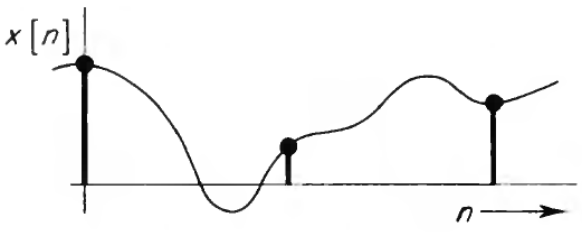
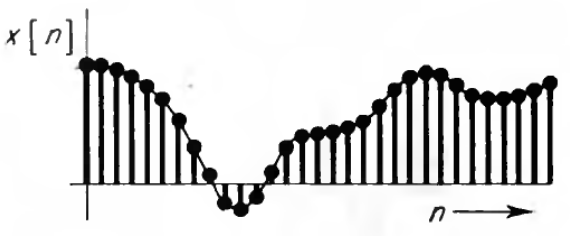
Digitale Signaal Bewerking

Practicum





HAN_UNIVERSITY OF APPLIED SCIENCES

Student : Niels Urgert (1654746)

Klas: ESE-3

Docent: ir drs E. J. Boks

Course : DSPL

Arnhem, **05-06-2024**

Inhoudsopgave

Figurenlijst

1. Inleiding

Het Digitale Signaal Bewerking (DSP) practicum had als doel het leren van de verschillende aspecten van digitaal signaalverwerking en de implementatie ervan in software. Het practicum bevat een reeks uit te voeren opdrachten, variërend van het ontwerpen van een ringbuffer tot het implementeren van filters en het genereren van frequentiebeelden.

Dit verslag biedt een overzicht van de uitvoering van het DSP practicum, inclusief de gebruikte methoden, de implementatie van de opdrachten en de behaalde resultaten.

2. Opdracht 1: Ringbuffer

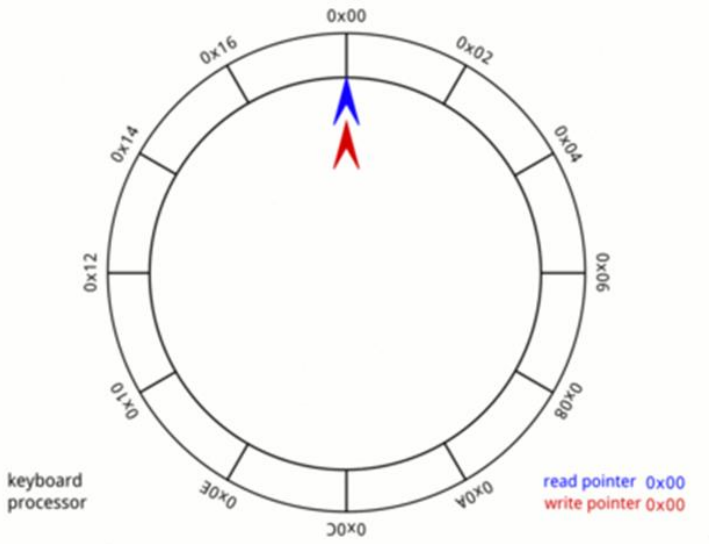
In opdracht 1 van het practicum moet er een aantal klassen worden geschreven voor de implementatie van de ringbuffer. De vereiste klassen zijn een ringbufferklasse en een complex getal klasse, die afgeleid is van de imaginaire getal klasse.

Een ringbuffer is een buffer dat een bepaalde hoeveelheid data kan opslaan zonder een begin of einde te hebben. In plaats daarvan wordt de data opgeslagen in een cirkelvormige structuur, waarbij nieuwe elementen worden toegevoegd aan het einde en oudere elementen worden verwijderd aan het begin. Hierdoor kan de ringbuffer continu nieuwe data opslaan, zelfs als alle beschikbare posities al zijn gebruikt (zie Figuur 1).

Een belangrijk kenmerk van een ringbuffer is dat het een beperkte grootte heeft. Wanneer nieuwe data wordt toegevoegd en de buffer vol is, zal het oudste element worden overschreven om ruimte te maken voor het nieuwe element. Dit betekent dat de ringbuffer altijd de meest recente data bevat binnen zijn capaciteit.

Het gebruik van een ringbuffer is vooral handig in situaties waarin een continue datastroom moet worden verwerkt en oudere data niet relevant is. Het wordt vaak gebruikt in toepassingen zoals audio- en videosignalen, waarbij real-time verwerking en een efficiënte geheugenopslag vereist zijn.

In figuur 2 wordt de ringbuffer, die in de software is geïmplementeerd, naar toegeschreven en uitgelezen.



Figuur 2 Ringbuffer

```
waarde [0] geschreven in buffer=[Re:2,000 Im:8,000]
waarde [1] geschreven in buffer=[Re:6,000 Im:17,000]
waarde [2] geschreven in buffer=[Re:14,000 Im:30,000]
waarde [3] geschreven in buffer=[Re:26,000 Im:47,000]
waarde [4] geschreven in buffer=[Re:42,000 Im:68,000]
waarde [5] geschreven in buffer=[Re:62,000 Im:93,000]
waarde [6] geschreven in buffer=[Re:86,000 Im:122,000]
waarde [7] geschreven in buffer=[Re:114,000 Im:155,000]
waarde [8] geschreven in buffer=[Re:146,000 Im:192,000]
waarde [9] geschreven in buffer=[Re:182,000 Im:233,000]
waarde [10] geschreven in buffer=[Re:222,000 Im:278,000]
waarde [11] geschreven in buffer=[Re:266,000 Im:327,000]
waarde [12] geschreven in buffer=[Re:314,000 Im:380,000]
waarde [13] geschreven in buffer=[Re:366,000 Im:437,000]
Buffer teruglezen. De buffer is 7 elementen groot.
Buffer element [n-2] = [Re:266,000 Im:327,000]
Buffer element [n-5] = [Re:146,000 Im:192,000]
Buffer element [2] = [Re:114,000 Im:155,000]
Buffer element [5] = [Re:222,000 Im:278,000]
buffer[n-0]=[Re:366,000 Im:437,000]
buffer[n-1]=[Re:114,000 Im:155,000]
buffer[n-2]=[Re:146,000 Im:192,000]
buffer[n-3]=[Re:182,000 Im:233,000]
buffer[n-4]=[Re:222,000 Im:278,000]
buffer[n-5]=[Re:266,000 Im:327,000]
buffer[n-6]=[Re:314,000 Im:380,000]
buffer[n-7]=[Re:366,000 Im:437,000]
buffer[n-8]=[Re:114,000 Im:155,000]
buffer[n-9]=[Re:146,000 Im:192,000]
buffer[n-10]=[Re:182,000 Im:233,000]
buffer[n-11]=[Re:222,000 Im:278,000]
buffer[n-12]=[Re:266,000 Im:327,000]
buffer[n-13]=[Re:314,000 Im:380,000]
```

Figuur 1 Output Ringbuffer

3. Opdracht 2: Exponentieel lopende -en lopende gemiddelde

In opdracht 2 moet het gemiddelde berekend worden van een reeks data van een ingelezen bestand. Deze data wordt berekend met twee versies van een lopend gemiddelde filter, waarbij de gebruiker het aantal punten in het filter kan instellen. Het originele signaal en het gemiddelde worden weergegeven in een grafisch venster. De twee lopende gemiddelde methoden zijn het lopende gemiddelde, zoals geïmplementeerd in opdracht 1 (Figuur 3) en een exponentieel lopend gemiddelde (Figuur 4).

Een lopend gemiddelde is een techniek om een reeks van gemiddelde waarden te berekenen uit een reeks van data punten. Dit gemiddelde wordt steeds berekend over een vaste omvang (aantal periodes) en verschuift telkens één periode. Het doel is om ruis in de data te verminderen en de onderliggende trend beter zichtbaar te maken. De methode wordt vaak gebruikt voor het gladstrijken van data om een algemene trend te identificeren. Dit filter is ook wel een FIR-filter (Finite Impulse Response) en de formule luidt:

- **y[n]** is het gefilterde signaal,
- **x[n]** is het originele signaal,
- **N** is het aantal punten in het gemiddelde (de lengte van het venster) en
- **n** is de positie in de tijdreeks (index).

$$y[n] = \frac{1}{N} \sum_{i=n-N+1}^{n} x[n]$$

De implementatie wordt weergegeven in figuur 3.

```
ttype som() const {
    ttype som = 0;
    for (unsigned short i = 0; i < aantal; i++)
        som += elementen[i];

    return som;
};

ttype gemiddelde() const {
    return (som() / aantal);
};
```

Figuur 3 Implementatie Lopend gemiddelde

Een exponentieel lopend gemiddelde is een type gewogen gemiddelde waarbij de recentere waarden in de datareeks een grotere invloed hebben op het gemiddelde dan oudere waarden. Dit wordt bereikt door een exponentiële weging toe te passen, waarbij de gewichten exponentieel afnemen naarmate de data verder in het verleden ligt. De methode wordt vaak gebruikt in financiële toepassingen zoals technische analyse, omdat het beter reageert op recente prijsveranderingen. In de software worden de horizontale lijnen, y-as waardes en de gegevens uit het bestand in de grafiek getekend. Het belangrijkste is het uitrekenen van de standaard en exponentiele gemiddelde waarde. Dit type filter is ook wel een IIR-filter (Infinite Impulse Response) en wordt als volgt gedefinieerd:

- **y[n]** is het gefilterde signaal,
- **x[n]** is het originele signaal,
- **α** is de gladmakingsfactor (hoe kleiner α, hoe meer vorige waarden meewegen)

$$\boldsymbol{y[n] = \propto \cdot x[n] + (1 - \propto) \cdot y[n-1]}$$

De implementatie wordt weergegeven in figuur 4.

```
float ExponentialAverageFilter::filter(const float input) {
    static float filteredValue = input;
    filteredValue = alfa * input + minalfa * filteredValue;
    return filteredValue;
}
```

Figuur 4 Implementatie Exponentieel lopend gemiddelde

Het verschil tussen de twee methode wordt in de onderstaande tabel samengevat

Kenmerk	Lopend Gemiddelde	Exponentieel Lopend Gemiddelde
Type	FIR-filter	IIR-filter
Geheugen	Houdt **N** vorige waarden bij	Slechts één vorige waarde
Vertraging	(N−1)/2 samples	Afhankelijk van **α**, minder dan bij FIR
Berekening	Gemiddelde over een venster	Recursief gewogen gemiddelde
Reactiesnelheid	Trager bij grote **N**	Sneller bij grote Recursief gewogen gemiddelde
Geschikt voor	Ruisonderdrukking	Adaptieve filtering en trenddetectie

De twee functies worden in figuur 5 toegepast. Afhankelijk van het geselecteerde filter, wordt deze weergegeven in de grafiek. De regelparameter kun je de weging van instellen.

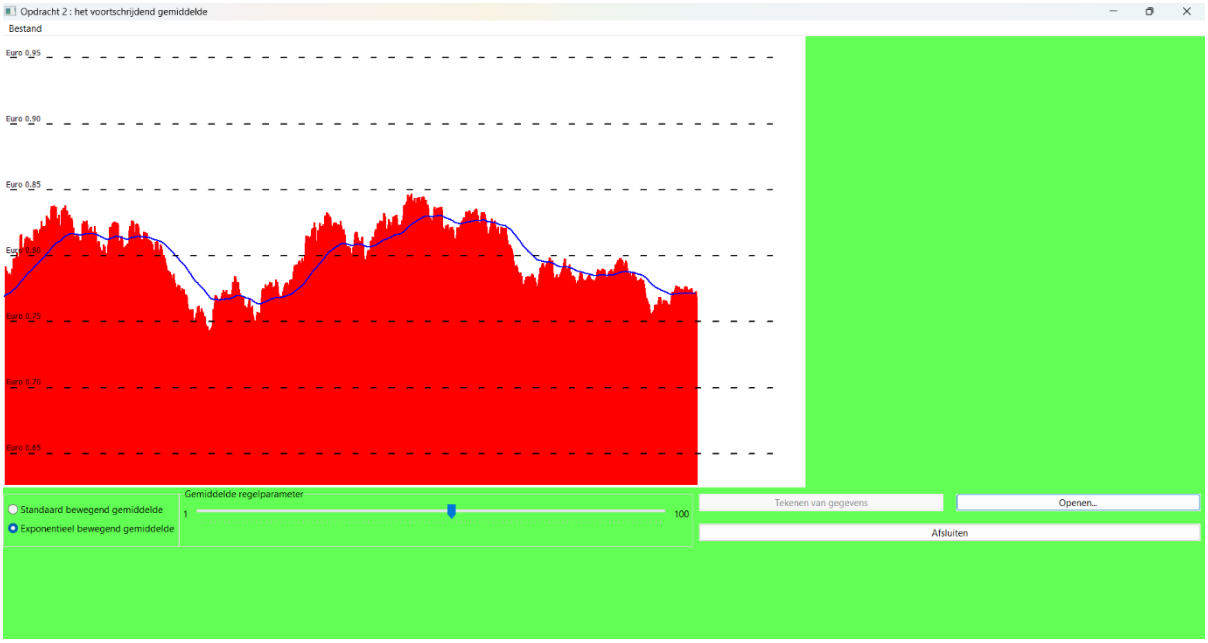
Voor het standaard lopend gemiddelde is de waarde van de slider direct de lengte ***N*** van het venster waarin het gemiddelde wordt berekend. Een grotere waarde betekent een sterker gladmakend effect, omdat dan meer waarden worden meegenomen.

Bij het exponentieel lopend gemiddelde wordt de waarde van de slider eerst bewerkt, zodat bij een lage regelparameter de **α**-waarde hoog is.

```
/* Filters verwerken */
const auto filterType = filterSelectionRadioBox->GetSelection();
int regelparameter = 101 - avgValueSlider->GetValue();

RingBuffer<double> ringBuffer(static_cast<unsigned short>(avgValueSlider->GetValue()));
ExponentialAverageFilter expFilter(1.0f / (static_cast<float>(regelparameter) + 1.0f));

// Tekengrafiek functie
auto tekenGrafiek = [&](auto& grafiekPunten, auto dataPunten) {
    for (unsigned short i = 0; i < data.GetCount(); i++) {
        const auto yAs = dataPunten(data.Item(i)) * scaleFactor;
        grafiekPunten.Add(wxPoint(static_cast<double>(i), yAs));
    }
    grafiek->zetTekenPen(wxPen(wxColour(wxT("BLUE")), 2, wxSOLID));
    grafiek->tekenSpline(grafiekPunten);
};

if (filterType == 0) {
    tekenGrafiek(averageSignalPoints, [&](double waarde) {
        return ringBuffer.gemiddelde(waarde);
        });
}
else {
    tekenGrafiek(expSignalPoints, [&](double waarde) {
        return expFilter.filter(waarde);
        });
}
```

Figuur 5 Implementatie van de Standaard en Exponentieel gemiddelde

De volledige werking van de software wordt in figuur 6 weergegeven. Hierin is de blauwe lijn het uitgerekende gemiddelde van de gegevens weergegeven in het rood. Met de slider wordt de zwaarte van de waardes geselecteerd.



Figuur 6 Output Exponentieel lopend gemiddelde

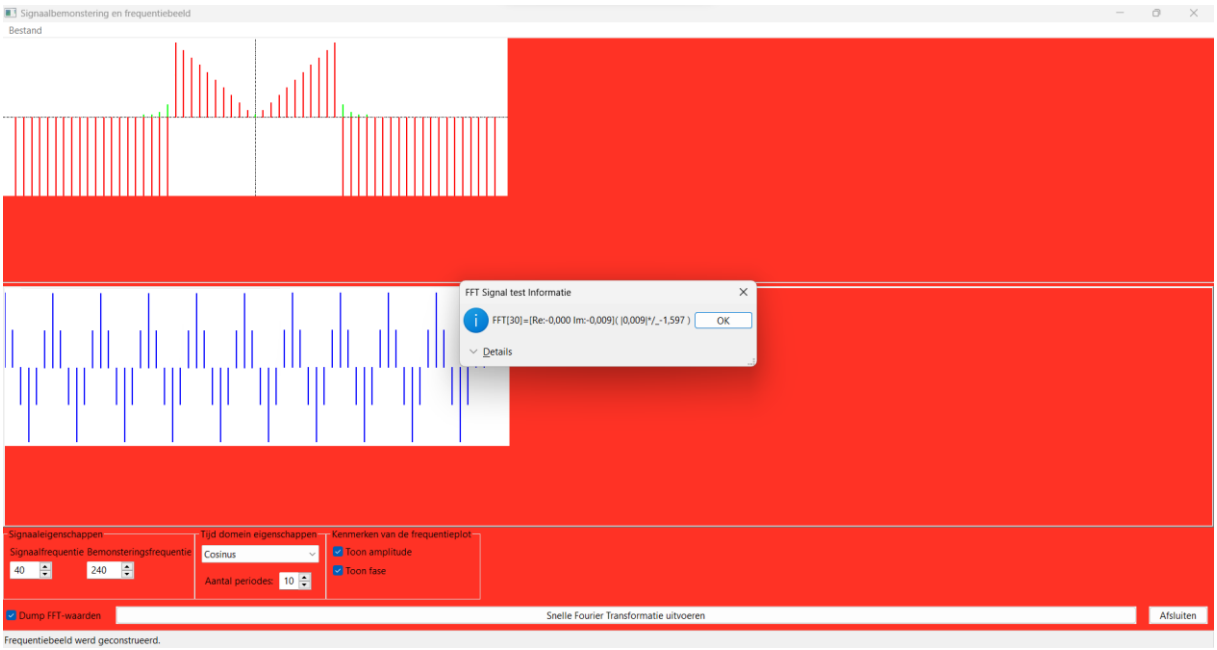
4. Opdracht 3: Fast Fourier Transform

Voor opdracht 3 is een Fast Fourier Transform (FFT) in het programma gemaakt om eenvoudig een testsignaal te kunnen instellen en dit signaal in het frequentiedomein te kunnen bekijken. Het testsignaal kan worden ingesteld op basis van de vorm (cosinus, driehoek, blokgolf of ingelezen uit een databestand), signaalfrequentie, bemonsteringsfrequentie en het aantal perioden. De code is geschreven met behulp van de FFTW toolkit om het frequentiebeeld te genereren.

De FFT is een efficiënte algoritme om een discrete Fourier-transformatie (DFT) van een reeks of een signaal te berekenen. Het vertaalt een tijd- of ruimtedomeinsignaal naar een frequentiedomeinrepresentatie. Dit betekent dat het de amplituden en fasen van de frequenties die aanwezig zijn in het signaal onthult. FFT wordt bijvoorbeeld gebruikt voor signaalverwerking om tijdsignalen in audiobewerking, beeldverwerking, en communicatie te analyseren en als filtering. In figuur 8 wordt de output weergegeven. De bovenste grafiek is het frequentiedomein en de onderste het tijdsdomein.

```
double* input;
fftw_complex* output;
fftw_plan p;

input = (double*)fftw_malloc(signaal.GetCount() * sizeof(double));
output = (fftw_complex*)fftw_malloc((signaal.GetCount() * sizeof(fftw_complex)) / 2 + 1);

for (auto i = 0; i < signaal.GetCount(); i++) {
	input[i] = signaal.Item(i);
}

p = fftw_plan_dft_r2c_1d(signaal.GetCount(), input, output, FFTW_PRESERVE_INPUT + FFTW_ESTIMATE);
fftw_execute(p);
```

Figuur 7 Implementatie van FFT d.m.v. FFTW toolkit



Figuur 8 Output FFT

5. Opdracht 4: Finite Impulse Response – Filter

In opdracht 4 is in het programma, de code gemaakt om een FIR-filter (Finite Impulse Response) te kunnen instellen om bepaalde signaalbanden door te laten of te blokkeren. De gebruiker kan parameters zoals de bemonsteringsfrequentie, start- en stopfrequenties van de doorlaatband, filterorde, versterkingsfactor, vensterfunctie en het aantal bits voor de fixed-point codering instellen. De applicatie moet FIR-coëfficiënten kunnen genereren op basis van de opgegeven frequentierespons, waarbij de coëfficiënten eerst in floating-point formaat worden berekend en vervolgens worden omgezet naar fixed-point (Qx-formaat). De rechthoek, driehoek en hamming functies volgens figuur 9, 10 en 12 zijn verwerkt in de software (zie Figuur 11 en 13). Hierin is **n** het sample en **M** de orde grootte.

$$w[n] = 1$$

Figuur 9 Rechthoek venster (Dirichlet)

$$w[n] = 1 - \frac{|n|}{M} \qquad -M \leq n \leq M$$

Figuur 10 Driehoek venster (Barlett) [1] (5.19)

```
double FilterVenster::driehoek(const Int32 n ) const {
    return std::max(0.0, (orde - fabs(n)) / orde);
}
```

Figuur 11 Implementatie van Driehoek venster

$$w[n] = 0,54 + 0,46 \cos\left(\frac{n\pi}{M}\right) \qquad -M \leq n \leq M$$

Figuur 12 Hamming venster [1] (5.22)

```
double FilterVenster::hamming(const Int32 n ) const {
    return 0.54 + 0.46 * cos((n * Pi) / orde);
}
```

Figuur 13 Implementatie van Hamming venster

Een FIR-filter is een type digitale filter dat wordt gebruikt in signaalverwerking om specifieke frequentiecomponenten van een signaal te bewerken of te verwijderen. Het wordt bijvoorbeeld toegepast voor audiobewerking om ruis te verwijderen, te filteren of bepaalde frequenties te versterken of verzwakken. Ook in ECG-machines wordt het gebruikt om ruis en artefacten uit hartsignalen te filteren. In Figuur 14 wordt de wiskundige formule van een FIR-filter weergegeven.

$$y[n] = \sum_{i=0}^{n-1} h[n] \cdot x[n-1]$$

Figuur 14 Wiskundige formule FIR-filter

De formule uit Figuur 14 is gebruikt voor het implementeren van het FIR-filter in de software. In Figuur 15 wordt het uiteindelijke implementatie van het FIR-filter weergegeven.

```
void FilterFirInt16::reset()
{
    filterMemory.reset();
}

Int16 FilterFirInt16::filter(const Int16 input)
{
    filterMemory.schrijf(input);
    Int64 som = 0;
    for (size_t i = 0; i < filterCoeffs.geefAantal(); i++)
    {
        const float coeff = filterCoeffs[i];
        const int16_t current_value = filterMemory.lees();
        som = som + coeff * current_value;
    }
    som /= scaleFactor;

    const Int16 result = static_cast<Int16>(som);

    return result;
}
```

Figuur 15 Implementatie van het FIR-filter

Figuur 16 Output FIR-Filter

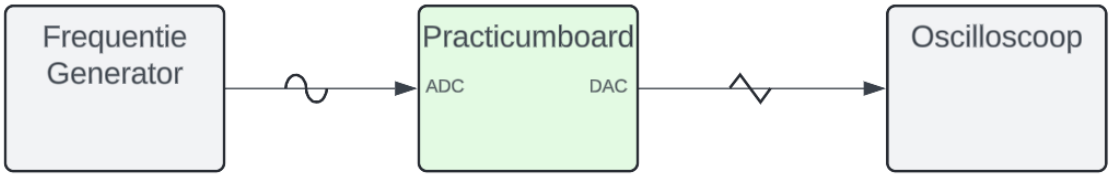
De extra optie voor het weergeven van de coördinaten in het tijdsdomein en het frequentiedomein is in de software geïmplementeerd voor de bonuspunten. De implementatie hiervan wordt in figuur 13 weergegeven.

```
void FilterVenster::tijdViewMuisBewegingHandler(wxMouseEvent &event)
{
#ifndef ExtraOpties
    event.Skip();
#else
    if (true == tijdDomeinCoords->IsEnabled()) {
        const wxPoint mouseCoord(tijdDomeinGrafiek-
>converteerMuisPositie(const_cast<wxMouseEvent&>(event)));
        int width = tijdDomeinGrafiek->GetClientSize().GetWidth();
        int n = std::round(((mouseCoord.x / static_cast<double>(width)) * 2.0) * orde);

        if (n >= -orde && n <= orde) {
            size_t coeffIndex = static_cast<size_t>(orde + n);

            if (coeffIndex < filterCoeffs.size()) {
                float sampleValue = berekenFloatingPoint(filterCoeffs[orde + n]);
                wxString info = wxString::Format("Coefficient h[%d] = %.4f", n, sampleValue);
                tijdDomeinCoords->SetLabel(info);
            }
        }
    }
#endif
}

void FilterVenster::freqViewMuisBewegingHandler(wxMouseEvent &event)
{
#ifndef ExtraOpties
    event.Skip();
#else
    if (true == freqDomeinCoords->IsEnabled()) {
        const wxPoint muiscoord(freqDomeinGrafiek->converteerMuisPositie(const_cast<wxMouseEvent
&>(event)));
        double omega = muiscoord.x * (Pi / freqDomeinGrafiek->GetClientSize().GetWidth());
        double dB = muiscoord.y * (H_Omega_max - H_Omega_min) / freqDomeinGrafiek-
>GetClientSize().GetHeight();

        if (omega >= 0 && omega <= Pi) {
            int index = static_cast<int>((omega / Pi) * FreqSpectrumPunten(taps));

            if (index >= 0 && index < H_Omega.size()) {
                double filterEffect = H_Omega[index];
                wxString info = wxString::Format("|H(%.4f*pi)| = %.2f dB at [%.4f*pi, %.2f dB]",
omega / Pi, filterEffect, omega / Pi, dB);
                freqDomeinCoords->SetLabel(info);
            }
        }
    }
#endif
}
```

Figuur 17 Implementatie van de extra muisbeweging opties

6. Opdracht 5: Implementatie van het FIR – Filter op het practicumboard

In opdracht 5 moet de FIR-filter toepast worden op het practicumboard. De filterwaarden en het ontworpen FIR-filter uit opdracht 4 zijn nodig voor opdracht 5. Voor de opdracht hoeft alleen in de klasse genaamd STM32FilterApp, de juiste bemonsteringsfrequentie, ADC- en DAC-instellingen worden toegepast om het filter op het practicumboard te testen.



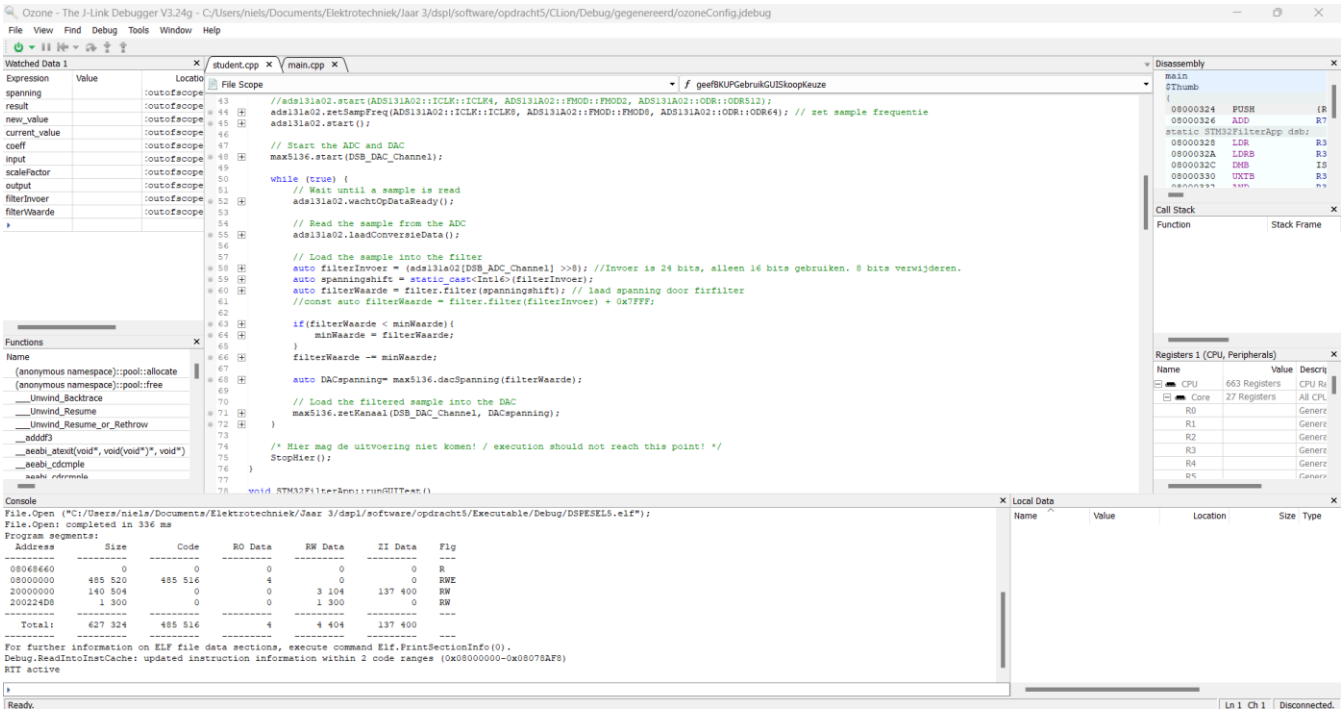
Figuur 18 Blokdiagram opstelling

De bemonstering frequentie van de ADC is ingesteld op 4 KHz. De ADC en DAC worden opgestart. De ADC wordt uitgelezen. De waarde uit de ADC wordt bewerkt door het filter, waarna de waarde uit het filter naar de DAC gestuurd.

```
void STM32FilterApp::runFilter()
{
    ads131a02.zetSampFreq(ADS131A02::ICLK::ICLK8, ADS131A02::FMOD::FMOD8, ADS131A02::ODR::ODR64);
    ads131a02.start(DSB_ADC_Channel);
    max5136.start(DSB_DAC_Channel);

    while(1) {
        ads131a02.wachtOpDataReady();   // Wacht op data van de ADC
        ads131a02.laadConversieData();  // Laad data in ADCdata

        // Bitgrootte is 16 bits. Invoer is 24 bits, verwijder 8 bits.
        Int16 ADCspanning =  static_cast<Int16>(ads131a02[DSB_ADC_Channel] >> 8);

        // Voer het signaal door de FIR-filter
        Int16 filterwaarde = filter.filter(ADCspanning);

        // Normaliseer filterwaarde om negatieve output te voorkomen
        if(filterwaarde < minWaarde){
            minWaarde = filterwaarde;
        }
        filterwaarde -= minWaarde;

        // Converteer filterwaarde naar een bruikbare DAC-spanning
        UInt16 DACspanning = max5136.dacSpanning(filterwaarde);

        // Stuur de spanning naar de DAC
        max5136.zetSpanning(DSB_DAC_Channel, DACspanning);
    }

    /* Hier mag de uitvoering niet komen! / execution should not reach this point! */
    StopHier();
```

Figuur 19 Implementatie van ADC en DAC

Voor de ontwikkeling van de software wordt gebruik gemaakt van de Jetbrains CLion-ontwikkelomgeving en voor het programmeren en debuggen van de embedded software op het practicumboard wordt Segger Ozone gebruikt.



Figuur 20 Ozone GUI opdracht 5

7. Aanpassingen

Na aanleiding van een aantal fouten en foutmeldingen in de aangeleverde softwarebestanden van DSP zijn een aantal extra aanpassingen gemaakt.

7.1 Opdracht 4

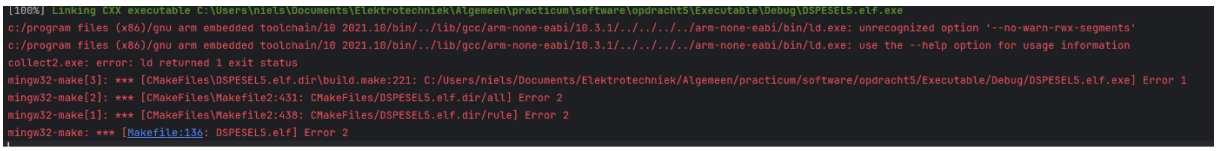
In het commentaar bij de omschrijving wordt gezegd: “h[n] = (Omega1/pi)*sinc(Omega1*n) : Dit is een laagdoorlaatfilter. vermenigvuldig dit in tijddomein met cos(Omega0*n)”

Bij het implementeren van de laagdoorlaat filter, zoals beschreven, komen er onjuiste waarde uit.
result = (omega1 / Pi) * sinc(omega1 * n) * cos(omega0 * n);

De code is anders geschreven, waardoor de sinc functie overbodig is geworden, namelijk:
result = (1 / (n * Pi)) * sin(omega1 * n) * cos(omega0 * n);

7.2 Opdracht 5

In de bestanden van opdracht 5 zijn een aantal software bestanden aangepast, anders kon opdracht 5 niet gecompileerd kon worden. De Makefile was de fout dat de optie **–no-warn-rwx-segments** een unrecognized option was. Deze regel is weggehaald uit de code, zie Figuur 21 en BasisSTM32.cmake.



Figuur 21 Foutmelding opdracht 5

BasisSTM32.cmake

```
# Verwijdert: error unrecognized option '–no-warn-rwx-segments'
#set(StandaardLinkVlaggen "--specs=nano.specs -mthumb -Wl,--gc-sections -
Wl,-Map,${PROJEKTNAAM}.map -nostartfiles -Wl,--no-warn-rwx-segments")
```

Als tweede was er de error dat de functie **xPortSysTickHandler();** nergens anders in de software gedefinieerd was. Dit gaf een foutmelding. Regel weggehaald.

cmsis_os.c

```
//Verwijdert: error nergens anders in de software gedefinieerd
 //xPortSysTickHandler();
```

Verder waren er meerdere definities van init. Deze **init();** functie is weggehaald.

syscalls.cpp

```
//Verwijderd: error meerdere definities van init
/*
void _init()
{

}*/
```

8. Bibliografie

[1] P. A. Lynn en W. Fuerst, Introductory Digital Signal Processing with computer applications, John Wiley & Sons, 1989.