NTNU

elsys
ELEKTRONISK SYSTEMDESIGN & INNOVASJON

Appliksjonsnotat

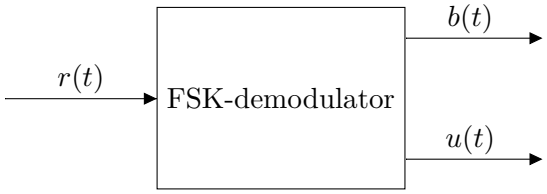
Tittel: FSK-demodulator med Arduino Uno	
Forfattere: Ole Sivert Aarhaug	
Versjon: 1.0	Dato: 16.11.18

Innhold

---

1 Innledning

Dette designnotatet tar for seg å designer en FSK-demodulator som kan detektere signaler sendt over konstante frekvenser, detektere om de er gyldige og dekomponere det til digitale verdier. f_0 er lavre grense definert som binær LAV og f_1 er øvre grense som er binær HØY



Figur 1: Illustrasjon av en generell FSK-demodulator hvor $r(t)$ er det analoge signalet som skal prosesseres, $b(t)$ er det digitale signalet ifra $r(t)$ og $u(t)$ er Høy når signalet er gyldig og ellers lav

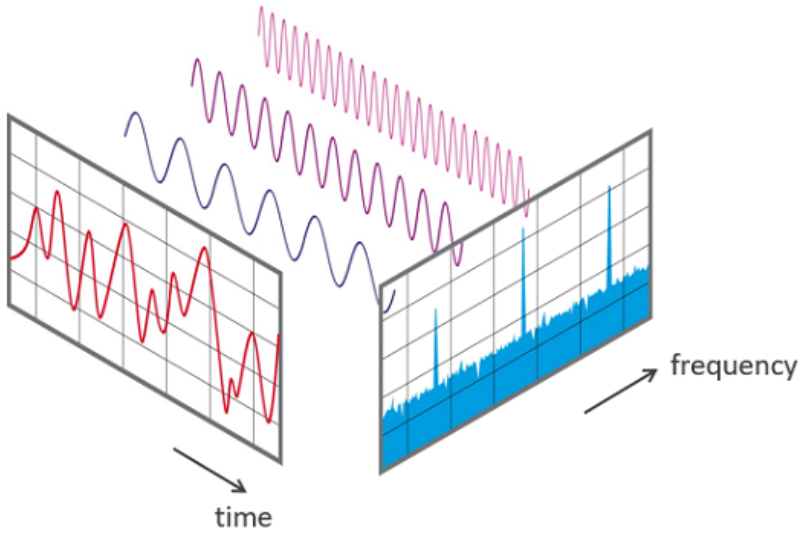
Systemtet illustrert i figur 1 skal også oppfylle en kravspesifikasjon hvor fotavtrykket til den realiserte løsningen skal være mindre enn $3\,\mathrm{cm}^3$

2 Prinsipiell løsning

Det er mange forskjellige måter å implementere en FSK-demodulator på, men her skal vi ta for oss en løsning som bruker FFT (Fast Fourier Transform) og digital logikk til å produsere de ønskede signalene

2.1 Fast Fourier Transform

Fast Fourier Transform også kjent som FFT er en algoritmer for å kalkulere diskre Fourier transformasjoner. Vi skal ikke her gå i detaljer på den avanserte matematikken rundt Fourier transformasjoner, men prinsippet er at de fleste signaler kan representeres som en sum av sinus funksjoner av varierende frekvens og amplitude. Dette kan vi gjøre med det innkommende signalet i $r(t)$ ved å punktprøve signalet her er det viktig at samplingsfrekvens må være dobblet som stor som den største frekvensen som skal måles (Nyquist samplingsteorem), derreter bruke FFT til å få en sum av sinus funksjoner for signalet.

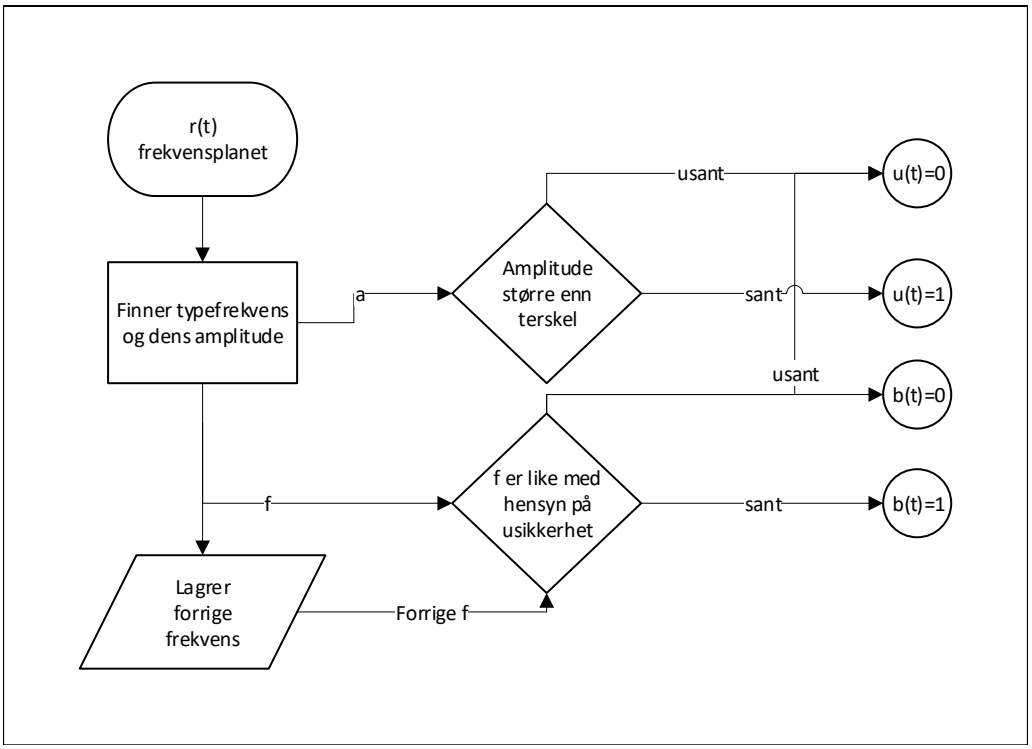


Figur 2: Sum av sinus funksjoner representert i tids og frekvensdomenet [1]

En annen viktig egenskap ved Fourier transformasjoner er at man kan beskrive tidsavhengige funksjoner i frekvensdomenet og motsatt som vist i figur 2. Dette kan vi bruke til å se hvilken typefrekvens man har i signalet.

2.2 Logikken til det digitale filteret

Utifra 2.1 ser man at det er mulig å få en typeferkvens utifra signalet etter FFT. Dermed siden f_0 og f_1 er kjent kan man med litt logikk bruke dette til å produsere $b(t)$ og $u(t)$



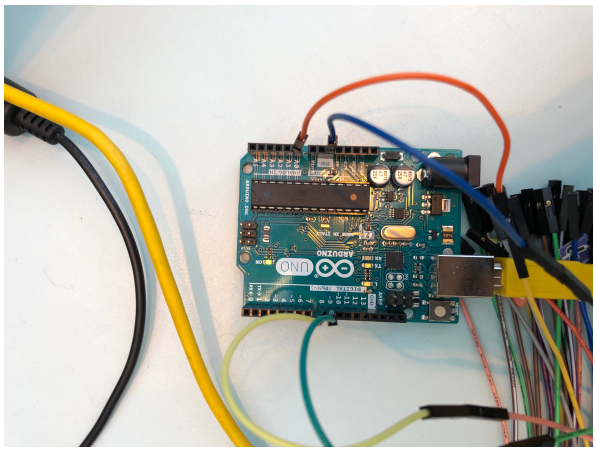
Figur 3: Flytdiagram som viser logikken til det digitale filteret

Figur 4 viser ett flyt diagram for hvordan det digitale filteret skal fungere. Filteret bruker amplituden til typefrekvensen til å bestemme om man har støy eller ikke på signalet. Frekvensen på signalet blir sammenlignet med den forrige typefrekvensen ifra FFT'en tatt før denne og hvis de er like innenfor ett bestemt avvik i Hz og lik f_1 som er bestemt som frekvensen for binær HØY vil $b(t)$ bli HØY, derrimot hvis dette ikke stemmer vil både $u(t)$ og $b(t)$ bli LAV

3 Realisering og test

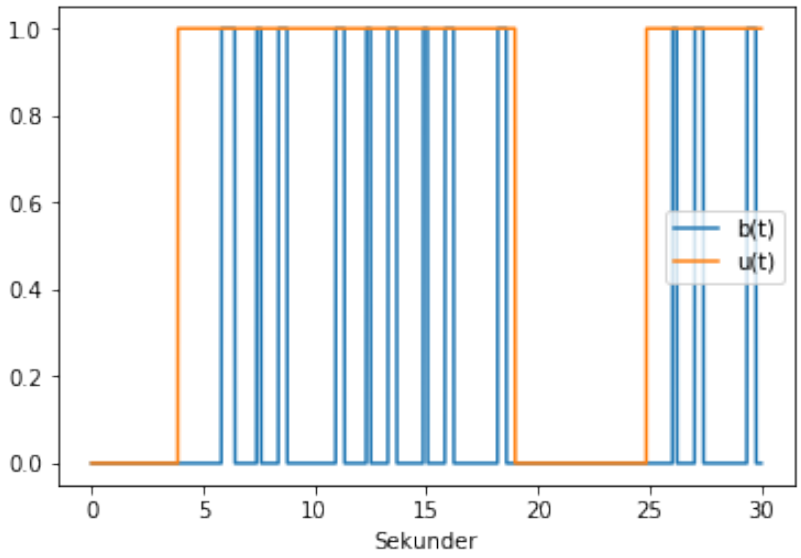
For å realisere dette filteret ble en ATmega328P mikrokontroller brukt som har en pakketype er under $3\,\text{cm}^3$. f_0 og f_1 ble hver satt til en frekvens i henhold til ett signal som varierte mellom to konstante frekvenser, $f_0 = 330\,\text{Hz}$ og $f_1 = 914\,\text{Hz}$

Ved hjelp av FFT og logikken fremlagt i flytdiagrammet i 2 kan det teoretiske filteret realiserers på en mikrokontroller. Hvor singalet blir samplet på $f = 2000\,\text{Hz}$ på grunn av nyquist-samplingsterorem. Deretter kan de samplede verdiene gå igjennom FFT algorytmen og bruke typefrekvensen som beskrevet i flytdiagramet i figur 4. Den resulterende C koden for mikrokontrolleren er vedlagt i 5.



Figur 4: Bilde av ATmega328P mikrokontrolleren koblett i en Arduino Uno for programmering

For å teste filteres påtrykker vi ett signal generert av en signalgenerator som inneholder f_0 og f_1. Signalgeneratoren kan også generere forskjellige typer støy som kan teste pålideligheten til filteret.



Figur 5: De binære signalene målt ifra mikrokontrolleren med logikkanalysator

Utifra figur 5 ser man klart at signalet kommer ut på den forventede formen. Hvis man ikke sender ett signal eller genererer syntesert støy med frekvensgeneratoren holder både $b(t)$ og $u(t)$ seg LAV

4 Konklusjon

Som man kan se utifra figur 5 produserer filteret dekodet digitale signaler når $r(t)$ har en frekvensverdi på f_0 eller f_1. $b(t)$ inneholder lengden $r(t)$ er enten f_0 eller f_1 og $u(t)$ er HØY når signalet er gyldig. Hele løsningen tar heller ikke større plass enn $3\,\mathrm{cm}^3$ på grunn av at, man trenger bare mikrokontrolleren for å bruke filteret. Filteret oppfyller dermed de kravene som ble satt og er en enkel og grei løsning for å dekode slike frekvensbaserte signaler.

Referanser

[1] Ukjent, Fast Fourier transform,
`https://en.wikipedia.org/wiki/Fast_Fourier_transform/` Lastet ned 16.11.2018

[2] Norwegian Creations,
What Is FFT and How Can You Implement It on an Arduino?
https://www.norwegiancreations.com/2017/08/what-is-fft-and-how-can-you-implement-it-on-an-arduino/
Lastet ned 16.11.2018

5 Vedlegg: C-Kode

Listing 1: C kode programert i Arduinoen

```
#include "arduinoFFT.h"

#define SAMPLES 128              //Must be a power of 2
#define SAMPLING_FREQUENCY 2000 //Hz, must be less than 10000 due to ADC

#define BASELOW 330
#define BASEHIGH 914

arduinoFFT FFT = arduinoFFT();

unsigned int sampling_period_us;
unsigned long microseconds;

double vReal[SAMPLES];
double vImag[SAMPLES];

bool validSignal;

double prevSample;

void setup() {
```

```
    Serial.begin(115200);
    pinMode(8, OUTPUT);
    pinMode(9, OUTPUT);

    validSignal =false;

    prevSample = 10.0;

    sampling_period_us = round(1000000*(1.0/SAMPLING_FREQUENCY));
}

void loop() {

    /*SAMPLING*/
    for(int i=0; i<SAMPLES; i++)
    {
        microseconds = micros();    //Overflows after around 70 minutes!

        vReal[i] = analogRead(0);
        vImag[i] = 0;

        while(micros() < (microseconds + sampling_period_us)){
        }
    }

    /*FFT*/
    FFT.Windowing(vReal, SAMPLES, FFT_WIN_TYP_HAMMING, FFT_FORWARD);
    FFT.Compute(vReal, vImag, SAMPLES, FFT_FORWARD);
    FFT.ComplexToMagnitude(vReal, vImag, SAMPLES);
    double peak = FFT.MajorPeak(vReal, SAMPLES, SAMPLING_FREQUENCY);

    /*PRINT RESULTS*/
    //Serial.println(prevSample - 2);
    Serial.println(peak);     //Print out what frequency is the most dominant.
    //Serial.println(prevSample + 2);


    validSignal = false;
    for(int i=0; i<(SAMPLES/2); i++)
    {
        Serial.print((i * 1.0 * SAMPLING_FREQUENCY) / SAMPLES, 1);
        Serial.print("␣");
        Serial.print(vReal[i], 1);
        Serial.print("␣");
        float freq = (i * 1.0 * SAMPLING_FREQUENCY) / SAMPLES;
        if(vReal[i] > 1000){
          validSignal = true;
```

```
        }
    }
    Serial.println("");

    if(((peak >= (prevSample - 2)) &&
        (peak <= (prevSample + 2))) &&
        ( peak >= 900.0 ) &&
        (validSignal)){
      Serial.println(1);
      digitalWrite(8, HIGH);
    }else{
      Serial.println(0);
      digitalWrite(8, LOW);
    }

    prevSample = peak;

    if(((peak >= (BASELOW - 2)) &&
        (peak <= (BASELOW + 2))) ||
        ((peak >= (BASEHIGH - 2)) && (peak <= (BASEHIGH + 2))) &&
        (validSignal)){
      Serial.println(1);
      digitalWrite(9, HIGH);
    }else{
      Serial.println(0);
      digitalWrite(9, LOW);
    }

    delay(10);  //Repeat the process every second OR:
    //while(1);       //Run code once
}
```