



TTT4280 – Lab

Tittel: Laboppgave 1 - Systemoppsett

Forfattere: Even Finnøy og Lisa Radford

Gruppe: 13

Dato: 10. februar 2026

Sammendrag

Denne rapporten undersøker oppsett og verifikasjon av en samplingskjede for analoge sensorsignaler, med Raspberry Pi 3 som måleplattform, tre eksterne MCP3201 ADC-er og et LC-lavpassfilter for filtrering av 3.3V-forsyningen. Det systemteoretiske grunnlaget for sampling, kvantisering, DMA og lavpassfiltrering presenteres, og filterets knekkfrekvens og demping analyseres både teoretisk og eksperimentelt. Systemet testes ved innsampling av sinusformede inngangssignaler med DC-offset, og resultatene evalueres i frekvensdomen ved FFT med vindusfunksjoner. Støynivået i systemet vurderes ved grafisk estimering av signal-til-støy-forholdet, definert som avstanden i dB mellom spektraltoppen ved inngangssignalets frekvens og nærmeste støytopp i frekvensspekteret. Sammenligning av spekter med og uten filter viser tydelig reduksjon av støykomponenter og overtoner over signalfrekvensen. SNR estimeres til omtrent 36.5 dB ved 1 kHz og 32.5 dB ved 50 Hz. Den målte SNR-en er lavere enn teoretisk maksverdi for en ideell 12-bits ADC, og mulige årsaker diskuteres. Samlet viser målingene at samplingskjeden fungerer etter hensikten og er egnet for videre bruk i senere laboratorieoppgaver.

Innhold

Ordliste	1
Symboler	2
1 Innledning	3
2 Systemteoretisk grunnlag	3
2.1 Analog-til-digital-konverter	3
2.2 Sampling og samplingsfrekvens	4
2.3 Direct Memory Access	4
2.4 Lavpassfilter	4
2.4.1 Knekkfrekvens	5
2.4.2 Resonans og demping	5
2.4.3 Signal-til-støy-forholdet	6
2.4.4 Frekvensanalyse ved hjelp av Diskrete og Fast Fourier Transform	6
3 Eksperimentelt metode	7
3.1 Raspberry Pi 3 og Pi Wedge	7
3.2 Lavpassfilter	8
3.3 ADC	9
4 Realisert krets	11
5 Resultater	11
5.1 Fast Fourier Transform	11
5.2 Signal to Noise Ratio	13
6 Diskusjon	14
7 Konklusjon	14
Referanser	15
A Vedlegg A	16
B Vedlegg B	16
C Vedlegg C	21

Ordliste

Forkortelse	Fullstendig form
ADC	Analog-til-digital-konverter
RPi	Raspberry Pi
SSH	Secure Shell
DMA	Direct Memory Access
CPU	Central Processing Unit
SNR	Signal-to-noise ratio
DFT	Discret Fourier Transform
FFT	Fast Fourier Transform
RPi3	Raspberry Pi 3
GPIO	General Purpose Input/Output
CS	Chip Seøect
IC	Integrated Ciruit

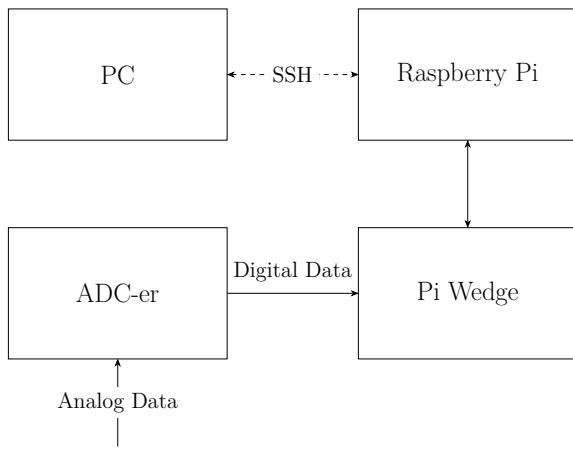
Symboler

Symbol	Forklaring	Benevning
B	Bitdybde (oppløsning i ADC)	–
b	Antall bit brukt i ADC	–
ΔV	Kvantiseringsssteg (spenningsoppløsning)	V
V_{DD}	Forsyningsspenning	V
F_S	Samplingsfrekvens	Hz
f_{\max}	Maksimal signalfrekvens	Hz
f_c	Knekkfrekvens	Hz
L	Induktans (spole)	H
C	Kapasitans (kondensator)	F
ζ	Dempningsfaktor	–
R_F	Filtermotstand	Ω
P_{signal}	Signaleffekt	W
P_{noise}	Støyeffekt	W
$x(n)$	Diskret tidsdomene-signal	–
N	Antall samples	–
$X(k)$	Diskret frekvensspekter	–
f_{klokke}	Klokkefrekvens	Hz
f_{sampler}	Samplingsfrekvens	Hz
T	Periode	s
$N_{\text{ADC_sykler}}$	Antall ADC-syklinger per konvertering	–
$f_{c,0}$	Knekkfrekvens for kanal 0	Hz
$f_{c,15}$	Knekkfrekvens for kanal 15	Hz
$f_{c,\text{teoretisk}}$	Teoretisk knekkfrekvens	Hz
V_{DD}	Digital forsyningsspenning	V
$V_{DD,F}$	Filtrert forsyningsspenning	V
CLK	Klokkesignal til ADC	–
D_{out}	Datautgang fra ADC	–
\overline{CS}	Chip Select	–
V_{ref}	Referansespenning	V
IN^+	Positiv analog inngang	V
IN^-	Negativ analog inngang	V
V_{SS}	Jord / negativ forsyning	V
$\text{SNR}_{\text{exp},1\text{kHz}}$	Eksperimentell SNR ved 1 kHz	dB
$\text{SNR}_{\text{exp},50\text{Hz}}$	Eksperimentell SNR ved 50 Hz	dB

1 Innledning

Analoge sensorsignaler må samples og digitaliseres for videre signalbehandling. Det er derfor viktig med valg av samplingsfrekvens, analog-til-digital-konvertere (ADC) sin opplosning og støynivå i systemet. Grunnet dette har direkte innvirkning på måleresultatet. Samplingskjeden må derfor settes sammen slik at signalet representeres korrekt før videre analyse.

Laboratorieøvelsen omhandler oppsett og verifikasjon av en komplett samplingskjede bestående av Raspberry Pi (RPi)[1] som måleplattform, eksterne ADC-er, et lavpassfilter og annen tilleggselektronikk, som sammen utgjør systemoppsettet. Det overordnende systemet er vist i blokkdiagrammet i figur 1.



Figur 1: Blokkskjema som viser overordnet systemarkitekturen. Analoge signaler konverteres til digitale data i ADC-ene og overføres via Pi Wedge til RPi-en. PC-en kommuniserer med RPi-en over SSH.

Blokkskjema 1 gir en oversikt over systemarkitekturen og dataflyten i systemoppsettet. ADC-ene utfører analog-til-digital-konvertering av den analoge daten som er inngangssignal, før de digitale dataene sendes videre til RPi-en gjennom Pi Wedge. RPi-en håndterer innsamling og behandling av data, mens PC-en kommuniserer med systemet via Secure Shell

(SSH). SSH er en protokoll som lar deg logge på en ekstern maskin direkte i terminalen til PC-en[2].

Som del av systemoppsettet blir det designet og implementert et enkelt lavpassfilter for å redusere støy på 3.3V spennings linjen til de analoge komponentene. Filterets frekvensrespons blir analysert både teoretisk og eksperimentelt for å verifisere korrekt funksjon, som vist i seksjon 2 og 3.

Målet med laboratorieøvelsen er å verifisere at målesystemet fungerer som forventet, og å sikre at det er egnet for videre bruk i senere laboratorieoppgaver innen akustikk, radar og optiske målinger[2].

2 Systemteoretisk grunnlag

2.1 Analog-til-digital-konverter

ADC-en benyttes for å representer kontinuerlige analoge signaler som diskrete digitale verdier. En ADC sampler inngangssignalet i tid og representerer amplituden med et endelig antall nivåer bestemt av ADC-en sin bitdybde[2]. Antall tilgjengelige nivåer i den digitale representasjonen er gitt ved likning 1.

$$B = 2^b \quad (1)$$

I likningen er b antall bit i ADC-en. Spenningsoppløsningen, ΔV , definert som minste spenningsforskjell som kan representeres og kan uttrykkes som vist i likning 2[3].

$$\Delta V = \frac{V_{DD}}{B} \quad (2)$$

V_{DD} er ADC-en sin forsyningsspenning. Oppløsningen setter en nedre grense for hvor små variasjoner i inngangssignalet som kan representeres i det digitaliserte signalet[3].

Ved ADC-prosesser vil kontinuerlige spenningsverdier $e(n)$ erstattes med diskrete verdier $e_q(n)$. Slik at $e_q(n)$ er gitt som vist i likning 3.

$$e_q(n) = e(n) + \epsilon_q(n) \quad (3)$$

Her er $\epsilon_q(n)$ kvantiseringsfeilen. Kvantisering avrunder målinger til nærmeste digitale nivå, noe som er nødvendig for lagring og behandling i datamaskiner. Derimot introduserer kvantiseringstøy og tap av nøyaktighet.

2.2 Sampling og samplingsfrekvens

Sampling innebærer at et kontinuerlig signal måles ved diskrete tidspunkter med en samplingsfrekvens, f_s . Samplingsfrekvensen bestemmer hvor ofte ADC-en utfører en måling, og er en sentral parameter i digitale målesystemer[3].

For å unngå aliasing må samplingsfrekvensen være større enn det dobbelte av signalets høyeste frekvenskomponent, kjent som Nyquist-kriteriet som vist likning 4.

$$f_s > 2f_{\max} \quad (4)$$

f_{\max} er den maksimale frekvensen. Dersom dette kriteriet ikke er oppfylt, vil høyfrekvente komponenter foldes ned i frekvensspekteret og forvrenge måledataene[4].

2.3 Direct Memory Access

Ved kontinuerlig sampling genereres store mengder data som må overføres effektivt til minne. Direct Memory Access (DMA) benyttes for å muliggjøre direkte dataoverføring mellom komponentene ekstern maskinvare og minne uten kontinuerlig involvering av central processing unites (CPU).

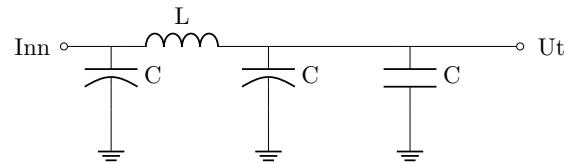
Bruk av DMA er særlig relevant i systemer som kjører et ikke-sanntidssystem, der pro-

sessplanlegging kan føre til ulike tidsforsinkelser. DMA bidrar til mer forutsigbar sampling og reduserer risikoen for tap av samples ved høye samplingsrater[3].

2.4 Lavpassfilter

For å redusere høyfrekvent støy i analoge målesystemer benyttes lavpassfiltrering. I det aktuelle målesystemet er det særlig ønskelig å dempe støy på 3.3V forsyning fra RPi-en, siden slik støy kan påvirke ADC-en sine målinger. Det brukes derfor et passivt LC-lavpassfilter som demper frekvenskomponenter over filterets knekkfrekvens og gir mer stabile signalforhold inn mot ADC-en.

Filteret som er vist i figur 2 er et LC-lavpassfilter der en spole er plassert i serie mellom inngang og utgang, mens flere kondensatorer er koblet til jord.



Figur 2: Prinsippskisse av et passivt lavpassfilter. En induktans i serie mellom inngang og utgang, kombinert med kondensatorer til jord, danner et andreordens lavpassfilter som demper høyfrekvente komponenter og reduserer støy i signalet.

Induktansen gir økende impedans med frekvens og demper raske endringer i signalet, mens kondensatorene kortslutter høyfrekvente komponenter til jord. Denne kombinasjonen gir et andreordens lavpassfilter som reduserer høyfrekvent støy før signalet videreføres til resten av systemet.

2.4.1 Knekkfrekvens

Knekkfrekvensen til et ideelt LC-lavpassfilter kan uttrykkes som vist i likning 5

$$f_c = \frac{1}{2\pi\sqrt{LC}}. \quad (5)$$

Uttrykket gjelder for et ideelt LC-filter uten tap. I praksis vil demping i komponentene påvirke frekvensresponsen og forskyve den faktiske knekkfrekvensen. For å ta hensyn til dette kan knekkfrekvensen bestemmes direkte fra frekvensresponsen til filteret. Overføringsfunksjonen for filteret i figur 2 kan skrives som vist i linkning 6.

$$H(\omega) = \frac{1}{1 + j\omega RC - \omega^2 LC}. \quad (6)$$

Grensefrekvensen, også kjent som cutoff frequency, defineres som vist i likning 7.

$$\begin{aligned} |H(\omega_c)| &= \frac{1}{\sqrt{2}} \\ (1 - \omega_c^2 LC)^2 + (\omega_c RC)^2 &= 2. \end{aligned} \quad (7)$$

Dette resulterer i følgende uttrykk, likning 8 for knekkfrekvensen. For oversiktens skyld introduseres her variablen K .

$$\begin{aligned} K &= \sqrt{\frac{2LC - R^2C^2 + \sqrt{(R^2C^2 - 2LC)^2 + 4L^2C^2}}{2L^2C^2}} \\ f_c &= \frac{1}{2\pi} K \end{aligned} \quad (8)$$

Uttrykket viser at demping i filteret fører til en forskyvning av knekkfrekvensen sammenlignet med idealtilfellet, likning 5. Ved lav demping nærmer resultatet seg uttrykket i likning 5.

2.4.2 Resonans og demping

Et LC-filter er et andreordens system og kan få en resonanstopp rundt egenfrekvensen der-

som dempingen er lav. Dette kan gi oversving i tidsdomenet og mulig en topp i frekvensresponsen. Slike effekter er uønsket i målesystemer. Dette grunnet at de kan forårsake støy og gi ustabile signalforhold.

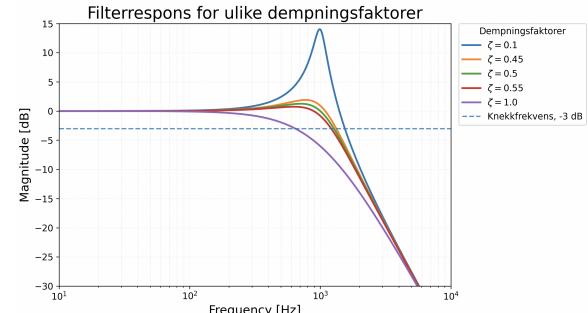
For å kontrollere denne responsen innføres en dempmotstand. Demping beskrives ved den dimensjonsløse dempningsfaktoren ζ , som angir forholdet mellom faktisk demping og kritisk demping i et andreordens system. Økende ζ gir redusert resonans.

For en standard andreordens modell kan en ønsket demping realiseres ved å velge en effektiv seriemotstand R_F slik som i likning 9.

$$R_F = 2\zeta L \cdot \left(\frac{1}{\sqrt{LC}} \right) \quad (9)$$

Her er R_F den effektive dempmotstanden. Figur 3 viser frekvensresponsen til et andreordens lavpassfilter for ulike verdier av ζ , der formel 10 blir brukt [5].

$$H = \frac{1}{\sqrt{(1 - x^2)^2 + (2\zeta x)^2}} \quad (10)$$



Figur 3: Frekvensrespons for et andreordens lavpassfilter for ulike verdier av dempningsfaktoren ζ .

Ved lav ζ oppstår en tydelig resonanstopp nær knekkfrekvensen, som kan forsterke uønsket støy i et smalt frekvensområde og gi ringing i

tidsdomenet. Når ζ økes, reduseres resonans toppen og responsen blir mer monoton avtagende. Et vanlig valg er å benytte en dempningsfaktor som gir begrenset resonans og en jevn overgang mellom passbånd og stopbånd.

I dette arbeidet er formålet med lavpassfilteret å dempe høyfrekvent støy på 3.3V forsyningen fra RPien, slik at støy ikke forplanter seg videre inn i målekjeden og påvirker ADC-målingene. Det er derfor ønskelig med en stabil filterrespons uten resonant forsterkning.

2.4.3 Signal-til-støy-forholdet

Signal-til-støy-forholdet (SNR) er et mål på forholdet mellom ønsket signal og tilstedevarrende støy i et målesystem[3]. SNR er avgjørende for systemets ytelse og målenøyaktighet, og uttrykker hvor godt signalet kan skilles fra støy. For en ideell ADC domineres støybidraget av kvantiseringssstøy, som følger av at signalets amplitude representeres med et endelig antall nivåer.

Den teoretiske maksimale SNR-en kan uttrykkes som vist i likning 11[3].

$$\begin{aligned} \text{SNR}_{\max} &= 10 \log_{10} \left(\frac{12(B \cdot \Delta V)^2}{8 \cdot \Delta V^2} \right) \\ &\approx 1.76 + 6.02 \cdot b[\text{dB}] \\ &= 74 \end{aligned} \quad (11)$$

Uttrykket forutsetter et fullskala sinusformet inngangssignal og uniform kvantisering. Her er b antall bit i ADC-en, og ΔV kvantiseringsteget. Forenklingen av uttrykket gir den velkjente tilnærmingen der SNR øker med omtrent 6.02 dB per ekstra bit, med et konstant ledd på 1.76 dB som følger av sinusformet signal.

Et høyt SNR indikerer at signalet dominerer over støyen, mens et lavt SNR innebærer at støyen har betydelig innvirkning på de målte verdiene. I et ideelt system begrenses SNR

av fundamentale støykilder, som termisk støy i passive komponenter og kvantiseringssstøy i ADC-en[3]. Teoretisk SNR gir dermed en øvre grense for hvor god signalgjengivelse som kan oppnås.

Bruken av lavpassfilter i målekjeden påvirker det effektive SNR ved å redusere støybidraget. Et godt dimensjonert filter vil dempe høyfrekvent støy uten å forsterke signalet rundt knekkfrekvensen og bidra til å øke SNR.

2.4.4 Frekvensanalyse ved hjelp av Discrete og Fast Fourier Transform

Frekvensanalyse av samplede signaler benyttes for å undersøke signalets innhold og for å identifisere støy og uønskede frekvenskomponenter. I praksis utføres denne analysen ved hjelp av den diskrete Fouriertransformen (DFT), som gir en representasjon av signalet i frekvensdomenet[3]. For et diskret signal $x[n]$ med N samples er DFT definert som vist i likning 12.

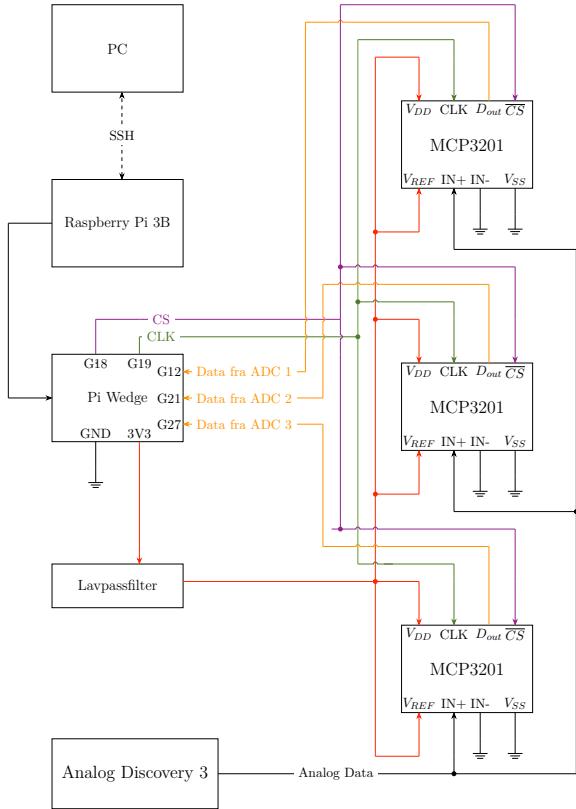
$$X[k] = \sum_{n=0}^{N-1} x[n] e^{-j2\pi kn/N}, \quad (12)$$

$X[k]$ beskriver amplituden og fasen til frekvenskomponentene i signalet. DFT gir dermed et grunnlag for å analysere hvordan energi er fordelt over frekvens.

Fast Fourier Transform (FFT) er en algoritme for å beregne den diskrete DFT når antall samples N er et helt multiplum av to, typisk $N \in [128, 256, 512, 1024, \dots]$. FFT gir samme resultat som DFT, men med vesentlig lavere beregningskompleksitet, og benyttes derfor i praktiske implementasjoner[3].

3 Eksperimentelt metode

Figur 1 i seksjon 1 viser som nevnt et overordnet blokkdiagram av systemoppsettet. Et mer detaljert blokkdiagram, som viser tydeligere overgangen mellom analog og digital daga er vist i figur 9.



Figur 4: Figuren viser et mer detaljert blokkdiagram basert på systemoppsettet i figur 1. Den analoge dataen blir sendt fra en Analog Discovery 3 inn til ADC-ene. Fra Pi Wedgen til ADC-ene er det nå udypt hvilke utganger og innganger som er koblet til hverandre. Det er her den digitale dataen overføres. Detaljer på ADC-ene er spesifisert i seksjon 3.3

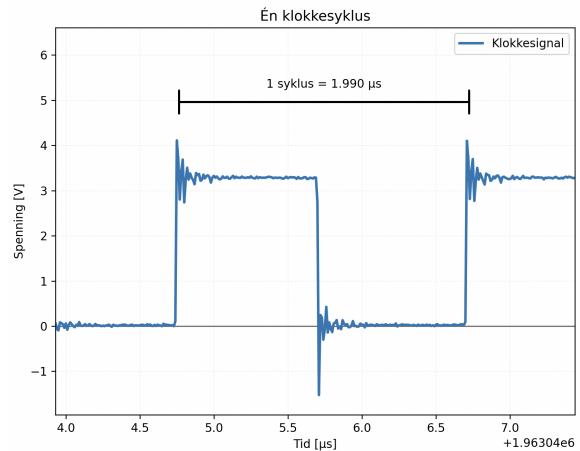
3.1 Raspberry Pi 3 og Pi Wedge

En Raspberry Pi 3 (RPi3) brukes for å lese av fra ADC-ene og sende dataene til en PC for videre behandling. RPi3-en har en 40-pins General Purpose Input/Output (GPIO) konnek-

tor, som brukes for å koble på eksterne systemer. En Pi Wedge blir brukt for å gjøre koblingen fra den 40-pins konnektoren til et brøbbrett mye lettere[6].

For å lese av ADC-ene, blir koden fra vedlegg 1 brukt. Denne leser av dataene fra hver ADC, og skriver den til en binær-fil. Den tar også inn en variabel for antall sampler som blir målt. For å konvertere denne binære-dataen til data som er mulig å behandle, brukes koden fra vedlegg 2.

Klokkefrekvensen er viktig, grunnet klokken avgjør hvor ofte ADC-ene kan konvertere og overføre det analoge signalet. Hvor mange klokkesykler som trengs for dette blir utdypet i seksjon 3.3. Klokkefrekvensen ble målt direkte fra RPi3-en, og er plottet i figur 5.



Figur 5: Målt klokkesignal fra RPi3. Figuren viser en klokkesyklus, med periode $T = 1.990\mu s$, bestemt som gjennomsnittet av fire påfølgende klokkesykler.

I blått er klokken fra RPi3-en målt og plottet. Her er det markert en klokkesykel på $1.990\mu s$, som er gjennomsnittet over fire klokkesykler. Basert på denne perioden kan klokkefrekvensen bestemmes som den inverse av periodetiden, f_{klokke} , slik det er vist i likning 13.

$$\begin{aligned} f_{\text{klokke}} &= \frac{1}{1 \text{ syklus}} \\ &= \frac{1}{1.990 \mu s} \\ &\approx 502.5 \text{ kHz} \end{aligned} \quad (13)$$

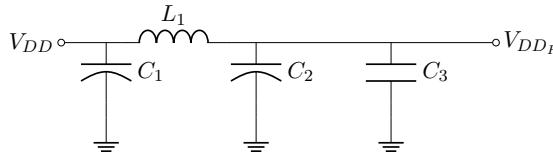
ADC-ene vi bruker trenger 16 klokkesykler, $N_{\text{ADC}_\text{sykler}}$, for å ta en sample og sende den til over til RPi3-en [7]. Mer om dette under seksjon 3.3. Ved å dele RPi3-en sin klokkefrekvens på antall sykler som trengs for én sample, finner man antall sampler ADC-ene kan ta per sekund. Ut i fra f_{klokke} på 502.5 vil antall samples fra ADC-en være gitt ved likning 14.

$$\begin{aligned} f_{\text{sampling}} &= \frac{f_{\text{klokke}}}{N_{\text{ADC}_\text{sykler}}} \\ &= \frac{502.2 \text{ kHz}}{16 \text{ sykler}} \\ &\approx 31407 \text{ sampler/s} \end{aligned} \quad (14)$$

ADC-ene kan dermed ta ≈ 31407 sampler per sekund. Som nevnt har koden fra vedlegg 1 en parameter for antall sampler. I dette arbeidet har det dermed blitt brukt f_{sampling} for å sikre oss ≈ 1 sekund med data.

3.2 Lavpassfilter

Som nevnt i seksjon 2.4, så skal 3.3V forsyningsspenningen fra RPi3-en filtreres. Figur 6 viser lavpassfilteret.



Figur 6: Et lavpassfilter med to polare og en upolær kondensator, samt en spole. Dette skaper et andre ordesns filter.

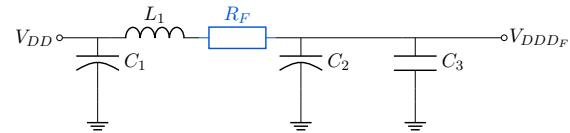
De ulike verdiene for kondensatorene og spolen er vist i tabell 3.

Tabell 3: Verdier til variablene i lavpasfilteret illustrert i figur 6

Variabel	Verdi	Benevning
C_1	100	μF
C_2	470	μF
C_3	1	nF
L_1	100	mH

$V_{DD} = 3.3V$ blir gitt fra RPi3-en, og deretter filtrert av lavpassfilteret. Denne filtreringen fjerner uønskede frekvenser over den bestemte knekkfrekvensen. Det filerte signalet, V_{DDF} , blir brukt til å gi spenning til ADC-ene.

Som nevnt i seksjon 2.4.2, så er det ønskelig med minst mulig resonans topp i filtret. Filteret som vist i figur 6 blir derfor modifisert ved å tilføre motstanden R_f , som skal dempe amplituderesponsen til filteret. Dette er illustrert i figur 7 som den blå motstanden.



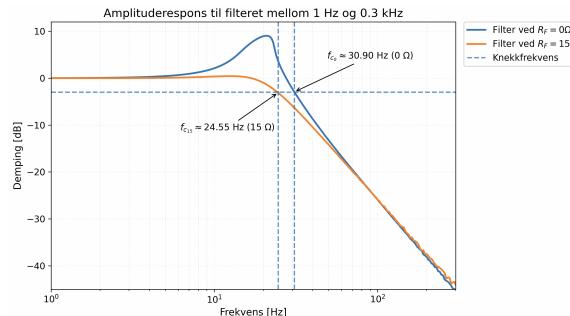
Figur 7: Lavpassfilteret med en motstand R_f for å minimere resonanse toppen til filtret. R_f er markert i blått.

For å finne tallverdien til R_f , må først en verdi for ζ bestemmes. Ut i fra grafen vist i figur 3, blir dempningsfaktoren satt til $\zeta = 0.5$. Grunnet den gir et relativt flatt filter med veldig lav resonans topp. For å finne R_f , brukes likning 9.

$$\begin{aligned} R_f &\approx 2 \cdot 0.5 \cdot 100 \text{ mH} \cdot \left(\frac{1}{\sqrt{100 \text{ mH} \cdot 470 \mu F}} \right) \\ &\approx 14.59 \Omega \\ &\approx 15 \Omega \end{aligned} \quad (15)$$

Fra formel 15, får vi at motstanden $R_F \approx 15\Omega$. I utrykket over blir kun C_2 av kondensatorene tatt med i beregningen. Dette kommer av at $C_2 \gg C_3$, og at C_1 ikke inngår i systemfunksjonen[8].

Med motstanden $R_F = 15\Omega$ fikk vi et kritisk dempet filter, med tilnærmet ingen resonans topp. I figur 8 under, blir det vist amplituderesponsen til filteret med og uten motstanden R_F .

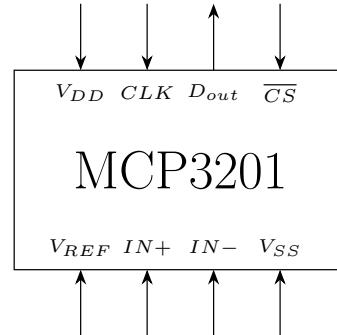


Figur 8: Amplituderespons for det målte lavpassfilteret med og uten motstanden R_F . Responsen uten motstand er vist i blått, mens responsen med $R_F = 15\Omega$ er vist i oransje. De stiplete linjene markerer -3dB -nivået og tilhørende knekkfrekvenser.

I blå har vi amplituderesponsen uten en motstand, mens i oransje har R_F en verdi på 15Ω . De lyseblå stipletlinjene viser til knekkfrekvensen. Den horisontale markerer -3dB -linjen, og de to vertikale viser til knekkfrekvensen til de to filterene. Som markert i plottet, så er knekkfrekvensen uten en motstand på $f_{c0} \approx 30.90 \text{ Hz}$, mens med motstand er $f_{c15Ω} \approx 24.55 \text{ Hz}$. Ved bruk av formel 8, blir den teoretiske knekkfrekvensen på $f_{c\text{teoretisk}} \approx 29.15 \text{ Hz}$. f_{c0} er nærmere $f_{c\text{teoretisk}}$, men $R_F = 15\Omega$ ble ikkevel inkludert i kretsen. Dette ble gjort til fordel for en flatere amplituderespons til filteret, i stedet for en treffsikker knekkfrekvens.

3.3 ADC

ADC-ene som brukes er av typen MCP3201, og har en oppløsning $b = 12$ bits[7]. Figur 9 viser pinout-en til denne komponenten.

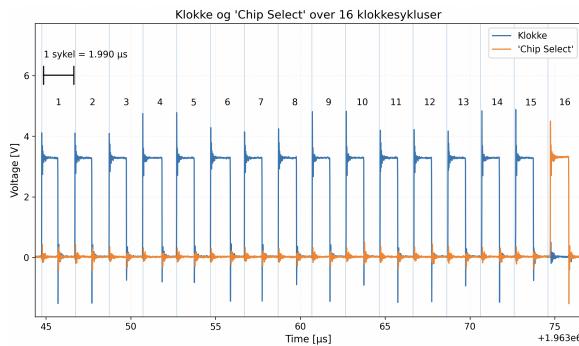


Figur 9: Pinout-diagram for ADC-en MCP3201.

De ulike pinsene har forskjellige funksjoner. V_{DD} og V_{REF} kobles direkte på det filtrerte $3.3V$ signalet, V_{DD_F} . Pin-en V_{DD} supplerer ADC-en med strøm, mens V_{REF} definerer hvilket spenningsnivå som samsvarer med maksimal digital output. Altså at $3.3V = 2^b = 4096$ fra likning 1.

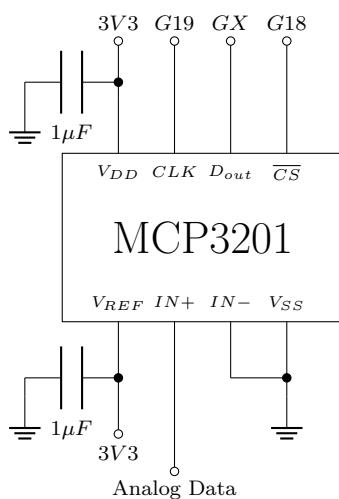
Videre har vi $\text{IN}+$ og $\text{IN}-$. Dette er pinsene som leser av det analoge signalet. Vi kobler $\text{IN}-$ til jord, sammen med V_{SS} som er jordingspinen til ADC-en. Videre har vi \overline{CS} , som står for Chip Select (CS) og blir ofte kallet enable. Dette er et signal som skrur ADC-en på dersom signalet er lavt og omvendt. CLK er klokkesingalet fra RPi3-en, som ble nevnt i seksjon 3.1. D_{out} er den digitale avlesningen som sendes til RPi3-en.

Fra databladet[7] til ADC-ene, blir det oppgitt at det krever 16 klokkesyklar for å ta en punktprøve og overføre den til RPi3-en. Ifølge databladet[7] trenger ADC-en 15 sykler fra klokken, og en fra CS. Ved å måle klokkesyklen til ADC-ene, som vist i figur 10, kan den forventede oppførselen bekreftes.



Figur 10: Målt klokkesignal fra ADC-ene. I blått er de 16 klokkesyklene per CS. CS er markert i oransje.

Som plottet i figur 10 illustrerer, fungerer klokken og CS slik som foreventet. Dette grunnet det er målt 15 klokkesyklar per CS. Et viktig poeng her er at internklokken på RPi3-en går hele tiden, men ADC-en trenger ikke klokken i sykel 16 og dermed blir den ikke sendt til ADC-en heller. Derfor ser det ut som klokken stopper under CS-signalet. Videre står det i databladet[7] at denne ADC-en trenger avkoblingskondensatorer. Disse er til for å stabilisere spenningen, samt redusere støy. I kretsen som skal implementeres, blir det brukt to av disse avkoblingskondensatorene. Figur 11 viser oppkopplingen av en av disse ADC-ene med avkoblingskondensatorer.

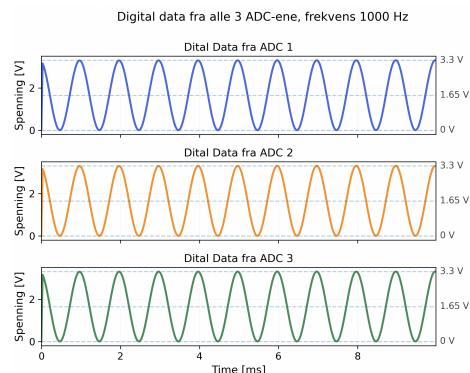


Figur 11: MCP3201 med to avkoblingskondensatorer på $1\mu F$.

Tre ADC-er krever tre datalinjer til RPi3-en, pin GX, der X står for enten 12, 22 eller 27. Disse viser dermed til GPIO-pinsene G12, G22 eller G27. Figur 13 illustrerer den digitale kommunikasjonen mellom Raspberry Pi 3 og Pi Wedge.

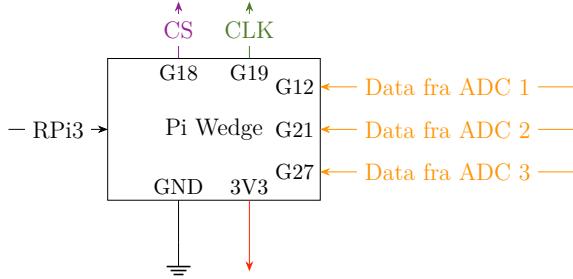
CLK og CS genereres av Raspberry Pi og benyttes til å styre ADC-ene, mens hver ADC har en egen datalinje tilbake til Raspberry Pi. Denne løsningen muliggjør parallel avlesning av flere ADC-kanaler ved bruk av felles klokke- og styresignaler.

ADC-ene ble også testet for å forsikre forventet resultat. Siden IN- ble jordet, betyr det av ADC-ene ikke kan måle spenninger under 0V. Derfor blir et $1kHz$ sinussignal på $1.65V$ sendt inn på IN+-pinen, med en DC-offset på $1.65V$. Ut i fra Dette gjør at ADC-ene får et signal mellom 0 og $3.3V$. Figur 14 viser en avlesning fra ADC-ene av dette signalet.



Figur 14: Plott av de digitale signalene fra de tre ADC-ene. I blått er ADC1 sitt digitale sinussignal. I oransje er det ADC2 og i grønt er det ADC3.

Figuren viser et pent, glatt sinussignal fra hver av de forskjellige ADC-ene. I figuren over er minimumspenningen og maksimalspenningen 0 og $3.3V$, samt offsetten på $1.65V$ markert på høyre side. Figuren viser dermed at et analogt signal kan bli sendt inn på inngangene på ADC-ene, og at et pent, tilsvarende digitalt signal blir sendt ut.

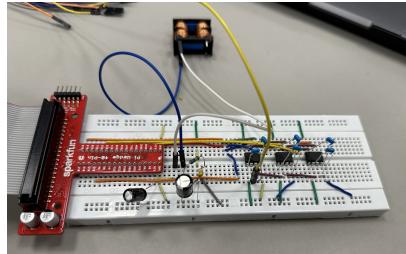


Figur 13: Pi Wedge med relevante pins.

4 Realisert krets

Det totale systemet er vist i figur 12, med lavpassfiltret i blått og ADC-ene slik som i blokkdiagrammet i figur 9.

Systemoppsettet viser hvordan alle komponentene skal kobles sammen. Videre skal dette realiseres på et brødbrett. Dette er fremstilt i figur 15.



Figur 15: Den realiserte kretsen fra figur 12 koblet opp på et brødbrett.

I figuren er den røde T-en til venstre er Pi Wedgen, og spolen som ligger på bordet er spolen i figur 7. De tre IC-ene til høyre er ADC-ene.

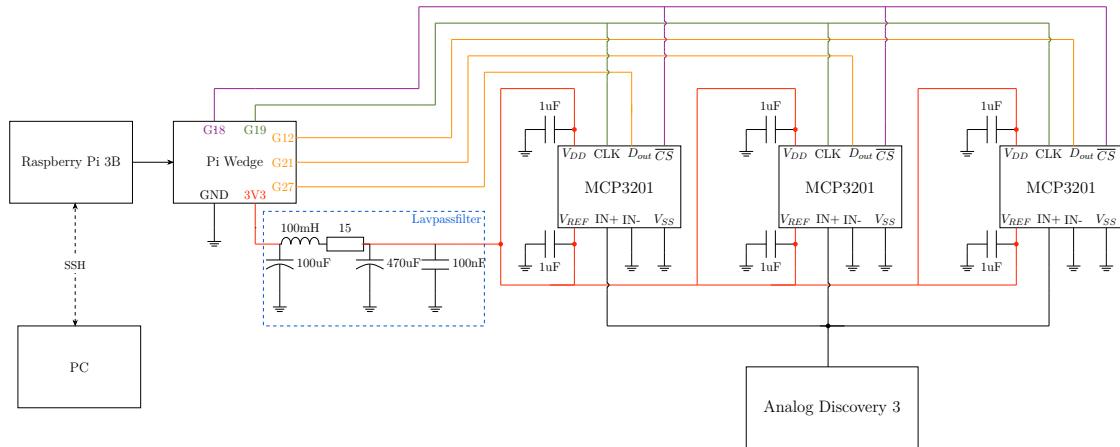
5 Resultater

Ut i fra seksjon 3 og 4 oppfører systemet seg som forventet. Dermed kan diverse analyser bli gjennomført på signalene.

5.1 Fast Fourier Transform

Som nevnt i seksjon 2.4.4, blir overtoner svært lette å identifisere dersom det blir gjennomført en FFT av et signal. FFT visualieres ved bruk av vindusfunksjoner. Poenget med vindusfunksjoner er å redusere amplituden til diskontinuitetene ved grensene til hver endelige sekvens som er tatt opp av ADC-en[9]. Figur 16 viser noen ulike vindusfunksjonene.

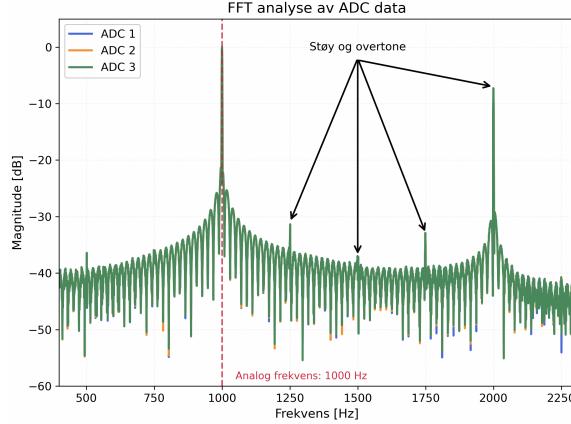
Som illustrert av figuren over, så er det lett å identifisere støy-toppene. Dermed er dette svært hjelpsomt for å finne ut blant annet om filteret som ble laget under seksjon 3.2 har ønsket effekt på dataene ut av ADC-ene. I figur 17 under blir dataen fremstilt ved vindusfunksjonen "Hamming", zoomet inn til første



Figur 12: Det fullstendige systemoppsettet. Lavpassfilteret markert i blått.

Resultater

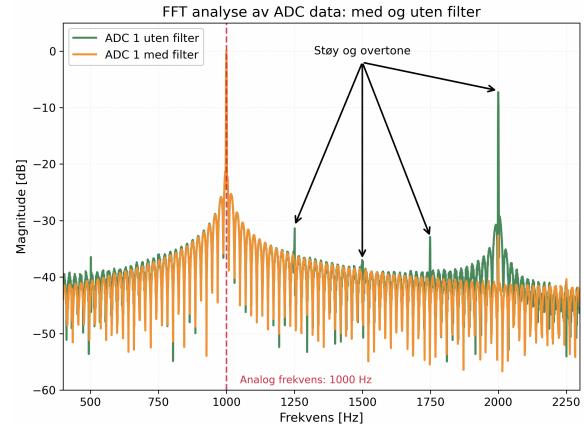
overtone.



Figur 17: FFT-analyse av ADC-data for alle tre ADC-kanalene uten lavpassfilter. Den stiplete linjen markerer den analoge signalfrekvensen på 1000 Hz.

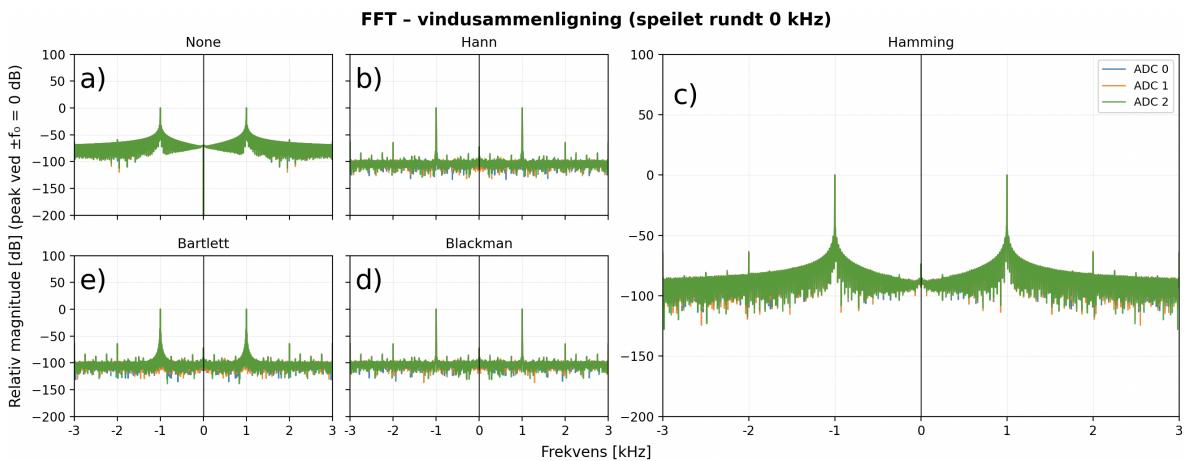
Her er frekvensen til det analoge signalet, 1000 Hz, markert av den røde stipletlinjen. Ved denne frekvensen, er det en veldig stor topp. Det er som forventet, siden dette er signalet som blir sendt inn til ADC-ene. Pilene peker på tre støy-topper og en overtone ved ≈ 2000 Hz. Det er spesielt disse toppene som filteret skal redusere. Ved å plotta en FFT-analyse av frekvensspekteret til ADC-

ene med og uten filteret, blir det fort synlig dersom filteret oppfører seg som forventet. Dette er gjort i figur 18.



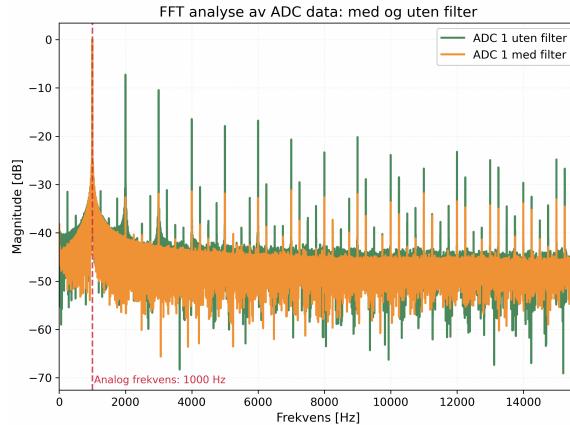
Figur 18: FFT-analyse av ADC-data for ADC 1 med og uten lavpassfilter. Sammenligningen viser at lavpassfilteret reduserer støy nivå og overtoner over signalfrekvensen på 1000 Hz.

Figuren over viser en sterkt reduksjon i både støy og overtoner, illustrert av de samme pilene fra figur 17. Dette er spesielt synlig dersom plottet er over en større del av frekvensspekteret, siden flere overtoner og mer støy kommer inn i bildet. Dette er gjort i figur 19.



Figur 16: FFT-sammenligning av samme signal ved bruk av ulike vindusfunksjoner. Spekteret er speilet rundt 0Hz og normalisert til 0dB ved f_0 . Figur a) viser FFT uten vindu, med tydelig spektrallekkasje og høye sidelober. Figur b)–e) viser henholdsvis Hann-, Hamming-, Blackman- og Bartlett-vindu.

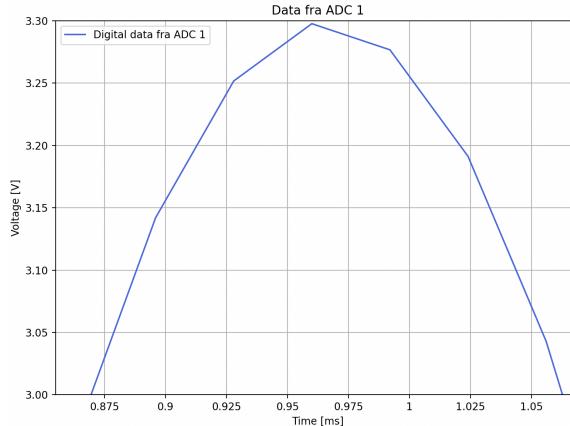
Resultater



Figur 19: FFT-analyse av ADC-data med og uten lavpassfilter.

Som figuren viser, er alle grønne topper, unntatt ved 1000 Hz, redusert.

Hvor disse støykomponentene kommer fra har mange potensielle kilder. En av de, som nevnt i seksjon 2.1, er kvantiseringstøy. Dersom figur 14 blir zoomet inn på, ser man at det ikke er et kontinuerlig signal man får ut av ADC-en. Figur 20 viser dette.



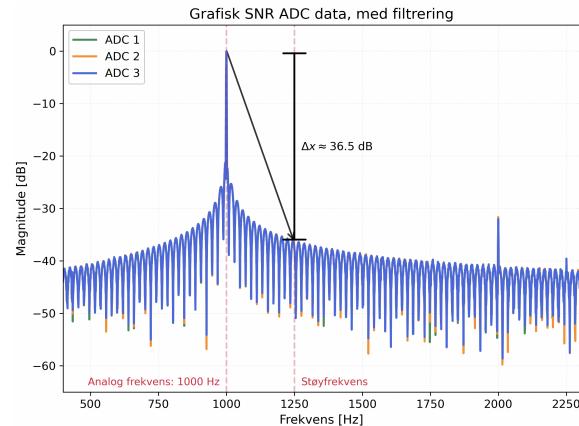
Figur 20: Tidsdomenevisning av digitalt signal fra ADC 1. Figuren illustrerer at utgangssignalet fra ADC-en er kvantisert og ikke kontinuerlig, som følge av begrenset oppløsning.

Figuren over er fra ADC 1, zoomet inn på en amplitudetopp. Den viser at det er en kant mellom hver måling fra ADC-ene, som fører

til kvantiseringstøyet.

5.2 Signal to Noise Ratio

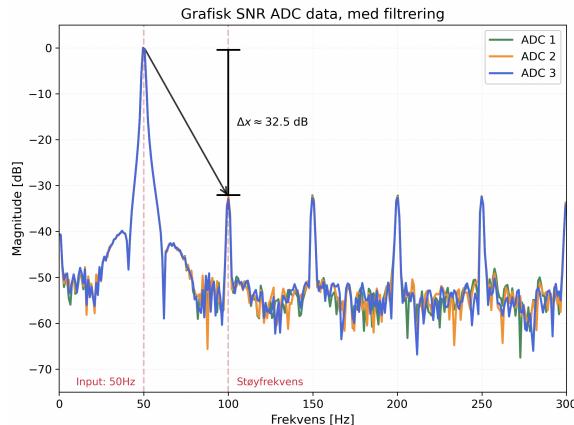
Videre skal systemet evalueres med hensyn på støy. Det blir dermed brukt SNR. I denne sammenhengen bruker vi grafisk SNR. I figur 18 brukte vi en oransje grafen. Ved bruk av Grafisk SNR, blir den oransje grafen i figur 18 brukt. Avstanden på y-aksen, i dB, mellom toppen på inngangssignalet ved 1000Hz og neste støy-topp. I figur 18 vil dette være mellom toppen på 1000Hz og neste topp ved 1250Hz. Dette er illustrert i figur 21



Figur 21: Grafisk SNR-beregning basert på FFT av ADC-data med filtrering.

Fra figuren over får vi at $SNR_{exp_{1\text{kHz}}} \approx 36.5 \text{ dB}$.

Det er også mulig å sende inn et signal med en annen frekvens for å se om SNR-en blir den samme. I figur 22 blir det samme signalet sent ut, men at frekvensen nå er på 50Hz.



Figur 22: Grafisk SNR-beregning basert på FFT av ADC-data med filtrering for et inngangssignal på 50 Hz.

Denne gir oss et litt lavere estimat

$$SNR_{exp_{50Hz}} \approx 32.5dB.$$

6 Diskusjon

Fra seksjon 3 ble hvert system verifisert hver for seg. Filteret fra seksjon 3.2 hadde en amplituderespons som var som forventet, og ADC-ene fra seksjon 3.3 hadde et pent digitalt signal ut. Likevel er avviket mellom teoretisk best case $SNR_{max} = 74dB$ og den målte $SNR_{exp_{1kHz}} \approx 36.5dB$. Detter har flere potensielle årsaker. Som nevnt i seksjon 2.1 og 5.2, så bidrar kvantiseringen til støy. Hadde ADC-en som ble brukt hatt en høyere bitdybde b , så hadde denne støykomponenten blitt mindre, siden ADC-en hadde hatt flere nivåer å velge mellom.

Videre ble det brukt relativt billige komponenter til å realisere systemet. Dette har mest sannsynlig introdusert store mengder støy, siden tallverdiene til disse komponentene ikke nødvendigvis stemmer med de teoretiske verdiene som er antatt. Selve ADC-en selv kan være unøyaktig på selve avlesningen, og dermed introdusere støykomponenter som egentlig ikke er til stede på analog-inngangen.

En annen feilkilde kan være brødbrettet. Det som ble brukt er ikke av høyeste kvalitet, og dermed kan påvirke signalene. I tillegg til feilkilder ved selve brødbrettet, ble det brukt ledninger for å koble forskjellige deler til hverandre. Dette er støymessig uheldig, siden lange ledninger fungerer som antenner. Dermed er der en høy sjans for at ledningene plukket opp elektromagnetisk støy fra området rundt der eksperimentet ble gjennomført.

Underveis gjennom dette prosjektet ble det arbeidet med et kretskort, som kretsen i denne rapporten skulle designes på. På grunn av for lite tid, ble ikke den ferdig til dette prosjektet. Å komprimere kretsen, og sørge for bedre isolering og oppkoppling, ville mest sannsynlig ført til en betraktlig lavere SNR.

7 Konklusjon

Gjennom denne rapporten ble det designet et system bestående av en Raspberry Pi 3, et filter og 3 Analog-til-digital konvertere (ADC-er) for å lage en avlesningskrets til bruk i senere labber. Gjennom tester ble det bekreftet at hver del fungerer godt hver for seg, men gav et noe høyere signal-til-støy forhold enn forventet. Dette kan komme fra flere kilder, blant annet kvantisering, billige komponenter og lange ledninger.

Likevel er signalet ut fra ADC-ene godt nok til at denne kretsen kan bli brukt til kommende labber.

Referanser

- [1] Raspberry Pi Ltd.: *Raspberry Pi 3 Model B*. <https://www.raspberrypi.com/products/raspberry-pi-3-model-b/>, 2023. Tilgjengelig: februar 2026.
- [2] *TTT4280 Sensors and Instrumentation – Laboratory Manual*, 2026.
- [3] Svensson, U. Peter, Egil Eide og Lise Lyngnes Randeberg: *Sensorer og instrumentering*. Institutt for elektroniske systemer, NTNU, 2025. Kompendium for emnet TTT4280.
- [4] Oppenheim, Alan V. og Alan S. Willsky: *Signals and Systems*. Pearson, 1997.
- [5] Haugen, Finn: *Second Order Systems*. TechTeach, 2009. https://techteach.no/publications/articles/second_order_systems.pdf, sjekket: 2026-02-10, Se p. 3 for frekvens respons magnitude formelen.
- [6] Electronics, SparkFun: *SparkFun Pi Wedge*. <https://www.sparkfun.com/sparkfun-pi-wedge.html#content-documentation>, sjekket: 2026-02-06 .
- [7] Inc., Microchip Technology: *MCP3201*. 2011. <https://ww1.microchip.com/downloads/aemDocuments/documents/APID/ProductDocuments/DataSheets/21290F.pdf>, sjekket: 2026-02-02 .
- [8] Wikipedia: *Transfer function*. https://en.wikipedia.org/wiki/Transfer_function, sjekket: 2026-02-06 .
- [9] ni: *Understanding FFTs and Windowing*. https://www.ni.com/en/shop/data-acquisition/measurement-fundamentals/analog-fundamentals/understanding-ddft-and-windowing.html?srsltid=AfmB0oqFFz0iiVB_YvwwyglFyU8sG2ucYZ2QV6AV3Sdvc7J-qxTPuOU1, sjekket: 2026-02-10 .

A Vedlegg A

Link til GitHub for plottkode:

<https://github.com/lisrad1513/TTT4280/tree/main/Labber/Labb1> (2026-02-10)

B Vedlegg B

```
1  /*
2   adc_sampler.c
3   Public Domain
4   January 2018, Kristoffer Kj      rnes & Asgeir Bj      rgan
5   Based on example code from the pigpio library by Joan @ raspi forum and github
6   https://github.com/joan2937 | http://abyz.me.uk/rpi/pigpio/
7
8   Compile with:
9   gcc -Wall -lpthread -o adc_sampler adc_sampler.c -lpigpio -lm
10
11 Run with:
12 sudo ./adc_sampler
13
14 This code bit bangs SPI on several devices using DMA.
15
16 Using DMA to bit bang allows for two advantages
17 1) the time of the SPI transaction can be guaranteed to within a
18     microsecond or so.
19
20 2) multiple devices of the same type can be read or written
21     simultaneously.
22
23 This code reads several MCP3201 ADCs in parallel, and writes the data to a
24     binary file.
25 Each MCP3201 shares the SPI clock and slave select lines but has
26 a unique MISO line. The MOSI line is not in use, since the MCP3201 is single
27 channel ADC without need for any input to initiate sampling.
28 */
29 #include <stdio.h>
30 #include <stdlib.h>
31 #include <unistd.h>
32
33 #include <pigpio.h>
34 #include <math.h>
35 #include <time.h>
36 ////////////// USER SHOULD MAKE SURE THESE DEFINES CORRESPOND TO THEIR SETUP //////////
37 #define ADCS 3          // Number of connected MCP3201. Changed from 5 to 3 for
38     labb 1
39 #define OUTPUT_DATA argv[2] // path and filename to dump buffered ADC data
40
41 /* RPi PIN ASSIGNMENTS */
42 #define MISO1 27          // ADC 1 MISO (GPIO pin number) Changed from 21 to 27 for
43     labb 1
44 #define MISO2 22          // ADC 2 MISO (GPIO pin number) Changed from 22 to 22 for
45     labb 1
```

```

44 #define MISO3 12      // ADC 3 MISO (GPIO pin number) Changed from 23 to 12 for
45     labb 1
46 #define MISO4 25      // ADC 4 MISO (GPIO pin number) Not used in labb 1
47 #define MISO5 26      // ADC 5 MISO (GPIO pin number) Not used in labb 1
48
49 #define MOSI 10        // GPIO for SPI MOSI (GPIO 10 aka SPI_MOSI). MOSI not in
    use here due to single ch. ADCs, but must be defined anyway.
50 #define SPI_SS 18      // GPIO for slave select (GPIO 18).
51 #define CLK 19         // GPIO for SPI clock (GPIO 19).
52 /* END RPi PIN ASSIGNMENTS */
53
54 #define BITS 12          // Bits per sample.
55 #define BX 4              // Bit position of data bit B11. (3 first are
    t_sample + null bit)
56 #define BO (BX + BITS - 1) // Bit position of data bit B0.
57
58 #define NUM_SAMPLES_IN_BUFFER 300 // Generally make this buffer as large as
    possible in order to cope with reschedule.
59
60 #define REPEAT_MICROS 32 // Reading every x microseconds. Must be no less than
    2xB0 defined above
61
62 #define DEFAULT_NUM_SAMPLES 31250 // Default number of samples for printing in
    the example. Should give 1sec of data at Tp=32us.
63
64 int MISO[ADCS]={MISO1, MISO2, MISO3, MISO4, MISO5}; // Must be updated if you
    change number of ADCs/MISOs above
65 ////////////// END USER SHOULD MAKE SURE THESE DEFINES CORRESPOND TO THEIR SETUP
    ///////////
66 /**
67 * This function extracts the MISO bits for each ADC and
68 * collates them into a reading per ADC.
69 *
70 * \param adcs Number of attached ADCs
71 * \param MISO The GPIO connected to the ADCs data out
72 * \param bytes Bytes between readings
73 * \param bits Bits per reading
74 * \param buf Output buffer
75 */
76 void getReading(int adcs, int *MISO, int OOL, int bytes, int bits, char *buf)
77 {
78     int p = OOL;
79     int i, a;
80
81     for (i=0; i < bits; i++) {
82         uint32_t level = rawWaveGetOut(p);
83         for (a=0; a < adcs; a++) {
84             putBitInBytes(i, buf+(bytes*a), level & (1<<MISO[a]));
85         }
86         p--;
87     }
88 }
89
90
91 int main(int argc, char *argv[])
92 {

```

```

93     // Parse command line arguments
94     long num_samples = 0;
95     if (argc <= 1) {
96         fprintf(stderr, "Usage: %s NUM_SAMPLES\n\n"
97                 "Example: %s %d\n", argv[0], argv[0],
98                 DEFAULT_NUM_SAMPLES);
99         exit(1);
100    }
101    sscanf(argv[1], "%ld", &num_samples);
102
103    // Array over sampled values, into which data will be saved
104    uint16_t *val = (uint16_t*)malloc(sizeof(uint16_t)*num_samples*ADCs);
105
106    // SPI transfer settings, time resolution 1us (1MHz system clock is used)
107    rawSPI_t rawSPI =
108    {
109        .clk      = CLK, // Defined before
110        .mosi     = MOSI, // Defined before
111        .ss_pol   = 1,   // Slave select resting level.
112        .ss_us    = 1,   // Wait 1 micro after asserting slave select.
113        .clk_pol  = 0,   // Clock resting level.
114        .clk_pha  = 0,   // 0 sample on first edge, 1 sample on second edge.
115        .clk_us   = 1,   // 2 clocks needed per bit so 500 kbps.
116    };
117
118    // Change timer to use PWM clock instead of PCM clock. Default is PCM
119    // clock, but playing sound on the system (e.g. espeak at boot) will start
120    // sound systems that will take over the PCM timer and make adc_sampler.c
121    // sample at far lower samplerates than what we desire.
122    // Changing to PWM should fix this problem.
123    gpiocfgClock(5, 0, 0);
124
125    // Initialize the pigpio library
126    if (gpioinit() < 0) {
127        return 1;
128    }
129
130    // Set the selected CLK, MOSI and SPI_SS pins as output pins
131    gpiosetmode(rawSPI.clk, PI_OUTPUT);
132    gpiosetmode(rawSPI.mosi, PI_OUTPUT);
133    gpiosetmode(SPI_SS, PI_OUTPUT);
134
135    // Flush any old unused wave data.
136    gpiowaveaddnew();
137
138    // Construct bit-banged SPI reads. Each ADC reading is stored separately
139    // along a buffer of DMA commands (control blocks). When the DMA engine
140    // reaches the end of the buffer, it restarts on the start of the buffer
141    int offset = 0;
142    int i;
143    char buf[2];
144    for (i=0; i < NUM_SAMPLES_IN_BUFFER; i++) {
145        buf[0] = 0xC0; // Start bit, single ended, channel 0.
146
147        rawwaveaddspi(&rawSPI, offset, SPI_SS, buf, 2, BX, B0, B0);
148        offset += REPEAT_MICROS;
149    }

```

```

149     // Force the same delay after the last command in the buffer
150     gpioPulse_t final[2];
151     final[0].gpioOn = 0;
152     final[0].gpioOff = 0;
153     final[0].usDelay = offset;
154
155     final[1].gpioOn = 0; // Need a dummy to force the final delay.
156     final[1].gpioOff = 0;
157     final[1].usDelay = 0;
158
159     gpioWaveAddGeneric(2, final);
160
161     // Construct the wave from added data.
162     int wid = gpioWaveCreate();
163     if (wid < 0) {
164         fprintf(stderr, "Can't create wave, buffer size %d too large?\n",
165             NUM_SAMPLES_IN_BUFFER);
166         return 1;
167     }
168
169     // Obtain addresses for the top and bottom control blocks (CB) in the DMA
170     // output buffer. As the wave is being transmitted, the current CB will be
171     // between botCB and topCB inclusive.
172     rawWaveInfo_t rwi = rawWaveInfo(wid);
173     int botCB = rwi.botCB;
174     int topOOL = rwi.topOOL;
175     float cbs_per_reading = (float)rwi.numCB / (float)NUM_SAMPLES_IN_BUFFER;
176
177     float expected_sample_freq_khz = 1000.0/(1.0*REPEAT_MICROS);
178
179     printf("# Starting sampling: %ld samples (expected Tp = %d us, expected Fs
180           = %.3f kHz).\n",
181           num_samples, REPEAT_MICROS, expected_sample_freq_khz);
182
183     // Start DMA engine and start sending ADC reading commands
184     gpioWaveTxSend(wid, PI_WAVE_MODE_REPEAT);
185
186     // Read back the samples
187     double start_time = time_time();
188     int reading = 0;
189     int sample = 0;
190
191     while (sample < num_samples) {
192         // Get position along DMA control block buffer corresponding to the
193         // current output command.
194         int cb = rawWaveCB() - botCB;
195         int now_reading = (float) cb / cbs_per_reading;
196
197         while ((now_reading != reading) && (sample < num_samples)) {
198             // Read samples from DMA input buffer up until the current output
199             // command
200
201             // OOL are allocated from the top down. There are BITS bits for
202             // each ADC
203             // reading and NUM_SAMPLES_IN_BUFFER ADC readings. The readings
204             // will be

```

```

200         // stored in top00L - 1 to top00L - (BITS * NUM_SAMPLES_IN_BUFFER)
201
202         // Position of each reading's 00L are calculated relative to the
203         // wave's top
204         // 00L.
205         int reading_address = top00L - ((reading % NUM_SAMPLES_IN_BUFFER)*
206 BITS) - 1;
207
208         char rx[8];
209         getReading(ADCS, MISO, reading_address, 2, BITS, rx);
210
211         // Convert and save to output array
212         for (i=0; i < ADCS; i++) {
213             val[sample*ADCS+i] = (rx[i*2]<<4) + (rx[(i*2)+1]>>4);
214         }
215
216         ++sample;
217
218         if (++reading >= NUM_SAMPLES_IN_BUFFER) {
219             reading = 0;
220         }
221         usleep(1000);
222     }
223
224     double end_time = time_time();
225
226     double nominal_period_us = 1.0*(end_time-start_time)/(1.0*num_samples)*1.0
e06;
227     double nominal_sample_freq_khz = 1000.0/nominal_period_us;
228
229     printf("# %ld samples in %.6f seconds (actual T_p = %f us, nominal Fs =
%.2f kHz).\n",
230           num_samples, end_time-start_time, nominal_period_us,
nominal_sample_freq_khz);
231
232     double output_nominal_period_us = floor(nominal_period_us); //the clock is
233     accurate only to us resolution
234
235     // Path to your data directory/file from previous define
236     const char *output_filename;
237     char hold_fname[32];
238     if (argc < 3) { // No filename supplied, get default
239         time_t t = time(NULL);
240         struct tm tm = *localtime(&t);
241         snprintf(hold_fname, 32, "./out-%4d-%02d-%02d-%02d.%02d.%02d.bin",
tm.tm_year+1900, tm.tm_mon+1, tm.tm_mday,
tm.tm_hour, tm.tm_min, tm.tm_sec);
242         output_filename = hold_fname;
243     } else {
244         output_filename = OUTPUT_DATA;
245     }
246
247     // Write sample period and data to file
248     FILE *adc_data_file = fopen(output_filename, "wb+");
if (adc_data_file == NULL) {

```

```

249     fprintf(stderr, "# Couldn't open file for writing: %s (did you
250         remember to change OUTPUT_DATA?)\n", output_filename);
251     return 1;
252 }
253
254     fwrite(&output_nominal_period_us, sizeof(double), 1, adc_data_file);
255     fwrite(val, sizeof(uint16_t), ADCS*num_samples, adc_data_file);
256     fclose(adc_data_file);
257     printf("# Data written to file. Program ended successfully.\n\n");
258
259     gpioTerminate();
260     free(val);
261
262     return 0;
263 }
```

Kode 1: Koden brukt for å lese av fra ADC-ene. Denne blir kjørt direkte på RPi3

C Vedlegg C

```

1 import numpy as np
2 import sys
3
4
5 def raspi_import(path, channels=5):
6     """
7         Import data produced using adc_sampler.c.
8
9     Returns sample period and a (`samples`, `channels`) `float64` array of
10    sampled data from all `channels` channels.
11
12    Example (requires a recording named `foo.bin`):
13    ```
14    >>> from raspi_import import raspi_import
15    >>> sample_period, data = raspi_import('foo.bin')
16    >>> print(data.shape)
17    (31250, 5)
18    >>> print(sample_period)
19    3.2e-05
20
21    ```
22    """
23
24    with open(path, 'r') as fid:
25        sample_period = np.fromfile(fid, count=1, dtype=float)[0]
26        data = np.fromfile(fid, dtype='uint16').astype('float64')
27        # The "dangling" `.astype('float64')` casts data to double precision
28        # Stops noisy autocorrelation due to overflow
29        data = data.reshape((-1, channels))
30
31    # sample period is given in microseconds, so this changes units to seconds
32    sample_period *= 1e-6
33    return sample_period, data
34
35
36 # Import data from bin file
```

```
37 if __name__ == "__main__":
38     sample_period, data = raspi_import(sys.argv[1] if len(sys.argv) > 1)
39     else 'foo.bin')
```

Kode 2: Koden brukt for å konvertere den binære filen fra vedlegg 1 til en brukbar array.