. blocksize, hvor en blocksize er 4 samples.

4 MATLAB Løsning

I det følgende gennemgås MATLAB implementeringen inspireret af [5] og [4].

4.1 Path-estimation

For at estimere secondary paths genereres der et hvidstøjssignal i frekvensspektret 100Hz til 2000Hz. I MATLAB gøres dette med en randn-funktion [6], hvorefter signalet båndpasfiltreres, se Figur 7. Signalet simuleres afspillet gennem højtaleren i et headset. Der benyttes en samplings-frekvens på 48kHz og der tages 20k samples.



Figur 7 - Det genererede hvidstøjssignal til estimering af secondary paths

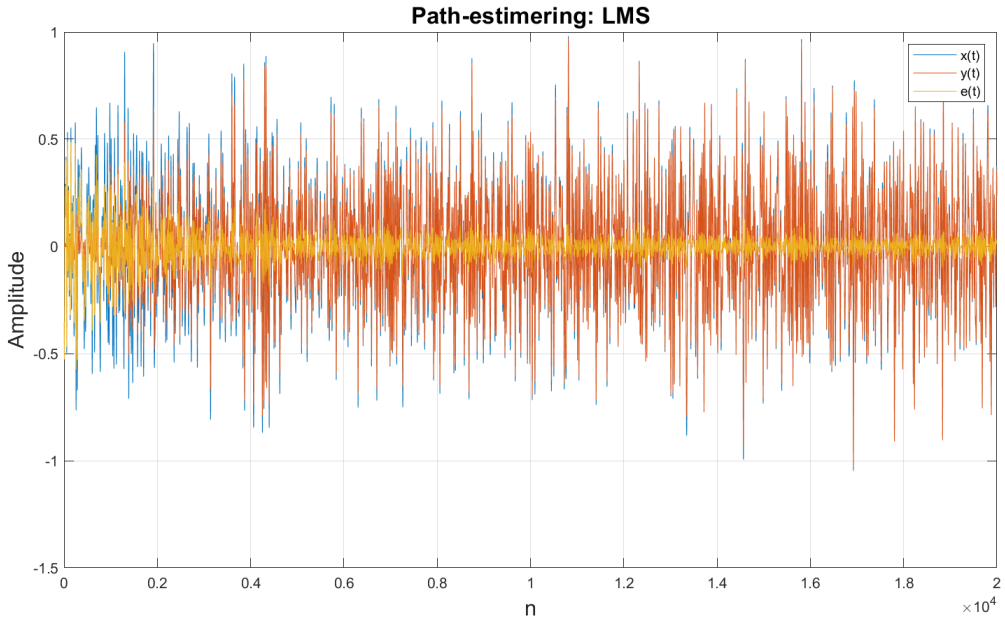
LMS-algoritmen er implementeret som en funktion i MATLAB, se Listing 1. Funktionen udfører først filtreringen af input-signalet med vægtene initieret til 0, hvorefter error udregnes, og til sidst adapterer funktionen vægtene. Dvs. hhv. formlerne fra teorien (2), (1) og (3).

```
function [y, e, w] = LMS(x , d , N, M, mu)
    % LMS takes 5 parameters:
    %   x:  Input signal
    %   d:  Desired signal
    %   N:  Number of samples
    %   M:  Number of filter-taps
    %   mu: Adaption stepsize
    %
    % And returns 3 vectors [y, e, w]
    %   y: Filtered signal
    %   e: Error signal
    %   w: Filter-weights

    w = zeros(1,M); % Weights
    y = zeros(1,N); % Filtreret signal
    e = zeros(1,N); % error-signal

    for n=1:N
```

```
        % Filtering
        for m=1:M
            if (n > m)
                y(n) = y(n) + quantize(w(m)*x(n-m+1),32);
            end
            y(n) = quantize(y(n),32);
        end

        % Error calculation
        e(n) = quantize(d(n) - y(n),16);

        % Weight adaption
        for m=1:M
            if (n > m)
                w(m) = w(m) + quantize(mu*e(n)*x(n-m),32);
            end
            w(m) = quantize(w(m),32);
        end
    end
end
```

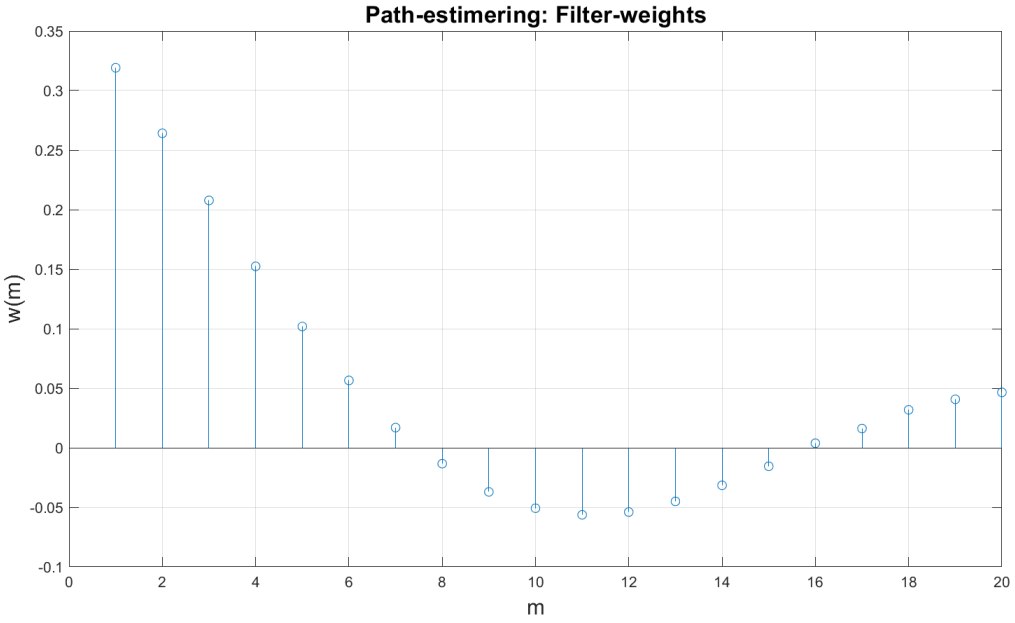
Listing 1 - LMS-funktion i MATLAB

Formålet er som sagt genskabe den hvide støj med LMS-algoritmen, således filter-vægtene simulerer det secondary path, $S(s)$, og det akustiske feedback path $F(s)$, beskrevet i afsnit 2.2. På Figur 8 ses et plot af LMS-funktionens outputs, error-signalet og det filtrerede signal, sammen med input-signalet. Filteret har 20 taps og LMS benytter en stepstørrelse, μ, på 0.002. Disse værdier er fundet gennem flere forsøg, hvor der findes en sammenhæng mellem antallet af taps og stepstørrelsen: Jo flere taps jo mindre stepstørrelse. Det er fordelagtigt at holde antallet af filter-taps til et minimum, således hastigheden optimeres i det endelige system, hvorfor der vælges 20 taps med en stepstørrelse på 0.002.

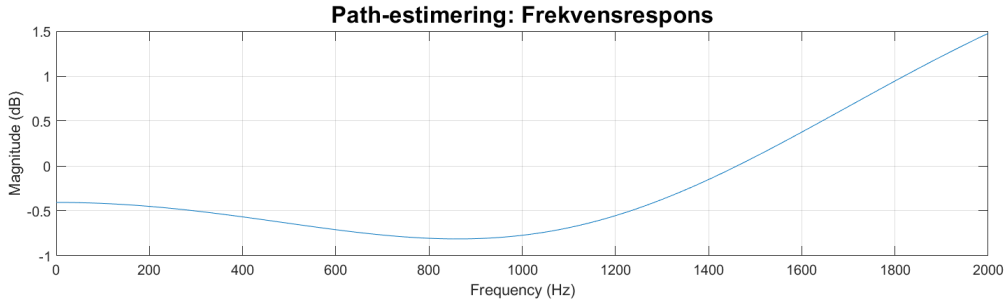


Figur 8 - Plot af input-signal, filtrerede signal og error-signalet fra LMS til path-estimering

Filtervægtene er plottet på Figur 9 og en frekvensrespons af filteret kan ses på Figur 10. Denne metode til estimering af overføringskarakteristikkerne, $\hat{S}(z)$ og $\hat{F}(z)$, er ens for begge, som vist på Figur 4.



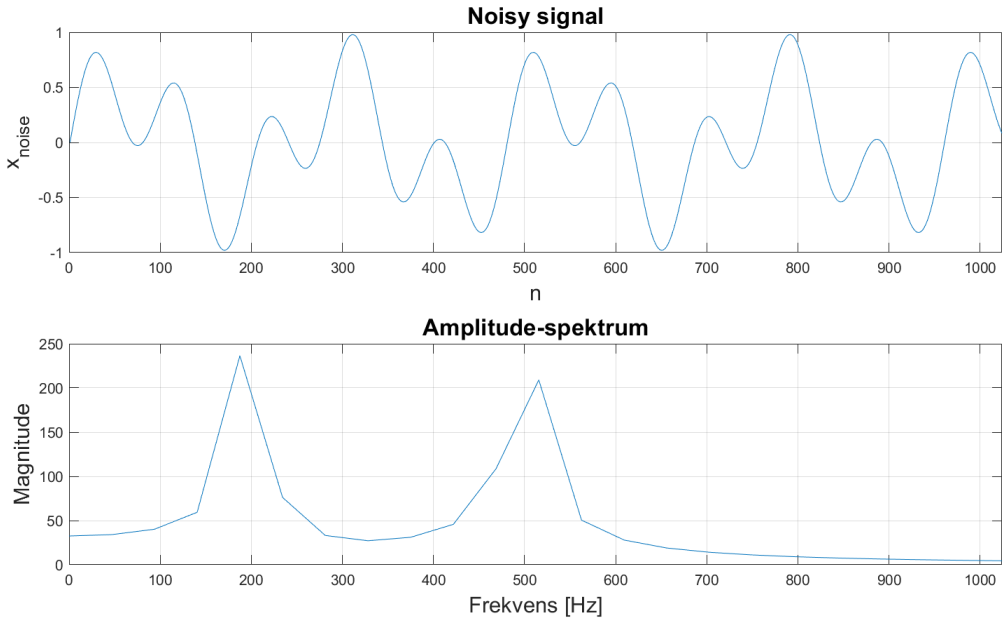
Figur 9 - Filter-vægtene til path-estimeringen



Figur 10 - Frekvensrespons af path-estimerings filter

4.2 ANC – Primary path estimation

Primary path estimation refererer til selve genereringen af et antistøjssignal med ANC-systemet, som vist på Figur 3. På Figur 11 ses et genereret støjsignal. Signalet består af en 200 Hz og en 500 Hz sinusbølge, som også fremgår af frekvenspektret på nederste del af Figur 11. Her benyttes der også en samplingsfrekvens på 48kHz, men der tages kun 1024 samples.



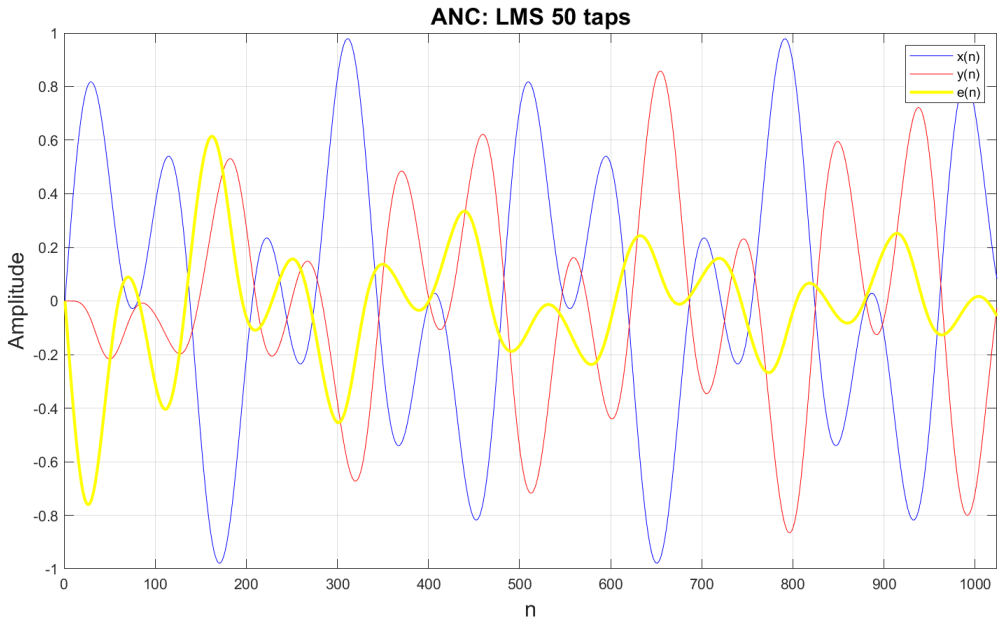
Figur 11 - Testsignal som består af 200 Hz og 500 Hz sinuskurver.

På Figur 12 ses inputsignalet x(n), det filtreret signal y(n) og den samlet fejl e(n). Formålet er at estimere y(n) til at være -x(n), således at e(n) ≈ 0.

$$e(n) = -x(n) - y(n)$$

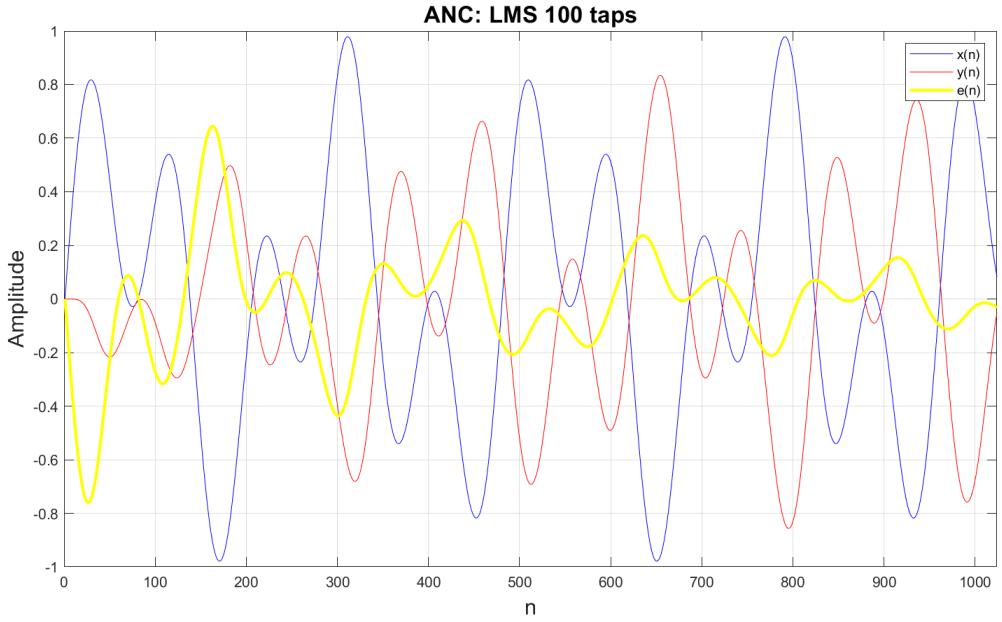
Ved at få y(n) til at være -x(n), vil signalerne lave destruktiv interferens. LMS-filteret som ses på Figur 12, bruger følgende $m_{filtertaps}$ og en μ:

$$m_{filtertaps} = 50 \qquad \mu = 0.001$$

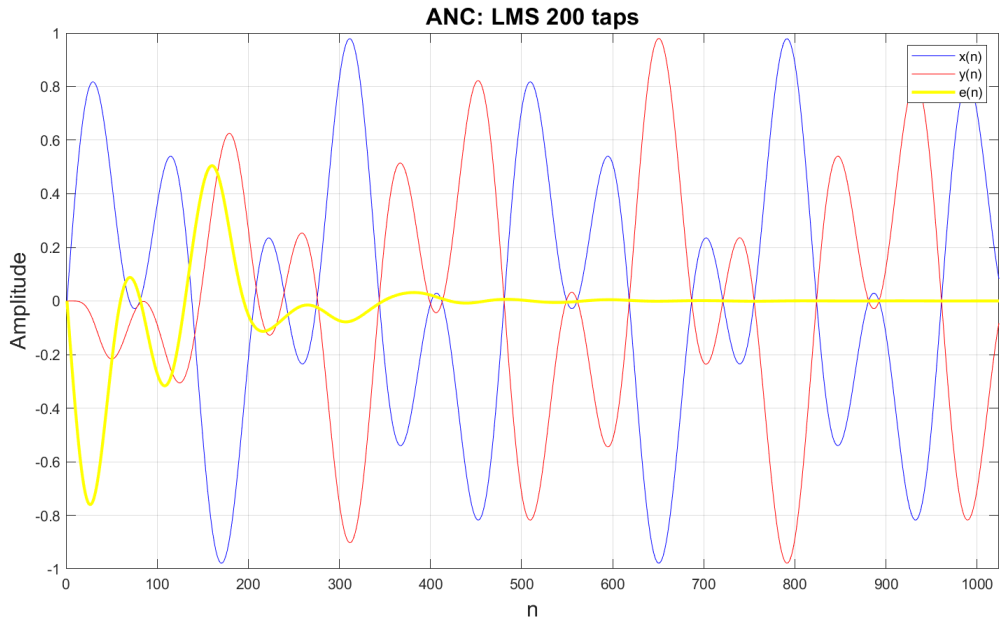


Figur 12 - Sinus-toner dæmpet med 50 taps LMS filter i MATLAB. x(n) er inputsignalet, altså sinus-støjen, y(n) er outputtet af filteret, dvs. antistøjs-signalet, og e(n) er error-signalet.

Hvis der til gengæld benyttes 100 eller 200 taps i LMS-filteret, ses det at indsvingningstiden, dvs. hvor lang tid der går før error-signalet bliver stabilt, bliver meget mindre. Desuden bliver error-signalet også meget mindre, da outputtet er mere præcist. Det skal bemærkes, at det ikke var nødvendigt at ændre μ på trods af ændringen i antal taps. Se Figur 13 og Figur 14. Det vil altså være fordelagtigt at bruge et 200 taps LMS-filter. Dette vil dog kræve en del flere clockcykler i en processor end et 50 taps filter.



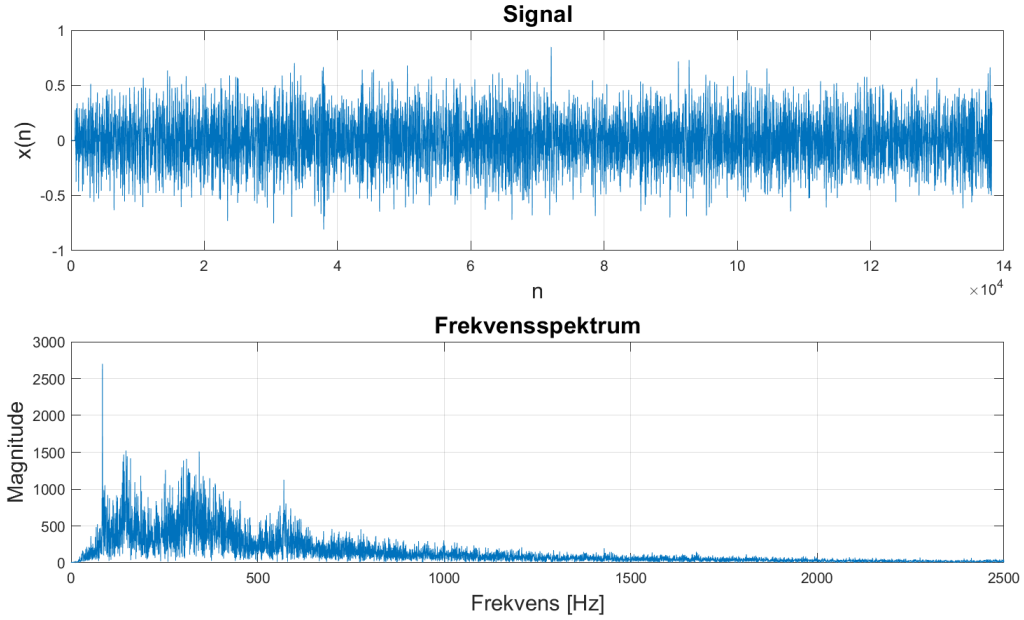
Figur 13 - Sinus-toner dæmpet med 100 taps LMS filter i MATLAB. x(n) er inputsignalet, altså sinus-støjen, y(n) er outputtet af filteret, dvs. antistøjs-signalet, og e(n) er error-signalet.



Figur 14 - Sinus-toner dæmpet med 200 taps LMS filter i MATLAB. x(n) er inputsignalet, altså sinus-støjen, y(n) er outputtet af filte-ret, dvs. antistøjs-signalet, og e(n) er error-signalet.

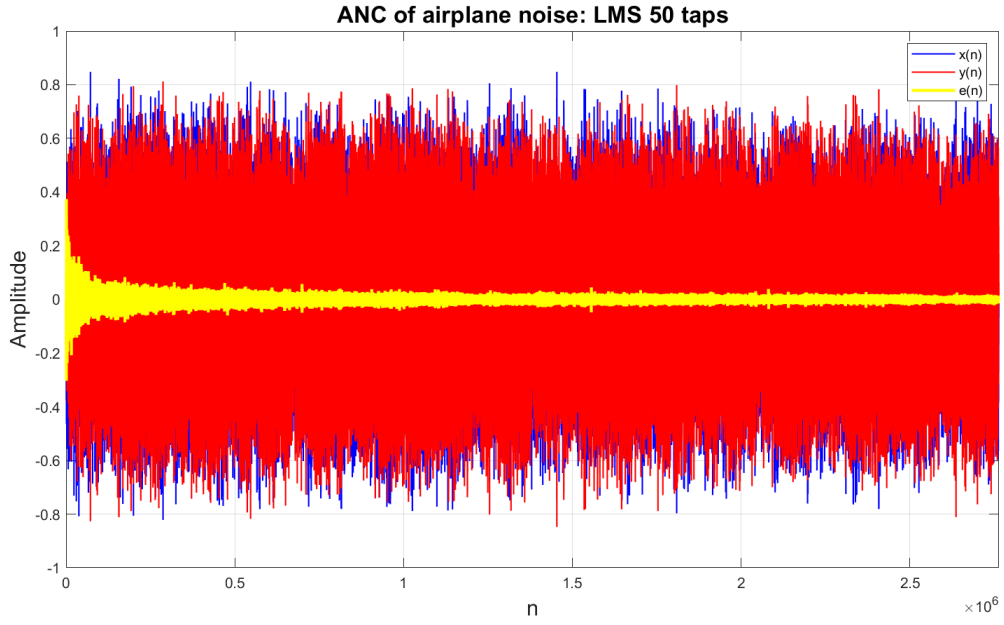
4.2.1 Et mindre forudsigeligt signal

Formålet med et ANC-headset er normalt ikke at fjerne enkelte sinus-toner, men derimod mindre forudsigelige signalet, som fx støjen fra et passagerfly [7] i kabinen. Derfor forsøges det nu at støjudligne et signal optaget inde i passager-kabinen i et passagerfly. Signalet kan ses på Figur 15. Her ses det at signalet næsten udelukkende indeholder frekvenser mellem 1 Hz og 1000 Hz. Dette er altså støj som konventionelle høretelefoner vil have svært ved at dæmpe udelukkende ved brug af ørekopperne.

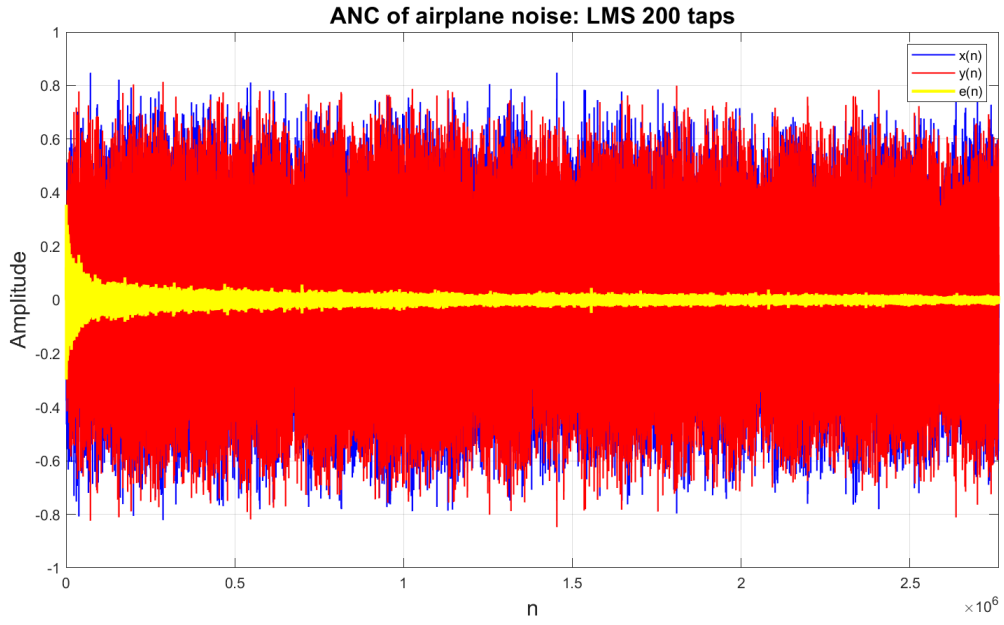


Figur 15 - Støjsignal optaget i et passagerfly

Dette signal føres ind den implementerede MATLAB-funktion LMS og plottes. Der forsøges både med et 50 taps LMS-filter, se Figur 16, og med et 200 taps LMS-filter, Figur 17.

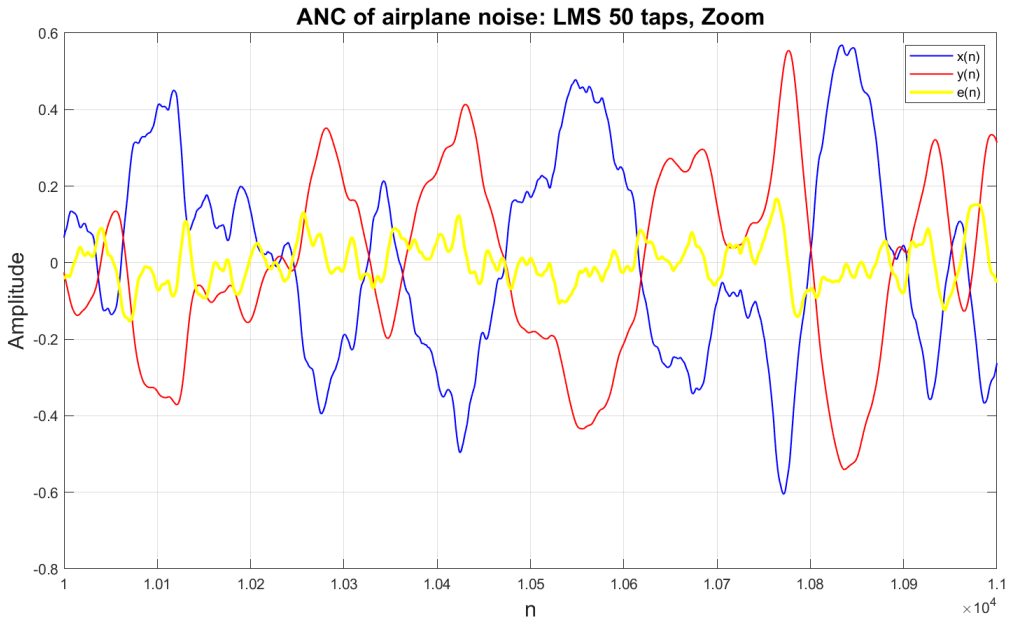


Figur 16 - Støjsignal optaget i et passagerfly dæmpet med 50 taps LMS filter i MATLAB. x(n) er inputsignalet, altså sinus-støjen, y(n) er outputtet af filteret, dvs. antistøjs-signalet, og e(n) er error-signalet.

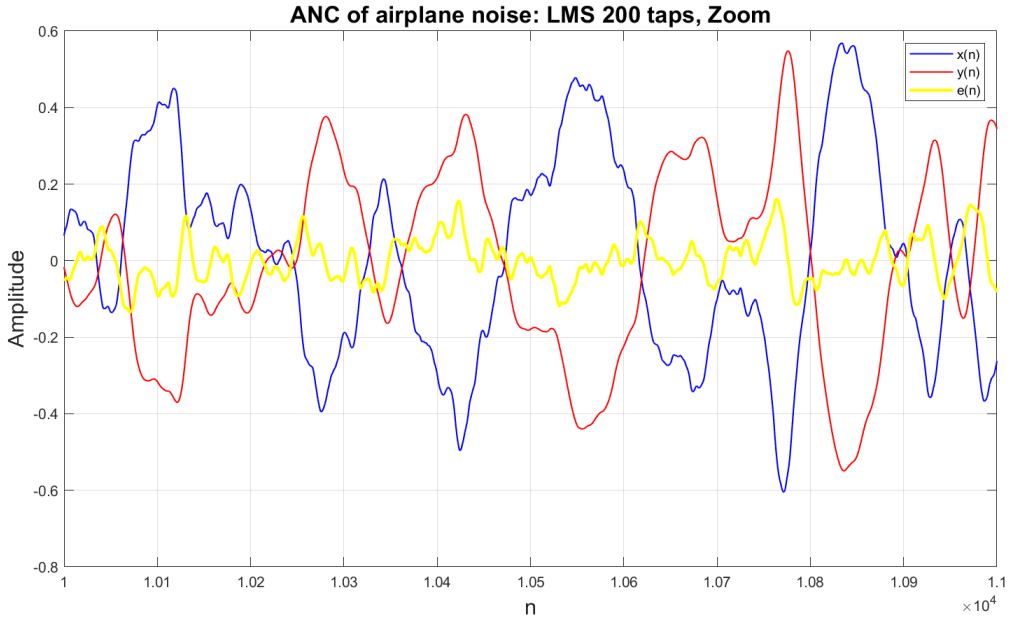


Figur 17 - Støjsignal optaget i et passagerfly dæmpet med 200 taps LMS filter i MATLAB. x(n) er inputsignalet, altså sinus-støjen, y(n) er outputtet af filteret, dvs. antistøjs-signalet, og e(n) er error-signalet.

Det er tydeligt at se på error-signalet at flystøjen bliver dæmpet, mens et zoom på plottet viser, at selve antistøjs-signalets agerer som forventet. Desuden er det svært at se forskel på 50- og 200-taps LMS-filteret. Dog kan det ses på de forstørrede plots, at et 200-taps LMS-filter er en smule mere detaljeret. Dette vurderes dog at være en negligibel forskel. Se Figur 18 og Figur 19.



Figur 18 - Zoom af plottet på Figur 16



Figur 19 - Zoom af plottet på Figur 17

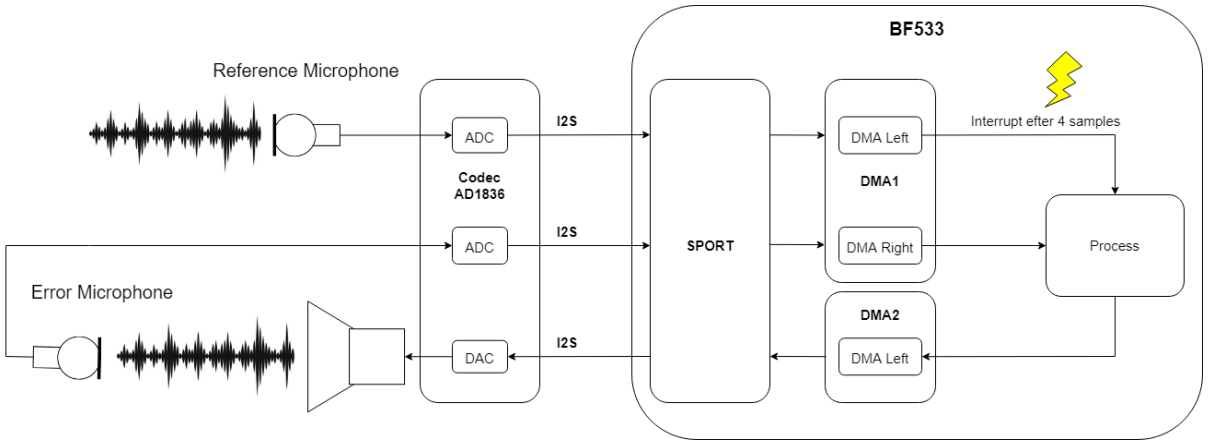
5 Design og Implementering

I dette afsnit vil design og implementering af ANC-systemet på BF533 blive gennemgået.

5.1 Systemdesign

På Figur 20 ses et konceptdiagram over opbygningen af systemet. I systemet anvendes der to DMA'er. DMA1 sørger for at flytte indkomne data fra SPORT-kanalen, modtaget fra codec, over i on-chip hukommelsen. Med en blocksize på 4, triggers et interrupt når DMA'en er fyldt, dvs. efter 4 samples. I denne interrupt rutine eksekveres funktionen `Process`. Efterfølgende sørger DMA2 for at flytte det processerede data fra Blackfins memory til en SPORT-kanal, hvorfra data sendes til Codeccen via I2S. De to ADC'er vil hver især have en samplingsfrekvens på 48 kHz. Både de to ADC'er og DAC'en der bliver brugt, har en opløsning på 16 bit. Med en opløsning på 16 bit, vil systemet have et dynamikområde på:

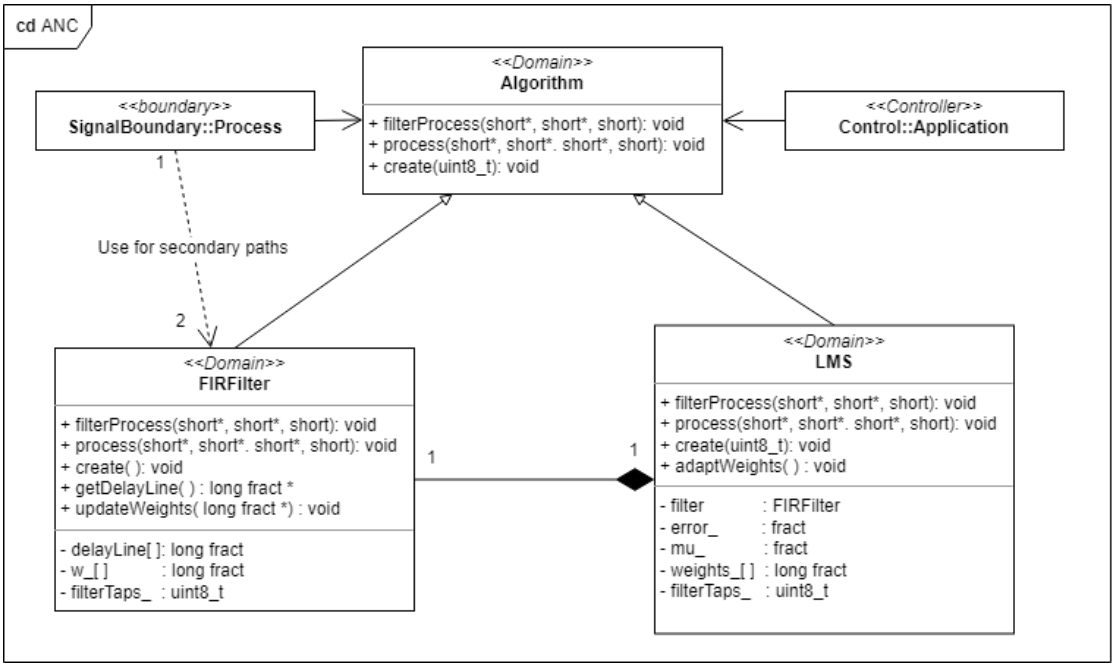
$$dynamikområde = 20 \cdot \log_{10}(2^{16}) \approx 96\ dB$$



Figur 20 - Konceptdiagram over opbygningen af systemet

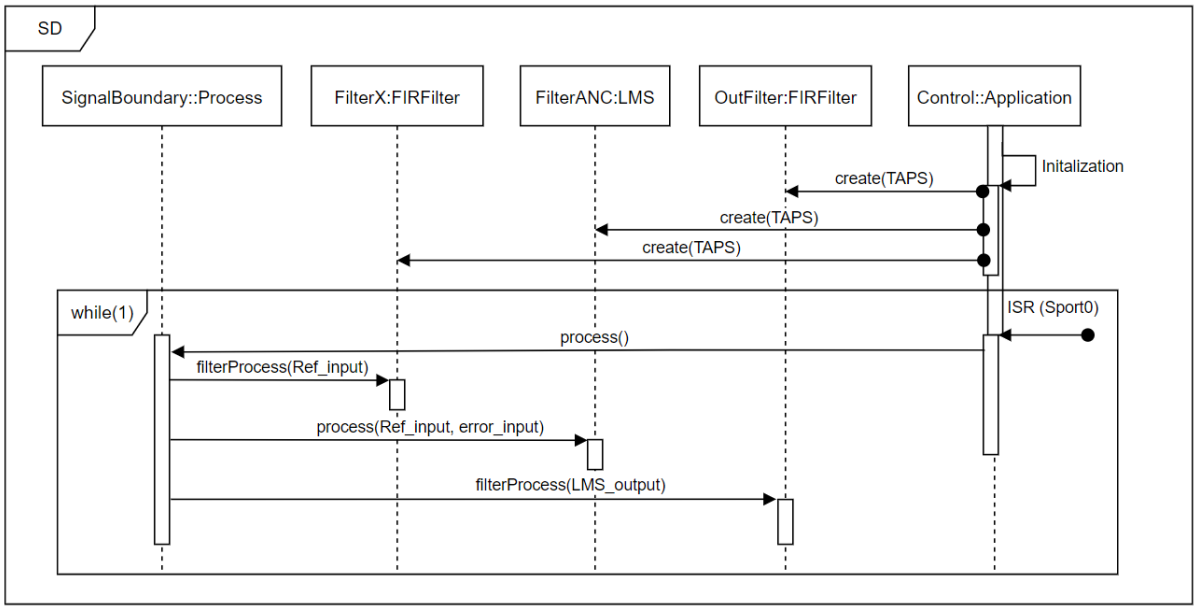
5.2 Softwaredesign

På Figur 21 ses klassediagrammet for applikationen ANC. Domain-klassen `Algorithm` er basisklasse, hvor `LMS` og `FIRFilter` er to afledte klasser. Når DMA'en er fyldt og der sker et interrupt kaldes funktionen *process* i signal-boundary-klassen `Process`. I denne funktion kaldes domainklassen Algorithm's funktion `process` som nedarver til funktionen `process` i både klassen `LMS` og `FIRFilter`. Funktionen `process` er kun implementeret i klassen `FIRFilter` som en placeholder, ligesom funktionen `filterProcess` kun er en placeholder i klassen `LMS`, men er implementeret i `FIRFilter`. Denne opbygning er designet således der kan oprettes FIRFilter objekter uden et dertilhørende `LMS`-objekt, netop med secondary-path og feedback-path filtrene for øje. `LMS`-klassen har en komposition til et `FIRFilter` objekt, således `LMS`-funktionen `process` kan benytte FIRFilter's funktion `filterProcess`.



Figur 21 - Klassediagram for applikationen ANC

Da der ikke er foretaget nogen måling og systemidentifikation af overføringsfunktionerne for secondary path, og feedback path er disse to filtre implementeret med nogle dummy-koefficienter, der danner udgangspunkt for en lavpas-karakteristik. Dette er gjort for at kunne analysere cycle- og memoryforbrug ud fra et mere realistisk udgangspunkt. På Figur 22 ses et simplificeret sekvensdiagram, der illustrerer hvorledes programmet eksekverer i hovedtræk.



Figur 22 - Forsimplet sekvensdiagram af programeksekveringen

Fra applikation kaldes alt initierende kode, som blandt andet sørger for at kalde funktionen `create()` i de `Algorithm`-objekter, som skal anvendes. Herefter følger en uendelig while-løkke, hvori SPORT_ISR trigger et funktionskald til `Process`, hvori Algorithm-objekternes `process` funktioner bliver kaldt.

5.3 Implementering

Funktionen `process` i LMS ser ud som set nedenfor:

```
void LMS::process(short* refInput, short * errorInput, short* output, short len)
{
	fract *errorPointer = (fract *)errorInput;

	for(int i = 0; i < len; i++)
	{
		m_FIRFilter.filterProcess(&refInput[i], &output[i], 1);

		error_ = errorPointer[i];

		adaptWeights();

		m_FIRFilter.updateWeights(weights_);
	}
 }
```

Listing 2 - Implementeringen af LMS-klassens funktion process()

Lig MATLAB-implementeringen følger funktionen følgende struktur:

- Filtrer inputsignal
- Bestem fejl (Her indhentet fra mikrofon)
- Udregn nye vægte
- Opdater vægte i FIR-filteret
- Gentag BLOCK_SIZE antal samples (4)

Funktionen `updateWeights` følger den matematiske formel fra teoriafsnittet.

```
void LMS::adaptWeights()
{
	long fract * delayLine = m_FIRFilter.getDelayLine();
	long fract wT = 0;
	for(uint8_t i = 0; i < filterTaps_; i++)
	{
	    weights_[i] = weights_[i] + (mu_ * error_ * delayLine[i]);
	}
}
```

Listing 3 - Implementeringen af LMS-klassens funktion adaptWeights()

FIRFilter-klassens `filterProcess` er i udgangspunktet implementeret på simpel form med en normal lineær buffer. Koden til dette kan ses nedenfor.

```
void FIRFilter::filterProcess(short* input, short* output, short len)
{
	fract *x = (fract *)input;
	fract *y = (fract *)output;
	long fract yTmp = 0;

	for(int n = 0; n < len; n++){

		for(int m = filter_taps_ - 1;  m > 0; m--)    {
			delayLine[m] = delayLine[m-1];
		}

		// Delaylines
		delayLine[0] = x[n];
```

```
                // Convolution
                yTmp = 0.0lr;

                for(int m = 0; m < filter_taps_; m++){
                        yTmp += (w[m] * delayLine[m]);
                }

                y[n] = (fract)yTmp;
        }
}
```

Listing 4 - Implementering af FIRFilter-klassens funktion filterProcess()

Kort opridset følger funktionen følgende:

- Shift alle samples i delayline ét sample bagud.
- Indsæt seneste sample på plads 0.
- Nulstil variablen yTmp.
- Fold vægte med delayline til udregning af output.
- Gentag `BLOCK_SIZE` antal samples (4)

5.3.1 Datatyper

Jævnført krav R9 om fixed-point implementering er det valgt at implementere filtrene med native fixed point typerne fract og long fract. Fract er en 16-bit datatype, som repræsenterer formatet 1.15, hvor long-fract er en tilsvarende 32-bit udgave, som tillader større opløsning (1.31). Fordelen ved fixed-point implementeringer er, at de er langt hurtigere og mindre krævende end floating point. Dette er væsentligt i batteridrevne applikationer, hvor optimering af cykel- og strømforbrug er essentielt.

$$fract_{res} = 2^{-15}$$

$$long\ fract_{res} = 2^{-31}$$

I den første iteration blev der udelukkende anvendt typen fract. Dette viste sig dog at være et problem, ifm. mere længerevarende tests (> 1000 samples), da det i nogen særtilfælde resulterede i en for lille opløsning, der ikke gjorde det muligt at repræsentere alle vægte, x-værdier og udregnede y-værdier akkurat nok. Denne fejl endte med at kaskade over flere samples fordi udregningen af vægte begyndte at fejle mere og mere. Løsning var at bruge long fracts til at gemme delayLine, vægte og det midlertidige y-signal for at sikrer at de korrekte vægte altid blev udregnet. Herudover anvendes der casting for at sikre korrekt overførsel mellem input/output buffere og udregnede værdier. Da short og fract begge er 16-bit datatyper behøves der ikke anvendes bitshift.

5.4 Cykleforbrug og Optimering

I et forsøg på at optimere koden er der implementeret en udgave af `filterProcess`(), hvor der i stedet anvendes en cirkulær buffer til at gemme og uddrage data fra delaylinjen. Dette skyldes, at delaylinjen fra den anden implementering virker til at være den mest åbenlyse bottleneck i algoritmen. Til dette anvendes funktionen circptr, som er en libraryfunktion, der sørger for korrekt pointer-inkrementering hele vejen rundt i bufferen. Koden til dette kan ses nedenfor:

```
void FIRFilter::filterProcess(short* input, short* output, short len)
{
	fract *x = (fract*)input;
	fract *y = (fract*)output;
	long fract y_ = 0; //9.31

	int n = 0; int m = 0;
	long fract * delay; //1.31

	for(n = 0; n < len; n++){

		*cir_ptr = x[n];

		cir_ptr = (long fract *)circptr(cir_ptr, sizeof(long fract), delay-
Line,    sizeof(long fract)*filter_taps_);
		delay = (long fract *)cir_ptr;

		y_ = 0.0lr;

		for(m = 0; m < filter_taps_; m++){
			y_ += w[m] * (*delay);
			delay = (long fract *)circptr(delay, sizeof(long fract),
delayLine, sizeof(long fract)*filter_taps_);
		}

		y[n] = (fract)y_;
	}
}
```

Listing 5 - Implementering af FIRFilter-klassens funktion filterProcess(), hvor der benyttes en circular-buffer frem for den lineærer.

På samme måde som før anvendes der casting til og fra long fract, for at sikre korrekte signaler.

På Tabel 1 ses antallet af clock-cycles, målt i crosscore, med forskellige kombinationer af optimizers og implementeringer. Det ses at uden compiler-optimering, med normale buffers og et 200-taps LMS-filter tager en gennemkørsel af `process` 130668 clock-cycles, hvilket er for mange i forhold til $Clock_{4-max}$ (udregnet i afsnit 2.3.1 til 45000 clock-cycles), hvorfor den er markeret som rød i tabellen. Med compiler-optimering sat til fastest, tager samme processering kun 32985 clock-cycles, hvilket er under grænsen og den er derfor markeret som grøn i tabellen. Det skal bemærkes at circular-buffers gør implementeringen hurtigere, men kun i de tilfælde hvor compiler-optimeringen fastest ikke er sat til. Dette skyldes måske, at compileren med fastest optimizer vælgere at udnytte platformens hardware cirkulære buffer i maskinkoden.

Implementering	Optimering	Antal clock cycles
LMS 200 tap FilterX 20 tap Outfilter 53 tap Normal Buffers	Fastest	32985
LMS 200 tap FilterX 20 tap Outfilter 53 tap Circular Buffers	Fastest	36632
LMS 200 tap FilterX 20 tap Outfilter 53 tap Normal Buffers	Ingen	130668
LMS 200 tap FilterX 20 tap Outfilter 53 tap Circular Buffers	Ingen	124632
LMS 50 tap FilterX 20 tap Outfilter 53 tap Normal Buffers	Fastest	11973
LMS 50 tap FilterX 20 tap Outfilter 53 tap Circular Buffers	Fastest	13533
LMS 50 tap FilterX 20 tap Outfilter 53 tap Normal Buffers	Ingen	48274
LMS 50 tap FilterX 20 tap Outfilter 53 tap Circular Buffers	Ingen	45826

Tabel 1 - Antal clock-cycles med og uden optimering. Grøn markering betyder at antallet af clock-cycles er inden for grænsen, imens en rød markering betyder at mængden af clock-cycles er for stor.

Ud fra resultaterne i Tabel 1 besluttes det at implementeringen med normal buffer, fastest compiler-optimering og et 200-taps LMS-filter er den ideelle implementering at benytte. Processering af 4 samples vil derved tage:

$$\frac{1}{540\ MHz} \cdot 32984\ cycles = 0.061\ ms$$

Dette er dermed implementeringen som benyttes fremover, bl.a. i afsnit 6, hvor implementeringen testes.

5.4.1 System Load

Der kigges også på hvor meget systemet belastes. Dette gøres ud fra det maksimale antal cykler vi har per block, som udregnet i 2.3.1 Hastighedsberegninger:

$$Clock_{4-max} = 45000 \text{ cycles}$$

Med den førnævnte implementering (200 tap LMS) betyder det en systemload på:

$$CPU_{LOAD}\% = \frac{32985}{Clock_{4-max}} \cdot 100\% = 73.3\%$$

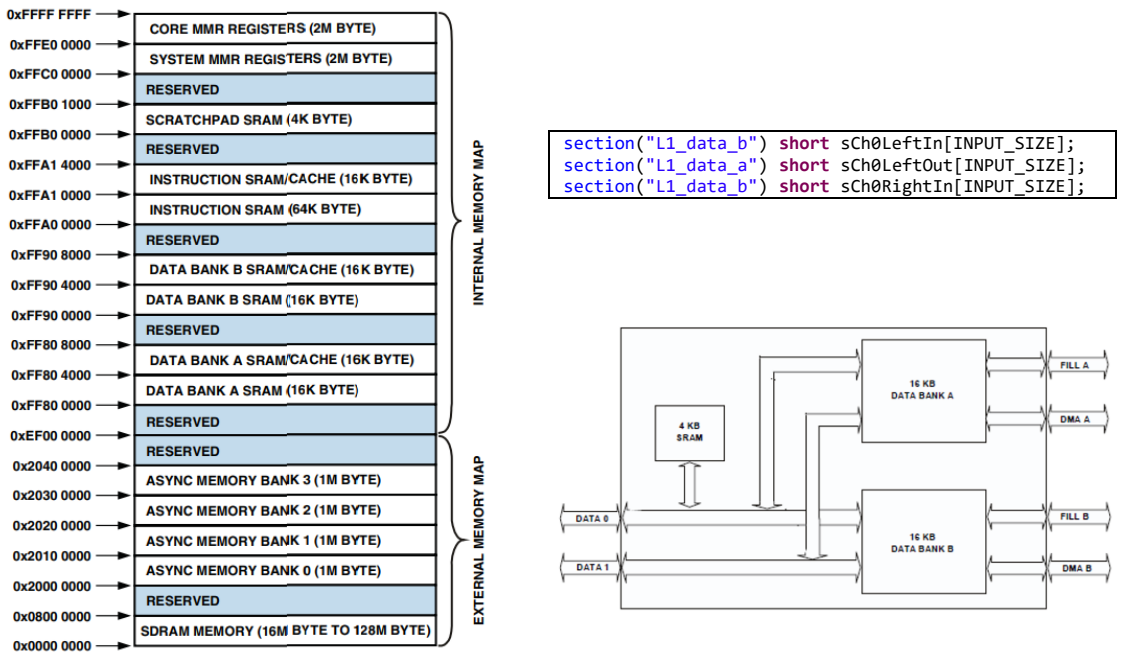
Eller med den simplere implementering (50 tap LMS):

$$CPU_{LOAD}\% = \frac{11973}{N_{\max\ \_block4}} \cdot 100\% = 26.6\%$$

Der er altså et rimeligt rum for at implementere yderligere features, og derudover vides det at det er på LMS-filterets længde, CPU_LOAD hurtigst kan justeres.

5.5 Memoryforbrug

Memoryforbruget er ligeledes blevet undersøgt. Der er to memorytyper, som kan være interessante at kigge på. Disse omtales i CrossCore dokumentationen som *Data Memory* og *Code Memory.* BF533 memorystruktur ser ud som set på Figur 23 nedenfor. Datahukommelsen, der blandt anvendes til filterkoefficienter, samplebuffers mm. placeres i enten databank A eller B (0xFF80/0xFF90), som hver især inklusive cache er 32 kB stor. Code memory placeres i *Instruction SRAM* (0xFFA0), som inklusiv cache er 80 kB.



Figur 23 - BF533 Memory arkitektur [1]

Databankerne tillader at man kan mindske cycleforbruget ved overførsel af data til og fra bufferne. Med separate banker, kan man outputte og inputte data paralleliseret. Dette kan defineres i koden, som set i kodesnippet ovenfor.

5.5.1 Datamemory

I CrossCore kan linkeren sættes op til at generere en XML-fil, som indeholder memory mappet for den applikation man kompilerer. I denne fremgår det, at de to banks er fyldt ud som set på Figur 24 nedenfor. (200-taps LMS-implementering):

Memory *MEM_L1_DATA_A*

Output section	Type	Start address	Size in words	Reserved size in words
L1_data_a_prio0	SHT_PROGBITS	0xff800000	0x610	0x800
L1_data_a_prio3	SHT_PROGBITS	0xff800610	0x651	0x0
L1_data_a_bsz_prio3	SHT_NOBITS	0xff800c90	0x1810	0x0
stack_and_heap_L1_data_a	SHT_NOBITS	0xff8024a0	0x0	0x5314

Memory *MEM_L1_DATA_B*

Output section	Type	Start address	Size in words	Reserved size in words
L1_data_b_prio0	SHT_PROGBITS	0xff900000	0x50	0x800
L1_data_b_prio0_tables	SHT_PROGBITS	0xff900050	0x1c	0x0
stack_and_heap_L1_data_b	SHT_NOBITS	0xff90006c	0x0	0x7748

Symbol	Demangled name	Address	Size	Binding
_FilterANC	FilterANC	0xff800c90	0x974	GLOBAL
_FilterX	FilterX	0xff801604	0x648	GLOBAL
_LowPass	LowPass	0xff801c4c	0x648	GLOBAL

Figur 24 - Data memory banks

Der er dermed stadig en rimelig mængde plads tilbage. Følgende er aflæst ud fra *stack_and_heap_L1_data_a/b* startadresse:

$$DATA_A = 0x24A0 = 9376\ words = 18752\ bytes \rightarrow \frac{DATA_A}{32000\ bytes} = 58.6\%$$

$$DATA_B = 0x6C = 108\ words = 216\ bytes \rightarrow \frac{DATA_B}{32000\ bytes} 0.675\%$$

5.5.2 Code memory

Afhængig af hvilken type compileroptimering der vælges, kan man justere en anelse på, hvor meget instruktionshukommelse der anvendes. Nedenfor ses med optimizer sat til ***fastest***.

Memory *MEM_L1_CODE*

Output section	Type	Start address	Size in words
L1_code	SHT_PROGBITS	0xffa00000	0x547e

program	0xffa0105c	0x90	src\Algorithm.doj
program	0xffa010ec	0x1c6	src\Application.doj
program	0xffa012b4	0x2d0	src\FIRFilter.doj
program	0xffa01584	0x15a	src\ISR.doj
program	0xffa016e0	0x3a2	src\Init.doj
program	0xffa01a84	0x2ea	src\LMS.doj
program	0xffa01d70	0x2aa	src\Process.doj

*Figur 25 - Code memory med optimizer: **fastest***

Hukommelsesforbruget er dermed:

$$CODESIZE_{fastest} = 0x547E = 21630\ words = 43260\ bytes \rightarrow 54\%\ af\ total$$

Sættes optimizer til smallest ser forbruget ud som set på Figur 26.

Memory *MEM_L1_CODE*

Output section	Type	Start address	Size in words
L1_code	SHT_PROGBITS	0xffa00000	0x524c

program	0xffa0105c	0x86	src\Algorithm.doj
program	0xffa010e4	0x19a	src\Application.doj
program	0xffa01280	0x1c8	src\FIRFilter.doj
program	0xffa01448	0x158	src\ISR.doj
program	0xffa015a0	0x370	src\Init.doj
program	0xffa01910	0x286	src\LMS.doj
program	0xffa01b98	0x254	src\Process.doj

Figur 26 - Code memory med optimizer: ***smallest***

$$CODESIZE_{smallest} = 0x524C = 21068\ words = 42136\ bytes \rightarrow 52\%\ af\ total$$

Her ses det at compileren formår at reducere pladsbrugen en anelse med alle filer. Størrelsen af programmet reduceres samlet set med 232 bytes, hvilket ikke er alverden. Derfor vil det fortsat give bedst mening at anvende *fastest*.

6 Test af implementering

I dette afsnit vil de forskellige resultater for projektet bliver præsenteret.

6.1 MATLAB versus Implementering

Til test af implementeringen benyttes klassen `AlgoTester`. `AlgoTester` sørger for at loade et signal gemt som txt-fil ind i BF533 via USB-forbindelsen, hvorefter signalet kan behandles som var det samplet direkte. Når signalet $y(n)$ er filtreret gemmes resultatet i en anden txt-fil. Ved denne type test er det ikke muligt at sample et error-signal, da error afhænger af implementeringens evne til at støjudligne. Derfor modificeres klassen LMS, således at error bliver beregnet ud fra input-signalet og output-signalet, ligesom i MATLAB implementeringen i Listing 1. På Listing 6 ses `LMS::process` og `LMS::calcError` funktionerne implementeret til BF533.

```
void LMS::process(short* refInput, short * errorInput, short* output, short len)
{
	//fract *errorPointer = (fract *)errorInput;

	for(int i = 0; i < len; i++)
	{
		m_FIRFilter.filterProcess(&refInput[i], &output[i], 1);

	//error_ = errorPointer[i];
		calcError(&refInput[i], &output[i], &errorInput[i]);   // Calcu-
late error

		adaptWeights();

		m_FIRFilter.updateWeights(weights_);
	}
 }
```

```
void LMS::calcError(short* d_signal, short* y_signal, short* e_signal)
{
	fract *d = (fract *)d_signal;
	fract *y = (fract *)y_signal;
	fract *e = (fract *)e_signal;
	error_ = -d[0] - y[0];
	e[0] = error_;
}
```

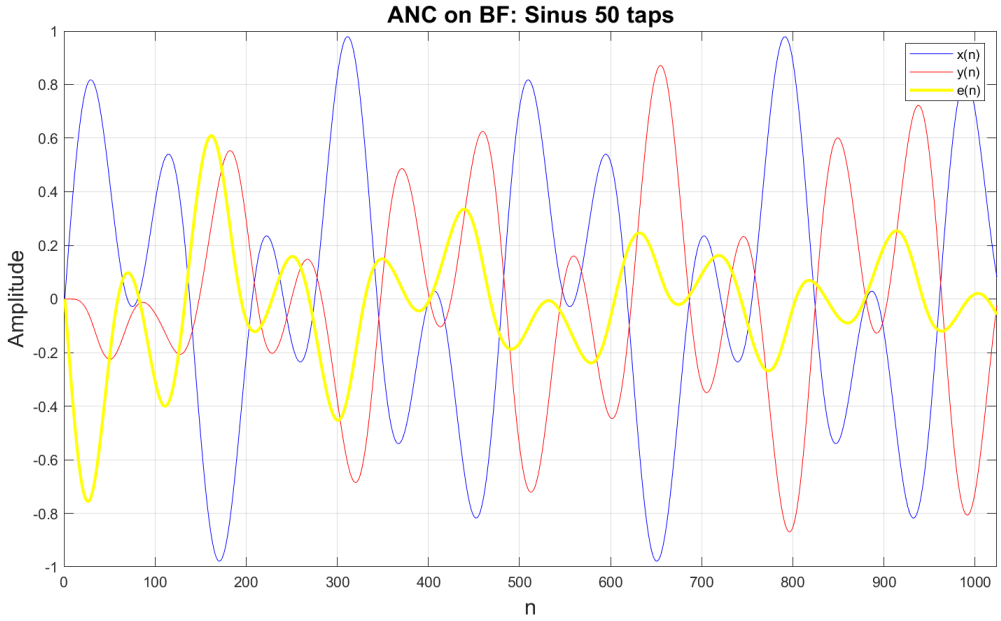
Listing 6 - Implementering af LMS::process og LMS::calcError på BF533. Benyttes til test af algoritme.

Det indlæste signal skal være af størrelsen 1024 samples, hvilket er fint til at eftervise det simple sinus-signal og processeringen deraf på Figur 14. Hvis signalet derimod er længere end 1024, deles input-signalets fil først op i et passende antal filer, hver med en størrelse på 1024, se evt. Appendix Listing 10. Herefter vil et for-loop i Application læse dem en efter en og udskrive filer med error- og output-signalet en efter en med en passende indeksering i filnavnet, se evt. Appendix Listing 8 og Listing 9. De mange filer kan derefter appendes til en samlet fil i MATLAB og verificeres.

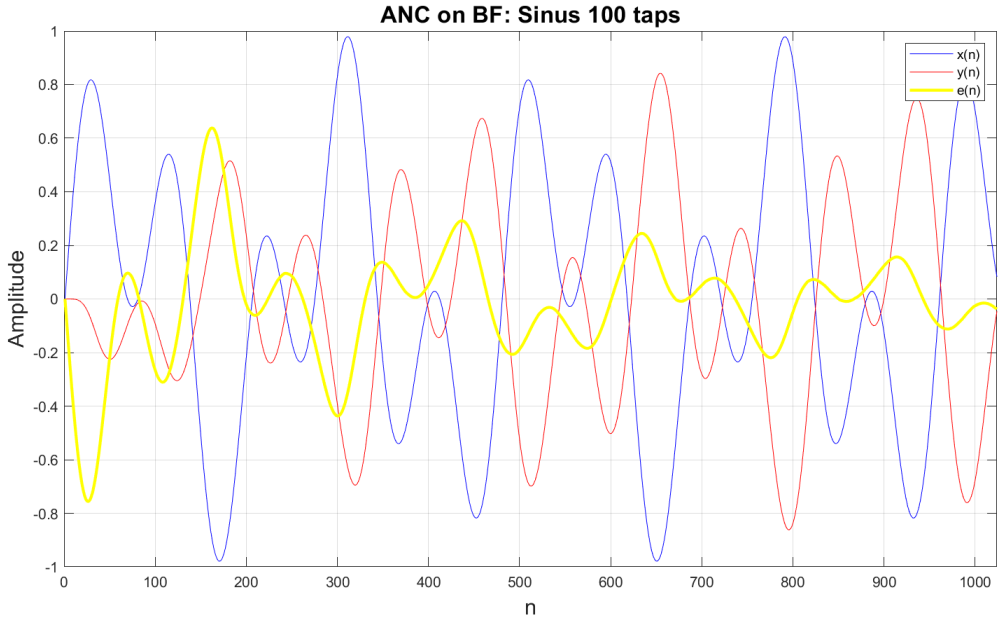
Implementeringen på BF533 benytter i denne test hverken secondary-path eller feedback-path, da disse ikke er estimeret, og dermed ville de påvirke resultatet af testen. Testen udføres ikke med prototype-headsettet, hvorfor akustikken ikke er et problem, der skal løses med de estimerede paths.

6.1.1 Dæmpning af simple sinus-signal

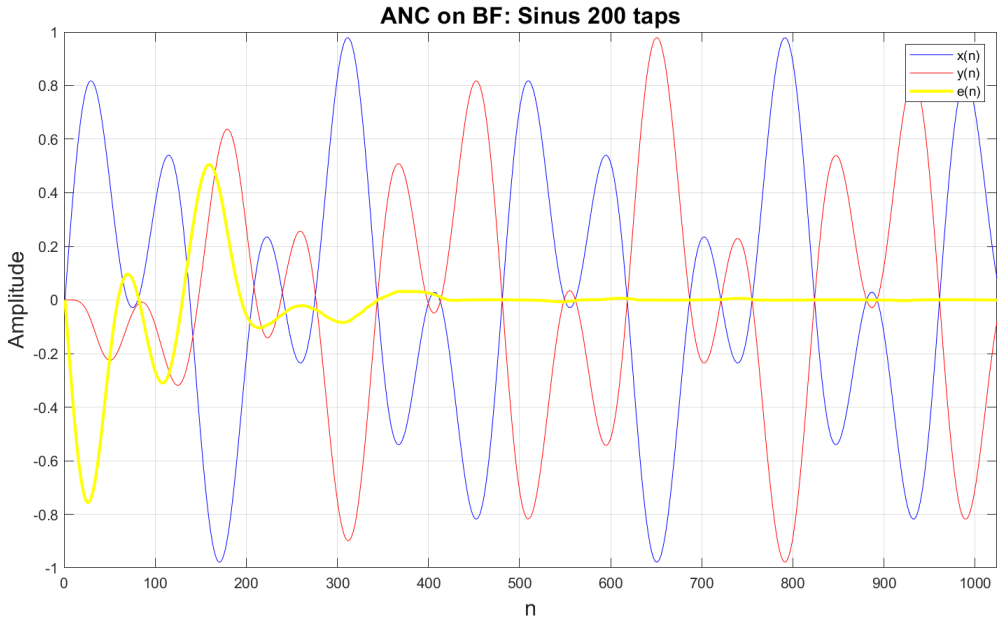
Til verificeringen af implementeringens dæmpning af et simpelt sinus-signal, benyttes det samme sinus-signal som til MATLAB-implementeringen, se Figur 11, på den måde kan de to resultater sammenlignes og implementeringen på BF533 verificeres. På Figur 27, Figur 28 og Figur 29, ses sinus-signalet processeret af BF533 med et hhv. 50-taps, 100-taps og 200-taps LMS-filter. Sammenligner man disse plots med de MATLAB-genererede plots på hhv. Figur 12, Figur 13 og Figur 14 ses der ingen forskel, hvilket betyder at implementeringen på BF533 fungerer som forventet og implementeringen er dermed verificeret for et simpelt sinus-signal.



Figur 27 - De to sinus signal fra Figur 11 processeret med BF533 ved brug af klassen AlgoTester. LMS-filteret har 50 taps.



Figur 28 - De to sinus signal fra Figur 11 processeret med BF533 ved brug af klassen AlgoTester. LMS-filteret har 100 taps.

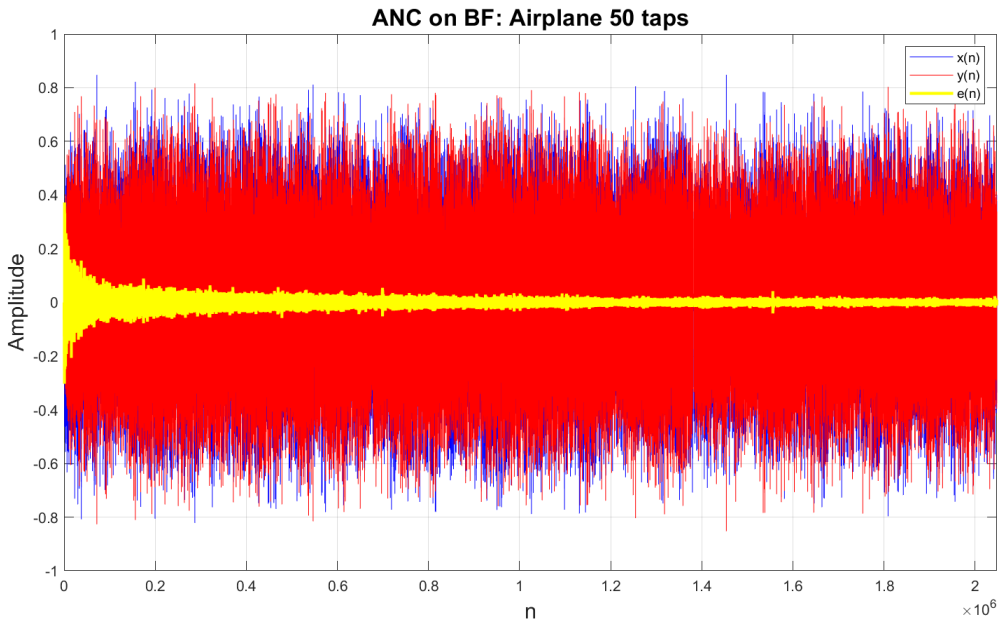


Figur 29 - De to sinus signal fra Figur 11 processeret med BF533 ved brug af klassen AlgoTester. LMS-filteret har 200 taps.

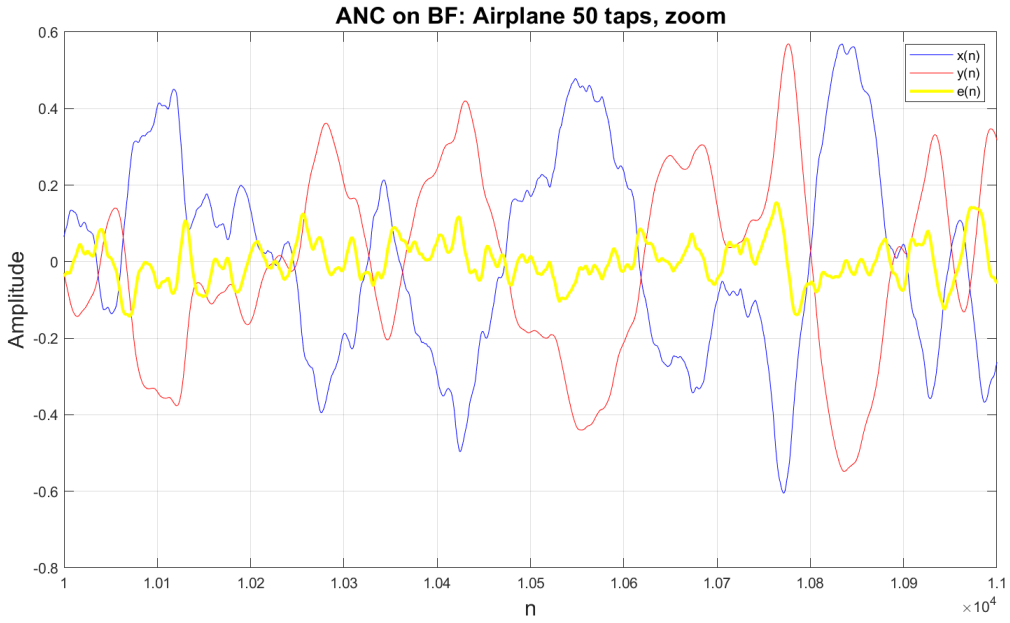
6.1.2 Dæmpning af passagerfly-støj

Implementeringen på BF533 testes også med støjsignalet fra et passagerfly, se evt. Figur 15, det samme signal som blev testet med MATLAB-implementeringen. Dette er en test af systemets funktionalitet over længere tid, med et signal der ikke er lige så forudsigeligt. På Figur 30 ses passagerfly-signalet efter det er processeret med BF533 med et 50 taps LMS-filter. Sammenlignes dette plot med det fra MATLAB-implementeringen på Figur 16, ses der igen ikke stor forskel. Det er dog lettere at sammenligne de to plots, hvis der zoomes ind på sammen område. Figur 31 viser

et zoom på samme samples som på Figur 18, og her er det tydeligt at se, at implementeringen på BF533 processerer som forventet af MATLAB-implementeringen.

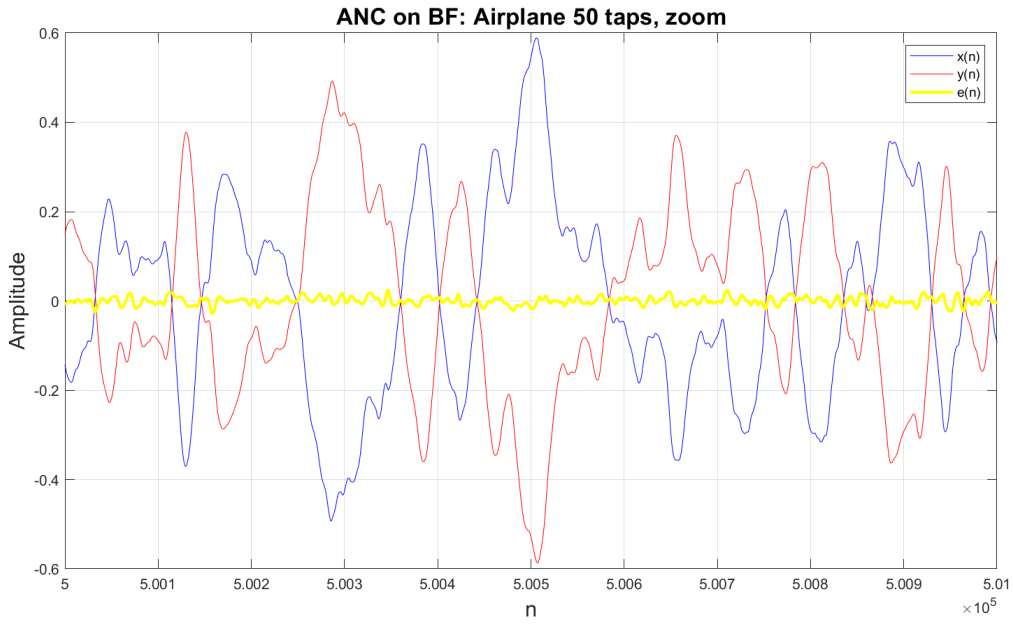


Figur 30 - Støjsignalet fra et passagerfly på Figur 15 processeret med BF533 ved brug af klassen AlgoTester. LMS-filteret har 50 taps.



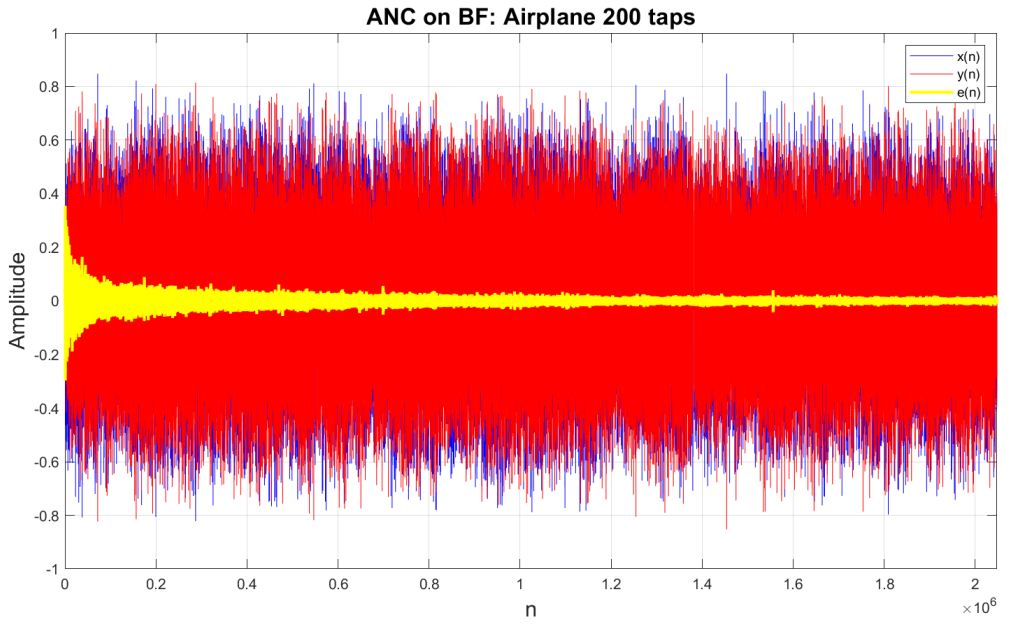
Figur 31 - Støjsignalet fra et passagerfly på Figur 14 processeret med BF533 ved brug af klassen AlgoTester. LMS-filteret har 50 taps. Zoomet ind på samples 10000 til 11000.

På Figur 32 ses endnu et zoom af plottet på Figur 30, her er det dog efter 500.000 samples. Det ses at error-signal er betydeligt lavere, hvilket viser at filter-vægtene altså heromkring er tilpasset meget godt.

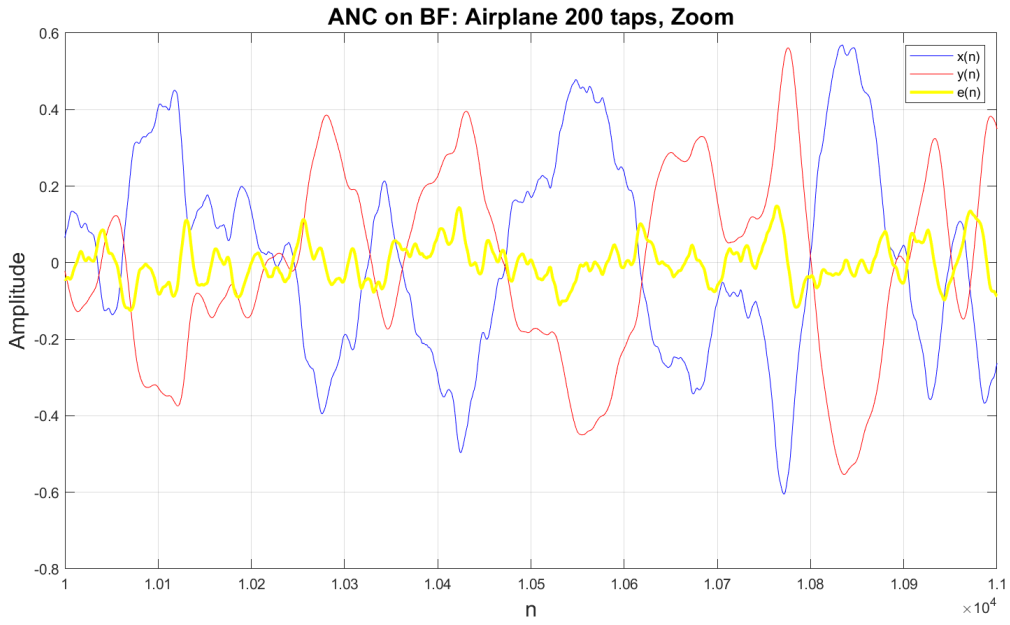


Figur 32 - Støjsignalet fra et passagerfly på Figur 14 processeret med BF533 ved brug af klassen AlgoTester. LMS-filteret har 50 taps. Zoomet ind på samples 500000 til 501000.

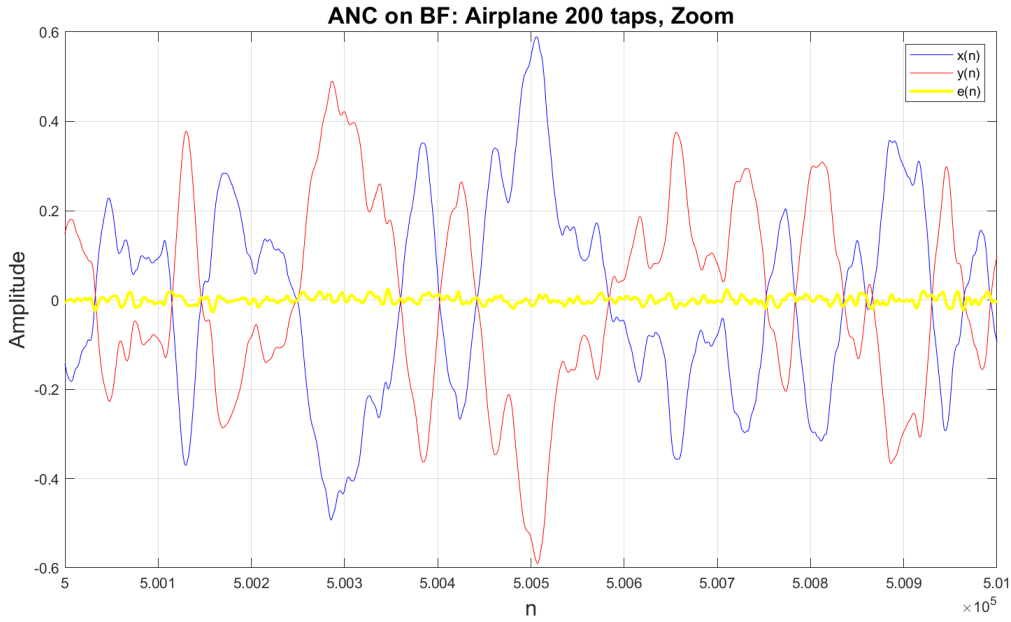
På Figur 33 ses signalet proseceret på BF533, men med et 200 taps LMS-filter, og på Figur 34 og Figur 35, ses de respektive zooms på samme samples som for 50 taps LMS-filteret. Igen agerer filteret som forventet af MATLAB-implementeringen, se evt. Figur 17 og Figur 19 til sammenligning. Det skal bemærkes at sammenligner man 200 taps LMS-filteret og 50 taps LMS-filteret, er der ikke nogen bemærkelsesværdig forskel for passagerflys-støjsignalet. Til gengæld må filteret antages at være hurtigere til at tilpasse sig med 200 taps, hvis man ser på plottene for sinus-tonen, altså Figur 27 og Figur 29.



Figur 33 - Støjsignalet fra et passagerfly på Figur 14 processeret med BF533 ved brug af klassen AlgoTester. LMS-filteret har 200 taps.



Figur 34 - Støjsignalet fra et passagerfly på Figur 14 processeret med BF533 ved brug af klassen AlgoTester. LMS-filteret har 200 taps. Zoomet ind på samples 10000 til 11000.



Figur 35 - Støjsignalet fra et passagerfly på Figur 14 processeret med BF533 ved brug af klassen AlgoTester. LMS-filteret har 200 taps. Zoomet ind på samples 500000 til 501000.

6.2 Dæmpning af forskellige signaler

Igen ved brug af AlgoTest indføres der input-signaler til BF533, hvormed implementeringens evne til at dæmpe forskellige signaler testes.

Der udregnes et Signal-støj-forhold [8] mellem error-signalet, $e(n)$, og støjsignalet, $x(n)$, hvormed det undersøges hvor meget støjen bliver dæmpet. Det besluttes at bruge den sidste halvdel af hele signalet, således indsvingningerne fra starten af LMS-adapteringen af vægtene ikke tages

med. Her er error-signalet nemlig højt, da alle vægte starter med en værdi på 0. Fra passagerfly-eksemplet udregnes følgende:

$$SNR = \frac{\frac{1}{N}\sum_{n=\frac{N}{2}}^{N-1} e(n)^2}{\frac{1}{N}\sum_{n=\frac{N}{2}}^{N-1} x(n)^2} = 0.004$$

Omregnet til dB:

$$SNR_{dB} = 10\ \log_{10}(SNR) = -33.98\ \text{dB}$$

Samme udregning udføres med forskellige signaler, bl.a. de to summerede sinus-toner på 200 Hz og 500 Hz, samt et sweep-signal der går fra 500 Hz til 2000 Hz. Der benyttes LMS-filteret med 200 taps. På Tabel 2 ses resultaterne af disse tests.

Støjsignal	**Dæmpning (SNR_{dB})**
To sinus-toner (200 Hz og 500 Hz)	-50.76 dB
Passagerfly	-33.98 dB
Sweep-signal (500 Hz - 2000 Hz på 0.32 s)	-51.17 dB

Tabel 2 - Udregnede signal-støj-forhold med forskellige input-signaler.

Passagerfly-støjsignalet har en høj dæmpning på -33.98 dB, hvilket er ~-14 dB bedre end kravet. Det skal dog pointeres, at dette er i et ideelt system uden akustisk interferens, hvormed resultatets kvalitet kunne variere.

6.3 Test og opbygning af prototype

I dette afsnit, vil der blive præsenteret de forskellige resultaterne for test af prototypen.

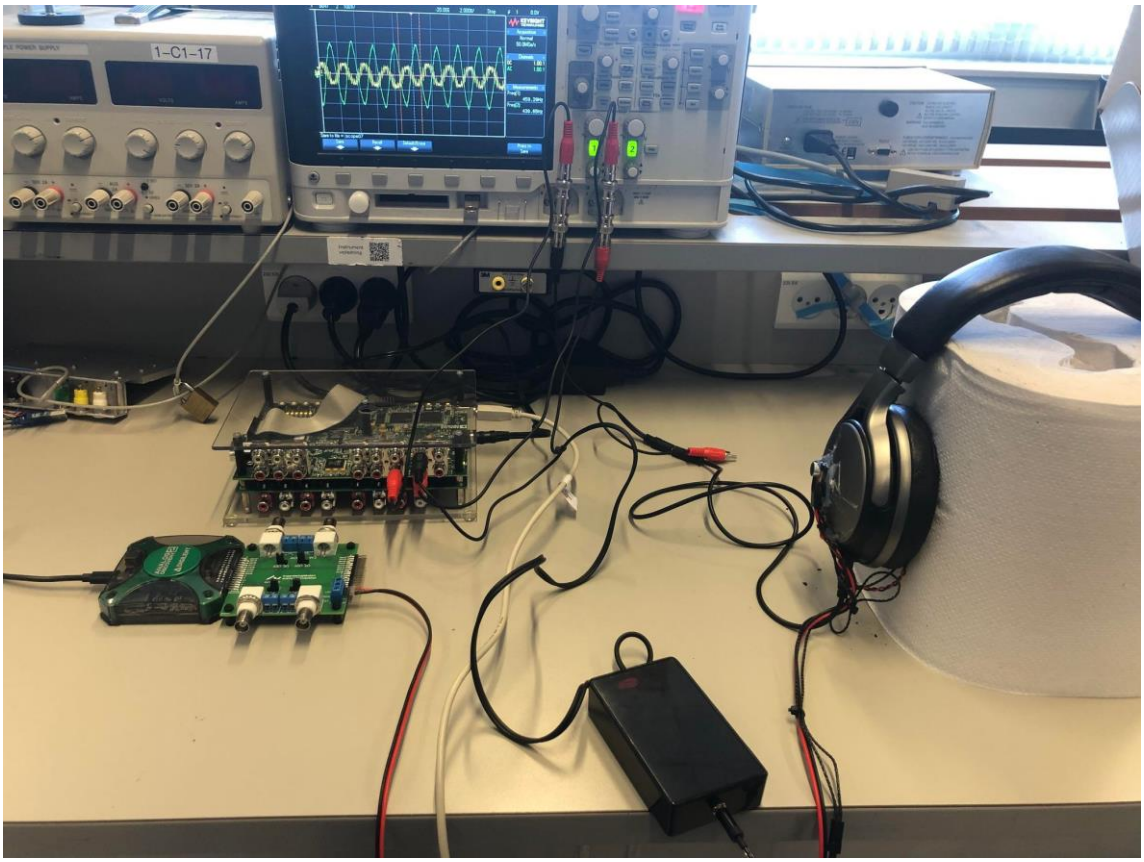
6.3.1 Prototype

På Figur 36 ses prototypen til projektet. Til error og reference-mikrofon er der valgt BOB-12758-boards [9]. Disse mikrofon-boards, har en forforstærker som forstærker mikrofonsignalet til et passende spændingsniveau. De to mikrofon-boards, bliver forsynet af en ekstern 5V strømforsyning. Error og reference-mikrofonerne er placeret på henholdsvis inde og ydersiden af højtalerkoppen. Outputtet fra reference og error-mikrofonen bliver sendt igennem et båndpas-filter, for at fjerne DC-komponenten samt nogle af de højere frekvenser af signalet, inden det bliver samplet af DSP'en. Lydsignalet til højtaleren bliver leveret igennem et AUX-kabel. På Figur 37 ses placeringen af de to mikrofoner.

Figur 36 - De samlede elementer som opgør prototypen for projektet.


Figur 37 - På venstre billede ses placeringen af reference-mikrofonen. På højre billede ses placeringen af error-mikrofonen.

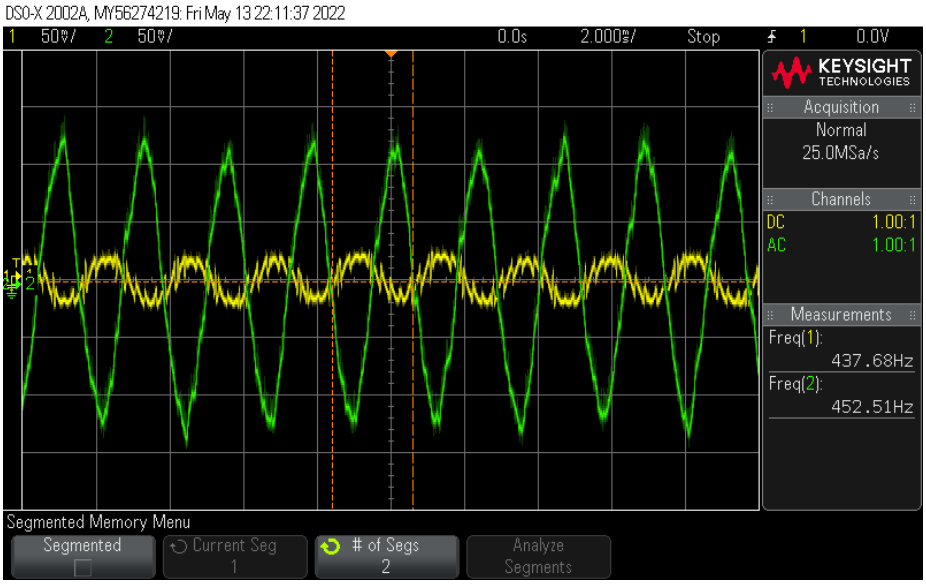
Til verifikation og test af systemet, er prototypen som ses på Figur 36 og Figur 37 tilsluttet blackfin og et oscilloskop. Oscilloskop vil måle referencesignalet fra reference-mikrofonen samt antistøjs-signalet fra Blackfin. På Figur 38 ses forsøgsopstillingen.



Figur 38 - Forsøgsopstilling

På Figur 39 ses et resultat for testen, med et testsignal på 450 Hz. Testsignalet er generet med en online sinus-generator, afspillet fra en telefon. På channel 1(gule signal) af oscilloskopet er outputtet fra Blackfin, som bliver afspillet i headset. Channel 2 (grønne signal) måler signalet på

reference-mikrofonen. I dette tilfælde er LMS-filtertaps 200 og filteret har en $\mu = 0.001$. Se eventuelt *Appendix: Test af protype*, for flere testscenarie.



Figur 39 - ~450 Hz inputsignal

6.4 Test af krav

I dette afsnit dannes der et overblik over hvorvidt kravene i kravspecifikationen er opfyldt, og hvor i rapporten de adskillige tests heraf er udført. Tabel 3 fortæller hvilket krav der testes, hvor i rapporten testen kan findes og hvorvidt det menes at kravet er opfyldt.

Nr.	Krav	Test	Resultat
R1	Systemet skal sikre minimum 20dB dæmpning af baggrundsstøj målt fra brugerens øre.	Der måles ikke fra brugerens øre, men i stedet benyttes AlgoTester til at teste implementeringen i MATLAB, se afsnit 6.1 og 6.2	OK
R2	Systemet skal kunne dæmpe baggrundsstøj i frekvensområdet 100 Hz - 2000 Hz.	Der tages et signal med støj fra et passagerfly i dette spektrum, se afsnit 6.1.2. Der testes også med et chirp-signal, se afsnit 6.2	OK
R3	Systemet skal have en latenstid fra input til output på maksimalt 0.146 ms pr. sample.	Antallet af benyttede antal clockcycles er målt, se afsnit 5.4. Det tager 0.061ms at processerer en bloksize af 4 samples.	OK
R4	Systemet skal have et dynamikområde på 96 dB (16-bit opløsning).	Der benyttes ADC'er og DAC'er med 16 bits opløsning, se afsnit 5.1	OK
R5	Systemet må maksimalt bruge 90% af BF533's ydeevne	I afsnit 2.3 udregnes det hvor mange clockcycles der må benyttes for at overholde dette krav, og det testes og realiseres i afsnit 5.4	OK
R6	Systemet skal kunne adapteres til batteridrevne applikationer og må maksimalt trække 70 mW (20mA, 3.3V) i filtermode.	Grundet tidsbegrænsning på projektet er dette krav hverken implementeret eller testet	IKKE OK
R7	Systemets softwareapplikation skal maksimalt fylde 80% instruktionshukommelse.	Dette er testet i 5.5.2 Code memory	OK
R8	Systemets filtrer må maksimalt anvende 80% af ram-hukommelse.	Dette er testet i 5.5.1 Datamemory	OK
R9	Systemets algoritme skal realiseres med fixed point aritmetik.	Dette er realiseret jf. afsnit 5.2	OK
DR1	Systemet skal sample med 48 kHz.	Der benyttes en samplingsfrekvens på 48kHz jf. afsnit 5.1	OK
DR2	Systemets algoritme må maksimalt bruge 45000 cycles pr. blocksize, hvor en blocksize er 4 samples.	Dette er testet i afsnit 5.4, hvor den endelige implementering ender med at benytte 32985 clockcycles	OK

Tabel 3 - Test af kravspecifikation

7 Diskussion

I projektet er et ANC-system til brug i høretelefoner tilnærmelsesvist blevet implementeret. I teorien forklares det hvorledes der kan tages hensyn til akustikken i høretelefonerne ved estimeringer af overføringskarakteristikkerne for antistøjs-signalets rejse mellem højtaler og de to mikrofoner. Disse estimater/målinger af overføringskarakteristikkerne er dog ikke blevet udført, da det ville kræve en mere stabil prototype og testmiljø, hvilket hurtigt blev klart var uden for fagets område.

Det var dog muligt at implementere LMS-algoritmen på DSP-platformen og teste denne ved brug af AlgoTester-klassen, hvor et kendt støjsignal kan indlæses til processeren via USB. Desuden blev antistøjs-signalet (outputtet) og error-signalet også udlæst fra BF533 via USB efter det indsendte støjsignal var processeret. Der blev testet de samme signaler i MATLAB og på BlackFin platformen. Dermed verificeres implementeringen ved sammenligning med MATLAB-løsningen. Dette er derfor udelukkende en test af algoritmens funktionalitet som den er implementeret til en DSP-platform, og ikke en reel test af et fuldendt ANC-system. Et komplet ANC-system designet til høretelefoner indebærer netop flere problematikker og ingeniørfaglige aspekter end udelukkende en implementeret algoritme. Selve kompleksiteten i et sådant fuldendt system strækker sig derfor langt ud over kursets læringsmål, hvorfor en afgrænsning blev nødvendig.

Selve implementeringen overholder definerede krav for memory, cycleforbrug, cpu load mv. Dog kan man godt argumentere for at visse aspekter kan optimeres yderligere. Eksempelvis anvendes der en samplefrekvens på 48 kHz, hvilket er langt over, hvad der reelt set er behov for, eftersom det kun er målet at udligne signaler op til 2 kHz. Vha. downsampling kunne man derved sparre en masse processeringskraft.

I afsnit 6.1, hvor implementeringen testes ved brug af test-koden i AlgoTester, ses det at det for et realistisk signal, i form af støjen i en kabine på et passagerfly, ikke gør nogen betydeligt hørbar forskel om der benyttes 50 eller 200 taps i det implementerede LMS-filter. Dog overholder implementeringen med 200 filter-taps kravene til maximum brug af clock-cykler jf. Tabel 1, hvorfor det stadig menes at denne er at foretrække. Ville man benytte en større block-size med mere end 4 samples, kunne man i så fald godt nøjes med 50 filter-taps, uden at forværre dæmpningskvaliteten bemærkelsesværdigt. De 200 filter-taps er til gengæld valgt, da disse viste en hurtigere tilpasning af antistøjs-signalet for de simple sinus-toner. Dermed kunne man også benytte flere filter-taps, hvis dette blev fundet nødvendigt, dog ville det kræve en mindre block-size.

8 Konklusion

Det er lykkedes at designe og implementere en LMS-algoritme til aktiv støjudligning på en DSP-platform, som er blevet testet og verificeret ved sammenligning med en MATLAB-løsning. I MATLAB er der fundet frem til at et LMS-filter med enten 50, 100 eller 200 taps og en dertilhørende stepstørrelse på 0.001 er tilstrækkeligt til at støjudligne de test-signaler der benyttes. Det samme gør sig gældende for den implementerede algoritme på DSP-platformen

LMS-algoritmen er blevet implementeret på et BF533 EZ-kit, med fixed point aritmetik. Den udviklede algoritme overholder de krav som er sat til latenstid og memory-forbrug. Derudover er det muligt at støjudligne ved brug af destruktiv interferens i frekvensområdet 100 Hz - 2000 Hz, med en minimumdæmpning på -25 dB. Ved brug af testkode er det desuden verificeret at LMS-algoritmen er i stand til at dæmpe et realistisk støjsignal fra en flykabine til en tilfredsstillende grad.

Det kan også konkluderes at den udviklede prototype ikke var god nok. Det blev meget synligt undervejs i projektet, at der var andre problemstillinger som spillede ind. Herunder akustikproblemer i forhold til udformningen af høretelefoner med placeringen af mikrofoner og højtaler, samt et dårligt testmiljø.

Referencer

[1] A. Devices, »ADSP-BF533,« [Online]. Available: https://www.analog.com/en/products/adsp-bf533.

[2] W.-S. G. a. S. M. Kuo, »Embedded Signal Processing with Micro Signal Architecture,« Wiley.

[3] B. Farhang-Boroujeny, Adaptive Filters. Theory and Applications, Second Edition, WILEY.

[4] C. Y. C. a. S. M. K. K. C. Chen, »Active noise control in a duct to cancel broadband noise«.

[5] Mathworks, »Active Noise Control Using a Filtered-X LMS FIR Adaptive Filter,« MATLAB & Simulink - MathWorks Nordic., [Online]. Available: https://se.mathworks.com/help/audio/ug/active-noise-control-us-ing-a-filtered-x-lms-fir-adaptive-filter.html.

[6] Mathworks, »https://se.mathworks.com,« [Online]. Available: https://se.mathworks.com/help/matlab/ref/randn.html.

[7] A. A. S. Effect, »orangefreesounds.com,« [Online]. Available: https://orangefreesounds.com/airplane-ambience-sound-effect.

[8] R. G. Lyons, Understanding Digital Signal Processing - 3rd edition, Prentice HALL.

[9] Sparkfun, »SparkFun Electret Microphone Breakout - BOB-12758 - SparkFun Electronics.",« [Online]. Available: https://www.sparkfun.com/products/12758.

Appendix

Appendixet i rapporten indeholder mindre relevante figurer, kode mm., som fyldte for meget i selve hovedrapporten uden større bidrag til yderligere indsigt i projektet. Dermed kan dette afsnit ikke læses uafhængigt af hovedrapporten, men skal ses som et opslagsværk for læseren. I hovedrapporten vil der løbende blive henvist til figurer, tabeller mm. i dette appendix.

Appendix: MATLAB løsning

```
clc; clear; close all;
%% Konstanter til plotvinduesstørrelser
x0=10;
y0=10;
width=1100;
height=600;

%% Path-estimering: White noise generator
fs = 48000;                 % Samplingsfrekvens
N = 1024;                   % Antal samples
N_s = 20000;                % Antal samples til path-estimering
n = 1:N;                    % Sample
n_s = 1:N_s;                 % Sample til path
f_res = fs/N;               % Frekvensopløsning
f_res_s = fs/N_s;           % Frekvensopløsning til path
f_axis = [0:N-1]*f_res;     % Frekvensakse
f_axis_s = [0:N_s-1]*f_res_s; % Frekvensakse

%%
% Generer hvid støj mellem 150Hz og 2000Hz
x = randn(1,N_s);
x = bandpass(x,[150,2000],fs);

% Fast-fourier-transformation af støjen
X = fft(x, N_s);

% Plot af signalet og dets frekvensspektrum
figure()
subplot(2,1,1)
plot(n_s,x)
grid on;
xlabel('n','FontSize', 15);
ylabel('x(n)','FontSize', 15);
title('Signal','FontSize', 16);
subplot(2,1,2)
plot(f_axis_s(1:end/2), abs(X(1:end/2)))
grid on;
xlabel('Frekvens [Hz]','FontSize', 15)
ylabel('Magnitude','FontSize', 15)
title('Frekvensspektrum','FontSize', 16)
xlim([0 2500])
set(gcf,'position',[x0,y0,width,height])
saveas(gcf,'White_nose_generator.png');

%% Path-estimering: LMS
M = 20;        % et M-tap filter
mu = 0.002;     % stepsize
help LMS
[y, e, w] = LMS(x, x, N_s, M, mu);
```

```
%% Path-estimering: Plot af LMS
figure()
plot(n_s,x)
hold on;
plot(n_s,y)
plot(n_s,e)
legend('x(t)', 'y(t)', 'e(t)')
grid on;
xlabel('n','FontSize', 15);
ylabel('Amplitude','FontSize', 15);
title('Path-estimering: LMS','FontSize', 16);
set(gcf,'position',[x0,y0,width,height])
saveas(gcf,'path_est_LMS.png');

%% Path-estimering: Plot af estimeret filter
figure()
stem(w)
grid on;
xlabel('m','FontSize', 15);
ylabel('w(m)','FontSize', 15);
title('Path-estimering: Filter-weights','FontSize', 16);
set(gcf,'position',[x0,y0,width,height])
saveas(gcf,'path_est_weights.png');

%% Path-estimering: Plot af frekvensresponset for estimeret filter
figure()
freqz(w,1,20000,'half',fs)
title('Path-estimering: Frekvensrespons', 'FontSize', 16)
grid on;
xlim([0 2000])
set(gcf,'position',[x0,y0,width,height])
saveas(gcf,'path_est_freqz.png');

%% ANC: Støjsignalet

t = [0:N-1]/fs;
x_noise = 0.5*(sin(2*pi*500*t) + sin(2*pi*200*t));
X_noise = fft(x_noise, N);


figure()
subplot(2,1,1)
plot(n,x_noise)
xlim([0 N])
grid on;
xlabel('n','FontSize', 15);
ylabel('x_{noise}','FontSize', 15);
title('Noisy signal','FontSize', 16);
subplot(2,1,2)
plot(f_axis(1:end/2), abs(X_noise(1:end/2)))
grid on;
xlabel('Frekvens [Hz]','FontSize', 15)
ylabel('Magnitude','FontSize', 15)
title('Amplitude-spektrum','FontSize', 16)
xlim([0 N])
set(gcf,'position',[x0,y0,width,height])
saveas(gcf,'noise_signal.png');
```

```
%% ANC: LMS 50 taps

M = 50;         % et M-tap filter
mu = 0.001;     % stepsize
[y, e, w] = LMS(x_noise, -x_noise, N, M, mu);

%% ANC: Plot af LMS 50 taps
figure()
plot(n,x_noise, 'b')
hold on;
plot(n,y, 'r')
plot(n,e,'y', 'LineWidth',2)
legend('x(n)', 'y(n)', 'e(n)')
grid on;
xlabel('n','FontSize', 15);
ylabel('Amplitude','FontSize', 15);
title('ANC: LMS 50 taps','FontSize', 16);
xlim([0 N])
set(gcf,'position',[x0,y0,width,height])
saveas(gcf,'ANC_LMS_50_taps.png');

%% Plot af filter-weights 50 taps
figure()
stem(w)
grid on;
xlabel('m','FontSize', 15);
ylabel('w(m)','FontSize', 15);
title('ANC: Filter-weights 50 taps','FontSize', 16);
set(gcf,'position',[x0,y0,width,height])
saveas(gcf,'anc_weights_50_taps.png');


%% ANC: LMS 100 taps

M = 100;         % et M-tap filter
mu = 0.001;     % stepsize
[y, e, w] = LMS(x_noise, -x_noise, N, M, mu);

e_rms = rms(e(500:end))

%% ANC: Plot af LMS 100 taps
figure()
plot(n,x_noise, 'b')
hold on;
plot(n,y, 'r')
plot(n,e,'y', 'LineWidth',2)
legend('x(n)', 'y(n)', 'e(n)')
grid on;
xlabel('n','FontSize', 15);
ylabel('Amplitude','FontSize', 15);
title('ANC: LMS 100 taps','FontSize', 16);
xlim([0 N])
set(gcf,'position',[x0,y0,width,height])
saveas(gcf,'ANC_LMS_100_taps.png');

%% Plot af filter-weights 100 taps
figure()
stem(w)
```

```
grid on;
xlabel('m','FontSize', 15);
ylabel('w(m)','FontSize', 15);
title('ANC: Filter-weights 100 taps','FontSize', 16);
set(gcf,'position',[x0,y0,width,height])
saveas(gcf,'anc_weights_100_taps.png');

%% ANC: LMS 200 taps

M = 200;         % et M-tap filter
mu = 0.001;     % stepsize
[y, e, w] = LMS(x_noise, -x_noise, N, M, mu);

e_rms = rms(e(500:end))

%% ANC: Plot af LMS 200 taps
figure()
plot(n,x_noise, 'b')
hold on;
plot(n,y, 'r')
plot(n,e,'y', 'LineWidth',2)
legend('x(n)', 'y(n)', 'e(n)')
grid on;
xlabel('n','FontSize', 15);
ylabel('Amplitude','FontSize', 15);
title('ANC: LMS 200 taps','FontSize', 16);
xlim([0 N])
set(gcf,'position',[x0,y0,width,height])
saveas(gcf,'ANC_LMS_200_taps.png');

%% Plot af filter-weights 200 taps
figure()
stem(w)
grid on;
xlabel('m','FontSize', 15);
ylabel('w(m)','FontSize', 15);
title('ANC: Filter-weights 200 taps','FontSize', 16);
set(gcf,'position',[x0,y0,width,height])
saveas(gcf,'anc_weights_200_taps.png');

%% Airplane noise
[y,Fs] = audioread('Airplane.mp3');
audiowrite('Airplane_48k.wav', y, fs);
[x_plane,fs_plane] = audioread('Airplane_48k.wav');

x_plane = x_plane(1:end);

N_plane = length(x_plane);

%% ANC airplane frekvensrespons
f_res_plane = fs_plane/N_plane;           % Frekvensopløsning til path
f_axis_plane = [0:N_plane-1]*f_res_plane;     % Frekvensakse

% Fast-fourier-transformation af støjen
X = fft(x_plane, N_plane);

% Plot af signalet og dets frekvensspektrum
n = 1:N_plane;
```

```
figure()
subplot(2,1,1)
plot(n,x_plane)
grid on;
xlabel('n','FontSize', 15);
ylabel('x(n)','FontSize', 15);
title('Signal','FontSize', 16);
subplot(2,1,2)
plot(f_axis_plane(1:end/2), abs(X(1:end/2)))
grid on;
xlabel('Frekvens [Hz]','FontSize', 15)
ylabel('Magnitude','FontSize', 15)
title('Frekvensspektrum','FontSize', 16)
xlim([0 2500])
set(gcf,'position',[x0,y0,width,height])
saveas(gcf,'cafe_noise_signal.png');

%% ANC Airplane: LMS 200 taps

M = 200;        % et M-tap filter
mu = 0.001;     % stepsize
[y, e, w] = LMS(x_plane, -x_plane, N_plane, M, mu);

%% ANC Airplane: Plot af LMS 200 taps
n = 1:N_plane;
figure()
plot(n,x_plane, 'b', 'LineWidth',1)
hold on;
plot(n,y, 'r', 'LineWidth',1)
plot(n,e,'y', 'LineWidth',2)
legend('x(n)', 'y(n)', 'e(n)')
grid on;
xlabel('n','FontSize', 15);
ylabel('Amplitude','FontSize', 15);
title('ANC of airplane noise: LMS 200 taps','FontSize', 16);
xlim([0 N_plane])
set(gcf,'position',[x0,y0,width,height])
saveas(gcf,'ANC_airplane_LMS_200_taps.png');

%%
sound(x_plane(1:end/5),fs)
audiowrite('airplane_no_ANC.wav',x_plane,fs)
%%
sound(e(1:end/5),fs)
audiowrite('airplane_ANC.wav',e,fs)

%% ANC Airplane, ZOOM: Plot af LMS 200 taps
n = 1:N_plane;
figure()
plot(n,x_plane, 'b', 'LineWidth',1)
hold on;
plot(n,y, 'r', 'LineWidth',1)
plot(n,e,'y', 'LineWidth',2)
legend('x(n)', 'y(n)', 'e(n)')
grid on;
xlabel('n','FontSize', 15);
ylabel('Amplitude','FontSize', 15);
title('ANC of airplane noise: LMS 200 taps, Zoom','FontSize', 16);
```

```
xlim([10000 11000])
set(gcf,'position',[x0,y0,width,height])
saveas(gcf,'ANC_zoom_airplane_LMS_200_taps.png');

%% ANC Airplane: LMS 50 taps

M = 50;        % et M-tap filter
mu = 0.001;     % stepsize
[y, e, w] = LMS(x_plane, -x_plane, N_plane, M, mu);

%% ANC Airplane: Plot af LMS 50 taps
n = 1:N_plane;
figure()
plot(n,x_plane, 'b', 'LineWidth',1)
hold on;
plot(n,y, 'r', 'LineWidth',1)
plot(n,e,'y', 'LineWidth',2)
legend('x(n)', 'y(n)', 'e(n)')
grid on;
xlabel('n','FontSize', 15);
ylabel('Amplitude','FontSize', 15);
title('ANC of airplane noise: LMS 50 taps','FontSize', 16);
xlim([0 N_plane])
set(gcf,'position',[x0,y0,width,height])
saveas(gcf,'ANC_airplane_LMS_50_taps.png');

%%
sound(x_plane(1:end/5),fs)
%%
sound(e(1:end/5),fs)

%% ANC airplane, ZOOM: Plot af LMS 50 taps
n = 1:N_plane;
figure()
plot(n,x_plane, 'b', 'LineWidth',1)
hold on;
plot(n,y, 'r', 'LineWidth',1)
plot(n,e,'y', 'LineWidth',2)
legend('x(n)', 'y(n)', 'e(n)')
grid on;
xlabel('n','FontSize', 15);
ylabel('Amplitude','FontSize', 15);
title('ANC of airplane noise: LMS 50 taps, Zoom','FontSize', 16);
xlim([10000 11000])
set(gcf,'position',[x0,y0,width,height])
saveas(gcf,'ANC_zoom_airplane_LMS_50_taps.png');
```

Listing 7 - Det fulde MATLAB-script til MATLAB-løsningen

Appendix: Test af implementering

```
void main(void)
{
	sysreg_write(reg_SYSCFG, 0x32); //Enable 64-bit,free-running cycle-coun-
ter

#if 1
```

```
	uint16_t numberOfFiles = 15;
	char x_signal_filename[35];
	char y_signal_filename[35];
	char e_signal_filename[35];

	InitAlgoProcess(); // Initialization of processing algorithms
	AlgoTester algoTest(&FilterANC);

	for(uint16_t i = 0; i < numberOfFiles; i++)
	{
		sprintf(x_signal_filename, "..\\src\\MATLAB\\x_signal_%d.txt", i);
		sprintf(y_signal_filename, "..\\src\\MATLAB\\y_signal_%d.txt", i);
		sprintf(e_signal_filename, "..\\src\\MATLAB\\e_signal_%d.txt", i);
		algoTest.runTest(x_signal_filename, y_signal_filename,
"..\\src\\MATLAB\\w_signal.txt", e_signal_filename);
	}

#else
	InitSystemHardware(); // Initialization of BF533
	InitAlgoProcess(); // Initialization of processing algorithms
	printf("Application running: SW4 turns filter on, SW5 turns filter
off\r\n");
	Init_Interrupt(); // Enable interrupts
	Enable_DMA();
#endif

	while(1)
	{

	}; // wait forever
}
```

Listing 8 - Main-funktion i Applikation.

```
short AlgoTester::runTest(char *inFileName, char *outFileName, char
*weightsFileName, char *errorsFileName)
{
	// Reading test signal in inFileName
	short error = readSignal(TestSignal, inFileName);


	if (error == 0)
	{
		// Test file loaded into TestSignal
		printf("Testing algorithm - input file %s with signal of %d sam-
ples\n", inFileName, N);

		// Processing algorithm on TestSignal
		m_pAlgo->process(TestSignal, errorSignal, OutputSignal, N);

		// Writing result from OutputSignal to outFileName
		error = writeSignal(OutputSignal, outFileName);

		error = writeSignal(TestSignal, inFileName);

		//short* weights = m_pAlgo->getWeights();

		//error = writeWSignal(weights, weightsFileName);
```

```
            //short* errorSignal = m_pAlgo->getErrors();

            error = writeSignal(errorSignal, errorsFileName);

            if (error == 0)
                  printf("Result signal of %d samples saved in %s \n", N, out-
FileName);
            else
                  printf("Error writing result to file %s \n", outFileName);
      }
      else
            printf("Error reading file %s with input test signal\n", inFile-
Name);

      return error;
}
```

Listing 9 - runTest fra klassen AlgoTester

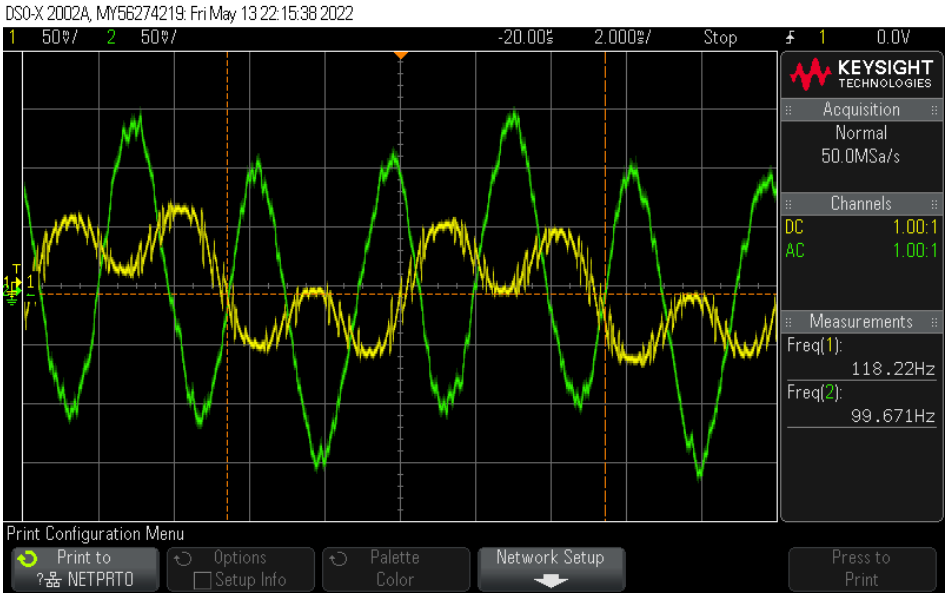
```
%% Konstanter til signaler
fs = 48000; % Samplings frequency
time = 0.021333*15; % Sec, equal to 1024 samples
t = 0:1/fs:time;
f1 = 500; % Hz
f2 = 2000; % Hz

%% Chirp signal
x = chirp(t,f1,time,f2);

%% Sinus-signal
x = 0.5*(sin(2*pi*500*t)+sin(2*pi*200*t));

%% Airplane-signal
fs = 48000;
[y,Fs] = audioread('Airplane.mp3');
audiowrite('Airplane_48k.wav', y, fs);
[x,fs] = audioread('Airplane_48k.wav');

%% Opdel x-signalet i numberOfFiles .txt filer
numberOfFiles = 15;
x16 = x*2^15;
filelength = 1024;

for f = 0:numberOfFiles-1
    filename = append('x_signal_', int2str(f), '.txt');
    fid = fopen(filename, 'w');
    for i=1:filelength
        if x16(f*filelength + i) > 0
            x16(f*filelength + i) = x16(f*filelength + i)-1;
        end
        xtext = num2str(round(x16(f*filelength + i)));
        fprintf(fid, '%s,\r\n', xtext);
    end
    fclose(fid);
end

%% Konstanter til plotvinduesstørrelser
x0=10;
y0=10;
```

```
width=1100;
height=600;

%% Plot af ANC

N = 1024*numberOfFiles;
n = 1:N;
figure()
plot(n, x(1:N), 'b')
hold on;
plot(n,y_signal(1:N), 'r')
plot(n,e_signal(1:N),'y', 'LineWidth',2)
legend('x(n)', 'y(n)', 'e(n)')
grid on;
xlabel('n','FontSize', 15);
ylabel('Amplitude','FontSize', 15);
title('ANC on BF: Sinus 200 taps','FontSize', 16);
xlim([0 N])
set(gcf,'position',[x0,y0,width,height])
saveas(gcf,'ANC_sinus_long_BF_200_taps.png');

%% Afspil x-signalet
sound(x(1:N),fs)

%% Skriv error-signalet til en .wav fil
audiowrite('ANC_BF_Airplane_error.wav', e_signal, fs);

%% Afspil error-signalet
sound(e_signal,fs)

%% Udregn SNR
SNR_200 = (1/(N/2)*sum(e_signal(end/2:end).^2))/(1/(N/2)*sum(x(end/2:end).^2))
SNR_200_db = 10*log10(SNR_200)

%% Load error-signalet
for f = 0:numberOfFiles-1
    filename = append('e_signal_', int2str(f), '.txt');
    load(filename);
end

%% Load output-signalet
for f = 0:numberOfFiles-1
    filename = append('y_signal_', int2str(f), '.txt');
    load(filename);
end

%% Load input-signalet
for f = 0:numberOfFiles-1
    filename = append('x_signal_', int2str(f), '.txt');
    load(filename);
end
%% Concatinate 100 input-filer
x_signal = [...
x_signal_0' ./ (2^15) ...
x_signal_1' ./ (2^15) ...
x_signal_2' ./ (2^15) ...
x_signal_3' ./ (2^15) ...
x_signal_4' ./ (2^15) ...
```

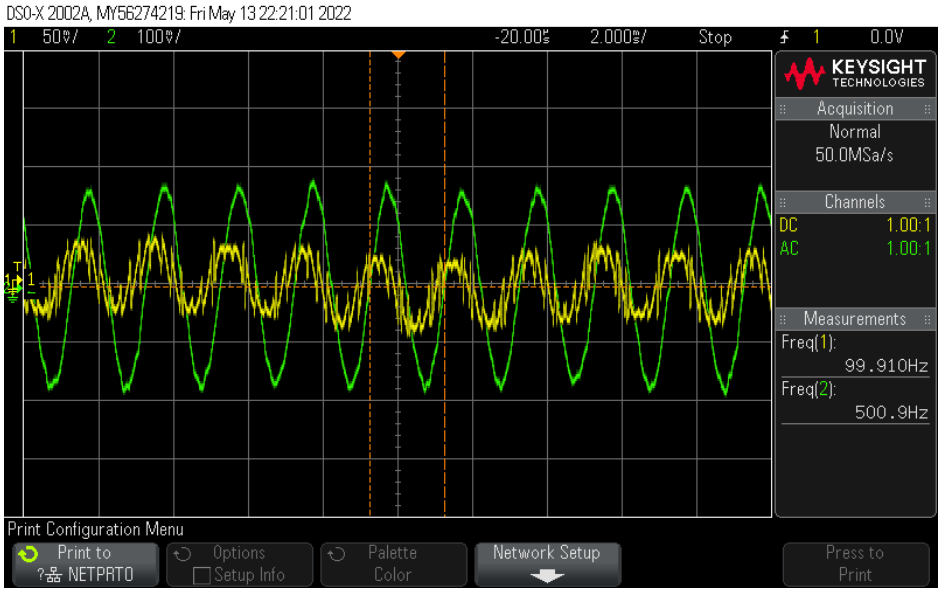
```
x_signal_5' ./ (2^15) ...
```

Listing 10 - MATLAB-koden til test af implementering. Kan generere forskellige støjsignaler og dele disse op i txt-filer, klar til indlæsning på BF533. Kan indlæse filer outputtet fra BF533 og plotte disse signaler.

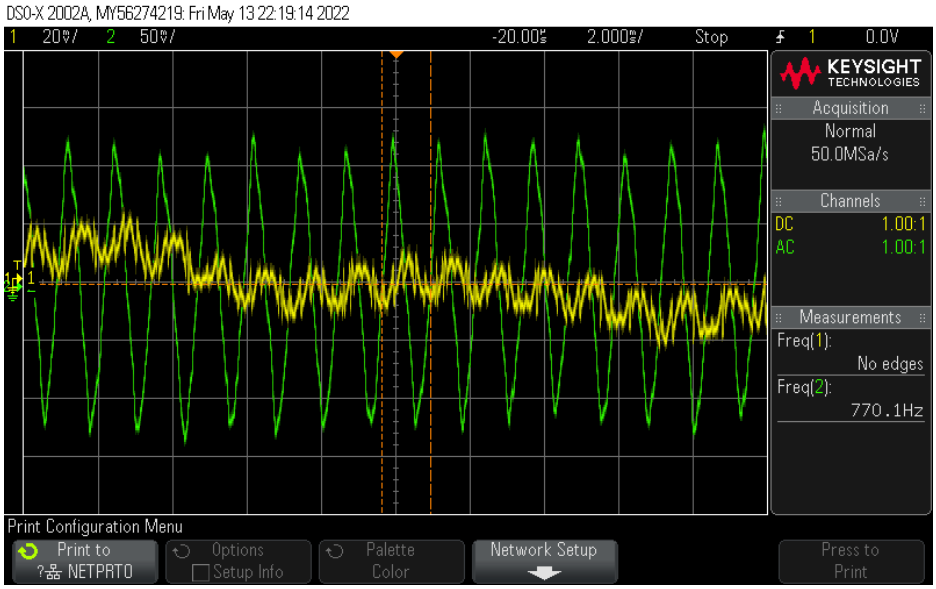
Appendix: Test af prototype



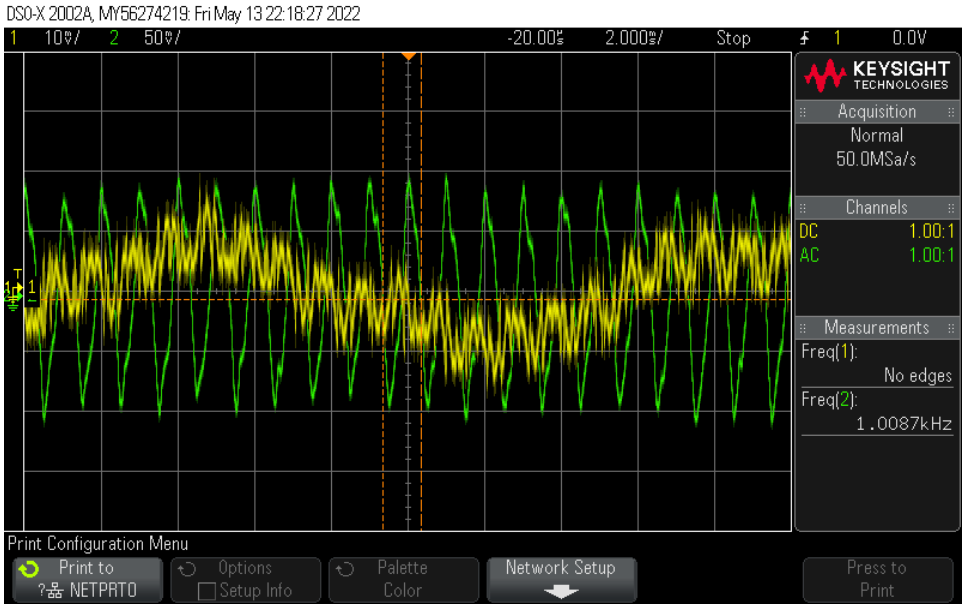
Figur 40 - 100 Hz inputsignal



Figur 41 - 500 Hz inputsignal



Figur 42 - 770 Hz inputsignal



Figur 43 - 1 kHz inputsignal