Innhold

1. Prosjektgjennomføringen:

1.1: Problemstilling:

Hovedoppgaven i dette prosjektet var å kjøre LEGO roboten igjennom en bane der banens farge gikk fra hvitt på den ene siden til sort på den andre. Roboten skulle styres med en logitech joystick og programmeres igjennom Matlab. Banen ble målt med en lyssensor og avviket fra sensorens initialmåling skulle være så liten som mulig. Roboten skulle også være i stand til å kjøre igjennom banen på egen hånd ved hjelp av numerisk integrasjon og derivasjon
I tillegg til dette kunne vi utvikle andre programmer/funksjoner for å vise kreativitet og kunnskap innen faget.

1.2: Gruppearbeid:

Gruppen bestod av 3 studenter som går første semester på bachelorutdanningen for elektroingeniør og en student som går første semester på masterutdanning for informasjonsteknologi på UIS. Ingen på gruppen har jobbet med Matlab eller LEGO Mindstorms tidligere. Arbeidet i gruppen har stort sett vært samlet, men vi har også jobbet parvis og enkelte har hatt med roboten hjem og jobbet individuelt. Pga. at softwaren til LEGO Mindstorms i utgangspunktet kun virker med Windows 7 har ikke alle hatt denne muligheten, men alle har uansett lagt inn en solid arbeidsmengde i prosjektet og vi ser på dette som et prosjekt vi har gjort sammen som en gruppe.
Arbeidet begynte med bygging av roboten og testing av motor og programvare. I første laboratorietime jobbet vi sammen rundt én datamaskin, men fant fort ut at fire personer rundt en datamaskin ikke fungerte så bra, så etter dette har vi brukt minst to datamaskiner ved hver samling.
Arbeidet har foregått ved tildelt tid på laboratoriet, og som regel utover den gitte tiden så langt det har passet de enkelte gruppemedlemmene, i tillegg til en del hjemmearbeid.

1.3: Metode:

1.3.1: Programvare.

Lego Mindstorms NXT:
Roboten ble bygget av et Lego Mindstorms NXTsett. Hovedkomponenten er en «NXT Intelligent Brick». Den kan ta imot input fra opptil fire forskjellige sensorer og styre opptil tre motorer. Den har også en høyttaler og kan spille av lyder. Settet inneholdt i tillegg tre motorer, trykksensor, lyssensor, ultralydsensor og lydsensor

Matlab:

Matlab (matrix laboratory) er et av de mest brukte matematiske progamvarene blant Ingeniører.
Matlab er utviklet av Mathworks og brukes mye til matrisemanipulasjon, plotting av funksjoner og behandling av data. Språket som brukes er basert på C.
Det er utviklet forskjellige typer «toolboxer» til Matlab for å utvide bruksområdene. Vi har for eksempel benyttet oss av RWTH Mindstorms Toolbox i dette prosjektet.

RWTH Mindstorms Toolbox er utviklet av RWTH Aachen University og inspirert av et prosjekt for første års ingeniørstudenter i 2007

Microsoft Visual Studio 2010 Ble benyttet for å kunne bruke joysticken.

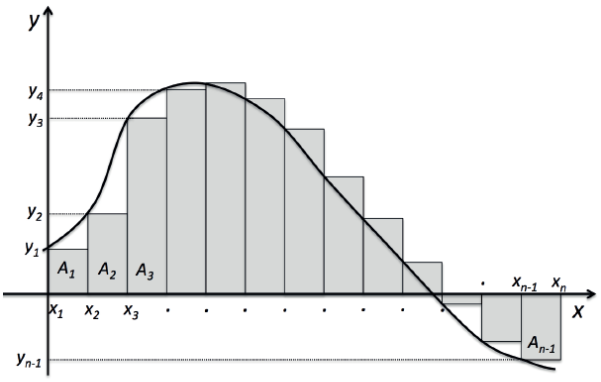
1.3.2: Arbeid med programmering.

Selve programmeringen av roboten har fra første stund vært det mest sentrale aspektet ved prosjektet, og de fleste programmene har blitt til ved og bare sette i gang med å skrive koder. Det har vist seg å fungere til en viss grad, men mange ganger har vi måttet stoppe opp og tenke på nytt og gjerne skrive ned hva vi egentlig ville ha ut av programmet for å få litt oversikt og en litt klarere ide om hva som måtte være med.
Etter hvert som vi begynte å få kontroll på det grunnleggende (vanlig kjøring rundt banen med integrering og derivering av areal) og skulle starte med flere programmer fant vi ut at vi skulle lage egne filer for hvert enkelt program i stedet for å prøve å få alle programmene inn i en fil, da denne ville blitt stor og uhåndterlig.
Derfor bestemte vi oss for å lage en hovedmeny der vi kan starte det programmet vi ønsker å kjøre.
Navnene på filene indikerer hva det enkelte programmet gjør. (auto kjører automatisk f.eks.)
Vi benyttet oss av dropbox for å dele filene slik at alle hadde tilgang til nyeste versjon av et program til enhver tid.

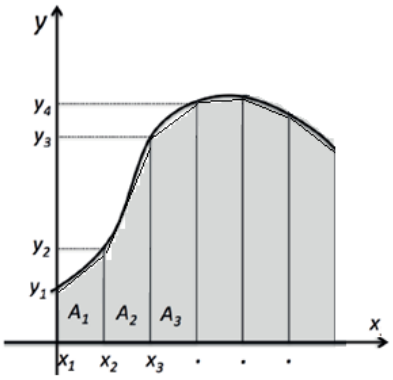
2: Programmenes oppbygging og funksjonalitet.

2.1: Numerisk Integrasjon:

Når vi har et problem med et ubesemt integral der løsningen er tilnærmet umulig å løse eksakt tar vi i bruk numerisk integrasjon. Ved å anvende integrasjonsteorien fra matematikken kan vi komme uendelig nærme den eksakte løsningen. Har vi en funksjon som endrer seg langs X-aksen av f.eks tiden får vi et areal mellom funksjonen og aksen. Dersom vi deler dette opp i uendelig mange små søyler og legger dem sammen får vi en tilnærmet eksakt løsning av arealet.



Figur 1: Prinsippet for numerisk integrasjon.



Figur 2: Trapesmetoden

I gruppens programmer blir numerisk integrasjon anvendt i manuell og automatisk kjøring av roboten rundt IDE sin racetrack. I løypen til racetracken har vi et felt som går fra hvitt mot sort(verdier fra ca 0-1100, midtpunkt på 550). Alt ettersom hvilken "gråtone" lyssensoren er på vil den fange opp ulike verdier, det er disse vi bruker for å integrere arealet.

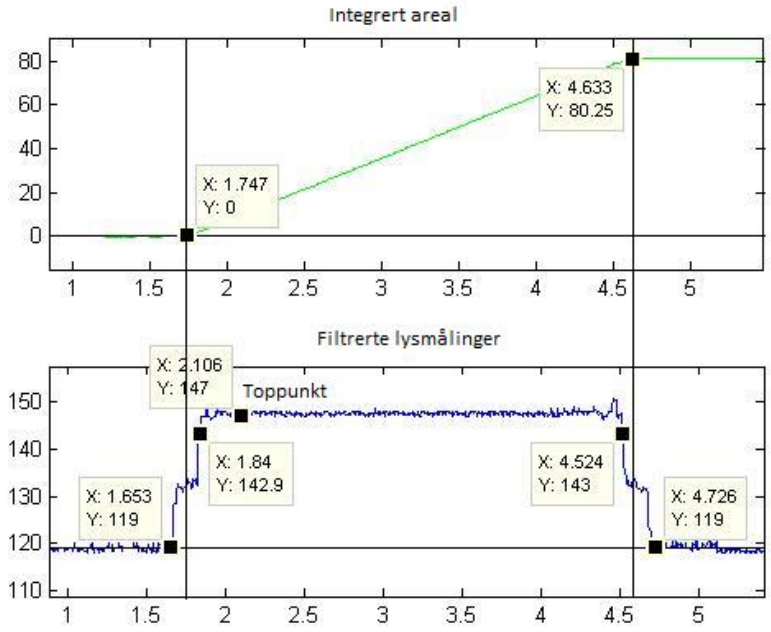
Vi tok utgangspunkt i Eulers forovermetode og omformet den til vår bruk.
Denne metoden er noe mer unøyaktig enn andre som f.eks trapesmetoden(figur 2), men ettersom at søylene vil bevege seg litt over og litt under funsjonen(grafen) vil dette utligne avviket mellom nøyaktig og målt areal noe. Vi tok en beslutning på at rektangen metoden, altså figur 1 var god nok for vår problemstilling.

Her er et utsnitt av koden vi brukte til å integrere arealet:

```
intAreal_new = intAreal(i-1) + deltaTime(i)*abs(avvik);
```

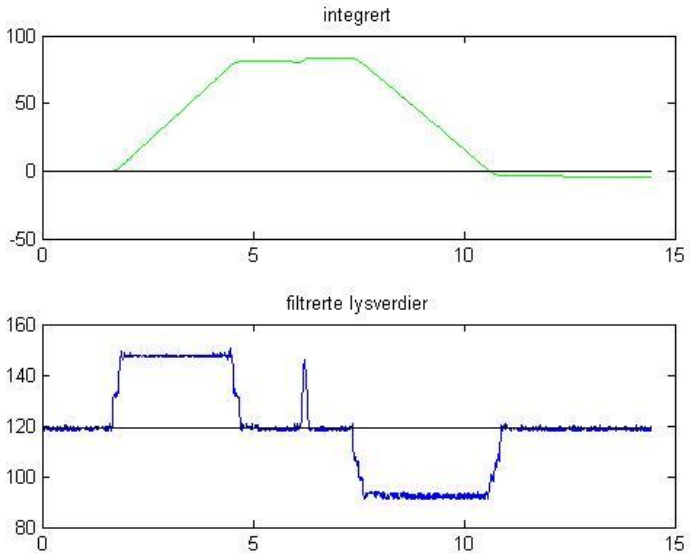
Der `intAreal_new` er det forrige arealet lagt sammen med tidsintervallet(`deltaTime`) muliplisert med absoluttverdien av første og neste lysmåling(avviket mellom dem).
Sammenlignet med figur 1 over får vi A1 pluss tiden mellom X2 og X3 multiplisert med Y2 som er lik A1+A2.
Når vi kjører langs banen vil det integrerte arealet tilsvare hvor mye avviket ble mellom første måling som blir vårt nullpunkt og hvor lysensoren faktisk måler(som funsjonen i figur 1, der lysensoren er grafen og nullpunktet er X-aksen).
At vi bruker absoluttverdi som en del av formelen er fordi det er ønskelig å summere både det negative og positive arealet, slik at vi kan bruke denne verdien som et resultat på hvor mye avviket ble til slutt.

Her er en figur som viser hvordan vi kontrollerte at resulatene vi fikk av å integrere den filtrerte lysverdien stemmer.



<u>Avviksverdi</u>

Δt = ((4.726+4524)/2)-((1.84+1.653)/2) = 2.878s
e = 147-119 = 28
A = e*t = 28*2.878 = 80.58



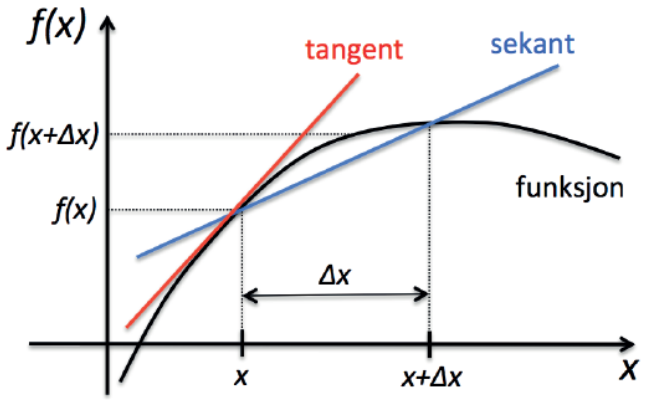
Figuren viser hvordan den integrerte grafen så ut i forhold til de målte lysverdiene.

2.2: Numerisk Derivasjon:

Likt som ved numerisk integrasjon har vi problemer med å få eksakte deriverte løsninger på alle problemer, derfor må vi anvende numerisk derivasjon for å komme så nærme som mulig den eksakte løsningen.
Den deriverte sier oss noe om hvor for fort noe endrer seg, feks. funksjonen på figur 3 nedenfor. Altså stigningstallet til tangenten i et gitt punkt(merket rødt på figur 3). Dette er om vi lar grensen gå mot 0 i definisjonen for den deriverte fra matematikken, delta x går mot 0.

$$f'(x) = \lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x}$$

Om vi derimot lar grensen gå mot en verdi($\Delta x \neq 0$) kan vi få en tilnærmet løsning for den deriverte, altså sekanten(merket blått på figur 3). Men fordi denne verdien kan variere vil vi komme lenger og lenger bort fra den eksakte(tangenten) desto større grensen blir.



Figur 3 : En funksjon f(x) med den eksakte(tangent) og tilnærmede deriverte(sekant).

I figuren representerer X-aksen tiden, da vil Δx være tidsskrittet mellom x og den neste målte verdien av x som blir $x+\Delta x$. Altså Δx er tidsintervallet mellom første og neste måling vi får fra lyssensoren når vi kjører manuelt eller automatisk som vi kan finne en tilnærmet verdi for.

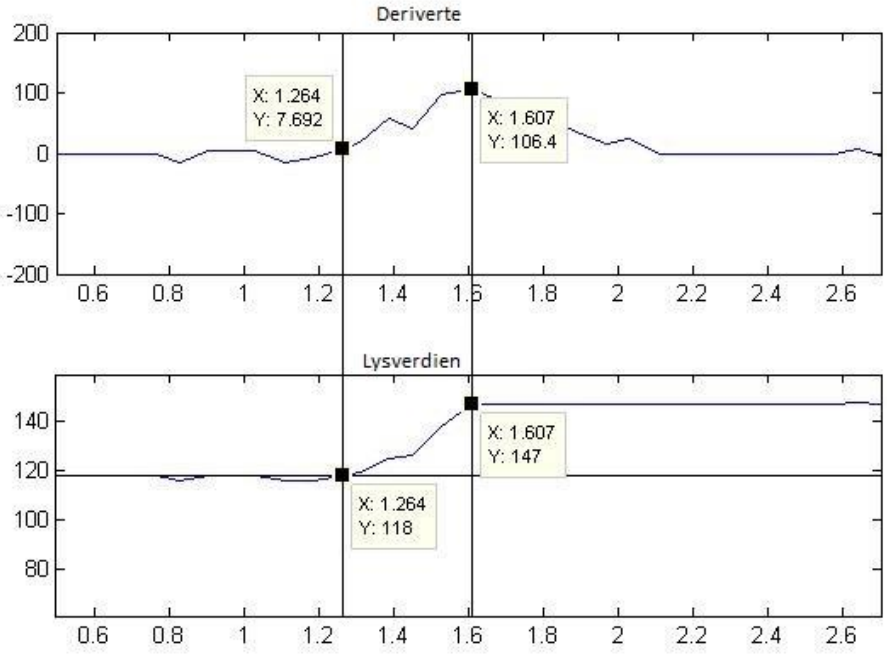
Et utsnitt av koden hvor vi anvender derivasjon:

```
deltaLys     = filtLys(i)-filtLys(i-1);
deltaTime(i) = etime(newTime,oldTime);
derivert(i)  = (deltaLys ./deltaTime(i));
```

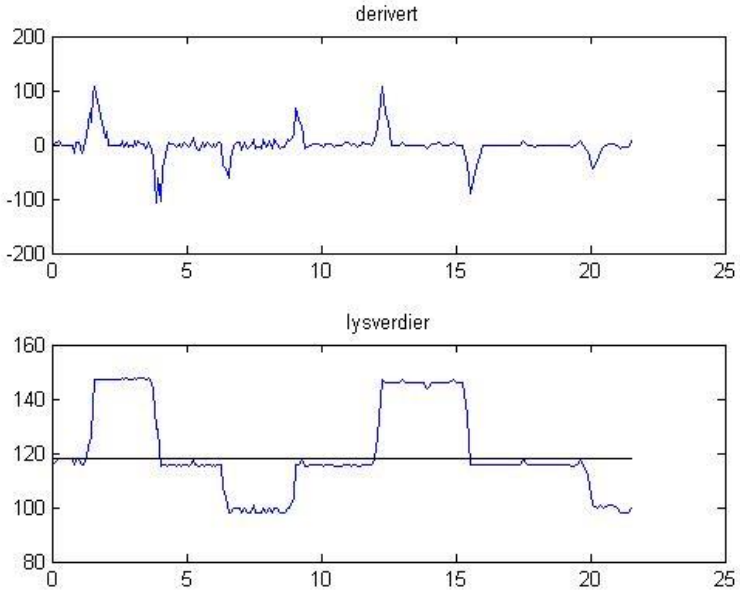
Der `derivert(i)` er den nåværende målte lysverdien subtrahert med den foregående lysverdien(`deltaLys(i)`) dividert med tidsintervallet mellom målingene(`deltaTime(i)`).

Numerisk derivasjon har vi anvendt i programmet der vi kjører roboten automatisk, for å unngå at roboten skal få for stort motorpådrag når den returnerer til nullpunktet.

Her er en figur som viser hvordan vi kontrollerte at resultatene vi fikk av å derivere den filtrerte lysverdien stemmer.



<u>Endringsverdien</u>
Δt = 1.607-1.264 = 0.343s
Toppunkt = 147
Bunnpunkt = 118
e = (147-118)/0.343 = 84.55



Denne figuren viser ufiltrerte verdier av den deriverte og lys

2.3: Filtrering:

Når vi startet å anvende måleinstrumentene som fulgte med NXTen merket vi fort at signalene vi fikk var veldig varierende. Altså inneholdt mye støy. Dette har mange faktorer og å motvirke dette problemet finnes mange metoder.

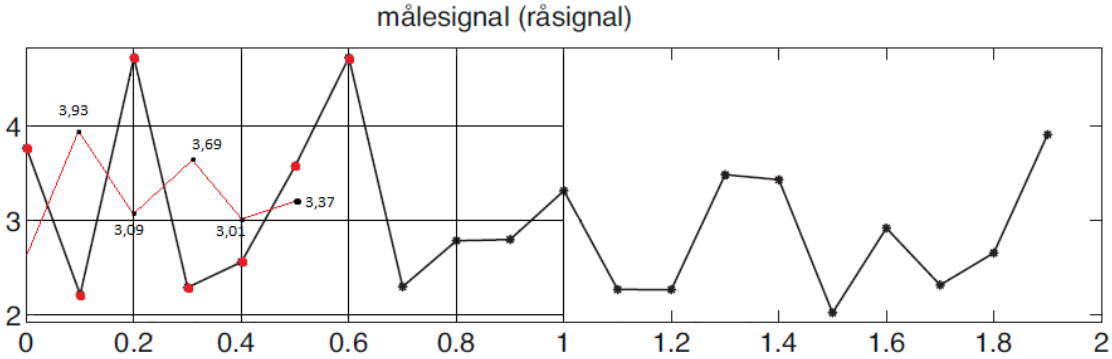
Metoden vi tok i bruk var FIR-filteret, et såkalt digitalt filter. Som går ut på å vekte en sum basert på tidligere målinger. Vi tok utgangspunkt i 3 målinger som koden under viser:

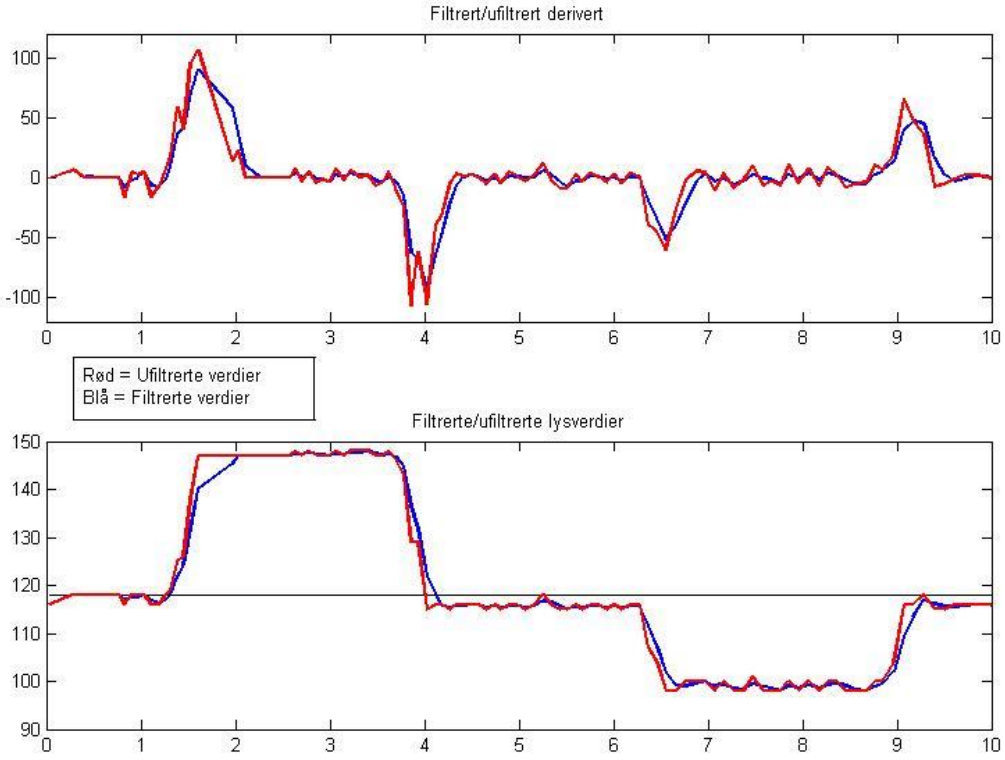
Filtrering av lys:
```
filtLys(i) = 0.5*Lys(i)+0.3*Lys(i-1)+0.2*Lys(i-2);
```

Filtrering av den deriverte:
```
filtDer(i) = 0.5*derivert(i)+0.3*derivert(i-1)+0.2*derivert(i-2);
```

Der `filtLys(i)` eller `filtDer(i)` er en sum laget av nåværende målte verdi dividert med en halv(1/2), andre måling bak i tid dividert med en tredje del(1/3) og tredje måling bak i tid er dividert med en femte del(1/5). Fordelen vi får ved å summere opp 3 ulike måleresultater på denne måten er at vi vil legge oss nærmere midtverdien av grafen for at svingningene i signalet skal avta. Dette kan fort vises matematisk om en setter inn noen verdier i koden for enten lys eller den deriverte. Her er et eksempel med 3 filtrerte målinger:

Vi setter inn verdiene merket rødt inn i formelen fra koden i programmet:
5, 2.3 og 3.7
filtLys(1) = (0.5*5)+(0.3*2.3)+(0.2*3.7) = 3,93
2.3, 5 og 2.2
filtLys(2) = (0.5*2.3)+(0.3*5)+(0.2*2.2) = 3,09
2.5, 2.3 og 5
filtLys(3) = (0.5*2.5)+(0.3*2.3)+(0.2*5) = 3,69
3.6, 2.5 og 2.3
filtLys(4) = (0.5*3.6)+(0.3*2.5)+(0.2*2.3) = 3,01
2.3, 5 og 3.6
filtLys(5) = (0.5*2.3)+(0.3*5)+(0.2*3.6) = 3,37





Denne figuren viser våre målinger av derivasjon og målte lysverdier med og uten filter.

2.4: Manuell kjøring av robot:

Det første vi måtte sette oss inn i var å få roboten til å kjøre ved hjelp av joystick. Vi fikk utdelt et program som initialiserte én motor og hvilke akser fra joysticken som styrte denne.

```
motorA = NXTMotor('A','SmoothStart',true); % initialiserer motor A

JoyForover(i)    = -joystick.axes(2)/327.68; % 32768 fremover, -32768
bakover
```

joystick.axes 2 styres ved at joysticken føres rett frem og tilbake.

For å initialisere motorB måtte vi doble alle kodene som inneholdt motorA og sette inn motorB i stedet. Dette gjorde at roboten kunne kjøre rett frem og tilbake.

```
motorA = NXTMotor('A','SmoothStart',true); % initialiserer motor A Høyre
motorB = NXTMotor('B','SmoothStart',true); % initialiserer motor B Venstre
```

Ved hjelp av «joytest» fant vi ut at å føre joystick fra side til side tilsvarte akse 1. Ved å sette inn denne aksen, og tilegne egne variabler for å kjøre fremover og å svinge til siden, kunne vi lage en liten kommando som gjorde at vi kunne kombinere disse to aksene:

```
PowerA(i) = floor((Forover+Side)/2);
PowerB(i) = floor((Forover+(-Side))/2);
```

I tillegg lagde vi et lite filter for å få jevnere kjøring der vi tok utgangspunkt i de tre siste verdiene fra joysticken med størst vekt på siste måling:

```
if i>3
        Forover = 0.5*JoyForover(i)+0.3*JoyForover(i-1)+0.2*JoyForover(i-
2);
        Side = 0.5*JoySide(i)+0.3*JoySide(i-1)+0.2*JoySide(i-2);
    else
        Forover = JoyForover(i);
        Side = JoySide(i);
    end
```

På grunn av små forstyrrelser fra joysticken hadde roboten en tendens til å kjøre av seg selv med en gang vi startet programmet. Dette fikset vi ved å lage en if sløyfe som tillater litt «dødgang» på styringen. Det vil si at joysticken måtte beveges over et visst antall akser for at roboten skulle reagere:

```
  if abs(PowerA(i))>4 && abs(PowerB(i))>4;
        motorA.Power = PowerA(i);
        motorB.Power = PowerB(i);
    else
        motorA.Power = 0;
        motorB.Power = 0;
```

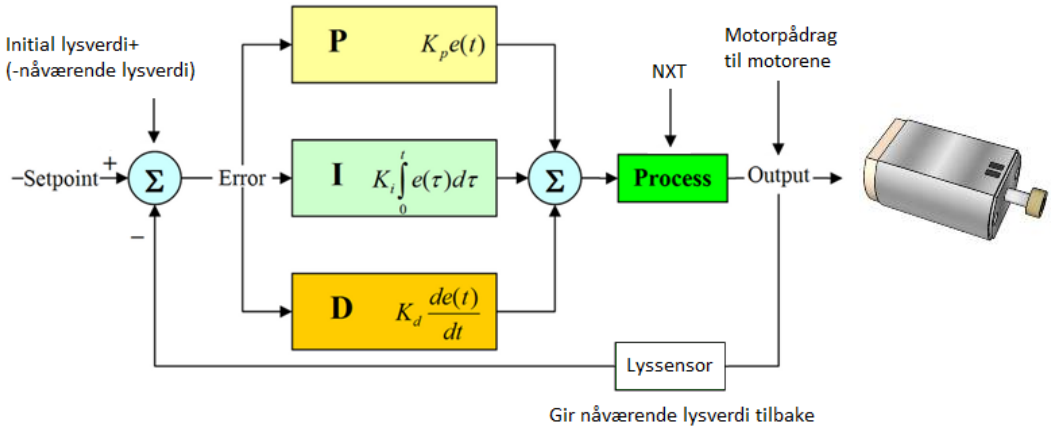
Når alt dette var på plass skulle roboten kunne kjøres som normalt både rett frem og til sidene. Vi støtte derimot på et problem med at roboten hadde en tendens til å kjøre litt fortere på det ene hjulet enn på det andre. Etter å ha prøvd mange forskjellige kombinasjoner for å fikse dette problemet, fant vi ut at den ene motoren rett og slett var defekt og sendte ut full kraft hele tiden. Dette kom vi frem til ved å sjekke vektorene for motorpådrag og sammenligne motorene. Etter at vi fikk tildelt en ny motor fungerte alt som det skulle.

2.5: Automatisk Kjøring:

Ettersom en stor del av prosjektet omhandlet å styre roboten ved joystick, altså manuell kjøring med lagring av informasjon som lysverdier, integrerte lysverdier osv. var det også naturlig å lage et program der roboten kunne kjøre automatisk. En stor fordel med dette var jo at mye av koden var allerede skrevet og at vi kunne sammenligne resultatene ettersom vi eliminerer feil som personen som styrer manuelt kan forårsake.

Hvordan vi fikk til å kjøre roboten automatisk gjennom løypen var i hovedsak ved bruk av en PID regulator som mottok verdier fra lyssensoren. En PID består av tre ulike forsterkninger som påvirker hvordan signalet ut vil bli. Altså en proposjonal, en integral og en derivert forsterkning.

En figur av hvordan PID regulatoren henger sammen.



Proposjonal-leddet:

Dette leddet forsterker den filtrerte lysverdien før den blir sendt til NXTen.

Dette er den første lysverdien som blir tatt inn i programmet.

```
Initial_Light = GetLight(SENSOR_3);
```

Lysverdien som kommer tilbake ved nåværende posisjon.

```
Lys(i) = GetLight(SENSOR_3);
```

```
filtLys(i) = 0.5*Lys(i)+0.3*Lys(i-1)+0.2*Lys(i-2);
Kp         = 0.9; Proposjonal-forsterkning
Avvik      = filtLys(i)-Initial_Light;
yPID(i)    = Kp*Avvik
```

Denne koden gir et eksempel på hvordan P leddet virker. Kjører roboten vekk fra det såkalte setpunktet som er 0 vil forskjellen mellom nullpunktet og den nåværende lysverdien øke, dermed vil også (Kp*Avvik) øke. Dette fører til at NXTen gir større og større pådrag til motorene for å rette opp avviket.

Dette leddet er i utgangspunktet nok til å kjøre roboten rundt racetracken. Men fordi at ved en viss forsterkning vil roboten bli ustabil og begynne å svinge mer og mer til den til slutt kjører utenfor banen.

Integral-leddet:

Dette leddet integrerer numerisk forskjellen mellom initial lysverdi og den filtrerte nåværende lysverdien, altså avviket i lysverdien og legger til tidligere integrerte verdier. Som vil si at når roboten kjører utenfor setpunktet vil I-leddet øke med tiden. Som igjen fører til mer og mer økning i pådraget til motorene jo lenger den avviker fra setpunktet. Ved å bruke proposjonal og integral forsterkerne sammen fikk vi en god regulator til å styre roboten gjennom IDE racetracken. Vi testet dette i praksis og var godt fornøyd med resulatene. Men dersom vi økte farten for å få ned tiden ville disse to leddene etterhvert bli ustabile og roboten begynne å svinge mer og mer rundt setpunktet, så dersom vi ville ha enda bedre resulater måtte vi innføre et nytt ledd til å bremse raske endringer.

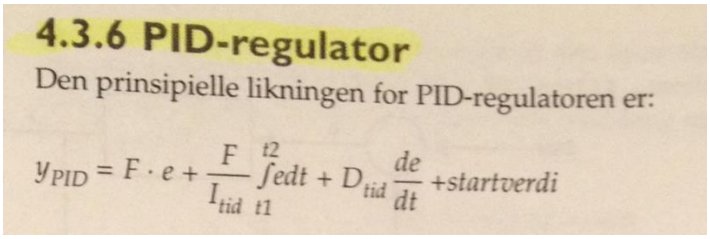
```
Avvik         = filtLys(i)-Initial_Light;
deltaTime(i)  = etime(newTime,oldTime);
integral_new  = deltaTime(i)*Avvik;
integral(i)   = integral_new;
Ki            = Kp/0.03; Integrator-forsterkning (0.03 ms)
yPID(i)       = Ki*integral(i)
```

Derivator-leddet:

Siste ledd i en PID regulator er derivasjons leddet. Dette leddet deriverer forskjellen mellom to målinger og som resultat får en vite hvor raskt lysverdien endrer seg. Poenget med et derivasjonsledd er å motvirke store, raske endringer i lysverdien.
D-leddet er det vi brukte lengst tid på å implementere, da det er litt mer komplisert å forklare og forstå virkningen av. Det første vi fant ut var at veldig små endringer i dette leddet gav veldig store utslag når roboten skulle kjøre. Vi begynte med å fjerne støy ved å innføre et filter vi kalte `filtDer(i)`. For ettersom den deriverte av støyen til lyssensoren ga veldig høye verdier var dette naturlig å begynne med. Men også med filter fikk vi veldig høye verdier og så oss nødt til å bruke dette leddet veldig forsiktig.

```
deltaTime(i) = etime(newTime,oldTime);
deltaLys     = filtLys(i)-filtLys(i-1);
filtLys(i) = 0.5*Lys(i)+0.3*Lys(i-1)+0.2*Lys(i-2);
derivert(i)  = (deltaLys ./deltaTime(i));
filtDer(i) = 0.5*derivert(i)+0.3*derivert(i-1)+0.2*derivert(i-2)
Kd         = 0.1; Derivat-forsterkning
yPID(i)     = Kd*filtDer(i)
```

Samler vi leddene ovenfor får vi en solid regulator for å kjøre automatisk i IDE racketracken. Vi tok utgangspunkt i en formel tatt fra en automasjonsbok som figuren nedenfor viser:



4.3.6 PID-regulator

Den prinsipielle likningen for PID-regulatoren er:

$$y_{PID} = F \cdot e + \frac{F}{I_{tid}} \int_{t1}^{t2} e\,dt + D_{tid} \frac{de}{dt} + startverdi$$

Her er koden vi brukte i programmet for å kjøre roboten automatisk:

```
%% K verdier til PID
Kp          = 0.9;
Ki          = Kp/0.03;
Kd          = 0.1;

yPID(i)      = Kp*Avvik+Ki*integral(i)+Kd*filtDer(i);
```

Verdiene vi valgte for (Kp og Kd) ble et resultat av mange forsøk med små justeringer for hver gang. Til Ki verdiene bruker vi Kp / tiden til den integrerte. Tiden til den integrerte er det samme som tiden i en runde i while-sløyfen. Dette fikk oss nærmere og nærmere et resultat vi kan se oss fornøyd med.

<u>Resultater fra kjøring gjennom løypen automatisk</u>

```
Brukt tid: 8.001 s

Areal: 169.7179 cm^2

Areal + tid= 8170.7179 poeng
```

2.6: Kreative Innslag:

2.6.1: Følge Hånden:

Hensikten med dette programmet var at roboten skulle følge etter en hånd eller en gjenstand i en viss avstand.

For å komme i gang med dette programmet måtte vi initialisere en ny sensor: ultralydsensoren. Denne sensoren måler avstand til et objekt.

Dette programmet var i utgangspunktet ganske enkelt å lage, og ble på en måte et grunnlag for labyrint programmet. Vi diskuterte også andre muligheter med dette programmet, for eksempel å lage en radar som søkte etter hindringer, evt. Søkte etter åpninger, men siden noen av de andre programmene tok ganske mye tid, rakk vi ikke å begynne med dette.

Det vi gjorde var å initialisere sensoren og rett og slett programmere den til å følge etter hånden om den var innenfor en viss avstand og rygge fra hånden dersom den var innenfor en annen gitt avstand.
Selve koden til dette ble så kort at vi har lagt den ved i rapporten:

```
Avstand(i) = GetUltrasonic(SENSOR_4);
    if Avstand(i)>0 && Avstand(i)<15
        motorA.Power = -10;
        motorB.Power = -10;
    elseif Avstand(i)>15 && Avstand(i)<30
        motorA.Power = 10;
        motorB.Power = 10;
    else
        motorA.Power = 0;
        motorB.Power = 0;
    end
```

Kort forklaring:
Dersom avstand er mellom 0 og 15, skal roboten rygge.
Dersom avstand er mellom 15 og 30, skal roboten kjøre fremover.
Dersom sensoren ikke registrerer noe skal den stå stille.

2.6.2: Labyrint:

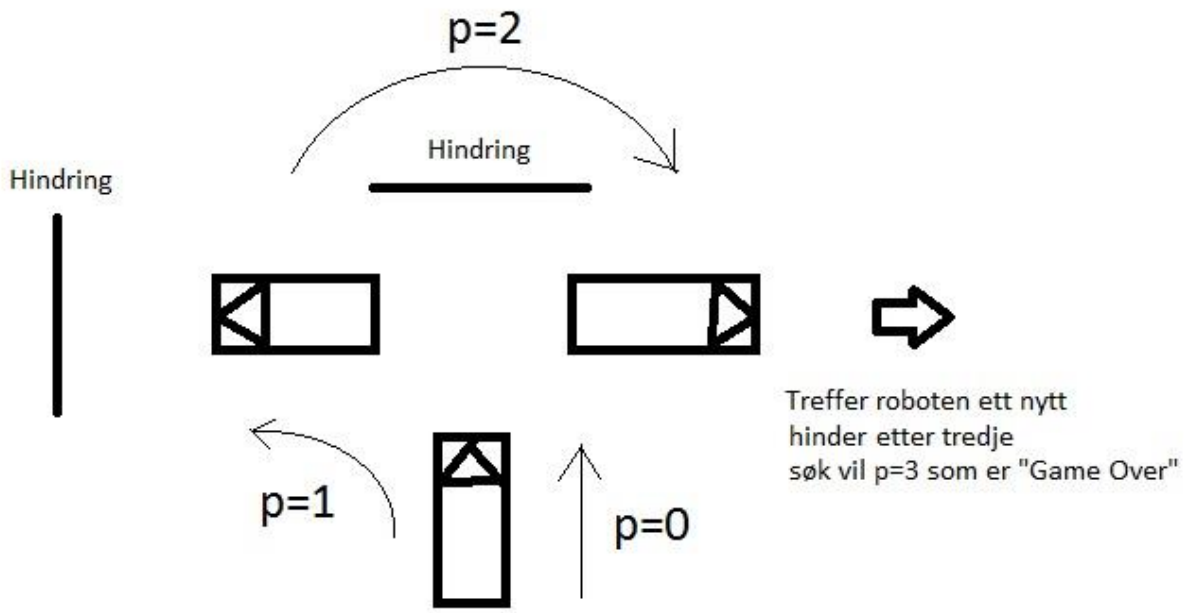
Hensikten med dette programmet var at roboten skulle greie å kjøre igjennom en «labyrint» eller en løype som vi hadde satt opp på forhånd. Roboten skulle kjøre rett frem så lenge det ikke var hindre i veien, og dersom den møtte på hindre skulle den stoppe og sjekke om den kunne kjøre til en av sidene. Dette ble gjort ved å dreie på motorC som styrte ultralydsensoren. Om roboten registrerte at det gikk an å kjøre til en av sidene skulle ultralydsensoren dreies tilbake til utgangsposisjon, og «kroppen» dreies i den retningen det var mulig å kjøre.

Dette prosjektet bød på mange utfordringer og mye hodebry. Et av de største problemene var rett og slett å få skrevet koden i riktig rekkefølge slik at roboten skjønte hva den skulle gjøre. I tillegg hadde vi litt problemer med at motorene ikke ville vente til en kommando var ferdig før den kjørte ut neste kommando.

Vi begynte med å skrive én stor if sløyfe med alle kommandoene som burde være med. Problemet med det var at den ble uoversiktlig, programmet sendte for mange kommandoer til NXT'en på en gang og vi hadde enda ikke funnet helt ut av hvordan vi skulle bruke tacholimits. Etter mye prøving og feiling for å få denne koden til å fungere ble vi nødt til å innse at vi nok hadde brukt feil metode fra starten.

For å få roboten til å bli ferdig med en kommando før Matlab sendte neste brukte vi «waitfor» kommandoen sammen med tacholimit. Dette gjorde at Matlab måtte vente på at motoren hadde nådd en gitt verdi før den sendte ut neste kommando. På grunn av tannhjul av diverse størrelser fikk vi et forhold som gjorde at motoren som styrte ultralydsensoren måtte rotere ca. 8 ganger for at sensoren skulle snurre én gang.

Det vi til slutt fant ut virket var å sette de forskjellige bevegelsene i forskjellige «moduser». Vi opprettet en modusvariabel «p» og satte de forskjellige bevegelsene til p=0, p=1, p=2 og p=3.
I p=0 skulle roboten kjøre rett frem så lenge det ikke var hindringer i veien, og rotere sensoren til venstre dersom den kom til en hindring. Dersom det ikke var hindring til venstre ble p=1.
I p=1 ble det foretatt en ny avstandsmåling og hvis avstanden var større enn en gitt verdi skulle den svinge hele roboten til venstre, og sette p=0 igjen slik at roboten kunne kjøre videre den veien.
Dersom avstanden i p=1 var mindre enn den gitte verdien skulle sensoren svinge 180 grader tilbake til høyre og sette p=2.
I p=2 ble det foretatt nok en ny avstandsmåling og roboten svingte til høyre dersom avstanden var over den gitte verdien, satte p=0 og kjørte rett frem den veien.
Men om avstanden i p=2 var mindre enn gitt verdi ble p=3, og programmet var ferdig. Da hadde roboten sjekket alle sider om det fantes en utvei men fant ingen.
For å gi en indikasjon på at roboten måtte gi tapt snudde den sensoren tilbake til utgangsposisjon, og NXT'en spilte av en låt fra Super Mario.



Figur med enkel forklaring til hvordan labyrint programmet fungerer.

2.6.3: Spille Super Mario Theme:

Det meste i dette programmet er egentlig hentet fra nettet, og linken til der vi hentet frekvenser og toner fra ligger som kommentar i programmet. Det vi gjorde var å skrive det litt om slik at det passet til NXT'en. Vi valgte likevel å ta med dette programmet fordi det viser enkelt hvordan man kan få NXT'en til å spille av en hvilken som helst melodi bare man kjenner de rette frekvensene og pausene imellom tonene, og i tillegg var dette programmet en slags inspirasjon til piano programmet og videre til morse programmet.
Utdrag av koden ser slik ut:

```
NXT_PlayTone(660, 100);
pause(0.150);
NXT_PlayTone(660, 100);
pause(0.300);
NXT_PlayTone(660, 100);
pause(0.300);
```

der (660, 100) betyr at den spiller av frekvens 660 i 100 ms.
Pause indikerer hvor lang tid det skal være mellom hver tone

2.6.4: Piano:

Piano programmet ble laget for å kunne bruke tastaturet som et enkelt piano samtidig som den tar opp og kan spille tilbake det du har spilt inn.
For å gjøre det enkelt har vi kun brukt én tastatur rad; (asdfghjkl), men her er det enkelt å legge til nye toner/halvtoner bare ved å lete frem frekvenser og tastaturverdier. Frekvensene har vi funnet på nettet og webadressen ligger som kommentar i begynnelsen av programmet. Tastaturverdiene er hentet ut fra ASCII tabell.
I tillegg har vi brukt kommandoen «waitforbuttonpress» som gjør at Matlab tolker tastaturtrykk i sanntid uten å måtte trykke Enter. Denne kommandoen ble funnet på mathworks.com.
Før man begynner å spille, er variabelen opptak satt til 0 som betyr at det ikke finnes noe opptak. Etter at man er ferdig med å spille blir opptak satt til 1, som betyr at det finnes et opptak, og avspillingsfunksjonen er tilgjengelig.
Variabelen p lagrer tastetrykk i ASCII kode og sammen med tellervariabelen (i) som blir oppdatert til (i+1) hver gang while sløyfen starter på nytt får vi lagret en celle med alle innspilte toner/tastetrykk
Det er også lagt inn en timer som tar tiden mellom hvert tastetrykk slik at når sangen spilles av på nytt så får vi rett pause mellom tonene.

Koden:

```
if p(i)== 97
            NXT_PlayTone(261, 100);
        elseif p(i) == 115;
            NXT_PlayTone(293, 100);
        elseif p(i) == 100;
```

sier at om man trykker på a, (ASCII verdi 97), så spilles frekvensen 261 som tilsvarer C4 på et piano i 100 ms.
Og om man trykker s, (ASCII verdi 115), så spilles frekvensen 293 som tilsvarer D4 på et piano i 100 ms.

Så er koden duplisert nedover i programmet og det er lagt inn nye ASCII verdier med tilsvarende frekvenser.

Når man er ferdig med å spille inn en sang kan man få Matlab/NXT til å spille den tilbake. Nå er alle tonene/tastetrykkene lagret i en egen celle, så avspillingsprogrammet spiller av den cellen på nytt og omformer igjen de lagrede ASCII kodene til frekvenser. Her bruker vi variabelen p(j) der p er variabelen som inneholder alle de forrige tastetrykkene, og (j) er en teller som blir oppdatert til (j+1) for hver gang while sløyfen starter på nytt. (tilsvarer (i) i innspillingsprogrammet.)

2.6.5: Morse:

Hensikten med dette programmet var å få NXT'en til å omforme bokstaver til morsekode og tilbake og på den måten kunne kommunisere med en annen NXT
Dette programmet består av en «krypteringsfil» der ASCII kode blir omgjort til morse (korte og lange pip fra NXT'en) og en «dekrypteringsfil» som mottar morsekoden og gjør om tilbake til ASCII kode.(og bokstaver) samt en hovedfil som behandler disse kommandoene
Krypteringskoden ser slik ut:

```
%% ASCII to morse
lang = 300;
kort = 100;
ingenT = 0;

if asciiNumber == 32        % = Space
    ms1 = kort;
    ms2 = lang;
    ms3 = kort;
    ms4 = lang;
    ms5 = kort;
    ms6 = lang;
```

Der lang = 300; betyr at lange pip varer i 300 ms.
Kort = 100 betyr at korte pip varer i 100 ms.
Og at ASCII-nummeret 32 (som tilsvarer mellomrom) blir omgjort til morsekoden: (.-.-.-)
For å få dette til å fungere måtte vi lagre alle cellene med ms1:ms6 for å få like store celler, så for de bokstavene som ikke har så mange pip har vi satt inn «ingenT = 0» som vil si at de som er merket ingenT får verdien og lengden 0 og ikke spilles av.

```
elseif asciiNumber == 101    % = e
    ms1 = kort;
    ms2 = ingenT;
    ms3 = ingenT;
    ms4 = ingenT;
    ms5 = ingenT;
    ms6 = ingenT;
```

Bokstaven «e» er f. eks. representert bare ved én prikk i morsealfabetet. Denne koden vil si at NXT'en spiller av én kort lyd og 5 «ikke-eksisterende» lyder som ikke har noen lengde.

Dekrypteringskode:

For å få Matlab til å tolke morsekoden har vi satt opp en egen tabell bestående av nuller og enere. Alle de forskjellige bokstavene har unike sammensetninger av nuller og enere slik vi bruker dette programmet.

Utdrag av tabellen vår: (a=1, z=26 og mellomrom=27. ukjente tegn / bokstaver blir automatisk satt til '*', nummer 42 i ASCII tabellen.)

```
morseABC{1} = [0 1];
morseABC{2} = [1 0 0 0];
morseABC{3} = [1 0 1 0];
morseABC{4} = [1 0 0];
```

Vi har lagt inn variabler som separerer de forskjellige tonene fra hverandre, pauser mellom toner og pauser mellom ord, samt to timere, en som tar tiden mellom de forskjellige målingene, (Det er denne målte differansen i tid som til slutt gjør at Matlab forstår hva slags signaler som skal tolkes.) Og en som blir resatt dersom det blir registrert lyder over en gitt verdi. Denne timeren gjør også at programmet slutter dersom det går 2.5 sekunder uten registrerte lyder over en viss verdi.

```
valuePause        = 0.5;
letterEnd         = 0.8;
sepMorse          = 0.25;
sepMorsePause     = 0.7;
timeToEnd         = 2.5;
```

«valuePause» setter tiden mellom hver enkelt tone til å være 0.5 sek.
«letterEnd» setter tiden mellom bokstaver til å være 0.8 sek.
«sepMorse» skiller mellom lange og korte lyder (kort = 100 ms, lang = 300 ms.)
«sepMorsePause» skiller mellom valuePause og letterEnd.
«timeToEnd» bestemmer at programmet er ferdig når det går over 2.5 sek. Uten at lyd er registrert.

Vi har også satt frekvensen på morsekoden vår til å være 440, og satt en variabel til å skille mellom morse og ikke morse for å filtrere bort bakgrunnstøy.

```
soundCut = 400;

sound(sound<soundCut)=0;
sound(sound>soundCut)=1;
```

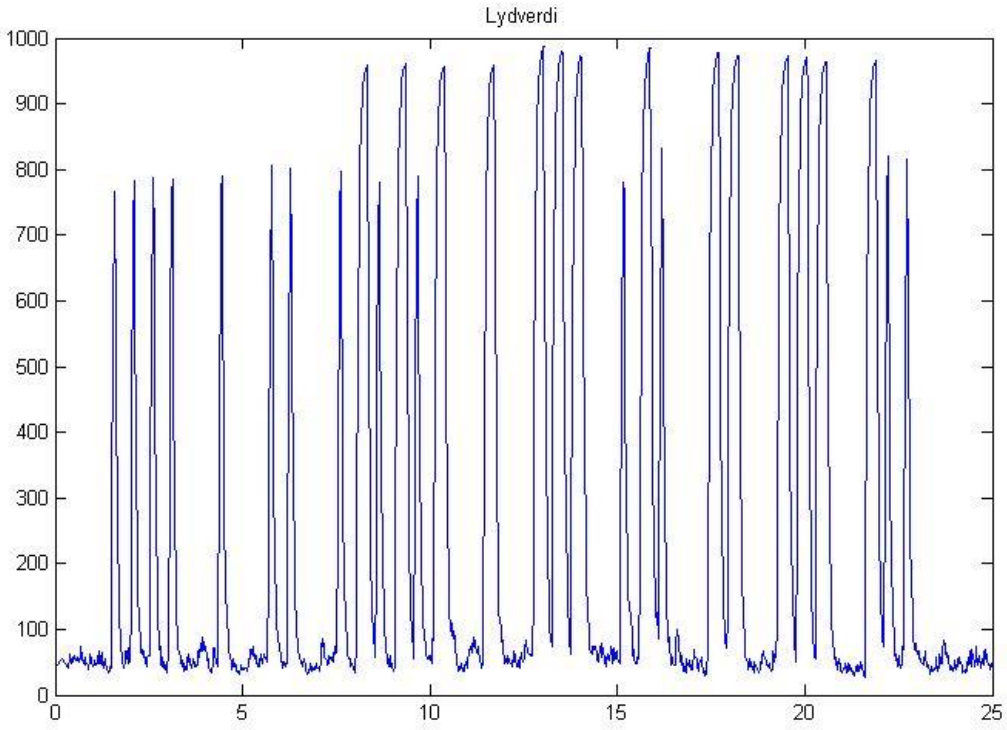
Med alle disse variablene på plass har vi fått en kode som skriver ut en vektor bestående av nuller(kort morse, under 0.25s), enere (lang morse, over 0.25s), toere (pause over 0.7s) og treere. (pause mellom nuller og enere i morseABC. Disse treerne blir fjernet i funksjonen)
For å få Matlab til å tolke den siste bokstaven i morsekoden, har vi satt inn denne koden for å være sikker på at vektoren slutter med 2:

```
if
morseCode1(end)~=2
morseCode1(end+1)=2;
end
```

Funksjonen vår kan da skille ut alle nuller og enere som ligger imellom hvert to-tall og omforme disse til ASCII kode, som vi igjen overfører til klartekst:

```
[tekstInAscii]=morseToAscii(morseTot);
```
(gjør morsekode om til ASCII)

```
disp(['Morsekode: ',char(tekstInAscii)]);
```
(gjør ASCII om til klartekst)



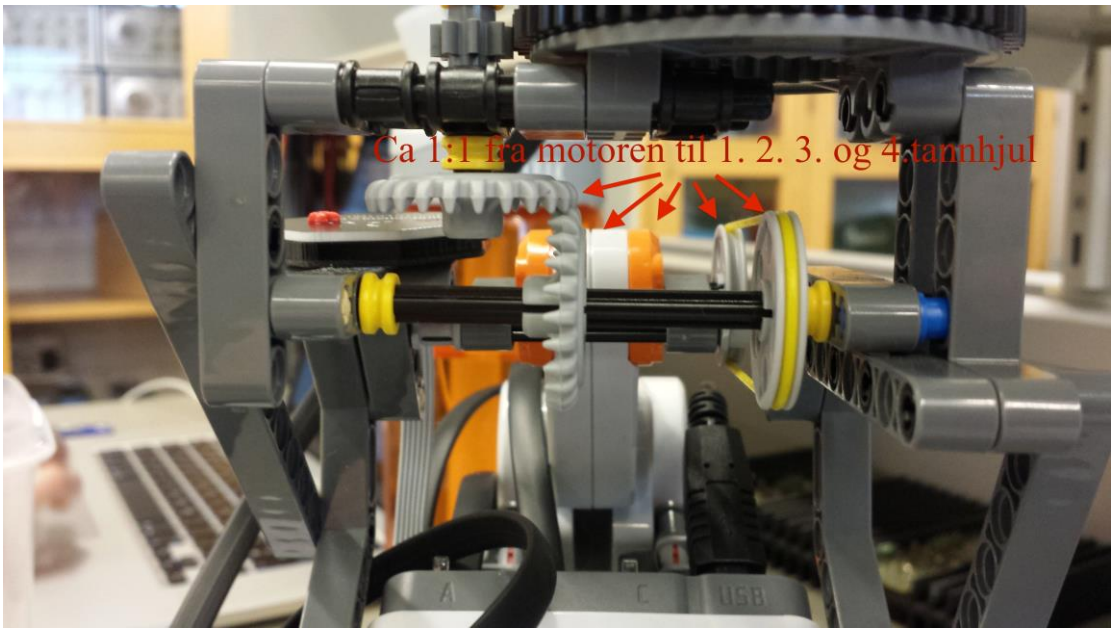
Figuren viser hvordan mottatt morse blir gjort om til en graf der alle toppene representerer en prikk eller strek i morsealfabetet. De tynneste strekene representerer kort støt (0) og de tykkeste strekene representerer langt støt(1). De lave verdiene imellom strekene representerer pauser som bestemmer om du fremdeles er på samme bokstav(3) eller om du starter en ny bokstav(2)
Hvis man ser på verdiene imellom ca. 7 og 11 på x-aksen, så ser man klart at der er det: kort, lang, kort, lang, kort, lang, som tilsvarer mellomrom.

2.6.6: Måle Areal Og Omkrets:

Vi har laget et program hvor roboten kan måle, ved hjelp av ultralydsensoren, både areal og omkrets av en hvilken som helst figur hvor roboten kan være inni. Vi plotter også esken live i et polarplott, ved vinkel og radiusen målt.

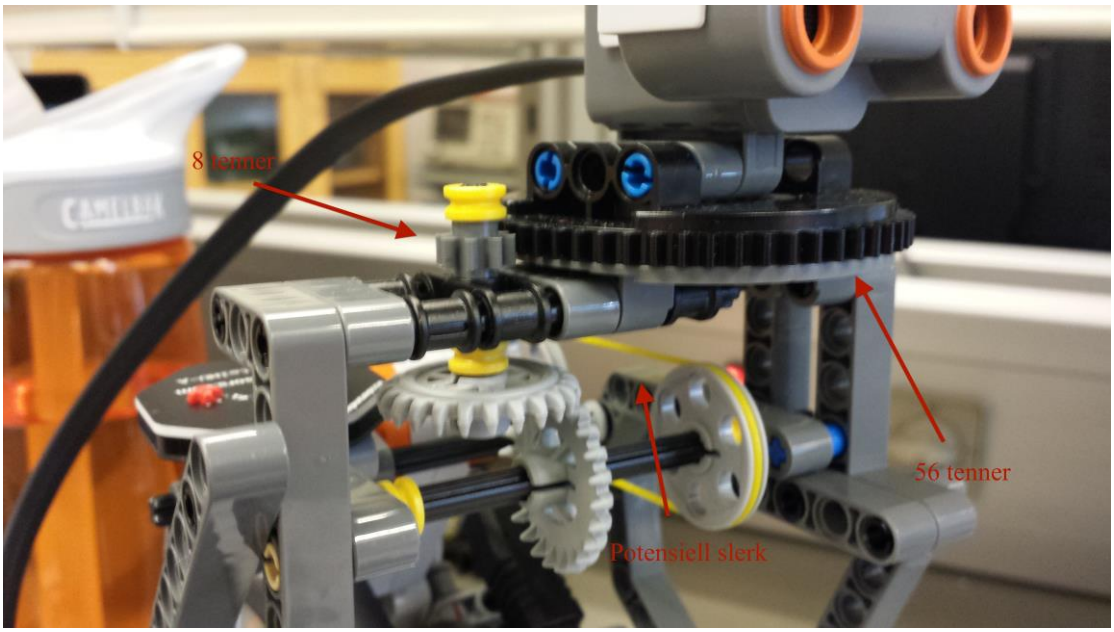
Vi bygde roboten slik at ultralydsensoren ble stående på toppen av NXTen, og var festet til motor c ved hjelp av tannhjul. Dette valgte vi fordi vi ville ha en utfordring og ikke gjøre likt som alle andre. Nå fikk vi en rekke forholdstall vi måtte forholde oss til. 360 grader for ultralydsensoren var da 2770 grader for motoren.

Dette tallet kom vi frem til ved at vi fant forholdstallene mellom de forskjellige hjulene. Frem til og med hjul nr.4 var forholdet ca 1:1, så dette var ikke tatt med i regnestykke.



Vi kom frem til at det store hjulet hadde 56 tenner, og det lille hadde 8 tenner.

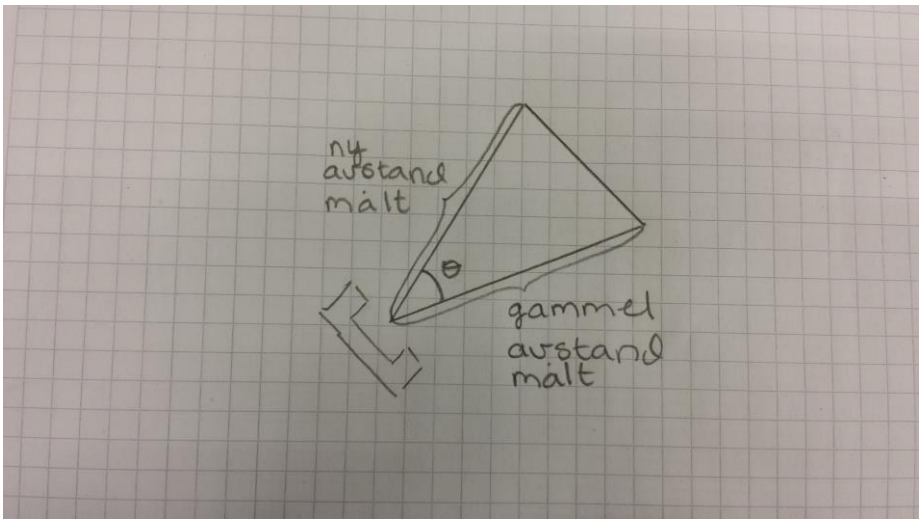
Det lille hjulet måtte da ta ca 7 runder før det store hadde gått en hel runde. Men når vi testet TachoLimit 2520 (360*7) ville ikke den gå en hel runde. Det er nemlig en del potensiell slerk i strikken som binder motoren og aksen tannhjul 2 og 3 roterer rundt.



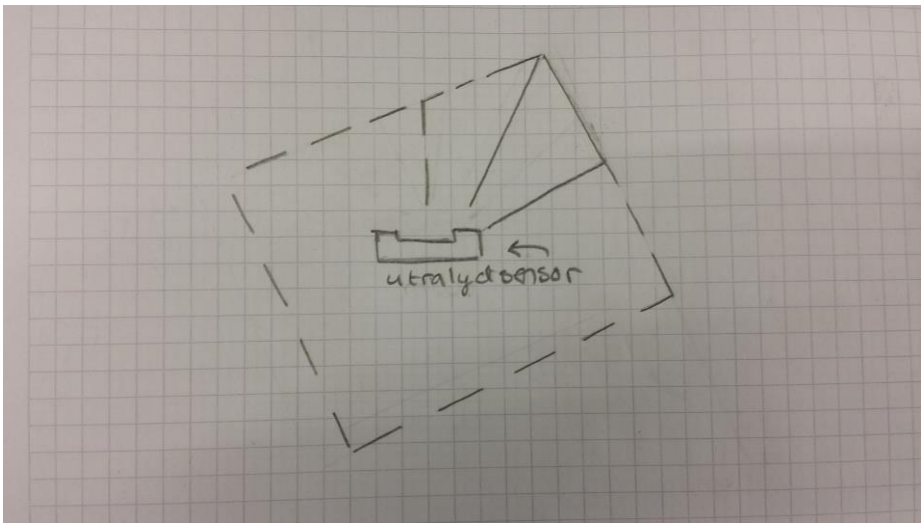
Det er også en del potensiell friksjon mellom tannhjulene, ettersom det er vanskelig å få de til å passe perfekt i hverandre. Vi måtte derfor teste oss frem til hvilken TachoLimit som ville få den til å rotere perfekt. Med 2520 som utgangspunkt gikk dette fort. Vi kom frem til et tall som var litt større, 2770. Forholdstallet var nå 7.694:1.

Første gangen vi skulle regne arealet sendte vi en TachoLimit til NXTen slik at den skulle rotere en hel runde og skanne i sanntid. I mens den roterte leste vi vinkelen fra NXTen mellom hver måling. Ut i fra dette regnet vi ut arealet ved hjelp av arealsetningen. Nå fikk vi så mange målinger sensoren klarte å ta, men det varierte veldig fra gang til gang.

Under er det illustrert hvordan sensoren i teorien tar målinger og hvordan han regner det ut. Du ser her den gamle avstanden, og den nye. Vinkelen imellom leser vi fra motoren, og alt dette blir lagret i celler. Hvor vi senere henter ut all informasjonen igjen.



Under ser du et utklipp av hvordan sensoren vil regne areal i teorien. Du ser her en av de trekantene som tilslutt vil tilsvare hele boksen. Alle målingene blir lagret i celler, hvor den ligger klar til vår bruk. Vinkelen mellom målingene leser vi av motoren, regner gjennom forhåndstallene og lagrer disse vinklene også i celler.



Arealet blir regnet ut fra de ”trekantene” de forskjellige målingene genererer, vi hadde vinklene og lengden på målingene, det var bare å regne ut. Siden NXTen hadde så mye data å gå igjennom før hver måling, ble resultatet ganske unøyaktig. Vi ville ha mer kontroll over NXTen slik at resultatet ville bli mer konsekvent. Vi bestemte oss da for en løsning, slik at det ville bli færre, men kontrollerte målinger. Vi lagde en while-sløyfe der sensoren bevegde seg omtrent 1.3 grader for så å ta en måling, vi bestemte oss for denne vinkelen fordi vi fant et forhold mellom tida det tok å skanne og nøyaktigheten av resultatet. Vi kunne i teorien delt gradene opp i enda mindre verdier en 1, som er den minste TachoLimiten du kan sende til motoren og slik fått et mer nøyaktig resultat en om vi hadde plasser sensoren rett på motoren. Men om vi hadde delt den opp i så små vinkler ville det tatt altfor lang tid. Vi ville ha TachoLimiten = 10, og siden 360 grader for sensoren er 2770 grader for motoren, ville dette gi oss 277 målinger. Graden mellom målingene fant vi ved å dividere 360 grader med antall målinger.

Nå når vi satte TachoLimiten lik 10, ville den nå ikke måle samtidig som den roterte, den ville stoppe for hver trekant, som vist i tegningen. For å forbedre ytterligere lot vi den ta 10 målinger mens den sto stille, og da ta gjennomsnittet av disse 10 målingene. Dette gjorde den for hver av de 277 målingene. Det tok nå betydelig lengre tid å måle og regne ut et areal, men resultatene veide opp for dette. Helt til slutt la vi til avviket som kom fra ultralydsensoren når den roterte. Vi målte radiusen fra roteringsaksen til selve sensoren og la til formelen for areal av en sirkel inn i matlab etter while-løkka vår.

Her har vi også lagt til avviket som ikke blir målt med ultralydsensoren. Nemlig det som er fra selve sensoren sine ”øyne” og inn til rotasjonsaksen. Vi målte dette til å bli nøyaktig 3 cm. Under ser du formelen hvor vi summerer arealet fra alle målingene:

arealTot(i) = sum(areal);

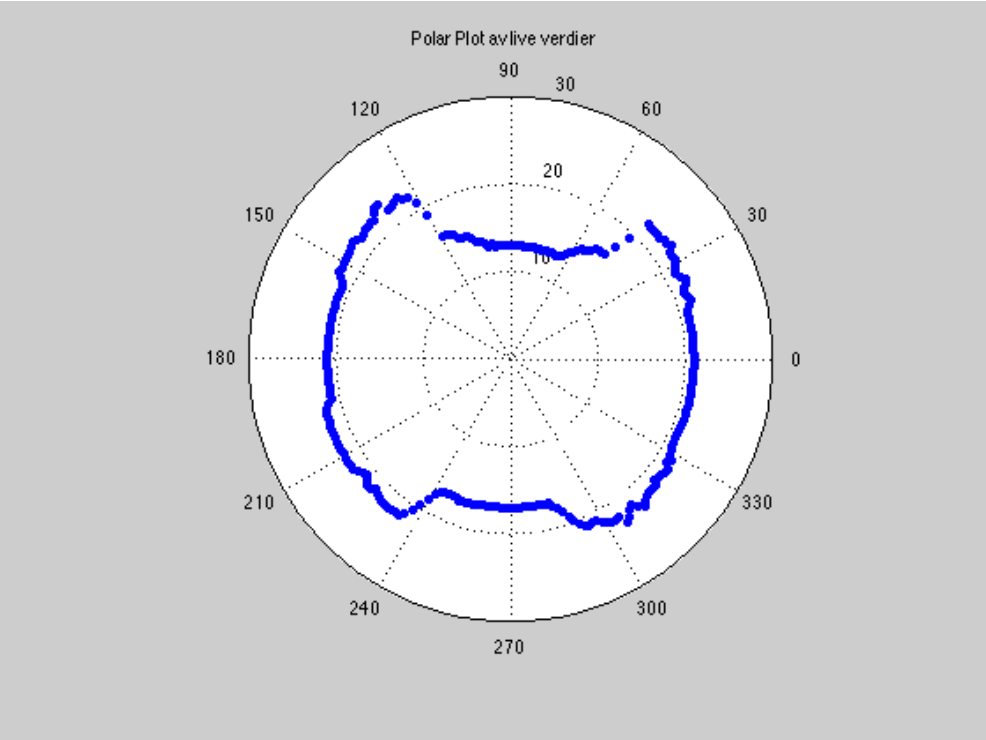
Her ser du formelen hvor vi legger til arealavviket lagd av sensorhodet:

totAreal = arealTot(end)+2*pi*3^2;

Arealet varierte til slutt kun med 15 cm^2 fra det reelle arealet.

Delprogram polarplot:

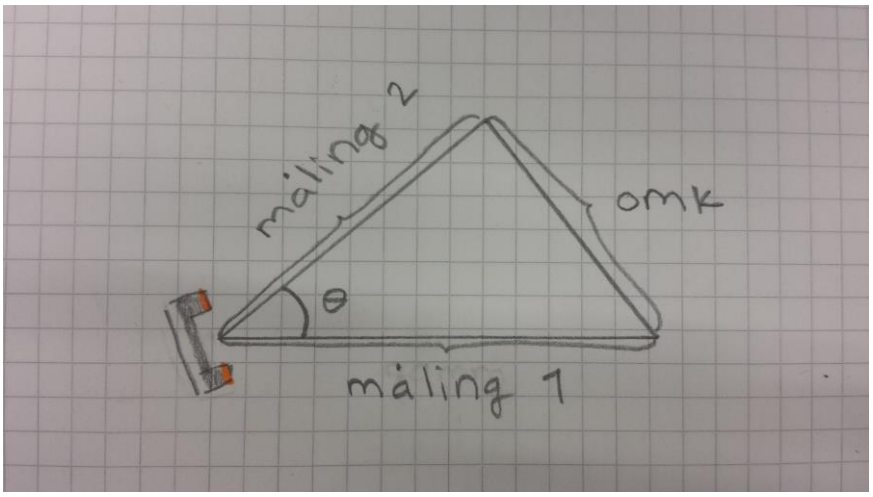
Når vi plottet arealet i polarplot, brukte vi vinkelen vi leste fra NXTen, og la de sammen for hver måling, slik fikk vi en reell plassering av prikkene som utgjorde plottet. Radiusen for hver av prikkene, er hver tilhørende avstandsmåling. Plottet er inne i while-løkken, og vi får da en live plott. Vi syns et polarplot ville være det beste, med tanke på at vi ikke måtte regne ut noen ekstra verdier av avstandsmålingene og bare kunne sette disse rett inn i plottet.



Delprogram omkrets

Når vi skulle regne ut omkretsen brukte vi den samme dataen som med arealutregningen, men istedenfor å bruke arealsetningen som vi brukte ved den utregningen, brukte vi nå cosinussetningen som gir oss lengden av endestykke på en trekant. Cosinussetningen er gitt ved: $c^2 = a^2 + b^2 - 2abcos(vinkelC)$. Da var det bare å sette inn våre verdier, med tanke på at vi hadde alt vi trengte.

I tegningen under tilsvarer ´omk´ ´c´i cosinussetningen og måling 1 og måling 2 tilsvarer a og b. Vinkelen er theta, og den leser vi fra cellene hvor vi har lagret vinklene mellom hver måling.



Formelen vår ble til slutt seende slik ut.

```
omk(i)= sqrt(((avstandFilt(i))^2)+((avstandFilt(i-1))^2)-
(2*avstandFilt(i)*avstandFilt(i-1)*cosd(deltaVinkel(i))));
```

Her tilsvarer avstandFilt(i) og avstandFilt(i-1) a og b i cosinussetningen. DeltaVinkel(i) er den tilsvarende vinkelen til avstandsFilt(i) og avstandsFilt(i-1), altså vinkelC i formelen.
Omkretsen for hver trekant blir lagret i celler, som på slutten av programmet blir summert og den endelige omkretsen kommer ut. Denne varierte med kun 5 cm fra den egentlige omkretsen.

2.6.7: Hovedmeny:

Formålet med programmet var å lage en hovedmeny som vi kunne bruke til å starte alle andre programmer fra.
Denne lagde vi ved å bruke funksjonen «menu» som genererer en meny med knapper der man enkelt kan skrive inn hva som skal stå på de enkelte knappene. Menyen tilegnet hver enkelt knapp til en numerisk verdi, og de verdiene igjen til hver enkelt funksjon.

Utdrag fra koden:

```
if valg==1
      run gr_1320_labyrint.m
      valg=1;
   elseif valg==2
      run gr_1320_manuell_lys.m
      valg=2;
   elseif valg==3
      run gr_1320_PID_slider.m
      valg=3;
```

Dette betyr at om man velger knapp 1, så kjøres programmet som er tilegnet den knappen, altså gr_1320_labyrint.m
Alle programmer er linket opp mot knappene på samme måte. Setter valg verdi tilbake etter programmet har kjørt, fordi vi bruker clear commando i de andre programmene.

3: Konklusjon:

Program 1: Manuell Kjøring:

Problemstilling: Få roboten til å kjøre gjennom IDE sin racetrack ved hjelp av joystick.
Kort oppsummert har vi intitialisert 2 motorer og tilegnet de forskjellige aksene på joysticken til forskjellig motorpådrag som gjør at vi kan manøvrere roboten igjennom banen. Måten vi har gjort dette på fører dessverre til et problem. Når man styrer joysticken 45 grader ut til en hvilken som helst side skjer det ingenting. Kreftene nuller ut hverandre. På grunn av en defekt motor hadde vi allerede brukt litt for lang tid på å få dette til å fungere, så vi bestemte oss for at det fungerte bra nok til at vi kunne la dette ligge og heller begynne på nye programmer.

Program 2: Automatisk Kjøring:

Problemstilling: Få roboten til å kjøre igjennom IDE sin racetrack automatisk.
Her måtte vi initialisere lyssensoren og hente kontinuerlige målinger som ble lagt inn i en PID regulator (Proportional, integral derivate regulator) som kontinuerlig integrerte og deriverte avviket fra startmålingen for å styre roboten inn mot optimal bane.
Løsningen vår fungerer bra, men ikke perfekt. Roboten har en tendens til å styre ganske langt ut når man kommer til svingen, men den greier å hente seg inn, og vi kjører raskere og med mindre avvik automatisk enn vi gjør manuelt.

Program 3: Følge Hånden:

Problemstilling: Få roboten til å følge etter hånden eller et hvilket som helst objekt.
Nok en sensor: ultralydsensoren ble initialisert, og var umiddelbart ganske enkel å skjønne. Den måler avstand, så vi satte bare inn en avstand som tilsvarer at roboten skal følge etter hånden, og en annen avstand der roboten skal rygge fra hånden.
Programmet fungerer etter hensikten.

Program 4: Labyrint:

Problemstilling: Får roboten til å kjøre igjennom en oppsatt bane ved hjelp av ultralydsensoren.
Vi initialiserer motorC for å få ultralydsensoren til å kunne bevege seg rundt uavhengig av resten av roboten. Vi legger så inn en kode som gjør at ultralydsensor skal bevege seg i et visst mønster dersom den treffer en hindring, og hva den skal gjøre dersom det ikke finnes hindringer. Løsningen tok tid å komme frem til, og ikke minst å få til å skrive riktig, men programmet fungerer etter hensikten.

Program 5: Spille Super Mario Theme:

Problemstilling: Få NXT'en til å spille av en lyd/melodi.
Dette programmet er i stor grad hentet fra internett, og bare skrevet litt om for å passe NXT'en. Programmet er først og fremst noe vi brukte for å få satt oss litt inn i hvordan man bruker NXT til å spille lyder, og inspirerte piano programmet og morse programmet.
Fungerer etter hensikten.

Program 6: Piano:

Problemstilling: bruke tastaturet som piano og få NXT'en til å spille av de ønskede tonene, i tillegg til å kunne ta opp «sangen» og spille av på nytt.
Videreføring av å spille av en lyd automatisk. Vi programmerte en rekke toner/frekvenser til å samsvare med forskjellige taster på tastaturet, og satte inn en funksjon som gjør at NXT'en registrerer tastetrykk i sanntid. Tastetrykkene blir lagret i en celle slik at de kan spilles av på nytt.
Fungerer etter hensikten.

Program 7: Morse:

Problemstilling: Få NXT'en til å kode om klartekst til morsekode og morsekode til klartekst.
Å kode tekst til morsekoder var en relativ grei sak. Vi brukte ASCII tabellen for å finne tallverdier som tilsvarer alfabetet med små bokstaver, og på samme måte brukte vi morsetabellen for å finne de tilsvarende bokstavene uttrykt ved korte og lange streker.
Å kode morsekoder tilbake til tekst var litt mer krevende da man hadde flere ting å ta hensyn til. (pauser imellom bokstaver, pauser imellom pip, lengde på pip osv.)
Dette løste vi ved å sette inn variabler som skiller mellom alle de forskjellige pipene og pausene.
Vi lagde også en egen alfabet-tabell som bestod av nuller og enere som tilsvarte korte og lange pip, og fikk Matlab til å lagre lydene i en vektor som nuller og enere med toere som indikerte hopp fra en bokstav til neste, og tilslutt overførte vi disse til ASCII kode og videre til klartekst.
Programmet fungerer etter hensikten.

Program 8: Måle Areal Og Omkrets:

Problemstilling: vi ønsket å bruke ultralydsensoren til å måle areal og omkrets av et hvilket som helst objekt ved å snurre rundt seg selv, ta målinger og legge sammen disse målingene i 2 forskjellige formler for å regne ut areal og omkrets av objektet.
Vi fikk ultralydsensoren til å snurre ca. 1.3 grader og foreta 10 målinger der gjennomsnittet ble lagret, så fortsatte sensoren å snurre 1.3 grader med tilsvarende målinger helt til den var kommet en runde rundt. På denne måten får vi masse små trekanter som vi kan regne ut arealet at, og til slutt legges det sammen og vi får et totalareal.
Omkrets blir regnet ut ved hjelp av cosinussetningen som gir lengden på motstående katet i hver trekant, og ved å legge alle disse katetene sammen får vi tilslutt et total areal.
Programmet fungerer etter hensikten.

Program 9: Hovedmeny:

Problemstilling: lage en meny som kan brukes til å sette i gang akkurat det programmet vi vil kjøre.
Her har vi brukt funksjonen «menu» som i seg selv genererer en meny med knapper som kan trykkes på. Trykker man på knappen «automatisk kjøring» så starter programmet.
Fungerer etter hensikten.

4: Kilder / referanser

Nettsider:
Mindstorms
http://www.mindstorms.rwth-aachen.de/trac/wiki/Documentation
[Hentet: 18.10.2013]

Demorse (22 Sep 2008)
http://www.mathworks.com/matlabcentral/fileexchange/21491-demorse/content/demorse.m
[Hentet: 12.11.2013]

Mathworks. (2013) *Find Peaks.*
http://www.mathworks.se/help/signal/ref/findpeaks.html
[Hentet: 13.11.2013]

MATLAB Central (2012)
http://www.mathworks.com/matlabcentral/answers/44227-finding-local-minimums-maximums-for-a-set-of-data
[Hentet: 13.11.2013]

Wikipedia (14 November 2013)
http://en.wikipedia.org/wiki/PID_controller
[Hentet: 18.10.2013]

Wikipedia (25 October 2013)
http://en.wikipedia.org/wiki/Piano_key_frequencies
[Hentet: 19.10.2013]

MATLAB Central (6. Juli 2012)
http://www.mathworks.com/matlabcentral/answers/101415
[Hentet: 19.10.2013]

MiktroTik (15. Mai 2008)
http://wiki.mikrotik.com/wiki/Super_Mario_Theme
[Hentet: 17.10.2013]

Utleverte dokumenter:

#1 Prosjekt i ING100 ingeniørfaglig innføringsemne
#2 Introduksjon til Matlab og LEGO NXT ved bruk av RWTH-Mindstorms Toolbox
#4 Numerisk integrasjon, numerisk derivasjon og filtrering.

Litteratur:

Industriell måleteknikk for automatikerfaget **Av Bjørnar Larsen 1996, utgave 2**

ISBN13 9788241202223 **Forlag** Vett og viten