Universitetet
i Stavanger

DET TEKNISK-NATURVITENSKAPELIGE FAKULTET

MASTEROPPGAVE

Studieprogram/spesialisering:	Vår.................semesteret, 20...... Åpen / Konfidensiell
Forfatter:	…………………………………………… (signatur forfatter)
Fagansvarlig: Veileder(e):	
Tittel på masteroppgaven: Engelsk tittel:	
Studiepoeng:	
Emneord:	Sidetall: ………………… + vedlegg/annet: ………… Stavanger, ……………….. dato/år

Kamerabasert conveyortracking implementert i ABBs kabinettsystem for lakkeringsroboter.

Peter Leupi

15. Juni 2014

Sammendrag

Conveyortracking er mye brukt innen automasjon i industrien på grunn av mulighetene det gir til fortløpende behandling av objekter på en conveyor. Dette gjelder kanskje i aller største grad for robotarmer, på grunn av deres høye fleksibilitet til å orientere seg i rommets tre dimensjoner.

Denne oppgaven er skrevet for ABB Robotics, etter et ønske om å se på mulighetene til å erstatte deres enkoderstyrte conveyortracking med maskinsyn.

Til dette har det blitt tatt i bruk et Cognex smartkamera, som har blitt integrert direkte inn i kabinettsystemt til lakkeringsroboten. Det har blitt skrevet relevante C++ klasser som kommuniserer med kameraet og behandler posisjonsdata sendt fra kameraet.

På grunn av forsinkelse i kameraet har det vært nødvendig å ta i bruk prediksjonsalgoritmer på signalet, samt esktrapolasjon på grunn av kameraets lave bilderate.

Resultatene fra simuleringer og tester i det virkelige systemet, viser at kameraet er i stand til å gi tilstrekkelig god tracking når det behandlede kamerasignalet sammenliknes med enkoderen. Men det er fremdeles mye videre arbeid som må gjøres før det eventuelt kan produktifiseres.

Forord

Jeg vil takke veilederene mine, Morten Mossige og Nina Svensen, for god veiledning og tilbakemelding gjennom arbeidet på masteroppgaven.

Jeg vil også takke Tore Fuglestad for eksepsjonell hjelp når jeg har møtt på problemer i jungelen som er IPS.

Takk til kollegene mine i ABB, som har vært hjelpsomme under hele oppgaven.

Til sist vil jeg takke familien min for støtte gjennom de fem årene mine som student ved UiS.

Innhold

1 Innledning **1**
1.1 Dagens løsning for conveyortracking . . . 1
1.2 Oppgavebeskrivelse . . . 3
1.3 Begrensning av oppgaven . . . 4

2 Conveyortracking med ABB roboter **5**
2.1 Laboppsett . . . 7
2.2 Programmering av tracking . . . 7
2.3 Softwaren som tar seg av tracking . . . 11
2.3.1 IpsDevice klassen . . . 11
2.4 Sanntidsanalyse og logging i RobView . . . 13

3 Conveyortracking med kamera **15**
3.1 Tidligere arbeid . . . 15
3.2 Cognex smartkamera . . . 17
3.3 Utfordringer og begrensninger . . . 18
3.3.1 Oppløsning . . . 18
3.3.2 Synsfelt . . . 18
3.3.3 Bildeforvrengning . . . 18
3.3.4 Dybde . . . 19
3.3.5 Utførelsestid og forsinkelse . . . 19
3.3.6 Valg av linse . . . 19
3.4 Kameratracking . . . 21
3.5 Objektmodellering . . . 21
3.6 Kantdeteksjon (Canny metoden) . . . 22
3.6.1 Glatting . . . 23
3.6.2 Finne gradienter . . . 24
3.6.3 Ikke-maksimum suppresjon . . . 25
3.6.4 Thresholding . . . 26
3.6.5 Kantfølging ved hysterese . . . 26
3.7 Deteksjon av hjørner . . . 28
3.8 Gjenkjenning av objekt fra modellens egenskaper . . . 29
3.9 PatMax algoritmen . . . 31

4 Programmering av smartkameraet **33**
4.1 Bildeoppsett . . . 34
4.2 Objektgjenkjenning . . . 36
4.3 Kommunikasjon . . . 37
4.4 Kalibrering av kameraet . . . 38

5 Integrering av kamera på PIB **41**
5.1 Valg av grensesnitt mellom kamera og PIB 41
5.2 Implementering i IPS 43
5.3 Implementering av UDP kommunikasjon med kameraet i IPS 46
5.4 Implementering av kameraklassene 47
5.5 `CamDevice` klassen 49
5.6 `CamEncoder` og `CamSync` klassene 52

6 Utvikling og testing av prediksjonsalgoritmer **53**
6.1 En nærmere kikk på kamerasignalet 53
6.2 Lesing av signalet i IPS 58
6.3 Metode for å ta hensyn til varierende forsinkelse 61
6.4 Lineært avvik mellom kamerasignalet og enkodersignalet 65
6.5 Prediksjonsalgoritmer 68
6.5.1 Konstant fart mellom sampler 68
6.5.2 Konstant fart med tilbakekobling 70
6.6 Simulering av `ConveyorTracker` 73
6.7 Testing av algoritmene 74

7 Resultater **75**
7.1 Resultat på Matlabtester 75
7.2 Konstant hastighet mellom sampler 77
7.2.1 Sammenlikning med enkodertrackeren *uten* lineær avvikskorreksjon 80
7.2.2 Sammenlikning med enkodertrackeren *med* lineær avvikskorreksjon 83
7.3 Konstant fart med tilbakekobling 86
7.3.1 Sammenlikning med enkodertrackeren 90
7.4 Resultater i praksis på PIB 93
7.5 Konstant fart mellom sampler 94
7.5.1 Konstant fart på conveyoren 94
7.5.2 Varierende fart på conveyoren 95
7.6 Konstant fart med tilbakekobling 96
7.6.1 Konstant fart på conveyoren 96
7.6.2 Varierende fart på conveyoren 97

8 Konklusjon og videre arbeid **99**

Referanser **101**

9 Forkortelser **103**

A Appendiks **105**
A.1 Kildekode 105
A.1.1 `CamDevice` (`hpp/cpp`) 105
A.1.2 `UDPserver` (`hpp/cpp`) 115

A.1.3 `CamEncoder` (`hpp/cpp`) 118
A.1.4 `CamSync` (`hpp/cpp`) 123

Figurer

1 Skisse av dagens løsning for conveyor-tracking. 2
2 Enkoderpulsenes og synksignalets vei gjennom PIB-kortet. 3
3 Kameraet skal integreres i mikrokontrolleren på PIB-kortet for å erstatte enkoder og synkbryter. 4
4 Overblikk over et conveyortracking system.[20] 5
5 Laboppsettet som er tatt i bruk for prosjektet. 7
6 (a):Enkoderen er festet direkte på beltet, slik at den spinner med samme hastighet som conveyoren. (b): Synkryteren er en metalldetektor som gir utslag når kobberklistremerkene passerer under sensoren. 8
7 **a**: Roboten får signal fra synkbryteren og venter til objektet når Sync offset, her 300 mm. **b**: Roboten beveger seg ned til punktet `p20`. **c** og **d**: Roboten holder relativ posisjon til objektet i 600 mm. **e**: Roboten flytter seg og dropper objektet. **f**: Roboten venter på nytt objekt. 10
8 Prinsippskisse for conveyortracking med enkoder i IPS. 12
9 Sanntidsobservering av signaler og parametere i RobView. Her er det `ConveyorTracker` signalene og parameterene som er vist. 13
10 Logging av signaler i RobViews Signal Analyzer. 14
11 Kameraoppsettet som er tatt i bruk i prosjektet. 16
12 Cognex smartkamera. 16
13 Cognex In-Sight 5400 smartkamera. 17
14 Brennvidden f er gitt av avstanden fra midten av linsen til brennpunktet [1]. 18
15 Barrel forvrengning. 19
16 Korrigert bilde. 19
17 Tomat foran hvit bakgrunn 22
18 Rødkomponenten minus Grønnkomponenten. 22
19 Thresholdet, binært bilde. 22
20 Det beregnede sentrum av tomaten er representert med en grønn firkant. . 22
21 Bilde som skal gjennomføres Canny kantdeteksjon på. 23
22 Bildet etter gaussisk filtrering. 24
23 Absoluttverdien til gradienten til det glattede bildet. 25
24 Resultat etter non-max suppression algoritmen. 26
25 Eksempel på resultat etter høy og lav thresholding. 27
26 Eksempel på endelig resultat etter Canny kantdeteksjon. 27
27 Bilde som skal detekteres hjørner i. 28
28 De detekterte hjørnene. 28
29 Objekt som skal gjenkjennes i søkebildet. 30
30 Søkebildet. 30
31 Matchet bilde. Punktmatchene representeres med de røde og grønne sirklene. 30
32 Kameraet har blitt koblet til i In-Sight Explorer. 33

33 Submenyen Start->Set Up Image. Her bestemmes parametere for bildetagning. . . . 34
34 Tilpasning av bildetagningsparametere er fullført. Start->Set Up Image. . 36
35 Programmet har laget en modell av objektet. . . . 37
36 Ser at objektet har blitt detektert, og har pikselkoordinatene 396.5,244.5. 37
37 UDP-strengen settes sammen i en egen meny. Ved å velge *Add*, kan man velge blandt de tilgjengelige variablene. Her finner man blandt annet posisjon og orientering til objektet, og med å skrive et skript i SpreadSheet, kan man i tillegg få ut et lokalt timestamp for bildetagning. . . . 38
38 I kalibreringsmenyen kan man ta bilder av sjakkmønster med kjent rutestørrelse for å kalibrere for bildeforvrengning. . . . 39
39 Resultat av kalibrering. . . . 39
40 Ønsket integrering av smartkameraet. . . . 41
41 Prinsippskisse for kommunikasjon mellom kamera og IPS. . . . 43
42 Skisse av arkitekturen for klassene som styrer conveyortracking med enkoder i IPS. . . . 44
43 Skisse for ønsket implementering av kameraet i IPS. . . . 45
44 Forenklet UML-diagram av `UDPserver` klassen. Plusstegn betyr at medlemmet av klassen er `public`, mens minustegn betyr at det er `private`. . . 47
45 Grensesnitt for å kommunisere med kameraet implementert i IPS. 48
46 Tenkt implementering av kameraklassene, der alle de gule klassene arver fra `IpsInputDevice`. . . . 49
47 Flytdiagram for konstruktøren til `CamDevice`. . . . 50
48 Flytdiagram for metoden `lesUdpOgKalkulerTing()`. . . . 51
49 Posisjon plottet mot kameraets timestamp. . . . 54
50 Posisjon plottet mot kameraets timestamp (Forstørret). . . . 54
51 Starten av plotet: Blå: Objektets posisjon plottet mot kameraets timestamp for bildetagning. Rød: Gjennomsnittlig stigningskurve. . . . 56
52 Midten av plotet: Blå: Objektets posisjon plottet mot kameraets timestamp for bildetagning. Rød: Gjennomsnittlig stigningskurve. . . . 56
53 Høyre del av plottet: Blå: Objektets posisjon plottet mot kameraets timestamp for bildetagning. Rød: Gjennomsnittlig stigningskurve. . . . 57
54 Avviket er størst i høyre kant av bildet. Dette kan komme av at kameraets akse står noe skjevt på conveyorplanet, eller av at kamerakalibreringen ikke er perfekt. . . . 57
55 Et overblikk over tidspunkt for forskjellige hendelser fra bildet blir tatt, til data mottas i IPS. . . . 59
56 Posisjonssignalet plottet mot den lokale klokken på IPS idét det mottas. . 60
57 Blå: Posisjonssignalet plottet mot den lokale klokken på IPS idét det mottas. Rød: Det samme signalet plottet mot kameraklokken, dvs. timestampet fra kameraet, men tidsforskjøvet slik at de to signalene ligger oppå hverandre. . . . 60

58 Øverst: Tidsdiagram for kamera og IPS. Nederst vises de estimerte bildetagningstidspunktene (De gule sirklene) på IPS klokken, som nå er plassert riktig relativt til den initielle antakelsen. De blå ringene viser tidspunktet IPS faktisk leser verdiene. 62
59 Blå: x-posisjon plottet mot estimert bildetagningstidspunkt. Rød: Den samme posisjonsdataen plotet mot samplingstidspunkt (I Matlab). 64
60 Blå: Tidsforskjellen mellom de estimerte bildetagningstidspunktene er lik forskjellen mellom timestampene kameraet sender, som nesten alltid er lik 60 ms (når kameraet er programmert til å ta bilde hvert 60. ms). Rød: Tidsforskjellen mellom avlesningstidspunkt er alltid hele multipler av 16, ettersom det kun sjekkes om ny data er tilgjengelig hvert 16. ms. 64
61 Blå: Posisjon lest fra kameraet. Rød: Logget signal fra enkodertrackeren. . . 65
62 Estimering av avvik fra enkoderposisjon som funksjon av objektets posisjon i bildet. 66
63 Etter avvikskorreksjon: Blå: Posisjon lest fra kameraet. Rød: Logget signal fra enkodertrackeren. 67
64 Estimer avvik mellom enkoder og kamera som funksjon av posisjonen til kameraet med lineær feilkorreksjon anvendt. 67
65 Rød: Rådata. Blå: Prediktert posisjon. Som man ser bommer prediksjonen når signalet er ulineært, og prediksjonen får sagtannform. 70
66 Plot av prediksjon med tilbakekoblet utgang. 72
67 Med denne metoden får man en glattere kurve, i bytte mot noe mer avvik i bratte svinger. 72
68 Blå: x-posisjon plottet mot estimert bildetagningstidspunkt. Rød: Den samme posisjonsdataen plotet mot samplingstidspunkt (I Matlab). 77
69 Blå: x-posisjon plottet mot estimert bildetagningstidspunkt. Rød: Den samme posisjonsdataen plotet mot samplingstidspunkt (I Matlab). 78
70 Ser at periodetiden til det ekstrapolerte signalet er 16 ms, som ønsket. . . 78
71 Ved å øke det initielt antatt forsinkelse, oppnår man at det predikterte signalet ligger lengre foran det avleste. Her er det brukt 200 ms. initielt antatt forsinkelse. Da oppnås det at prediksjonen når en gitt posisjon $\approx$ 200 ms før det avleste signalet (når farten er konstant). 79
72 Det grønne signalet er rådataen plottet mot estimert bildetagningstidspunkt. Det predikerte signalet forsøker å treffe disse samplene. 79
73 Blå: Simulert conveyor-tracker fra kameraprediksjon. Rød: Logget signal fra enkodertrackeren. 80
74 Blå: Simulert conveyor-tracker fra kameraprediksjon. Rød: Logget signal fra enkodertrackeren. 81
75 Blå: Simulert conveyor-tracker fra kameraprediksjon. Rød: Logget signal fra enkodertrackeren. 81
76 Blå: Feilen $e = ct_kamera - ct_enkoder$ som funksjon av tiden. Den røde linjen representerer $e = 0$. 82

77 Blå: Feilen $e = ct_kamera - ct_enkoder$ som funksjon av tiden. Den røde linjen representerer $e = 0$. . . . 82
78 Estimering av lineært avvik som funksjon av objektets posisjon i bildet. . 83
79 Blå: ct-signalet beregnet fra kamera med lineær feilkorreksjon anvendt. Rød: Logget enkodertracking. . . . 84
80 Blå: ct-signalet beregnet fra kamera med lineær feilkorreksjon anvendt. Rød: Logget enkodertracking. . . . 84
81 Blå: ct-signalet beregnet fra kamera med lineær feilkorreksjon anvendt. Rød: Logget enkodertracking. . . . 85
82 Feilen $e = ct_enkoder - ct_kamera$ (i millimeter) som funksjon av tiden etter korreksjon for lineært avvik. Rød linje representerer $e = 0$. . . . 85
83 Blå: Estimert posisjon basert på Algoritme 3. Rød: Rådata fra kameraet. . 86
84 Estimatet gir brå start idét objektet kommer i synet for kameraet. . . . 87
85 Forstørrelse av området før objektet kommer ut av synsfeltet til kameraet. 87
86 Lavpassfiltrering av farten ble droppet ettersom det førte til en del kluss på tilbakekoblingen. . . . 88
87 Det ble istedet valgt å dempe den første fartsutregningen, som gir et nokså pent resultat. . . . 88
88 Ved å øke oppdateringsraten til 1 kHz (hvert millisekund), ser man at denne algoritmen gir en penere kurve enn den som ikke tar i bruk tilbakekobling (ingen brå hakk). . . . 89
89 Ved å øke oppdateringsraten til 1 kHz (hvert millisekund), ser man at denne algoritmen gir en penere kurve enn den som ikke tar i bruk tilbakekobling (ingen brå hakk). . . . 89
90 Hele plottet. Blå: ct-signalet beregnet fra kamera. Rød: Logget enkodertracking. . . . 90
91 Starten av trackingen. Blå: ct-signalet beregnet fra kamera. Rød: Logget enkodertracking. . . . 91
92 Slutten av trackingen. Blå: ct-signalet beregnet fra kamera. Rød: Logget enkodertracking. . . . 91
93 Avvik . . . 92
94 Hele plottet. Blå: Kameratrackeren. Rød: Enkodertrackeren. . . . 94
95 Starten av plottet. Blå: Kameratrackeren. Rød: Enkodertrackeren. . . . 94
96 Slutten av plottet. Blå: Kameratrackeren. Rød: Enkodertrackeren. . . . 94
97 Avvik. . . . 94
98 Hele plottet. Blå: Kameratrackeren. Rød: Enkodertrackeren. . . . 95
99 Starten av plottet. Blå: Kameratrackeren. Rød: Enkodertrackeren. . . . 95
100 Slutten av plottet. Blå: Kameratrackeren. Rød: Enkodertrackeren. . . . 95
101 Avvik. . . . 95
102 Hele plottet. Blå: Kameratrackeren. Rød: Enkodertrackeren. . . . 96
103 Starten av plottet. Blå: Kameratrackeren. Rød: Enkodertrackeren. . . . 96
104 Slutten av plottet. Blå: Kameratrackeren. Rød: Enkodertrackeren. . . . 96
105 Avvik. . . . 96

106 Hele plottet. Blå: Kameratrackeren. Rød: Enkodertrackeren. 97
107 Starten av plottet. Blå: Kameratrackeren. Rød: Enkodertrackeren. 97
108 Midten av plottet. Blå: Kameratrackeren. Rød: Enkodertrackeren. 97
109 Avvik. 97

1 Innledning

Conveyortracking er et viktig konsept i automasjonsindustrien, på grunn av mulighetene det gir til kontinuerlig behandling av objekter på samlebånd (eng.: conveyor). Dette gjelder kanskje i aller høyeste grad for robotarmer, på grunn av deres høye fleksibilitet til å orientere seg i rommets tre dimensjoner.

ABBs lakkeringsroboter er intet unntak. Her tas conveyortracking i bruk for å fortelle hvor robotene skal sikte lakk-applikatorene sine, blant annet på bilkarosseri som kommer kjørende forbi på samlebånd. I dagens system for conveyortracking brukes en tofaseenkoder i kombinasjon med et synkroniseringssignal fra en synkbryter, for eksempel en laserbryter, til å bestemme retning, fart og posisjon til objektet på samlebåndet. Denne formen for tracking er begrenset til én dimensjon: retningen på conveyoren. Det betyr at alle objektene som skal behandles må leveres til roboten med samme orientering og posisjon på tvers av conveyoren dersom man ønsker samme resultat hver gang.

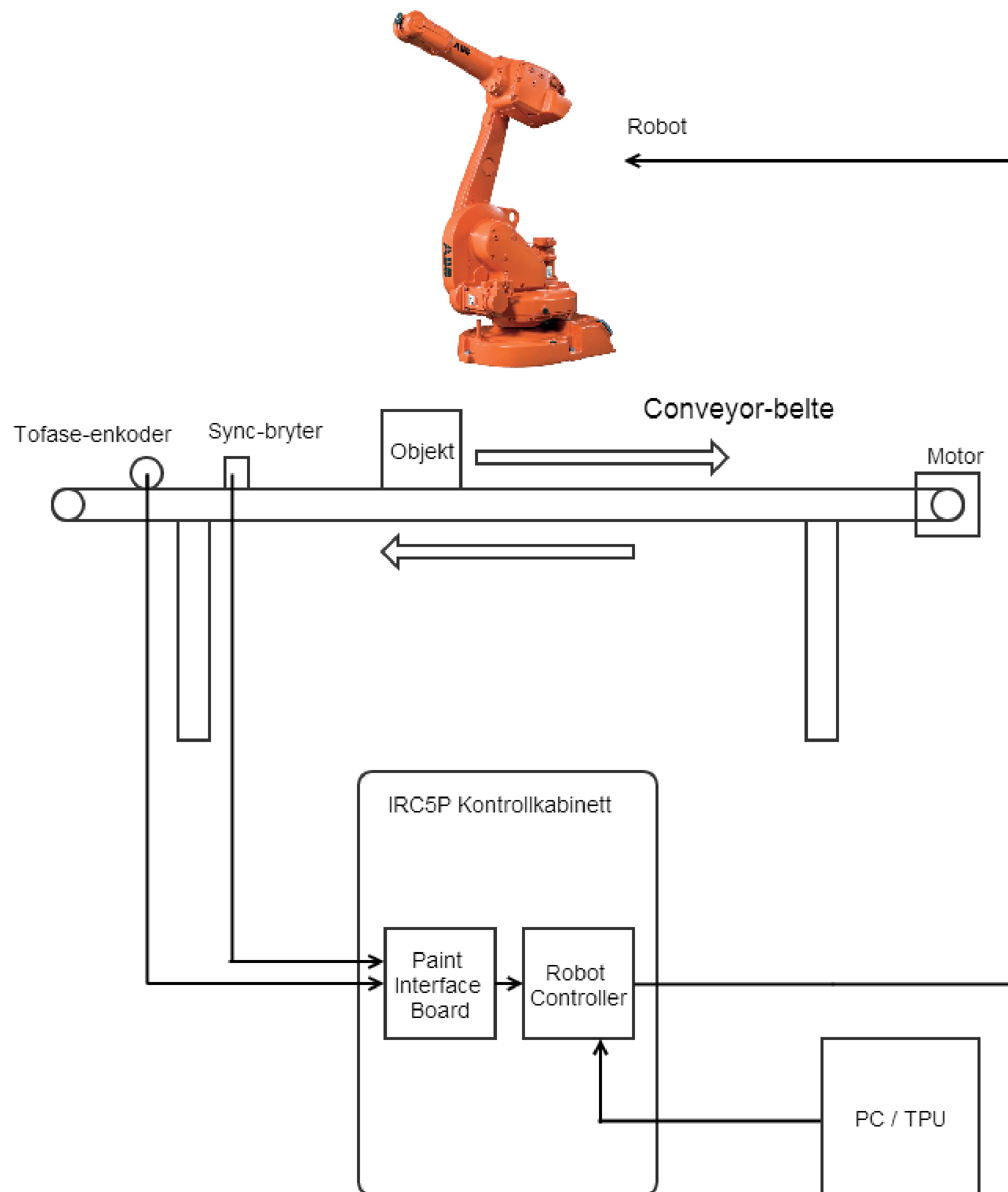
Denne oppgaven er skrevet for ABB Robotics, etter et ønske om å se på mulighetene til å ta i bruk maskinsyn til å erstatte enkoderen for conveyortracking. Deres ønske har vært å ta i bruk et smartkamera og implementere det direkte i robotens kabinettsystem. Oppgaven er i utgangspunktet å emulere oppførselen til enkoderen og synkbryteren ved hjelp av smartkameraet, og å se hvor godt kameraet presterer i forhold til enkoderen.

1.1 Dagens løsning for conveyortracking

La oss begynne med å gå igjennom hvordan dagens løsning for conveyortracking ser ut. En enkel skisse av oppsettet er vist i Figur 1.

Tofaseenkoderen er av den optiske typen, og fungerer slik at den sender ut to pulstog med en frekvens som er proporsjonal med farten på conveyoren. Enkoderen kan være festet enten med et hjul som triller sammen med beltet, eller direkte på motoren som driver beltet. Pulstogene fra enkoderen skapes av at en lyskilde er plassert bak et hjul med to hullete spor som snurrer sammen med beltet, med optiske sensorer på andre siden av hullene. Ettersom hullesporene er litt forskjøvet i forhold til hverandre, oppnås en faseforskyvning på de to pulstogene som kan brukes til å bestemme retningen på conveyorbeltet.

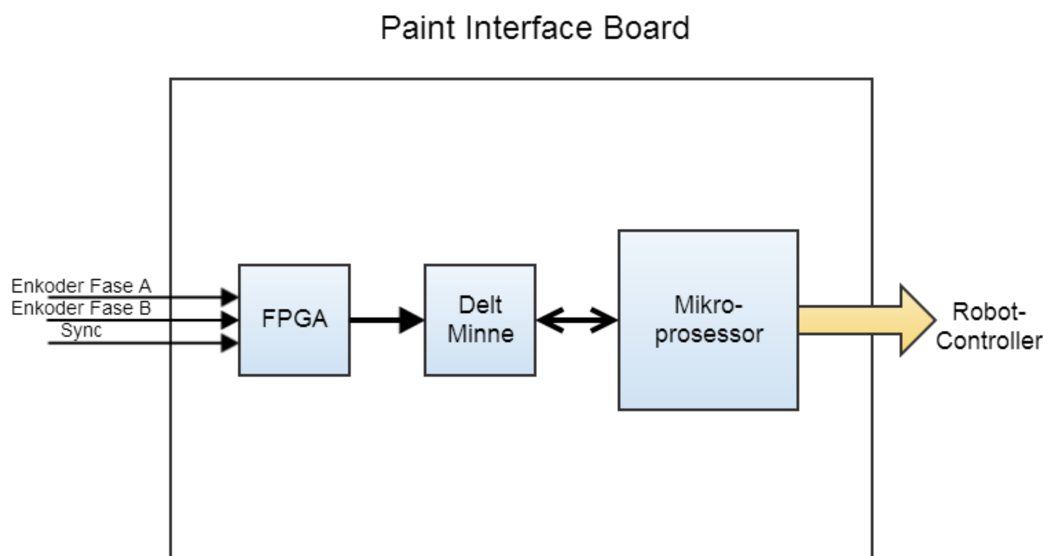
Den andre sensoren som brukes er en synkroniseringsbryter, referert til som synkbryteren, som gir en puls idét et objekt passerer den på samlebåndet. Synkbryteren er ofte en optisk bryter som brytes når noe kommer mellom kilden og den optiske sensoren, men også andre typer er i bruk. Dette er ikke viktig, for resultatet er det samme, nemlig en puls idét objektet passerer bryteren.



Figur 1: Skisse av dagens løsning for conveyor-tracking.

Disse to sensorene, enkoderen og synkbryteren, brukes i kombinasjon for å tracke objekter på samlebåndet.

De tre signalene (To fra enkoderen og ett synksignal) sendes inn i *Paint Interface Board* (PIB) i kontrollkabinettet, der hjernen(e) til roboten sitter. Her går pulstogene fra enkoderen inn på en FPGA som teller pulsene. Med et visst tidsintervall skriver FPGAen antall pulser som har blitt detektert inn til et felles register som deles med mikroprosessoren på PIB-kortet, sammen med et timestamp for når FPGAen skrev til registeret. Mikrokontrolleren leser disse tallene med gjevne mellomrom, og beregner med dette nye verdier for farten og posisjonen til objektet på beltet. Figur 3 viser en skisse av signalenes vei gjennom PIB-kortet.



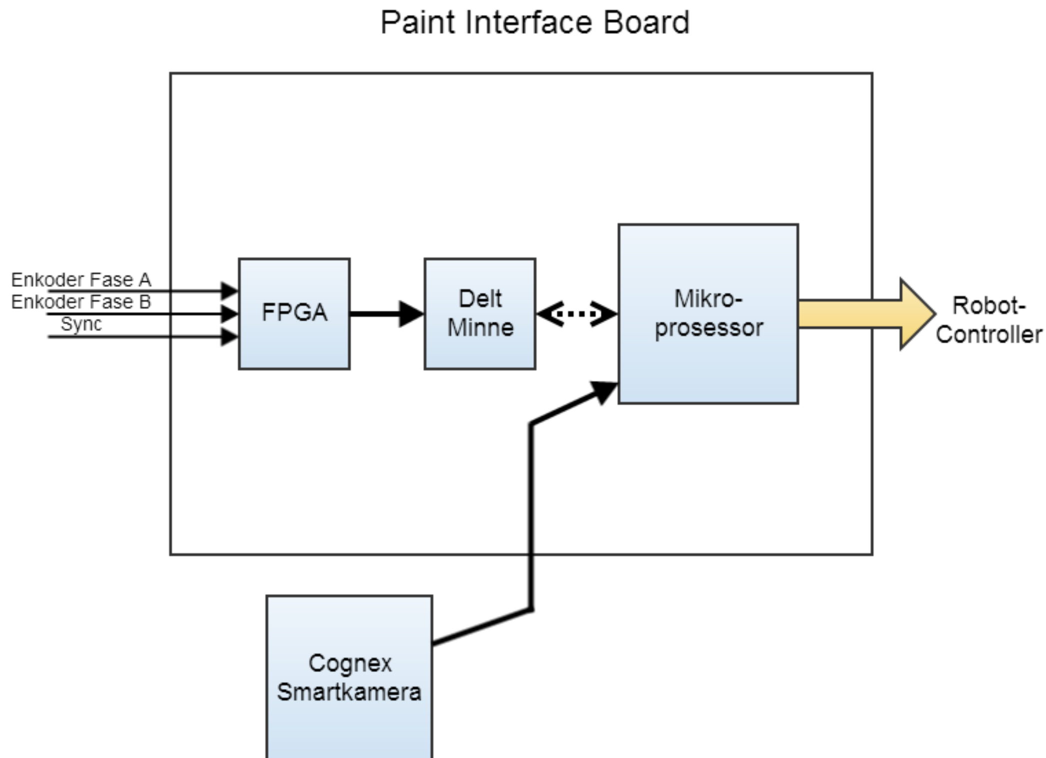
Figur 2: Enkoderpulsenes og synksignalets vei gjennom PIB-kortet.

Informasjonen som beregnes på mikrokontrolleren kommuniseres så videre til hoveddatamaskinen i systemet, som kontrollerer roboten.

1.2 Oppgavebeskrivelse

Oppgaven som ble gitt av ABB, er å erstatte dagens trackingsensorer, nemlig enkoderen og synkbryteren, med et smartkamera som kontinuerlig tracker objektene på conveyoren. Kameraet skal programmeres og integreres i systemet som kjører på mikrokontrolleren, og relevante klasser må skrives for å tilpasse kameraet til de klassene som tar seg av conveyortracking i dagens system. En ønsker å sammenlikne hvor godt kameraet presterer i forhold til enkoderen. Tanken er i første omgang å emulere enkoderens og synkbryterens virkemåte, slik at mye av det som tar seg av tracking i dag fremdeles kan tas i bruk. Ved å velge denne innfallsvinkelen unngår man å måtte endre eller erstatte hele biblio-

teket som i dag tar seg av tracking, noe som ville ha vært en altfor stor oppgave til å la seg gjennomføre i løpet av en masteroppgave (ref. softwareteamet hos ABB). Kameraet skal fungere som en selvstendig sensor, slik at ingen PC er nødvendig for å drive trackingsystemet.



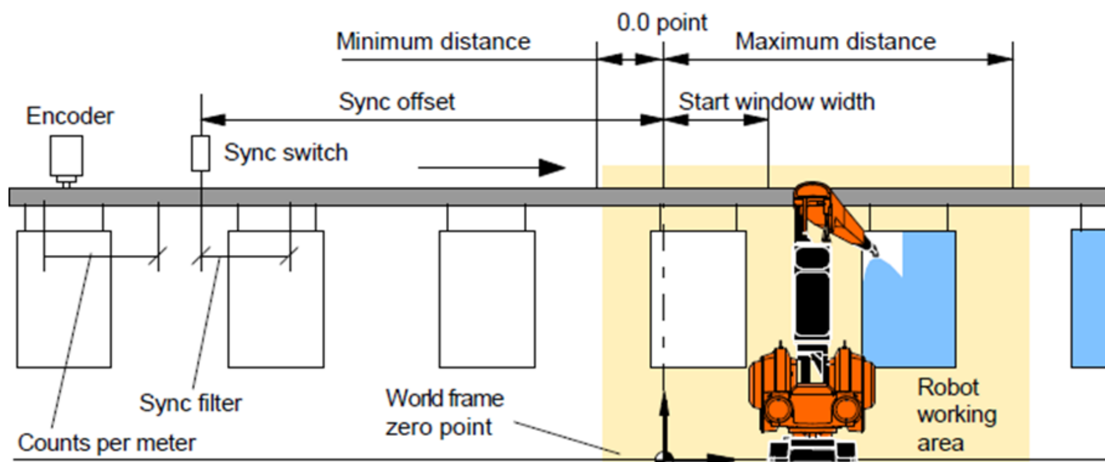
Figur 3: Kameraet skal integreres i mikrokontrolleren på PIB-kortet for å erstatte enkoder og synkbryter.

1.3 Begrensning av oppgaven

For å gjøre oppgaven gjennomførbar, gjøres det noen begrensninger. Oppgaven vil fokusere på å få implementert grunnleggende kameratracking i robotsystemet. Det vil kun bli sett på tracking av ett objekt om gangen. Det vil i tillegg ikke bli tatt hensyn til lakkeringsprossessen i dette prosjektet, som i seg selv byr på utfordringer på grunn av deformering av objektene som trackes. Disse punktene vil bli stående som kilder til videre arbeid.

2 Conveyortracking med ABB roboter

Til å begynne med er det viktig å forstå hvordan Conveyortracking utføres med ABBs lakkeringsroboter. Et overblikk over et conveyorsystem er vist i Figur 4.



Figur 4: Overblikk over et conveyortracking system.[20]

Som beskrevet i innledningen er en enkoder og en synkbryter installert på conveyoren. Det er ved hjelp av disse to sensorene trackingen utføres. Enkoderen sender ut pulser, og antall pulser som er sendt ut er direkte proporsjonalt med distansen som conveyoren har flyttet seg. En parameter som kalles *Counts per meter* (pulser per meter) definerer dette forholdet [20]. Forholdet varierer fra system til system, og må tilpasses for det gitte conveyorsystemet. Avstanden som conveyorbeltet har flyttet seg, Δl, beregnes enkelt med formelen

$$\Delta l = \frac{antall\ talte\ pulser}{pulser\ per\ meter}$$

I tillegg tas det hensyn til hvilken retning conveyoren beveger seg ved hjelp av enkoderens tofasethet, og Δl får dermed ulikt fortegn avhengig av retningen på bevegelsen. I software leses antall talte pulser hvert 16. millisekund, og conveyorens posisjon beregnes med den samme raten.

Synkbryteren (Sync switch) gir en puls når et objekt passerer bryteren. Signalet fra synkbryteren filtreres i software for å unngå falske synksignaler og prell. Når synkbryteren gir utslag, plasseres objektet i kø til tracking, og dersom det er det eneste objektet i køen, vil roboten "koble seg på" objektet når objektet kommer innenfor en definert rekkevidde. Man sier da at objektet er *tilkoblet* (eng.: connected). Trackingen av objektet kan da begynne ved å beregne Δl siden synkbryteren slo inn.

Det er en del andre parametere som må nevnes: Disse er *Sync offset*, *Start window width*, *Minimum distance* og *Maximum distance* (se Figur 4).

- *Sync offset* - Definerer hvor mange millimeter roboten skal vente etter at synkbryteren har slått inn før den kobler seg på et objekt. *Nullpunktet* for objekter på conveyoren, dvs. der hvor deres posisjon har posisjon lik null, er gitt av posisjonen til synkbryteren pluss Sync offset. Eller sagt på en annen måte, når synkbryteren slår ut på et objekt, vil objektet ha posisjon −*Sync offset*, og når det har beveget seg *Sync offset* millimeter, vil det ha posisjon lik null.

- *Start window width* - Dersom roboten er opptatt med en annen oppgave når objektet kommer innenfor rekkevidde (dvs. at objektet er kommet forbi Sync offset), definerer denne parameteren hvor langt objektet kan passere før det blir droppet. Dersom objektet passerer forbi startvinduet mens roboten er opptatt, vil det altså bli droppet helt og fjernet fra køen.

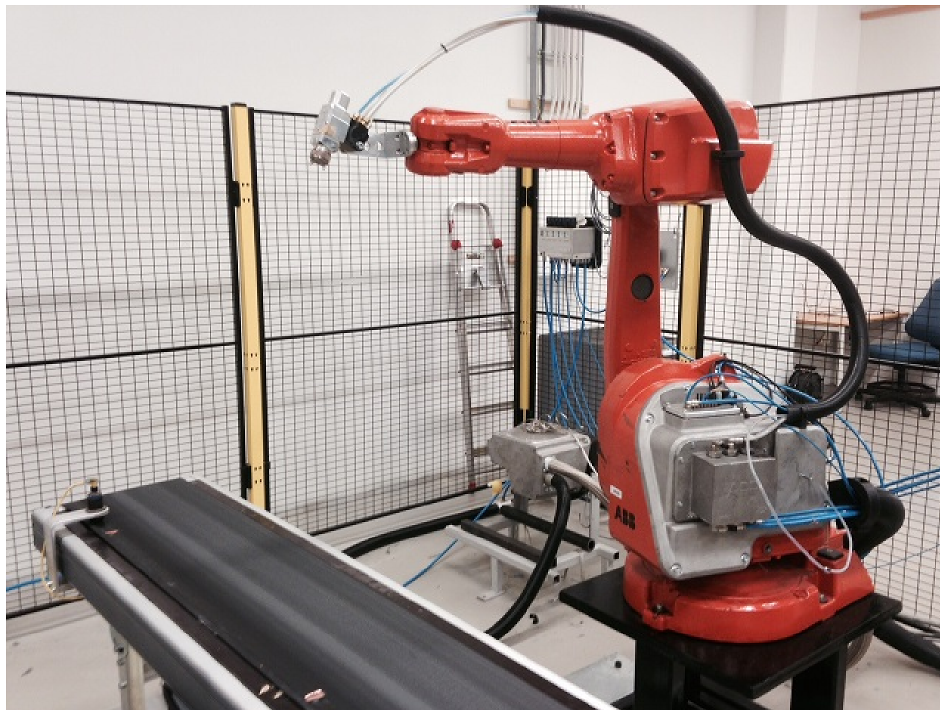
- *Minimum distance* - Definerer hvor langt *foran* nullpunktet et tilkoblet objekt kan trackes før roboten kobler seg fra (dropper objektet). Minimum distance er altså et negativt tall eller lik null.

- *Maximum distance* - Definerer hvor langt *forbi* nullpunktet et objekt tillates å trackes før det droppes (Positivt tall).

Det er viktig å huske på at trackingen kun utføres i én dimensjon med dette oppsettet. Det er derfor vanlig å plassere roboten og conveyorbeltet slik at conveyorbeltet er parallelt med en av robotens akser. Dette gjør ting mye enklere ved instilling av conveyoraksene, men man er ikke *låst* til å gjøre det slik. I denne oppgaven tas et ferdig kalibrert conveyorsystem i bruk, og det blir derfor ikke nødvendig å gå dypere inn på dette temaet i denne oppgaven.

2.1 Laboppsett

Til prosjektets formål har det blitt tatt i bruk et laboppsett på ABBs lokaler på Bryne. Se Figur 5. ABBs robot IRB52 er tatt i bruk. Dette er en relativt liten 6-akse robot. Et nokså lite conveyorbelte er også satt opp foran roboten. Enkoderen er festet med hjulet direkte på beltet, slik at enkoderen spinner med samme hastighet som conveyoren. Se Figur 6a. Synkbryteren er en metalldetektor som reagerer når kobberklistremerker som er festet på beltet passerer under sensoren. Kobberklistremerkene er altså objektene som skal trackes av roboten. Se Figur 6b.



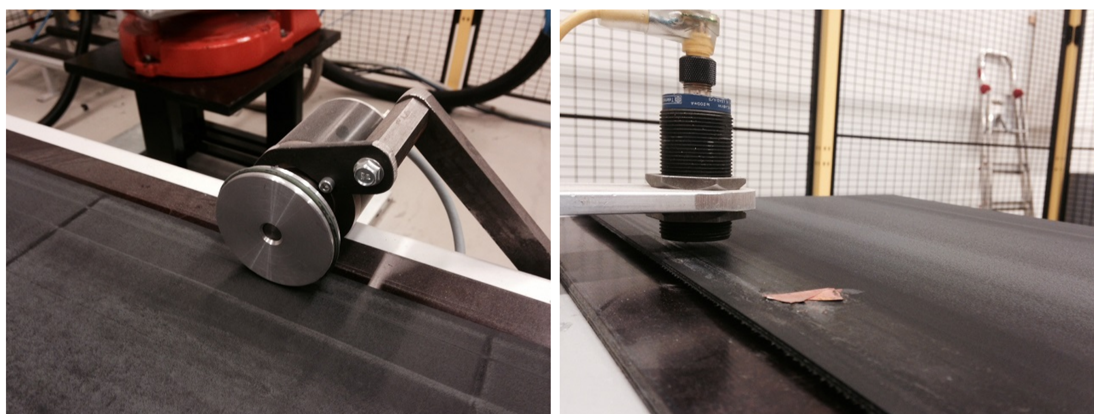
Figur 5: Laboppsettet som er tatt i bruk for prosjektet.

Dette laboppsettet blir utgangspunktet for utvikling og testing av den kamerabaserte trackingen. I neste avsnitt vil det gås gjennom programmeringen av et veldig basisk trackingprogram som kjøres på laben.

2.2 Programmering av tracking

ABBs roboter programmeres i programmeringsspråket Rapid, som er utviklet av ABB til formålet. Et enkelt programeksempel for tracking er vist under. Programmets formål er følgende:

1. Flytt roboten til første punkt.
2. Vent på objekt.



Figur 6: (a):Enkoderen er festet direkte på beltet, slik at den spinner med samme hastighet som conveyoren. (b): Synkryteren er en metalldetektor som gir utslag når kobberklistremerkene passerer under sensoren.

3. Flytt roboten til andre punkt.
4. Behold relativ posisjon til objektet helt til objektet har beveget seg 280 mm fra trackingens startposisjon.
5. Flytt roboten til tredje punkt.
6. Dropp objektet.

Prograreksempel:

```
1  MODULE M18
2
3    PROC mainM18()
4      PaintL p10,v1000,fine,tool0;
5      WaitWobj wobjtrack;
6      PaintL p20,v500,fine,tool0\WObj:=wobjtrack;
7      WaitWobj wobjtrack\RelDist:=280;
8      PaintL p30,v500,fine,tool0\WObj:=wobj0;
9      DropWobj wobjtrack;
10   ENDPROC
11
12 ENDMODULE
```

Rapidprogrammer kalles *moduler*. Navnet på modulen må ha en bestemt form for at Main Computeren i systemet skal godta den, nemlig en *M* etterfulgt av et tall. Som man kan se i eksempelet over, heter dette programmet M18.

I programmet benyttes tre *robtargets*, `p10`, `p20` og `p30`, som er koordinater gitt i forhold til robotens koordinatsystem. Deklarasjonen av disse er ikke tatt med i programeksempelet, men punktene i eksempelet ble funnet ved å *jogge* (styre roboten manuelt ved hjelp

av joystick) til tre punkter og lese av verdiene. Disse forteller i tillegg til romkoordinater (xyz) hvilken orientering roboten skal angripe punktet med.

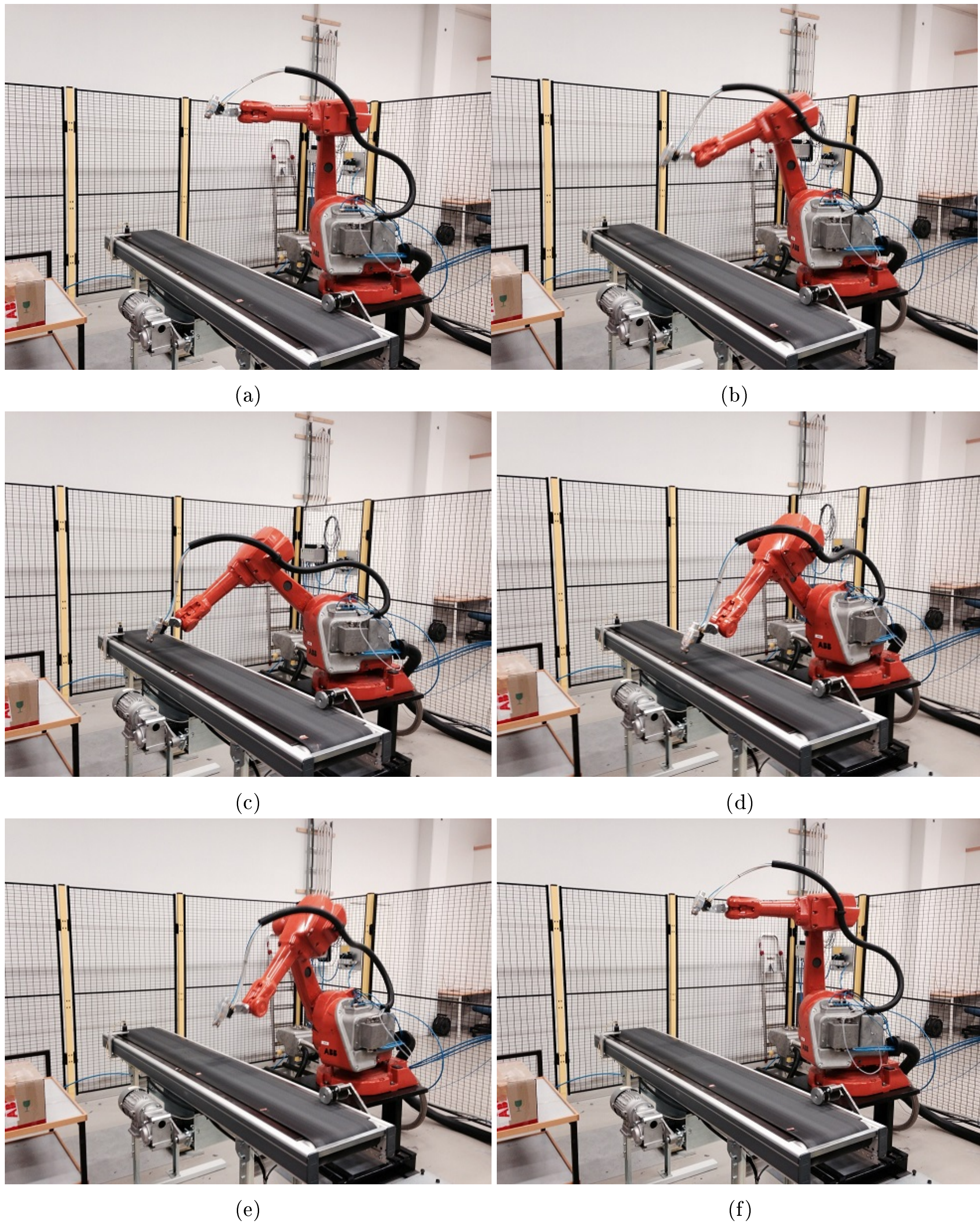
I likhet med de fleste andre programmeringsspråk starter programpekeren på første linje inne i main.

`PaintL` er en funksjon som flytter roboten til et valgt `robtarget`. Man definerer i tillegg hvilken hastighet og presisjon roboten skal ta i bruk, samt hva slags verktøy som sitter på enden av robotarmen. Hastighetene velges fra et utvalg innebygde hastigheter som er definert som en `v` etterfulgt av et tall, der høyere tall betyr høyere hastighet. I eksempelet er hastighetene `v1000` og `v5000` tatt i bruk litt på måfå. Presisjon `fine` er den høyeste mulige presisjonen, og med dette valget vil roboten kjøre helt inntil punktet og stoppe før den fortsetter. Andre valg for presisjon er på lik linje med hastigheter definert med en `z` etterfult av et tall, der høyere tall betyr *mindre* presisjon.

Etter at roboten har flyttet seg til punktet `p10`, kommer programlinjen `WaitWobj wobjtrack;`, som gir beskjed om å vente til et objekt har beveget seg forbi Sync offset og inn i startvinduet. Roboten kobler seg da på objektet, og utfører neste linje med kode, som er en forflytning til neste punkt, p20. Forskjellen her fra den tidligere `PaintL` instruksjonen, er at punktet roboten nå vil korrigere punktet med den relative distansen som objektet flytter seg med underveis. Det betyr at hvis objektet flytter seg 10 cm fra roboten begynner å bevege seg til den når punktet, vil den stanse 10 cm vekk fra `p20` i retningen til objektets bevegelse.

Den neste programlinjen, `WaitWobj wobjtrack\RelDist:=280;`, sørger for at roboten nå holder relativ posisjon til objektet helt til objektet har beveget seg 280 millimeter videre i hvilken som helst retning. Det betyr at dersom conveyoren kjøres fram og tilbake, vil roboten fremdeles holde relativ posisjon til objektet helt til det er 280 millimeter unna punktet der trackingen begynte.

Programmet lastes nå over på robotens Main Computer, og programmet kjøres. Se bildemontagen i Figur 7.



Figur 7: **a**: Roboten får signal fra synkbryteren og venter til objektet når Sync offset, her 300 mm. **b**: Roboten beveger seg ned til punktet `p20`. **c** og **d**: Roboten holder relativ posisjon til objektet i 600 mm. **e**: Roboten flytter seg og dropper objektet. **f**: Roboten venter på nytt objekt.

2.3 Softwaren som tar seg av tracking

I innledningen ble det gått igjennom enkoderens og synkbryterens signalers vei gjennom systemet. Vi skal nå ta en mer detaljert kikk på hvordan trackingen utføres i softwaren PIB-kortets mikroprosessor.

På prosessoren på PIB-kortet kjører et system som heter *Integrated Process System* (IPS), som er utviklet hos ABB på Bryne. IPS er et distribuert system som kjører på mange av kortene i lakkeringsrobotenes kontrollkabinett. Enkelt forklart er IPS et system som tar seg av integrering og behandling av alle de eksterne enhetene som er i bruk i robotsystemet: lakk-applikatorer, lakkpistoler, sikkerhetssignaler, høyspenningsregulering, enkodere og mye mer. Alle tilkoblede signaler kan enkelt konfigureres med filtere og regulatorer. På bunnen av IPS ligger et stort C/C++ bibliotek, der man finner blant annet egne klasser som beskriver tofaseenkoderen og de digitale inngangene som kan konfigureres som syncsignal. Et forenklet overblikk over arkitekturen i IPS når det gjelder conveyor-tracking er vist i Figur 8. [1]

Som skissen viser, hentes signalene inn til klassene `Encoder` og `DigitalInput`. `Encoder` kommuniserer med FPGAen på PIB-kortet via et delt minne, der FPGAen skriver de talte pulsene fra enkoderen med gjevne mellomrom. `Encoder` arver fra en klasse som heter `IpsInputDevice`, som igjen arver fra `IpsDevice`. Med dette følger det et ferdig sett med funksjoner; bl.a. typer som kalles ”signaler” og ”parametere”, samt muligheten til å oppdatere seg selv hvert 16 millisekund. Disse signalene hentes videre til `Conveyor`-klassen. I `Conveyor` regnes hastigheten til conveyorbeltet ut, utifra de talte pulsene og en definert parameter som forteller forholdet mellom pulser per sekund og millimeter per sekund. Den regner i tillegg ut posisjonen til conveyorbeltet, relativt til starttidspunktet. Deretter sendes disse tallene, samt det boolske signalet fra synkbryteren, videre til klassen `ConveyorTracker`. Her beregnes objektetes posisjon på beltet, ved å kombinere kunnskapen om conveyorens posisjon med synksignalet (som forteller hvor objektet er på et gitt tidspunkt). Farten og posisjonen til objektet på beltet sendes så videre til to `Sensor`-klasser, som kommuniserer videre med et øvre lag for kommunikasjon med robotkontrolleren. På robotkontrolleren brukes disse to signalene til å bestemme hvor roboten skal bevege seg[2].

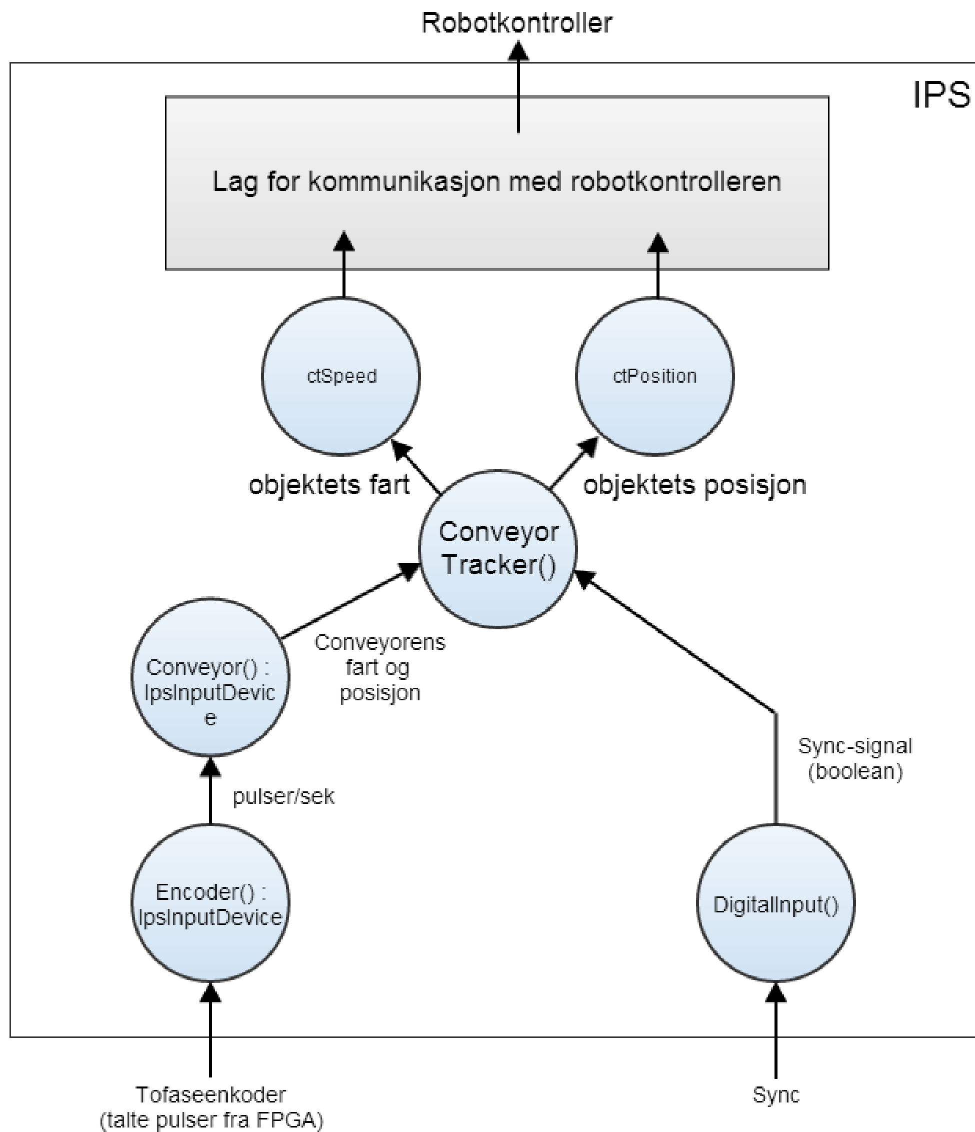
2.3.1 IpsDevice klassen

Mange av klassene som brukes for conveyortrackingen arver fra en klasse som heter `IpsDevice`, eller en underklasse av denne.

`IpsDevice` er en klasse som er skrevet for å gi dynamiske signalbaner i software. Klassen har definerte funksjoner for *signaler* og *parametere*. Et `IpsDevice` kan nokså enkelt

[1] I noen tilfeller har undertegnede tatt seg friheten til å endre navnene på klassene fra det de heter i IPS, ettersom de i noen tilfeller kan være litt kryptiske.

[2] Farten som `CT` sender til robotkontrolleren brukes til å ekstrapolere posisjonen



Figur 8: Prinsippskisse for conveyortracking med enkoder i IPS.

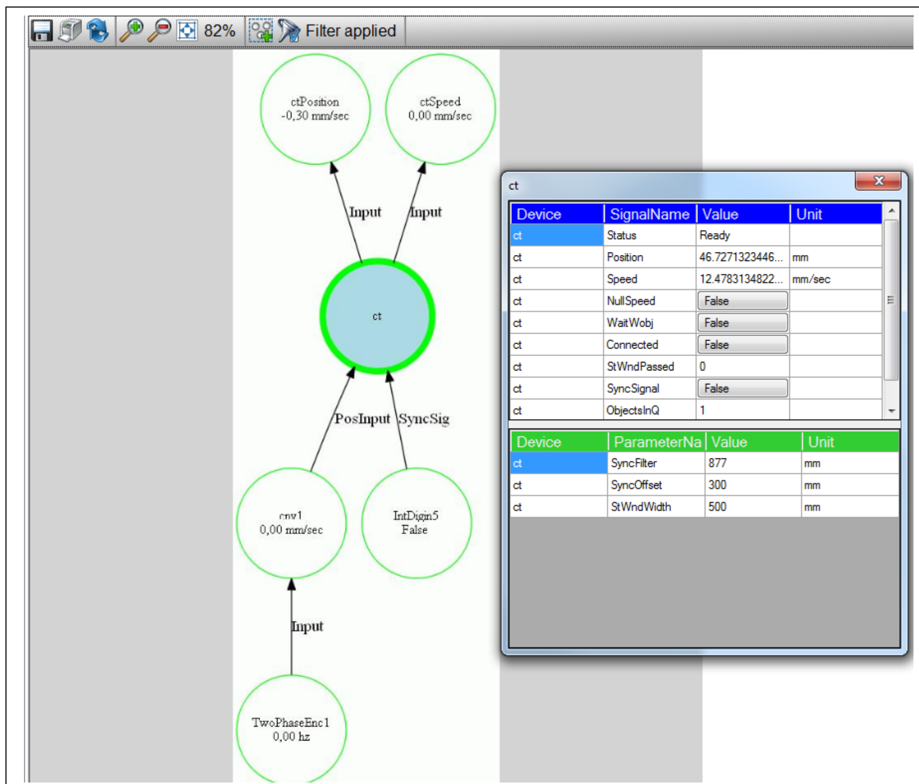
"kobles" til et annet `IpsDevice`, slik at signalene fra det ene kan brukes videre av et annet. Dette kan blandt annet gjøres i en egen konfigurasjonsfil som legges til i IPS. IPS holder automatisk styr på alle instanser av `IpsDevice`. Ved å implementere lese-metoden **`read(int signalnumber)`**, kan man gjøre signalene tilgjengelig til å leses på serieporten, eller i programmet *RobView*, der man har mulighet til å logge signalene.

Ved å la klasser arve fra `IpsDevice`, finnes det altså ferdige måter å logge variablene i klassen, ved å definere signaler og parametere som tar i bruk disse variablene.

2.4 Sanntidsanalyse og logging i RobView

Signalene og parameterene i IPS kan observeres i sanntid ved å bruke programmet RobView, som også er utviklet hos ABB på Bryne. Dette programmet gir brukeren/utvikleren muligheten til å se på de forskjellige klassene og variablene som er i bruk i IPS. Dette gjøres i RobView ved å koble seg til IPS-systemet over LAN med IP addressen til PIB-kortet.

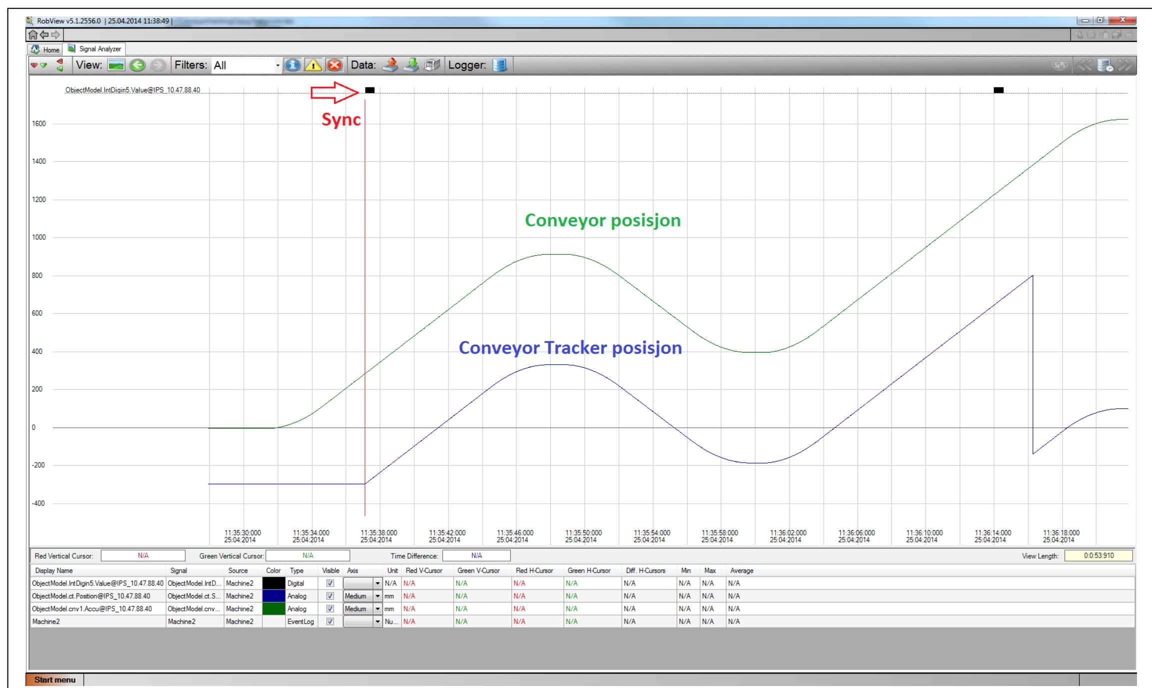
Figur 9 viser et utklipp av vinduet i RobView, der funksjonen *IPS Explorer* er i bruk. Denne funksjonen gir et blokkdiagram som viser de virtuelle signalgangene i IPS. Her har det valgt å bare vise de delene av IPS som er relevante for conveyortracking. Navnene som vises i diagrammet er navnene som tildeles objektene i konstruktør-funksjonen til klassene, der de lagres som en tekststreng. Blokken i diagrammet som har navnet "ct", representerer `ConveyorTracker`-objektet. Tilsvarende representerer "cnv1", "TwoPhaseEnc1" og "IntDigin5" henholdsvis `Conveyor`- `Encoder`- og `Synk`- objektene. Ved å trykke på objektene, får man opp et vindu der signalene og parameterene til klassen vises i sanntid. Se igjen Figur 9, der signalene og parameterene til `ConveyorTracker` vises i det lille vinduet.



Figur 9: Sanntidsobservering av signaler og parametere i RobView. Her er det `ConveyorTracker` signalene og parameterene som er vist.

En annen svært nyttig funksjonalitet i RobView er muligheten til å logge de virtuelle

signalene i IPS. Her kan man velge hvilke signaler man ønsker å logge, og resultatet kan ses i Figur 10. Her har posisjonen til conveyoren, posisjonen til conveyortrackeren og synksignalet blitt logget mens conveyoren har blitt kjørt fram og tilbake. Man kan se at signalene oppfører seg i tråd med det som ble gått igjennom i avsnitt 2: Conveyortrackerens posisjon er i ro helt til synk-signalet slår inn, hvor den da starter på −300 når Sync Offset er definert til 300. Den vil da begynne å oppdatere seg med den relative endringen i conveyorens posisjon, helt til den går utenfor arbeidsområdet. Objektet vil da droppes, som kan ses i det brå fallet i den blå kurven, og den vil da begynne å tracke neste objekt i køen, eller vente på et nytt dersom ingen objekt er i køen.



Figur 10: Logging av signaler i RobViews Signal Analyzer.

Dataloggen kan så eksporteres som en kommaseparert tekstfil, noe som i denne oppgaven har blitt brukt til å få data inn i Matlab, for å utvikle og teste signalbehandlingsalgoritmer.

3 Conveyortracking med kamera

I dette kapittelet ses det først litt på hva som tidligere er gjort når det gjelder conveyortracking med kamera. Deretter ses det på utfordringer med kamera anvendt til conveyortracking. Til slutt kommer en kort gjennomgang av kameratracking.

3.1 Tidligere arbeid

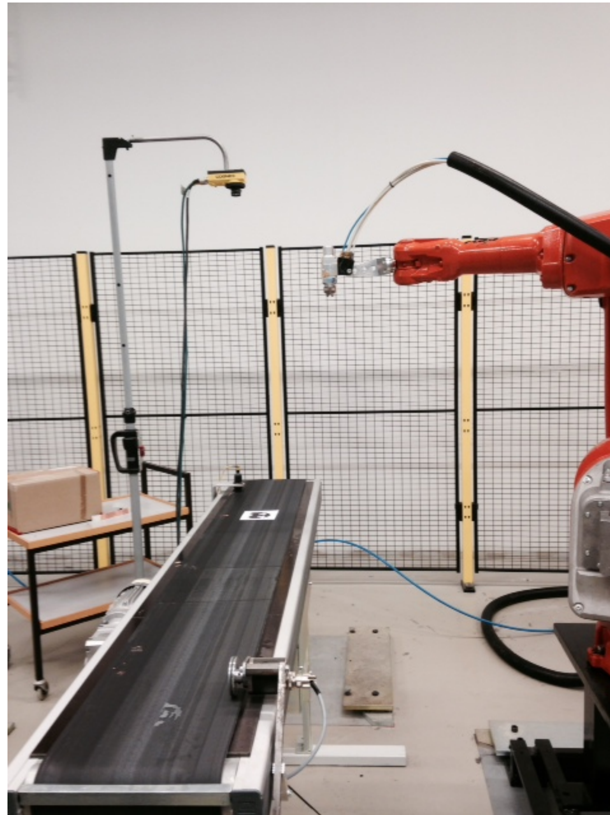
Kamera som supplement til conveyortracking er ikke noe nytt i seg selv. [10] beskriver et conveyorsystem som tar i bruk kamera som supplement til conveyortracking. Her brukes kameraet til å beskrive posisjonene til objekter på conveyoren én gang, og deretter blir objektenes posisjon tracket av en enkoder. Kameraet plasseres *før* robotens arbeidsområde, i en stasjonær posisjon rett over conveyoren. På denne måten kan roboten få vite objektenes forflytning på tvers av conveyoren, samt deres orientering. Denne formen for tracking på conveyorbelte tas blandt annet i bruk av ABB's Flexpicker roboter (`https://www.youtube.com/watch?v=wg8YYuLLoMO`).

[19] presenterer et system med live tracking av objekter med et kamera som er festet direkte på robotarmen. Denne måten å gjøre conveyortracking på virker som å overkomplisere ting dersom man er ute etter hurtig og robust conveyortracking. Det vil kreve langt mer komplisert databehandling å måtte ta stilling til et kamera i bevegelse, og objektgjenkjenning vil være langt mere komplisert ettersom objektet må gjenkjennes fra mange forskjellige vinkler og skalaer. Dette vil koste på utførelseshastigheten, som bør holdes så lav som mulig i et sanntids trackingsystem.

Utifra oppgaven som er gitt av ABB, kreves det en annen løsning i dette prosjektet. Ettersom det kun er kameraet som skal ta seg av trackingen, er det naturlig å feste kameraet slik at det ser objektet i arbeidsområdet til roboten, i motsetning til [10], der kameraet er festet foran arbeidsområdet. Kameraet vil bli plassert stasjonært over conveyoren, pekende normalt ned på conveyorplanet. På denne måten vil objektene som skal gjenkjennes til en viss grad begrenses til én vinkel, nemlig ovenifra. I tillegg begrenses trackingen til ett plan, conveyorplanet[3].

Figur 11 og 12 viser det valgte oppsettet av kameraet over conveyoren.

[3]Dersom man ser bort i fra høyden til objektet, noe vi kommer tilbake til i avsnitt 3.3.4



Figur 11: Kameraoppsettet som er tatt i bruk i prosjektet.

Figur 12: Cognex smartkamera.

3.2 Cognex smartkamera

Kameraet som er tatt i bruk i oppgaven, er et Cognex In-Sight 5400 kamera. Dette er et IP67 beskyttelse, noe som er nødvendig dersom kameraet skal tas i bruk i lakkeringskabiner, som må være eksplosjonssikre. Kameraet har innebygget datamaskin som tilbyr nokså kraftig databehandling.

Litt informasjon om kameraet:

Oppløsning	640x480
Farger	Gråskala
Bildetagning	60 fps (maks)
Beskyttelse	IP67
Patmax	Ja

Patmax er en patentert objektgjenkjenningsteknologi fra Cognex, og er et av de viktigste varemerkene deres. Det gås gjennom en teori om hvordan denne fungerer i avsnitt 3.9. In-Sight kameraene til Cognex programmeres i programmet In-Sight Explorer, som er et GUI-basert programmeringsmiljø. En gjennomgang av dette kommer i seksjon 4.

Figur 13: Cognex In-Sight 5400 smartkamera.

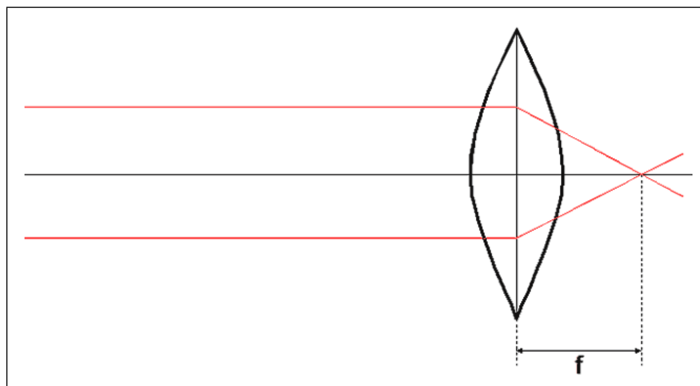
3.3 Utfordringer og begrensninger

3.3.1 Oppløsning

Bildet fra kameraet er begrenset av en gitt oppløsning. For Cognex kameraet som er tatt i bruk, er denne på 640x480 piksler. Det at bildet består av piksler, fører til usikkerhet om tings virkelige posisjon. Jo lengre unna kameraet plasseres fra objektene den skal gjenkjenne, jo større areal dekker hver piksel. Kameraet må plasseres slik at man det gir en god balanse mellom synsfelt og objektdetalj.

3.3.2 Synsfelt

Kameratracking er begrenset av kameraets synsfelt (Eng.: Field Of View, FOV). FOV (i vinkler) avhenger av kameraets brennvidde, som kan endres ved å bruke forskjellige typer linser. Brennvidde er linsens evne til å samle lys, gitt av avstanden fra midten av linsen, til *brennpunktet*, som er det teoretiske punktet der lysstrålene krysser hverandre bak linsen, som vist i Figur 14[1].



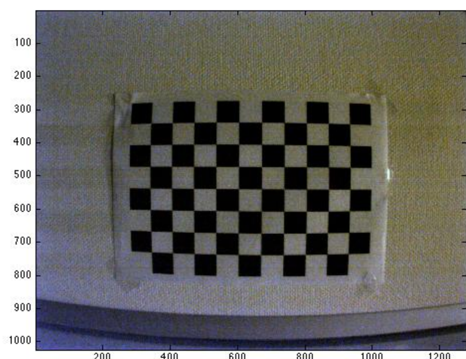
Figur 14: Brennvidden f er gitt av avstanden fra midten av linsen til brennpunktet [1].

Mindre brennvidde fører til bredere synsfelt, og omvendt. For conveyortracking er det ønskelig med et synsfelt som i hvertfall dekker arbeidsområdet som roboten skal jobbe i. I tillegg til å gi bredere synfelt, fører linser med kort brennvidde også til økt *bildeforvrengning*.

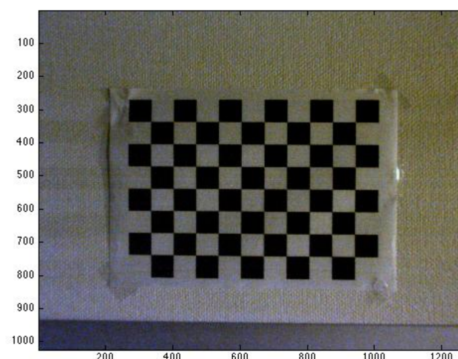
3.3.3 Bildeforvrengning

Bildeforvrengning i optikk, er at rette linjer i scenen blir projisert ulineært på bildebrikken slik at det får bøyning i bildet. Den vanligste formen for bildeforvrengning, er (tilnærmet) radielt symmetrisk forvrengning. Cognex kameraet er til en viss grad preget av *barrel* forvrengning, som er vist i Figur 15. For å utføre nøyaktig tracking, må slike avvik korrigeres for. Dette kan gjøres ved å lage en matematisk modell av forvrengningen. Til dette er det vanlig å ta i bruk sjakkmønster, for å få en mengde punkter i bildet som man vet den virkelige relative avstanden mellom ved å detektere hjørnene mellom rutene.

Disse kan brukes for å lage en modell av forvrengningen, og korrigere for det. Et korrigert bilde er vist i Figur 16. Selv om det er mulig å korrigere for forvrengning, er likevel best å ha minst mulig forvrengning i utgangspunktet, ettersom det alltid vil være modellfeil.



Figur 15: Barrel forvrengning.



Figur 16: Korrigert bilde.

3.3.4 Dybde

Et bilde tatt med et tradisjonelt kamera er alltid begrenset til to dimensjoner. Det betyr at det er umulig å bestemme avstanden til et objekt i bildet, med mindre man har informasjon om objektet på forhånd (eller om man triangulerer ved hjelp av stereosyn eller projiserte mønster). Til dette prosjektet har det blitt bestemt å anta én kjent dybde på z-aksen, noe som vil være naturlig med én type objekt på conveyoren, da alle vil ha den samme z-koordinaten. Dersom det er flere forskjellige typer objekter på conveyoren, må dette endres, men det ses bort i fra i dette prosjektet.

3.3.5 Utførelsestid og forsinkelse

Kameraet bruker tid på ta bilde, detektere objekt, og å sende data videre til robotsystemet. Dette fører til forsinkelse i signalet, som må tas hensyn til med prediksjon.

3.3.6 Valg av linse

Kameraet var bestilt på forhånd da denne oppgaven skulle skrives, med en medfølgende linse med 8 millimeter brennvidde. Denne gir 800x600 millimeter FOV på conveyoren når kameraet står 1200 millimeter over conveyoren. Dette gir at hver piksel har bredde og høyde på 1.25 millimeter i conveyorplanet. Det ble vurdert å bestille flere linser til å eksperimentere med, men da det var to måneders bestillingstid fra Cognex, ble dette

latt være, ettersom andre ting måtte prioriteres. I videre arbeid må valg av linse bli tatt bedre hensyn til.

3.4 Kameratracking

Kameratracking er oppgaven å følge ett eller flere objekter i en scene, fra de først blir synlige til de forlater scenen [11].

Bilder og video gir høyt detaljerte oppsummeringer av komplekse, dynamiske miljøer [22]. Ved å bruke maskinsyn kan man automatisk detektere og gjenkjenne objekter, tracke bevegelsene dere og beregne 3D scenegeometri. Bruk av kameratracking av objekter spenner over et bredt felt av applikasjoner, alt fra mann-maskin grensesnitt, robotnavigasjon, trafikkovervåkning, sportsdømming, dataspill og militære formål.

Maskinsyn har i den siste tiden blitt mer og mer vanlig å ta i bruk i industrien. Dette kommer av at maskinsyn som felt har utviklet seg raskt, samtidig som prisen på datakraft og kameraer stadig synker. Dette gjør at kamerasystemer nå i mange tilfeller kan konkurrere med enklere sensorer som tidligere har vært i bruk.

Kameratracking er et stort tema, og denne seksjonen prøver ikke å gå i dybden på det. Det er i hovedsak en kort innføring i objektgjenkjenning, for å kunne forklare litt om hvordan Cognex kameraets algoritme for objektdeteksjon, Patmax, muligens fungerer.

3.5 Objektmodellering

Objektmodellering handler om å definere objektet med et sett med egenskaper (eng.: features), og å lage en representasjon av objektet i bildet (punktrepresentasjon, geometrisk form, silhuett, etc.). Effektiv og robust tracking avhenger av en god modell av objektet.

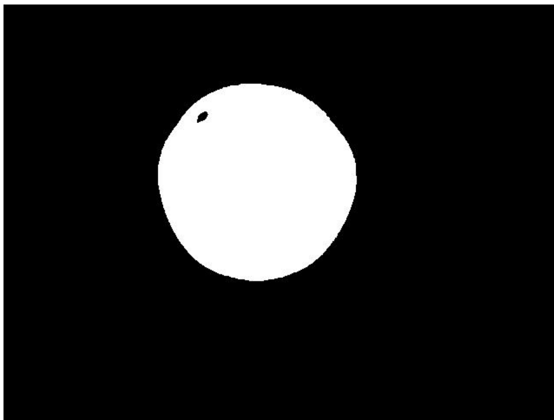
Egenskapene til objektet som velges kan være svært mye forskjellig. Farge eller lysintensitet er kanskje de enkleste, da man enkelt f.eks. kan finne en rød tomat foran en hvit bakgrunn ved å lete etter rødfarge i bildet. Figur 17 viser et bilde av en rød tomat mot en hvit bakgrunn. Ved å subtrahere grønnkomponenten i bildet fra rødkomponenten, får man gråskalabildet vist i Figur 18. Dette kan enkelt thresholdes for å få bildet vist i Figur 19. Ved å ta gjennomsnittlig pikselposisjon til de hvite pikslene, kan man estimere sentrum av tomaten, representert med en grønn prikk i Figur 20. Man har da oppnådd å gjenkjenne tomaten, og representere den som et punkt i bildet.

Dette er kanskje det enkleste eksempelet på objektgjenkjenning, og det blir fort mye mer komplisert når bakgrunnen og objektet ikke har distinktive farger. Da blir det nødvendig å ta i bruk andre former for objektmodeller, f.eks. ved å se på mønstere på objektet.

En vanlig metode for å modellere objekter, er å definere et sett med interessepunkter (eng.: points of interest) på objektet, og å lete etter steder i bildet der liknende mønster med punkter opptrer. Interessepunkter kan defineres på flere måter, for eksempel ved å se på kanter og hjørner på objektet [11]. Fordelen med å definere et sett med interessepunkter fremfor å lete etter selve objektet (det originale bildet av objektet) i et bildet,

Figur 17: Tomat foran hvit bakgrunn

Figur 18: Rødkomponenten minus Grønn-komponenten.



Figur 19: Thresholdet, binært bilde.

Figur 20: Det beregnede sentrum av tomaten er representert med en grønn firkant.

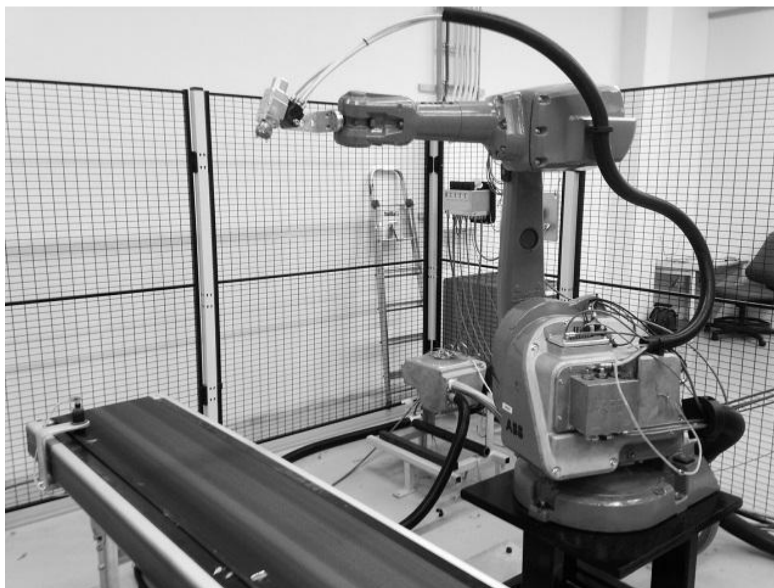
er at datamengden som skal behandles blir svært redusert.

3.6 Kantdeteksjon (Canny metoden)

Kanter i bilder er svært interessant, ettersom man med kanter kan få en god idé om hva bildet inneholder, selv om datamengden er svært redusert i forhold til originalbildet. Dette gjør kanter nyttige når det gjelder interessepunkter for tracking. Figur 27 viser bilde av laboppsettet. Vi skal nå se på hvordan kantene kan detekteres i bildet.
En svært mye brukt algoritme for dette er *Canny* kantdeteksjons algoritmen (eng.: Canny edge detection), oppkalt etter John F. Canny som utviklet den i 1986 [5].

Canny kantdeteksjonsalgoritmen består av 5 steg.

- Glatting (lavpassfiltrering)



Figur 21: Bilde som skal gjennomføres Canny kantdeteksjon på.

- Finne gradienter
- Ikke-maksimum suppresjon (non-maximum suppression)
- Høy og lav thresholding
- Kant-følging ved hysterese (edge tracking by hysteresis)

3.6.1 Glatting

Det første steget i Cannys kantdeteksjons algoritme går ut på å glatte bildet. Dette kommer av at bilder inneholder støy, som kan gi utslag på gradienten i bildet. Derfor glatter man, slik at at støyen i stor grad forsvinner, mens kantene forblir i bildet på bekostning av at de har blitt mindre skarpe[4]. Det vanligste er å ta i bruk et gaussisk filter for glatting. Dersom salt-og-pepper støy er tilstede er det vanlig å ta i bruk et medianfilter for dette [4]. Figur 22 viser bildet etter gaussisk filtrering.

Har at

$$I_g = F_g * I$$

der F_g er det gaussiske filteret, I er det originale bildet, I_g er det filtrerte bildet og * er konvolusjonsoperatoren.

[4]Glatting fører til at piksler får samme verdi som pikslene i nærheten. Derfor forsvinner støy, mens kanter, som er samlinger med piksler med liknende verdier, blir mindre skarpe, men beholdes i bildet.

Figur 22: Bildet etter gaussisk filtrering.

3.6.2 Finne gradienter

Neste steget i Cannys er å finne gradientene i bildet. Til dette tar man i bruk *differensielle filtere*, som finner den deriverte av bildet. Et eksempel på differensielle filtere er *Sobel-operatorene*, som er gitt av matrisene

$$K_{GX} = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

og

$$K_{GY} = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

Gradienten til bildet i x- og y-retning finnes ved å konvolvere bildet med henholdsvis K_{GX} og K_{GY}.

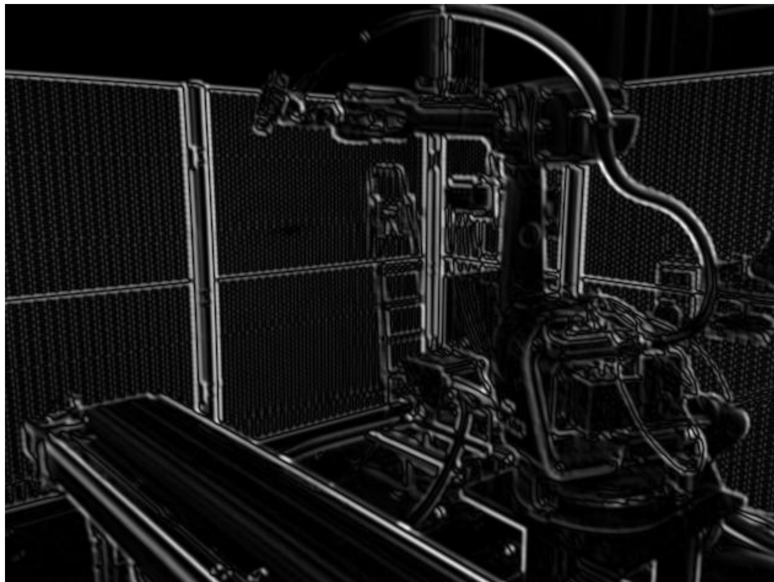
$$G = Gx\vec{i} + Gy\vec{j} = (I_g * K_{GX})\vec{i} + (I_g * K_{GY})\vec{j}$$

der G er gradienten til det glattede bildet, I_g er det glattede bildet og $\vec{i}$ og $\vec{j}$ er enhetsvektor i henholdsvis x- og y-retning. Det er vanskelig å vise fram gradienten G i et bilde, men absoluttverdien $|G|$ kan ses i Figur 23.

En viktig ting å nevne er at man som regel utfører glatting og differensiering i én kovolusjon [11]. Ettersom

$$Gx = K_{GX} * (F_G * I) = (K_{GX} * F_G) * I = H_X * I$$

der H_X er konvolusjonen mellom glattefilteret og differensialfilteret. På denne måten kan man bruke en enkelt konvolusjon på originalbildet istedet for to, noe som gjør utregningen langt mer kosteffektiv.



Figur 23: Absoluttverdien til gradienten til det glattede bildet.

3.6.3 Ikke-maksimum suppresjon

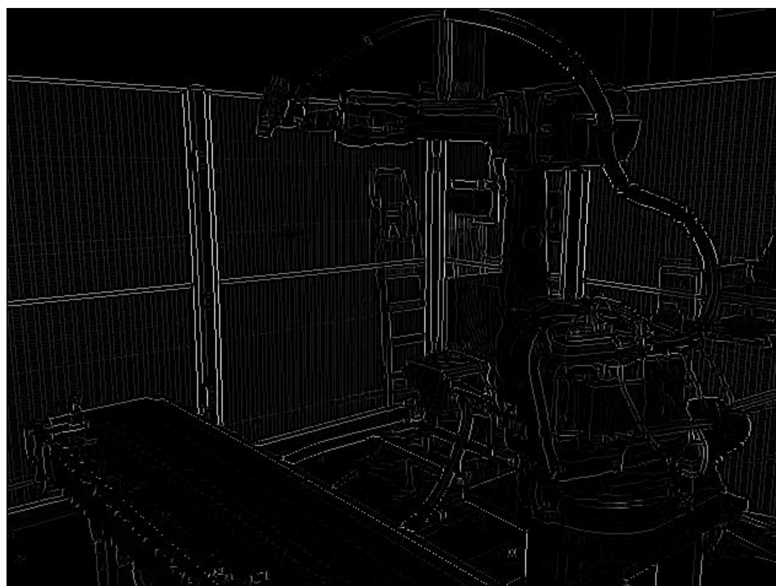
Matrisen som man nå sitter med, $G = Gx\vec{i} + Gy\vec{j}$, er gradienten til bildet. Det neste steget i Canny's metode er å finne de lokale maksima i bildet $J = |G|$, langs gradienten. Ettersom gradienten vil peke *over* kanten, vil de lokale maksima kunne finnes ved å sammenlikne piksler med neste piksel i gradientretning, og pikselen i motsatt retning. For å forenkle beregningene, ser man kun på de nærmeste 8 pikslene. Gradientretningens vinkel i xy-planet, θ, finnes ved

$$\theta = atan\frac{|G_y|}{|G_x|}$$

Algoritmen er:
For hver piksel i bildet:

1. Rund av gradientens retningen til nærmeste 45 grader (0,45,90, etc...)
2. Sammenlikne pikselen med pikslene foran og bak i gradientretning.
3. Dersom denne pikselens verdi er høyere enn begge, behold pikselverdien i det nye bildet. Hvis ikke, sett den korresponderende pikselen til null.

Med denne algoritmen utført får man bildet i Figur 24. Her er alle kantene kun én piksel bred.



Figur 24: Resultat etter non-max suppression algoritmen.

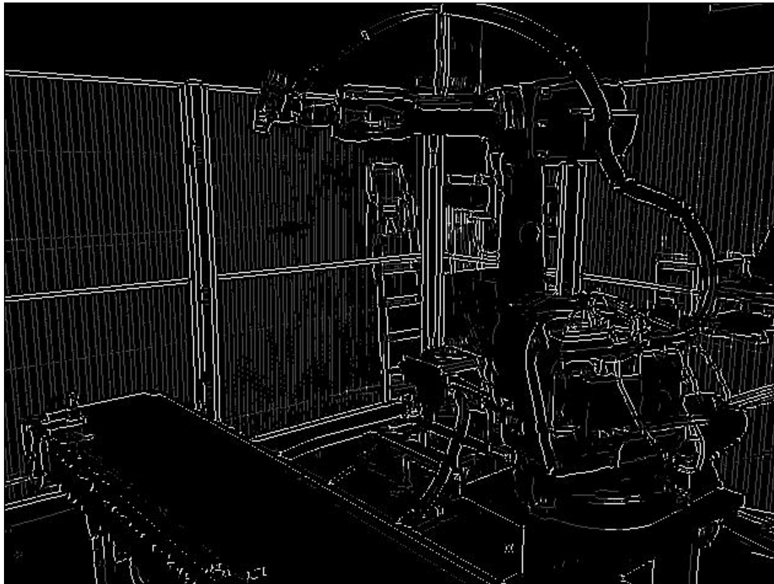
3.6.4 Thresholding

Det neste steget i Canny algoritmen går ut på å skille *sterke* kanter fra *svake kanter*. Tanken er at de svake kantene kun har lov til å bli med i det ferdige kant-bildet dersom de er i kontakt med en sterk kant. Til dette brukes to thresholds, ett høyere enn det andre. Gradienter som har høyere verdi enn det høye thresholdet, tolkes som sterke, mens gradienter med verdi mellom det lave og høye thresholdet, tolkes som svake. Pikslene som har verdi lavere enn det lave thresholdet kastes fullstendig og settes til null. Etter dobbel thresholding, ender man opp med noe som Figur 25, men man får forskjellige resultater basert på hvilke threhold verdier man tar i bruk. De svake og sterke kantene er representert med svakere og sterkere intesnitet.

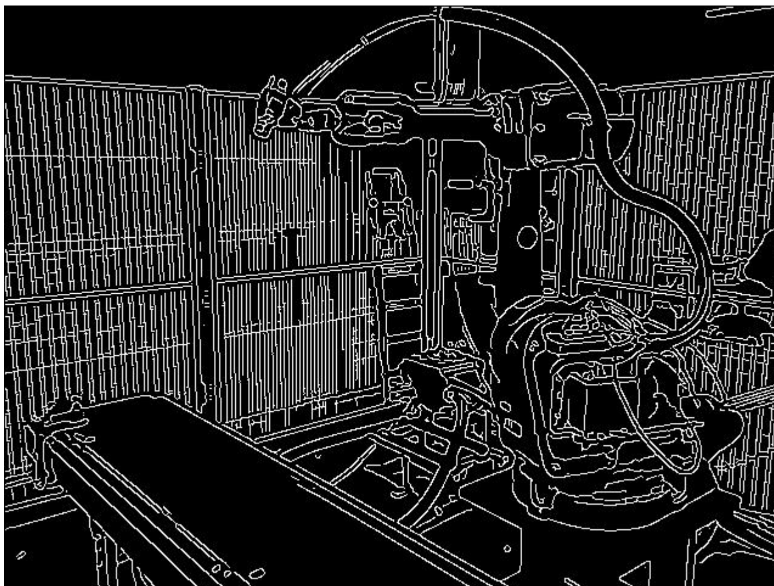
3.6.5 Kantfølging ved hysterese

De sterke kantene (dvs. de som hadde verdi høyere enn det høye thresholdet) havner rett inn i det endelige bildet, ettersom disse tolkes som "sikre". Det gjenstår da å bestemme hvilke av de svake kantene som skal være med i det endelige bildet. Dette gjøres ved å følge de svake kantene, og se om de treffer en sterk kant. Dersom de gjør det, tolkes de som sterke kanter. Hvis ikke, kastes de.

Et eksempel på et endelig bilde vises i Figur 26.



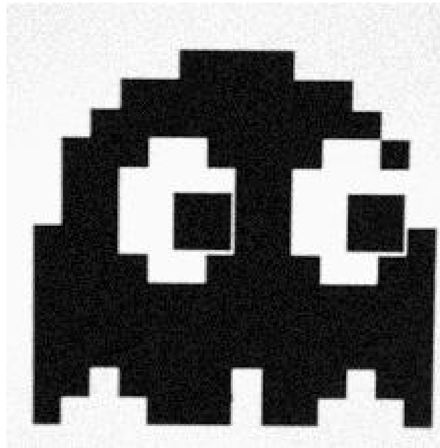
Figur 25: Eksempel på resultat etter høy og lav thresholding.



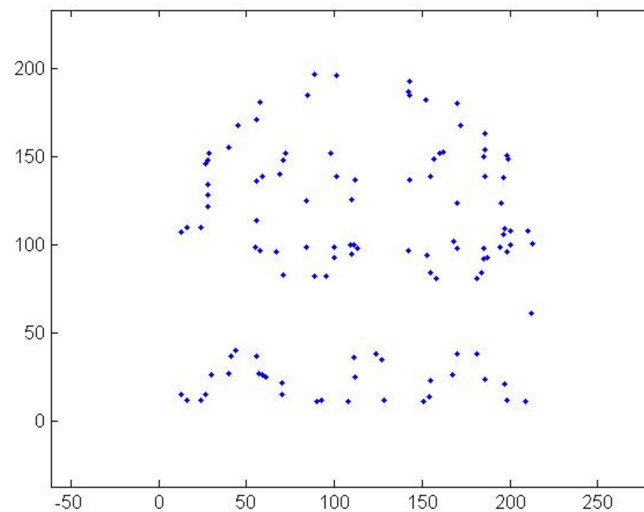
Figur 26: Eksempel på endelig resultat etter Canny kantdeteksjon.

3.7 Deteksjon av hjørner

En annen metode å finne interessepunkter på er å detektere hjørnene på et objekt. En av de mest brukte metodene for dette er Harris hjørnedeteksjon [12]. Denne bygger detekteringen av hjørner på at i et område i bildet der det befinner seg et hjørne, vil det være to eller flere sterke gradientretninger, på grunn av kanter som møtes. Det vil ikke bli gått gjennom hjørndedeteksjon i detalj, men resultatet på hjørnedeteksjon vises i Figur 27 og 28. Man har med Harris' metode kommet frem til en Nx2 matrise som inneholder punkkoordinatene til hjørnene som er funnet. Punktene er plottet i Figur 28.



Figur 27: Bilde som skal detekteres hjørner i.



Figur 28: De detekterte hjørnene.

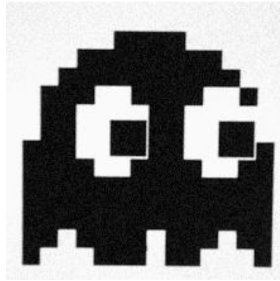
3.8 Gjenkjenning av objekt fra modellens egenskaper

Når man har et sett med interessepunkter, som f.eks. de som finnes med kant- og hjørnedeteksjon, er er det vanlig å hente ut features (egenskaper) i bildet rundt disse punktene. Dette kan være ting som farge, lysintensitet, tekstur, gradient, eller andre bestemte interressant egenskaper. Objektgjenkjenning går ut på å lage en tilsvarende modell av søkebildet, og å prøve å finne *treff* (eng.: matches) mellom den trente modellen og søkemodellen, ved å sammenlikne de utvalgte interessepunktene og de tilhørende features'ene. Ved å rotere, skalere og translere på den trente modellen, og se om punkter sammenfaller med punkter i søkemodellen, kan man finne et beste treff. Dersom det beste treffet gir en verdi over et definert threshold, velges det å si at objektet er detektert i søkebildet.

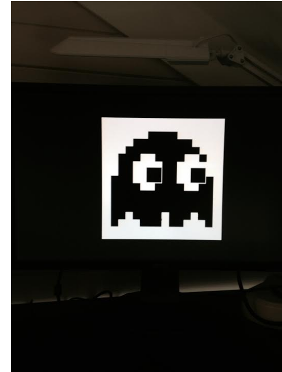
Et Matlab eksempel på gjenkjenning basert på hjørnepunkter og features som hentes rundt hjørnepunktene er vist under. Målet er å finne hvor bildet (eller objektet om man vil) i Figur 29 passer inn i søkebildet i Figur 30.

```
1  %Finn hjoernene i begge bildene model og search
2  points1 = detectHarrisFeatures(model);
3  points2 = detectHarrisFeatures(search);
4
5  %Hent ut features rundt disse punktene:
6  [features1, valid_points1] = extractFeatures(model, points1);
7  [features2, valid_points2] = extractFeatures(search, points2);
8
9  %Finn elementer i de to listene med features (model og search),
10 %som matcher
11 indexPairs = matchFeatures(features1, features2);
12
13 %hent ut de matchende punktene
14 matched_points1 = valid_points1(indexPairs(:, 1), :);
15 matched_points2 = valid_points2(indexPairs(:, 2), :);
16
17 %vis resultat
18 figure; showMatchedFeatures...
19     (model, search, matched_points1, matched_points2);
```

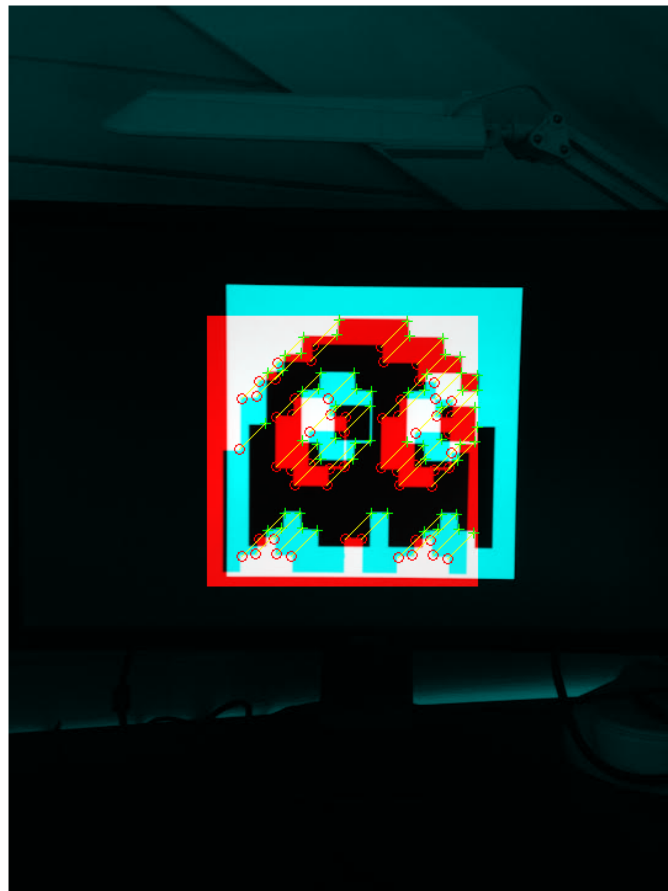
Resultatet på søket kan ses i Figur 31.



Figur 29: Objekt som skal gjenkjennes i søkebildet.



Figur 30: Søkebildet.



Figur 31: Matchet bilde. Punktmatchene representeres med de røde og grønne sirklene.

3.9 PatMax algoritmen

Cognex hevder selv at deres algoritme for objektgjenkjenning som går under navnet Patmax® er "gullstandarden" innen maskinsyn [6]. Som den første patenterte høy-nøyaktighet, høy-hurtiget, høy-ytelses objekt lokaliseringsteknologien innen maskinsyn, hevder de at Patmax er en av grunnene til at maskinsyn har blitt langt mere attraktivt i industrien i nyere år.

Noen fakta om Patmax som Cognex selv annonserer: [9]

- Patmax baserer seg på geometriske modeller, og ikke på pikselrutenett representasjoner som ikke kan effektivt og nøyaktig bli rotert eller skalert.
- Tar i bruk tabeller med features som kan transformeres hurtig og nøyaktig for mønster-matching.
- Mindre påvirket av endring i farge/lyssetting enn pikselbaserte modeller.
- Den geometriske modellen lages ved å plukke ut punkter langs kantene til objektet. Hurtigheten til algoritmen avhenger blandt annet av minste radiusen mellom hvert punkt som plukkes ut[5].

[15] presenterer en teori om Patmax, med de følgende stegene:

1. Lavpassfiltrering for å redusere bildestøy.
2. Høypassfiltrering for å fremheve kanter
3. Thresholding for å få et binært bilde med kanter og ikke-kanter.
4. Morphologiske operasjoner for å fjerne uviktige kantsegmenter, og for å reparere skadede segmenter.
5. Segmentering av kantene.
6. Kalkuler en beskrivelse av segmentsettet, kalt et mønster.

Til og med Steg 3 er dette forslaget svært likt Canny metoden for kantdeteksjon[5], som ble gjennomgått i avsnitt 3.6 (minus høypassfiltreringen for fremheving av kantene). Ifølge [15] er Steg 4 strengt tatt ikke nødvendig når en mer sofistikert kantdeteksjonsalgoritme som Canny har blitt tatt i bruk, ettersom den allerede har laget et binært bilde med én piksel brede kanter.

Ved segmentering, menes å plukke ut kantpunktene som ligger intill hverandre, og legge dem i lister. Hvert element i listen inneholder pikselkoordinatene og gradienten i det

[5]Dette kalles *granularitet*. En granularitetsverdi på 6 betyr at kantpunkter vil ha en radius på 6 piksler hvor ingen andre kantpunkter kan eksistere [9]

korresponderende punktet. Man har dermed redusert kantbildet til et sett med lister som inneholder pikselkoordinatene til hvert kantpunkt, samt gradienten i punktet.

[15] foreslår deretter at modellen beskrives av korte linjesegmenter mellom punktene, ved å ta i bruk *stick-growing* [18].

Ved å utføre den samme behandlingen av bildet som skal søkes i, går teorien ut på å prøve å få linjesegmentene til å matche. Ettersom det er alt for krevende å søke gjennom et høy-resolusjonsbilde, og prøve å matche en detaljert linjerepresentasjon[15], foreslås det derfor å lage minst to modeller, en fra høyresolusjonsbildet og en fra et nedsamplet bilde. Tanken er å først prøve å matche lav-resolusjonsmodellen med en tilsvarende modell av lav-resolusjons søkebilde. Hvis en match finnes, søkes det videre i dette området i høy-resolusjonmodellen. Slik skalering av søket er vanlig også for andre søkealgoritmer. Beste match i søkebildet, som vil si transformasjonen av modellen som gir minste feil, finnes ved å finne minste verdi av uoverensstemmelsen E mellom den trente modellen og søkemodellen. I spesialtilfellet der det ikke er noen forskjell i skalering, svarer dette til å minimere gradienten

$$\nabla E(X, Y, \theta) = 0$$

der X,Y og θ er parameterene i transformasjonsmatrisen

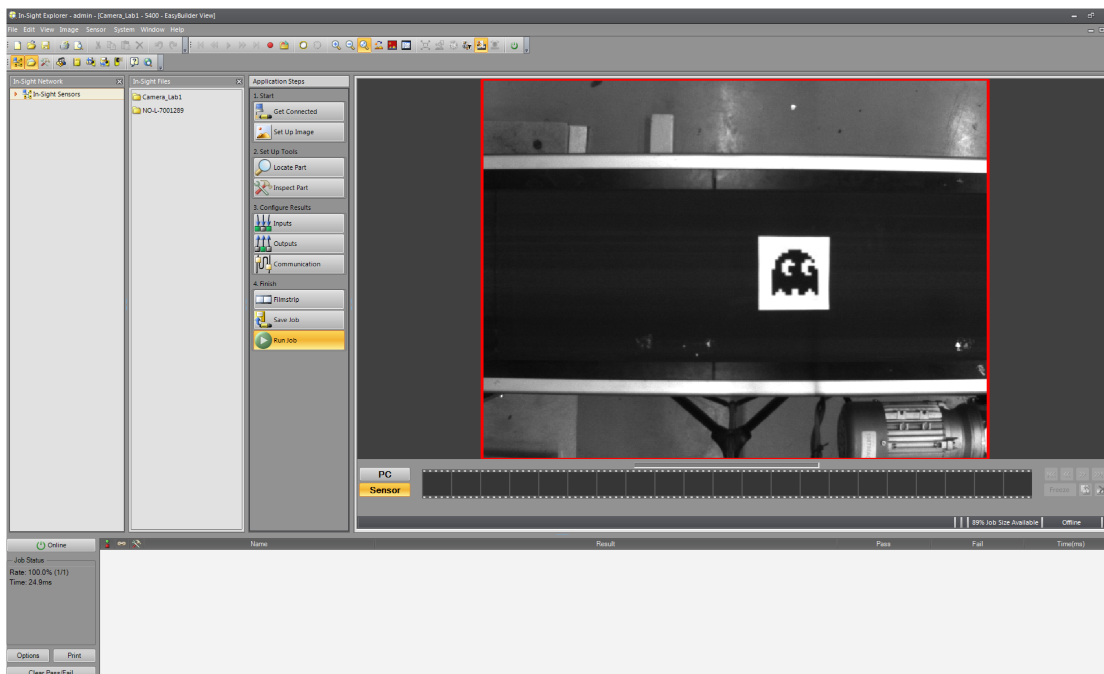
$$T = \begin{bmatrix} cos(\theta) & -sin(\theta) & X \\ sin(\theta) & cos(\theta) & Y \\ 0 & 0 & 1 \end{bmatrix}$$

[15]
I den neste seksjonen gjennomgås programmering av Cognex kameraet, der Patmax algoritmen tas i bruk i praksis.

4 Programmering av smartkameraet

Cognex leverer et GUI basert programmeringsmiljø til sin In-Sight serie, kalt In-Sight Explorer (ISE). I dette avsnittet gås det gjennom programmering av smartkameraet.

Når ISE starter opp, får man valget å koble til In-Sight kameraene som detekteres på LAN. Ved å koble til får man automatisk et bilde av hva kameraet ser på skjermen. Hovedmenyen til ISE er vist i Figur 32.



Figur 32: Kameraet har blitt koblet til i In-Sight Explorer.

Programmeringsmenyen består av 4 *Application steps* (Program-steg) [8]. Disse kan ses til venstre for bildet fra kameraet i menyen (Figur 32). Stegene er:

1. Start
   →Get Connected
   →Set Up Image
2. Set Up Tools
   →Locate Part
   →Inspect Part

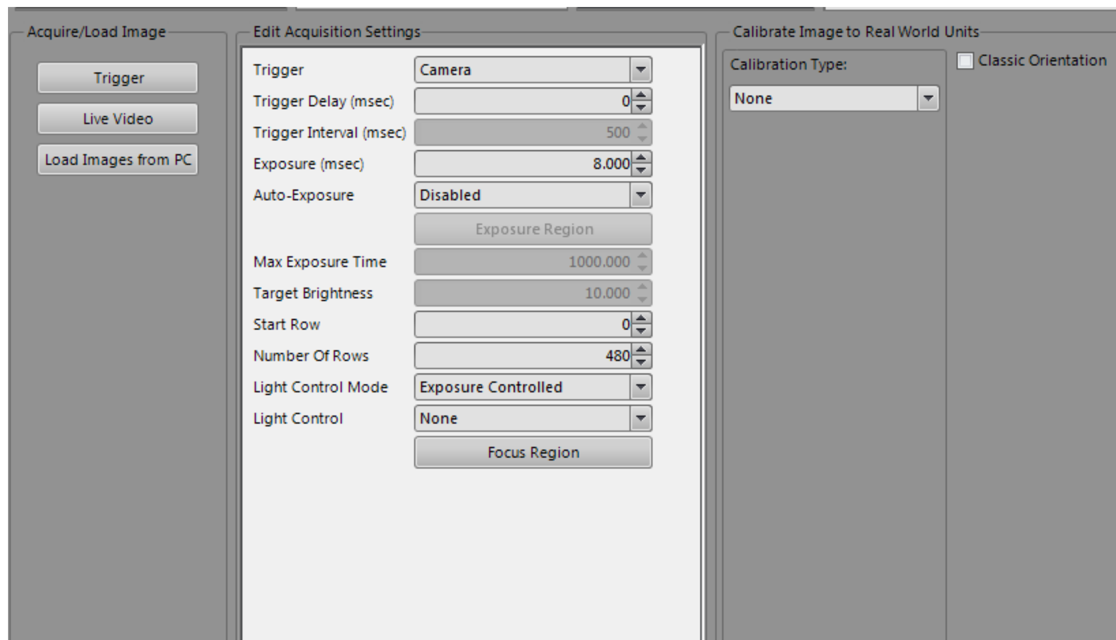
3. Configure Results
   →Inputs
   →Outputs
   →Communications

4. Finish
   →Filmstrip
   →Save Job
   →Run Job

Under *Start* finner man to valg; *Get Connected* og *Set Up Image*. Under *Get Connected* kan man velge hvilket kamera man skal koble seg til, noe som allerede har blitt gjort.

4.1 Bildeoppsett

I *Set Up Image* definerer man bildeoppsettet kameraet skal bruke. Se Figur 33.



Figur 33: Submenyen Start->Set Up Image. Her bestemmes parametere for bildetagning.

Noen viktige valg under Set Up Image er:

- Trigger
  Bestemmer hvordan bildetagning skal trigges. Man har her valgene *Camera*, *Contінous*, *Manual*, *External* og *Network*. *Camera* gjør at kameraet trigger selv, på tidspunkt da det blir programmert til å trigge. *Continous* setter opp kameraet til å trigge kontinuerlig med et bestemt tidsintervall som defineres i *Trigger Interval*.

External setter opp kameraet til å bli trigget av et eksternt fysisk signal, som kobles på en av ledningene i strømforsyningskabelen. *Manual* gjør at brukeren selv må trigge kameraet i ISE. *Network* gjør at kameraet kan trigges over LAN.

- Trigger Interval
  Her bestemmes tidsintervallet for bildetagning når trigging er satt til *Continuous*.

- Exposure
  Bestemmer eksponeringstid i millisekunder.

- Auto-Exposure
  Med denne funksjonen aktivert, bestemmer kameraet selv hvilken eksponeringstid som skal anvendes, basert på lysintensiteten i bildet (Mørke rom vil føre til lengre eksponering og omvendt).

- Start Row
  Tillater brukeren å velge hvor stor del av bildebrikken som skal tas i bruk. Dette er veldig hendig med tanke på conveyortracking, ettersom conveyoren ikke dekker hele bildet. *Start Row* definerer hvilken rad som er den første som skal leses.

- Number Of Rows
  Hvor mange rader som skal leses, i etterkant av *Start Row*.

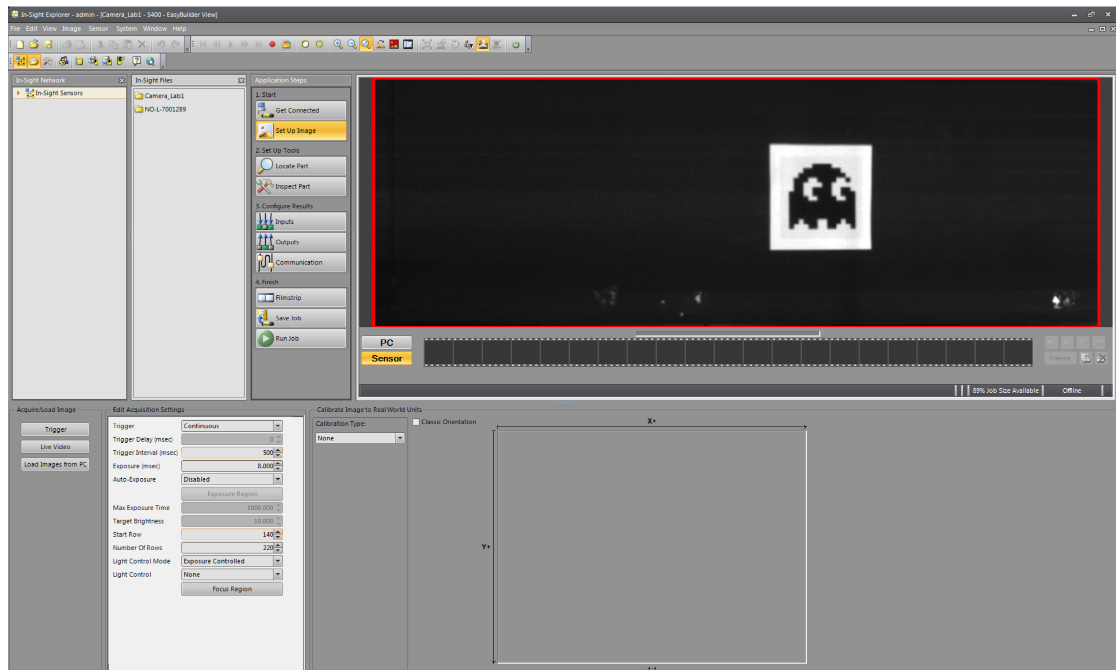
Det er ønskelig at kameraet sender ut bilder med konstant framerate, uten at IPS skal måtte spørre om det, ettersom denne spørringen og venting på svar tar tid, noe som vil senke bilderaten. Derfor er *Continous* trigger et naturlig valg.

Trigger intervallet må tilpasses etter hvor lang tid programmet som skal utføres krever, derfor avventes settingen av dette.

Eksponeringen tilpasses etter lyssetingen i rommet. I tilfellet på labben gir 8 millisekunder godt resultat, så velger dette.

Bildet tilpasses til å kun dekke conveyoren. Ved å velge Start Row = 140 og Number Of Rows = 220, oppnås dette. Man vil da redusere datamengden i hvert bilde fra $480 \cdot 640 \cdot 8bit = 300kB$ til $220 \cdot 640 \cdot 8bit = 137.5kB$ (Kameraet bruker 8 bit per piksel).

Det ferdige bildeoppsettet er vist i Figur 34. Man kan se at bildet nå er begrenset til å dekke conveyoren.



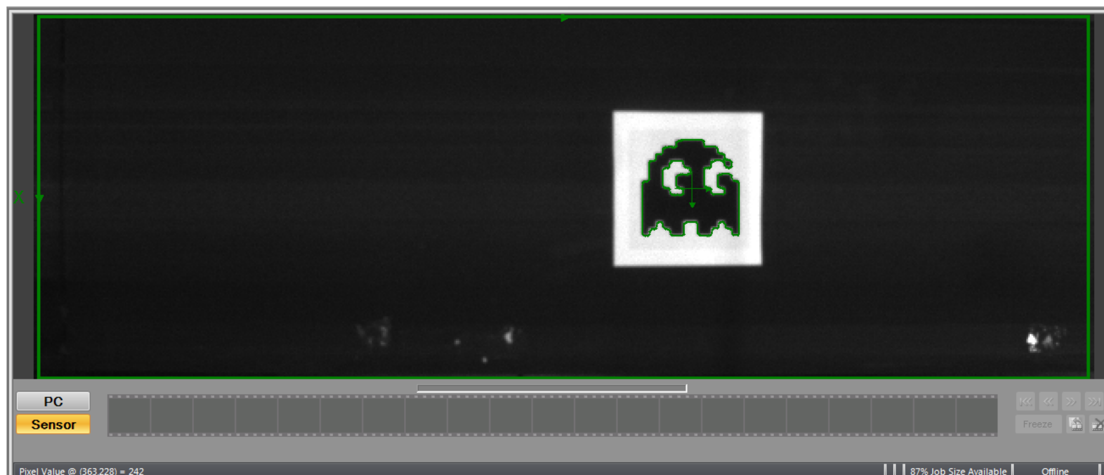
Figur 34: Tilpasning av bildetagningsparametere er fullført. Start->Set Up Image.

4.2 Objektgjenkjenning

Under *Set Up Tools* finner man submenyen *Locate Part*. I denne menyen kan man bestemme hva kameraet skal lete etter i bildet. Øverst i submenyen finner man valget *PatMax®Pattern*, som er gjenkjenningsalgoritmen som ble diskutert i avsnitt 3.9.

Patmax trenes ved å plassere et rektangel over bildet som ønskes gjenkjent. Søkeområdet bestemmes også, og man valget mellom en del forskjellige former på dette. conveyoren er nokså rektangulær, så velger rektangulært vindu som dekker hele bildet. Figur 35 viser det trente bildet, med et kryss i midten som indikerer posisjonen til objektet. Koordinatene til det detekterte objektet er gitt av sentrum av rektangelen man plasserte rundt objektet.

I *Run Job* menyen kan man nå se koordinatene til objektet som er detektert. Dette er vist i Figur 36. Ser at objekter har pikselkoordinatene [396.5,244.5]. Patmax gir altså ut posisjon med sub-piksel nøyaktighet. Orienteringen til objektet er 0, ettersom orienteringen er gitt i forhold til det trente bildet, og dette *er* det trente bildet.



Figur 35: Programmet har laget en modell av objektet.

			Name	
			Pattern_1	(396.5,244.5) 0.0° score = 100.0

Figur 36: Ser at objektet har blitt detektert, og har pikselkoordinatene 396.5,244.5.

4.3 Kommunikasjon

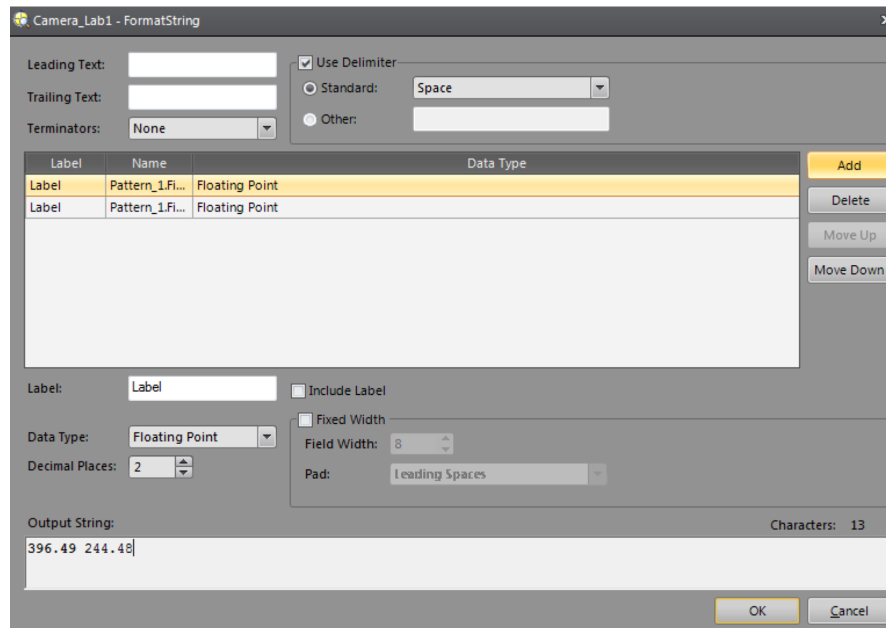
Mellom kamera og IPS skal UDP-kommunikasjon implementeres. En diskusjon om valg av grensesnitt kommer i avsnitt 5.1.

Ethernet-kommunikasjon settes opp i *Communications*-menyen, hvor man sette opp tilpassede kommunikasjonsmåter [8].

Velger UDP kommunikasjon i dropdown-menyen. Man må nå definere hva kameraet skal sende ut. I UDP-menyen finnes det et streng-oppsett, der man kan sette sammen en streng der man inkluderer de ønskede variablene. Denne menyen er vist i Figur 37. Man er her mest interessert i posisjonen og orienteringen til objektet, samt et timestamp for når bildet blir tatt, slik at det skal være mulig å estimere hvor objektet er når IPS mottar det.

I streng-oppsettet finner man blandt annet posisjon og orientering til objektet. Disse legges enkelt til i UDP-strengen. For å få timestampet, måtte det skrives et lite skript i *In-Sight Explorer Spreadsheet View*, som er et Excel-liknende vindu i ISE. Her bestemmes det at en rute i spreadsheetet skal lese klokken når kameraet trigges. Denne verdien kan deretter velges i UDP streng-oppsettet.

Nøyaktigheten på tallene kan man selv velge. Her blir det valgt nøyaktighet på 2 desima-



Figur 37: UDP-strengen settes sammen i en egen meny. Ved å velge *Add*, kan man velge blandt de tilgjengelige variablene. Her finner man blandt annet posisjon og orientering til objektet, og med å skrive et skript i SpreadSheet, kan man i tillegg få ut et lokalt timestamp for bildetagning.

ler. Ved å separere de forskjellige dataene med mellomrom, får man en streng på formen "<x.xx> <y.yy> <o.oo> <timestamp>", der 'x.xx' er x-koordinaten til det trackede objektet i millimeter, med nøyaktighet på 2 desimaler. Tilsvarende gjelder for 'y.yy', som gir y-koordinaten. 'o.oo' gir orienteringen til objektet i forhold til det trente bildet, i grader. 'timestamp' er tidspunktet da bildetagning startet på kameraets lokale klokke, og er gitt på formen 'tt mm ss msmsms', der 't' er timer, 'm' er minutter, 's' er sekunder og 'ms' er millisekunder. Et eksempel på en streng fra kameraet er "353.24 22.45 1.97 11 27 46 393", som gir at objektet har posisjon $p = [353.24, 22.45]^T$ og orientering $\theta = 1.97°$, samt at bildet ble tatt klokken 11:27:46.393 lokal kameratid[6].

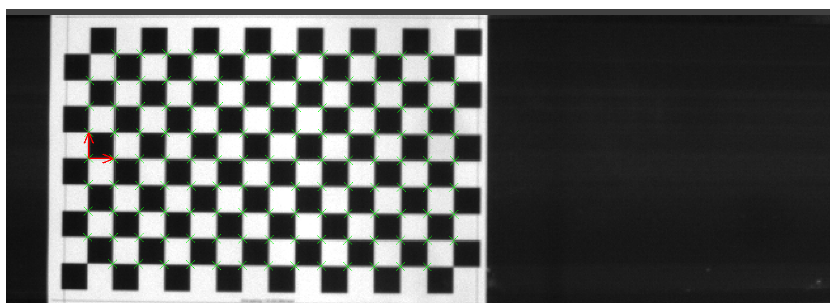
4.4 Kalibrering av kameraet

Noe som må tas hensyn til med maskinsyn, er bildeforvrengning, som ble nevnt i avsnitt 3.3.3,.

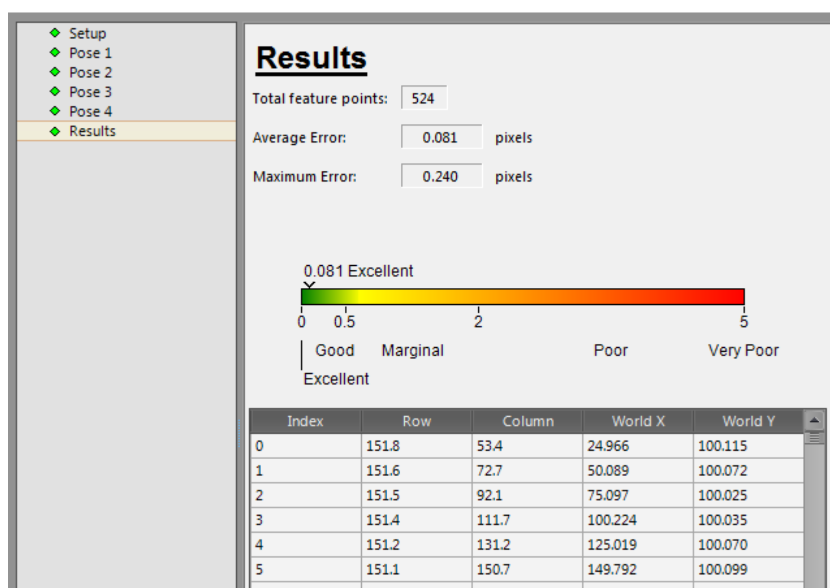
In-Sight Explorer har ferdige funksjoner for å kalibrere for dette. I *Set Up Image* menyen kan man velge kalibreringsmetode. Ved å velge kalibrering ved hjelp av *Grid* (rutenett), kan man enkelt kalibrere bildet ved å bruke bilder av "sjakkmønster", som beskrevet i av-

[6]Kameraets timestamp blir egentlig gitt med dato/måned/år i tillegg, men valgte å fjerne dette, ettersom det strengt tatt ikke er nødvedig å ha med når man jobber med millisekunder.

snitt 3.3.3. Ved å ta flere bilder av sjakkruter med kjente størrelser, slik at de overlapper hverandre, detekterer programmet hjørnene i mønsteret og lager en modell av forvrengningen. Deretter brukes modellen til å rette opp feilen. Det er tatt i bruk sjakkmønster med 25x25 millimeter ruter til kalibreringen. Når man kalibrerer bildet til ett plan, slik som her (conveyorplanet), kan også velge å kalibrere slik at pikselverdiene automatisk blir oversatt til millimeter i et koordinatsystem man selv velger i bildet. Pikselverdiene beregnes utifra at kameraet kjenner størrelsen på rutene i sjakkmønsteret. Figur 38 og 39 viser et glimt av kalibreringsprosessen i In-Sight Explorer. Kamerakalibrering med fire bilder av overlappende sjakkmønstere oppnår en gjennomsnittlig feil på 0.081 piksler, som i følge programmet er svært bra. Har nå oppnådd at kameraet sender ut en streng over UDP, som inneholder posisjon gitt i millimeter, orientering gitt i grader og et lokalt timestamp for bildetagning.

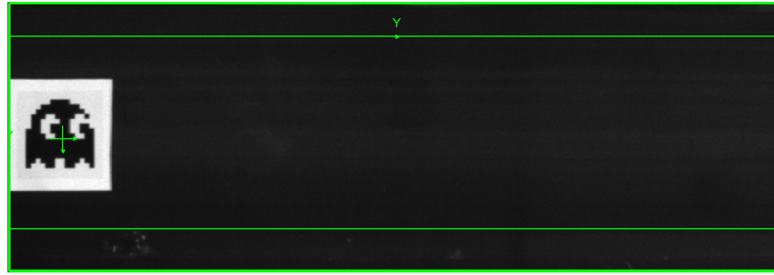


Figur 38: I kalibreringsmenyen kan man ta bilder av sjakkmønster med kjent rutestørrelse for å kalibrere for bildeforvrengning.

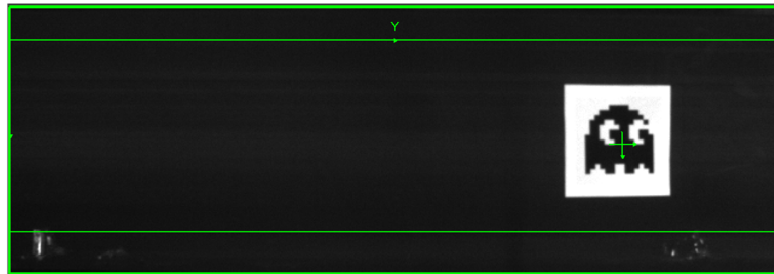


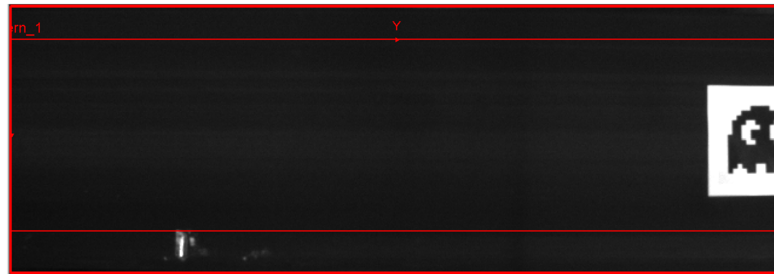
Figur 39: Resultat av kalibrering.

Y


Y

Y


Y

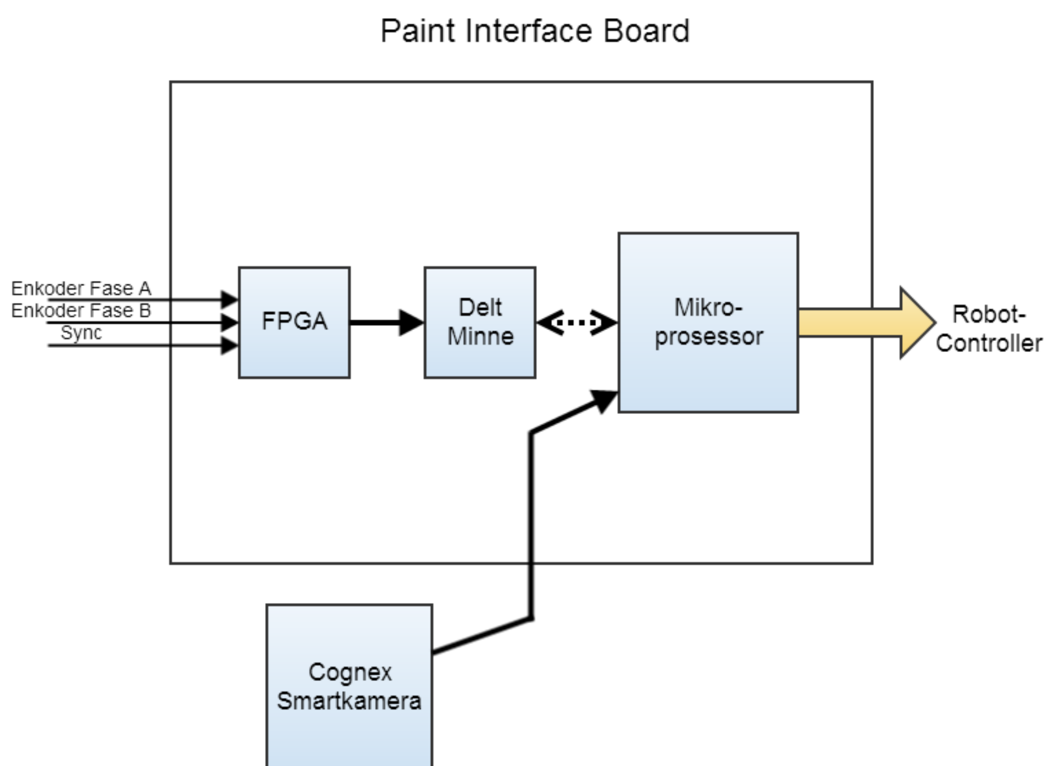

Y

5 Integrering av kamera på PIB

I de neste avsnittene blir implementering av kameraklassene gjennomgått. All koden kan finnes i vedlegget.

Som nevnt i introduksjonen, skal kameraet integreres direkte med PIB-kortet, som sitter i robotens kabinettsystem. Her må relevante C++ klasser skrives for å kommunisere med kameraet og behandle dataen, slik at roboten kan bruke informasjonen fra kameraet til å tracke objektet på conveyoren.

Se igjen på blokkdiagrammet som ble vist i innledningen av hvordan Cognex kameraet ønskes implementert i robotsystemet, vist i Figur 40.



Figur 40: Ønsket integrering av smartkameraet.

5.1 Valg av grensesnitt mellom kamera og PIB

Kameraet skal erstatte enkoderen og synkbryteren. Kameraet er utstyrt med Ethernet port, og brukeren av det står nokså fritt til å bestemme hvilken kommunikasjonsprotokoll som skal anvendes, samt hva som skal sendes fra kameraet. En rekke forskjellige kommunikasjonsprotokoller mellom kameraet og PIB-kortet har blitt sett på under utviklingen

av systemet. De to mest relevante valgene som ble undersøkt er *Transmission Control Protocol* (TCP) og *User Datagram Protocol* (UDP). En sammenlikning av TCP og UDP vises i listen under.

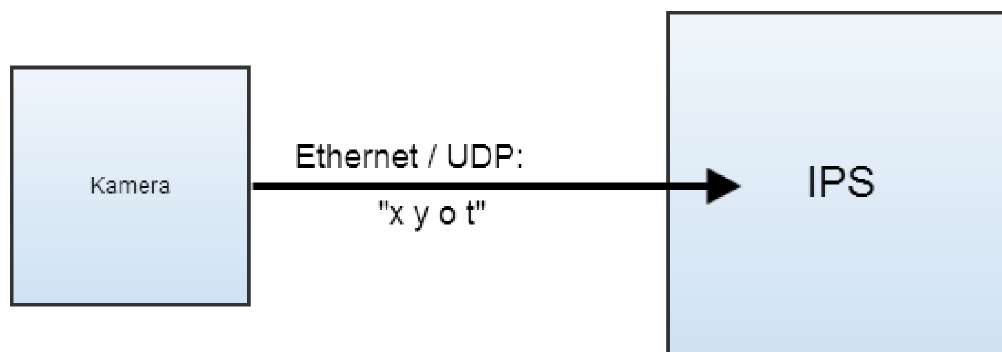
TCP	UDP
- Tilkoblingsorientert protokoll	- Tilkoblingsløs protokoll
- Hver side av kommunikasjonen samarbeider for å sikre at pakkene kommer fram	- Datapakkene sendes ut uten å bry seg om at pakkene kommer fram
- Bedre egnet for applikasjoner som krever høy pålitelighet for dataoverføring	- Egnet for applikasjoner som krever rask og effektiv overføring
- Pakkene sorteres i spesifisert rekkefølge	- Ettersom hver pakke sendes individuelt, blir ikke rekkefølgen tatt hensyn til
- Tregere enn UDP	- UDP er raskere ettersom det ikke blir gjort noe med feilsendte pakker, samt at det ikke er noen handshakes
- Garanti for at pakkene kommer fram intakte	- Ingen garanti for at pakkene kommer fram

Fordeler og ulemper ved TCP og UDP for det aktuelle systemet:

TCP	UDP
Fordeler: - Høyere pålitelighet - Garanti for at pakkene kommer fram intakte	Fordeler: - Raskere overføring - Tilkobling er ikke nødvendig, noe som betyr at kameraet kan programmeres til å spyle ut informasjon uten at noen må be om det
Ulemper: - Tregere overføring - Noe mer komplisert å implementere	Ulemper: - Ingen garanti for at pakkene kommer fram

For det aktuelle kamerasystemet er hastigheten på overføring nokså viktig, ettersom systemet er avhengig av å få posisjonsdata raskt, slik at roboten kan oppdatere posisjonen sin. UDP er vanligvis å foretrekke for sanntidsapplikasjoner, ettersom man da unngår uønsket nettverksdelay, samt at raten på datatransmisjonen kun bestemmes av senderen [16]. Det er også tanken at kameraet kun skal sende ut en tekststreng med koordinatene til objektet og et timestamp, derfor vil all informasjon kunne bli sendt i en enkelt pakke. Dette er enda en grunn til å bruke UDP. Den eneste grunnen til å velge TCP fremfor UDP, er garantien om at pakken vil komme fram, men denne vil være nokså irrelevant med tanke på at feilraten på overføring mest sannsynlig vil være neglisjerbar. Valget falt derfor på å implementere UDP-kommunikasjon mellom PIB og kameraet.

Tanken er dermed å sende x- og y-koordinatene og orienteringen til objektet som trackes, samt et timestamp som inneholder tidspunktet da bildet ble tatt, fra kameraet til IPS over Ethernet. Ethernetprotokollen UDP velges, og dataen skal sendes i en tekststreng på formen ”x y o t”, der ’x’ er x-koordinaten til det trackede objektet, ’y’ er y-koordinaten, ’o’ er orienteringen og ’t’ er et timestamp med tidspunktet da bildet ble tatt.



Figur 41: Prinsippskisse for kommunikasjon mellom kamera og IPS.

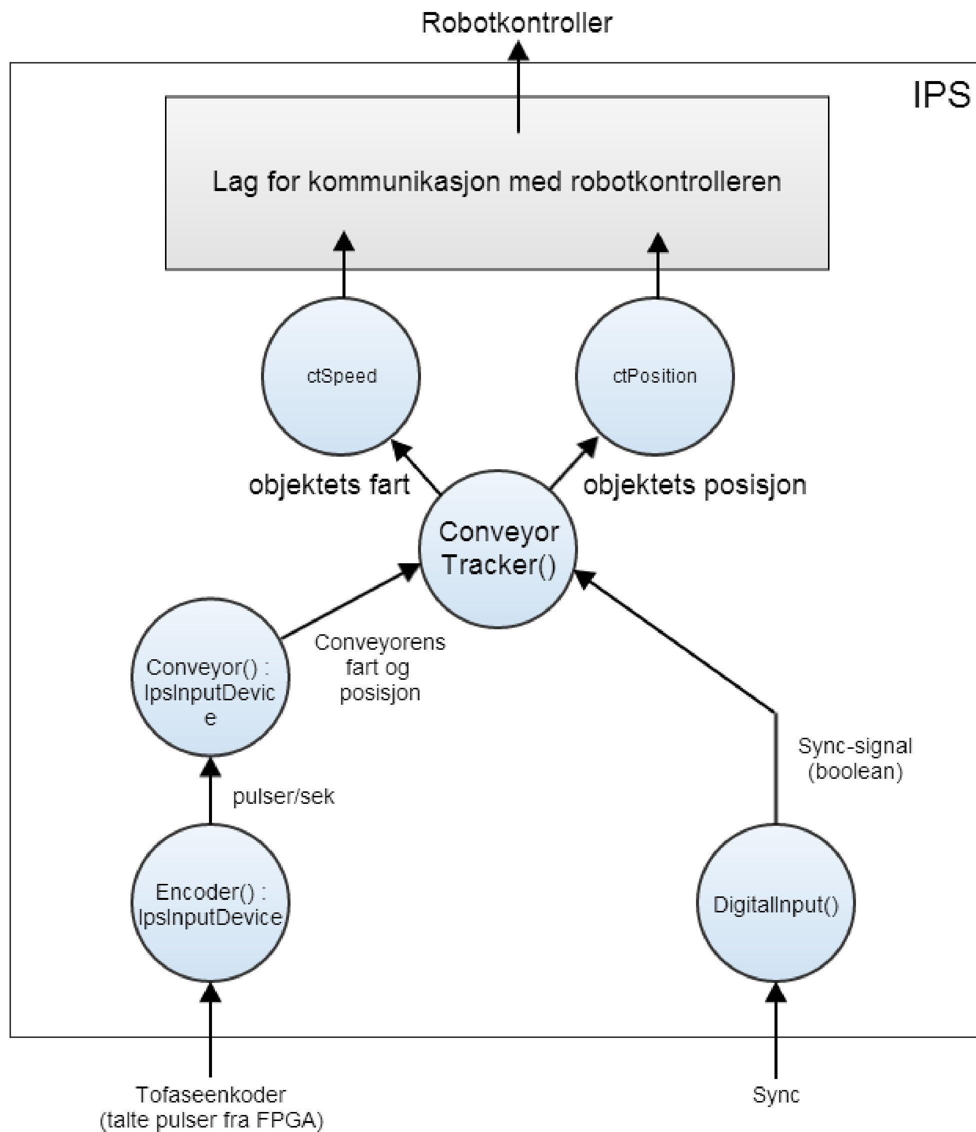
Dermed må det skrives relevante C++ klasser i IPS for UDP-kommunikasjon med kameraet, slik at dataen kan leses og behandles i IPS.

5.2 Implementering i IPS

Se igjen på skissen fra avsnitt 2.3 av arkitekturen i IPS for klassene som angår conveyor-tracking med enkoder og synkbryter. Figur 42.

Som beskrevet i avsnitt 2.3, henter `ConveyorTracker` fart og posisjon fra `Conveyor`-klassen, og synksignalet fra `DigitalInput`-klassen. Med disse tre signalene er `ConveyorTracker` i stand til å sende de nødvendige variablene til robotkontrolleren, som deretter utfører tracking av objektet. En nokså naturlig fremgangsmåte for å implementere kameratrackingen er derfor å erstatte alt nedenfor `ConveyorTracker` med kamerarelaterte klasser, slik at `ConveyorTracker` i stedet mottar de tre signalene fra disse. Man vil da oppnå tilsvarende éndimensjonal tracking som med enkoderen. En ulempe med denne fremgangsmåten er at man tilpasser seg veldig til det gamle systemet, og mister muligheten til å tracke i flere dimensjoner, noe som bruk av kamera tillater.

En annen mulighet ville være å utvide `ConveyorTracker`-klassen og de øvrige kommunikasjonslagene til robotkontrolleren, samt å endre på koden som kjører på robotkontrolleren, til å håndtere tracking i flere dimensjoner. Etter diskusjon med kolleger i ABB ble det likevel besluttet at den førstnevnte fremgangmåten måtte være veien å gå, ettersom det

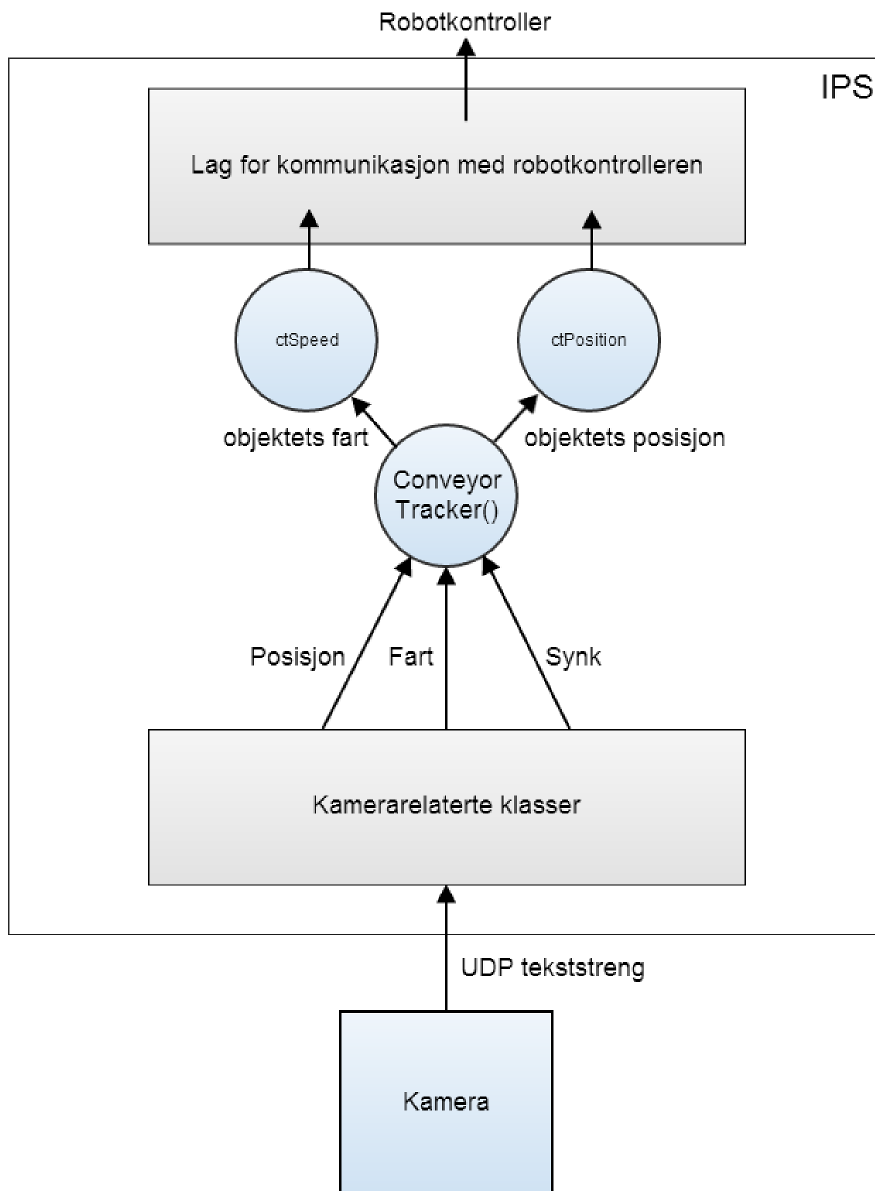


Figur 42: Skisse av arkitekturen for klassene som styrer conveyortracking med enkoder i IPS.

ville være en altfor stor oppgave å utføre sistnevnte i løpet av en masteroppgave.

Dermed ble det bestemt at trackingen som implementeres i denne oppgaven, begrenses til én dimensjon, for å tilpasse kameraet til dagens system. Men y-posisjon og orientering blir gjort klare til å anvendes i IPS, slik at `ConveyorTracker` kan utvides i videre arbeid til å ta i bruk disse.

En enkel skisse av kameratracking i IPS, slik det skal implementeres, kan dermed ses i Figur 43.



Figur 43: Skisse for ønsket implementering av kameraet i IPS.

5.3 Implementering av UDP kommunikasjon med kameraet i IPS

Det er nå fastslått i stor grad hvordan kameraet skal implementeres i IPS. IPS skal motta informasjonen om det trackede objectet over Ethernet med UDP, i form av en tekststreng som inneholder koordinatene til objektet og et timestamp for når bildet ble tatt. Det må derfor implementeres en UDP-klasse som kan kommunisere med kameraet i IPS.

Den enkleste måten å utføre dette i C++ er å implementere en UDP socket. En socket er et nettverksendepunkt definert av en IP-addresse, et portnummer og en nettverksprotokoll, og er nokså enkelt å implementere i C++. En socket kan enten defineres som en server eller en klient. Når UDP-kommunikasjon velges på kameraet, kan man velge om kameraet skal fungere som server eller klient. Dersom man velger å la kameraet være server, må datatransmisjonene *poll'es*, det vil si at en klient må *spørre* om data før kameraet sender det ut. Om man lar kameraet fungere som klient, kan man be det sende ut data til en definert IP-addresse og port med en gang dataen er klar. Det siste er helt klart en bedre løsning i dette tilfellet, ettersom datapollingen tar tid, noe som fører til økt forsinkelse. Det er i tillegg helt unødvendig å kreve at data skal polles, ettersom kameraet skal sende den samme informasjonen til det samme endepunktet hver gang. Det bestemmes derfor at kameraet skal fungere som klient, og en UDP socketserver skal implementeres i IPS.

Det har blitt implementert en klasse, `UDPserver()`, som har én enkelt funksjon, nemlig å lese data fra kameraet når en ny tekststreng er tilgjengelig, og å gjøre denne tilgjengelig for andre klasser som skal bruke informasjonen. Kildekoden for `UDPserver` finnes i vedlegget.

Et forenklet UML-diagram av `UDPserver` klassen er vist i Figur 44.
`UDPserver` er implementert med tre metoder (i tillegg til konstruktør og dekostruktør):

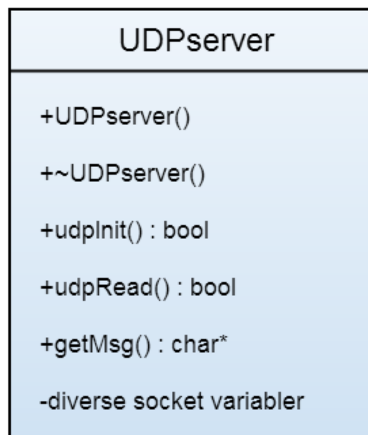
```
public bool udpInit(int port)

public bool udpRead()

public char* getMsg()
```

`udpInit` setter opp en UDP socketserver med portnummeret som tas inn som argument til metoden. Funksjonen returnerer en `bool` som er `true` dersom socketen blir suksessfullt satt opp, og `false` dersom det oppstår en feilmelding. Socketen blir satt opp til å kunne bli kontaktet av alle IP addresser, slik at forskjellige kameraer kan benyttes, uten å måtte endre IP addressen deres.

`udpRead` sjekker om en ny tekststreng er tilgjengelig på socketen. Dersom dette er tilfellet, leses strengen inn i en privat streng, og funksjonen returnerer `true`. Dersom ingen ny streng er tilgjengelig, returneres `false`.



Figur 44: Forenklet UML-diagram av `UDPserver` klassen. Plusstegn betyr at medlemmet av klassen er `public`, mens minustegn betyr at det er `private`.

`getMsg` returnerer strengen som sist ble lest inn i den private strengvariabelen.

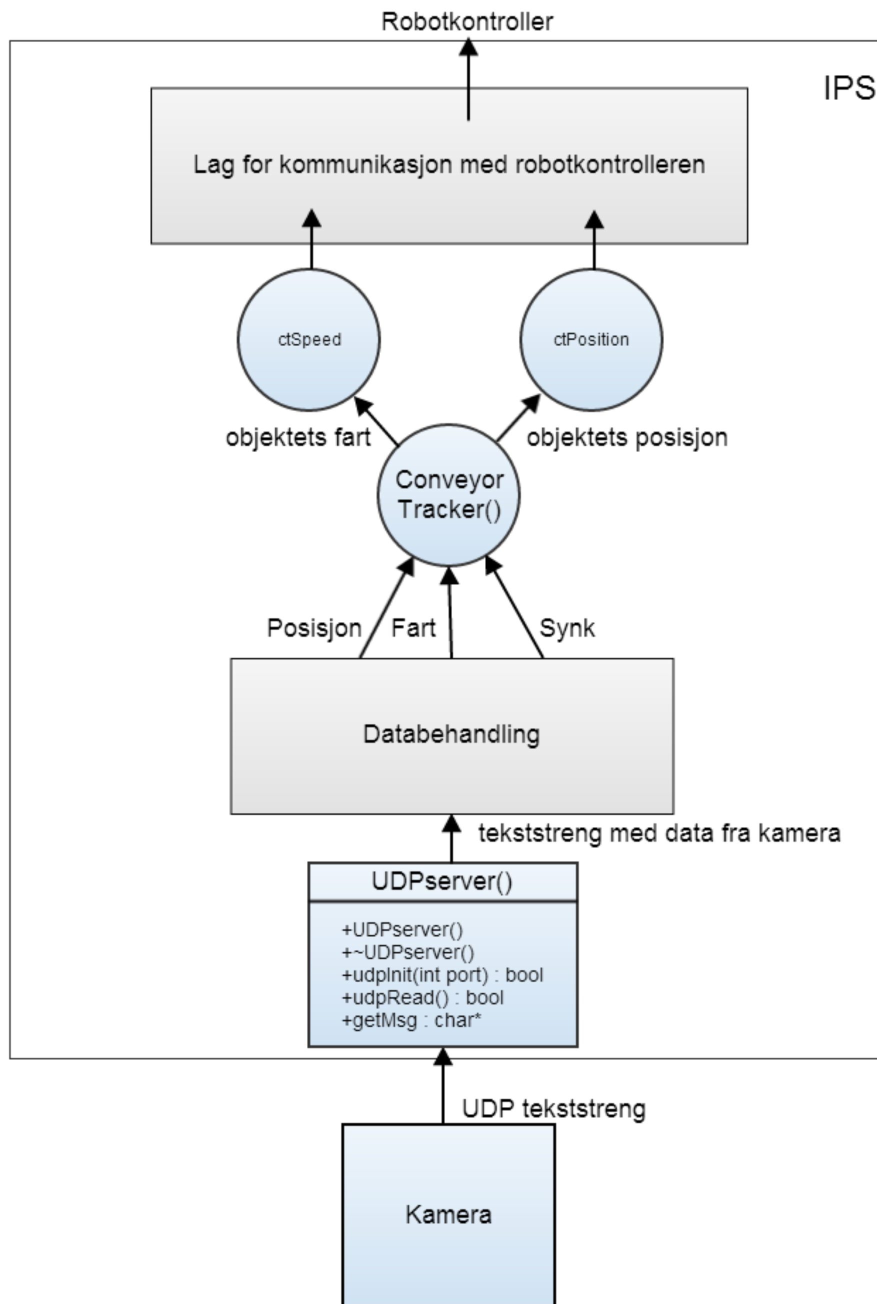
En enkel oppsummering av det som er oppnådd vises i Figur 45. Nå gjenstår det å implementere klasser som bruker rådataen fra kameraet til å beregne posisjon, fart og synksignal, og å gjøre disse tilgjengelig for `ConveyorTracker`.

5.4 Implementering av kameraklassene

`ConveyorTracker` (`CT`) er nødt til å motta posisjon og hastighet fra et `IpsDevice` objekt med et bestemt oppsett av signalene. Det er derfor nødvendig å la klassen som kommuniserer med `CT` arve fra en klasse i IpsDevice hierarkiet. Ved å la klassen arve fra `IpsInputDevice`, får man med andre nyttige metoder, blandt annet `MonitorState()` som automatisk blir kalt av en egen tråd hvert 16. millisekund.

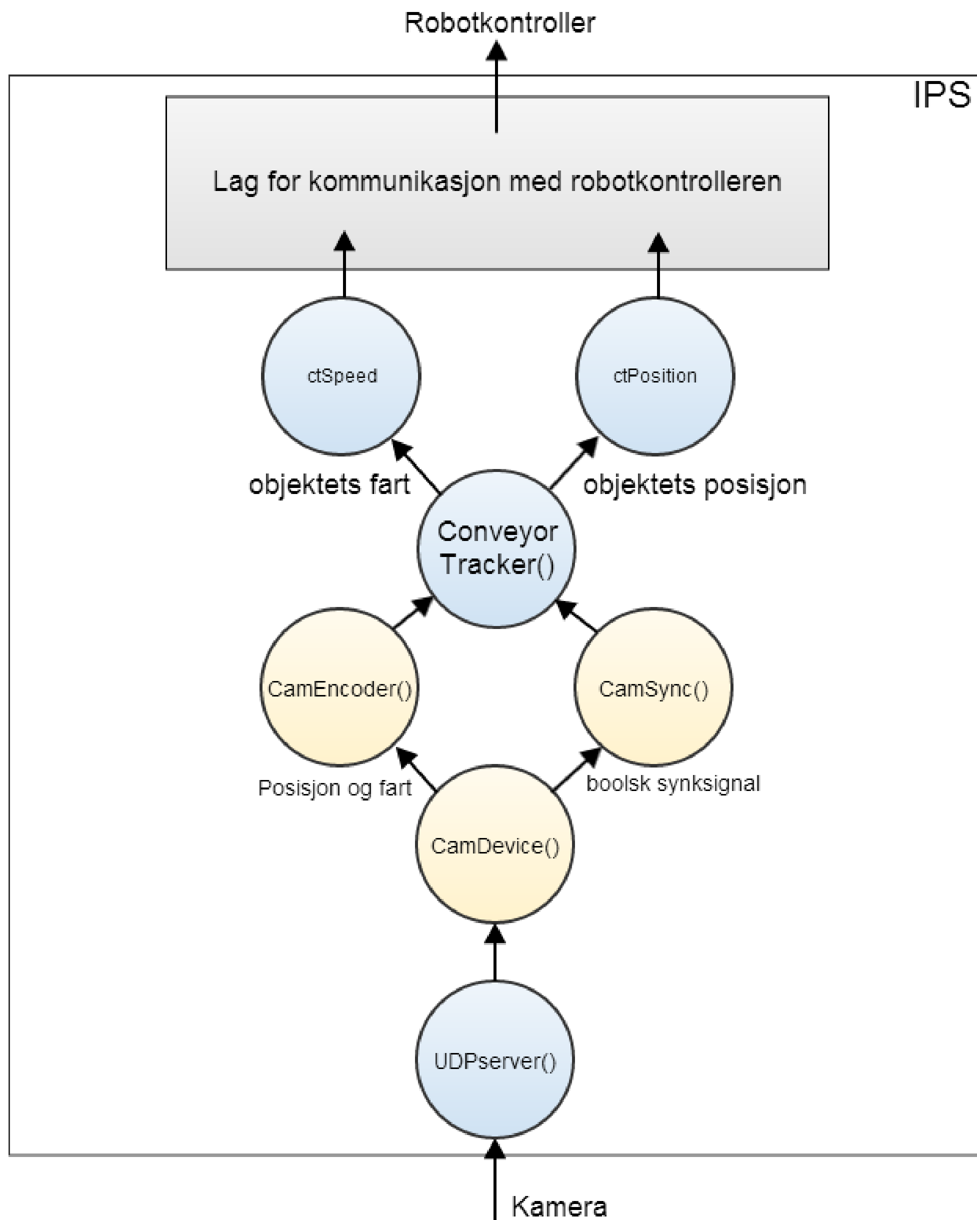
Det samme gjelder for synksignalet, som `CT` må få fra et `IpsDevice` som har et boolsk synksignal.

Det ble derfor bestemt å implementere tre klasser: Én klasse som leser kameradataen fra `UDPserver` og beregner posisjon, fart og synksignal, og to klasser som henter disse signalene og gjør dem tilgjengelig for `ConveyorTracker`. En skisse av den tenkte implementeringen ses i Figur 46. `CamDevice` (der navnet hinter til at det arver fra IpsDevice) skal lese kameradataen fra `UDPserver()` med gjevne mellomrom og beregne nye verdier for posisjon og fart, og sjekke om objektet befinner seg innenfor et definert synk-intervall i bildet. Disse signalene skal så leses av `ConveyorTracker` hvert 16. millisekund gjennom å kalle `Read`-metoder i klassene `CamEncoder()` og `CamSync`, som igjen leser verdiene fra



Figur 45: Grensesnitt for å kommunisere med kameraet implementert i IPS.

`CamDevice`. I de neste avsnittene blir implementeringen av disse klassene gått igjennom.

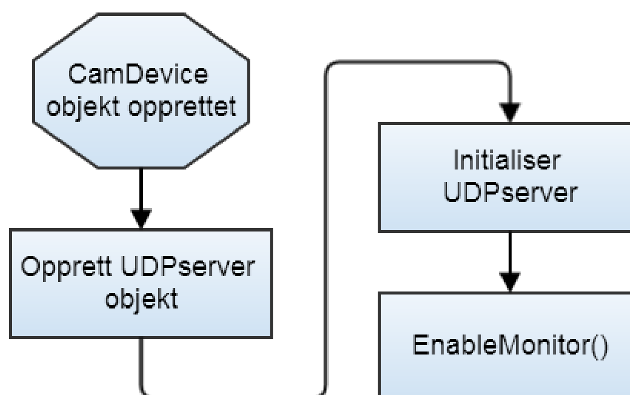


Figur 46: Tenkt implementering av kameraklassene, der alle de gule klassene arver fra `IpsInputDevice`.

5.5 `CamDevice` klassen

For å forklare implementeringen av `CamDevice`-klassen, blir det brukt noen forenklede flytdiagrammer til å forklare virkemåten til klassen. Når `CamDevice` objektet oppret-

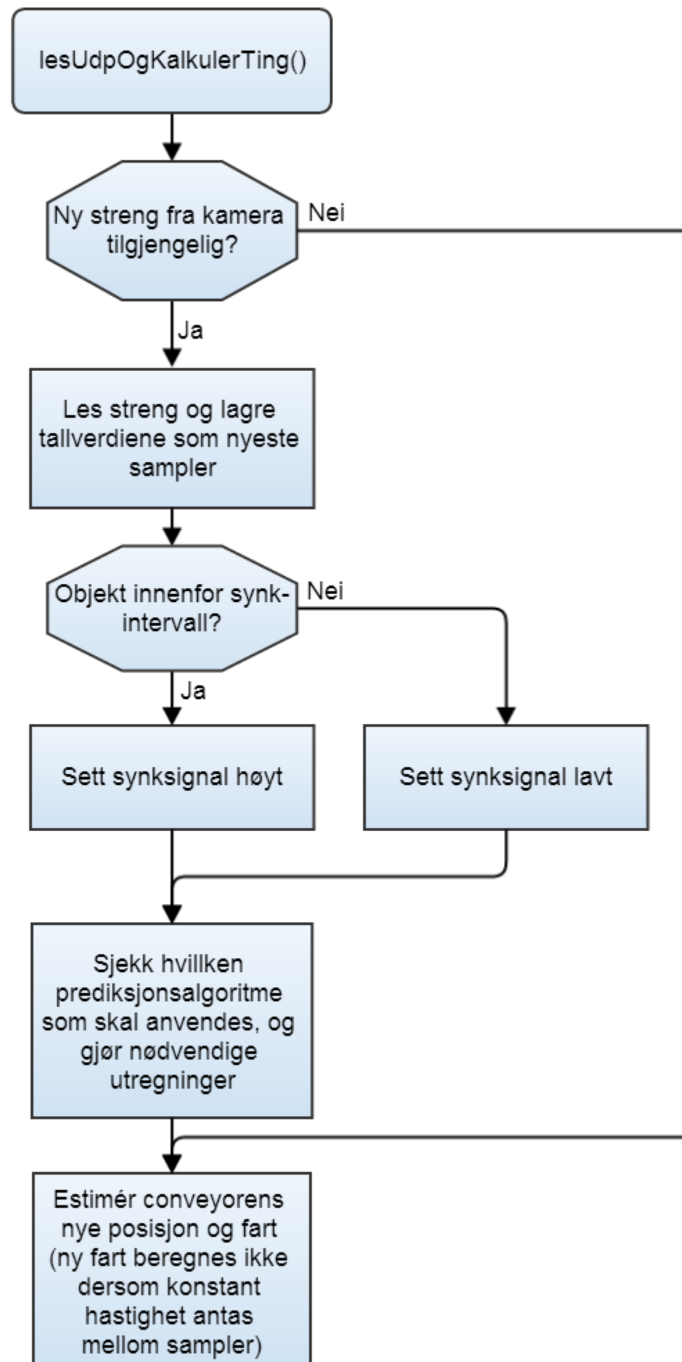
tes, noe som blir gjort under oppstart av IPS, kalles konstruktørmetoden. Her blir et `UDPserver` objekt opprettet og initialisert, og deretter kalles metoden `EnableMonitor()`, som er arvet fra `IpsInputDevice`. Denne gir beskjed til en annen del av IPS at metoden `MonitorState()`, som også arves fra `IpsInputDevice`, skal kalles hvert 16. millisekund. Flytdiagrammet til kontstruktøren kan ses i Figur 47.



Figur 47: Flytdiagram for konstruktøren til `CamDevice`.

I metoden `MonitorState()` er det kun én ting som blir gjort, og det er at metoden `LesUdpOgKalkulerTing()` blir kalt. Det er i `LesUdpOgKalkulerTing()` at alt som skal leses og beregnes blir gjort, og i og med at den kalles i `MonitorState()`, blir den som sagt kalt hvert 16. millisekund. Et forenklet flytdiagram av `LesUdpOgKalkulerTing()` er vist i Figur 48.

`LesUdpOgKalkulerTing()` sjekker om en ny tekststreng har blitt mottatt fra kameraet. Dersom en ny streng har blitt mottatt, leser den ut tallverdiene fra tekststrengen og legger dem i sine private variabler. Det er her snakk om det trackede objektets x-koordinat, y-koordinat, orientering (vinkel) og timestamp. Deretter blir det sjekket om objektet befinner seg innenfor et definert synkintervall, dvs om objektet befinner seg innenfor et bestemt intervall langs conveyor-aksen i bildet ($x_a < x < x_b$), og det boolske synksignalet settes til `true` eller `false` alt ettersom om kriteriene møtes. Det er definert en `enum`, `predictiontype`, som brukes til å bestemme hvilken prediksjonsalgoritme som skal tas i bruk. Alt ettersom hvilken prediksjonsalgoritme som skal anvendes, gjøres nødvendige beregninger for den gitte prediksjonalgoritmen. Deretter estimeres ny posisjon og fart hver gang `MonitorState` kalles hvert 16. millisekund, helt til en ny streng er mottatt fra kameraet. Den lokale variabelen av `predictiontype`, `mPredType` kan endres gjennom `SetParam()` funksjonen som er arvet fra `IpsDevice`. Det betyr at man kan skifte prediksjonsalgoritme fra RobView eller over serieporten mens systemet kjører.



Figur 48: Flytdiagram for metoden `lesUdpOgKalkulerTing()`.

Resultatet fra `CamDevice`-klassen er at man beregner estimater av posisjonen og farten til objektet på conveyoren hvert 16. millisekund. `ConveyorTracker` trenger å få disse signalene fra to selvstendige `IpsDevice` klasser, og derfor implementeres klassene `CamEncoder` og `CamSync`.

5.6 `CamEncoder` og `CamSync` klassene

`CamEncoder` og `CamSync` er nokså enkle klasser, ettersom de kun skal route signalene fra `CamDevice` videre til `CT`.

Begge klassene inkluderer `CamDevice.hpp`, slik at de kan opprette et privat `CamDevice` objekt. Addressen til instansen av `CamDevice` som er i bruk til tracking, sendes inn gjennom konstruktøren. Dermed kan klassene lese variablene `CamDevice` via `get()`-metoder.

Det er opprettet `IpsDevice`-signaler (arvede metoder), som tar i bruk variablene som leses fra `CamDevice`. Deretter *kobles* signalene til `CT` i en egen konfigurasjonsfil som legges inn i IPS. Går ikke inn på dette i detalj, men resultatet er at `CamEncoder` router fart- og posisjonssignalet fra `CamDevice`, og `CamSync` router det virtuelle synksignalet.

Dermed har man oppnådd at `CT` mottar tilsvarende signaler (fart, posisjon og synk) fra kamera, og burde fungere likt dersom signalene fra kameraet oppfører seg som de skal. Å få de til å gjøre det, blir temaet i neste seksjon.

Kildekode til alle de fire klassene, `UDPserver`, `CamDevice`, `CamEncoder` og `CamSync`, finnes i vedlegget.

6 Utvikling og testing av prediksjonsalgoritmer

I denne seksjonen gås det gjennom utvikling av algoritmer for å forbedre signalet som mottas fra kameraet, samt prediksjons/ekstrapolasjonsalgoritmer for å oppnå samme oppdateringsrate som det som er i brukt i enkodertrackingen.

Alt ettersom komplisiteten til mønsteret kameraet programmeres til å finne, samt hvor stort søkeområde i bildet som anvendes, bruker kameraet mellom 40 og 100 millisekunder på å gjennomføre en jobb og sende det ut på ethernetporten med de testene som er gjort i dette prosjektet. Dette er altfor dårlig oppdateringshastighet for posisjonen dersom man sammenlikner det med enkoderen, der dagens oppdateringsrate er én gang hvert 16. millisekund i IPS. Det må derfor å utføres signalbehandling for å forbedre posisjonssignalet fra kameraet, slik at også det får en oppdateringsrate tilsvarende enkoderen, dvs. hvert 16. ms.

Det må i tillegg tas høyde for tiden det tar fra kameraet tar bildet til data mottas i IPS. Ettersom kameraet kun sender ut et lokalt timestamp (Se avsnitt 4), er denne forsinkelsen ukjent. I tillegg leses data bare hvert 16. millisekund i IPS, så her er det også 16 millisekunders usikkerhet. Det må derfor utvikles algoritmer til å redusere forsinkelsen til et konstant offset, ved å hele tiden sammenlikne klokken i IPS med kameraets timestamp.

I de kommende avsnittene ser vi først nærmere på signalet som sendes fra kameraet, og ser på metoder som er utviklet for å forbedre signalet som mottas i IPS. Deretter ses det på forskjellige prediksjonsalgoritmer.

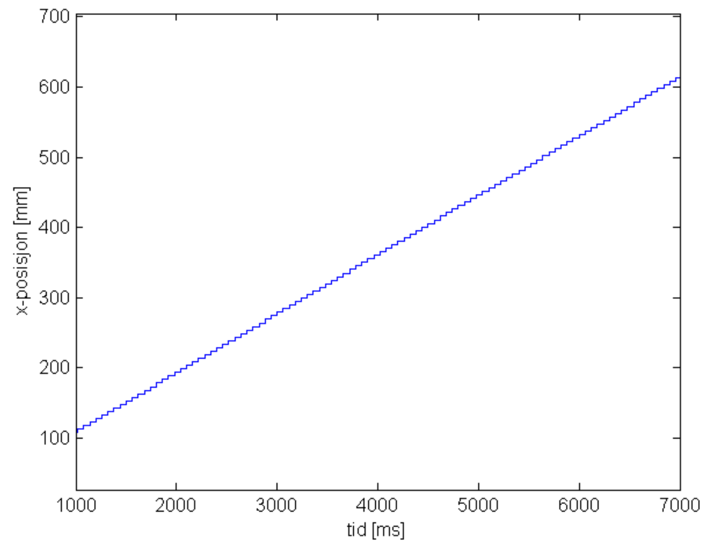
6.1 En nærmere kikk på kamerasignalet

Som nevnt i avsnitt 4, sender kameraet ut en streng på formen "<x.xx> <y.yy> <o.oo> <timestamp>", der 'x.xx' er x-koordinaten til det trackede objektet i millimeter, med nøyaktighet på 2 desimaler. Tilsvarende gjelder for 'y.yy', som gir y-koordinaten. 'o.oo' gir orienteringen til objektet i forhold til det trente bildet, i grader. 'timestamp' er tidspunktet da bildetagning startet på kameraets lokale klokke, og er gitt på formen 'tt mm ss msmsms', der 't' er timer, 'm' er minutter, 's' er sekunder og 'ms' er millisekunder. Et eksempel på en streng fra kameraet er "353.24 22.45 1.97 11 27 46 393", som gir at objektet har posisjon $p = [353.24, 22.45]^T$ og orientering $\theta = 1.97°$, samt at bildet ble tatt klokken 11:27:46.393 lokal kameratid.

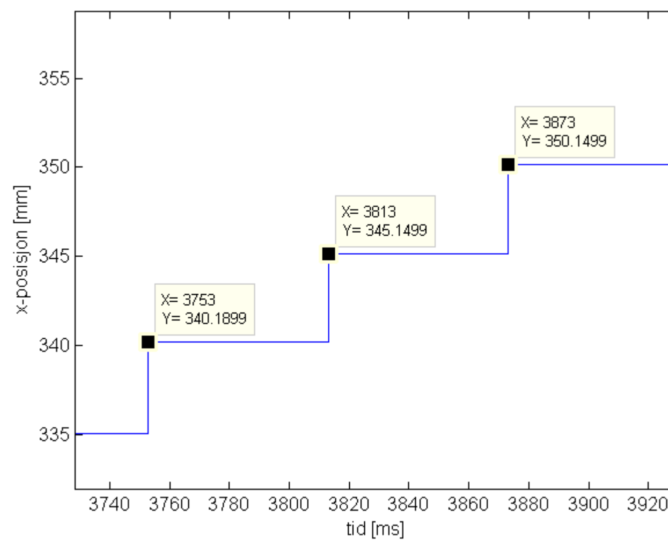
Figur 49 og 50 viser x-posisjonen som sendes fra kameraet når et objekt trackes på conveyoren med konstant hastighet. Kameraet er innstilt på å ta bilde hvert 60. millisekund. Posisjonen er plottet mot *kameraets egen klokke*, dvs. timestampet som mottas sammen med signalet. Y-aksen i plottene representerer x-posisjonen til objektet i millimeter, og x-aksen tiden i millisekunder.

Som datatip'ene i Figur 50 viser, er det 60 millisekunder mellom hvert sample i bil-

det. Analyse av timestamp-signalet i Matlab viser at tidsforskjellen mellom hvert bilde har gjennomsnittsverdi 60, med et standardavvik på 0.16 (av 7424 sampler er 90 av dem lik 59, 119 er lik 61, og resten er lik 60). Kameraet er altså tilsynelatende svært nøyaktig til å ta i bruk den periodetiden man ber det om, og gir så og si konstant samplingsrate.



Figur 49: Posisjon plottet mot kameraets timestamp.



Figur 50: Posisjon plottet mot kameraets timestamp (Forstørret).

Dersom man antar at conveyoren har konstant fart[7], ville posisjons-samplene fulgt en perfekt linje dersom kamerakalibreringen var perfekt. For å undersøke hvordan det ligger an i det virkelige systemet, plasseres det en rød linje i plottet. Linjen er et gjennomsnitt av 20 linjer beregnet av to og to punkter fra samplene, fra likningen

$$y = \frac{y_2 - y_1}{x_2 - x_1}(x - x_1) + y_1$$

Tallet 20 er valgt tilfeldig, tanken er bare å ta gjennomsnittet av mange nok til å få et godt estimat av den gjennomsnittlige stingningskurven.

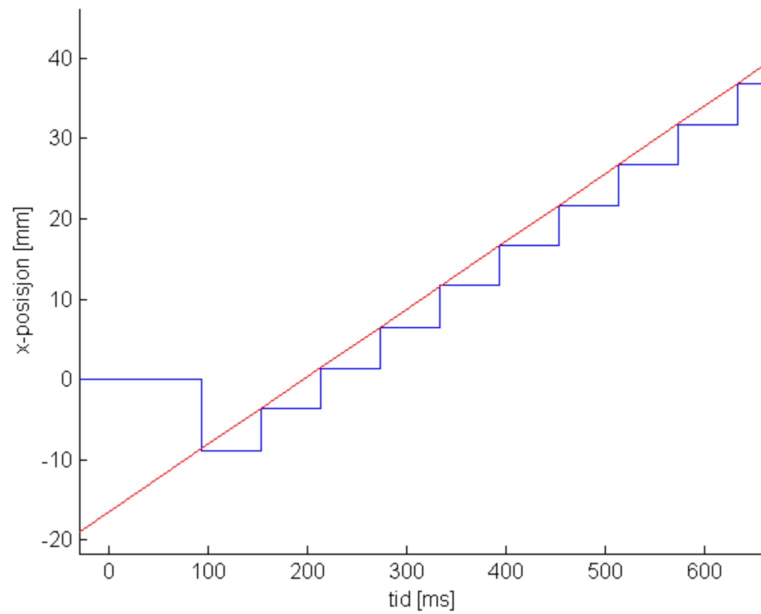
I dette tilfellet oversettes likningen til

$$x = \frac{x_j - x_i}{t_j - t_i}(t - t_i) + x_i$$

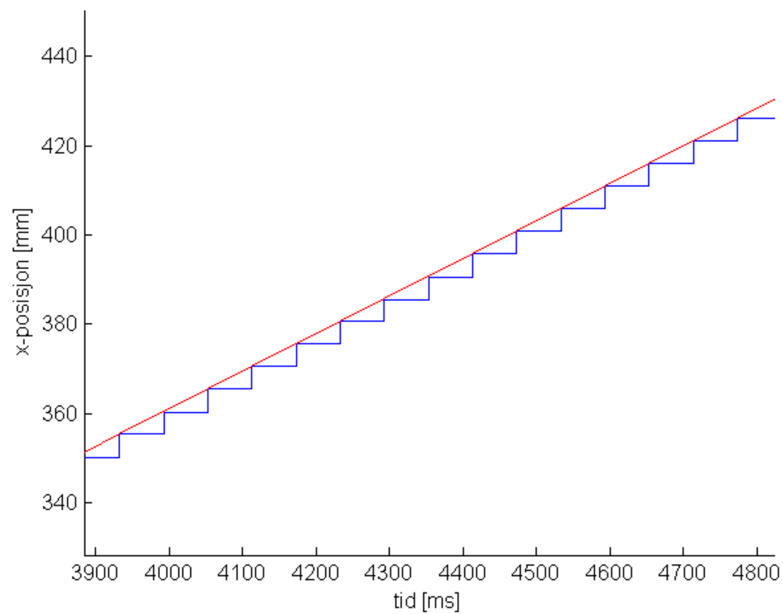
med 20 forskjellige verdier for i og j. Dette gir 20 likninger på formen $x = at + b$, og linjen i plottet er altså gitt av gjennomsnittet av parameterene a og b.

Som plottene i Figur 52 - 54 viser, treffer linjen svært godt i midten og venstre halvdel av bildet, og også i mesteparten av høyre halvdel. Helt til høyre i bildet, rett før objektet kommer utenfor kameraets synsfelt, ser det ut til å være en noe avvik i lineariteten i bildet. Avviket starter ca 10% inn fra høyre kant av bildeplanet, og forverres gradvis mot kanten. Dette avviket kan komme av flere ting; kameraet står nødvendigvis ikke helt normalt på conveyorplanet, det er muligens noe skjevt, og det kan også komme av at kameraforvrengningen ikke er fullstendig kalibrert for. Men selv på det største avviket, er det kun snakk om én millimeter fra den gjennomsnittlige stigningskurven, se Figur 54. En slik grad av linearitet på kamerasignalet burde være mer enn godt nok til lakkeringsformål, der sub-millimeternøyaktighet ikke er nødvendig.

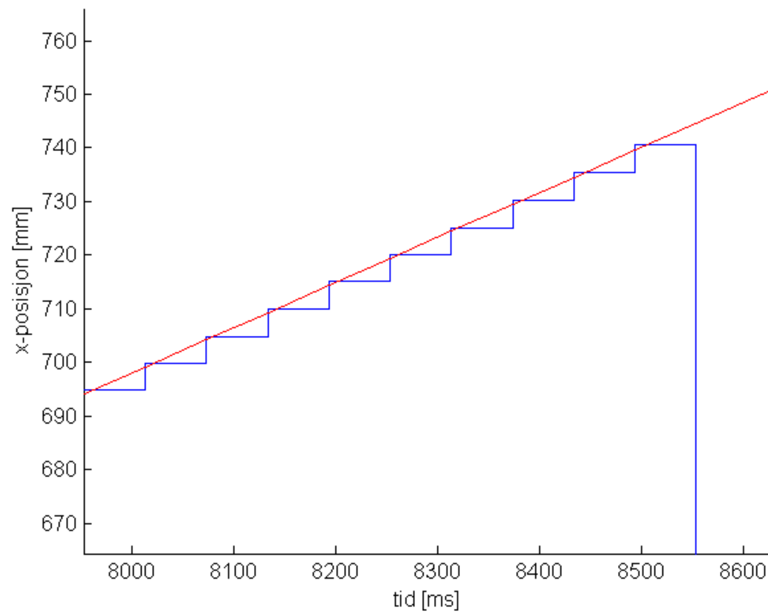
[7] I virkeligheten vil den naturligvis ikke være 100% konstant, men med conveyoren innstilt på konstant fart, må man kunne anta at den er tilnærmet konstant.



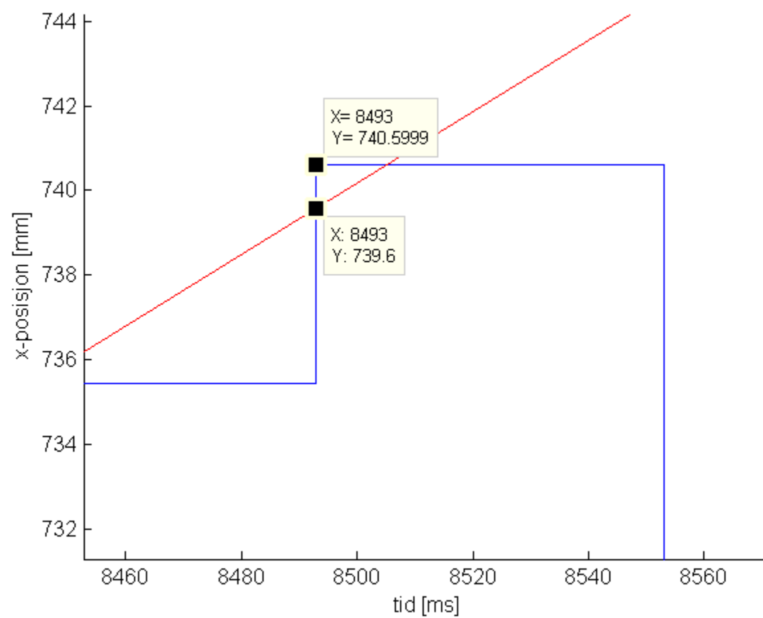
Figur 51: Starten av plotet: Blå: Objektets posisjon plottet mot kameraets timestamp for bildetagning. Rød: Gjennomsnittlig stigningskurve.



Figur 52: Midten av plotet: Blå: Objektets posisjon plottet mot kameraets timestamp for bildetagning. Rød: Gjennomsnittlig stigningskurve.



Figur 53: Høyre del av plottet: Blå: Objektets posisjon plottet mot kameraets timestamp for bildetagning. Rød: Gjennomsnittlig stigningskurve.



Figur 54: Avviket er størst i høyre kant av bildet. Dette kan komme av at kameraets akse står noe skjevt på conveyorplanet, eller av at kamerakalibreringen ikke er perfekt.

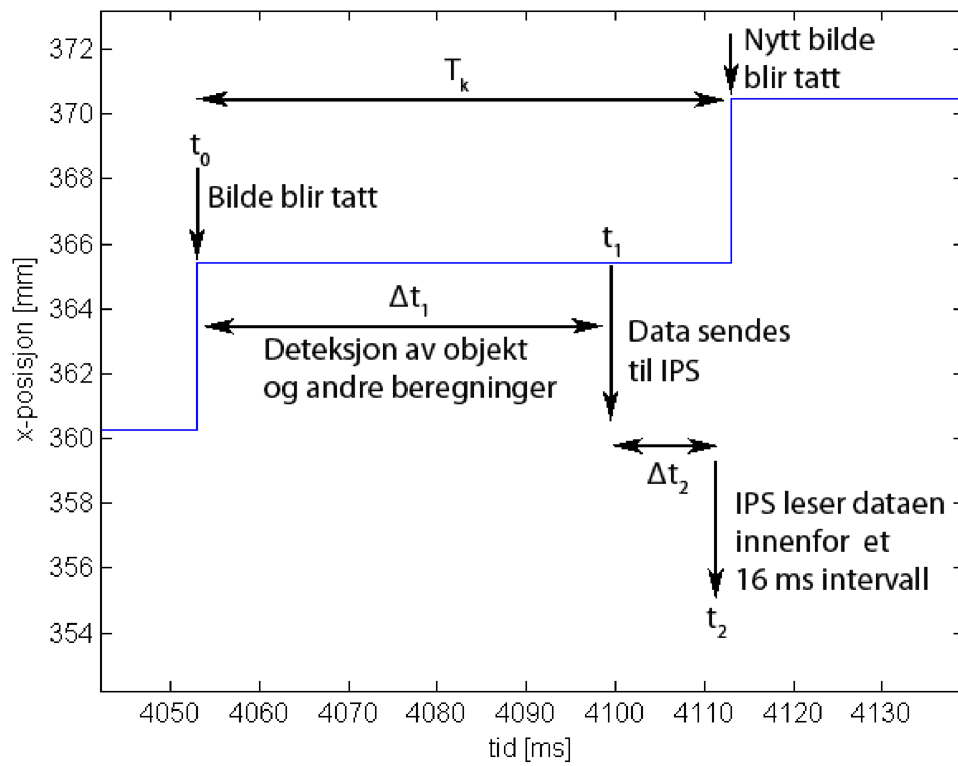
6.2 Lesing av signalet i IPS

Figur 55 viser en oversikt over tidspunktene til forskjellige hendelser fra et bilde blir tatt til data mottas i IPS. Bildet tas ved tidspunkt t_0. Deretter bruker prosessoren på kameraet Δt_1 millisekunder på å finne objektet i bildet, eller eventuelt på å finne ut at objektet ikke er tilstede, samt å sette sammen strengen som skal sendes til IPS. Ved tidspunkt t_1 sendes denne strengen over ethernet til IPS. Ettersom IPS sampler hvert 16. ms, leses strengen i et intervall på $\approx 0 - 16$ ms etter at dataen har blitt sendt. Kaller tiden fra dataen blir sendt til IPS leser den for Δt_2. T_k er kameraets periodetid, som innstilles ved programmering av kameraet. I eksempelet er denne på 60 millisekunder.

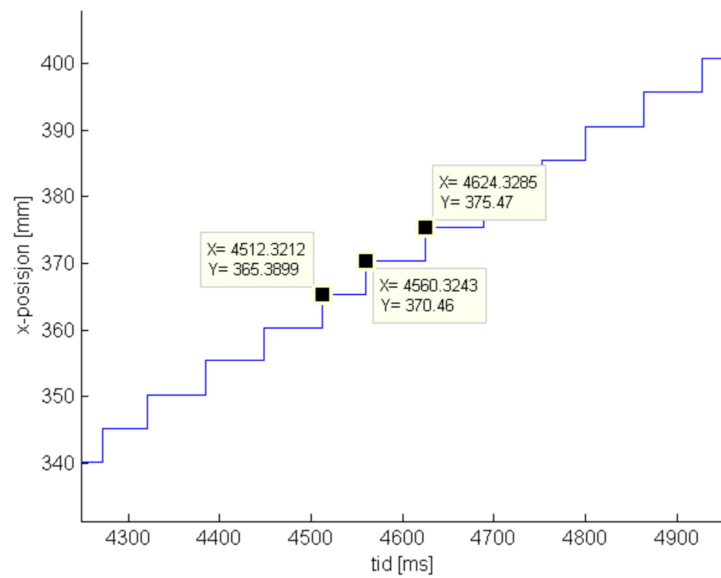
Både Δt_1 og Δt_2 er ukjente variabler. Det går derfor varierende tid mellom hver gang et bilde blir tatt, til IPS mottar dataen. Dersom man ikke tar hensyn til dette, ender man opp med et signal som ser ut som det som er vist i Figur 56. Her er x-posisjonen som mottas fra kameraet plottet mot den lokale tiden på IPS idét samplene mottas. Dette signalet er mye mer rotete enn hva kameraet sender ut. Samplingsintervallene varierer mellom 48 og 64 millisekunder, noe som er forventet når kameraet har 60 ms periodetid, og IPS leser hvert 16 millisekund. En sammenlikning av signalet kamera sender ut og det som mottas på IPS vises i Figur 57. Tidsforskjellen mellom kameraklokken og IPS-klokken er ukjent, så plottene er flyttet i tid slik at de ligger oppå hverandre.

Den beste løsningen på dette problemet, hadde vært om IPS og kameraet var klokkesynkronisert. Da kunne man enkelt ha brukt den estimerte farten til objektet til å estimere posisjonen, fordi man kjenner tidspunktet for bildetagning i forhold til den synkroniserte klokken. Det ble forsøkt å implementere klokkesynkronisering ved hjelp av *Precision Time Protocol* (PTP, IEEE-1588 standard), da det var oppgitt at kameraet støttet dette, men dette lyktes det ikke i å få til, da kameraet ikke reagerte på synkroniseringsmeldingene på nettverket[8]. Det ble istedet utviklet en metode for å estimere bildetagningstidspunktet relativt i forhold til en initielt antatt forsinkelse. På denne måten reduseres den variable forsinkelsen til et konstant offset, ved å sammenlikne timestampene som mottas fra kameraet med den lokale klokken på IPS hver gang et nytt sample mottas. En gjennomgang av dette kommer i neste avsnitt.

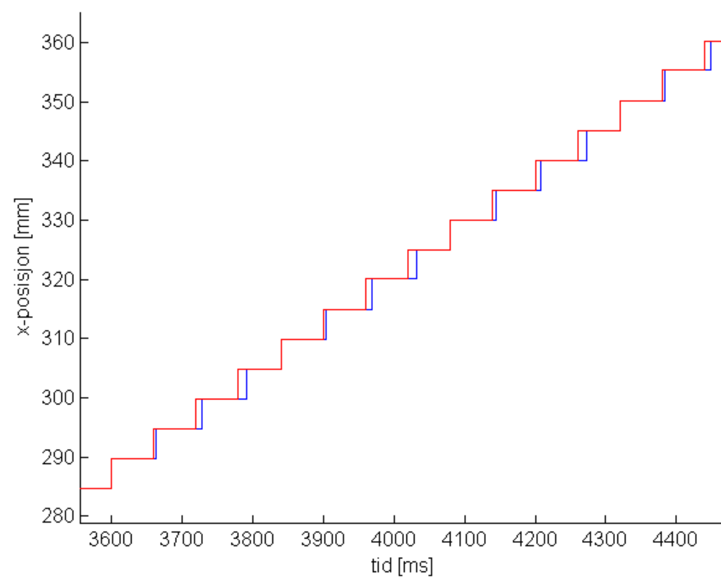
[8]Det ble i ettertid oppdaget at kameraet støtter en annen versjon av protokollen enn den som kjører på operativsystemet på PIB.



Figur 55: Et overblikk over tidspunkt for forskjellige hendelser fra bildet blir tatt, til data mottas i IPS.



Figur 56: Posisjonssignalet plottet mot den lokale klokken på IPS idét det mottas.



Figur 57: Blå: Posisjonssignalet plottet mot den lokale klokken på IPS idét det mottas. Rød: Det samme signalet plottet mot kameraklokken, dvs. timestampet fra kameraet, men tidsforskjøvet slik at de to signalene ligger oppå hverandre.

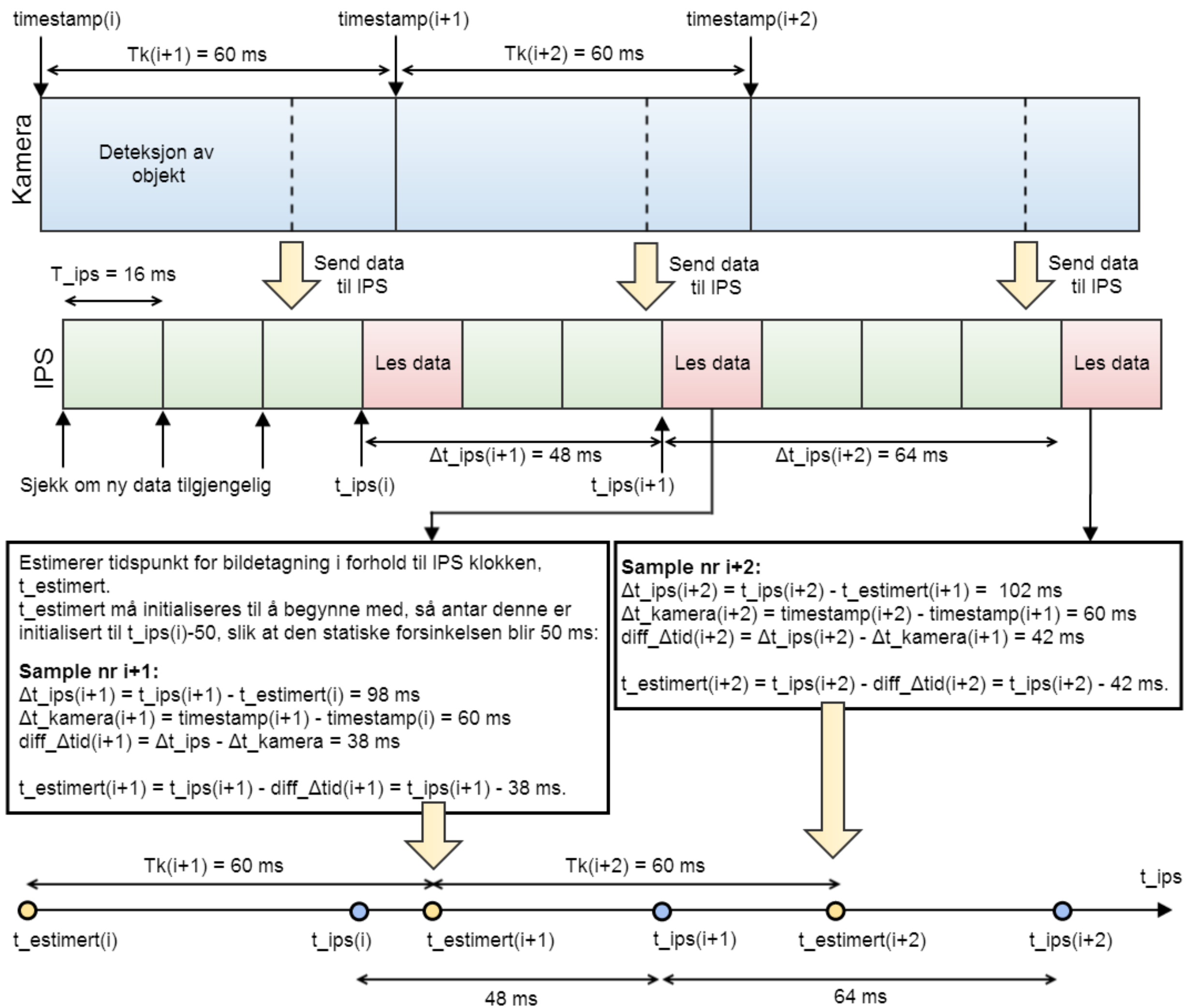
6.3 Metode for å ta hensyn til varierende forsinkelse

Dette avsnittet presenterer algoritmen som er tatt i bruk for å redusere den varierende forsinkelsen på kamerasignalet til et konstant tidsoffset mellom IPS- og kameraklokken. Målet er å estimere tidspunktet for bildetagning i forhold til IPS-klokken, forskjøvet i tid med et konstant avvik fra "ekte" tidspunkt for bildetagning. Dette konstante avviket fra det sanne tidspunktet vil aldri være kjent, ettersom IPS- og kameraklokken ikke er synkroniserte.

Figur 58 viser et tidsdiagram for kameraet og IPS. Ettersom periodetiden til IPS ($T_{ips} =$ 16 ms) ikke går gjevnt opp i periodetiden til kameraet ($T_k = 60$ ms), fører det til en varierende forskjell i tid mellom hver gang IPS sampler kamerasignalet. Denne tiden er markert som Δt_{ips} i figuren. Ettersom IPS ikke kan vite hvor lenge det er siden bildet ble tatt når den mottar signalet, må en forsinkelse antas ved initialisering. I eksempelet i bildet antas en initell forsinkelse på 50 ms, dvs. at det første sampelet som mottas, antas å være forsinket med 50 ms, slik at posisjonen som leses antas å være fra 50 ms tilbake i tid når det leses. Målet med metoden som presenteres, er å finne ut hvert bildetagningstidspunkt relativt til det initielt antatte tidspunktet, slik at den varierende forsinkelsen kan reduseres til et statisk (ukjent) offset. Eller formulert på en annen måte: Formålet er å plassere alle påfølgende sampler i det korrekte tidspunktet på IPS-klokken, med et avvik gitt av feilen på den initielt antatte forsinkelsen på første mottatte sample. Dette vil gjøre det langt enklere å utføre prediksjon, fordi man "vet" hvilket tidspunkt bildet er tatt i forhold til IPS klokken. All prediksjon vil naturligvis også være begrenset av det ukjente konstante avviket.

Algoritmen går ut på å sammenlikne $T_k(i)$, dvs tidsforskjellen mellom timestampene som kameraet sender, med den målte tidsforskjellen mellom samplingstidspunkt i IPS. Ved å hele tiden justere for denne forskjellen, finner man samplingstidspunktet på IPS-klokken, med et konstant avvik gitt av filen i den initielle antakelsen. Algoritmen er vist i Algoritme 1.

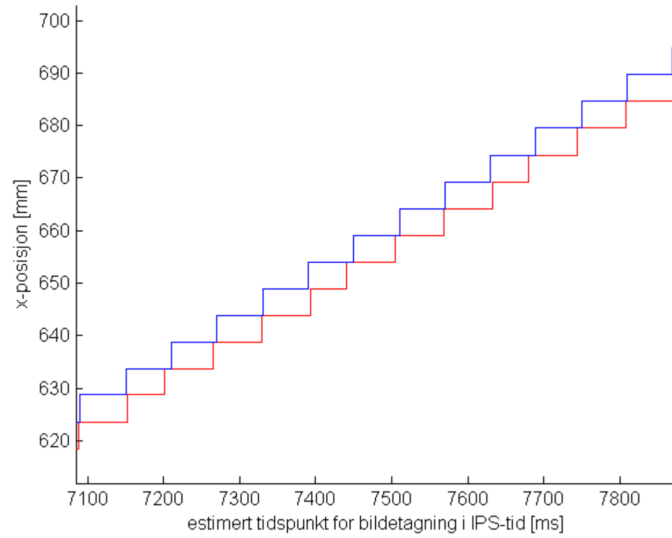
Resultatet ved simulering i Matlab (med ekte kameradata) kan ses i Figur 59 og 60. Matlab sjekker om ny data er tilgjengelig hvert 16. ms ved å ha en venteløkke som sjekke om 16 millisekunder har gått siden sist. Her er den blå kurven x-posisjonen plottet mot de estimerte tidspunktene for biltetagning, mens den røde kurven er den samme posisjonsdataen plottet mot tidspunktet da den ble lest. Den initielle forsinkelsen er antatt til 50 ms, derfor ligger de estimerte tidspunktene ≈ 50 ms foran avlesingstidspunktene. Avstanden mellom samplene på den blå kurven er alltid lik forskjellen i timestampene som er mottatt, som jo er hensikten.



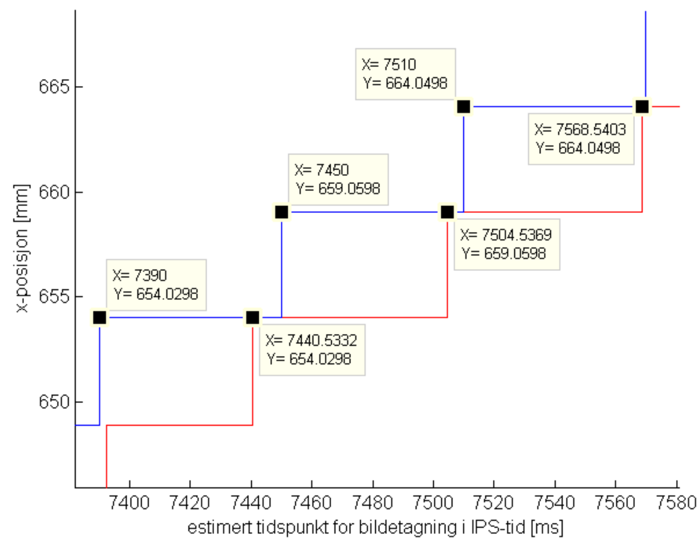
Figur 58: Øverst: Tidsdiagram for kamera og IPS. Nederst vises de estimerte bildetagningstidspunktene (De gule sirklene) på IPS klokken, som nå er plassert riktig relativt til den initielle antakelsen. De blå ringene viser tidspunktet IPS faktisk leser verdiene.

Input:
x_r: Rå posisjon mottatt fra kamera
timstamp: timestamp fra kamera
Output:
$\hat{t}_b$: estimert tidspunkt for bildetagning i forhold til den lokale klokken i IPS, med konstant avvik fra faktisk tidspunkt.
while *1* **do**
    if *Ny kamerastreng tilgjengelig* **then**
        x_r = les posisjon fra kamerastreng;
        `// lagre forrige timestamp, og les nytt:`
        forrige_timestamp = timestamp;
        timestamp = les timestamp fra kamerastreng;
        `// Estimer tidspunkt for bildetagning på ips klokken:`
        t = les klokke;
        $\Delta t = t - \hat{t}_b$;
        $\Delta timestamp = timestamp - forrige_timestamp$;
        diff_Δtid = $\Delta t - \Delta timestamp$;
        $\hat{t}_b = t -$diff_Δtid;
    end
end

Algoritme 1: Estimering av tidspunkt for bildetagning i forhold til lokal klokke på IPS.



Figur 59: Blå: x-posisjon plottet mot estimert bildetagningstidspunkt. Rød: Den samme posisjonsdataen plotet mot samplingstidspunkt (I Matlab).

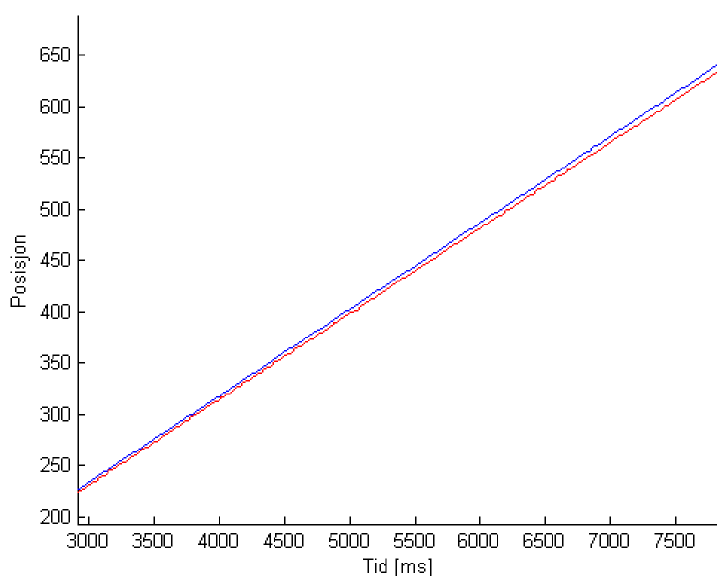


Figur 60: Blå: Tidsforskjellen mellom de estimerte bildetagningstidspunktene er lik forskjellen mellom timestampene kameraet sender, som nesten alltid er lik 60 ms (når kameraet er programmert til å ta bilde hvert 60. ms). Rød: Tidsforskjellen mellom avlesningstidspunkt er alltid hele multipler av 16, ettersom det kun sjekkes om ny data er tilgjengelig hvert 16. ms.

6.4 Lineært avvik mellom kamerasignalet og enkodersignalet

Selv om det har blitt forsøkt å kalibrere kameraet så godt som mulig for bildeforvrengning, og kameraet har blitt forsøkt plassert med aksen normalt på conveyorplanet, viser det seg at det likevel blir et visst avvik i stigningskurven mellom enkodertrackingen og kamerasignalet.

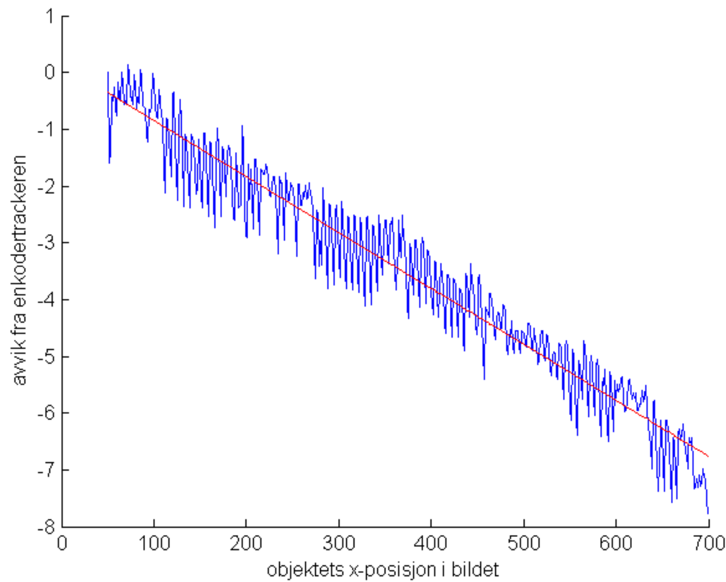
Figur 61 viser et sammenlikningsplot mellom stigningskurvene til enkoderen og kamerasignalet. Man kan se at kameraet (blå kurve), stiger noe hurtigere enn enkoderen (rød kurve). Dette kan komme av at kameraet står skjevt på conveyoren, eller at kamerakalibreringen ikke er god nok. Figur 62 viser plot av avviket $e = x_{enkoder} - x_{kamera}$, der enkodersignalet er resamplet i Matlab til å passe med kamerasamplene. Avviket er gitt som funksjon av den målte kameraposisjonen. Man kan se på plottet at denne ser ut til å være tilnærmet lineær.



Figur 61: Blå: Posisjon lest fra kameraet. Rød: Logget signal fra enkodertrackeren.

For å prøve å rette opp i dette, velges det å estimere det lineære avviket, og korrigere for dette i beregning av posisjon gitt av kameraet.

Den lineære feil-funksjonen estimeres med *Least Squares* metoden (Minste kvadraters metode)[13]. Metoden går ut å på finne den lineære funksjonen $y = ax + b$ som gir minste sum av kvadratisk feil for alle punkter i datasettet. Når funksjonen for det lineære avviket er estimert, dvs. vi har funnet (estimert) a og b, kan parameteren a brukes til å rette opp i det lineært økende avviket. Det statiske avviket, b, er vi ikke interessert i, ettersom kameraet kun bryr seg om posisjonen når objektet er til stedet i bildet, mens



Figur 62: Estimering av avvik fra enkoderposisjon som funksjon av objektets posisjon i bildet.

enkoderens posisjon fortsetter å løpe konstant.

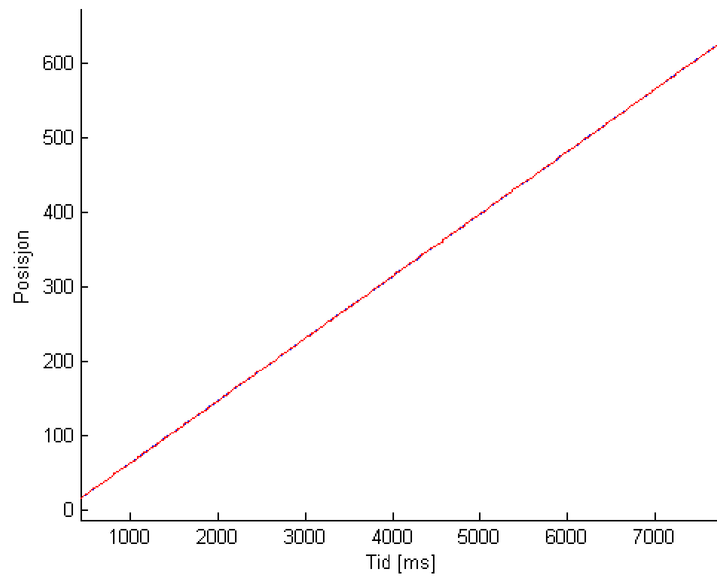
Når man med LS metoden har estimert a og b, antas det at

$$e = x_{enkoder} - x_{kamera} = ax_{kamera} + b$$

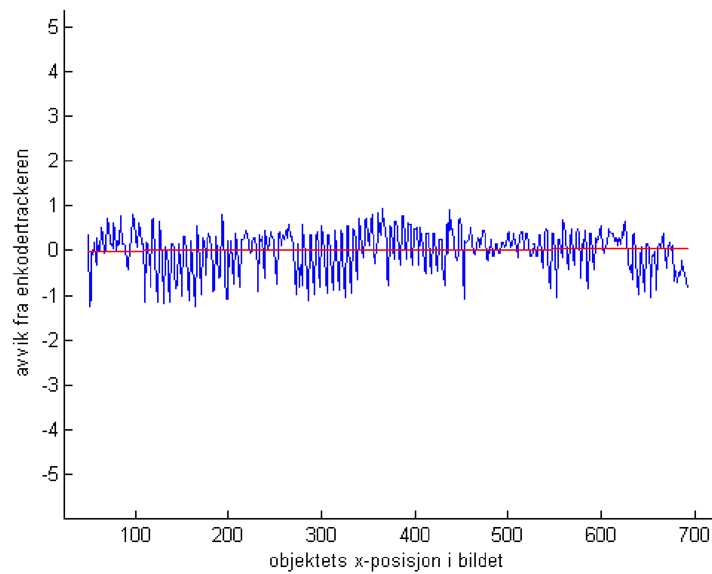
altså avviket mellom enkoderen og kamera gitt som funksjon av posisjonen fra kameraet. Dette gir videre at

$$x_{enkoder} - b = (1 - a)xkamera$$

Forventer altså nå at dersom signalet fra kameraet multipliseres med $(1 - a)$, blir man i stor grad kvitt det lineære avviket. Figur 63 viser sammenlikningsplottet etter at avviks-korriksjon er anvendt på kamerasignalet, og Figur 82 viser tilsvarende feilplot for dette. Det lineære avviket er praktisk talt borte, så metoden ser ut til å fungere svært bra.



Figur 63: Etter avvikskorreksjon: Blå: Posisjon lest fra kameraet. Rød: Logget signal fra enkodertrackeren.



Figur 64: Estimer avvik mellom enkoder og kamera som funksjon av posisjonen til kameraet med lineær feilkorreksjon anvendt.

6.5 Prediksjonsalgoritmer

I de kommende avsnittene ses det på to forskjellige prediksjonsalgoritmer.

I avnitt 6.5.1 gås det gjennom prediksjon der det antas konstant hastighet mellom hvert sample.

I avnitt 6.5.2 gås det gjennom prediksjon der det anvendes tilbakekoblet utgang.

6.5.1 Konstant fart mellom sampler

Med objekter på conveyor er det nokså logisk å anta konstant hastighet mellom hvert sample. Dette fordi conveyoren som regel er innstilt på konstant hastighet så lenge den er i gang. Begge prediksjonsalgoritmene som testes antar dette.

Algoritme 1 som er beskrevet i avsnitt 6.3, tas i bruk for å tidfeste samplene som mottas i IPS. Farten til objektet, v, estimeres ved å ta forflytningen mellom siste og forrige sample, og dele den på tidsforskjellen i siste og forrige timestamp. Posisjonen i ethvert tidspunkt, $\hat{x}$, beregnes ved å legge til produktet av farten og passert tid mellom estimert bildetagningstidspunkt og avlest tid i IPS:

$$\hat{x} = x_r + \hat{v} * (t - \hat{t}_b)$$

der $\hat{x}$ er estimert nåværende posisjon, $\hat{v}$ er estimert fart, t er avlest tid, og $\hat{t}_b$ er estimert bildetagningstidspunkt gitt av Algoritme 1.

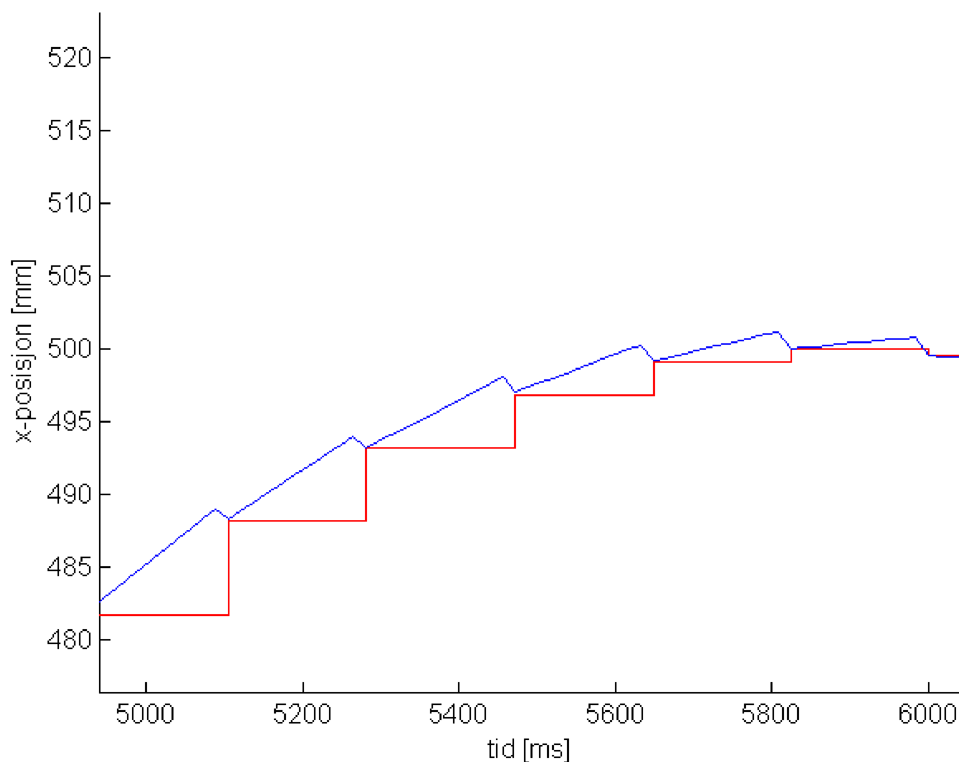
På denne måten estimeres ny posisjon hver gang `MonitorState()` funksjonen blir kalt, hvert 16. millisekund.
Den fullstendige algoritmen for ekstrapolering av signalet med antatt konstant hastighet mellom samplene er vist i Algoritme 2.

Input:
x_r: Posisjon mottatt fra kamera
timstamp: timestamp fra kamera
Output:
$\hat{x}$: estimert nåværende posisjon.
while *1* **do**
  if *Ny kamerastreng tilgjengelig* **then**
    `// lagre forrige posisjon og timestamp, og les nye:`
    forrige_$x_r = x_r$;
    x_r = les posisjon fra kamerastreng;
    forrige_timestamp = timestamp;
    timestamp = les timestamp fra kamerastreng;
    $\hat{t}_b$ = estimer bildetidspunkt(Algoritme 1);
    `// Estimer fart:`
    Δtimestamp = timestamp - forrige_timestamp;
    $\hat{v} = (x_r$-forrige_$x_r)/\Delta$timestamp;
  end
  `// Beregn nåværende posisjon:`
  t = les klokke;
  $\hat{x} = x_r + \hat{v} * (t - \hat{t}_b)$;
end

Algoritme 2: Prediksjon / ekstrapolasjon av signalet ved antatt konstant fart mellom samplene.

6.5.2 Konstant fart med tilbakekobling

En av ulempene med metoden over, er at posisjonsestimatet kan gjøre store hopp idét nye sampler mottas. Det kommer av at estimatet $\hat{x}$ er direkte påvirket av råverdien. Dersom conveyoren aksellerer, vil prediksjonen bomme, og man vil få et sagtannmønster på $\hat{x}$. Et eksempel på dette er vist i Figur 65. Det blå signalet er prediktert posisjon, og det røde er mottatt rådata. Ettersom rådataen har ulineær kurve, bommer prediksjonen og det oppstår et sagtannmønster ved hvert nye sample.



Figur 65: Rød: Rådata. Blå: Prediktert posisjon. Som man ser bommer prediksjonen når signalet er ulineært, og prediksjonen får sagtannform.

En annen metode å estimere farten på, er å bruke et signal med tilbakekobling, istedet for å summere rådataen direkte inn i prediksjonen. Ved å la

$$\hat{x}(i) = \hat{x}(i-1) + \hat{v}(t(i) - t(i-1))$$

unngår man de stygge hakkene. Men nå lever $\hat{x}$ sitt eget liv, ettersom det ikke er noen direkte kobling til den avleste posisjonen. Får å unngå at signalet får store avvik fra faktisk posisjon, legges det til et feil-ledd i beregningen av $\hat{v}$, slik at avviket vil korrigeres for i form av øke farten hvis mottatt posisjon er høyere enn forventet, og senke farten hvis den er mindre enn forventet.

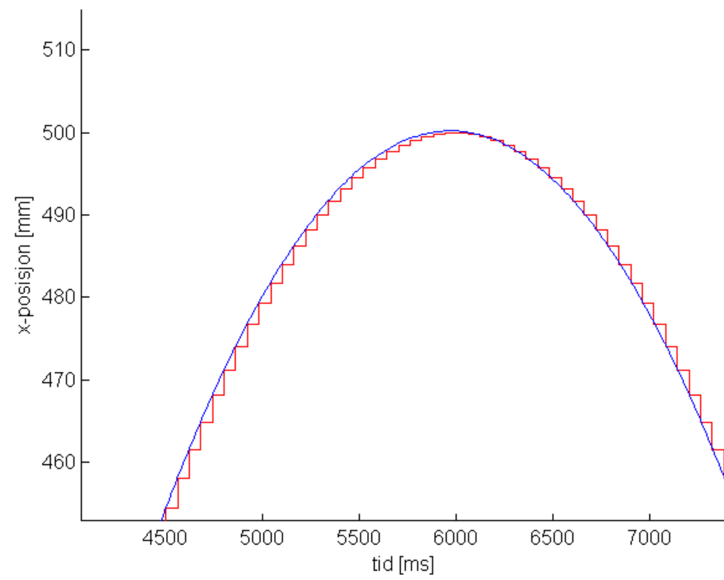
$$\hat{v}' = \hat{v} - c\hat{e}$$

der $\hat{v}'$ er farten korrigert for estimert avvik og $\hat{e}$ er avvik på prediksjonen fra estimert faktisk posisjon, og c er vektingsfaktor.

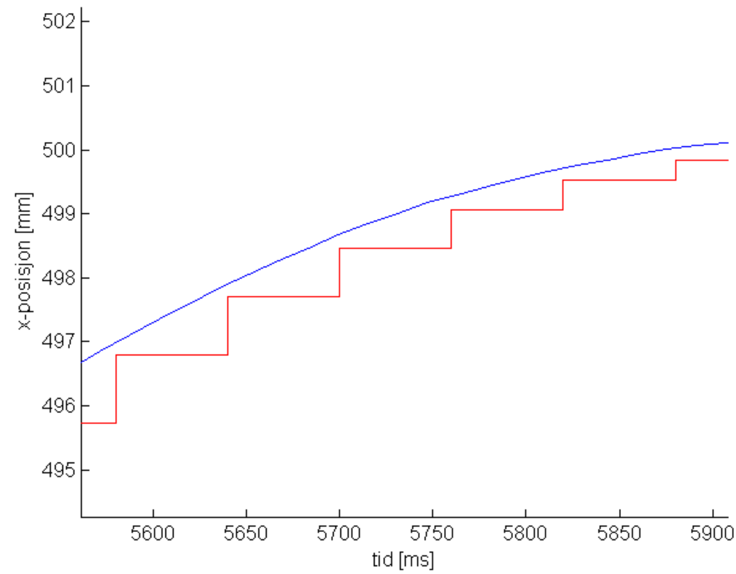
```
Input:
x_r: Posisjon mottatt fra kamera
timstamp: timestamp fra kamera
Output:
x̂: estimert nåværende posisjon.
c = 0.01;
while 1 do
    if Ny kamerastreng tilgjengelig then
        // Oppdater x̂:
        forrige_t = t;
        t = les klokke;
        x̂ = x̂ + v̂ * (t − forrige_t);
        // lagre forrige posisjon og timestamp, og les nye:
        forrige_x_r = x_r;
        x_r = les posisjon fra kamerastreng;
        forrige_timestamp = timestamp;
        timestamp = les timestamp fra kamerastreng;
        t̂_b = estimer bildetidspunkt(Algoritme 1);
        // Estimer fart:
        Δtimestamp = timestamp - forrige_timestamp;
        v̂ = (x_r-forrige_x_r)/Δtimestamp;
        // Estimer nåværende posisjon basert på nytt sample:
        x_n = x_r + v̂ · (t − t̂_b);
        // Beregn feilen i x̂:
        ê = x̂ − x_n;
        v̂' = v̂ − c · ê;
    end
    // Beregn ny posisjon:
    forrige_t = t;
    t = les klokke;
    x̂(i) = x̂ + v̂(t − forrige_t);
end
```

Algoritme 3: Prediksjon med tilbakekobling.

Noen eksempelbilder av resultat med denne metoden ses i Figur 66 og 67. Man ser at man har byttet vekk den hakkete sagtannsformen med noe mer avvik i ulineære områder (der hvor posisjonen akselerer).



Figur 66: Plot av prediksjon med tilbakekoblet utgang.



Figur 67: Med denne metoden får man en glattere kurve, i bytte mot noe mer avvik i bratte svinger.

6.6 Simulering av `ConveyorTracker`

For at `ConveyorTracker` klassen skal starte trackingen, må den motta et synksignal som sier fra om at objektet er på et bestemt sted. Dette gjøres ved å ha en boolsk variabel, `kameraSynk`, som settes høy når objektet er innenfor en bestemt intervall langs x-aksen i bildet. Deretter beregner `ConveyorTracker` objektets posisjon utifra avstanden objektet har beveget seg siden synk slo inn. Ved testing i Matlab emuleres denne oppførselen med en boolsk variabel, `connected` som settes til `true` når objektet befinner seg i synk-intervallet, og `false` når objektet har beveget seg en maksimal avstand. Synk-intervallet er plassert slik at hardware-synken og "kamerasynken" slår inn samtidig, slik at kurvene blir enkle å sammenlikne. I testene i det neste kapitellet brukes et synkintervall ved $50mm < \hat{x} < 65mm$. Et kobbermerke er festet på conveyoren, slik at denne slår ut samtidig som at posisjonen til objektet kommer inn i synkintervallet. Når synk slår inn, begynner `ConveyorTracker` sin posisjon (kaller denne for `ct`) å oppdatere seg slik at verdien er lik endring i posisjon siden synk slo inn. Både enkodertrackeren og den simulerte Matlabtrackeren settes til å tracke helt til `ct` har verdi større enn 650, deretter avsluttes trackingen, og `ct` blir satt til 0.

```
Output: ct: simulert ConveyorTracker posisjon.
bool connected = false;
ct_max = 650;
while 1 do
    if Ny kamerastreng tilgjengelig then
        Gjør nødvendige beregninger for den aktuelle prediksjonsalgoritmen;
    end
    forrige_x̂ = x̂;
    x̂ = estimer posisjon;
    if 50<x̂<65 then
        connected = true;
    end
    if connected then
        ct = ct + x̂− forrige_x̂;
    else
        ct = 0;
    end
    if ct > ct_max then
        connected = false;
    end
end
```

Algoritme 4: Simulering av conveyortracker.

6.7 Testing av algoritmene

Begge algoritmene har blitt testet i Matlab. Rådataen fra kameraet, samt enkodertrackerens posisjon, logges i RobView og eksporteres til Matlab i form av en kommaseparert tekstfil. RobView har en noe varierende samplingrate (10-40 millisekunder periodetid), noe som gjør at rådataen fra kamerasignalet kan leses helt feilfritt (ettersom den har lavere rate). Men enkodertrackeren, som oppdateres hvert 16. millisekund, blir noe deformert i loggen. Dette gjør at sammenlikningene ikke er fullstendig representative for det faktiske systemet, og avviket mellom kameraet og enkoder er antakeligvis *mindre* enn det simuleringene viser på grunn av deformeringen av enkodersignalet[9].

I tillegg til simulering i Matlab, har algoritmene blitt implementert i IPS. Resultatene på både Matlab-simuleringene og implementeringen i IPS kan ses i neste seksjon.

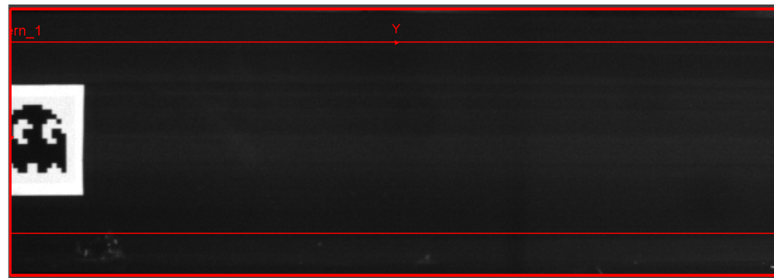
[9]Dersom det gjøres litt kvalifisert(?) gjetning.

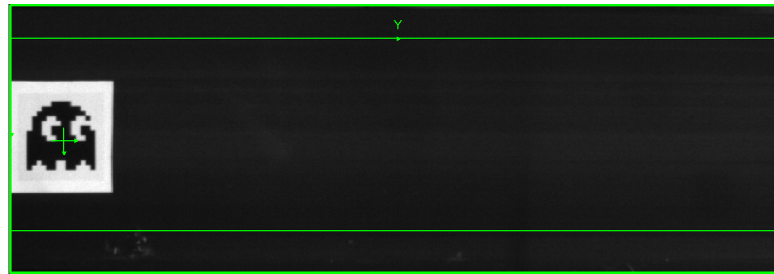
7 Resultater

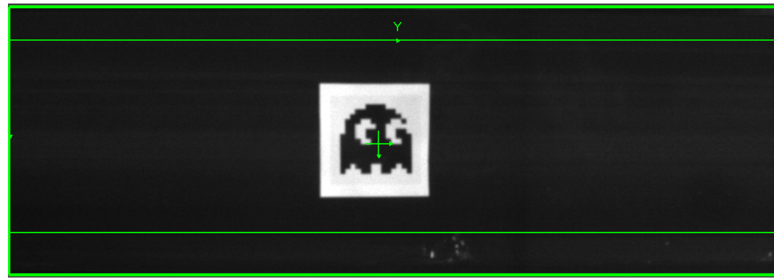
I denne seksjonen ses det på resultatene fra de forskjellige testene som har blitt gjort.

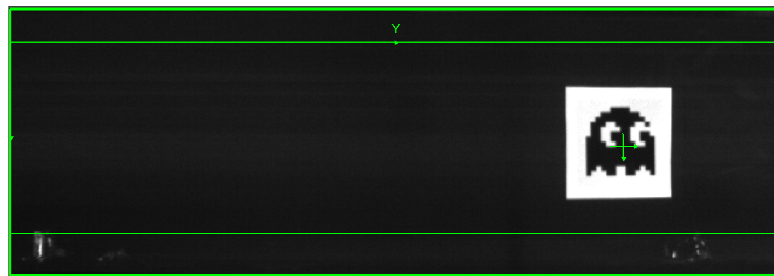
7.1 Resultat på Matlabtester

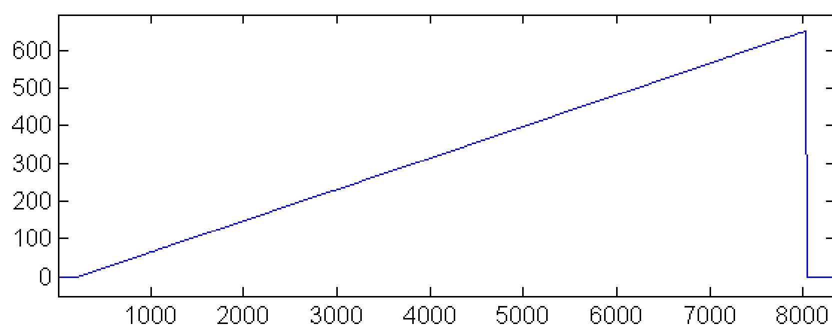
Begge algoritmene som er beskrevet i seksjon 6 har blitt testet med rådata fra kameraet. Signalet fra `ConveyorTracker` har blitt logget samtidig for å kunne gjøre sammenlikninger. I de neste avsnittene gås det gjennom resultatene for de forskjellige prediksjonsalgoritmene.









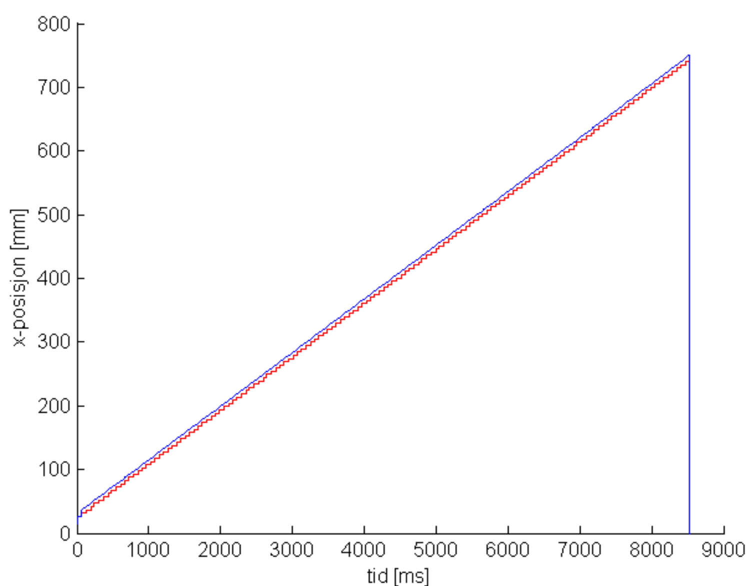

600
500
400
300
200
100
0
1000
2000
3000
4000
5000
6000
7000
8000

7.2 Konstant hastighet mellom sampler

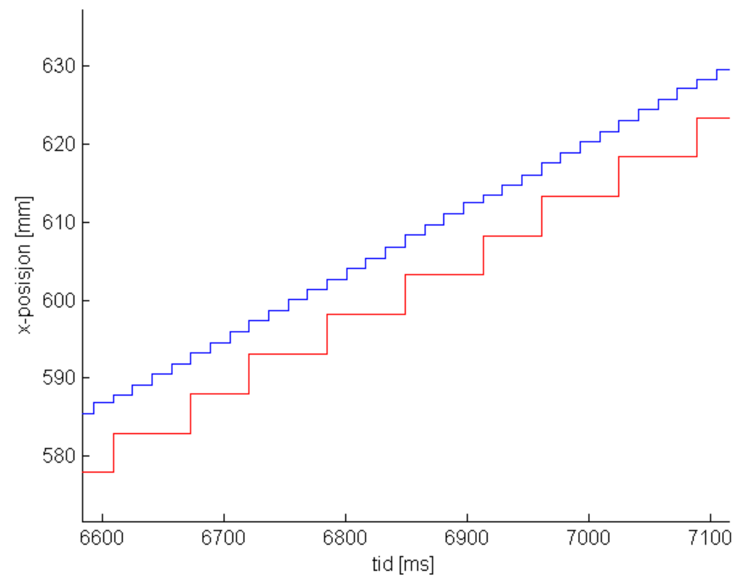
Vi ser først på prediktert posisjon uten å sammenlikne med enkodertrackeren. Figur 68 - 72 viser plot av prediktert posisjon med Algoritme 2.

I alle figurene bortsett fra Figur 71 er initielt antatt forsinkelse satt til 50 ms. I Figur 71 er den satt til 200 ms. Da blir signalet forflyttet tilbake i tid, slik at prediksjonen ligger lengre foran avlest signal (200 ms. mot 50 ms.).

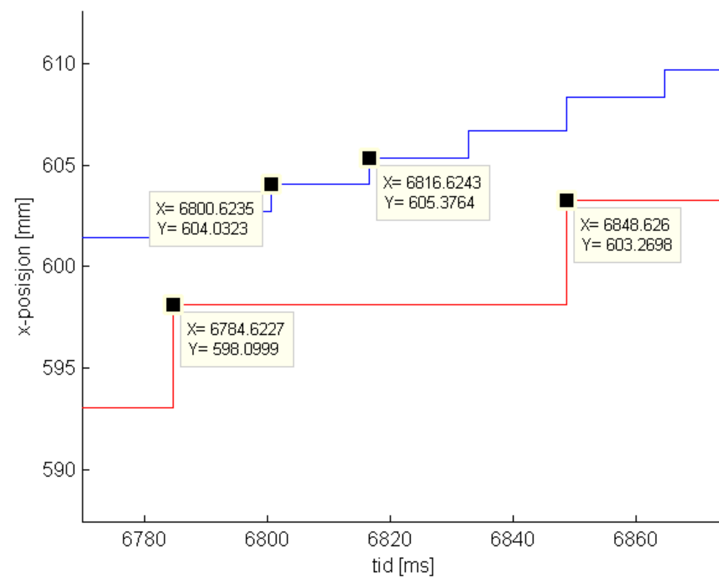
I Figur 72 har det blitt benyttet 1 millisekund samplingsintervall i stedet for 16 millisekunder. Da kommer uregelmessigheter noe bedre fram, ved at man ser noe mer hakkete transisjoner ved samplingstidspunktene fra kameraet, i situasjoner der prediksjonen bommer noe. Den grønne kurven i disse plottene viser rådataen plotet mot estimert bildetagningstidspunkt (som beregnes idét rådataen mottas). Som man kan se på kurven treffer den blå (predikerte) kurven de grønne samplene nokså bra, som er hensikten, ettersom prediksjonen (blå) forsøker å treffe de estimerte bildetagningstidspunktene (grønn).



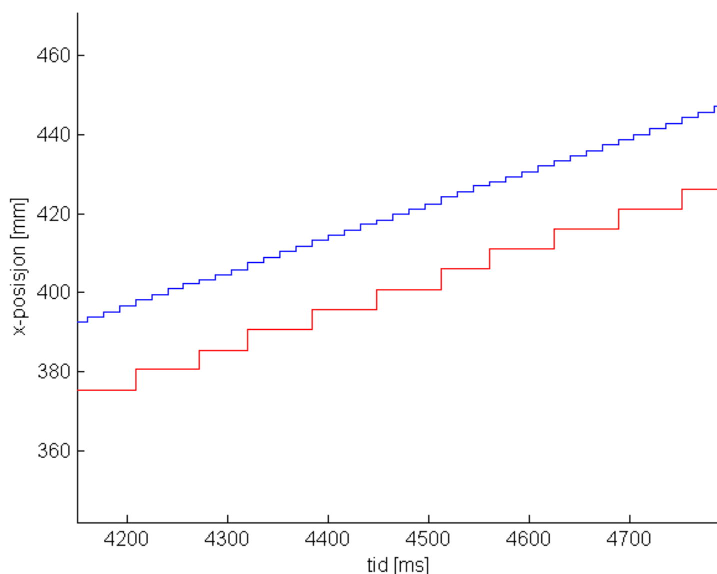
Figur 68: Blå: x-posisjon plottet mot estimert bildetagningstidspunkt. Rød: Den samme posisjonsdataen plotet mot samplingstidspunkt (I Matlab).



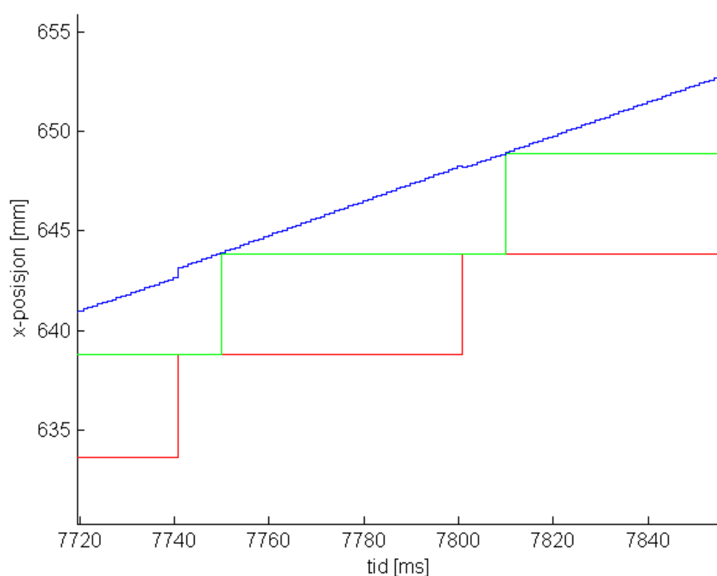
Figur 69: Blå: x-posisjon plottet mot estimert bildetagningstidspunkt. Rød: Den samme posisjonsdataen plotet mot samplingstidspunkt (I Matlab).



Figur 70: Ser at periodetiden til det ekstrapolerte signalet er 16 ms, som ønsket.



Figur 71: Ved å øke det initielt antatt forsinkelse, oppnår man at det predikterte signalet ligger lengre foran det avleste. Her er det brukt 200 ms. initielt antatt forsinkelse. Da oppnås det at prediksjonen når en gitt posisjon $\approx$ 200 ms før det avleste signalet (når farten er konstant).



Figur 72: Det grønne signalet er rådataen plottet mot estimert bildetagningstidspunkt. Det predikerte signalet forsøker å treffe disse samplene.

7.2.1 Sammenlikning med enkodertrackeren *uten* lineær avvikskorreksjon

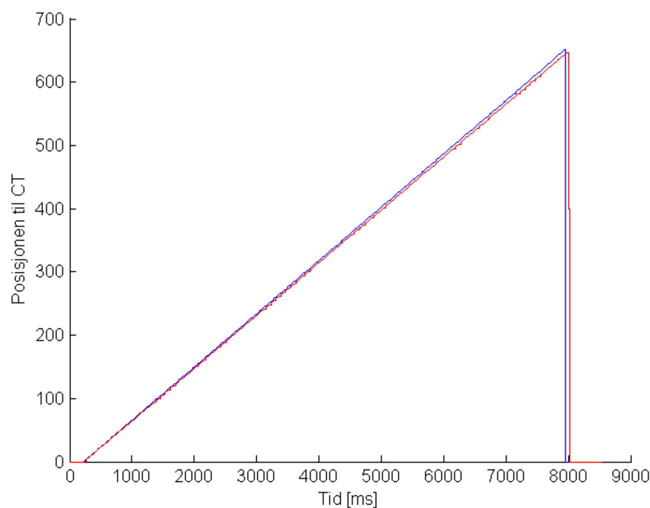
I dette avsnittet vises resultatet av `CT`-simulering i Matlab, basert på Algoritme 4, Simulering av conveyortracker.

Figur 73 viser den simulerte kameratrackingen (blå), og den loggede enkodertrackingen. Figur 74 og 75 viser forstørrelser av det samme plottet i start- og sluttfasen. De to kurvene starter å tracke nokså samtidig, men man ser at kameratrackingen har et voksende avvik. Kameratrackingen ser ut til å beregne en noe hurtigere stigning enn enkoderen. På det meste er avviket $\approx 8mm$, helt på slutten av plottet.

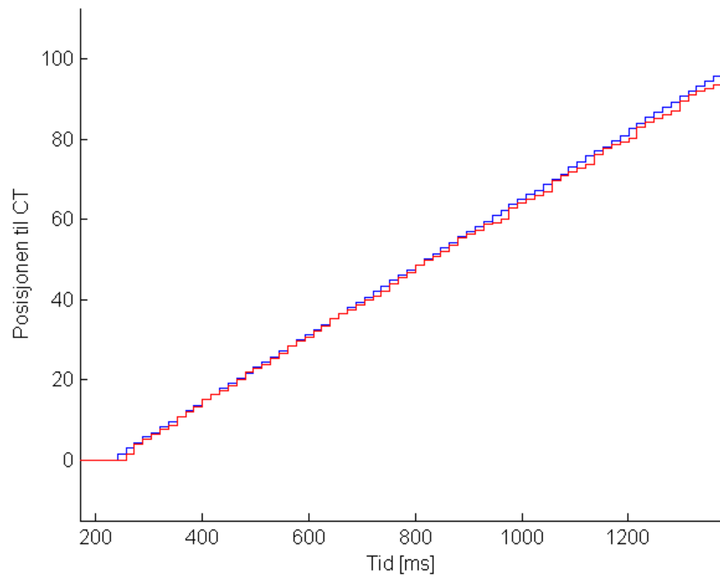
Avviket $e = kamera_ct - enkoder_ct$ er vist i Figur 76. Det store hoppet på slutten av kurven kommer av at de to ct-variablene når maksdistansen på forskjellige tidspunkt. Dette hoppet er derfor ikke spesielt interessant, så det klippes vekk. Et forstørret plot av avviket er vist i Figur 77. Her har området før trackingen startet, samt området etter at ct når over maksgrensen blitt kuttet vekk, slik at man bare sitter igjen med området der både `ct_kamera` og `ct_enkoder` tracker. Den røde streken i plottet representerer $e = 0$.

Noen egenskaper ved avviket:

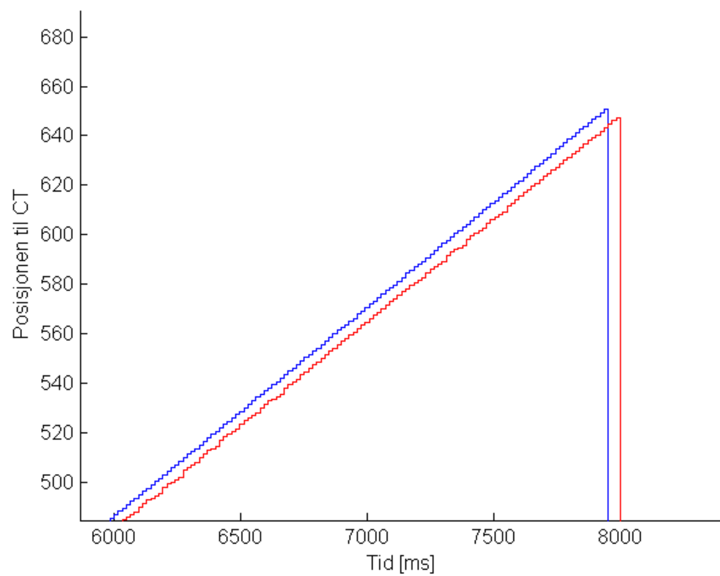
$\hat{\sigma}$	$1.9118mm$
$\hat{\mu}$	$-3.5513mm$
største avvik	7.7543



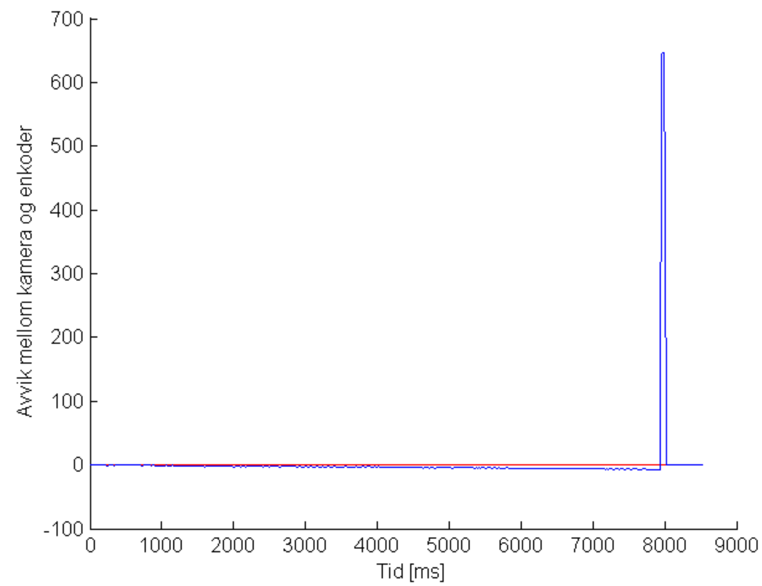
Figur 73: Blå: Simulert conveyor-tracker fra kameraprediksjon. Rød: Logget signal fra enkodertrackeren.



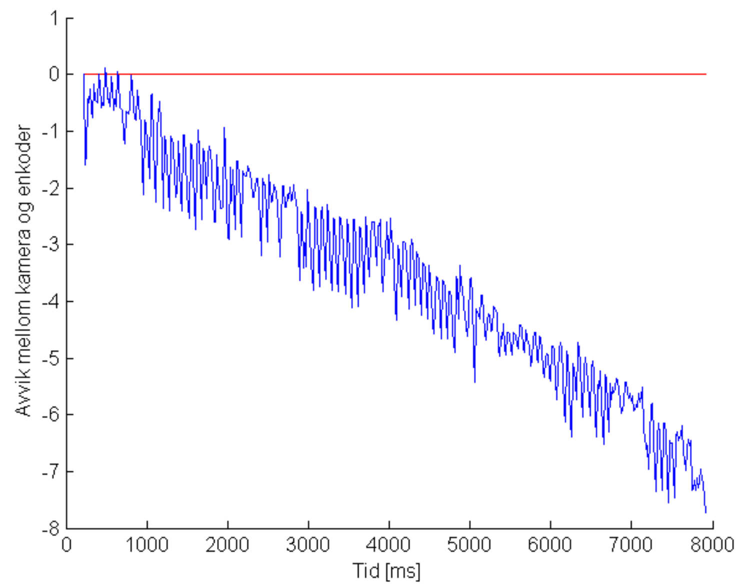
Figur 74: Blå: Simulert conveyor-tracker fra kameraprediksjon. Rød: Logget signal fra enkodertrackeren.



Figur 75: Blå: Simulert conveyor-tracker fra kameraprediksjon. Rød: Logget signal fra enkodertrackeren.



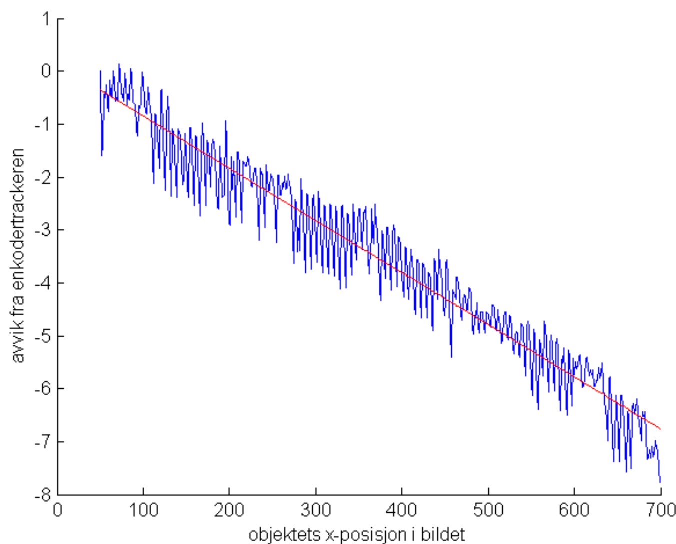
Figur 76: Blå: Feilen $e = ct_kamera - ct_e nkoder$ som funksjon av tiden. Den røde linjen representerer $e = 0$.



Figur 77: Blå: Feilen $e = ct_kamera - ct_e nkoder$ som funksjon av tiden. Den røde linjen representerer $e = 0$.

7.2.2 Sammenlikning med enkodertrackeren *med* lineær avvikskorreksjon

Anvender lineær avvikskorreksjon som beskrevet i avsnitt 6.4, og finner at parameterene $a = -0.0099$ og $b = 0.1369$. Denne linjen er plottet i rødt i Figur 78, sammen med avviket som funksjon av objektets posisjon.

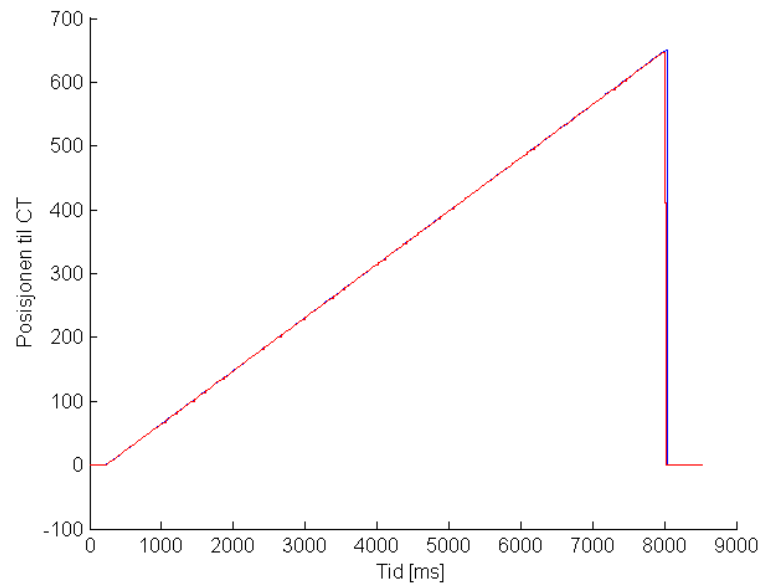


Figur 78: Estimering av lineært avvik som funksjon av objektets posisjon i bildet.

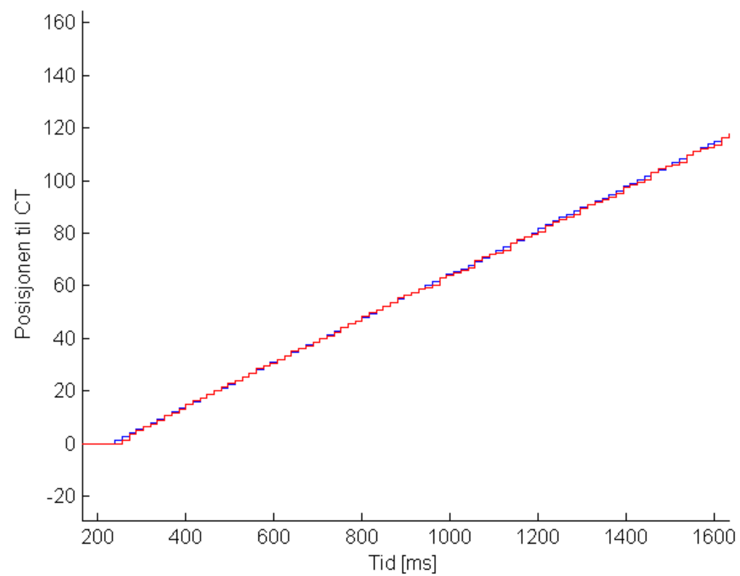
Redigerer koden slik at de leste samplene multipliseres med $(1 - a) = (1 - 0.0099)$, som beskrevet i avsnitt 6.4. Når simuleringen kjøres på nytt med den oppdaterte algoritmen for lineær feilkooreksjon, oppnås det at den lineære feilen så og si forsvinner fra trackingsignalet. Ved å estimere lineært avvik på samme måte som tidligere ved hjelp av LS-metoden (på det nye signalet som har anvendt lineær avvikskorreksjon), fås det ut at parameteren $a = 0.0000$ fra Matlab, dvs det lineære avviket er tilnærmet lik null med 5 signifikante tall. Figur 79 - 81 viser plot av simulert `ct` basert på kamera (i blått), og logget enkodertracker (i rødt). Figur 82 viser det nye estimerte avviket som funksjon av objektets posisjon.

Noen av dette avvikets egenskaper:

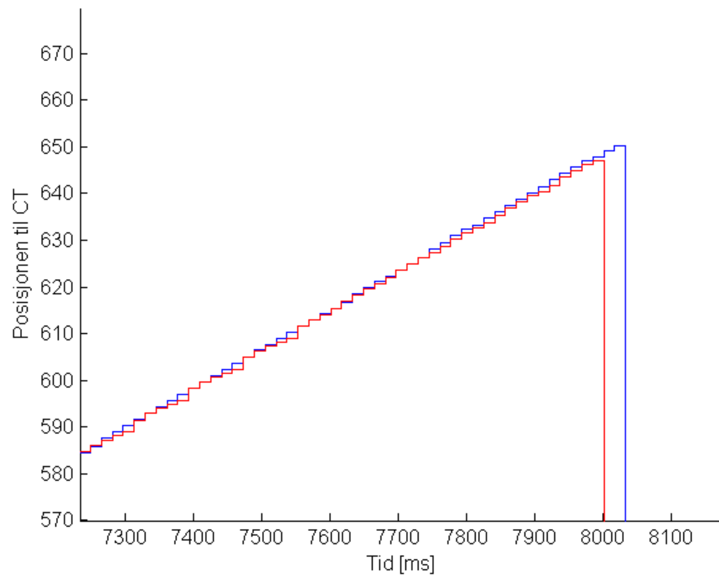
$\hat{\sigma}$	$0.4832mm$
$\hat{\mu}$	$0.0115mm$
største avvik	1.2305



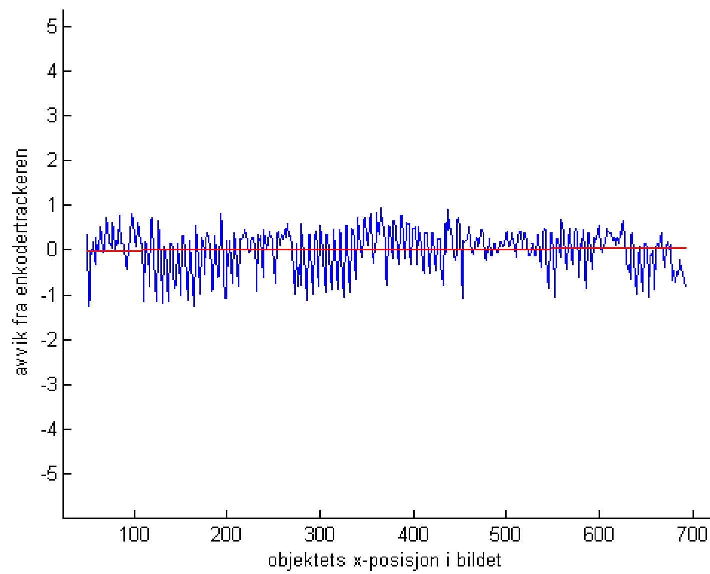
Figur 79: Blå: ct-signalet beregnet fra kamera med lineær feilkorreksjon anvendt. Rød: Logget enkodertracking.



Figur 80: Blå: ct-signalet beregnet fra kamera med lineær feilkorreksjon anvendt. Rød: Logget enkodertracking.



Figur 81: Blå: ct-signalet beregnet fra kamera med lineær feilkorreksjon anvendt. Rød: Logget enkodertracking.



Figur 82: Feilen $e = ct_enkoder - ct_kamera$ (i millimeter) som funksjon av tiden etter korreksjon for lineært avvik. Rød linje representerer $e = 0$.

7.3 Konstant fart med tilbakekobling

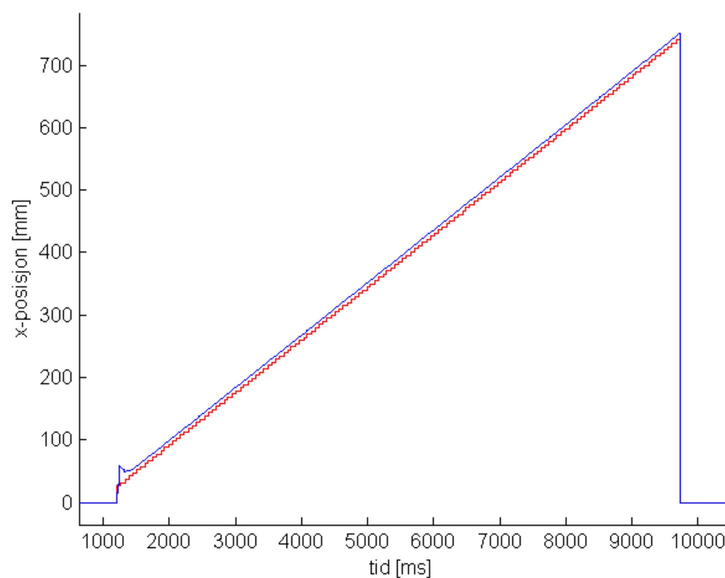
Figur 83 - 85 viser plot av prediksjonen gitt av Algoritme 3 (i blått) og rådata plotet mot avlesningstidspunkt (i rødt). Det første blikkfanget i plottet er kanskje den bråe økningen idét objektet kommer i syne for kameraet. Som plottet viser, stiger kurven forbi synkintervallet (dvs. over 50) før den skal, så dette må fikses. Den første tanken som faller inn er kanskje å lavpassfiltrere farten. Men det viser seg å føre til kluss på grunn av tilbakekoblingen, da det oppstår svingninger på utgangen, se Figur 86.

En enklere løsning ble valgt; ganske enkelt at dersom forrige fart var null, halvér denne farten:

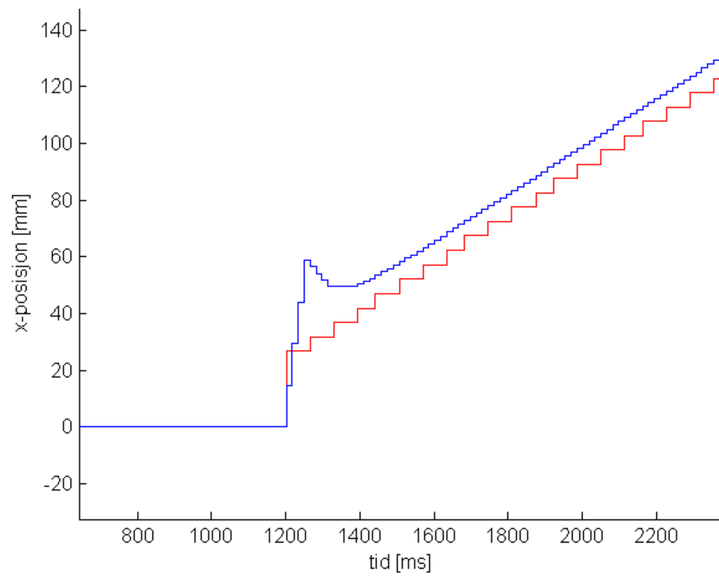
```
1 forrige_v = v;
2 v   = beregn fart gitt av algoritme 3
3 if forrige_v == 0
4     v = v*0.5;
5 end
```

Resultatet etter denne oppdateringen kan ses i Figur 87.

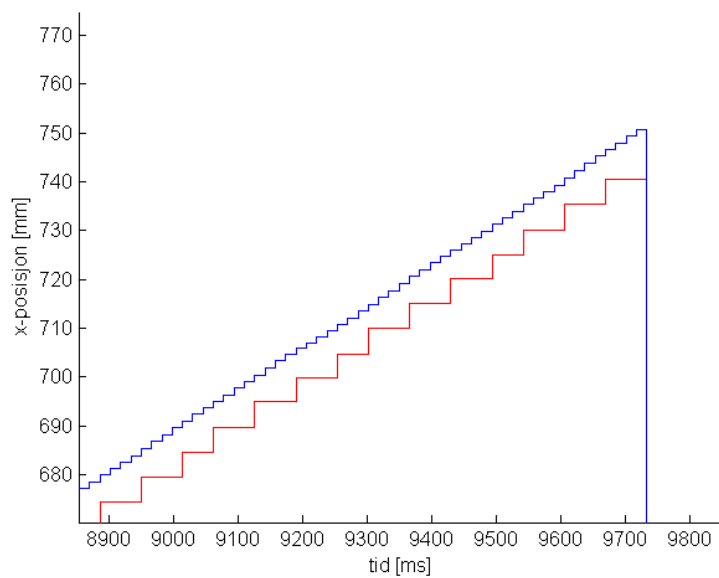
Ved å endre Matlab-simuleringen til å oppdatere hvert millisekund (istedet for hvert 16.) ser man tydelig at denne algoritmen får en glattere og penere kurve enn algoritmen *uten* tilbakekobling. Se Figur 88 og 89. I neste avsnitt sammenliknes resultatet fra denne algoritmen med enkodertrackeren.



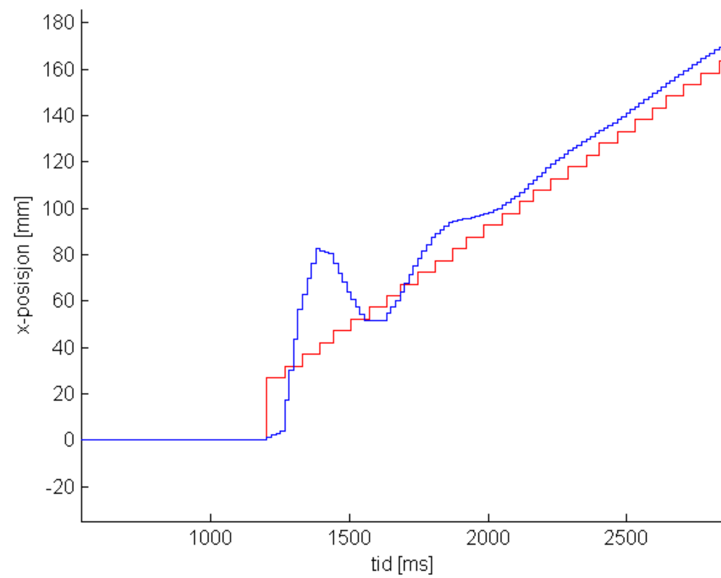
Figur 83: Blå: Estimert posisjon basert på Algoritme 3. Rød: Rådata fra kameraet.



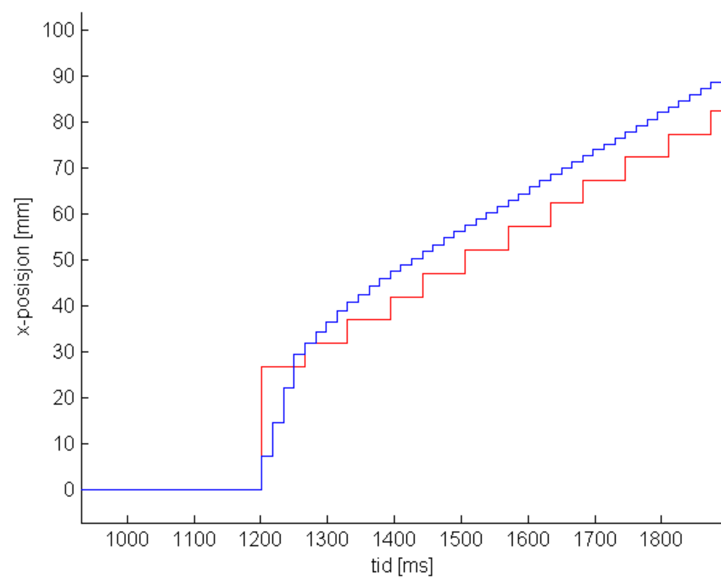
Figur 84: Estimatet gir brå start idét objektet kommer i synet for kameraet.



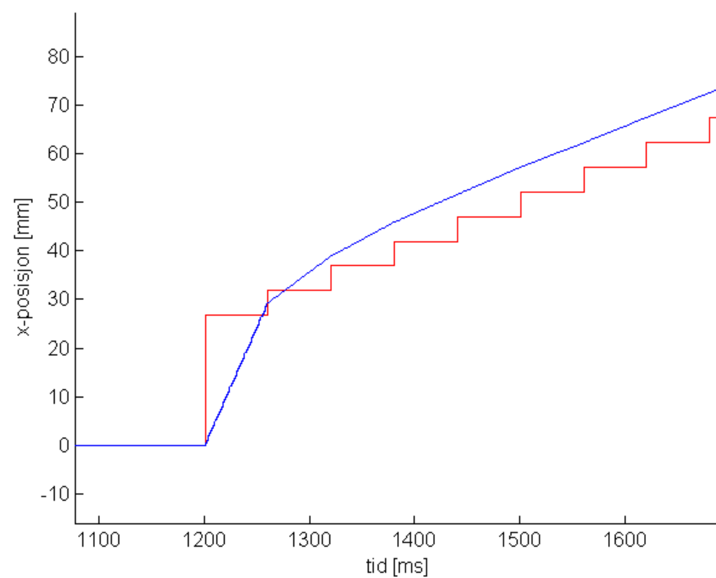
Figur 85: Forstørrelse av området før objektet kommer ut av synsfeltet til kameraet.



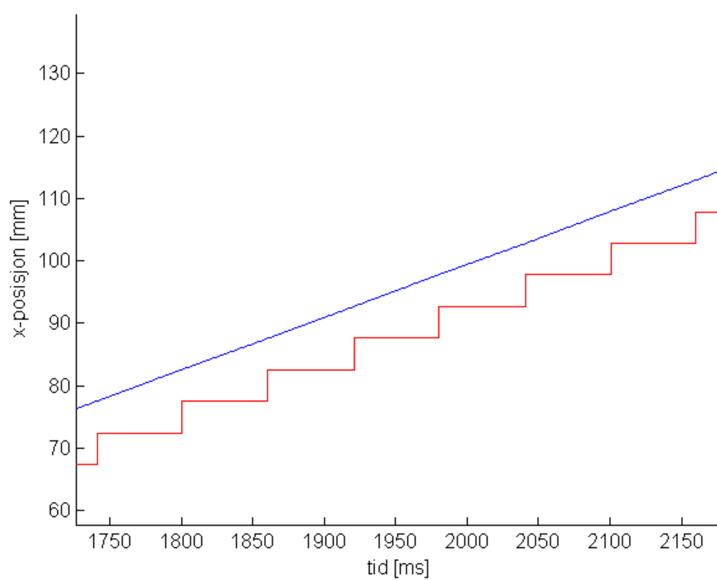
Figur 86: Lavpassfiltrering av farten ble droppet ettersom det førte til en del kluss på tilbakekoblingen.



Figur 87: Det ble istedet valgt å dempe den første fartsutregningen, som gir et nokså pent resultat.



Figur 88: Ved å øke oppdateringsraten til 1 kHz (hvert millisekund), ser man at denne algoritmen gir en penere kurve enn den som ikke tar i bruk tilbakekobling (ingen brå hakk).



Figur 89: Ved å øke oppdateringsraten til 1 kHz (hvert millisekund), ser man at denne algoritmen gir en penere kurve enn den som ikke tar i bruk tilbakekobling (ingen brå hakk).

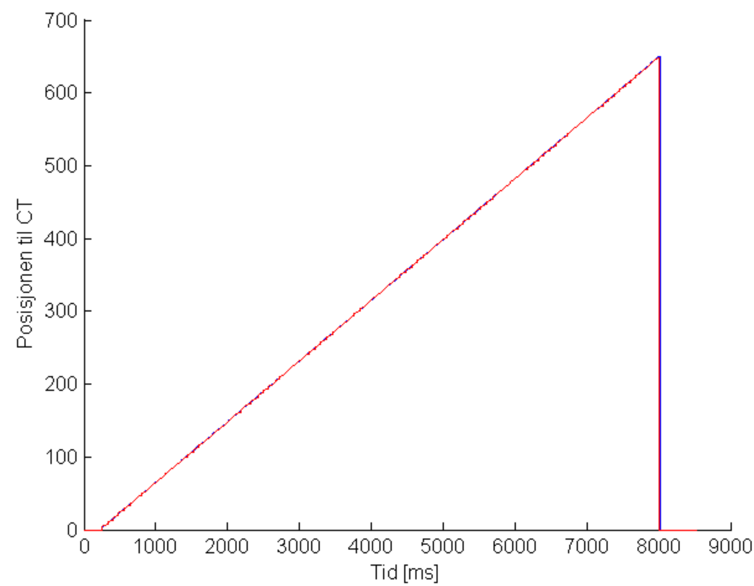
7.3.1 Sammenlikning med enkodertrackeren

Korreksjon for det lineært økende avviket er anvendt også her. Ettersom $a = -0.0099$ er kjent, og denne ulineariteten er en egenskap ved kameraoppsettet (enten at kameraet står skjevt, eller ikke optimal kalibrering), kan den samme verdien brukes her også.

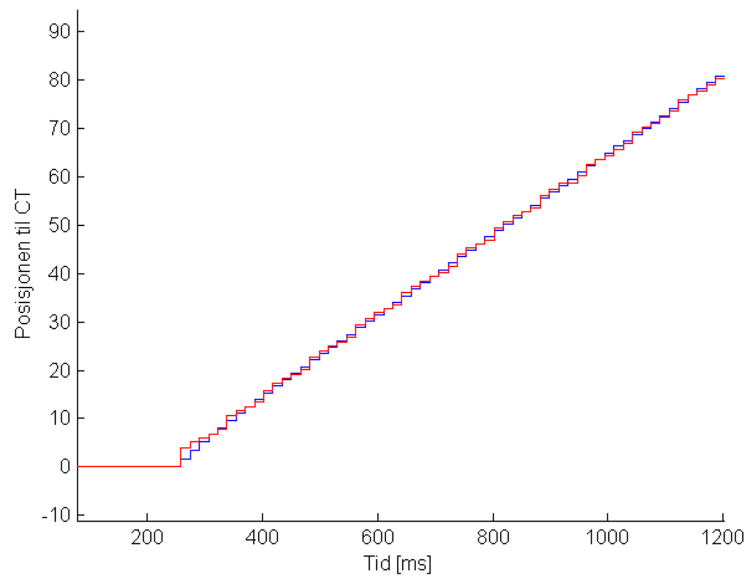
Egenskaper ved avviket:

$\hat{\sigma}$	$0.5617mm$
$\hat{\mu}$	$-0.0498mm$
største avvik	2.1421

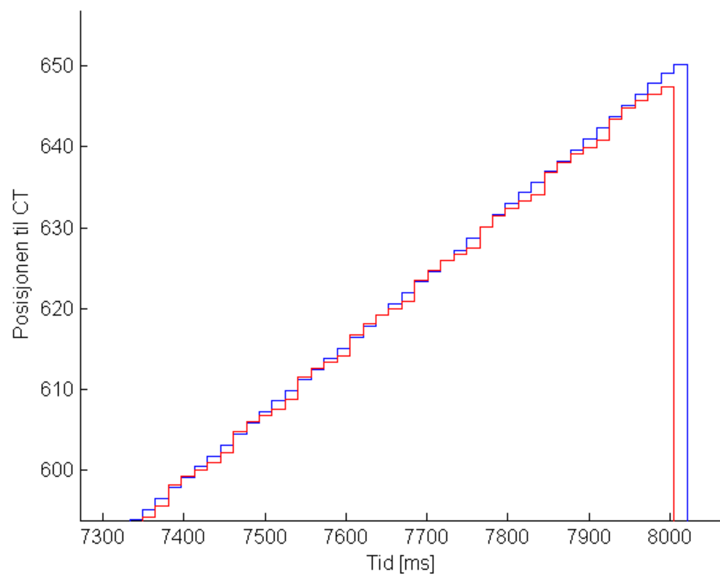
Ser at man får en penere kurve på bekostning av marginalt dårligere følgeegenskaper.



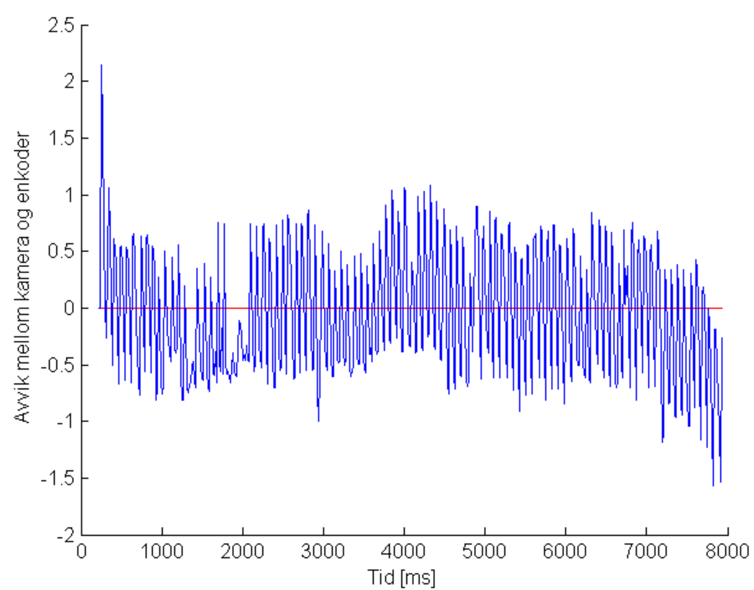
Figur 90: Hele plottet. Blå: ct-signalet beregnet fra kamera. Rød: Logget enkodertracking.



Figur 91: Starten av trackingen. Blå: ct-signalet beregnet fra kamera. Rød: Logget enkodertracking.



Figur 92: Slutten av trackingen. Blå: ct-signalet beregnet fra kamera. Rød: Logget enkodertracking.



Figur 93: Avvik

7.4 Resultater i praksis på PIB

I de følgende avsnittene vises resultatene på implementering i IPS på PIB-kortet. Plottene viser sammenlikning mellom conveyortracking med kamera og enkoder.

Plottene er logget i RobView, som henter signalene direkte fra IPS. Ettersom RobView ikke holder konstant samplingrate, og sampler med mellom 20-40 ms intervaller, er disse signalene desverre ikke fullstendige representasjoner av signalene i IPS. Samplingraten er lavere enn raten på IPS-signalene, og signalene leses ikke nødvendigvis i de riktige samme iterasjonene i IPS. Derfor burde resultatene i de neste avsnittene kun ses på som en pekepinn på om ting fungerer som forventet, basert på simuleringene, enn selvstendige resultater i seg selv.

I avsnitt 7.5 vises resultater ved algoritmen for antatt konstant fart mellom sampler (Algoritme 2).
I avsnitt 7.6 vises resultater ved algoritmen for antatt konstant fart mellom sampler (Algoritme 3).

7.5 Konstant fart mellom sampler

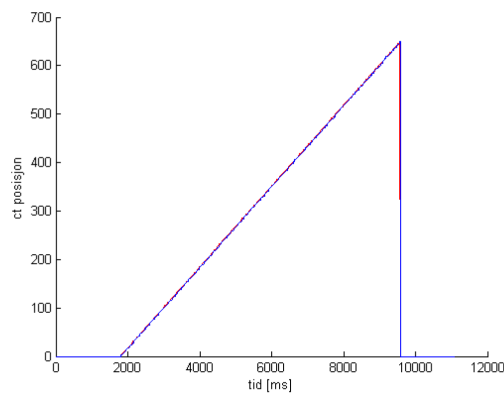
7.5.1 Konstant fart på conveyoren

Egenskaper ved avviket:

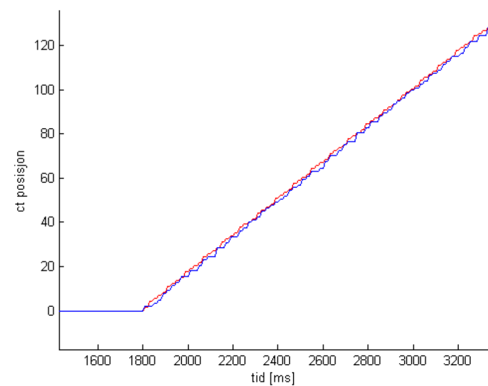
$\hat{\sigma}$	$0.86mm$
$\hat{\mu}$	$1.55mm$
største avvik	$3.52mm$

Dersom avviket korrigerers slik at $\mu = 0$ $(e' = e - \mu)$

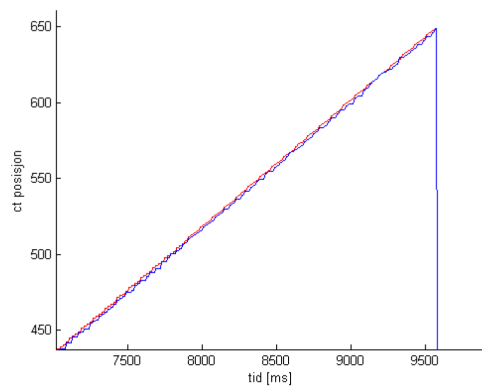
største avvik	$1.97mm$



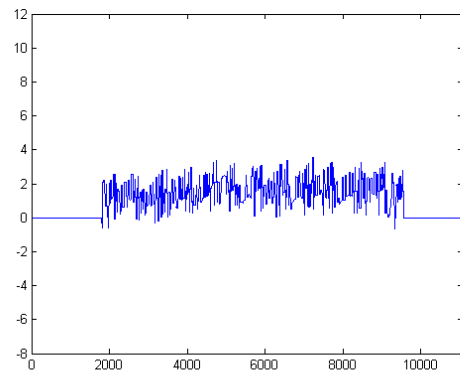
Figur 94: Hele plottet. Blå: Kameratrackeren. Rød: Enkodertrackeren.



Figur 95: Starten av plottet. Blå: Kameratrackeren. Rød: Enkodertrackeren.



Figur 96: Slutten av plottet. Blå: Kameratrackeren. Rød: Enkodertrackeren.



Figur 97: Avvik.

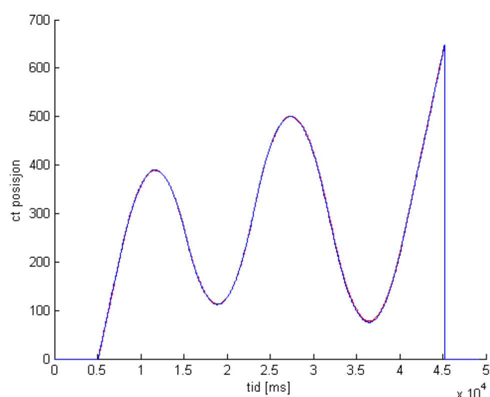
7.5.2 Varierende fart på conveyoren

Egenskaper ved avviket:

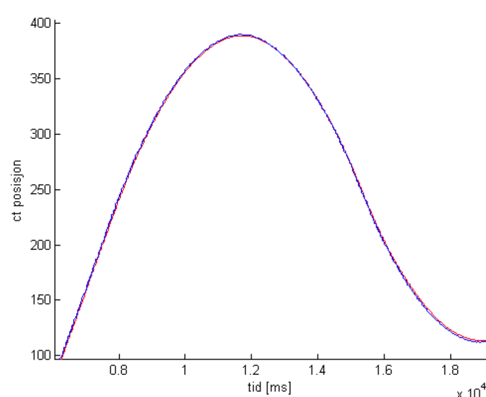
$\hat{\sigma}$	$1.89mm$
$\hat{\mu}$	$1.20mm$
største avvik	$5.81mm$

Dersom avviket korrigerers slik at $\mu = 0$ ($e' = e - \mu$)

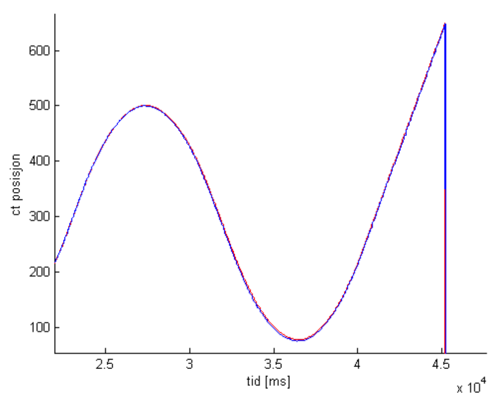
største avvik	4.60



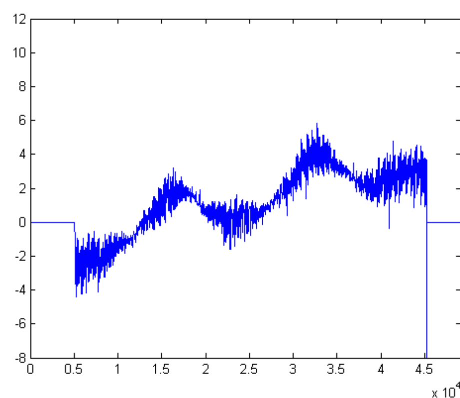
Figur 98: Hele plottet. Blå: Kameratrackeren. Rød: Enkodertrackeren.



Figur 99: Starten av plottet. Blå: Kameratrackeren. Rød: Enkodertrackeren.



Figur 100: Slutten av plottet. Blå: Kameratrackeren. Rød: Enkodertrackeren.



Figur 101: Avvik.

7.6 Konstant fart med tilbakekobling

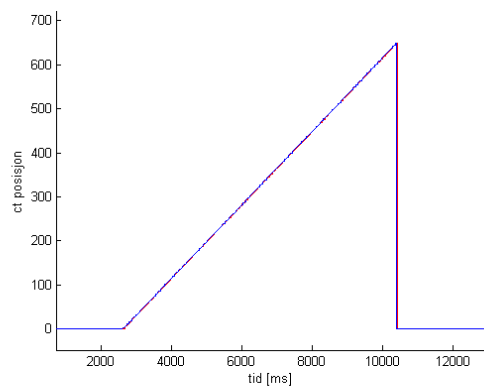
7.6.1 Konstant fart på conveyoren

Egenskaper ved avviket:

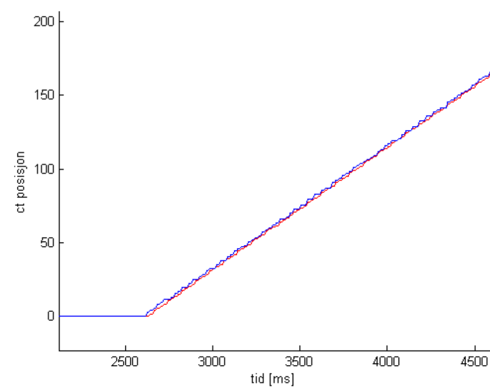
$\hat{\sigma}$	$0.81mm$
$\hat{\mu}$	$-1.14mm$
største avvik	$3.47mm$

Dersom avviket korrigerers slik at $\mu = 0$ $(e' = e - \mu)$

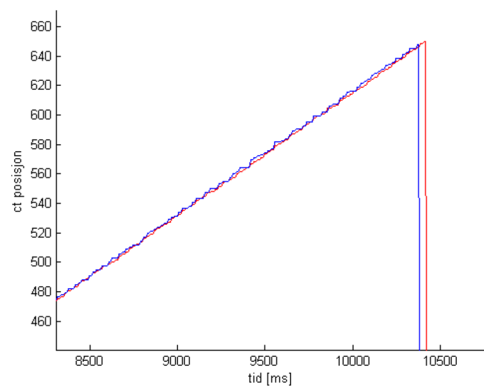
største avvik	3.69



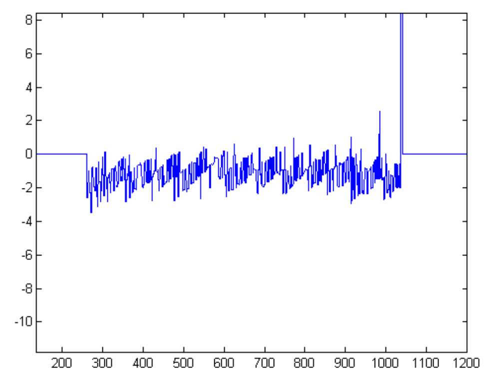
Figur 102: Hele plottet. Blå: Kameratrackeren. Rød: Enkodertrackeren.



Figur 103: Starten av plottet. Blå: Kameratrackeren. Rød: Enkodertrackeren.



Figur 104: Slutten av plottet. Blå: Kameratrackeren. Rød: Enkodertrackeren.



Figur 105: Avvik.

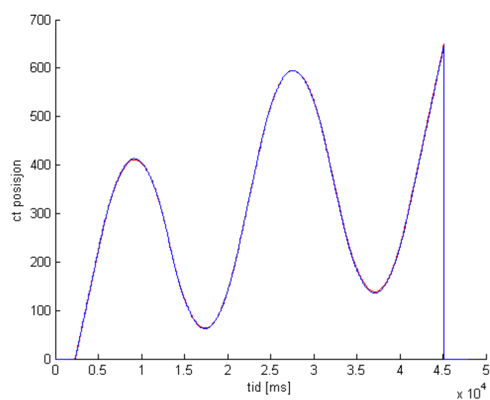
7.6.2 Varierende fart på conveyoren

Egenskaper ved avviket:

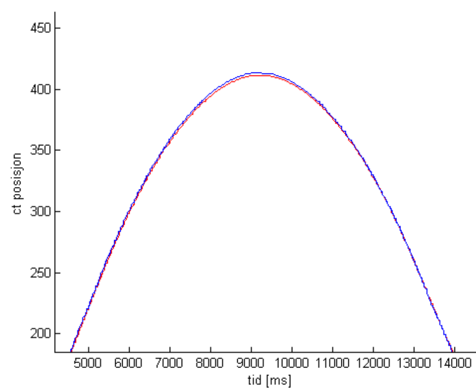
$\hat{\sigma}$	$1.84mm$
$\hat{\mu}$	$0.96mm$
største avvik	$5.80mm$

Dersom avviket korrigerers slik at $\mu = 0$ $(e' = e - \mu)$

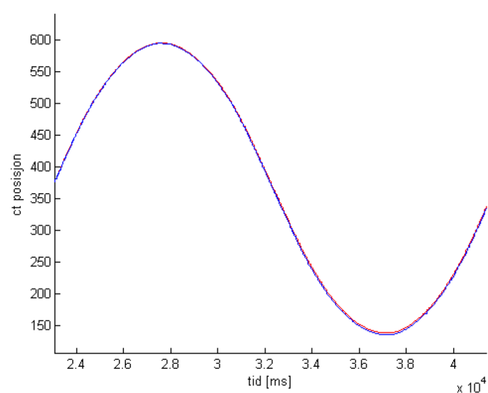
største avvik	4.81



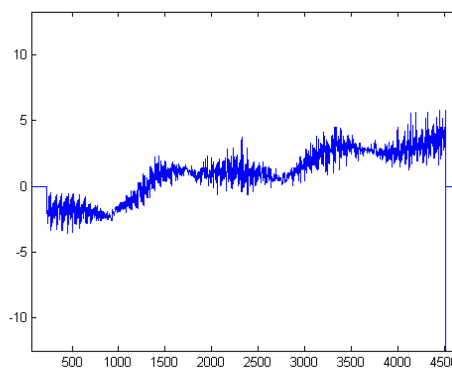
Figur 106: Hele plottet. Blå: Kameratrackeren. Rød: Enkodertrackeren.



Figur 107: Starten av plottet. Blå: Kameratrackeren. Rød: Enkodertrackeren.



Figur 108: Midten av plottet. Blå: Kameratrackeren. Rød: Enkodertrackeren.



Figur 109: Avvik.

8 Konklusjon og videre arbeid

Utifra sammenlikningene av enkodertrackeren med de endelige resultatene av kameratracking, ser det ut til at kameraløsningen er i full stand til å erstatte enkoderen til tracking når conveyoren holder konstant fart, i oppsettet som har blitt testet. Med standardavvik på rundt 0.5 millimeter mellom kamera- og enkodersignal i simuleringer, viser kameraet overaskende bra prestasjon.

Sammenlikningene fra implementasjon i det virkelige systemet burde vektlegges mindre ettersom signalene som logges ikke er fulstendig representative for de faktiske signalene i systemet. Dette fordi de samples med varierende rate og ikke nødvendigvis i samme iterasjon, og dermed inneholder mere støy. Men også disse signalene presterer tilsynelatende bra. Begge prediksjonsalgoritemen gir rundt 0.85 millimeter standardavvik (på de loggede, deformerte signalene) fra enkodersignalet med konstant fart på conveyoren, og litt under 1.9 millimeter standardavvik med varierende fart.

De to algoritmene presterer såpass likt om man kun ser på det totale avviket, at det ikke er lett å fatte en beslutning for hvilken som er det beste valget. Men dersom man er opptatt av at signalene skal ha en viss estetisk kvalitet over seg, er Algoritme 3, som tar i bruk et tilbakekoblet signal, å foretrekke, ettersom kurven ganske enkelt *ser* penere ut[10].

Denne oppgaven har begrenset seg til tracking i én dimensjon, for å tilpasse seg til det eksisterende systemet. Men signalene for tracking på tvers av conveyoren og orientering ligger klare til å anvendes, dersom `ConveyorTracker` utvides til å håndtere flere dimensjoner.

En svakhet er at det ikke har blitt eksperimentert med forskjellige linser. Med en linse med kortere brennvidde, hadde man kunnet få bredere synsfelt, og større arbeidsdistanse (avstand fra conveyoren til kameraet). I videre arbeid må dette tas bedre hensyn til.

Av å se på kurvene, og på roboten mens trackingen pågår, er det lite tvil om at kamerasystemet gir nøyaktig nok tracking til lakkeringsformål. Men det er fremdeles utfordringer som ikke er sett på i denne oppgaven før kamerasystemet kan tas i bruk. Denne oppgaven har kun sett på tracking av ett enkelt objekt om gangen. For å erstatte dagens system, må kamerasystemet være i stand til å håndtere flere objekter. Et annet problem er det begrensede synsfeltet til kameraet. Ettersom rekkevidden til trackingen er begrenset av synsfeltet, er ikke ett kamera i stand til å håndtere tracking på en lang conveyor, i motsetning til enkoderen som teoretisk kunne håndtert en uendelig lang conveyor. Dette kan løses med flere kameraer, som eventuelt kan samarbeide. Det siste er at man i denne oppgaven har sett bort i fra selve lakkeringsprossessen, som fører til utfordringer når objektet som trackes deformeres i bildet, som konsekvens av at lakk sprayes på. Alle disse er problemer som må tas stilling til i et eventuelt videre arbeid, dersom kameratracking

[10]Man unngår sagtannsform på signalet

skal tas i bruk som erstatning for enkoderen.

Referanser

[1] Camera techniques (bilde). www.grelf.net/camera_tech.html.

[2] Linux sockets tutorial. http://www.linuxhowtos.org/C_C++/socket.htm.

[3] Programming with udp sockets. http://www.cs.rutgers.edu/ px-k/417/notes/sockets/ udp.html.

[4] Leah Bar, Nir Sochen, and Nahum Kiryati. Image deblurring in the presence of salt and-pepper noise. In *Scale Space and PDE Methods in Computer Vision*, pages 107–118. Springer, 2005.

[5] John Canny. A computational approach to edge detection. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, (6):679–698, 1986.

[6] Cognex. http://www.cognex.com/pattern-matching- technology.aspx.

[7] Cognex. *In-Sight Explorer Help.*

[8] Cognex. In-sight explorer training course, powerpoint presentations.

[9] Cognex. Patmax applications. Powerpoint presentation.

[10] Peter I Corke. Visual control of robot manipulators-a review. *Visual servoing*, 7:1–31, 1993.

[11] David A Forsyth and Jean Ponce. *Computer vision: a modern approach.* Prentice Hall Professional Technical Reference, 2002.

[12] Chris Harris and Mike Stephens. A combined corner and edge detector. In *Alvey vision conference*, volume 15, page 50. Manchester, UK, 1988.

[13] Finn Haugen. *Advanced Dynamics and Control.* TechTeach, 2010.

[14] Anand Singh Jalal and Vrijendra Singh. The state-of-the-art in visual object tracking. *Informatica (03505596)*, 36(3), 2012.

[15] Fredrik Kånge. Methods for real-time bin-picking using 2d vision, 2007.

[16] Lars-Ake Larzon, Mikael Degermark, and Stephen Pink. *UDP lite for real time multimedia applications.* Hewlett-Packard Laboratories, 1999.

[17] Silver Moon. Udp socket programming in winsock. http://www.binarytides.com/udp-socket-programming-in- winsock/.

[18] Randal C. Nelson. Finding line segments by stick growing. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 16(5):519–523, 1994.

[19] Nikolaos P Papanikolopoulos, Pradeep K Khosla, and Takeo Kanade. Visual tracking of a moving target by a camera mounted on a robot: A combination of control and vision. *Robotics and Automation, IEEE Transactions on*, 9(1):14–35, 1993.

[20] ABB Robotics. *Application Manual, Conveyor Tracking.*

[21] ABB Robotics. *Unit description for service IRC5P.*

[22] Erik Blaine Sudderth. *Graphical models for visual object recognition and tracking.* PhD thesis, Massachusetts Institute of Technology, 2006.

[23] Alexis Wilke. A c++ implementation of a udp client/server. http://linux.m2osw.com/c-implementation-udp- clientserver.

9 Forkortelser

ABB Asea Brown Boveri. Internasjonalt selskap med hovedkontor i Zürich

CT Conveyortracker

FPGA Field Programmable Gate Array

IPS Integrated Process System. distribuert system som kjører på mange av kortene i robotkabinettet.

ISE In-Sight Explorer. Cognex programvare til kameraet.

PIB Paint Interface Board. Et av kortene som tar seg av styring av forskjellige lakkeringsenheter.

A Appendiks

A.1 Kildekode

A.1.1 CamDevice (hpp/cpp)

```
1  #ifndef CamDeviceUdp_HPP
2  #define CamDeviceUdp_HPP
3
4  /****************************************************************
5
6  CamDevice.hpp opens up a UDP object that listens to the camera port ...
       specified in the
7  constructor of the CamDevice object. The Function ...
       readUdpAndCalcStuff() reads the text string
8  received from the camera, and gets the X,Y and orientation coordinates ...
       of the tracked object
9  in the image.
10
11 It then calculates the speed, position and sync signals for the ...
       conveyor tracker.
12 These three, in turn, are used by the CamEncoder class and the CamSync ...
       class to emulate a
13 traditional encoder and sync signal.
14
15 ****************************************************************/
16 /*lint -w0*/
17 #include "UDPserver.hpp"
18 #include "Ips4\Dev\IpsInDev.hpp"
19
20
21 class CamDeviceUdp: public IpsInputDevice
22 {
23 public:
24     CamDeviceUdp (const string& name, int portnumber);
25     ¬CamDeviceUdp();
26
27     // signal functions
28     enum { sigX=IpsInputDevice::sigNEXT, sigY, sigO, sigSync, ...
           sigSpeed, sigPreSync, sigSampleNo, sigTimestamp, ...
           sigLocalClock, sigNEXT };
29     virtual IpsUnit SignalUnit(int signo) const;
30     virtual const char* SignalName(int signo) const;
31     virtual bool Read(int signo, real* result) const;
32     //virtual bool Write(int signo, real value);
33
34     // param functions
35     enum { parPredictionType=IpsInputDevice::parNEXT, parNEXT };
36     virtual IpsUnit ParamUnit(int param) const;
37     virtual const char* ParamName(int param) const;
38     virtual bool GetParam(int param, real* result) const;
```

```
39      virtual bool DefParam(int param, real* dflt, real* minval, real* ...
            maxval) const;
40
41      virtual bool SetConnection(int no, IpsDevice* target, int signal);
42
43      double getPos();
44      double getSpeed();
45      bool getSync();
46      double getInterpX();
47      double getRawX();
48
49  protected:
50      virtual bool SetParam(int param, real value);
51      virtual void MonitorState();
52
53  private:
54      void calcSpeed();
55      void interpolatePos();
56      void tilbakePos();
57      //void interpolate2();
58      void calcConveyorPos();
59      void checkForSyncSignal();
60      void checkForPreSync(); //Pre-sync is used by robot to get objects ...
            orientation
61                                          //and Y offset before tracking starts.
62      void lowpass(double x, double y, double o);
63
64      double mX, mY, mO, mSpeed, mPrevX, mPrevY, mPrevLocalClock, ...
            mPrevXraw;//, mAcc;
65      double mInterpX, mTimestamp, mPrevTimestamp, mDeltaTimestamp;//, ...
            mAcc, mPrevAcc, mPrevSpeed, mLastTime;
66      double mSampleNo, mLocalClock, mClockStart, ...
            mEstimatedTimeOfAquisition, mXhat;
67      const int CAM_DELAY;
68      bool mSync, mPreSync, mFirstRound;
69      char* cameraMsg;
70      enum predictiontype{ConstSpeed, ConstSpeedTilbake};
71      predictiontype mPredType;
72
73      UDPserver *mUdp;
74      bool Init(int portnumber);
75      bool readUdpAndCalcStuff();
76
77  };
78
79  #endif
```

```
1
2  #include <string.h>
3  #include "sysdef.h"
4  #include <math.h>
5
6  #include "ips4\sup\AccuLog.hpp"
7  #include "ips4\sup\StrUtil.hpp"
8
9  #include "CamDeviceUdp.hpp"
10
11 /*lint -w0*/
12
13 /*****************************************************************
14             Constructor / Deconstructor
15 ****************************************************************/
16
17 CamDeviceUdp::CamDeviceUdp (const string& name, int portnumber)
18     : IpsInputDevice(name, false, sigNEXT-1, parNEXT-1), mX(0), mY(0), ...
           mO(0), mSpeed(0), mPrevX(0), mPrevY(0),mPrevXraw(0), mInterpX(0),
19         mTimestamp(0), mPrevTimestamp(0), mDeltaTimestamp(0), ...
               mSampleNo(0), mLocalClock(0), mClockStart(0), ...
               CAM_DELAY(60), mSync(false), mPreSync(false), ...
               mFirstRound(true), mPredType(ConstSpeed), ...
               mEstimatedTimeOfAquisition(-60),mXhat(0)
20 {
21     mUdp = new UDPserver();
22     Init(portnumber);
23
24     EnableMonitor(true);
25 }
26
27 CamDeviceUdp::¬CamDeviceUdp()
28 {
29 }
30
31 /*****************************************************************
32             Signal Functions
33 ****************************************************************/
34
35 IpsUnit CamDeviceUdp::SignalUnit(int signo) const
36 {
37
38     switch (signo)
39     {
40     case sigValue: return mmSec;     // speed
41     case sigAccu:   return mm;       // pos
42     case sigX:  return mm;//raw value
43     case sigY:  return mm;//raw value
44     case sigO: return AngDegr;
45     case sigSync:
46     case sigPreSync:
47     case sigSampleNo: return mSec;
48     case sigTimestamp: return mSec;
```

```
49      case sigLocalClock: return mSec;
50      default:            return IpsInputDevice::SignalUnit(signo);
51      }
52  }
53
54  bool CamDeviceUdp::Read(int signal, real* result) const
55  {
56
57      switch (signal)
58      {
59      case sigValue:
60          * result = mSpeed;
61          return true;
62      case sigAccu:
63          * result = mInterpX;
64          return true;
65      case sigX:
66          * result = mX;
67          return true;
68      case sigY:
69          * result = mY;
70          return true;
71      case sigO:
72          * result = mO;
73          return true;
74      case sigSync:
75          * result = mSync;
76          return true;
77      case sigPreSync:
78          * result = mPreSync;
79          return true;
80      case sigSampleNo:
81          * result = mSampleNo;
82          return true;
83      case sigTimestamp:
84          * result = mDeltaTimestamp;
85          return true;
86      case sigLocalClock:
87          * result = mLocalClock;
88          return true;
89      default:
90          return IpsInputDevice::Read(signal, result);
91      }
92  }
93
94  /*
95  bool CamDeviceUdp::Write(int signo, real value)
96  {
97      switch (signo)
98      {
99      case 1: return false;
100     default:
101         return IpsInputDevice::Write(signo, value,1);
```

```
102      }
103      return false;
104  }*/
105
106
107  const char* CamDeviceUdp::SignalName(int signo) const
108  {
109      switch(signo)
110      {
111      case sigValue: return "Speed";
112      case sigAccu: return "Position";
113      case sigX: return "X";
114      case sigY: return "Y";
115      case sigO: return "O";
116      case sigSync: return "Sync";
117      case sigPreSync: return "Pre-Sync";
118      case sigSampleNo: return "sampleNo";
119      case sigTimestamp: return "camDt";
120      case sigLocalClock: return "LocalClock";
121      default: return IpsInputDevice::SignalName(signo);
122      }
123  }
124  /*****************************************************************
125              Param Functions
126  *****************************************************************/
127
128
129
130
131  IpsUnit CamDeviceUdp::ParamUnit(int param) const
132  {
133
134      switch (param)
135      {
136      case parPredictionType: return Integer;
137      default: return IpsInputDevice::ParamUnit(param);
138      }
139  }
140
141  const char* CamDeviceUdp::ParamName(int param) const
142  {
143      switch (param)
144      {
145      case parPredictionType: return "PredType";
146      default: return IpsInputDevice::ParamName(param);
147      }
148  }
149
150  bool CamDeviceUdp::SetParam(int param, real value)
151  {
152      switch (param)
153      {
154      case parPredictionType: mPredType = (predictiontype)((int)value);
```

```
155         return true;
156     default:
157         return IpsInputDevice::SetParam(param, value);
158     }
159 }
160
161 bool CamDeviceUdp::GetParam(int param, real* result) const
162 {
163     switch  (param)
164     {
165     case parPredictionType:
166         *result = (real)mPredType;
167         return true;
168     default:
169         return IpsInputDevice::GetParam(param, result);
170     }
171 }
172
173 bool CamDeviceUdp::DefParam(int param, real* dflt, real* minval, real* ...
        maxval) const
174 {
175     switch (param)
176     {
177     case parPredictionType:
178         *minval = (real) 0;
179         *dflt = (real) 0;
180         *maxval = (real) 1;
181         return true;
182     default:
183         return IpsInputDevice::DefParam(param, dflt, minval, maxval);
184     }
185 }
186
187 bool CamDeviceUdp::SetConnection(int conn, IpsDevice* dev, int slot)
188 {
189     switch (conn)
190     {
191     case parPredictionType: return false;
192     default:
193         return IpsInputDevice::SetConnection(conn, dev, slot);
194     }
195 }
196
197 void CamDeviceUdp::MonitorState()
198 {
199     readUdpAndCalcStuff();
200
201     if(mMirror)
202         mMirror->DirectWrite(mSpeed);
203 }
204
205
206
```

```
207 /*****************************************************************
208         CAMERA SPECIFIC FUNCTIONS
209 *****************************************************************/
210
211     //Initialise UDP socket with port <portnumber>
212     bool CamDeviceUdp::Init(int portnumber)
213     {
214         bool initok = mUdp->udpInit(portnumber);
215
216         //Start time for time difference calculation
217         mPrevLocalClock = double(clock_ms());
218         mClockStart = mPrevLocalClock;
219
220         mEstimatedTimeOfAquisition = mPrevLocalClock-60;
221
222         return initok;
223     }
224
225     bool CamDeviceUdp::readUdpAndCalcStuff()
226     {
227         mPrevLocalClock = mLocalClock;
228         mLocalClock = double(clock_ms());
229         //if(mFirstRound)
230         //{
231         //  mEstimatedTimeOfAquisition = mLocalClock-120;
232     //      mFirstRound = false;
233
234     //   }
235
236         bool readOK = FALSE;
237         //udp read from camera, true if data has been read.
238         if(mUdp->udpRead())
239         {
240             if(mX == 0)
241             {
242                 mEstimatedTimeOfAquisition = mLocalClock-120;
243             }
244
245             mSampleNo = mSampleNo+1;
246
247             mPrevX = mX;
248             mPrevY = mY;
249
250             //Get data string that has been read:
251             //Has the form "X.XX Y.YY O.OO hh mm ss msmsms",
252
253             cameraMsg = mUdp->getMsg();
254
255             double Xraw, Yraw, Oraw;
256             Xraw = 0;
257             Yraw = 0;
258             Oraw = 0;
259
```

```
260
261             // read the camera string into number vaiables:
262             int hours, mins, secs, msecs;
263             hours = 0;
264             mins = 0;
265             secs = 0;
266             msecs = 0;
267             sscanf (cameraMsg,"%lf %lf %lf %d %d %d ...
                    %d",&Xraw,&Yraw,&Oraw, &hours, &mins, &secs, &msecs);
268
269
270             mPrevTimestamp = mTimestamp;
271             //calculate absolute timestamp; no. of milliseconds
272             mTimestamp = msecs+secs*1000+mins*60*1000+hours*60*60*1000;
273             mDeltaTimestamp = mTimestamp-mPrevTimestamp;
274
275             //Calculate ips time since last sample:
276             double dIpsTime = mLocalClock - mEstimatedTimeOfAquisition;
277             double diffTimes = dIpsTime - mDeltaTimestamp;
278
279             mEstimatedTimeOfAquisition = mLocalClock - diffTimes;
280
281             //printf("aquisition %lf \n",mEstimatedTimeOfAquisition);
282
283             lowpass(Xraw,Yraw,Oraw);// as of now, this function only ...
                    saves the variables to mX,mY,mO
284
285             //the parameter parPredictionType which can be set with ...
                    ips camera wp
286             //determines what type of prediction/extrapolation is done ...
                    on the signal
287             double error;
288             switch(mPredType)
289             {
290                 case ConstSpeed: calcSpeed(); break;
291                 case ConstSpeedTilbake: calcSpeed();
292                     mXhat = (mX + mSpeed*(mLocalClock-            ...
                                    mEstimatedTimeOfAquisition))*
293                             (1-0.0145);                           ...
                                    error = mInterpX+mSpeed*(mLocalClock-
294                         mPrevLocalClock) - mXhat;
295                     mSpeed = mSpeed-error*0.01;
296                     break;
297                 default: break;
298             }
299             //printf("speed %lf \n\n",mSpeed);
300             checkForPreSync();
301             checkForSyncSignal();
302             readOK = TRUE;
303         }
304
305         if(mX == 0){
306             mSpeed = 0;
```

```
307         }
308
309         switch(mPredType)
310         {
311             case ConstSpeed: interpolatePos(); break;
312             case ConstSpeedTilbake: tilbakePos(); break;
313             default: break;
314         }
315
316         return readOK;
317     }
318
319
320     void CamDeviceUdp::calcSpeed()
321     {
322         //mSpeed = double(mSpeed/3 + ...
                ((mX-mPrevX)/(mDeltaTimestamp/1000))*2/3);
323         mSpeed = (mX-mPrevX)/mDeltaTimestamp;
324     }
325
326     void CamDeviceUdp::calcConveyorPos()
327     {
328     }
329     void CamDeviceUdp::checkForSyncSignal()
330     {
331         //check if object is withing sync boundaries
332         if((mInterpX<80 && mInterpX>65))
333             mSync = true;
334         else mSync = false;
335     }
336
337     void CamDeviceUdp::checkForPreSync()
338     {
339         //check if object is withing pre-sync boundaries
340         if((mX<170 && mX>150))
341             mPreSync = true;
342         else mPreSync = false;
343     }
344
345     void CamDeviceUdp::lowpass(double x, double y, double o)
346     {
347         mX = x;//(X+x)/2;
348         mY = y;//(Y+y)/2;
349         mO = o;//(O+o)/2;
350     }
351
352     void CamDeviceUdp::interpolatePos()
353     {
354         //mInterpX = mX + mSpeed*(clock_ms()+60-
355             mPrevLocalClock)/1000;
356             mInterpX = (mX + mSpeed*(double(clock_ms())-
357             mEstimatedTimeOfAquisition))*(1-0.0145);
358     }
```

```
359
360     void CamDeviceUdp::tilbakePos()
361     {
362         //mInterpX = mX + mSpeed*(clock_ms()+60-mPrevLocalClock)/1000;
363         mInterpX = mInterpX + mSpeed*(mLocalClock-mPrevLocalClock);
364         mPrevLocalClock = double(clock_ms());
365     }
366
367     double CamDeviceUdp::getPos()
368     {
369         return mInterpX;
370     }
371     double CamDeviceUdp::getSpeed()
372     {
373         return mSpeed;
374     }
375     bool CamDeviceUdp::getSync()
376     {
377         return mSync;
378     }
379     double CamDeviceUdp::getInterpX()
380     {
381         return mInterpX;
382     }
383     double CamDeviceUdp::getRawX()
384     {
385         return mX;
386     }
```

A.1.2 `UDPserver` (hpp/cpp)

```
#ifndef UDPserver_HPP
#define UDPserver_HPP

//include OS appropriate network classes (Linux / Windows)
#include "os/osnet.h"


#include <string>
#include <sstream>
//#include <Time.h>

//#pragma comment(lib,"ws2_32.lib") //Winsock Library

#define BUFLEN 512



class UDPserver
{
    public:
        UDPserver();
        ¬UDPserver();
        bool udpRead();
        bool udpInit(/*FILE* out, */int port);
        char* getMsg();

    private:
        int s, rc, on;
        struct sockaddr_in server;
       int slen, recv_len;
        char buf[BUFLEN];
        fd_set readset;
        bool data_received;

#if defined(WIN32)
        WSADATA wsa;
#endif

};


#endif
```

```
1
2 /*--------------------- I n c l u d e   F i l e s ...
      ---------------------------------------*/
3
4 //#include <unistd.h>
5 #include <stdio.h>
6
7 #include "sysdef.h"
8 #include "UDPserver.hpp"
9
10 #include "os/osdef.h"
11
12 /*lint -w0*/
13
14 UDPserver::UDPserver()
15     : data_received(FALSE)
16 {
17 }
18
19
20 bool UDPserver::udpInit(/*FILE* out,*/ int port)
21 {
22     bool initOK = TRUE;
23         //Initialise Winsock
24 #if defined(WIN32)
25     //fprintf(out,"Initialising Winsock...\n");
26     if(WSAStartup(MAKEWORD(2,2),&wsa) != 0)
27     {
28         //fprintf(out,"Failed. Error code: %d \n",WSAGetLastError());
29         initOK = FALSE;
30     }
31     //else fprintf(out, "Inititalised.\n");
32 #endif
33
34     //Create a socket
35     if((s=socket(AF_INET, SOCK_DGRAM, 0)) == -1)
36     {
37         //fprintf(out,"Could not create socket. \n");
38         initOK = FALSE;
39     }
40     //else fprintf(out,"Socket created.\n");
41
42     //Prepare the sockaddr_in structure
43    server.sin_family = AF_INET;
44    server.sin_addr.s_addr = INADDR_ANY;
45     server.sin_port = htons( port );
46
47     //Bind
48     if( bind(s ,(struct sockaddr *)&server , sizeof(server)) == ...
        SOCKET_ERROR)
49     {
50         //fprintf(out, "Bind failed.");
51         initOK = FALSE;
```

```
52      }
53      //else fprintf(out, "Bind done");
54
55      return initOK;
56  }
57
58  bool UDPserver::udpRead()
59  {
60
61          //clear the buffer by filling null. It might have previously ...
                received data.
62          memset(buf,'\0',BUFLEN);
63
64          FD_ZERO(&readset);
65          FD_SET(s, &readset);
66
67          //timeval struct for timeout of select function.
68          struct timeval tim;
69          tim.tv_sec = 0;
70          tim.tv_usec = 0;
71
72          //if data is available, read data.
73          if(select(s + 1, &readset, NULL, NULL, &tim)>0)
74          {
75              recv(s,buf,BUFLEN,0);
76              data_received = TRUE;
77          }
78          else
79          {
80                  data_received = FALSE;
81          }
82
83      return data_received;
84  }
85
86  char * UDPserver::getMsg()
87  {
88      return buf;
89  }
90
91  UDPserver::¬UDPserver()
92  {
93      closesocket(s);
94  #if defined(WIN32)
95      WSACleanup();
96  #endif
97  }
```

A.1.3 CamEncoder (hpp/cpp)

```
#ifndef CamEncoder_HPP
#define CamEncoder_HPP

// remove "multiple IpsInDev Lint error:
//lint -e537

#include "CamDeviceUdp.hpp"
#include "Ips4\Dev\IpsInDev.hpp"

class CamEncoder: public IpsInputDevice
{
public:
    CamEncoder (CamDeviceUdp *cam, const string& name);
    ¬CamEncoder();

    // signal functions
    enum { sigRawX=IpsInputDevice::sigNEXT, sigNEXT};
    virtual IpsUnit SignalUnit(int signo) const;
    virtual const char* SignalName(int signo) const;
    virtual bool Read(int signo, real* result) const;
    //virtual bool Write(int signo, real value);

    // param functions
    enum { parMyPar=IpsInputDevice::parNEXT, parNEXT };
    virtual IpsUnit ParamUnit(int param) const;
    virtual const char* ParamName(int param) const;
    virtual bool GetParam(int param, real* result) const;
    virtual bool DefParam(int param, real* dflt, real* minval, real* ...
        maxval) const;

    enum { connInput=IpsInputDevice::connNEXT };
    virtual bool SetConnection(int no, IpsDevice* target, int signal);
    virtual IpsDevice* GetConnection(int no, const char** usage, int* ...
        signo);

protected:
    virtual bool SetParam(int param, real value);
    virtual void MonitorState();

private:
    CamDeviceUdp *mCamera;
};

#endif
```

```

#include <string.h>
#include "sysdef.h"

#include "ips4\sup\AccuLog.hpp"
#include "ips4\sup\StrUtil.hpp"

#include "CamEncoder.hpp"


/****************************************************************
            Constructor / Deconstructor
****************************************************************/

CamEncoder::CamEncoder (CamDeviceUdp *cam, const string& name)
    : IpsInputDevice(name, true, sigNEXT-1, parNEXT-1)
{
    mCamera = cam;
    //EnableMonitor(true);
}

CamEncoder::¬CamEncoder()
{
}

/****************************************************************
            Signal Functions
****************************************************************/

IpsUnit CamEncoder::SignalUnit(int signo) const
{

    switch (signo)
    {
    case sigValue: return mmSec;
    case sigAccu:   return mm;
    case sigRawX:   return pix;
    default:            return IpsInputDevice::SignalUnit(signo);
    }
}

bool CamEncoder::Read(int signal, real* result) const
{

    switch (signal)
    {
    case sigValue:
        * result = mCamera->getSpeed();
        return true;
    case sigAccu:
        * result = mCamera->getPos();
        return true;
    case sigRawX:
```

```
54          * result = mCamera->getRawX();
55          return true;
56      default:
57          return IpsInputDevice::Read(signal, result);
58      }
59  }
60
61  /*
62  bool CamEncoder::Write(int signo, real value)
63  {
64      switch (signo)
65      {
66      case 1: return false;
67      default:
68          return IpsInputDevice::Write(signo, value,1);
69      }
70      return false;
71  }*/
72
73  const char* CamEncoder::SignalName(int signo) const
74  {
75      switch(signo)
76      {
77      case sigValue: return "Speed";
78      case sigAccu: return "Position";
79      case sigRawX: return "RawX";
80      default: return IpsInputDevice::SignalName(signo);
81      }
82  }
83  /*****************************************************************
84              Param Functions
85  *****************************************************************/
86
87
88
89
90  IpsUnit CamEncoder::ParamUnit(int param) const
91  {
92
93      switch (param)
94      {
95      case 1: return IpsInputDevice::ParamUnit(1);
96      default: return IpsInputDevice::ParamUnit(param);
97      }
98  }
99
100 const char* CamEncoder::ParamName(int param) const
101 {
102     switch (param)
103     {
104     case 1: return "myParam";
105     default: return IpsInputDevice::ParamName(param);
106     }
```

```
107 }
108
109 bool CamEncoder::SetParam(int param, real value)
110 {
111     switch (param)
112     {
113     case 1: return false;
114     default:
115         return IpsInputDevice::SetParam(param, value);
116     }
117 }
118
119 bool CamEncoder::GetParam(int param, real* result) const
120 {
121     switch  (param)
122     {
123     case 1: return false;
124     default:
125         return IpsInputDevice::GetParam(param, result);
126     }
127 }
128
129 bool CamEncoder::DefParam(int param, real* dflt, real* minval, real* ...
        maxval) const
130 {
131     switch (param)
132     {
133     case 1: return false;
134     default:
135         return IpsInputDevice::DefParam(param, dflt, minval, maxval);
136     }
137 }
138
139 bool CamEncoder::SetConnection(int conn, IpsDevice* dev, int slot)
140 {
141     switch (conn)
142     {
143     case 1: return false;
144     default:
145         return IpsInputDevice::SetConnection(conn, dev, slot);
146     }
147 }
148
149 IpsDevice* CamEncoder::GetConnection(int no, const char** usage, int* ...
        signo)
150 {
151     switch (no)
152     {
153     case connInput:
154         if (signo) *signo = sigValue;
155         if (usage) *usage = "CamEncInput";
156         return mCamera;
157
```

```
158     default:
159         return IpsInputDevice::GetConnection(no, usage, signo);
160     }
161 }
162
163 void CamEncoder::MonitorState()
164 {
165     if(mMirror)
166         mMirror->DirectWrite(mCamera->getSpeed());
167 }
```

A.1.4 CamSync (hpp/cpp)

```
1  #ifndef CamSync_HPP
2  #define CamSync_HPP
3
4  // remove "multiple IpsInDev Lint error:
5  //lint -e537
6
7  #include "CamDeviceUdp.hpp"
8  #include "Ips4\Dev\IpsInDev.hpp"
9
10 class CamSync: public IpsInputDevice
11 {
12 public:
13     CamSync (CamDeviceUdp *cam, const string& name);
14     ¬CamSync();
15
16     // signal functions
17     //enum { sigX=IpsInputDevice::sigNEXT, sigNEXT };
18     virtual IpsUnit SignalUnit(int signo) const;
19     virtual const char* SignalName(int signo) const;
20     virtual bool Read(int signo, real* result) const;
21     //virtual bool Write(int signo, real value);
22
23     // param functions
24     enum { parMyPar=IpsInputDevice::parNEXT, parNEXT };
25     virtual IpsUnit ParamUnit(int param) const;
26     virtual const char* ParamName(int param) const;
27     virtual bool GetParam(int param, real* result) const;
28     virtual bool DefParam(int param, real* dflt, real* minval, real* ...
           maxval) const;
29
30     enum { connInput=IpsInputDevice::connNEXT };
31     virtual bool SetConnection(int no, IpsDevice* target, int signal);
32     virtual IpsDevice* GetConnection(int no, const char** usage, int* ...
           signo);
33
34 protected:
35     virtual bool SetParam(int param, real value);
36     virtual void MonitorState();
37
38 private:
39     CamDeviceUdp *mCamera;
40 };
41
42 #endif
```

```
1
2  #include <string.h>
3  #include "sysdef.h"
4
5  #include "ips4\sup\AccuLog.hpp"
6  #include "ips4\sup\StrUtil.hpp"
7
8  #include "CamSync.hpp"
9
10
11 /****************************************************************
12             Constructor / Deconstructor
13 ****************************************************************/
14
15 CamSync::CamSync (CamDeviceUdp *cam, const string& name)
16     : IpsInputDevice(name, true, sigNEXT-1, parNEXT-1)
17 {
18     mCamera = cam;
19     //EnableMonitor(true);
20 }
21
22 CamSync::¬CamSync()
23 {
24 }
25
26 /****************************************************************
27             Signal Functions
28 ****************************************************************/
29
30 IpsUnit CamSync::SignalUnit(int signo) const
31 {
32
33     switch (signo)
34     {
35     case sigValue: return Bit;
36     default:             return IpsInputDevice::SignalUnit(signo);
37     }
38 }
39
40 bool CamSync::Read(int signal, real* result) const
41 {
42
43     switch (signal)
44     {
45     case sigValue:
46         * result = mCamera->getSync(); // place sync signal in result. ...
               10 is the camdevice's sync signo.
47         return true;
48     default:
49         return IpsInputDevice::Read(signal, result);
50     }
51 }
52
```

```
53  /*
54  bool CamSync::Write(int signo, real value)
55  {
56      switch (signo)
57      {
58      case 1: return false;
59      default:
60          return IpsInputDevice::Write(signo, value,1);
61      }
62      return false;
63  }*/
64
65  const char* CamSync::SignalName(int signo) const
66  {
67      switch(signo)
68      {
69      case sigValue: return "Sync";
70      default: return IpsInputDevice::SignalName(signo);
71      }
72  }
73  /************************************************************
74              Param Functions
75  ************************************************************/
76
77
78
79
80  IpsUnit CamSync::ParamUnit(int param) const
81  {
82
83      switch (param)
84      {
85      case 1: return IpsInputDevice::ParamUnit(1);
86      default: return IpsInputDevice::ParamUnit(param);
87      }
88  }
89
90  const char* CamSync::ParamName(int param) const
91  {
92      switch (param)
93      {
94      case 1: return "myParam";
95      default: return IpsInputDevice::ParamName(param);
96      }
97  }
98
99  bool CamSync::SetParam(int param, real value)
100 {
101     switch (param)
102     {
103     case 1: return false;
104     default:
105         return IpsInputDevice::SetParam(param, value);
```

```
106     }
107 }
108
109 bool CamSync::GetParam(int param, real* result) const
110 {
111     switch  (param)
112     {
113     case 1: return false;
114     default:
115         return IpsInputDevice::GetParam(param, result);
116     }
117 }
118
119 bool CamSync::DefParam(int param, real* dflt, real* minval, real* ...
        maxval) const
120 {
121     switch (param)
122     {
123     case 1: return false;
124     default:
125         return IpsInputDevice::DefParam(param, dflt, minval, maxval);
126     }
127 }
128
129 bool CamSync::SetConnection(int no, IpsDevice* dev, int slot)
130 {
131     switch (no)
132     {
133     case 1: return false;
134     default:
135         return IpsInputDevice::SetConnection(no, dev, slot);
136     }
137 }
138
139 IpsDevice* CamSync::GetConnection(int no, const char** usage, int* signo)
140 {
141     switch (no)
142     {
143     case connInput:
144         if (signo) *signo = sigValue;
145         if (usage) *usage = "CamSyncInput";
146         return mCamera;
147
148     default:
149         return IpsInputDevice::GetConnection(no, usage, signo);
150     }
151 }
152
153 void CamSync::MonitorState()
154 {
155     if(mMirror)
156         mMirror->DirectWrite(mCamera->getSync());
157 }
```