INF122: Oblig 0

Universitetet i Bergen

2022

I denne obligen skal vi lage et Haskellprogram som gjør litt enkel signalbehandlig ved hjelp av lister. Programmet har som mål å lese inn (simulert) sensordata fra et aksellerometer som en gående person bærer med seg og tolke dataen for å telle skritt.

Oppgaven er ikke helt realistisk, vi gjør noen forenklinger ettersom fokuset er på å skrive Haskellkode. Men grunnprinsippene er de samme som i virkelig signalbehandling.

Enkel støyfjerning med lavpass- og høypassfilter

Ute i den virkelige verden dataen vi har tilgjengelig ofte full av støy og feilkilder. I mange tilfeller er man ikke opptatt av den nøyaktige verdien til en hver tid, men heller ute etter å se den generelle trenden. Dette kan vi se på som at vi vil **filtrere bort høyfrekvent støy**. Et enkelt de fleste har sett er 7-dagers gjennomsnittskurvene for COVID-19 tilfeller (Se figur 1).

Et hvert signal kan deles opp i et spektrum av informasjon av forskjellig frekvens. De lavfrekvente delene styrer utviklingen på lang sikt, mens de høyfrekvente delene har større innvirkning på kort sikt. En funksjon som reduserer de høyfrekvente delene, og beholder de lavfrekvente, kalles **et lavpassfilter** (Her er "pass" som i "passere"). Motsatt kaller vi en funksjon som tar bort de lavfrekvente delene, og beholder de høyfrekvente, **et høypassfilter**. Se figur 2 for hvordan høypass- og lavpassfiltre endrer et eksempelsignal.

Matematisk kalles det å analysere et signal etter frekvens Fourier-analyse, men for denne oppgaven skal vi holde oss til å bruke rullende gjennomsnitt som et enkelt lavpassfilter og subtraksjon av lavpassfilteret som høypassfilter.

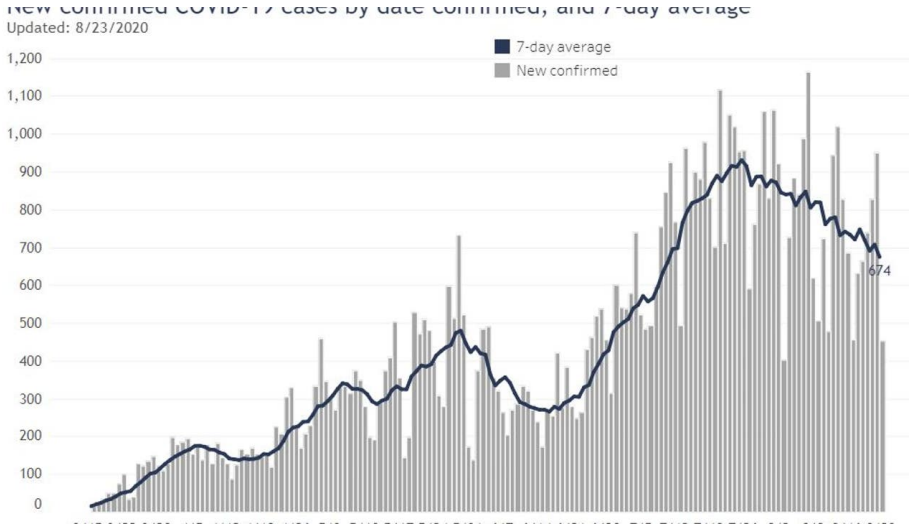


Figure 1: Eksempel på 7-dagers gjennomsnittskurve.

En skritteller

Hypotesen vi jobber ut ifra er at signalet vi får fra akselloremeteret gir et gjentagende mønster som vi kan gjenkjenne. Omtrent som en sinus-kurve (se Figur 2).

Perioden til et slikt gjentagende signal er intervallet fra topp til topp. Hvis signalet er glatt og går opp og ned rundt 0-punktet, slik som kurven over, så vil **kurven krysse null-linjen to ganger i hver periode**. Så hvis vi teller hvor mange ganger kurven krysser nullpunktet, og deler på to, vet vi hvor mange perioder som har gått.

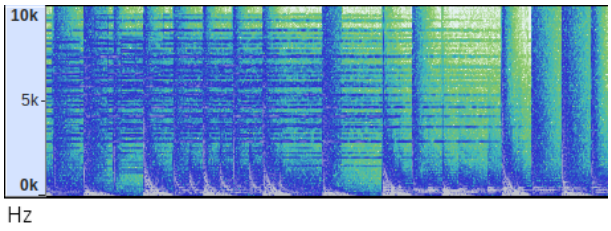
Vi antar at en periode av signalet vårt tilsvarer et skritt. Så antallet skritt er antallet perioder i signalet fra sensoren, altså antall null-kryssninger delt på to.

Men, hvis det er høyfrekvent støy på signalet kan det hende at vi får flere kryssinger av null-linjen hver periode. Og dersom kurven blir forskjøvet kan det hende den har perioder hvor den ikke krysser null. Det er disse to tilfellene vi må korrigere for ved hjelp av lavpass og høypass filtere. (Se figur 4 og 5)

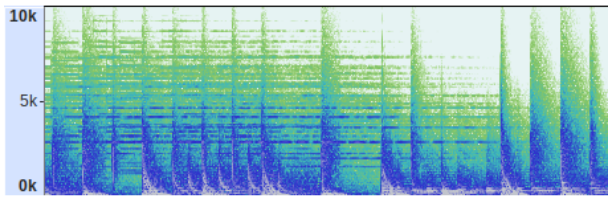
Signalet fra sensoren

Sensoren gir et konstant signal med en samplingsrate på 100Hz. Det vil si at avstanden mellom hvert datapunkt er ett hundredels sekund.

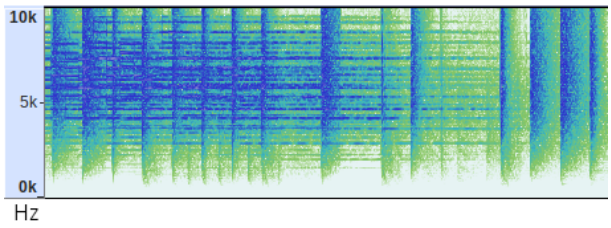
Sensoren gir signalet som trippler (a_x, a_y, a_z) som gir aksellerasjonen i hver retning. Siden vi ikke vet hvilken retning sensoren er orientert tar vi (for enkelhets skyld) summen av koordinatene før vi gjør noen videre signal behandling.



Spectrogram av orginal signalet (rytmisk lyd)



Spectrogram av filtrert signal (lavpass)



Spektrogram av filtrert signal (høypass)

Figure 2: Spektrogrammer av filtrerte signaler

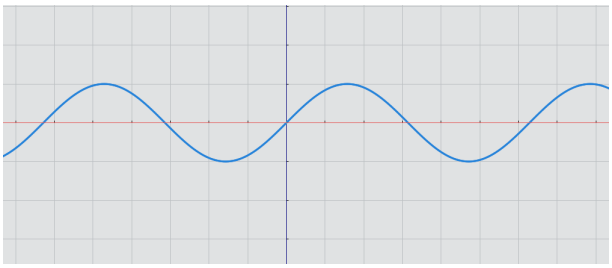


Figure 3: En sinuskurve.

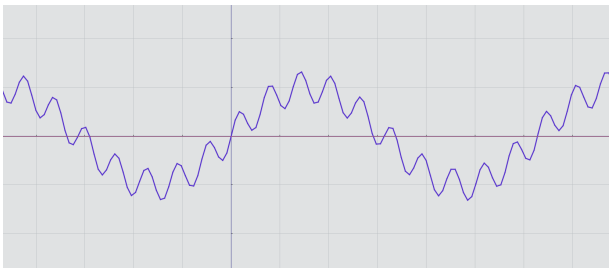


Figure 4: Dette signalet har høyfrekvent støy som gjør at den krysser x-aksen flere steder.

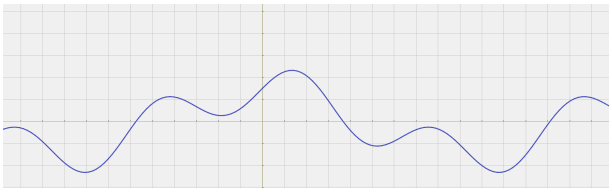


Figure 5: Dette signalet har lavfrekvent støy som gjør at den ikke krysser x-aksen for hver periode.

Summen av de tre koordinatene har ikke noen spesiell fysisk tolkning, men gir et nyttig signal som er lite sensitivt for rotasjonen til sensoren.

Testdata for sensoren er tilgjengelig på MittUiB. Testfilen består av et målingspunkt per linje.

Oppgavekoden

Hovedideen i koden er at vi representerer signalet vårt som lister hvor de nyeste inputene fra sensoren ligger fremst i listen. Filterfunksjonene våre blir funksjoner av typen `[a] -> a` som regner ut den siste verdien til den filtrerte signalet fra hele signalhistorien. (Denne typingen forhindrer noen effektiviseringer, men forenkler strukturen til programmet.)

En `main` funksjon leser inn signalet fra standard input og bruker filterne på den. Etter filtrering kjøres funksjonen zeroCrossings som teller hvor mange ganger kurven krysser x-aksen og deler på to for å estimere antall perioder.

Koden er delt i to filer. En fil som heter Oblig0Common.hs hvor dere skal implementere de manglende funksjonene. Oblig0Ver0.hs inneholder main funksjonen, som er ferdig skrevet. I oppgave 6 skal dere lage en alternativ main funksjon som skal ligge i en tredje fil som heter Oblig0Ver1.hs.

Formatet på testdataen er et trippel "(a,b,c)" per linje, hvor a, b og c er flyttall.

Oppgaver

Løs hver av oppgavene under. Oppgave 0 til 5 løses i filen Oblig0Common.hs, som du finner i filseksjonen på MittUiB. Oppgave 6 skal løses i en egen fil som heter Oblig0Ver1.hs som du selv lager, hvor modulnavnet skal være Main.

Oppgave 0 (2 poeng)

Lag en funksjon som regner gjennomsnittet av liste:

```
average :: (Fractional a) => [a] -> a
```

Fremgangsmåte: Bruk de innebygde funksjonene `sum` og `length`.

Oppgave 1 (3 poeng)

Lag en funksjon teller hvor mange 0-punktskryssninger det er i en liste.

```
zeroCrossings :: (Num a, Ord a) => [a] -> Integer
```

En 0-punktskryssing er to etterfølgende elementer i listen, x og y, slik at x er større enn 0 og y er mindre eller lik null, eller motsatt at x er mindre enn 0 og y er større eller lik 0.

Fremgangsmåte: Bruk rekursjon og pattern matching.

Oppgave 2 (2 poeng)

I begynnelsen av signalet har vi lite å analysere, så for at ikke filterene skal tolke kanten som et hopp i signalet utvider vi signalet med falske fortidsverdier.

Lag en funskjon som utvider en endelig liste til en uendelig liste ved å repetere det siste element ad infinitum. Hvis inputlisten er tom skal funksjonen produsere en uendelig liste med 0-er.

```
extend :: Num a => [a] -> [a]
```

Fremgangsmåte: Bruk rekursjon og patternmatching. For å repetere et enkelt element, si `x`, ad infinitum kan du bruke `repeat x`.

Eksempel: `extend [1,2,3]` = `[1,2,3,3,3,3`$\cdots$ og `extend [] = [0,0,0,0,`$\cdots$

Oppgave 3 (2 poeng)

Nå er vi kommet til å skrive lavpassfilteret vi skal bruke på signalet. Vi gjør det enkelt og lager en familie av lavpassfiltre som tar gjennomsnittsverdien til et gitt antall tidligere målinger. Dermed blir oppgaven å skrive en funksjon som tar et heltall n og en liste med verdier og returerer gjennonsnittet av de n nyeste verdiene.

```
lpf :: (Fractional a) => Integer -> [a] -> a
```

Fremgangsmåte: Hent ut de første elementene i listen og bruk gjenomsnitts-funksjonen du skrev i oppgave 0.

Oppgave 4 (2 poeng)

Nå skal vi lage høypassfilteret. Igjen tar vi en snarvei: Lavpassfilteret fjerner den høyfrekvente delen av signalet, så hvis vi subtraherer det filtrerte signalet fra det orginale, burde vi stå igjen med den høyfrekvente delen som lavpassfilteret tok bort.

I denne oppgaven skal du skrive en funksjon for høypassfilter som tar det samme paramereret som lavpass filteret, og subtraherer verdien i lavpass filteret med den nyeste verdien til signalet (første element i listen).

```
hpf :: (Fractional a) => Integer -> [a] -> a
```

Fremgangsmåte: Bruk lavpassfilteret og subtraher resultatet fra første verdien i listen.

Oppgave 5 (2 poeng)

Nå skal du kalibrere parameterne for lavpass og høypass. Parameterne er satt i heter lowPassCutoff og highPassCutoff, og må tilpasses signalet du får inn for å filtrere bort støyen og beholdet signalet vi er interessert i for å telle skritt.

Det ligger testdata i filen testData0 på MittUiB.

Denne dataen tilsvarer 555 skritt. Kalibrér filterverdiene slik at du kommer innenfor 2% feilmargin.

Hint: høypass cutoff skal settes lavere enn lavpass cutoff. Hvis lavpass verdien blir for stor detekteres det for få skritt.

Fremgangsmåte: Kjør programmet på testverdiene og juster til du har mindre enn 2% feilmargin.

Du kan kjøre programmet på testdataen ved å redirigere filen til programmets input:

På linux/mac:

```
runhaskell Oblig0Ver0.hs Oblig0Common.hs < testData0
```

På windows (i cmd):

```
runhaskell .\Oblig0Ver0.hs .\Oblig0Common.hs  < testData0
```

På windows (i powershell):

```
cmd /c "runhaskell .\Oblig0Ver0.hs .\Oblig0Common.hs < testData0"
```

Oppgave 6 (2 poeng) (∗)

Når du har gjort de foregående oppgavene skal programmet fungere fint. Men måten vi har satt det opp krever at hele signalet leses inn før det prossesseres. I denne oppgaven skal du lage en ny versjon av main funksjonen til programmet (kall den nye filen din Oblig0Ver1.hs, med modulnavn Main) som leser inn signalet kontinuerlig og skriver ut en linje med teksten "Step!" hver gang et nytt skritt detekteres. På denne måten kan programmet kjøres kontinuerlig uten å avsluttes.

Det vil at "Step!" skrives ut for hvert steg (f.eks. ca 555 ganger for testData0), og det skal skrives ut så snart det er nok data til å si at et steg har forekommet. For eksempel, her er en kjøring med litt overdreven data for å gi steg etter kort tid, som viser hvordan "Step!" skrive ut mellom inputdataene.

```
(0,0,0)
(1,0,0)
(1,0,0)
(-1,0,0)
(-1,0,0)
Step!
(1,0,0)
(1,0,0)
(1,0,0)
(-1,0,0)
(-1,0,0)
Step!
```

(Dette eksemplet er ment for illustrasjon. Det er ikke sikkert dine filterverdier gir steg etter disse verdiene.)

Fremgangsmåte: Bruk rekursjon og en hjelpefunksjon til å lese input linje for linje og gjør filter og stegdeteksjon hver gang det kommer ny input. Husk å flushe stdout for hver linje du skriver ut: enten ved å skifte til linebuffering eller manuelt flushe med hFlush stdout etter hver linje.

(En mer avansert løsning kan være å bruke at IO i Haskell er lat og gjøre beregninene mer uniformt på listen av inputs.)