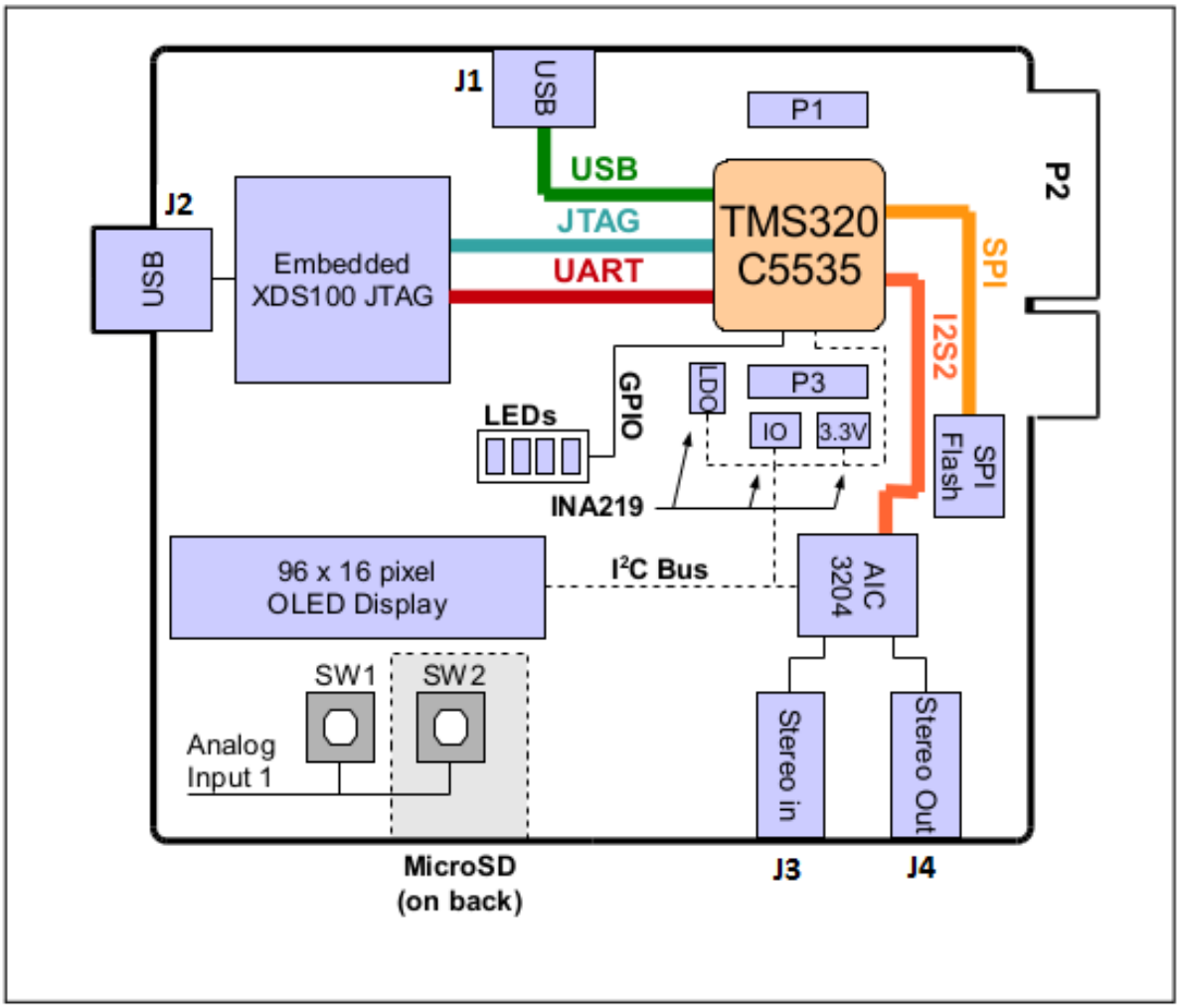
E4DSA Case 2 - IIR notch-filter og real-time-implementering på TMS320C5535

Janus Bo Andersen [1]

20. marts 2020



[1] ja67494@post.au.dk

Indhold

1. Indledning

Anden case i E4DSA er design og real-time-implementering af et IIR notch-filter på DSP-hardware. Filteret er designet og testet i MATLAB. MATLAB er også brugt til at kvantisere koefficienter og teste fixed-point filter-algoritmen. Filteret er implementeret på DSP-hardware vha. Code Composer Studio (CCS). Target er TMS320C5535 (eZDSP kit). MATLAB og C-kode er gengivet med forskellige baggrundsfarver for lette genkendelighed i denne journal.

Beklager, at journalerne bliver lange; men det er også for at kunne have detaljerne til fremtiden.

2. Opgave 1: Analyse af inputsignal

Inputsignalet er "Bright Side of the Road" af Van Morrison fra albummet "Into the Music" (1979). Signalet indeholder de første 60 s af nummeret. Signalet er i mono, 32-bit floating point, og samplingsfrekvensen er 48 kHz (professionel standard). Nummeret er blevet saboteret med en ren sinustone, der er ekstremt ubehagelig at lytte til i mere end et par sekunder. Så den skal fjernes. Tonens frekvens er fundet vha. spektralanalysen nedenfor.

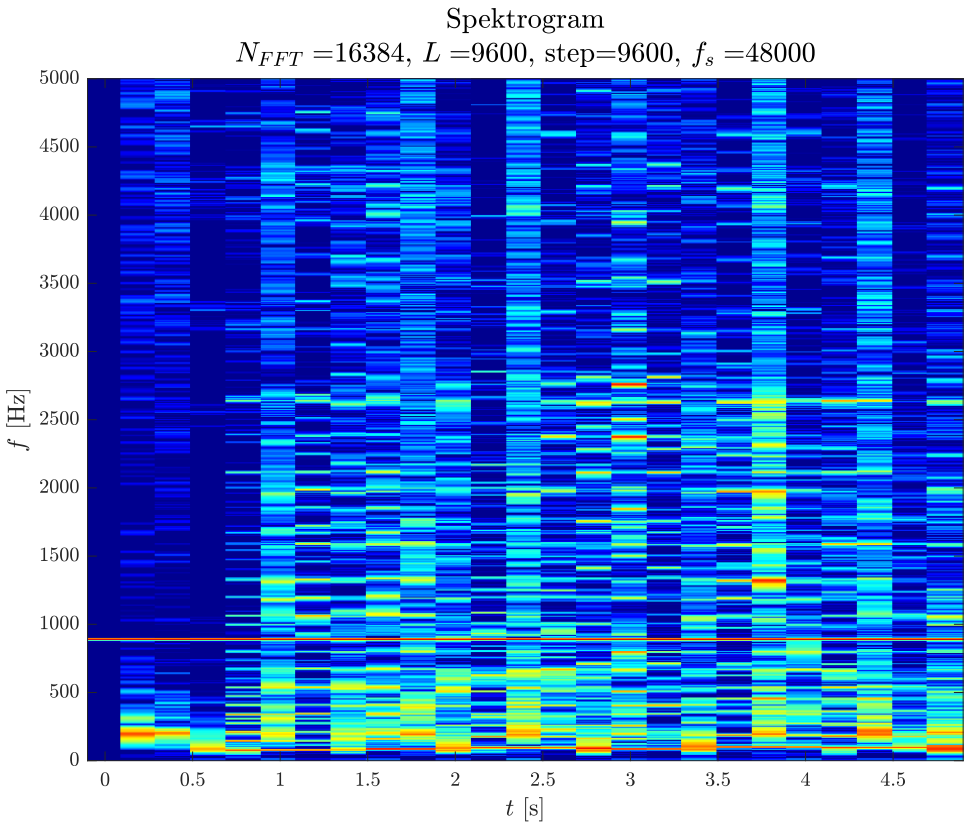
```
clc; clear all; close all;
```

```
[x, fs] = audioread('musik_tone_48k.ogg');
% soundsc(x, fs);
% clear sound;
```

Spektrogrammet beregnes med rullende FFT'er. De første 5 s vises her, men sinustonen er stationær i hele signalets længde. Hver FFT er zero-padded[1]. Vindueslængden på $L = \frac{f_s}{5} = 9600$ samples giver et ok trade-off mellem frekvensopløsning og tidsopløsning. $\Delta f = \frac{f_s}{L} = 5.0$ Hz og tidsopløsning $T_{STFT} = \frac{L}{f_s} = 0.20$ sek. Steppet er sat, så der ikke er overlap på vinduer. Der laves 25 FFT'er[2].

```
tl = 5;                                 % tidslængde til analyse, sekunder
x_ = x(1:tl*fs)';                       % udvalgt sektion af signal
L = fs/5;                               % vindueslængde 9600 => 5 Hz opløsning
stepSize = L;                           % step er 1 vindue (0% overlap)
Nfft = 2^nextpow2(L);                   % 2^14
figure();
spectrogram0(x_, L, Nfft, stepSize, fs, [0 5000]); % Vis kun op til 5 kHz
```

[1] Det giver pænere spektrum, "sampler" DTFT'en på flere punkter
[2] Der analyseres 5 s signal med 5 STFT/s, så i alt 25 FFT'er beregnes

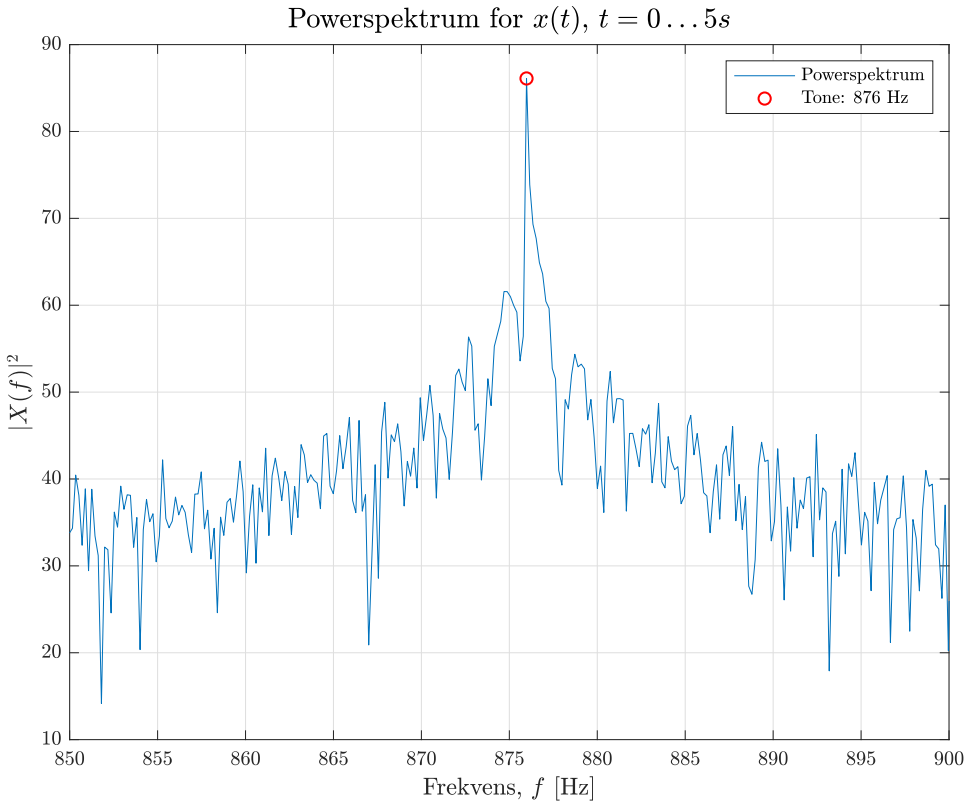


Spektrogrammet viser ved den røde vandrette streg en stationær ren tone omkring 875 Hz.

Pga. lav grafikopløsning i figuren ser det også ud som om, at der er en ren tone ved en lavere frekvens (100 Hz?). Hvis der zoomes ind, ses det dog, at denne hverken er stationær eller er til stede konstant.

En enkelt FFT med 240000 samples benyttes for at pinpointe den eksakte frekvens for den forstyrrende tone. Det giver en frekvensopløsning på $\Delta f = 0.2$ Hz. Der zero-paddes igen. Figuren nedenfor viser, at den forstyrrende tone er lokaliseret ved 876±0.2 Hz.

```
Nfft = 2^nextpow2(length(x_));      % 2^18 -> zero-padding med 22144 0'er
X_ = fft(x_, Nfft);
X_pow = X_ .* conj(X_);             % Powerspektrum
f_vec = (fs / Nfft) * (0:Nfft-1);   % Tilhørende frekvensakse
[val, idx] = max(10*log10(X_pow));  % Find frekvenssample med højeste power
fidx = f_vec(idx);                  % Tilhørende frekvensbin

figure();
plot(f_vec, 10*log10(X_pow)); hold on;
plot(fidx, val, 'ro'); hold off;
xlim([850 900]); grid on;
legend({'Powerspektrum',['Tone: ', num2str(round(fidx)), ' Hz']});
ylabel('$|X(f)|^2$', 'Interpreter','Latex', 'FontSize', 12);
xlabel('Frekvens, $f$ [Hz]', 'Interpreter','Latex', 'FontSize', 12);
title('Powerspektrum for $x(t)$, $t=0 \ldots 5s $', ...
    'Interpreter', 'Latex', 'FontSize', 14)
```



Der optræder ingen harmoniske af de 876 Hz (det kunne jo sagtens have været tilfældet). Det kan ikke ses her, men det er undersøgt. Vi slipper derfor for at implementere noget mere avanceret, som fx et comb-filter. Planen er nu at designe et notch-filter[3], der dæmper/filtrerer præcis frekvensen 876 Hz bort.

3. Opgave 2: Filterdesign

Filteret skal være 2. orden af typen IIR, og skal designes vha. placering af poler og nulpunkter. Det kan man gøre manuelt for en lav filterorden. Vi kender allerede centerfrekvensen, som kan omregnes til vinklen for de kompleks-konjugerede nulpunkter (og tilhørende poler).

3.1 Pol-nulpunktsplacering i et digitalt notch-filter

Et 2. ordens digitalt notch-filter har to "pol-nulpunktspar". Pol og nulpunkt i et par placeres på en radial, altså med samme vinkel, hvilken netop afgør filterets centerfrekvens. Et pol-nulpunktspar har et kompleks-konjugeret pol-nulpunktspar. Dvs. filteret alt-i-alt har to poler og to nulpunkter. Pol og nulpunkt skal ligge *tæt* på hinanden. Nulpunktet kan ligge på enhedscirklen, og modulus for polen skal være mindre end modulus for nulpunktet. Polerne skal ligge inden for enhedscirklen for at sikre stabilitet.

[3] Narrow-band båndstop

Ideen er, at ved evaluering af frekvensrespons $H(e^{j\omega})$ rundt på enhedscirklen, vil pol og nulpunkt i et par ophæve hinanden set for frekvenser relativt langt væk fra centerfrekvensen (fordi parret ligger tæt). For frekvenser tæt på centerfrekvensen vil nulpunktet blive dominerende, fordi det ligger relativt tættere på enhedscirklen end polen. Nulpunktet giver netop dæmpningen i centerfrekvensen.

Så længe pol og nulpunkt ikke ligger oveni hinanden, vil det være sådan, at jo tættere de er på hinanden, jo skarpere bliver notchet. MATLABs `filterDesigner` kan bruges til analysen.

Der er en grænse for hvor skarpt/stejlt et filter, der kan implementeres på fixed-point. Det er grundet den finitte præcision i kvantisering af koefficienter samt i beregninger. Modulus for polerne i filteret, der udvikles nedenfor, er fundet iterativt i et trade-off mellem at have et skarpt notch og at have et filter, som performer godt på target fixed-point platformen.

3.2 Ligninger for et digitalt notch-filter

Centerfrekvensen for notch-filteret er ω_c [rad/s] med $\omega_c = \pi \frac{f_c}{f_s/2}$. Modulus for nulpunkterne vælges til 1, og for polerne $r < 1$. Da pol-nulpunktsparrene er komplekst konjugerede, fås følgende nulpunkter og poler: $z_0 = e^{j\omega_c}$, $z_1 = e^{-j\omega_c}$, $p_0 = re^{j\omega_c}$ og $p_1 = re^{-j\omega_c}$. Filterets overføringsfunktion findes i den faktoriserede form [1, 6-38 s. 285]:

$$H(z) = \frac{(z - z_0)(z - z_1)}{(z - p_0)(z - p_1)} = \frac{(z - e^{j\omega_c})(z - e^{-j\omega_c})}{(z - re^{j\omega_c})(z - re^{-j\omega_c})} \tag{3.1}$$

Ganges paranteserne ud og benyttes Eulers relation $2\cos(\omega) = e^{j\omega} + e^{-j\omega}$, så har vi:

$$H(z) = \frac{z^2 - 2\cos(\omega_c)z + 1}{z^2 - 2r\cos(\omega_c)z + r^2} \tag{3.2}$$

Overføringsfunktionen er altså en ratio af to polynomier med reale koefficienter, og svarer til [1, ex. 6.14 s. 340]. For nemheds skyld indføres nu en gain-faktor G til at normalisere responset til et gain på 1 i pasbåndet, og der forkortes med z^{-2} for at se systemfunktionen [1, 6-25 s. 277]:

$$\frac{Y(z)}{X(z)} = G \cdot H(z) = G\frac{1 - 2\cos(\omega_c)z^{-1} + z^{-2}}{1 - 2r\cos(\omega_c)z^{-1} + r^2z^{-2}} \equiv \frac{b_0 + b_1z^{-1} + b_2z^{-2}}{a_0 - a_1z^{-1} - a_2z^{-2}} = \frac{B(z)}{A(z)} \tag{3.3}$$

Ved sammenligning ses, at $b_0 = b_2 = G$, $b_1 = -2G\cos(\omega_c)$, $a_0 = 1$ (altid!), $a_1 = 2r\cos(\omega_c)$ og $a_2 = -r^2$. Ovenstående omskrives til IIR-filterets output i z-domænet:

$$\begin{aligned} Y(z)(1 - 2r\cos(\omega_c)z^{-1} + r^2z^{-2}) &= X(z)G(1 - 2\cos(\omega_c)z^{-1} + z^{-2}) \\ \implies Y(z) &= GX(z) - 2G\cos(\omega_c)z^{-1}X(z) + Gz^{-2}X(z) + 2r\cos(\omega_c)z^{-1}Y(z) - r^2z^{-2}Y(z) \end{aligned} \tag{3.4}$$

Differensligningen findes via den inverse z-transformation heraf. Her benyttes delay-relationen: hvis $X(z) \longleftrightarrow x(n)$ så $z^{-k}X(z) \longleftrightarrow x(n-k)$. Dette giver differensligningen, og ved sammenligning med standardformen kan koefficienterne bekræftes [1, 6-21 s. 276]:

$$\begin{aligned} y(n) &= Gx(n) - 2G\cos(\omega_c)x(n-1) + Gx(n-2) + 2r\cos(\omega_c)y(n-1) - r^2y(n-2) \\ &\equiv b_0x(n) + b_1x(n-1) + b_2x(n-2) + a_1y(n-1) + a_2y(n-2) \end{aligned} \tag{3.5}$$

3.3 Design af notch-filter

For $f_c = 876$ Hz er $\omega_c = 0.1147$ rad/s. Værdien for r svarer til "selectivity" (Q-faktor) for et analogt notch-filter: Jo højere r, jo stejlere filter[1]. For et analogt filter er Q-faktor givet ved $Q = \frac{\omega_c}{\text{Bandwidth}}$. Antaget nogenlunde tilsvarende for det digitale filter: båndbredden omkring notch-frekvensen mindskes som r øges. Værdien r er valgt til $r = 0.99$ (ved iterative forsøg) for at få et stejlt filter, der også fungerer godt på target. Dermed kan filterets koefficienter beregnes:

```
r = 0.99;                              % Modulus for poler ("Q-faktor")
wc = pi * round(fidx) / (fs / 2);   % Centerfrekvens [rad/s]
K = 1;                                 % Til første iteration af DC gain

for iteration=1:2
    G = 1/K;                  % Skalering så DC Gain = 1 (0 dB). G=0.9991.

    % Følgende koefficienter ift. differensligning.
    b0 = G;                   % z^0
    b1 = -2*cos(wc)*G;        % z^-1
    b2 = G;                   % z^-2
    a1 = 2*r*cos(wc);         % z^-1
    a2 = -r*r;                % z^-2

    % Følgende polynomier ift. overføringsfkt.
    b = [b0 b1 b2];           % B(z)
    a = [1 -a1 -a2];          % A(z)

    K = sum(b)/sum(a);        % DC gain (Lyons s. 300)
end
```

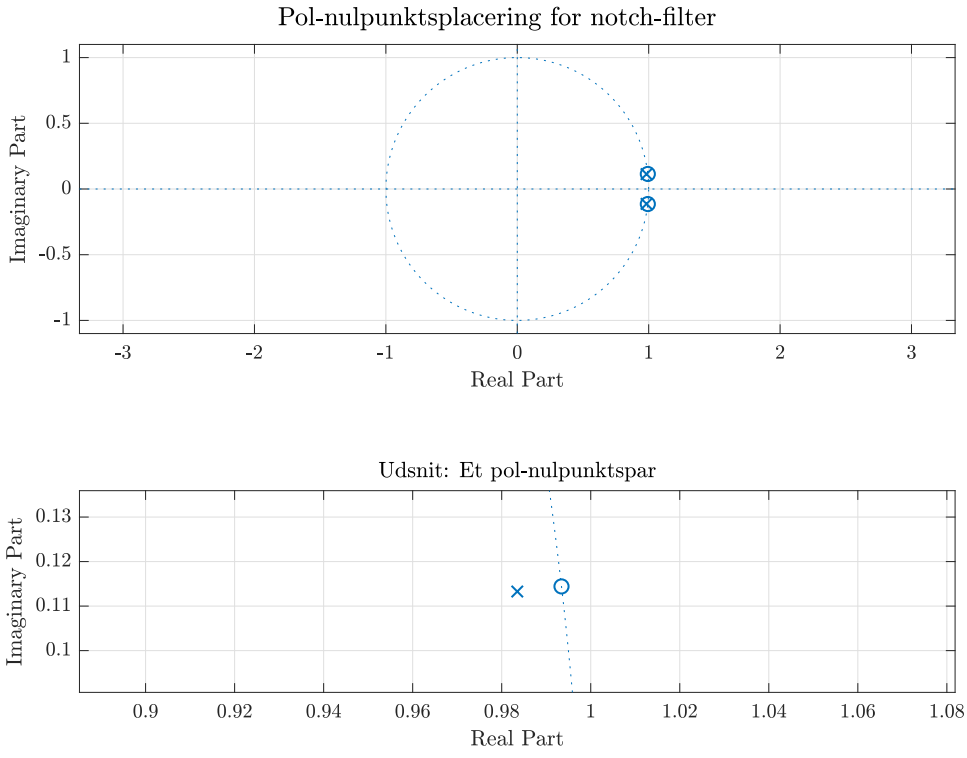
3.4 Pol-/nulpunktsdiagram

Figuren nedenfor viser, hvad der indledningsvist blev beskrevet om pul-nulpunktsplacering: De ligger meget tæt, med nulpunkterne på enhedscirklen.

```
p = roots(a);                                        % Find poler

figure()
sgtitle('Pol-nulpunktsplacering for notch-filter')
subplot(211)
zplane(b,a); grid on;

subplot(212)
zplane(b,a); grid on;
title('Udsnit: Et pol-nulpunktspar')
```

[1] Der må være en algebraisk sammenhæng via den bilineære transformation?

```
xlim([real(p(1))*0.9 real(p(1))*1.1]);      % Vis tæt udsnit af et pol-
ylim([imag(p(1))*0.8 imag(p(1))*1.2]);        % nulpunktspar
```



3.5 Differensligning og systemfunktion

Differensligningen nedenfor er ikke til direkte implementering på DSP, da koefficienter skal skaleres og kvantiseres først. Overføringsfunktion er også opskrevet.

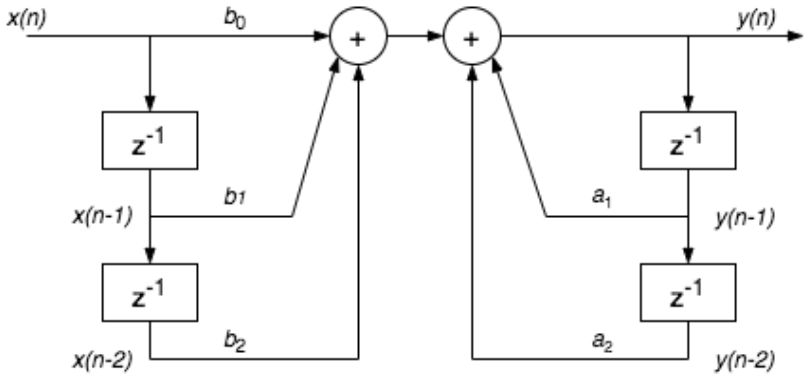
```
% Differensligning:
feedforward = [num2str(b0) ' x(n) ' ...
               num2str(b1) ' x(n-1) + ' ...
               num2str(b2) ' x(n-2)'];
feedback =    [num2str(a1) ' y(n-1) ' ...
               num2str(a2) ' y(n-2) '];
diffeq = ['y(n) = ' feedforward ' + ' feedback];
disp(diffeq);

Hsys = tf(b, a, 1/fs)   % Vis fin overføringsfunktion
```

```
y(n) = 0.99761 x(n) -1.9821 x(n-1) + 0.99761 x(n-2) + 1.967 y(n-1) -0.9801 y(n-2)

Hsys =

  0.9976 z^2 - 1.982 z + 0.9976
  -----------------------------
     z^2 - 1.967 z + 0.9801
```

```
Sample time: 2.0833e-05 seconds
Discrete-time transfer function.
```

3.6 Signalgraf (direkte form 1)

Filteret bliver implementeret på direkte form 1. Figuren nedenfor viser filterstrukturen. Værdien af koefficienterne er angivet i foregående afsnit. Disse er først endelige efter skalering og kvantisering til target.



Figur 3.1: Signalgraf

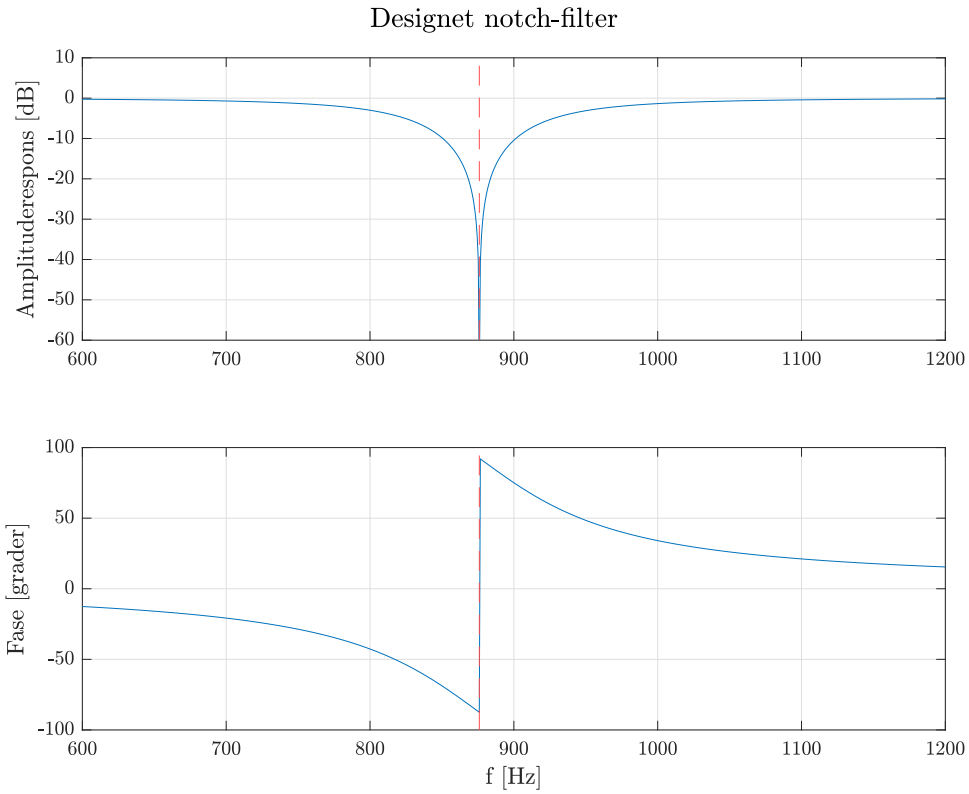
Signalgrafen viser, at denne implementering er en kaskade af en feedforward-sektion og en feedback-sektion [2, s. 158]. Ved implementering i C vil feedforward-sektion have 3 memory-elementer, hhv. 1 til $x(n)$ (input) og 2 til $x(n-1)$ og $x(n-2)$ (delay-line). Feedback-sektionen skal ligeledes have 3 memory-elementer; 1 til $y(n)$ (output) og 2 til $y(n-1)$ og $y(n-2)$ (delay-line). Det kræver 5 multiplikationer og 4 summationer at beregne et ouput. Hvis vi havde benyttet direkte form 2 (byttet om på sektionerne) kunne vi nøjes med i alt 2 memory elementer til hele delay-line [2, s. 159].

3.7 Frekvensrespons

Frekvensresponset (amplitude- og faserespons) regnes for IIR notch-filteret. Frekvensreponset $H(e^{j\omega})$ regnes vha. ratioen på to FFT'er [3].

```
1  Nfft = 2^16;
2  H = fft(b, Nfft) ./ fft(a, Nfft);
3  figure();
4  sgtitle('Designet notch-filter', 'Interpreter','Latex', 'FontSize', 14)
5  subplot(211)
6  plot( (fs/Nfft)*(0:Nfft-1), 20*log10(abs(H)));
7  xlim([600 1200]); ylim([-60 10]); grid on;
8  ylabel('Amplituderespons [dB]', 'Interpreter','Latex', 'FontSize', 12);
9  xline(round(round(fidx)),'r--');
10 subplot(212)
11 plot( (fs/Nfft)*(0:Nfft-1), (180/pi)*unwrap(angle(H)));
```

```
12 xlim([600 1200]); grid on;
13 xline(round(round(fidx)),'r--');
14 xlabel('f [Hz]', 'Interpreter','Latex', 'FontSize', 12);
15 ylabel('Fase [grader]', 'Interpreter','Latex', 'FontSize', 12);
```



Amplituderesponset viser, at filteret rammer den ønskede frekvens (rød, lodret linje), og at gain i hele pasbåndet er 0 dB. Filteret er dog **ikke** ideelt, da transitionen fra pas- til stopbånd ikke er undelig skarp/hurtig (roll-off ikke uendeligt stejl). Punkterne for $-3\,\text{dB}$ ligger ved hhv. 800 Hz og 950 Hz. Dvs. ca. 150 Hz båndbredde i stopbåndet. Filterets selektivitet (Q-faktor) er da ca. $\frac{876}{150} = 6$. Det kan nok gøres bedre.

Faseresponset viser en kvalitet ved et IIR notch-filter: Fasen er flad i det meste af pasbåndet (dvs. group-delay er 0), og i det meste af pasbåndet vil der ikke opstå faseforvrængning. Nærmere center-frekvensen er fasen ikke-lineær, og der kan opstå faseforvrængning i området ca. 700-800 Hz og igen ved ca. 950-1050 Hz. Det er altså en fordel at være mere selektiv i valg af frekvens; forudsat at det kan håndteres med fixed-point på target, at frekvensen kendes på forhånd og at frekvensen er stationær.

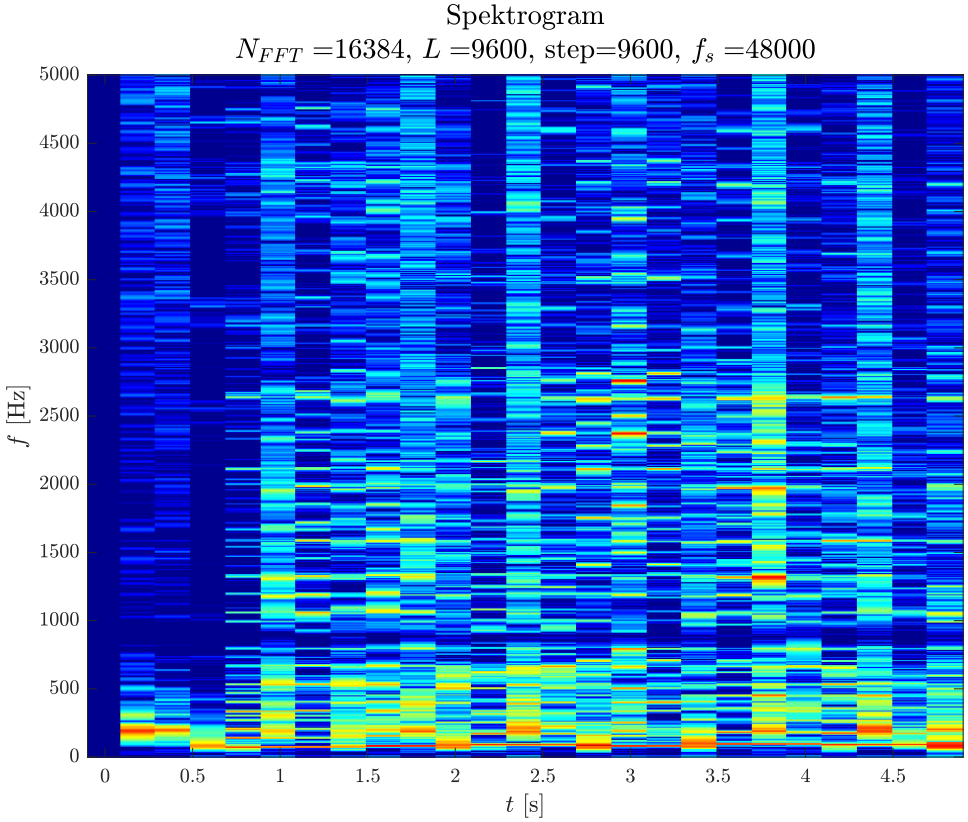
Et IIR-filter af højere orden (kaskade) kunne også give mening, og det kunne designes ud fra yderligere krav til responset: Stejlhed (bandwidth el. Q-faktor), dæmpning i stopbånd, faserespons/-group delay, osv. Vi arbejder med lyd her, og kunne nok bruge et filter med maksimalt lineær fase (Bessel). Så det kunne fx også være et designkriterium. Design vha. et analogt prototypefilter og konvertering til digitalt filter med den bilineære z-transformation ville give mening.

Det er ærgerligt, at der ændres på musiksignalet i de nævnte frekvensområder, da der netop er meget god lyd her - bas, percussion, osv. En bedre tilgang ville være et adaptivt filter, der helt specifikt kunne fjerne sinustonen.

3.8 Test af filter i Matlab

Filteret testes på lydklippet. For at påvise, at der ikke længere er en forstyrrende frekvens i signalet, beregnes igen et spektrogram, med samme indstillinger som tidligere.

```
xfilt = filter(b,a,x);                  % filtrér lydklippet
% soundsc(xfilt,fs)                     % lyt til klippet

tl = 5;                                 % tidslængde til analyse, sekunder
x_ = xfilt(1:tl*fs)';                   % udvalgt sektion af filtreret signal
L = fs/5;                               % vindueslængde 9600 => 5 Hz opløsning
stepSize = L;                           % step er 1 vindue (0% overlap)
Nfft = 2^nextpow2(L);                   % 2^14
figure();
spectrogram0(x_, L, Nfft, stepSize, fs, [0 5000]); % Vis kun op til 5 kHz
```



Spektrogrammet viser nu, at den rene tone er fjernet fra signalet. Det samme er bekræftet ved at lytte til signalet. Spektrogrammet viser også tydeligt, at en del information er mistet pga. dæmpning i frekvensområdet omkring centerfrekvensen. Der ses et tydeligt "dødt" område omkring filterets centerfrekvens. Filteret virker altså - i hvert fald med 64-bit-præcision - som ønsket.

4. Opgave 3: Algoritmeudvikling og implementering på signalprocessor

4.1 Kvantisering og algoritmeudvikling

Dette afsnit analyserer og tester kvantisering af filterkoefficienterne. Desuden testes algoritmen (floating-point $\longrightarrow$ fixed-point), der implementeres på target DSP'en. Der foretages test af algoritmer *og* koefficienter sammen, før der implementeres på hardware.

Koefficientkvantisering ændrer filterets karakteristik / respons. Så der er risiko for et anderledes respons end hvad fås med infinite precision:

- Ændrede koefficienter (afrunding) ændrer filterets respons: Kritiske frekvenser flyttes - muligvis signifikant. I yderste konsekvens bliver IIR-filteret også ustabilt, hvis kvantisering (afrunding) flytter pol(er) uden for enhedscirklen [2, s. 170].
- Kvantisering fra 64-bit *floating-point* til 16-bit *fixed-point* finite precision giver løbende afrundings-/trunkeringsfejl i filtrering: Fordi det er et IIR-filter, så feedes disse fejl tilbage ind filteret, og kan akkumulere. Det kan give oscillationer (limit cycles) [1, s. 293].
- Filterorden og filterstruktur påvirker ovenstående risiko for ustabilitet forårsaget af kvantisering [1, s. 293]: Det kan være en bedre strategi at implementere kaskadekobling af 1./2.ordenssystemer, end at have et filter af højere orden [2, s. 78] [4]. Så det er fornuftigt nok at arbejde med et 2. ordens filter her, og så evt. kaskadere (og skalere), efter behov.

Med MATLABs `fixed-point designer`, `filterDesigner` og `fvtool` kan man analysere effekten af kvantisering af filteret.

4.1.1 S15-kvantisering af filterkoefficienter

Værdiområdet for S15 er $1 \leq x < 1 - 2^{-15}$. Koefficienterne b_1 og a_1 er numerisk større end $1 - 2^{-15}$ men numerisk mindre end 2. Så ved division med 2 nedskaleres til S15. Der er umiddelbart to metoder til at håndtere dette i differensligningen:

- Nedskalér *kun* de to koefficienter med 2, og "opvej" skaleringen ved at indregne deres led dobbelt i differensligningen. Dvs. $y(n) = \ldots + \frac{b_1}{2}x(n-1) + \frac{b_1}{2}x(n-1) + \ldots$.
- Nedskalér *alle* koefficiencter med faktor 2. Skaleringen opvejes i differensligningen ved at akkumulere dobbelt, dvs. afsluttende MAC-operation er `akk += akk`. Da både $B(z)$ og $A(z)$ skaleres, ændres frekvensresponset ikke.

Førstnævnte vælges, da det påvirker færrest koefficienter. Herunder vises princippet i S15-kvantiseringen. I det følgende repræsenterer b_n bits og *ikke* filterkoefficienter!

I S15 *forestiller* vi os, at der er et binærkomma $k = 15$ pladser fra højre (LSB), og at MSB repræsenterer tallets fortegn. S15 bitmønsteret med indsat binærkomma er $b_0.b_1b_2b_3 \cdots b_{13}b_{14}b_{15}$. Mønsteret repræsenterer radix-10 kommatalsværdien $-b_0 + \sum_{n=1}^{k=15} b_n 2^{-n}$.

Så for et tal $0 \leq u_1 < 1$ gælder repræsentationen $\sum_{n=1}^{k=15} b_n 2^{-n} \longleftrightarrow u_1$ som er ækvivalent til $b_1 2^{14} + b_2 2^{13} + \ldots + b_{15} \longleftrightarrow u_1 2^{15}$. Venstresiden er et heltal på binær form (radix-2 med 15-bits). En evt. overskydende kommadel i $u_1 2^{15}$ på højresiden trunkeres, da der ikke er bits på venstresiden til at repræsentere den.

Derfor kan u_1 omregnes fra kommatal til S15 med `floor`$(u_1 2^{15})$. Tydeligvis fås en numerisk mindre fejl, hvis vi i stedet benytter `round`$(u_1 2^{15})$, antaget at evt. oprunding kan indeholdes i wordlength uden overflow[1]. Lagring af dette i signed heltalsbitmønster lader vi compiler/MATLAB håndtere (det er 2-komplement).

Det negative tal $u_2 = -u_1$ repræsenteres ved $u_2 = -u_1 \longleftrightarrow$ `-round`$(u_1 2^{15})$ = `round`$(u_2 2^{15})$. For begge tal benyttes altså en skaleringsfaktor $K = 2^k = 2^{15}$, som svarer til bitshift og cast `(short) (u1 << 15)`. Ved lagring som bits håndteres fortegnsbit med to-komplement. Den binære repræsentation af u_2 kan fx[2] beregnes ved $(2^{16})_2 - (u_1)_2$. Et negativt tal vil altid have $b_0 = 1$.

Regneeksempel: To tal, $u_1 = 0.9$ og $u_2 = -0.9$, konverteres til S15 og multipliceres som på en 16-bit fixed-point platform.

```
u1 = 0.9; u2 = -u1;
B = 16;                 % B er længden på hele binær-ordet (word length)
k = 15;                 % k er længden på brøk-delen (fraction length)
K = 2^k;

U1 = round(u1*K);       % u1 -> S15
U2 = round(u2*K);       % u2 -> S15
disp( ['u1 -> S15: ' num2str(U1) newline ...
       'u2 -> S15: ' num2str(U2)] );
disp([newline 'To-komplementtallenes bits:']);
disp(['U1 bin -> ' num2str( bitget(int16(U1), 16:-1:1)' )' newline ...
      'U2 bin -> ' num2str( bitget(int16(U2), 16:-1:1)' )' ]);
```

```
u1 -> S15: 29491
u2 -> S15: -29491

To-komplementtallenes bits:
U1 bin -> 0111001100110011
U2 bin -> 1000110011001101
```

Da skaleringsfaktoren også multipliceres, kræves 32-bit for repræsentere produktet $U_1 U_2$ = `round`$(u_1 u_2 2^{30})$. S15-formatet kan genetableres ved at dividere med skaleringsfaktoren, så i S15: $u_1 u_2 =$ `round`$(U_1 U_2 2^{-15})$. Decimatallet genetableres ved yderlige nedskalering med 2^{15} uden afrunding.

```
disp( ['Multiplikation af u1 og u2 i S15 -> ' num2str(round(U1*U2/K)) ] );
```

[1] Det kræver mange flere operationer at tage `round` end at tage `floor`, så mens det er OK til omregning af koefficienter i MATLAB , så duer det slet ikke til løbende MAC-beregninger på DSP.

[2] Alternativt $2^{16} - 1 - (u_1)_2 + 1$, hvilket er et-komplement og addering med 1.

```
disp( ['Resultat i decimal -> ' num2str(round(U1*U2/K)/K) ] );
```

```
Multiplikation af u1 og u2 i S15 -> -26542
Resultat i decimal -> -0.81
```

På en fixed-point-platform caster man u_1 og u_2 til en bredere type, `U1U2 = (long) u1 * u2` og konverterer og trunkerer bagefter vha. `(short) (U1U2 >> 15)`[3]. Dette kan emuleres:

```
U1U2_Q30 = int32(U1)*int32(U2);                              % (long) U1*U2

% Manipulation af bitmønstrene:
U1U2_S15_bits = bitget(U1U2_Q30, 31:-1:16 );                 % U1U2 >> 15
U1U2_S15 = sum( int16(U1U2_S15_bits) .* ...
                 int16([-1*2^15 2.^(14:-1:0)]) );            % (short) ...
u1u2 = sum( double(U1U2_S15_bits) .* [-1 2.^(-1:-1:-15)] ); % radix-10 dec.

disp( ['(U1*U2) bits     -> '  num2str(bitget(U1U2_Q30, 32:-1:1)' )' ]);
disp( ['(U1*U2) S15 bits  -> ' num2str(U1U2_S15_bits' )' newline ...
       '(U1*U2) i S15     -> ' num2str(U1U2_S15) newline ...
       '(u1*u2) i decimal -> ' num2str(u1u2) ] );
```

```
(U1*U2) bits     -> 11001100001010010010001111010111
(U1*U2) S15 bits  -> 1001100001010010
(U1*U2) i S15     -> -26542
(u1*u2) i decimal -> -0.81
```

Hvilket demonstrerer, at det rigtige resultat også frembringes ved direkte bitmanipulationer.

4.1.2 Test af effekt af kvantisering i Matlab

For at se effekt af kvantisering, oprettes et diskret filterobjekt som direkte form 1, med kvantiserede koefficienter (fixed-point). Denne fremgangsmåde er inspireret af [2, s. 125 ff.].

```
Hd = dfilt.df1(b,a);              % Ikke-kvantiseret filter
Hdq = copy(Hd);                   % Kopiér objekt så vi evt. kan sammenligne

% Kvantisér filteret
Hdq.Arithmetic='fixed';           % Fixed-point-filter
Hdq.RoundMode='floor';            % Trunkering
Hdq.CoeffAutoScale = false;
Hdq.CoeffWordLength = 16;
Hdq.DenFracLength = 14;           % Kan ikke reduceres til S15 pga værdiomr.
Hdq.NumFracLength = 14;
Hdq.ProductMode='SpecifyPrecision';
```

[3] Det giver selvfølgelig en anden fordeling for afrundingsfejl end eksemplerne med `round` vist her.

```
Hdq.ProductWordLength = 32;      % (long) b0*x(n) ind i akkumulator
Hdq.InputWordLength = 16;
Hdq.InputFracLength = 15;
Hdq.OutputWordLength = 16;
Hdq.OutputFracLength = 15;
Hdq.CastBeforeSum = true;

x_filt_q = filter(Hdq, x);       % Filtrér med kvantiseret filter
x_filt_fp = double(x_filt_q);    % Konvertér filtreret sign. til floating pt
% soundsc(x_filt_fp, fs);            % Afspil filtreret lydsignal - OK!
% freqz(Hdq, 'half');              % åbner FVtool og viser frekvensrespons,
                                   % pol-/nulpunktsdiagram mv.
```

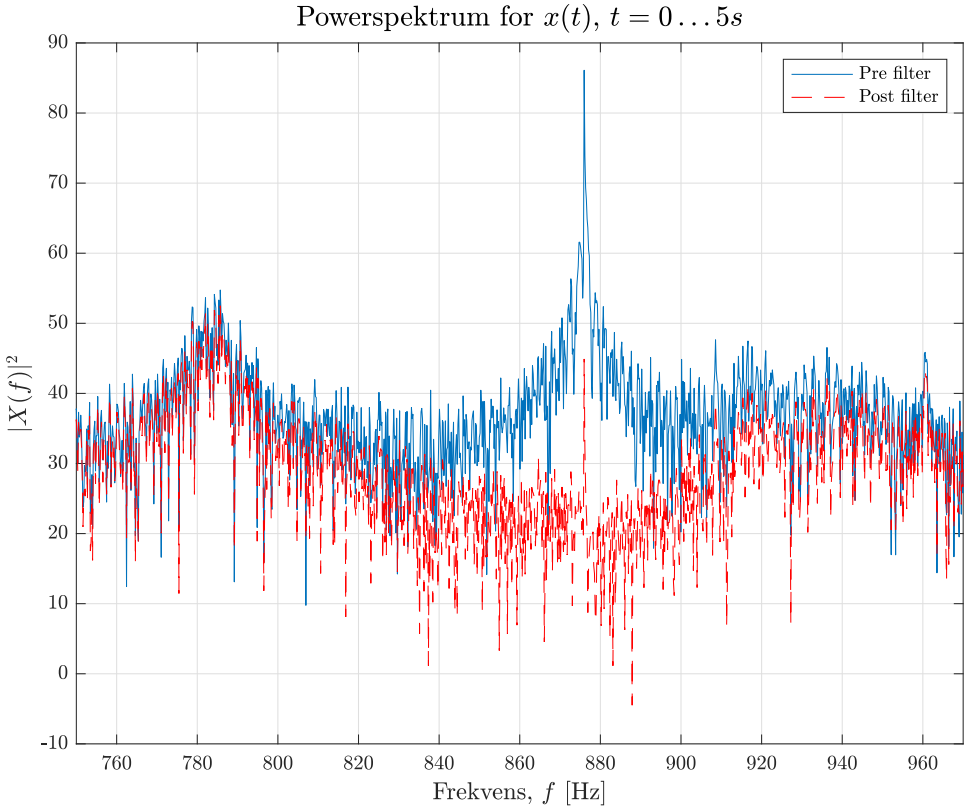
Ovenstående forsøg bekræfter, at filteret også virker, når det er kvantiseret (med trunkering). `FVtool` viser et pol-/nulpunktsdiagram (ikke illustreret her), som bekræfter at polerne stadig ligger inden for enhedscirklen (stabilt). Der vises også et frekvensrespons, som bekræfter, at centerfrekvensen ikke er flyttet signifikant sfa. kvantisering. De oprindelige og kvantiserede koefficienter vises.

4.1.3 Powerspektrum for kvantiseret filter

Sammenligning af powerspektra for kvantiseret filtrering og oprindeligt signal vises i figuren nedenfor. Tre observationer:

- Den rene tone er dæmpet omkring 40 dB (ingen uendelig dæmpning med kvantiseret filter). Dog er en del af signalet omkring tonen også blevet dæmpet i processen.
- Powerspektra matcher fint i pasbåndet, så kvantiseret filtrering har ikke generelt "ødelagt" signalet.
- Når frekvensen nærmes centerfrekvensen, ændres signalet gradvist, hvilket stemmer overens med de tidligere nævnte frekvensområder.

```
x_post_ = x_filt_fp(1:tl*fs);          % Udvælg 5s af filtreret signal
Nfft = 2^nextpow2(length(x_post_));    % 2^18 -> zero-padding med 22144 0'er
X_post_ = fft(x_post_, Nfft);
X_post_pow = X_post_ .* conj(X_post_);       % Powerspektrum
f_vec = (fs / Nfft) * (0:Nfft-1);            % Tilhørende frekvensakse

figure();
plot(f_vec, 10*log10(X_pow)); hold on;
plot(f_vec, 10*log10(X_post_pow), 'r--'); hold off;
xlim([750 970]); grid on;
legend({'Pre filter', 'Post filter'});
ylabel('$|X(f)|^2$', 'Interpreter','Latex', 'FontSize', 12);
xlabel('Frekvens, $f$ [Hz]', 'Interpreter','Latex', 'FontSize', 12);
title('Powerspektrum for $x(t)$, $t=0 \ldots 5s $', ...
    'Interpreter', 'Latex', 'FontSize', 14)
```



Kvantiseringsfejlen (pga. præcision og trunkering) har et spektrum og en sandsynlighedsfordeling. Men her antages bare, at denne fejl er jævnt fordelt hvidstøj med middelværdi på 0, og at der ikke skal foretages yderligere.

4.1.4 Output af Matlab-kvantiserede filterkoefficienter

Det indebyggede `filterDesigner`-værktøj kan eksportere filterkoefficienterne til en C-header:

- `File > Import filter from Workspace > Filter object > Hdq`.
- `Targets > Generate C Header > Export as > Signed 16-bit integer > Export`.

Da koefficienterne ligger i intervallet $-2 \leq x < 2 - 2^{-14}$, vil `filterDesigner` eksportere som Q2.14, dvs. med kun 14 fractional bits.

Bemærk også fortegn på a_1 og a_2 (DEN[1] og DEN[2]). Fortegnene er som i polynomiet $A(z)$, dvs. omvendt af hvad der bruges i differensligningen, som jeg har opskrevet den.

For koefficienterne i værdiområdet for S15, dvs. b_0, b_2 og a_2, vil et venstre bitshift (<< 1) konvertere fra Q2.14 til S15. For koefficienter uden for værdiområde (dvs. b_1 og a_1), kan koefficienterne benyttes direkte jf. diskussionen omkring skalering med 2 og efterfølgende dobbelt-addition. Koefficienter modsvarende MATLABs benyttes i implementeringen.

```
1 /* General type conversion for MATLAB generated C-code  */
2 #include "tmwtypes.h"
3 /*
4  * Expected path to tmwtypes.h
5  * /Applications/MATLAB_R2018b.app/extern/include/tmwtypes.h
```

```
 */
const int NL = 3;
const int16_T NUM[3] = {
    16345, -32475,  16345
};
const int DL = 3;
const int16_T DEN[3] = {
    16384, -32227,  16058
};
```

Listing 4.1: Eksporterede filterkoeff.

4.2 Algoritmetest: Floating-point implementering af IIR-differensligning i Matlab

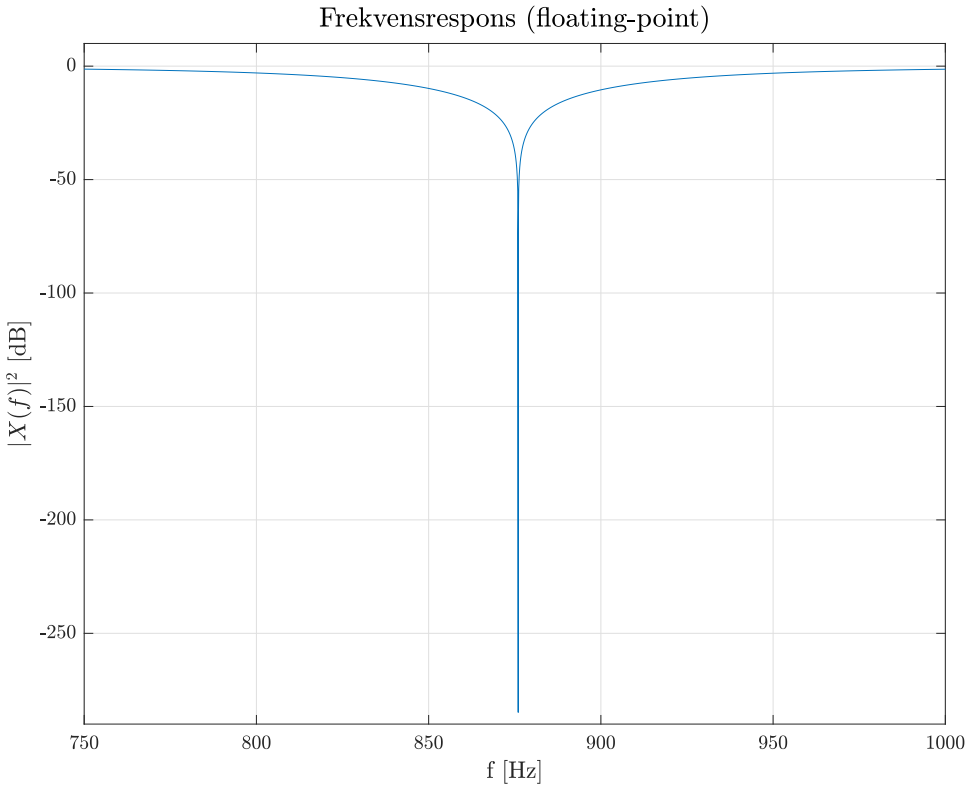
Forud for en testalogritme med fixed-point, laves en reference med floating-point. Der benyttes en lineær buffer. Det giver et par få ekstra operationer. Det giver ikke mening at implementere en cirkulær buffer i MATLAB da der ikke findes pointers, og delay lines kun består af 2 elementer hver.

```
delay_x = [0 0];    % delay line til feedforward
delay_y = [0 0];    % delay line til feedback

N = length(x);
y = zeros(1,N);

% impulsrespons
delta = [1 zeros(1, N-1)];

%in = x;            % Beregn filtrering af lydsignal
in = delta;         % Beregn impulsrespons

for n = 1:N
    % Implementér differensligning:
    y(n) = b0*in(n) + b1*delay_x(1) + b2*delay_x(2) + ...
            a1*delay_y(1) + a2*delay_y(2);

    % Opdatér delay line ved at shifte nyeste værdi ind
    delay_x = [in(n) delay_x(1)];
    delay_y = [y(n) delay_y(1)];
end
```

Ved aflytning af det filtrerede signal bekræftes det, at algoritmen har virket som ønsket. Der er også kørt en impuls igennem filteret, og impulsresponset er tranformeret til frekvensrespons herunder.

```
% soundsc(y, fs)
% clear sound
```

```
3
4  f_vec = (0:N-1)*(fs/N);
5  figure();
6  plot(f_vec, 20*log10(abs(fft(y))));
7  grid on; xlim([750 1000]); ylim([-290 10]);
8  xlabel('f [Hz]', 'Interpreter','Latex', 'FontSize', 12);
9  ylabel('$|X(f)|^2$ [dB]', 'Interpreter','Latex', 'FontSize', 12);
10 title('Frekvensrespons (floating-point)', 'Interpreter','Latex', 'FontSize'
       , 14);
```



4.3 Algoritmeudvikling: Fixed-point implementering af IIR-differensligning i Matlab

Ud fra floating-point-algoritmen udarbejdes her en fixed-point-algoritme, som emulerer DSP'ens MAC. Formålet er at forstå og analysere forskellene til floating-point, og at forberede implementering på target DSP'en. Der benyttes S15-kvantiserede koefficienter og input (16-bit præcision). "Akkumulatoren" i MATLAB er 64-bit, mens DSP'ens egen 32/40-bit akkumulator har 8 guard-bits, og kan håndtere 256 32-bit additioner uden overflow. Så overflow-aspektet behøver vi ikke at simulere her.

```
1  K = 2^15;                    % Benyttes til <<15 og >>15 operationer
2
3  % Koefficienter i S15
4  b0_ = round(b0 * K);         % Svarer til (short) (b0 << 15)
5  b1_ = round(b1/2 * K);       % Bem. b1/2, så skal akkumuleres dobbelt
6  b2_ = round(b2 * K);
```

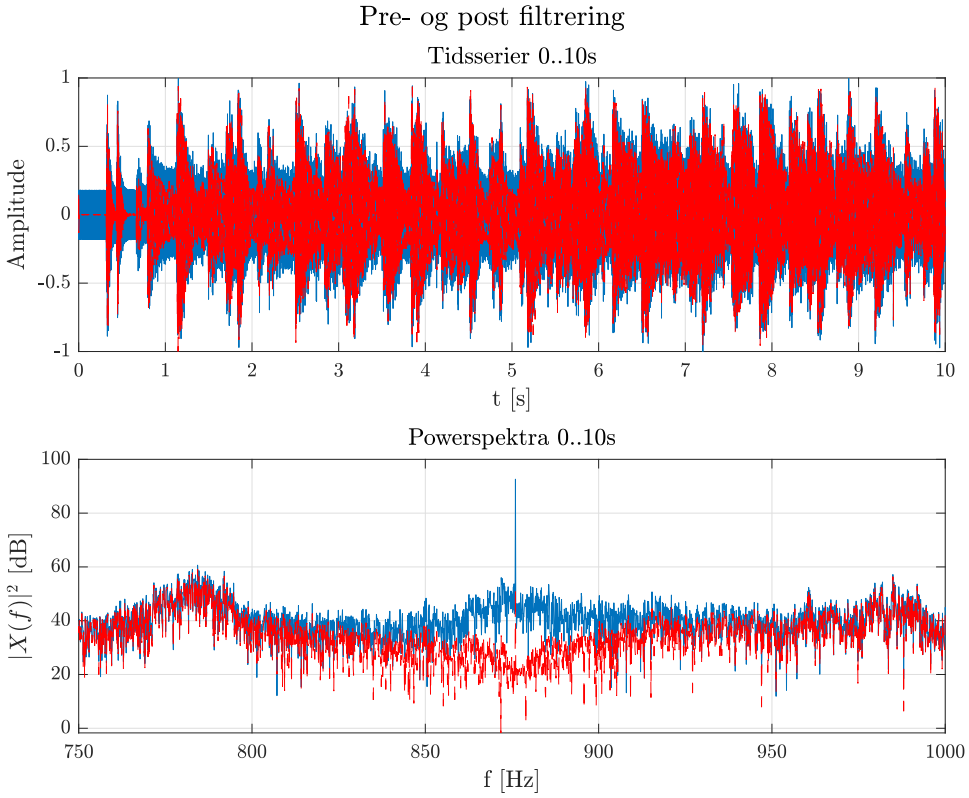
```
a1_ = round(a1/2 * K);      % Bem. a1/2, så skal akkumuleres dobbelt
a2_ = round(a2 * K);

% Kvantisér input. Dette skal ikke gøres i C.
% Først skaleres til værdiområde for S15
% -> der er meget få elementer |x|>1 , disse clippes i stedet for at
% re-skalere hele serien.
x_ = x;
x_(x_ >= 1) = 1-2^-15;      % Clippes til max-værdi for S15
x_(x_ < -1) = -1;           % Clippes til min-værdi for S15
x_ = round(x_*K);           % Kvantisering

N = length(x_);
y = zeros(1, N);

dx = [0 0];                 % Nulstil delay lines
dy = [0 0];

for n=1:N
    % Implementér differensligning med MAC-operationer
    acc = b0_*x_(n);
    acc = acc + b1_*dx(1);
    acc = acc + b1_*dx(1);  % Adderer 2 gange fordi koefficient er b1/2
    acc = acc + b2_*dx(2);
    acc = acc + a1_*dy(1);
    acc = acc + a1_*dy(1);  % Adderer 2 gange fordi koefficient er a1/2
    acc = acc + a2_*dy(2);

    y(n) = round(acc/K);    % (short) (acc >>15),  Q2.30 -> Q1.15 (S15)

    % Opdatér delay line til næste iteration ved at shifte nyeste værdi ind
    dx = [x_(n) dx(1)];     % Bliver [x(n-1) x(n-2)]
    dy = [y(n) dy(1)];      % Bliver [y(n-1) y(n-2)]

end

y = y/K;                    % Omregn y tilbage til værdiområdet [-1;1[
```

Afspilning af nummeret efter filtrering bekræfter, at den forstyrrende tone er fjernet. Filteret virker altså også med fixed-point aritmetik. Det er oplagt at sammenligne tids- og frekvensserier hhv. før og efter filtrering, for at se filterets indvirkning.

```
% soundsc(y, fs)
% clear sound
```

```
Tmax = 10;      % sek
n = (1:Tmax*fs);
```

```
t_vec = (n-1)/fs;
f_vec = (0:Tmax*fs-1)*(fs/(Tmax*fs));

figure();
setlatexstuff('latex');
sgtitle('Pre- og post filtrering', 'Interpreter','Latex', 'FontSize', 14)

subplot(211)
plot(t_vec, x(n) / max(abs(x(n))));
hold on;
plot(t_vec, y(n) / max(abs(y(n))), 'r--');
hold off;
grid on;
xlabel('t [s]', 'Interpreter','Latex', 'FontSize', 12);
ylabel('Amplitude', 'Interpreter','Latex', 'FontSize', 12);
title('Tidsserier 0..10s', 'Interpreter','Latex', 'FontSize', 12);

subplot(212)
plot(f_vec, 20*log10(abs(fft(x(n) / max(abs(x(n)))))));
hold on;
plot(f_vec, 20*log10(abs(fft(y(n) / max(abs(y(n)))))), 'r--');
hold off;
grid on;
xlim([750 1000]);
ax = gca; ax.XAxis.Exponent = 0;
xlabel('f [Hz]', 'Interpreter','Latex', 'FontSize', 12);
ylabel('$|X(f)|^2$ [dB]', 'Interpreter','Latex', 'FontSize', 12);
title('Powerspektra 0..10s', 'Interpreter','Latex', 'FontSize', 12);
```



Øverste figur over tidsdomænet viser, at bortfiltrering af tonen har taget energi (amplitude) ud af signalet. Det er mest markant i starten af skæringen, hvor der kun er sinustonen og ingen musik. For den filtrerede tidsserie ses i øvrigt en indsvingning af filteret.

I effektspektra (kun vist i intervallet 750-1000 Hz, ses, at filtreringen har dæmpet den rene 876 Hz-tone med ca. 40 dB. Kun i den nære omegn af notchet (notch båndbredde), er frekvensindholdet blevet ændret af filteret. Det er svært at høre, at "der mangler noget" i forhold til originaloptagelsen.

Ovenstående analyse giver noget vished om, at algoritme og koefficienter svarende til ovenstående burde virke på signalprocessoren. Næste skridt er implementering på hardware.

4.4 Opsætning af projekt i CCS

Som udgangspunkt for kodeprojektet er opskriften "Audio Loop Through" benyttet [5]. Da jeg i dette projekt driver input med wavegenerator, og ikke en mikrofon, er gain i ADC/DSP reduceret til 0 dB. Samplingsfrekvens er fastholdt på 48 kHz da filteret er designet dertil, og der intet behov er for at ændre derpå. Opsætning i `main.c` ses herunder:

```
1 printf("E4DSA Case 2 (Janus) - IIR notch filter DSP: ");
2
3 /* Setup sampling frequency and 0 dB gain for line in */
4 set_sampling_frequency_and_gain(SAMPLES_PER_SECOND, 0);
```

Listing 4.2: Opsætning i main.c

Desuden er bias af mikrofon slået fra (til fx mikrofoner i headset, som skal have bias for at virke). Denne registerindstilling er nævnt i [6].

```
AIC3204_rset( 0, 1 );      // Select page 1
//AIC3204_rset( 51, 0x48); // power up MICBIAS with AVDD (0x40) or LDOIN (0x48)
```

Listing 4.3: aic3205_init.c

4.5 Implementering af differensligning i C

Implementering af filteret er i to filer, `iir_notch.h` og `iir_notch.c`. I header-filen findes koefficienter og prototype på funktion. I c-filen findes implementeringen. I `main.c` er der implementeret et uendeligt loop, der tager input fra ADC (line-in), kalder filterfunktion og sender resultat ud af DAC (line-out). Som det ses, benyttes kun højre kanal, for at kunne benytte den venstre kanal til en anden filtrering eller til det ufiltrerede output i forbindelse med tests.

```
while (1) {
  aic3204_codec_read(&left_input, &right_input);
  right_output = filter_iir_notch(iir_b, iir_a, right_input);
  left_output =  0;
  aic3204_codec_write(left_output, right_output);
}
```

Listing 4.4: Uendelig løkke i main.c

Header-filen indeholder koefficienterne beregnet i designafsnittet. Som kan ses i koden, er der eksperimenteret med forskellige sæt koefficienter, der repræsenterer forskellige Q-faktor, for at se og høre forskel på filterets funktion og respons.

```
/*
 * iir_notch.h
 *
 *  Created on: 10 Mar 2020
 *      Author: Janus Bo Andersen (JA67494)
 *      Interface for the IIR filter function
 *      Defines the filter coefficients for a 876 Hz notch filter
 */

#ifndef IIR_NOTCH_H_
#define IIR_NOTCH_H_

    /*                          b0     b1 / 2   b2 */
 /* const signed int iir_b[3] = {32738, -32523, 32738}; */
    const signed int iir_b[3] = {32690, -32475, 32690};

    /*                          a0     a1 / 2   a2 */
 /* const signed int iir_a[3] = {32767,  32520, -32702}; */
    const signed int iir_a[3] = {32767,  32227, -32116};

    /* The filter takes b and a coefficients, and input from line-in */
    signed int filter_iir_notch(const signed int * b,
```

```
                                const signed int * a, signed int input);

#endif /* IIR_NOTCH_H_ */
```

Listing 4.5: iir_notch.h

Implementering af filterfunktionen er inspireret af [7, kap. 7, slide 39] og [2, s. 181]. Som kan ses, er delay lines implementeret som lineære buffers (så få koefficienter, at det er en ligegyldig optimering at bruge en cirkulær buffer her). Filteret er i direkte form 1, så der er delay lines for både $x(n)$ og $y(n)$. Bemærk også, at MAC-operation for b_1 og a_1 forekommer dobbelt, da disse to koefficienter er halveret for at passe ind i S15-formatet. Algoritme og casts/shifts er som udviklet i et tidligere afsnit.

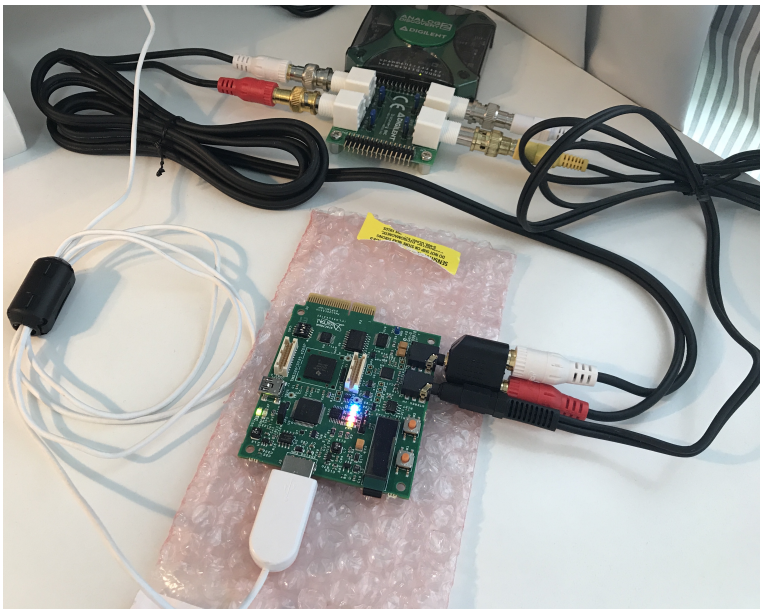
```
# define NATIVE_MAX 32767   /*  2^15 - 1 */
# define NATIVE_MIN -32768  /* -2^15     */

/* The filter takes b and a coefficients, and input from line-in */
signed int filter_iir_notch(const signed int * b,
                            const signed int * a, signed int input) {

    /* Delay line as static variables with persistence between calls */
    static signed int dx[2] = {0, 0};    /* x(n-1), x(n-2) */
    static signed int dy[2] = {0, 0};    /* y(n-1), y(n-2) */

    /* Accumulator 32-bit (8 guard bits for overflow) */
    long acc = 0;

    /* difference equation, coerce all data into
     * sign extended 32-bit words during calculation */
    acc =  ( (long) b[0] * input );     /* b0 x(n)*/
    acc += ( (long) b[1] * dx[0] );     /* b1 x(n-1) */
    acc += ( (long) b[1] * dx[0] );     /* added twice due to scaling */
    acc += ( (long) b[2] * dx[1] );     /* b2 x(n-2) */
    acc += ( (long) a[1] * dy[0] );     /* a1 y(n-1) */
    acc += ( (long) a[1] * dy[0] );     /* added twice due to scaling */
    acc += ( (long) a[2] * dy[1] );     /* a2 y(n-2) */

    /* coerce back into 16-bit word size */
    acc >>= 15;

    /* check for overflow and use saturation logic */
    if (acc > NATIVE_MAX) {
        acc = NATIVE_MAX;   /* Saturate instead of overflow */
    } else if (acc < NATIVE_MIN) {
        acc = NATIVE_MIN;   /* Saturate instead of underflow */
    }

    /* Update delay line */
    dx[1] = dx[0]; /* x(n-2) = x(n-1) */
    dx[0] = input; /* x(n-1) = x(n) */
    dy[1] = dy[0]; /* y(n-2) = y(n-1) */
    dy[0] = (short) acc; /* y(n-1) = y(n) */
```

```
40
41      /* Return value */
42      return (short) acc;
43  }
```

Listing 4.6: iir_notch.c

5. Opgave 4: Test på target

Nedenstående opstilling er benyttet til test på target DSP-hardware. Analog Discovery's signalgenerator (AWG) er sluttet til line-in. Line-out er forbundet til oscilloskop. Hvide stik er AWG1 og scope CH1, som bærer højre lydkanal. Forsyning til eZDSP-kittet er med påsat ferrit for at dæmpe evt. højfrekvent støj fra bl.a. computerens strømforsyning.



Figur 5.1: Testopstilling

AWG benyttes til at afspille musiksignalet eller lave sweeps. Spektrumanalysator kan analysere frekvensindhold i output fra target. Netværksanalysator kan bruges til at lave en frekvenskarakteristik.

Line level for "consumer"-udstyr er $-10\,\mathrm{dBV}$ dvs. 316 mV RMS[1] [8]. Mic-level er typisk meget lavere, fx $-40\,\mathrm{dBV}$. For at undgå clipping (eller at brænde ADC'en i line-in af), holdes amplituden fra AWG på maks. 50 mV ($-29\,\mathrm{dBV}$)[2].

[1] $0\,\mathrm{dBV}$ er 1 V RMS. $-10\,\mathrm{dBV}$ svarer til et sinussignal med peak-amplitude 0.447 VPK.
[2] Peak-amplitude på 50 mV svarer cirka til 35 mV RMS, som er $-29\,\mathrm{dBV}$.

5.1 Test 1: Musik med sinustone og frekvenssweep

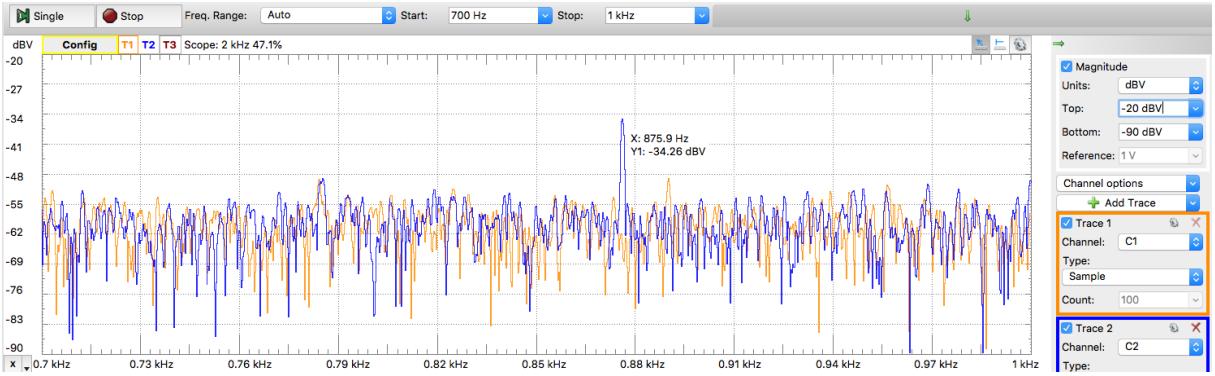
Det filtrerede musiksignal er aflyttet med en højttaler for at bekræfte, at lydkvaliteten stadig er som forventet, og at sinustonen er dæmpet. Forsøget kan ses/høres her: https://youtu.be/urbXrjlm0hs.

Et sweep fra 700-1050 Hz er også blevet forsøgt. Det auditive indtryk er, som forventet, at frekvenserne tæt omkring centerfrekvensen dæmpes. Forsøget kan ses/høres her: https://youtu.be/BOKc1OGQojs.

Baseret på test 1 konkluderes, at filteret virker.

5.2 Test 2: Musiksignal og spektrumanalysator

Musiksignalet afspilles igen fra AWG1 (50 mV peak-amplitude). Spektrumanalysatoren benyttes til måle og sammenligne frekvensindhold i hhv. et filtreret og et ikke-filtreret outputsignal for frekvensområdet 700-1000 Hz. CH1 (orange) er filtreret og CH2 (blå) er ikke-filtreret.



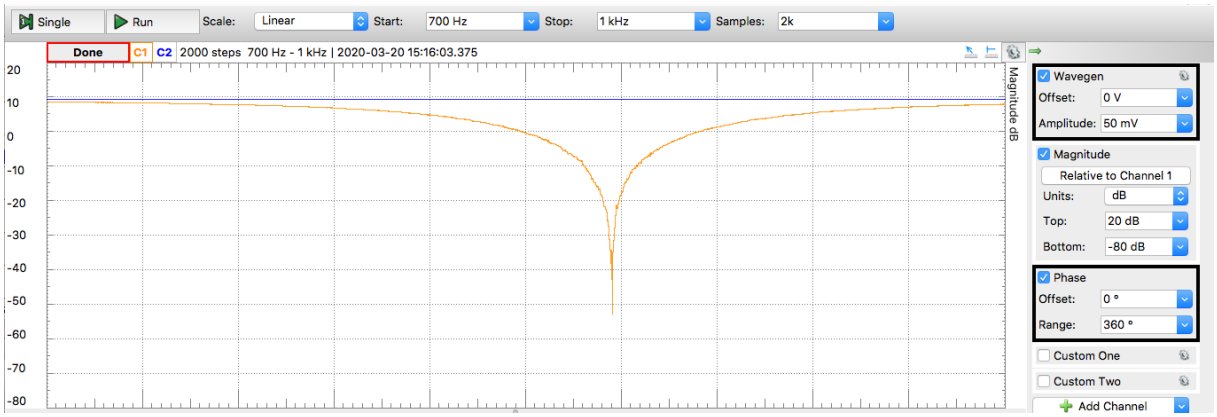
Figur 5.2: Sammenligning af filtreret og ikke-filtreret output

Figuren viser, at det ikke-filtrerede signal har en ren tone omring 876 Hz, og at tonen er dæmpet i det filtrerede signal. Som forventet :-) De to spektra er ikke sammenfaldende, bl.a. fordi det filtrerede signal er forsinket gennem filteret. Det ses også, at niveauet for den rene tone ligger lidt under det beregnede maksniveau: −34 dBV ift. −29 dBV beregnet.

5.3 Test 3: Frekvenskarakteristik

Frekvenskarakteristikken optages med Waveforms' netværksanalysator. Igen undersøges frekvensområdet 700-1000 Hz. Der optages 2000 samples over frekvensområdet, og AWG er sat op til at køre 64 perioder for hver frekvens i sweep'et.

Figuren nedenfor viser en karakteristik, der som forventet ligner spektra beregnet i MATLAB. Den maksimale dæmpning i notchet er på ca. −60 dB, hvilket er højere end de ca. −40 dB dæmpning af sinustonen, der blev observeret i MATLAB. Forklaringen er nok, at kvantisering har flyttet filterets centerfrekvens en lille smule, så sinustonens 876 Hz ikke bliver "ramt" med den fulde dæmpning.



Figur 5.3: Frekvenskarakteristik

Selvom gain i hardware er sat til 0 dB (for ADC'en), sker der en forstærkning, som er urelateret til selve filteret, hvilket kan bekræftes ved niveau for det ikke-filtrerede (blå) signal. Højst sandsynligt sker forstærkningen i DAC'en. Jeg har ikke (endnu) forsøgt at slå dette gain fra.

Baseret på frekvenskarakteristikken konkluderes, at filteret virker, men at der kunne gøres mere for at "tune" det kvantiserede filter til den ønskede centerfrekvens.

6. Opgave 5: Fri leg

Denne opgave blev der desværre ikke tid til denne gang :-(

7. Forbedringsmuligheder

- "Tuning" af det kvantiserede filter til mere præcist at ramme den ønskede centerfrekvens.
- "Optimering" af det kvantiserede filter til at være skarpere/stejlere.
- Lave en kaskade af 2. ordensfiltre til at få et mere selektivt filter (smallere stop-båndbredde og højere dæmpning i centerfrekvensen).
- Benytte et adaptivt filter til *kun* at fjerne sinustonen uden at dæmpe nogen omkringliggende frekvenser.

8. Konklusion

I denne case er der designet og implementeret et 2. ordens IIR notch-filter i MATLAB. Algoritmer og koefficienter er også udarbejdet til implementering af filteret på DSP-hardware. Hardwareimplementeringen er testet med tre forskellige metoder, og det er verificeret, at filteret virker på target, som ønsket. En række forbedringsmuligheder er nævnt til at få et filter med endnu bedre performance.

Det har været en interessant case - især fordi en række hensyn skulle tages til ikke-ideelle forhold under implementering på DSP-hardware. Bl.a. kvantisering til 16-bit, men også andre hardware-faktorer.

9. Kildehenvisning

[1] Richard G Lyons. *Understanding Digital Signal Processing*. 3. udg. Prentice Hall, 2010.

[2] Sen M. Kuo, Bob H. Lee og Wenshun Tian. *Real-Time Digital Signal Processing. Fundamentals, Implementations and Applications*. 3. udg. Wiley, 2013.

[3] Julius O. Smith. *Introduction to Digital Filters. Frequency Response as a Ratio of DTFSs*. 2007. URL: https://www.dsprelated.com/freebooks/filters/Frequency_Response_Ratio_DTFTs.html.

[4] Julius O. Smith. *Introduction to Digital Filters. The Four Direct Forms*. 2007. URL: https://www.dsprelated.com/freebooks/filters/Four_Direct_Forms.html.

[5] Kristian Peter Lomholdt. *TI TMS320C5535 eZDSP Gettting Started*. 2019. URL: https://blackboard.au.dk/bbcswebdav/pid-2489266-dt-content-rid-7948066_1/xid-7948066_1.

[6] Texas Instruments. *Porting C5000 Teaching ROM to C5535 eZdsp. Edit aic3204init.c*. 2017. URL: https://processors.wiki.ti.com/index.php/Porting_C5000_Teaching_ROM_to_C5535_eZdsp#Edit_aic3204_init.c.

[7] Texas Instruments. *C5000 teaching ROM. Chapter 7. IIR Filters*. 2010. URL: https://e2e.ti.com/support/archive/universityprogram/educators/w/wiki/2040.c5000-teaching-rom.

[8] Wikipedia.org. *Line Level. Nominal Level*. 2020. URL: https://en.wikipedia.org/wiki/Line_level#Nominal_levels.

10. Hjælpefunktioner

Der er til projektet implementeret en række hjælpefunktioner.

10.1 setlatexstuff

```
function [] = setlatexstuff(intpr)
% Sæt indstillinger til LaTeX layout på figurer: 'Latex' eller 'none'
% Janus Bo Andersen, 2019
    set(groot, 'defaultAxesTickLabelInterpreter',intpr);
    set(groot, 'defaultLegendInterpreter',intpr);
    set(groot, 'defaultTextInterpreter',intpr);
end
```

10.2 spectrogram0

Implementeret af Kristian Lomholdt, E4DSA. Let modificeret, Janus, feb. 2020. Baseret på Manolakis m.fl., s. 416.

```
function S=spectrogram0(x,L,Nfft,step,fs,ylims)
% Spektrogram. Beregner og viser spektrogram
% Baseret på: Manolakis & Ingle, Applied Digital Signal Processing,
%             Cambridge University Press 2011, Figure 7.34 p. 416
% Parametre:  x:    inputsignal
%             L:    vinduesbredde ("segmentlængde")
%             Nfft: DFT størrelse. Der zeropaddes hvis Nfft>L
%             step: stepstørrelse
%             fs:   samplingsfrekvens
% Forlkaring:
% x  |-------------------------------------------------------------------|
%    |----------|                                                      N-1
%               L-1
%          |----------|
%        step
%
% KPL 2019-01-30

% transpose if row vector
    if isrow(x); x = x'; end

    N = length(x);
    K = fix((N-L+step)/step);
    w = hanning(L);
```

```
    time = (1:L)';
    Ts = 1/fs;
    N2 = Nfft/2+1;
    S = zeros(K,N2);
    for k=1:K
        xw = x(time).*w;
        X  = fft(xw,Nfft);
        X1 = X(1:N2)';
        S(k,1:N2) = X1.*conj(X1); % samme som |X1|^2 - effektspektrum
        time = time+step;
    end
    S = fliplr(S)';
    S = S/max(max(S)); % normalisering



    tk = (0:K-1)'*step*Ts;
    F = (0:Nfft/2)'*fs/Nfft;

    colormap(jet);     % farveskema, prøv også jet, summer, gray, ...
    imagesc(tk,flipud(F),20*log10(S),[-100 10]);

    axis xy
    ylabel('$f$ [Hz]', 'Interpreter','Latex', 'FontSize', 12);
    ylim(ylims);

    xlabel('$t$ [s]', 'Interpreter','Latex', 'FontSize', 12);

    title(['Spektrogram', newline, ...
            '$N_{FFT}=$' num2str(Nfft) ...
            ', $L=$' num2str(L) ...
            ', step=' num2str(step) ...
            ', $f_s=$' num2str(fs)], ...
            'Interpreter', 'Latex', 'FontSize', 14)
end
```

10.3 smoothMag

KPL E3DSB

```
function Y = smoothMag(X,M)
% Smoothing of signal. Eg. frequency magnitude spectrum.
% X must be a row vector, and M must be odd.
% KPL 2016-09-19
    N=length(X);
    K=(M-1)/2;
    Xz=[zeros(1,K) X zeros(1,K)];
```

```
8       Yz=zeros(1,2*K+N);
9       for n=1+K:N+K
10          Yz(n)=mean(Xz(n-K:n+K));
11      end
12      Y=Yz(K+1:N+K);
13  end
```