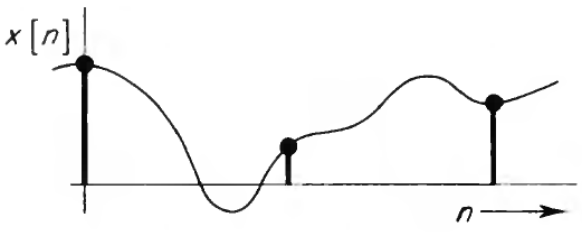
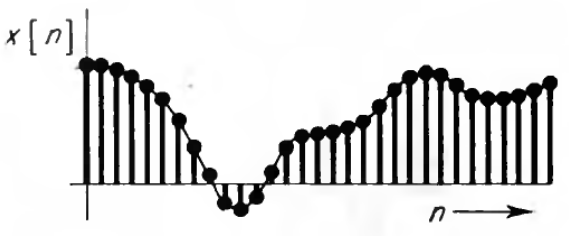
Digital Signal Processing

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 Digital Signal Processing (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.

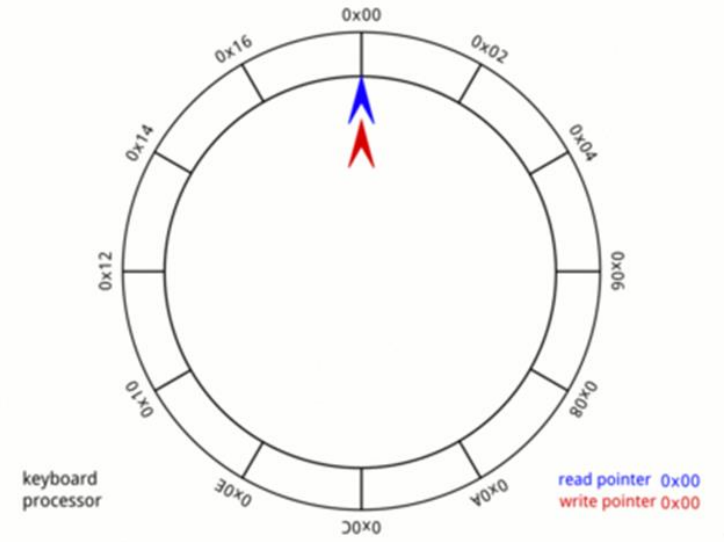
Een ringbuffer is een buffer dat een bepaalde hoeveelheid data kan opslaan zonder een begin of einde te hebben. In plaats daarvan wordt de data opgeslagen in een cirkelvormige structuur, waarbij nieuwe elementen worden toegevoegd aan het einde en oudere elementen worden verwijderd aan het begin. Hierdoor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

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.

```
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 2 Output Ringbuffer



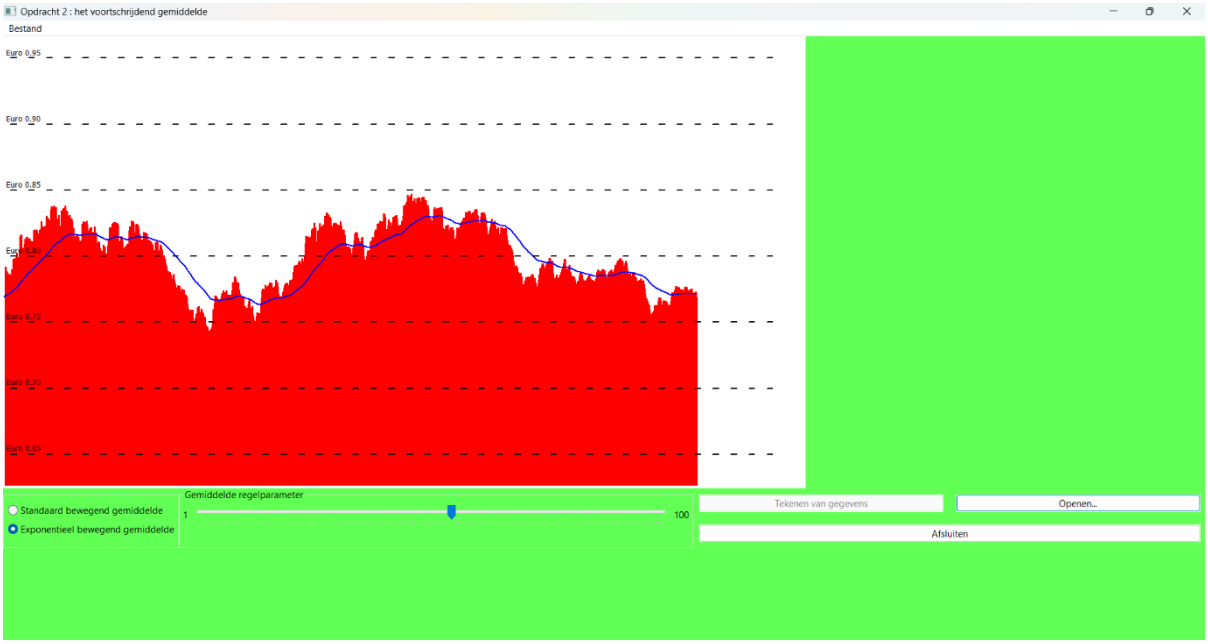
Figuur 1 Ringbuffer

3. Opdracht 2: Exponentieel lopende -en lopende gemiddelde

In opdracht 2 moet de 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 en een exponentieel lopend gemiddelde.

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.

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.

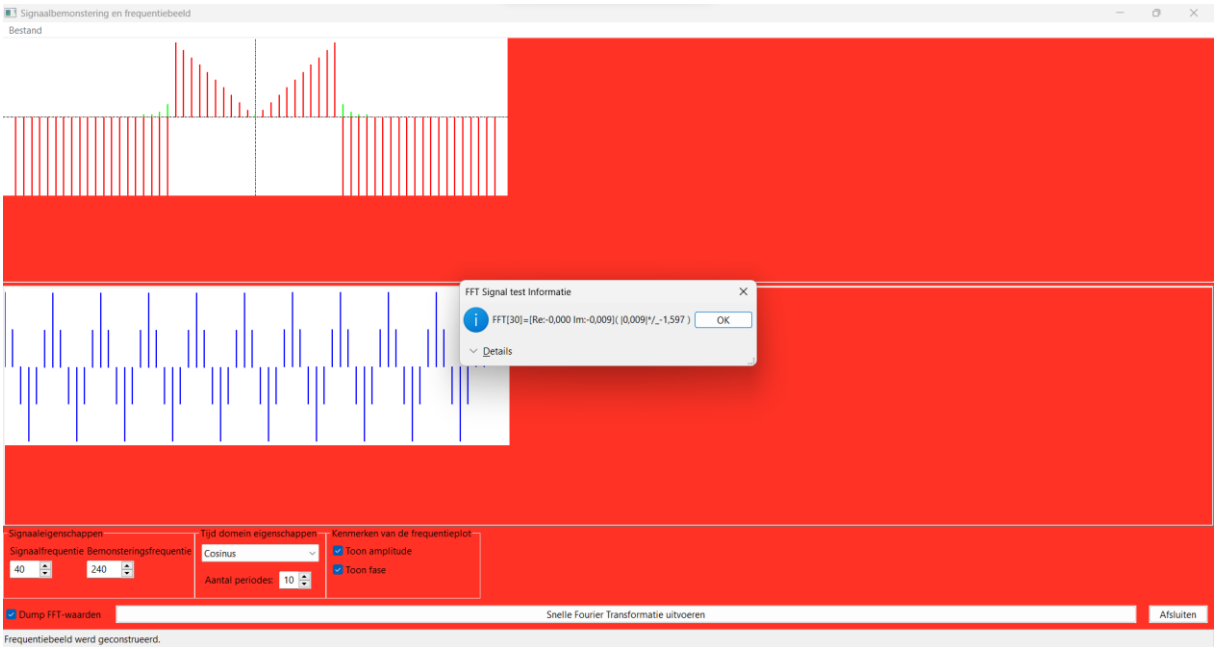


Figuur 3 Output Expontieel lopend gemiddelde

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.



Figuur 4 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 5, 6 en 7 zijn verwerkt in de software.

$$w[n] = 1.$$

Figuur 5 Rechthoek window (Dirichlet)

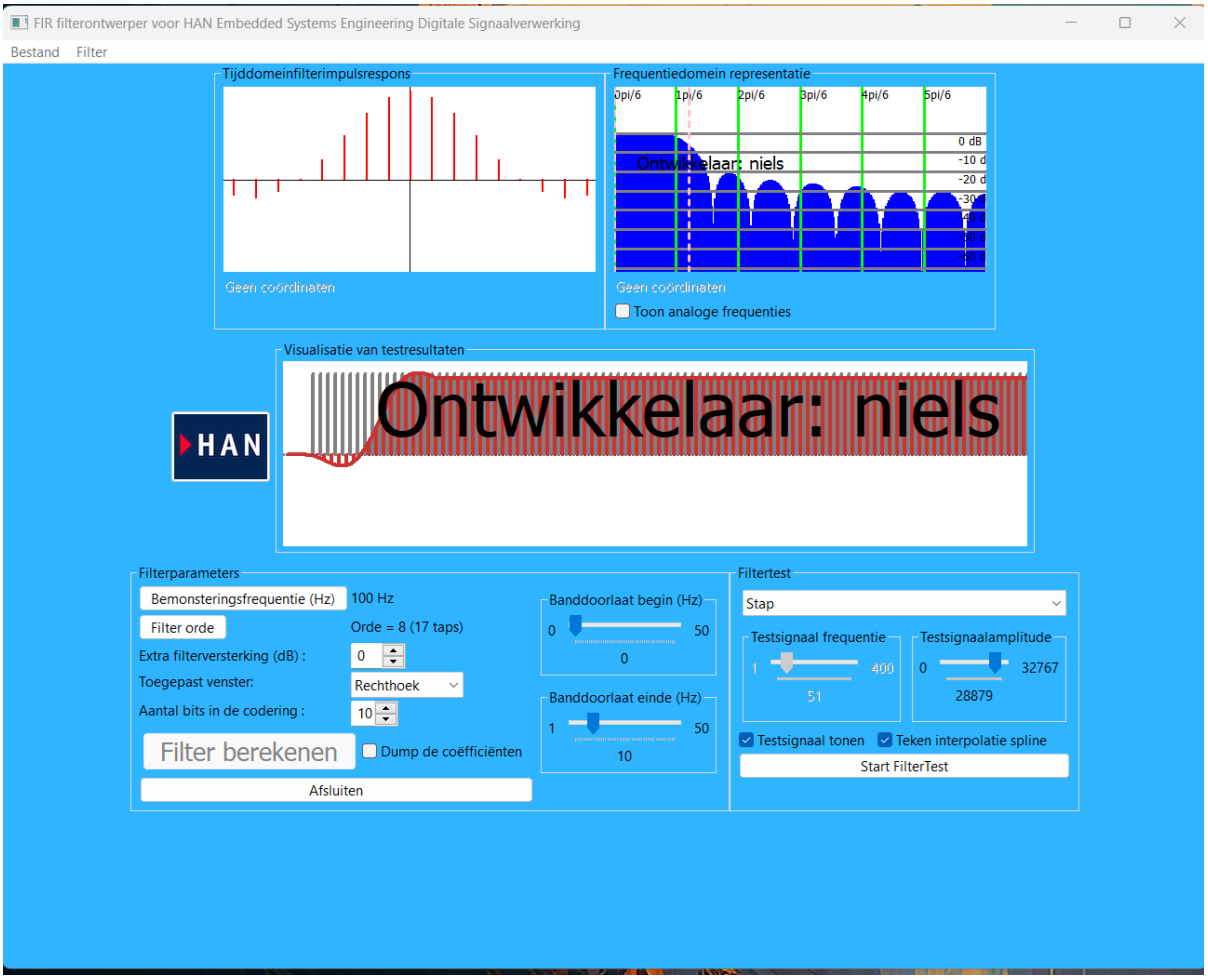
$$w[n] = \frac{(M+1)-|n|}{(M+1)^2}, \quad -M \le n \le M$$

Figuur 6 Driehoek window (Barlett) [1]

$$w]n] = 0.54 + 0.46 \cos\left(\frac{n\pi}{M}\right), \qquad -M \le n \le M$$

Figuur 7 Hamming window [1]

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.

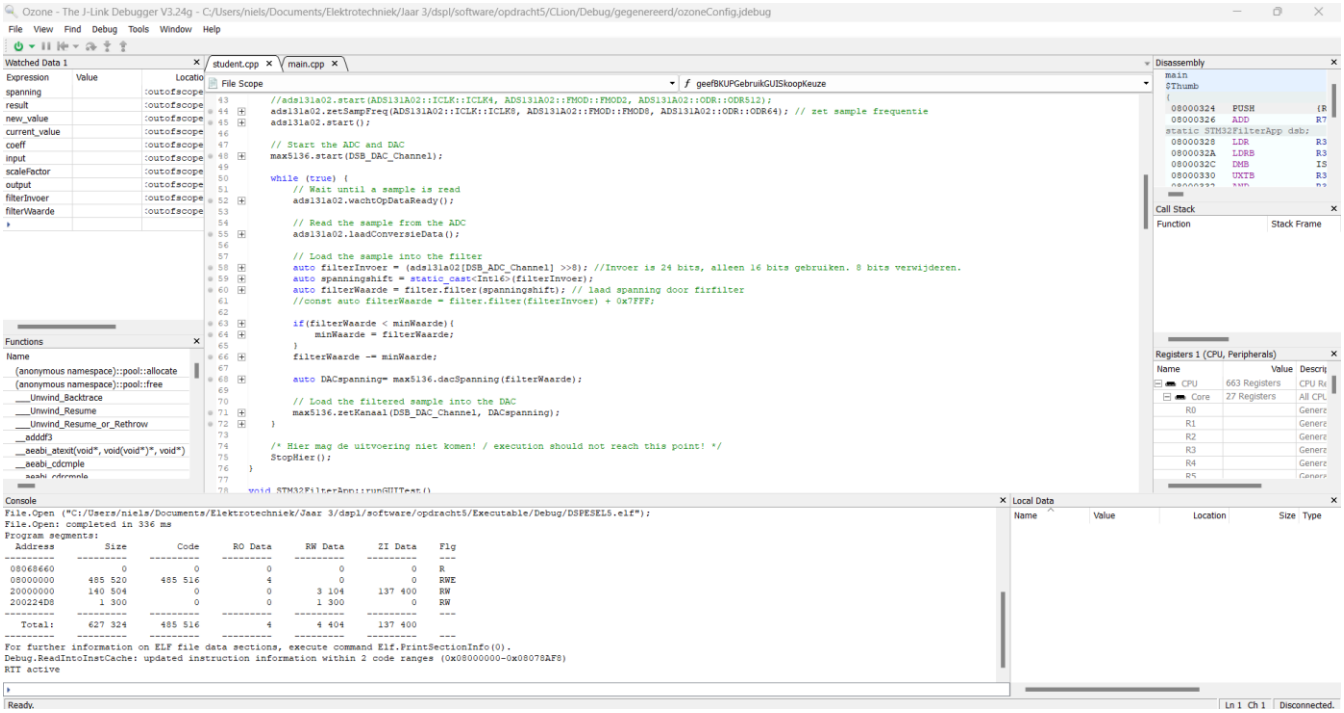


Figuur 8 Output FIR-Filter

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 de filter op het practicumboard te testen.

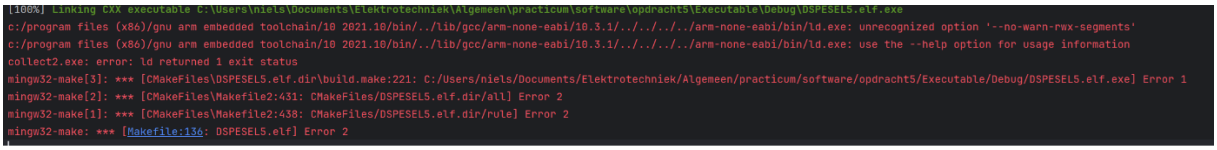
Voor de ontwikkeling van de software wordt gebruik gemaakt van de Jetbrains CLion-ontwikkelomgeving en voor het programmeren en debuggen van de embedded software op het practicumboard wordt Segger Ozone gebruikt.



Figuur 9 Ozone GUI opdracht 5

6.1 Aanpassingen

Na aanleiding van een aantal foutmeldingen in de software bestanden van DSP zijn een aantal aanpassingen gemaakt, zodat opdracht 5 gecompileerd kon worden.



Figuur 10 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")
```

cmsis_os.c

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

syscalls.cpp

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

}*/
```

7. Bibliografie

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

Bijlage A: Programlisting

Opdracht 2

```
float ExponentialAverageFilter::filter(const float input)
{
    static float filteredValue = input; // initialize filteredValue to input on the first call
    filteredValue = alfa * input + minalfa * filteredValue; // compute the filtered value
    return filteredValue; // return the filtered value
}
```

Figuur A.1 expAverage.cpp

```
void GemWaardeVenster::drawDataHandler(wxCommandEvent& event)
{
    const auto filterType = filterSelectionRadioBox->GetSelection();

    auto regelparameter = avgValueSlider->GetValue();
    int u = 0;
    for (int x = 1; x <= regelparameter; ++x) {
        u = 101 - x;
    }
    regelparameter = u;


    ExponentialAverageFilter expFilter(1.0f / (static_cast<float>(regelparameter) + 1.0f));

    RingBuffer<double> ringBuffer(static_cast<unsigned short>(avgValueSlider->GetValue()));

    LijnLijst lineList;
    PuntLijst originalSignalPoints;
    PuntLijst averageSignalPoints;
    PuntLijst expSignalPoints;
    auto iterator = data.begin();

    grafiek->maakSchoon();
    grafiek->zetOffset(wxPoint(2, 2));

    // Add horizontal lines to the graph
    for (int yAs = 10, val = 70; yAs < grafiek->GetMaxHeight() / 5; yAs += 20, val += 5) {
        for (int spatie = 10, xAs = 0; xAs < grafiek->GetMaxWidth(); xAs += 30)
            lineList.Add(LijnStuk(wxPoint(xAs + spatie, yAs), wxPoint(xAs + 20, yAs)));
    }
```

Figuur A.2 gemWaarde.cpp (deel 1)

```
    // Add original signal points to variable originalSignalPoints
    for (double xAs = 0.0f; iterator < data.end(); iterator++, xAs++) {
        const auto yAs = ((*iterator - 0.0f) * (1800.0f - 2.0f) / (1.0f - 0.0f) + 2.0f);
        originalSignalPoints.Add(wxPoint(xAs, yAs));
    }

    grafiek->zetTekenPen(wxPen(wxColour(wxT("RED")), 2, wxSOLID));
    grafiek->tekenStaven(originalSignalPoints, false, false);

    if (filterType == 0) {
        // Add average signal points to variable averageSignalPoints using ringBuffer
        for (unsigned short i = 0; i < data.GetCount(); i++) {
            const auto yAs = ((ringBuffer.gemiddelde(data.Item(i)) - 0.0f) * (1800.0f - 2.0f)
/ (1.0f - 0.0f) + 2.0f);
            averageSignalPoints.Add(wxPoint(static_cast<double>(i), yAs));
        }
        grafiek->zetTekenPen(wxPen(wxColour(wxT("BLUE")), 2, wxSOLID));
        grafiek->tekenSpline(averageSignalPoints);
    }
    else {
        iterator = data.begin();
        // Add exponential signal points to variable expSignalPoints using expFilter
        for (double xAs = 0.0f; iterator < data.end(); iterator++, xAs++) {
            const auto yAs = ((expFilter.filter(*iterator) - 0.0f) * (1800.0f - 2.0f) / (1.0f
- 0.0f) + 2.0f);
            expSignalPoints.Add(wxPoint(xAs, yAs));
        }
        grafiek->zetTekenPen(wxPen(wxColour(wxT("BLUE")), 2, wxSOLID));
        grafiek->tekenSpline(expSignalPoints);
    }

    grafiek->zetTekenPen(wxPen(wxColour(wxT("BLACK")), 2, wxSOLID));
    grafiek->tekenLijnen(lineList, true);

    // Add labels to the graph
    grafiek->zetKleineTekst(wxString(wxT("Euro 0.65")), wxPoint(5, 72));
    grafiek->zetKleineTekst(wxString(wxT("Euro 0.70")), wxPoint(5, 177));
    grafiek->zetKleineTekst(wxString(wxT("Euro 0.75")), wxPoint(5, 282));
    grafiek->zetKleineTekst(wxString(wxT("Euro 0.80")), wxPoint(5, 387));
    grafiek->zetKleineTekst(wxString(wxT("Euro 0.85")), wxPoint(5, 492));
    grafiek->zetKleineTekst(wxString(wxT("Euro 0.90")), wxPoint(5, 597));
    grafiek->zetKleineTekst(wxString(wxT("Euro 0.95")), wxPoint(5, 702));

    /* laat dit hieronder staan. / Leave this statement in place. */
    gemVeranderd = false;
}
```

Figuur A.3 gemWaarde.cpp (deel 2)

Opdracht 3

```
void SignaalVenster::tekenReeksHandler(wxCommandEvent &event)
{
    const UInt32 aantalSampPerPeriod = sampFreq / sigFreq;

    venster_statusbar->SetStatusText(_("FFTW start of calcation"));

    if (false == (aantalSampPerPeriod >1))
            wxLogError(_("The signal could not be sampled, because")+wxT(" \n")+
                   wxString::Format(_("the number of samples per period=%d"),
aantalSampPerPeriod));
    else
    {
        PuntLijst punten; /* wxArray van wxPoints */

        if (SignaalType::DataBestand != signalChoice) //sigKeuze
        {
            const wxCoord amplitude = 1024; //32768;   /* amplitude = 80% van scherm */
            const double hoekFreq = 2 * Pi * sigFreq / sampFreq;
            auto normHoek = 0.0;

            UInt32 stap = 0;

            signaal.Clear();

#if defined(InterfaceTaalNederlands)

            for (auto hoek = 0.0; hoek< 2 *Pi* aantalPerioden; hoek += hoekFreq)
#else
            const auto angularFreq = 2.0 * Pi*sigFreq / sampFreq;
            for (auto angle = 0.0; angle< 2 *Pi* aantalPerioden; angle += angularFreq)
#endif
            {
                auto signalValue = 0.0f;

                switch (signalChoice)
                {
                    case SignaalType::Cosinus:
                            signalValue = static_cast<double>(amplitude) * cos(hoek);
                        break;

                    case SignaalType::Driehoek:
                            signalValue = static_cast<double>(amplitude) * (((2 / Pi) *
asin(sin(hoek))));
                        break:
```

Figuur A.4 signaal.cpp (deel 1)

```
                    break;

                case SignaalType::Blokgolf: //std::sin(normHoek) >= 0
                        signalValue = static_cast<double>(amplitude) * ((sin(hoek) >=
0.0) ? 1.0 : -1.0);
                    break;
                case SignaalType::DataBestand:
                    signalValue = -1.234f;
                    wxFAIL_MSG(_("Not allowed."));
                    break;

                default:
                    wxFAIL_MSG(wxString::Format(_("(SignaalVenster::tekenReeksHandler)
Wrong signal choice (%ld)."),
                                                static_cast<UInt32>(signalChoice)));
                    return;
                    break;
            }
            signaal.Add(signalValue);
            punten.Add(wxPoint(stap++, static_cast<int>(signalValue))); //* 100 ?
        }
        signaal.Shrink();
    }
    else
    {
        if (0 == signaal.GetCount())
        {
            wxLogError(_("No valid data present in the file!"));
            return;
        }
        else
        {
            for (auto i = 0; i<signaal.GetCount(); i++)
                punten.Add(wxPoint(i, static_cast<int>(100 * signaal[i])));

        }
    }

    wxLogDebug(_("Number of values=%d"), static_cast<UInt32>(signaal.GetCount()));

    signaalGrafiek->maakSchoon();
    signaalGrafiek->tekenStaven(punten, true);
```

Figuur A.5 signaal.cpp (deel 2)

```
        /* Calculate FFT */
        Complex SigComplex;
        PolairGetal SigPolair;
        PuntLijst AmplitudePunten;
        PuntLijst FasePunten;

        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);
        //assert(p == nullptr);
        fftw_execute(p);

        for (auto i = 0, minI = 0; i < (signaal.GetCount() / 2 + 1); i++) {
            //fftw_execute(p);
            auto real = output[i][0];
            auto imaginair = output[i][1];

            SigComplex = Complex(real, imaginair);
            SigPolair = PolairGetal(SigComplex);
            AmplitudePunten.Add(wxPoint(i, SigPolair.Mag() * 100));
            AmplitudePunten.Add(wxPoint(minI, SigPolair.Mag() * 100));

            if (SigPolair.Mag() < faseToonGrens) {
                FasePunten.Add(wxPoint(i, 0));
                FasePunten.Add(wxPoint(minI, 0));
            }
            else {
                FasePunten.Add(wxPoint(i, SigPolair.Arg() * 100));
                FasePunten.Add(wxPoint(minI, SigPolair.Arg() * 100));
            }

            minI--;
            wxLogDebug(_("FFT[%d] = |%lf|/_%lf"), i, SigPolair.Mag(), SigPolair.Arg());
        }
```

Figuur A.6 signaal.cpp (deel 3)

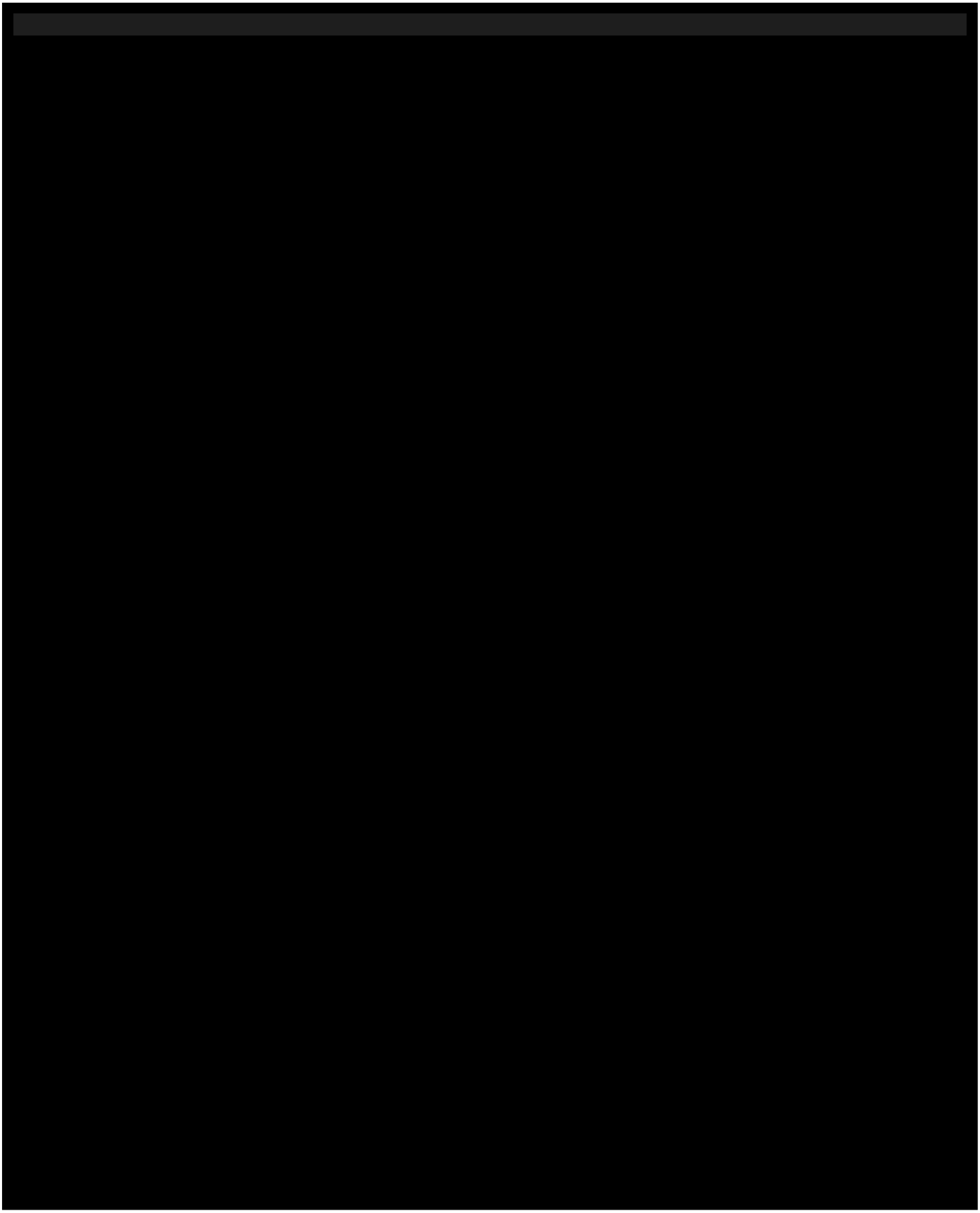
```
        fftwGrafiek->maakSchoon();
        fftwGrafiek->zetTekenPen(wxPen(wxColour(wxT("BLACK")), 2, wxSOLID));
        fftwGrafiek->tekenAssenstelsel();

        if (ampCheckBox->GetValue() && faseCheckBox->GetValue()) {
            fftwGrafiek->zetTekenPen(wxPen(wxColour(wxT("GREEN")), 2, wxSOLID));
            fftwGrafiek->tekenStaven(AmplitudePunten, true);
            fftwGrafiek->zetTekenPen(wxPen(wxColour(wxT("RED")), 2, wxSOLID));
            fftwGrafiek->tekenStaven(FasePunten, true);
        }
        else if (ampCheckBox->GetValue()) {
            fftwGrafiek->zetTekenPen(wxPen(wxColour(wxT("GREEN")), 2, wxSOLID));
            fftwGrafiek->tekenStaven(AmplitudePunten, true);
        }
        else if (faseCheckBox->GetValue()) {
            fftwGrafiek->zetTekenPen(wxPen(wxColour(wxT("RED")), 2, wxSOLID));
            fftwGrafiek->tekenStaven(FasePunten, true);
        }

        fftw_destroy_plan(p);
        fftw_free(input);
        fftw_free(output);

        venster_statusbar->SetStatusText(_("Frequency image was constructed."));
    }
}
}
```

Figuur A.7 signaal.cpp (deel 4)

Opdracht 4

```
void FilterFirInt16::reset()
{
    int i = scaleFactor;
    if(i > 1)
        return;

    filterMemory.reset();
    filterCoeffs.reset();
}

/* Implementatie van het filter */
Int16 FilterFirInt16::filter(const Int16 input)
{
    filterMemory.schrijf(input);
    Int64 som = 0;
    /* Start of student code */
    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 A.8 firfilter.cpp



Figuur A.9 filterDesigner.cpp (deel 1)

Opdracht 5

```
void STM32FilterApp::runFilter()
{
    //ADCData sampleData;
    // Set the sampling frequency of the ADC 16,38MHz / 2 / 4 / 521 = 4KHz
    //ads131a02.start(ADS131A02::ICLK::ICLK4, ADS131A02::FMOD::FMOD2,
ADS131A02::ODR::ODR512);
    ads131a02.zetSampFreq(ADS131A02::ICLK::ICLK8, ADS131A02::FMOD::FMOD8,
ADS131A02::ODR::ODR64); // zet sample frequentie
    ads131a02.start();

    // Start the ADC and DAC
    max5136.start(DSB_DAC_Channel);

    while (true) {
        // Wait until a sample is read
        ads131a02.wachtOpDataReady();

        // Read the sample from the ADC
        ads131a02.laadConversieData();

        // Load the sample into the filter
        auto filterInvoer = (ads131a02[DSB_ADC_Channel] >>8); //Invoer is 24 bits, alleen
16 bits gebruiken. 8 bits verwijderen.
        auto spanningshift = static_cast<Int16>(filterInvoer);
        auto filterWaarde = filter.filter(spanningshift); // laad spanning door firfilter
        //const auto filterWaarde = filter.filter(filterInvoer) + 0x7FFF;

        if(filterWaarde < minWaarde){
            minWaarde = filterWaarde;
        }
        filterWaarde -= minWaarde;

        auto DACspanning= max5136.dacSpanning(filterWaarde);

        // Load the filtered sample into the DAC
        max5136.zetKanaal(DSB_DAC_Channel, DACspanning);
    }

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

Figuur A.10 student.cpp