NTNU

elsys
ELEKTRONISK SYSTEMDESIGN & INNOVASJON

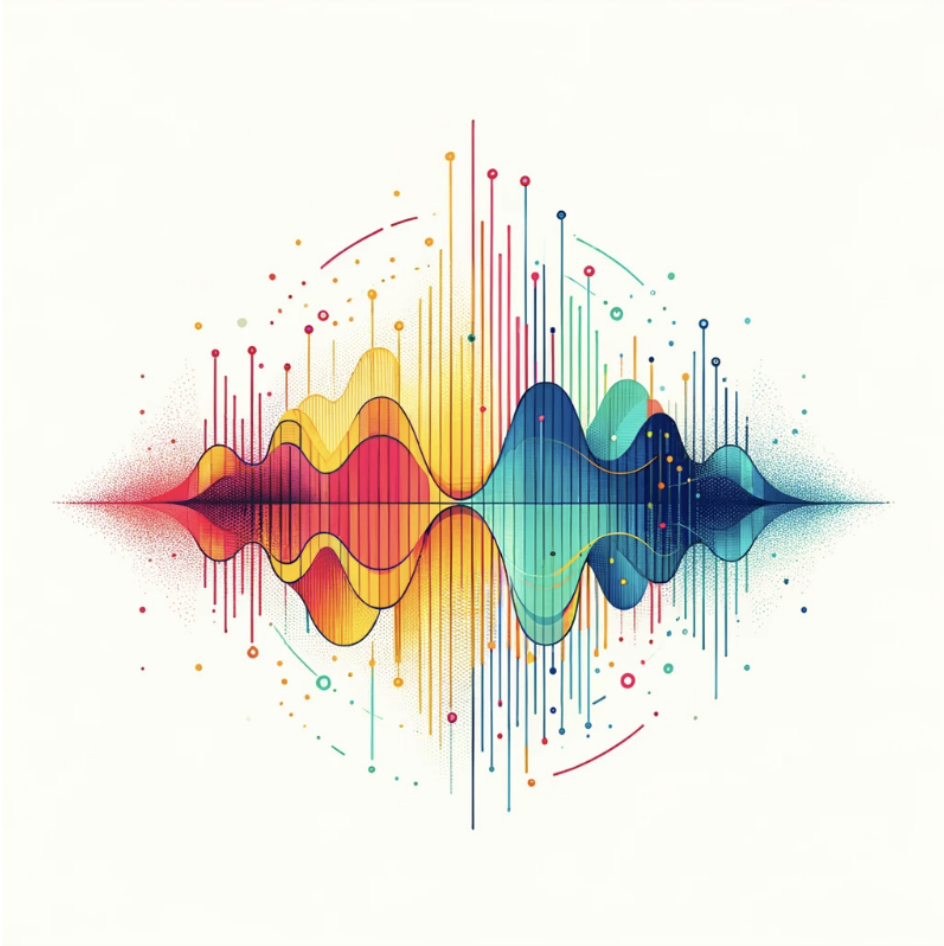
TTT4280 – Labrapport 1

Tittel: Systemoppsett

Skrevet av: Freider Fløan og Theodor Qvist

Gruppe: 44

Dato: 16. februar 2024



Figur 1: Dalle-3.

Sammendrag

Det ble utviklet en måleplattform for å utføre synkroniserte målinger fra fem parallellkoblede analog-til-digital konvertere (ADC-er) med en bitdybde på 12. For å eliminere støy fra ADC-enes strømkilde, er det benyttet et lavpass filter. Videre er bypass-kondensatorer implementert ved målenodene til ADC-ene for å dempe målestøy.

Innhold

Liste over Forkortelser

AD-Konverter Analog til Digital Konverter

ADC Analog to Digital Converter

FFT Fast Fourier Transform (Rask Fourier-transformasjon)

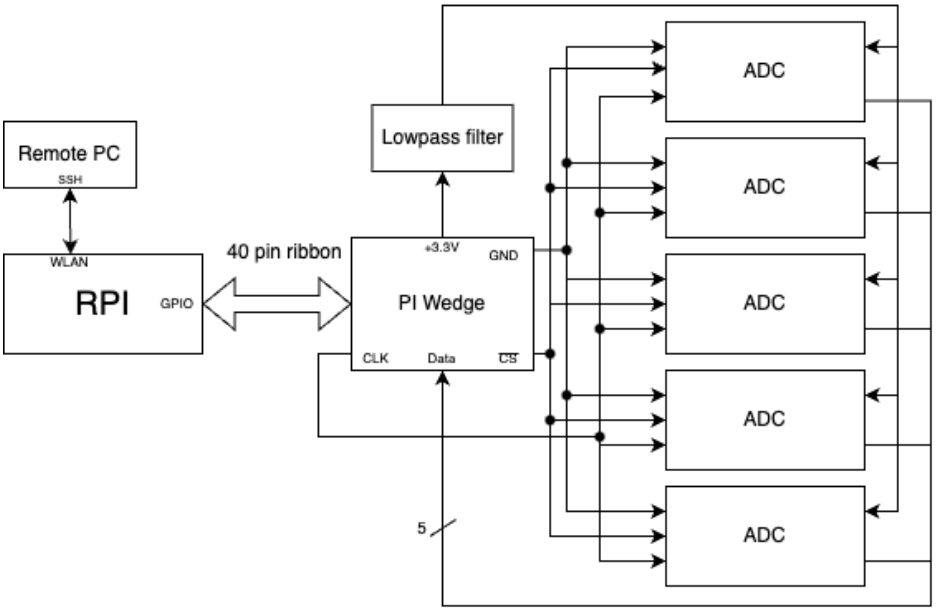
IC Integrated Circuit (Integrert Krets)

RMS Root Mean Square (Kvadratisk Middelverdi)

SNR Signal-to-Noise Ratio (Signal-til-Støy-Forhold)

1 Innledning

I denne labrapporten utforskes oppsettet og valideringen av et målesystem basert på Raspberry Pi 3B, kombinert med eksterne AD-konvertere for å registrere analoge signaler. Målet er å skape en plattform for effektiv datainnsamling fra mikrofoner og en radar, med fokus på systemets ytelse og nøyaktighet. I figur 2 er et blokkskjema for hvordan oppsettet av kretsen skal være.



Figur 2: Blokkskjema av systemet

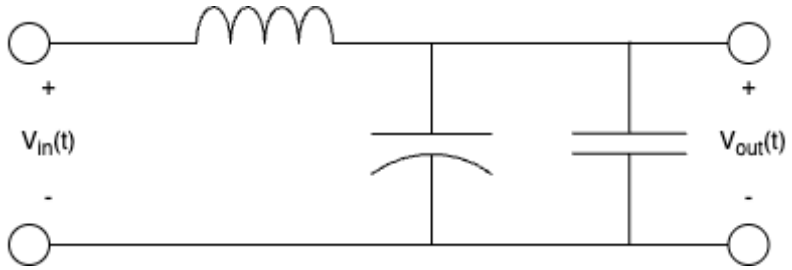
Videre undersøker vi Raspberry Pi's funksjonalitet som måleplattform sammen med AD-konvertere.

2 Teori

I denne seksjonen vil vi utforske et teoretiske grunnlaget for bruk av Raspberry Pi 3B og AD-konvertere i vårt målesystem.

2.1 Lavpass filter

En elektrisk krets kan variere i hvor mye strøm den trekker fra en spenningskilde, noe som kan føre til at spenningen over kretsen endres. Slike variasjoner i spenningen kan introdusere støy i følsomme kretser, som for eksempel ADC-er. I noen tilfeller kan spenningskilden selv være ustabilt, spesielt hvis den kommer fra en digital enhet der spenningsnivået endrer seg i takt med enhetens prosesseringsaktivitet. For å gjøre spenningskilden mer stabil, anbefales det å installere et lavpassfilter ved spenningskilden, noe som kan oppnås med et enkelt RC-filter. For ytterligere å redusere støy fra både spenningskilden og kretsen, kan et Pi-filter være et bedre alternativ. Figur 12 viser et slikt filter.



Figur 3: Lavpass PI filter.

Et Pi-filter er i prinsippet et LC-filter designet for å fungere effektivt i begge retninger, og det inkluderer en ekstra bypass-kondensator for bedre ytelse i filtrering av støy. I dette prosjektet er filteret designet til å kun fungere en vei. Når støy kommer fra inngangsspenningen (V_in), vil støyen møte et lavpassfilter bestående av en spole og kondensator. Her fungerer kondensatoren som en bypass-kondensator, effektivt omgående høyfrekvent støy.

For å bestemme LC-filterets knekkfrekvens, som er frekvensen hvor filteret begynner å dempe signaler fra 3 dB, må man utlede filterets systemfunksjon. Systemfunksjonen tar hensyn til filterets komponenter og deres arrangement, men i dette tilfellet ser vi bort fra effekten av inngangskondensatoren for å forenkle analysen. Knekkfrekvensen kan deretter beregnes ved å løse systemfunksjonen for den frekvensen som resulterer i en 3 dB demping av signalet.

$$H(s) = \frac{1}{s^2CL + 1} \tag{1}$$

Vi kan deretter sette $s = j\omega$ inn i formel 1 og amplituden til $\frac{1}{\sqrt{2}}$.

$$f_c = \frac{1}{2\pi}\sqrt{\frac{1+\sqrt{2}}{CL}} \tag{2}$$

2.2 Raspberry Pi

Raspberry Pi 3B er en enkeltkorts datamaskin, som tillater direkte tilkobling og kommunikasjon med eksterne enheter, inkludert AD-konvertere. Videre muliggjør Raspberry Pi's evne til å kjøre operativsystemer og derfor støtte programmeringsspråk behandling og analyse av innsamlede data.

I tilleg implementeres Direct Memory Access (DMA) for å effektivisere dataoverføringen mellom AD-konverterene og Raspberry Pi's minne. Ved å tillate data å bli overført direkte til minnet uten å belaste prosessoren, reduserer DMA forsinkelser og flaskehalser i dataflyten. Dette er spesielt viktig i applikasjoner som krever høy samplingsrate og sanntidsbehandling, hvor hver millisekund av forsinkelse kan påvirke nøyaktigheten og påliteligheten av de innsamlede dataene.

2.3 ADC

En Analog-til-Digital konverter (ADC) er en integrert krets som omformer de kontinuerlige analoge signalene i verden rundt oss til digitale verdier som kan behandles av Raspberry Pi. Valget av AD-konvertere baseres på nødvendig oppløsning og samplingsrate for å fange opp signalenes uten for mye støy. En AD-konverter med N-bits oppløsning produserer digitale verdier som representerer 2^N mulige nivåer, noe som dikterer systemets evne til å skille små variasjoner i det analoge signalet. Samplingsraten, som må være minst det dobbelte av signalets høyeste frekvensinnhold i henhold til Nyquist-Shannon samplingsteoremet, dette gjøres for å unngå aliasing og sikre en troverdig digital representasjon av det analoge signalet.

Det vil være ønskelig å ta punktprøver med AD-konverterene og det er derfor hensiktsmessig å undersøke antallet samples nødvendig for å gjøre det. Utfra databladet til MCP3201 AD-konverter kan man finne ut at det er nødvendig med 16 samples. Fra databladet kan man også finne oppløsningen (eller steglengden) til AD-konverterene, se formel 3 [1].

$$\text{Oppløsning} = \Delta V = \frac{V_{ref}}{2^N} \tag{3}$$

Det vil også brukes bypass kondensatorer for å stabilisere strømforsyningen og minimere støy. Dette gjøres for å fungere som en lokal strømkilde på tilkoblingspunktene på den integrerte kretsen, og ved å filtrere bort støy fra strømforsyningen ved å kortslutte høyfrekvent støy til jord. I oppkoblingen vil bruk av bypass kondensatorer være viktig for å sikre at ADC-en opererer stabilt uten forstyrrelser fra strømforsyningen eller andre kilder til elektrisk støy.

2.4 Signalbehandling - FFT

I signalbehandling delen av labben, brukes Fast Fourier Transform (FFT) for å analysere signalene som er samlet inn via det oppsatte systemet. FFT vil tillate å konvertere de tidsdomene-baserte signalene fra ADCen til deres frekvensdomenerepresentasjoner. Dette er spesielt nyttig for å identifisere og analysere frekvenskomponentene i signalene som er fanget opp, for eksempel for å observere harmoniske forstyrrelser, signal-til-støy-forhold (SNR) eller for å validere at systemet filtrerer ut uønskede frekvenser effektivt. Bruken av FFT i denne labben vil gi innsikt i hvordan det digitale signalet oppfører seg i frekvensdomenet, og hvordan systemets ytelse kan optimaliseres videre basert på disse observasjonene. [2]

2.5 Signal-to-Noise Ratio

SNR defineres som forholdet mellom signalstyrken og bakgrunnsstøyen i systemet. Dette forholdet, ofte uttrykt i desibel (dB), gir en indikator på signalets renhet. For et gitt signal med en kjent frekvenskomponent ω_k, beregnes SNR ved å sammenligne styrken av denne komponenten, representert ved P_k, med det totale frekvensinnholdet ω_n i signalet:

$$\text{SNR} = \frac{P_k}{P_n}. \tag{4}$$

Teoretisk maksimal SNR, uttrykt i dB, kan estimere ved å anvende følgende formel:

$$\text{SNR}_{\text{max}} = 10 \log_{10} \left(\frac{e^2_{\text{rms,signal,in}}}{\epsilon^2_{\text{rms}}} \right), \tag{5}$$

hvor $e_{\text{rms,signal,in}}$ representerer rotmiddelkvadratet (RMS) av signalets inngangsenergi, og ϵ_{rms} er RMS-verdien av kvantiseringstøy. Ved å sette inn verdier for RMS-styrken til signalet og kvantiseringstøyet, kan vi estimere en øvre grense for SNR:

$$\text{SNR}_{\text{max}} \approx 1.76 + 6.02 \cdot N, \tag{6}$$

der N representerer antall bits i ADC-konverteringen. Det er imidlertid viktig å bemerke at denne beregningen kun tar hensyn til støy fra kvantiseringen i ADC-en. I praksis vil ytterligere støykilder, som elektronisk støy fra komponentene og miljømessig støy, bidra til å redusere SNR.

For vårt eksperiment, med ADC-er som opererer med en oppløsning på $N = 12$ bits, resulterer dette i en teoretisk maksimal SNR på ca. 74 dB. Likevel er dette en idealisering, og de faktiske målingene vil sannsynligvis vise et lavere SNR på grunn av nevnte ekstra støykilder.

2.6 Vindusfunksjoner

Vindusfunksjoner spiller en viktig rolle i digital signalbehandling, spesielt når det gjelder spektralanalyse. En vindusfunksjon anvendes på et signal for å minimere effektene av spektral lekkasje ved å redusere signalverdiene ved endepunktene til null.

2.7 Zero-Padding

Zero-padding er en teknikk som brukes for å forbedre frekvensoppløsningen i spektralanalyse. Ved å legge til nuller til slutten av et signal før utførelsen av en Fourier-transformasjon, kan lengden på det diskrete Fourier-transformerte signalet økes, noe som resulterer i et frekvensspektrum med finere frekvensoppløsning. Det er viktig å merke seg at mens zero-padding forbedrer evnen til å skille mellom nærliggende frekvenskomponenter i spekteret, introduserer det ikke ny informasjon i signalet.

Sammen utgjør vindusfunksjoner og zero-padding kraftige verktøy for å utføre mer nøyaktige og informative spektralanalyser.

3 Systemoppsett

3.1 Oppsett av Raspberry Pi

Raspberry Pi 3B ble forberedt for å fungere som hjernen i vårt målesystem. Første trinn i oppsettet involverte installasjonen av Raspberry Pi OS, et operativsystem optimalisert for Raspberry Pi, som tilbyr støtte for nødvendig nettverkskonfigurasjon, programmeringsspråk og datahåndteringsverktøy.

Etter installasjonen av operativsystemet ble nettverksinnstillingene konfigurert for å muliggjøre fjernaksess via Secure Shell (SSH), en protokoll som tillater sikker tilkobling og kommandoutførelse over nettverk. Etter innsamling ble de digitale dataene overført fra Raspberry Pi til host ved hjelp av Secure Copy Protocol (SCP). SCP-kommandoen som ble brukt for denne prosessen, er generelt formatert som følger:

```
scp [Bruker]@[RaspberryPi-adresse]:[Sti/til/fil] [Destinasjonssti]
```

3.2 Oppkobling lavpass filter

Lavpass filteret ble koblet med komponentene listet i Tabell 1.

Tabell 1: Komponenter brukt i lavpassfilter.

Type	Verdi
Induktor	100 mH
Kondensator	100 nF
Kondensator	470 µF

Knekkfrekvensen med de brukte komponentene ble regnet ut til å være;

$$f_c = \frac{1}{2\pi}\sqrt{\frac{1+\sqrt{2}}{CL}} = \frac{1}{2\pi}\sqrt{\frac{1+\sqrt{2}}{474.1 \cdot 10^{-6} \cdot 0.1}} = 36Hz \tag{7}$$

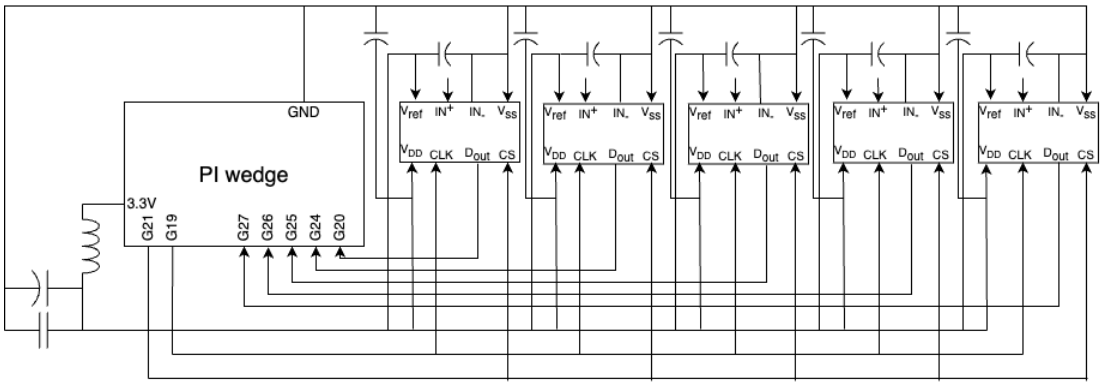
3.3 Tilkobling av AD-konvertere

For å lese av de analoge signalene til digitale signaler på Raspberry Pi koblet vi fem Analog-Digital-Converters (ADC). Vi benyttet MCP3201 ADC som var utdelt, med 12-bits oppløsning.

Hver ADC ble koblet til Raspberry Pi gjennom GPIO-pinnene. Dette inkluderte tilkobling av SPI (Serial Peripheral Interface) kommunikasjonslinjer, som består av klokke (SCLK), master

in slave out (MISO), og chip select (CS) pinner, for å muliggjøre synkronisert datautveksling mellom Raspberry Pi og ADCene.

Et kretsskjema ble utarbeidet, figur 4, for å illustrere oppkoblingen, hvor signalveier, strømforsyning, og jording ble nøye indikert for å sikre klarhet i systemets oppbygning. Dette skjemaet fremhevet hvordan alle fem ADCene delte en felles klokke og chip select signal, mens de opprettholdt separate MISO linjer for å tillate individuell datautlesning.



Figur 4: Kretsskjema av oppkoblingen.

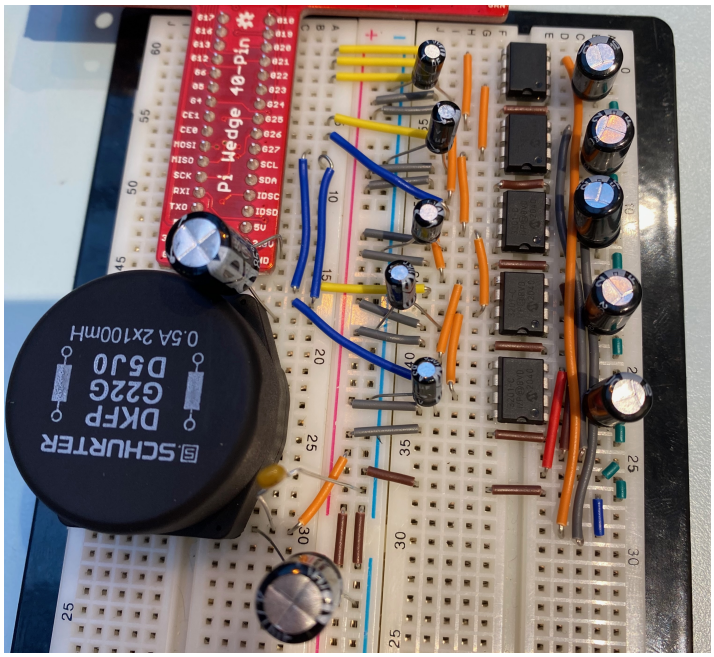
De brukte kondensatorverdiene for bypass kondensatorene i oppkoblingen er listet i 2.

Tabell 2: Kondensatorverdier brukt i oppkobling av ADC-ene.

Type	Verdi	Antall
Kondensator fra V_{ref} til IN_-	100 µF	5
Kondensator fra GND til V_{DD}	1 µF	5

I tillegg til den fysiske tilkoblingen, ble det foretatt en kalibrering av systemet for å validere funksjonaliteten og nøyaktigheten av datainnsamlingen. Dette inkluderte testing med kjente signaler for å bekrefte at AD-konverterenes utlesninger korresponderte nøyaktig med de forventede verdiene.

Figur 5 er et bilde av den oppkoblede kretsen, med lavpass filter og 5 ADC-er.



Figur 5: Bilde av oppkoblet krets.

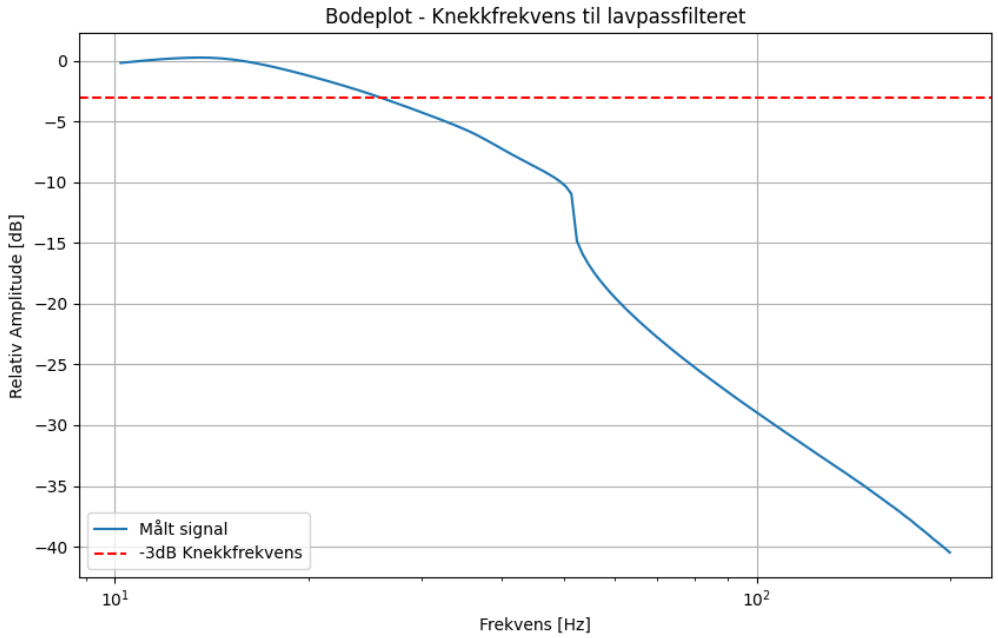
3.4 Konfigurasjon av målesystemet

Etter oppsettet av Raspberry Pi og tilkoblingen av AD-konvertere, var neste trinn å konfigurere målesystemet for optimal datainnsamling. Samplingsraten som ble brukt var gitt fra utdelt kode på 31250. Med tanke på Nyquist-Shannons samplingsteorem var samplingsraten tilstrekkelig høy til å fange opp alle relevante frekvenskomponenter i de analoge signalene uten risiko for aliasing.

4 Resultater og Diskusjon

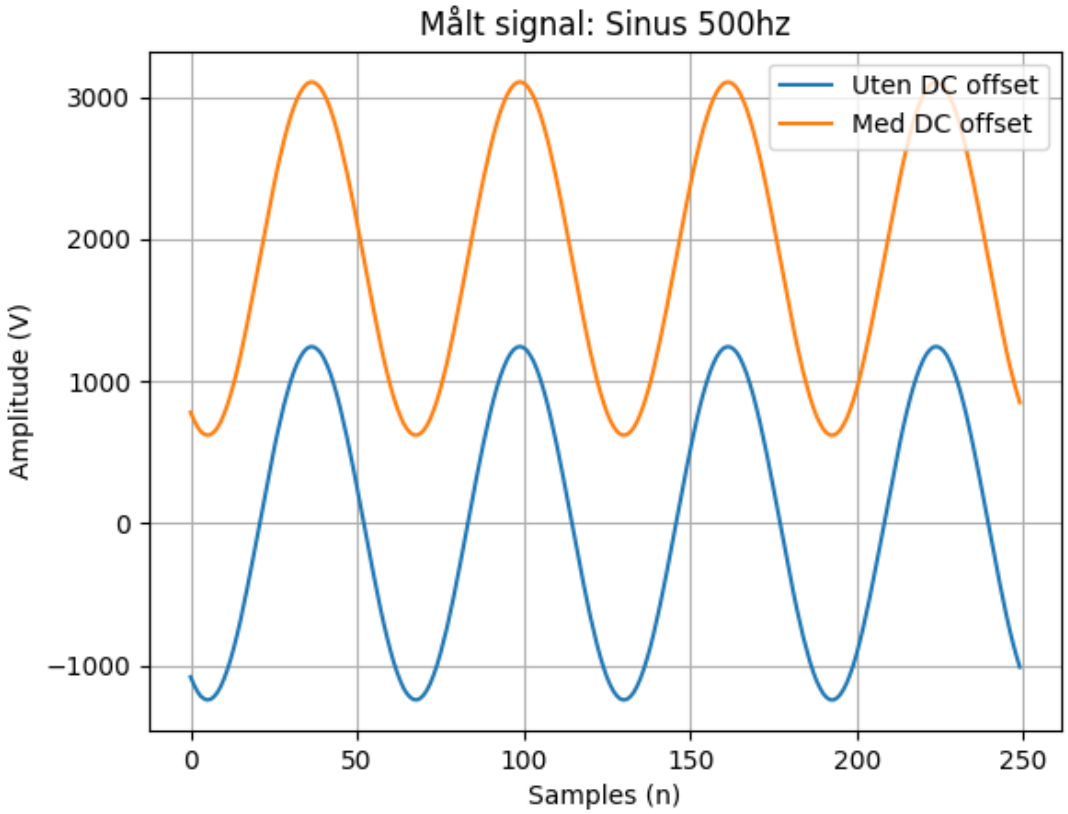
4.1 Test av systemet

Figur 6 viser frekvensresponsen til filteret beskrevet i seksjon 3.2, hvor knekkfrekvensen ble målt til 26Hz.



Figur 6: Knekkfrevens til lavpassfilteret fra seksjon 3.2.

Feilmarginen i knekkfrekvensen fra målt verdi til teoretisk verdi, som diskuteres i seksjon 3.2, kan skyldes unøyaktigheter i komponentverdiene som brukes i filteret. Etter testing av knekkfrevens, ble et sinus signal på 1V varierende mellom 0.5V og 1.5V på 500 Hz påtrykt en av AD-konverterne vist i figur 7.



Figur 7

Siden AD-konverterne ikke måler negative signaler, ble det lagt til en likespenningskomponent på inngangssignalet til AD-konverteren, som ble korrigert, se listing 2, gjennom signalbehandlingen i etterkant, noe som resulterte i tydeligere målinger under Fourier-analysen. For å konvertere målt data til leselig CSV-verdier bruker vi den utdelte koden raspi_import.py fra vedlegg A.

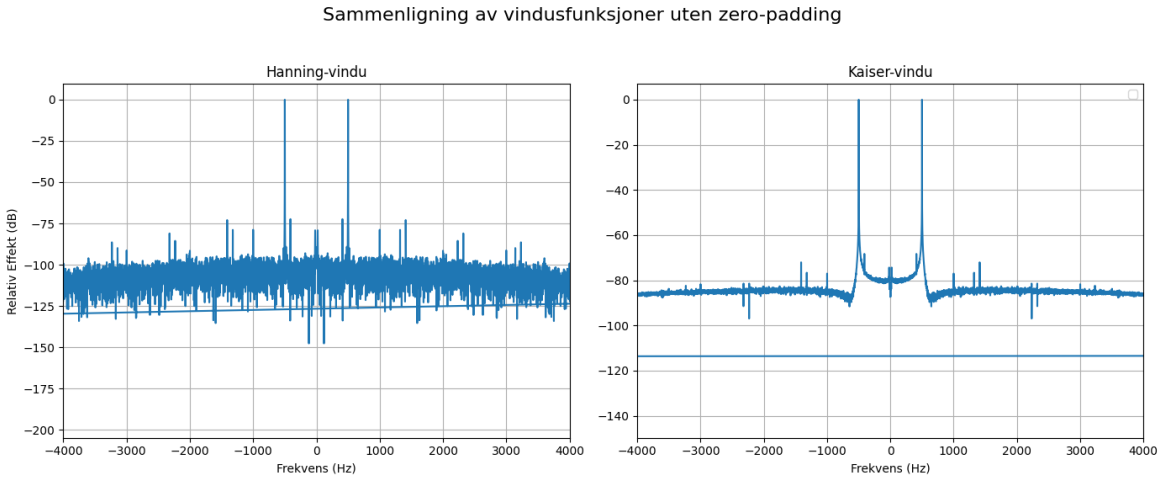
```
1 plt.plot(data[31000:, 0]-np.mean(data[0:, 0]), label="Uten DC offset")
```

Kode 1: Se C for fullstendig kode av plottingen til målt signal fra AD-konverterne.

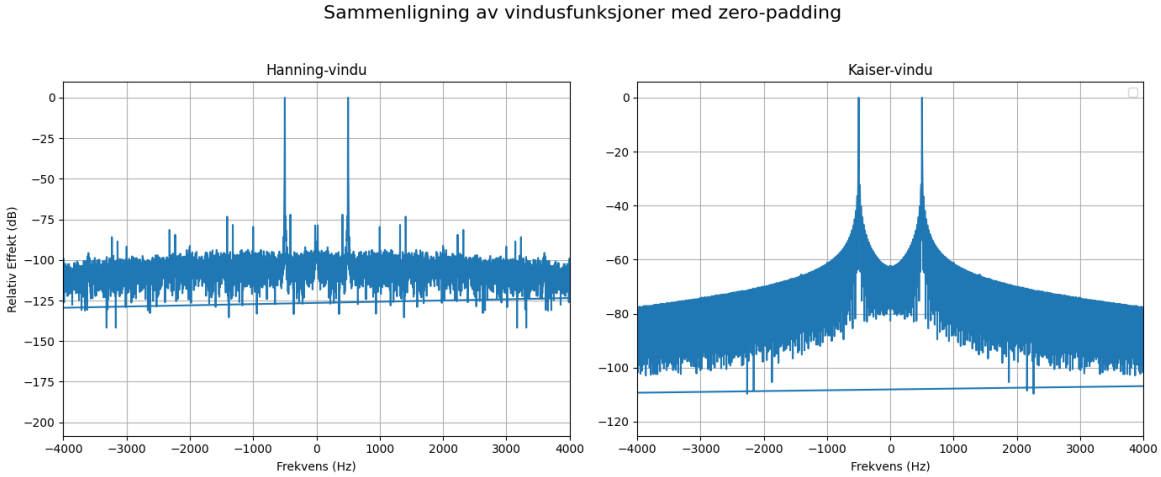
4.2 Spektrumsanalyse

I denne rapporten har vi valgt å se nærmere på Kaiser- og Hanning-vinduer. Hanning-vinduet er kjent for sin evne til å redusere spektral lekkasje med en relativt enkel form, noe som gjør det til et godt valg for et bredt spekter av applikasjoner. Kaiser-vinduet, derimot, tilbyr justerbarhet gjennom sin β-parameter, som tillater finjustering av vindusfunksjonens form for å balansere mellom hovedlobe-bredde og sidelobe-nivåer. Denne fleksibiliteten er spesielt verdifull i applikasjoner der kravene til signalanalyse kan variere. Figur 9 og figur 8 viser

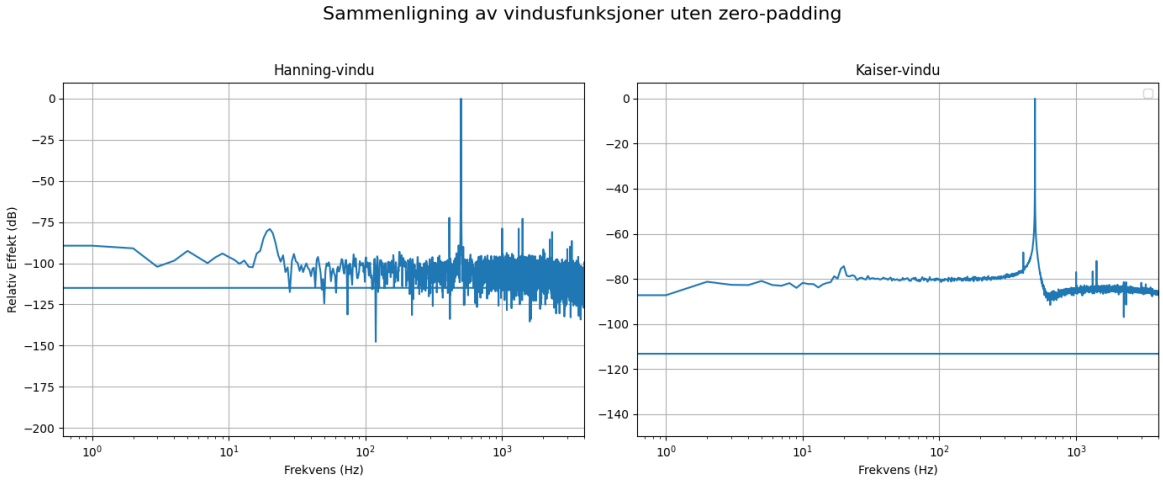
hvordan disse to vindu-funksjonene påvirker det målte signalets frekvensspekter, der figur 10 viser hvordan det kan visualiseres med log-skala på frekvensaksen.



Figur 8: Hanning- og kaiservindu-funksjon uten zero-padding.



Figur 9: Hanning- og kaiservindu-funksjon med zero-padding.

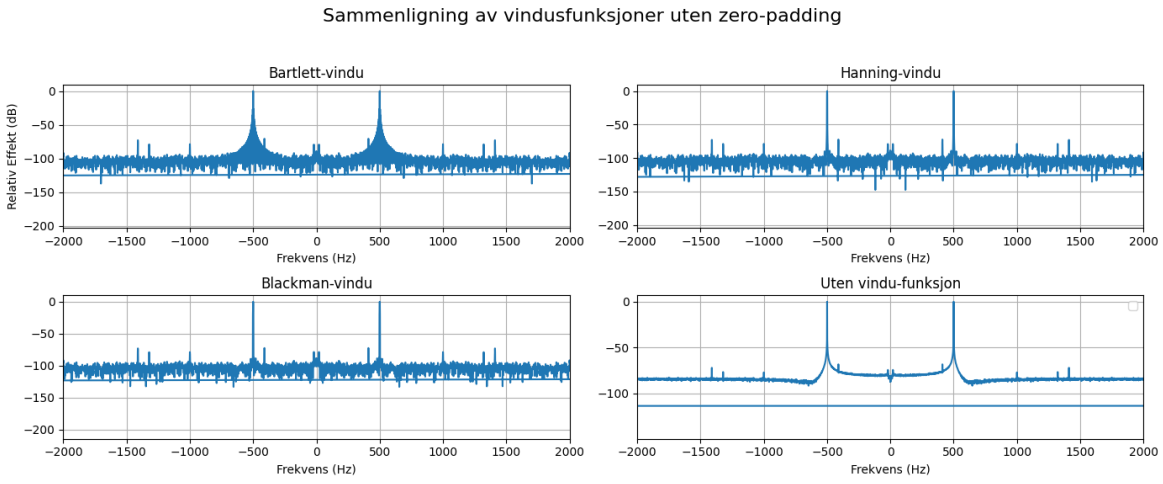


Figur 10: Hanning- og kaiservindu-funksjon med zero-padding i log-skala på frekvensaksen.

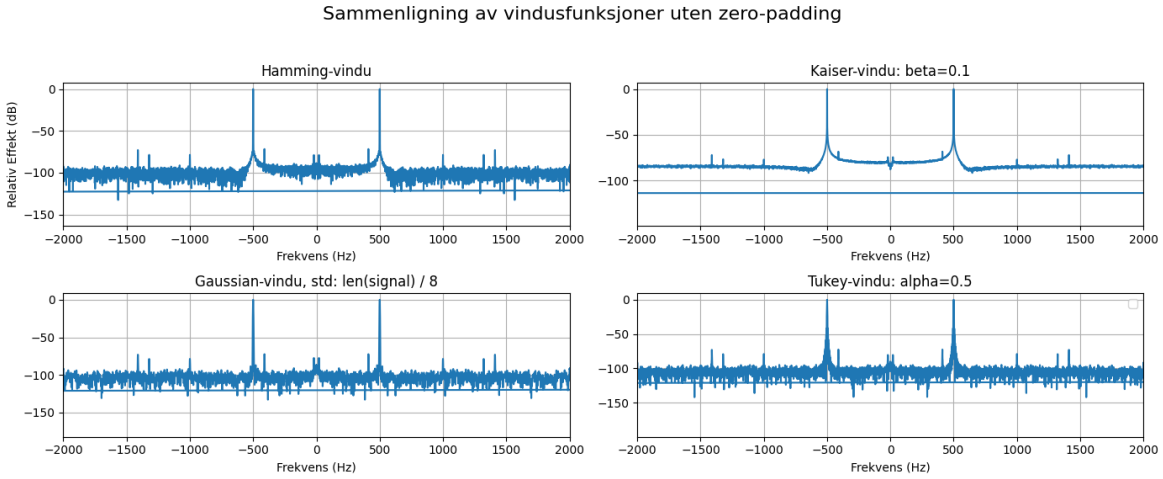
4.3 Effekt av Zero-Padding

Zero-padding ble brukt for å forbedre frekvensoppløsningen i spektrumsanalysen som diskutert i seksjon 2.7.

For å gi en sammenligning, inkluderer vi også analyser ved bruk av Bartlett-, Blackman-, Hamming-, Gaussian- og Tukey-vinduer. Selv om disse ikke er hovedfokuset, gir de innsikt i hvordan ulike vindusfunksjoner kan påvirke spektrumanalysen, og bidrar til en mer helhetlig forståelse av vindusfunksjonenes rolle i digital signalbehandling.



Figur 11: Bartlett-, Blackman-, Hanning-vinduer og uten signalet uten vinduet.



Figur 12: Hamming-, Kaiser-, Gaussian- og Tukey-vinduer.

Det er først her, når vi ser på resultatene ved siden av plottet uten en vindu-funksjon at effekten kommer tydelig frem, der de uønskede frekvensene dempes mer enn 20dB før og etter vindu- funksjonene. For den fullstendige koden og ytterligere detaljer om implementasjonen, vennligst se vedlegg C.

4.4 Måling av Signal-til-Støy-Forhold

Gitt vårt oppsett, hvor vi påtrykte en kjent sinusformet signal på systemet, ble SNR målt ved først å definere det rene signalet som en sinusfunksjon med en kjent frekvens på 500 Hz. Støyen ble deretter estimert som differansen mellom det målte signalet og det rene signalet. Fra teorien i seksjon 2.5 regnet vi ut at maksimal SNR med AD-konverterne var omtrent 74dB. Ved bruk av koden vist under, ble SNRen til vårt signal målt til ≈69.3dB.

```
1
2  import numpy as np
3  data = np.loadtxt('Lab/TTT4280/csv/200_hz.csv', delimiter=',')
4  signal_with_noise = data[:, 0]
5
6  frequency = 500
7  time = np.arange(len(signal_with_noise)) / sampling_rate
8  pure_signal = np.sin(2 * np.pi * frequency * time)
9
10 noise_estimate = signal_with_noise - pure_signal
11
12
13 signal_rms = np.sqrt(np.mean(pure_signal**2))
14 noise_rms = np.sqrt(np.mean(noise_estimate**2))
15 snr_linear = signal_rms / noise_rms
16 snr_db = 20 * np.log10(snr_linear)
17
18 print(f"SNR (linear): {snr_linear}")
19 print(f"SNR (dB): {snr_db}")
```

Kode 2: Pythonkode for å regne ut SNR.

$$\mathrm{SNR}_{\mathrm{dB}} = 20 \cdot \log_{10}\left(\frac{\mathrm{RMS}_{\mathrm{signal}}}{\mathrm{RMS}_{\mathrm{støy}}}\right)$$

hvor $\mathrm{RMS}_{\mathrm{signal}}$ er den kvadratiske middelverdien av det rene signalet, og $\mathrm{RMS}_{\mathrm{støy}}$ er den kvadratiske middelverdien av støyestimatet. Dette ga oss et mål på forholdet mellom ønsket signal og bakgrunnsstøyen i vårt system.

5 Konklusjon

I denne rapporten er det utført en gjennomgang av teorien og metodene som er nødvendige for å etablere et målesystem. Systemet, som er bygget med analog-til-digital konvertere og en mikrodatamaskin, er i stand til å samle inn og lagre data. Dataene er deretter blitt behandlet og analysert på en datamaskin. Rapporten demonstrerer hvordan man kan redusere støy fra strømforsyningen ved hjelp av et lavpass filter, samt hvordan kvantiseringsstøy fra ADC-ene kan identifiseres gjennom dataanalysen. Dette er utført ved hjelp av Fast Fourier Transform og spektralanalyse, og det er illustrert hvordan teknikker som bruk av vindusfunksjoner og zero-padding bidrar til å frembringe klare spektre som effektivt representerer dataene.

Referanser

[1] *2.7V 12-Bit A/D Converter with SPI® Serial Interface.* Microchip Technology Inc., 1999.

[2] U. Peter Svensson, Egil Eide, Lise Lyngnes Randeberg: *TTT4280 Sensorer og instrumentering.* NTNU, 2013.

A Vedlegg A

```
/*
adc_sampler.c
Public Domain
January 2018, Kristoffer Kj      rnes & Asgeir Bj      rgan
Based on example code from the pigpio library by Joan @ raspi forum and github
https://github.com/joan2937 | http://abyz.me.uk/rpi/pigpio/

Compile with:
gcc -Wall -lpthread -o adc_sampler adc_sampler.c -lpigpio -lm

Run with:
sudo ./adc_sampler

This code bit bangs SPI on several devices using DMA.

Using DMA to bit bang allows for two advantages
1) the time of the SPI transaction can be guaranteed to within a
   microsecond or so.

2) multiple devices of the same type can be read or written
   simultaneously.

This code reads several MCP3201 ADCs in parallel, and writes the data to a
    binary file.
Each MCP3201 shares the SPI clock and slave select lines but has
a unique MISO line. The MOSI line is not in use, since the MCP3201 is single
channel ADC without need for any input to initiate sampling.
*/

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#include <pigpio.h>
#include <math.h>
#include <time.h>
/////// USER SHOULD MAKE SURE THESE DEFINES CORRESPOND TO THEIR SETUP ///////
#define ADCS 5      // Number of connected MCP3201.

#define OUTPUT_DATA argv[2] // path and filename to dump buffered ADC data

/* RPi PIN ASSIGNMENTS */
#define MISO1 20    // ADC 1 MISO (BCM 4 aka GPIO 4).
#define MISO2 24    //     2
#define MISO3 25    //     3
#define MISO4 26    //     4
#define MISO5 27    //     5

#define MOSI 10     // GPIO for SPI MOSI (BCM 10 aka GPIO 10 aka SPI_MOSI).
    MOSI not in use here due to single ch. ADCs, but must be defined anyway.
#define SPI_SS 21   // GPIO for slave select (BCM 8 aka GPIO 8 aka SPI_CE0).
#define CLK 19      // GPIO for SPI clock (BCM 11 aka GPIO 11 aka SPI_CLK).
/* END RPi PIN ASSIGNMENTS */

#define BITS 12            // Bits per sample.
```

```
54 #define BX 4                // Bit position of data bit B11. (3 first are
       t_sample + null bit)
55 #define B0 (BX + BITS - 1) // Bit position of data bit B0.
56
57 #define NUM_SAMPLES_IN_BUFFER 300 // Generally make this buffer as large as
       possible in order to cope with reschedule.
58
59 #define REPEAT_MICROS 32 // Reading every x microseconds. Must be no less than
        2xB0 defined above
60
61 #define DEFAULT_NUM_SAMPLES 31250 // Default number of samples for printing in
        the example. Should give 1sec of data at Tp=32us.
62
63 int MISO[ADCS]={MISO1, MISO2, MISO3, MISO4, MISO5}; // Must be updated if you
       change number of ADCs/MISOs above
64 /////// END USER SHOULD MAKE SURE THESE DEFINES CORRESPOND TO THEIR SETUP
       ///////
65
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],
       DEFAULT_NUM_SAMPLES);
98         exit(1);
99     }
100    sscanf(argv[1], "%ld", &num_samples);
101
102    // Array over sampled values, into which data will be saved
103    uint16_t *val = (uint16_t*)malloc(sizeof(uint16_t)*num_samples*ADCS);
```

```
104
105     // SPI transfer settings, time resolution 1us (1MHz system clock is used)
106     rawSPI_t rawSPI =
107     {
108        .clk     =  CLK,  // Defined before
109        .mosi    =  MOSI, // Defined before
110        .ss_pol  =  1,    // Slave select resting level.
111        .ss_us   =  1,    // Wait 1 micro after asserting slave select.
112        .clk_pol =  0,    // Clock resting level.
113        .clk_pha =  0,    // 0 sample on first edge, 1 sample on second edge.
114        .clk_us  =  1,    // 2 clocks needed per bit so 500 kbps.
115     };
116
117     // Change timer to use PWM clock instead of PCM clock. Default is PCM
118     // clock, but playing sound on the system (e.g. espeak at boot) will start
119     // sound systems that will take over the PCM timer and make adc_sampler.c
120     // sample at far lower samplerates than what we desire.
121     // Changing to PWM should fix this problem.
122     gpioCfgClock(5, 0, 0);
123
124     // Initialize the pigpio library
125     if (gpioInitialise() < 0) {
126        return 1;
127     }
128
129     // Set the selected CLK, MOSI and SPI_SS pins as output pins
130     gpioSetMode(rawSPI.clk,  PI_OUTPUT);
131     gpioSetMode(rawSPI.mosi, PI_OUTPUT);
132     gpioSetMode(SPI_SS,      PI_OUTPUT);
133
134     // Flush any old unused wave data.
135     gpioWaveAddNew();
136
137     // Construct bit-banged SPI reads. Each ADC reading is stored separatedly
138     // along a buffer of DMA commands (control blocks). When the DMA engine
139     // reaches the end of the buffer, it restarts on the start of the buffer
140     int offset = 0;
141     int i;
142     char buf[2];
143     for (i=0; i < NUM_SAMPLES_IN_BUFFER; i++) {
144         buf[0] = 0xC0; // Start bit, single ended, channel 0.
145
146         rawWaveAddSPI(&rawSPI, offset, SPI_SS, buf, 2, BX, B0, B0);
147         offset += REPEAT_MICROS;
148     }
149
150     // Force the same delay after the last command in the buffer
151     gpioPulse_t final[2];
152     final[0].gpioOn = 0;
153     final[0].gpioOff = 0;
154     final[0].usDelay = offset;
155
156     final[1].gpioOn = 0; // Need a dummy to force the final delay.
157     final[1].gpioOff = 0;
158     final[1].usDelay = 0;
159
160     gpioWaveAddGeneric(2, final);
```

```
161
162     // Construct the wave from added data.
163     int wid = gpioWaveCreate();
164     if (wid < 0) {
165         fprintf(stderr, "Can't create wave, buffer size %d too large?\n",
    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
     = %.3f kHz).\n",
180     num_samples,REPEAT_MICROS,expected_sample_freq_khz);
181
182     // Start DMA engine and start sending ADC reading commands
183     gpioWaveTxSend(wid, PI_WAVE_MODE_REPEAT);
184
185     // Read back the samples
186     double start_time = time_time();
187     int reading = 0;
188     int sample = 0;
189
190     while (sample < num_samples) {
191         // Get position along DMA control block buffer corresponding to the
    current output command.
192         int cb = rawWaveCB() - botCB;
193         int now_reading = (float) cb / cbs_per_reading;
194
195         while ((now_reading != reading) && (sample < num_samples)) {
196             // Read samples from DMA input buffer up until the current output
    command
197
198             // OOL are allocated from the top down. There are BITS bits for
    each ADC
199             // reading and NUM_SAMPLES_IN_BUFFER ADC readings. The readings
    will be
200             // stored in topOOL - 1 to topOOL - (BITS * NUM_SAMPLES_IN_BUFFER)
    .
201             // Position of each reading's OOL are calculated relative to the
    wave's top
202             // OOL.
203             int reading_address = topOOL - ((reading % NUM_SAMPLES_IN_BUFFER)*
    BITS) - 1;
204
205             char rx[8];
206             getReading(ADCS, MISO, reading_address, 2, BITS, rx);
207
208             // Convert and save to output array
```

```
209             for (i=0; i < ADCS; i++) {
210                 val[sample*ADCS+i] = (rx[i*2]<<4) + (rx[(i*2)+1]>>4);
211             }
212 
213             ++sample;
214 
215             if (++reading >= NUM_SAMPLES_IN_BUFFER) {
216                 reading = 0;
217             }
218         }
219         usleep(1000);
220     }
221 
222     double end_time = time_time();
223 
224     double nominal_period_us = 1.0*(end_time-start_time)/(1.0*num_samples)*1.0
    e06;
225     double nominal_sample_freq_khz = 1000.0/nominal_period_us;
226 
227     printf("# %ld samples in %.6f seconds (actual T_p = %f us, nominal Fs =
    %.2f kHz).\n",
228         num_samples, end_time-start_time, nominal_period_us,
    nominal_sample_freq_khz);
229 
230     double output_nominal_period_us = floor(nominal_period_us); //the clock is
     accurate only to us resolution
231 
232     // Path to your data directory/file from previous define
233     const char *output_filename;
234     char hold_fname[32];
235     if (argc < 3) {  // No filename supplied, get default
236         time_t t = time(NULL);
237         struct tm tm = *localtime(&t);
238         snprintf(hold_fname, 32, "./out-%4d-%02d-%02d-%02d.%02d.%02d.bin",
239                  tm.tm_year+1900, tm.tm_mon, tm.tm_mday,
240                  tm.tm_hour, tm.tm_min, tm.tm_sec);
241         output_filename = hold_fname;
242     } else {
243         output_filename = OUTPUT_DATA;
244     }
245 
246     // Write sample period and data to file
247     FILE *adc_data_file = fopen(output_filename, "wb+");
248     if (adc_data_file == NULL) {
249         fprintf(stderr, "# Couldn't open file for writing: %s (did you
    remember to change OUTPUT_DATA?)\n", output_filename);
250   return 1;
251     }
252 
253     fwrite(&output_nominal_period_us, sizeof(double), 1, adc_data_file);
254     fwrite(val, sizeof(uint16_t), ADCS*num_samples, adc_data_file);
255     fclose(adc_data_file);
256     printf("# Data written to file. Program ended successfully.\n\n");
257 
258     gpioTerminate();
259     free(val);
260 
```

```
261     return 0;
262 }
```

Kode 3: C-kode brukt for å ta målinger med RPi skrevet av Kristoffer Kjørnes & Asgeir Bjørgan.

B Vedlegg B

```
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.
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     32.0
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     if len(sys.argv) > 1:
39         input_file = sys.argv[1]
40     else:
41         input_file = 'foo.bin'
42
43     sample_period, data = raspi_import(input_file)
44
45     # Specify the output file name
46     output_file = 'csv/clean_square.csv'
47
48     # Save the data to the output file
```

```
    np.savetxt(output_file, data, delimiter=',', fmt='%f')

    print(f"Data written to {output_file}")
```

Kode 4: Utdelt Python kode som konverterer målt data fra RPi til csv fil.

C Vedlegg C

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.signal import windows
from scipy.signal import butter, filtfilt

data = np.loadtxt('Lab/TTT4280/csv/200_hz.csv', delimiter=',')
signal = data[:, 0] # F rste kolonne fra 5 forskjellige ADCer
signal = signal - np.mean(signal)  # Fjerner DC offset
sampling_rate = 31250


def moving_average(signal, window_size=5):
    window = np.ones(window_size) / window_size
    return np.convolve(signal, window, mode='same')
signal = moving_average(signal)


def process_signal(window_type, beta=None, use_zero_padding=True, log_scale=False):
    if window_type == 'bartlett':
        window = windows.bartlett(len(signal))
    elif window_type == 'hann':
        window = windows.hann(len(signal))
    elif window_type == 'blackman':
        window = windows.blackman(len(signal))
    elif window_type == 'hamming':
        window = windows.hamming(len(signal))
    elif window_type == 'kaiser' and beta is not None:
        window = windows.kaiser(len(signal), beta)
    elif window_type == 'gaussian':
        std = len(signal) / 8
        window = windows.gaussian(len(signal), std)
    elif window_type == 'tukey':
        alpha = 0.5
        window = windows.tukey(len(signal), alpha)
    elif window_type == 'none':
        window = 1
    else:
        raise ValueError("Unknown window type")

    signal_windowed = signal * window

    if use_zero_padding:
        n_padded = 2**np.ceil(np.log2(len(signal_windowed)))
        signal_padded = np.pad(signal_windowed, (0, int(n_padded - len(signal_windowed))), 'constant')
        fft_result = np.fft.fft(signal_padded)
```

```
        fft_freq = np.fft.fftfreq(len(signal_padded), 1/sampling_rate)
    else:
        fft_result = np.fft.fft(signal_windowed)
        fft_freq = np.fft.fftfreq(len(signal_windowed), 1/sampling_rate)
    if log_scale:
        plt.xscale('log')
    else:
        plt.xscale('linear')

    fft_magnitude_db = 20 * np.log10(np.abs(fft_result))
    max_magnitude = np.max(np.abs(fft_result))
    fft_magnitude_db_relative = 20 * np.log10(np.abs(fft_result) /
    max_magnitude)

    return fft_freq, fft_magnitude_db_relative

min_freq = -2000
max_freq = 2000
zero_padding = False
log_scale = False

plt.figure(figsize=(14, 6))
plt.suptitle('Sammenligning av vindusfunksjoner uten zero-padding', fontsize
    =16)

# Hanning-vinduet
plt.subplot(1, 2, 1)
fft_freq, fft_magnitude_db = process_signal('hann', use_zero_padding=
    zero_padding, log_scale=log_scale)
plt.plot(fft_freq, fft_magnitude_db)
plt.title("Hanning-vindu")
plt.xlabel("Frekvens (Hz)")
plt.ylabel("Relativ Effekt (dB)")
plt.xlim([min_freq, max_freq])
plt.grid()

# Kaiser-vinduet
plt.subplot(1, 2, 2)
fft_freq, fft_magnitude_db = process_signal('kaiser', beta=0.1,
    use_zero_padding=zero_padding, log_scale=log_scale)
plt.plot(fft_freq, fft_magnitude_db)
plt.title("Kaiser-vindu")
plt.xlabel("Frekvens (Hz)")
plt.xlim([min_freq, max_freq])
plt.grid()

plt.legend()
plt.tight_layout(rect=[0, 0.03, 1, 0.95])
plt.show()

plt.figure(figsize=(14, 6))
plt.suptitle('Sammenligning av vindusfunksjoner uten zero-padding', fontsize
    =16)
# Bartlett-vinduet

plt.subplot(2,2,1)
fft_freq, fft_magnitude_db = process_signal('bartlett', use_zero_padding=
```

```
    zero_padding, log_scale=log_scale)
plt.plot(fft_freq, fft_magnitude_db)
plt.title("Bartlett-vindu")
plt.xlabel("Frekvens (Hz)")
plt.ylabel("Relativ Effekt (dB)")
plt.xlim([min_freq, max_freq])
plt.grid()

# Hanning-vinduet
plt.subplot(2, 2, 2)
fft_freq, fft_magnitude_db = process_signal('hann', use_zero_padding=
    zero_padding, log_scale=log_scale)
plt.plot(fft_freq, fft_magnitude_db)
plt.title("Hanning-vindu")
plt.xlabel("Frekvens (Hz)")
plt.xlim([min_freq, max_freq])
plt.grid()

# blackmann-vinduet
plt.subplot(2, 2, 3)
fft_freq, fft_magnitude_db = process_signal('blackman', use_zero_padding=
    zero_padding, log_scale=log_scale)
plt.plot(fft_freq, fft_magnitude_db)
plt.title("Blackman-vindu")
plt.xlabel("Frekvens (Hz)")
plt.xlim([min_freq, max_freq])
plt.grid()

# None
plt.subplot(2, 2, 4)
fft_freq, fft_magnitude_db = process_signal('none', use_zero_padding=
    zero_padding, log_scale=log_scale)
plt.plot(fft_freq, fft_magnitude_db)
plt.title("Uten vindu-funksjon")
plt.xlabel("Frekvens (Hz)")
plt.xlim([min_freq, max_freq])
plt.grid()

plt.legend()
plt.tight_layout(rect=[0, 0.03, 1, 0.95])
plt.show()

plt.figure(figsize=(14, 6))
plt.suptitle('Sammenligning av vindusfunksjoner uten zero-padding', fontsize
    =16)

# Hamming-vinduet
plt.subplot(2,2,1)
fft_freq, fft_magnitude_db = process_signal('hamming', use_zero_padding=
    zero_padding, log_scale=log_scale)
plt.plot(fft_freq, fft_magnitude_db)
plt.title("Hamming-vindu")
plt.xlabel("Frekvens (Hz)")
plt.ylabel("Relativ Effekt (dB)")
plt.xlim([min_freq, max_freq])
plt.grid()

```

```
# Kaiser-vinduet
plt.subplot(2, 2, 2)
fft_freq, fft_magnitude_db = process_signal('kaiser', beta=0.1,
    use_zero_padding=zero_padding, log_scale=log_scale)
plt.plot(fft_freq, fft_magnitude_db)
plt.title("Kaiser-vindu: beta=0.1")
plt.xlabel("Frekvens (Hz)")
plt.xlim([min_freq, max_freq])
plt.grid()

# Gaussian-vinduet
plt.subplot(2, 2, 3)
fft_freq, fft_magnitude_db = process_signal('gaussian', use_zero_padding=
    zero_padding, log_scale=log_scale)
plt.plot(fft_freq, fft_magnitude_db)
plt.title("Gaussian-vindu, std: len(signal) / 8")
plt.xlabel("Frekvens (Hz)")
plt.xlim([min_freq, max_freq])
plt.grid()

# Tukey
plt.subplot(2, 2, 4)
fft_freq, fft_magnitude_db = process_signal('tukey', use_zero_padding=
    zero_padding, log_scale=log_scale)
plt.plot(fft_freq, fft_magnitude_db)
plt.title("Tukey-vindu: alpha=0.5")
plt.xlabel("Frekvens (Hz)")
plt.xlim([min_freq, max_freq])
plt.grid()

plt.legend()
plt.tight_layout(rect=[0, 0.03, 1, 0.95])
plt.show()
```

Kode 5: Python kode som plotter det målte signalet fra RPi.