# Teknisk dokumentation

Mats Tjäder

**Version 1.0**

Status

| | | |
|---|---|---|
| Granskad | | |
| Godkänd | Martin Enqvist | 2005-05-18 |

# PROJEKTIDENTITET

Linköpings tekniska högskola, Instutionen för systemteknik, ISY

| Namn | Ansvar | Telefon | E-post |
|---|---|---|---|
| Björn Wedell | kundansvarig (KUN) | 070-6655356 | bjowe774@student.liu.se |
| Mats Tjäder | dokumentansvarig (DOK) | 070-3545400 | mattj452@student.liu.se |
| Henrik Einarsson | designansvarig bild (DESB) | 070-3484676 | henei960@student.liu.se |
| Marcus Olofsson | designansvarig robot (DESR) | 070-6713303 | marol791@student.liu.se |
| Johannes Eklånge | testansvarig (TST) | 070-7711529 | johek016@student.liu.se |
| Johan Nordin | presentationsansvarig (PRES) | 073-1507959 | johno660@student.liu.se |
| Alexander Konradsson | projektledare (PL) | 070-2058260 | aleko181@student.liu.se |

**Projekthemsida:** http://www.cyd.liu.se/~mattj452/TSRT71
**Kund:** Henrik Tidefelt, ISY LiTH, 013-281311, tidefelt@isy.liu.se
**Beställare:** Martin Enqvist, 013-282306, maren@isy.liu.se
**Kursansvarig**: Anders Hansson, 013-281681, hansson@isy.liu.se
**Handledare**: Erik Wernholt, 013-281333, erikw@isy.liu.se

# Innehåll

Dokumenthistorik

| version | datum | utförda förändringar | utförda av | granskad |
|---|---|---|---|---|
| 0.1 | 2005-05-16 | Första versionen | Hela gruppen | Hela gruppen |
| 1.0 | 2005-05-18 | Andra versionen, godkänd | Hela gruppen | Hela gruppen |

# 1 Inledning

I denna projektkurs gjordes under år 2004 ett projekt där en industrirobot av modell ABB IRB1400 konfigurerades för att spela minigolf på en speciell bana. Den kunde dels styras manuellt genom att man angav utslagsvinkel och hastighet, dels genom att användaren endast angav vinkel och roboten automatiskt slog med rätt styrka för att bollen skulle gå i hål. I samma kurs 2005 har projektet gått ut på att få roboten att prestera bättre bland annat med hjälp av en digitalkamera. Material från föregående år har använts. Roboten kan utföra samma saker som förra året, men mycket bättre. Den har dessutom hjälp av kameran för att kunna bestämma mer exakt vad som är fel i ett slag och därmed förbättra sig till nästa slag. Roboten kan hitta en boll som stannat på banan och slå den i hålet därifrån. Banan har även utökats med en green.

## *1.1 Parter*

Kund är Henrik Tidefelt, ISY, LiTH. Projektet har utförts av en projektgrupp bestående av sju studenter i årskurs 4 på Y-programmet på kursen TSRT71 Reglerteknisk projektkurs.

## *1.2 Mål*

Syftet är att vidareutveckla det befintliga demonstrations-styrprogramet till industriroboten ABB IRB1400 vilket framtogs våren 2004 i projektet ”Golfspelande Industrirobot”. Roboten skall programmeras att spela minigolf på en liten men svår minigolfbana. Detta sker bl.a. med en mer avancerad matematisk modell, en kamera som identifierar bollbanan samt förbättrad dynamik.

## *1.3 Användning*

Den minigolfspelande roboten skall visas upp vid olika informations- och reklamevenemang vid Linköpings universitet. Roboten skall utgöra ett exempel på en tillämpning av modellering och reglerteknik.

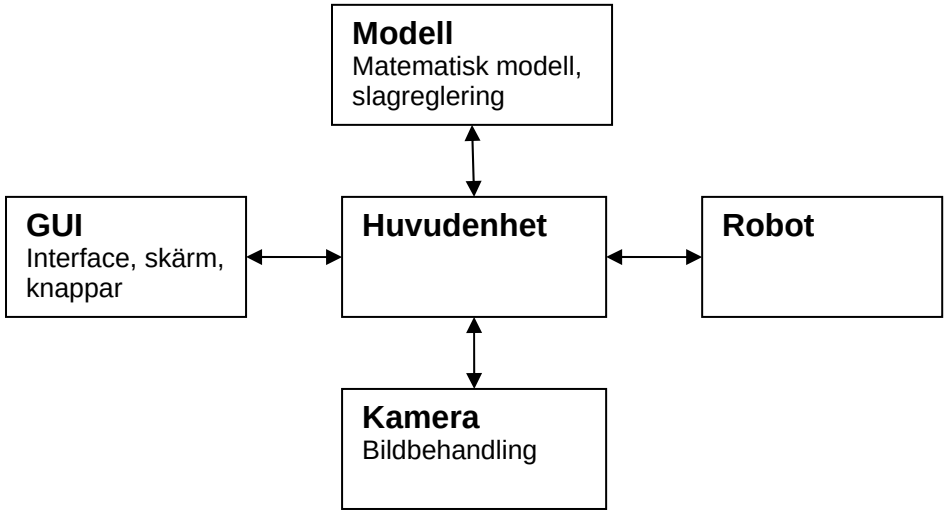
# 2 Översikt av systemet

Systemet är uppbyggt av fem moduler: kamera, matematisk modell, grafiskt användargränssnitt, huvudenhet och robot. Det grafiska användargränssnittet och huvudenheten är ett program skrivet i Java där användaren med hjälp av knappar kan styra roboten att spela och ge olika order. Användaren kan få roboten att slå bollen med angiven vinkel respektive hastighet, hitta bollen och välja mellan olika moder. Programmet kan också visa en bild av banan där beräknad bana utifrån angiven vinkel och hastighet syns. Efter ett slag (av robot eller människa) visas även bollens egentliga bana. PI-reglering används för att förbättra ett av roboten missat slag.

Den matematiska modellen används i tävlings-, uppvisnings- och manuell mod och har implementerats i Matlab. Den tar information från huvudenheten och efter beräkningar skickas information om hastighet och eventuellt utslagsvinkel för bollen tillbaka. I detta projekt har den matematiska modellen förbättras gentemot 2004 års projekt.

En digitalkamera används för att identifiera bollens bana. Detta för att huvudheten via den matematiska modellen ska kunna räkna ut nya parametrar vid ett misslyckat slag för att förbättra detta.

Roboten har programmerats i programspråket Rapid 3.0 och utför olika kommandon från huvudenheten.




*Figur 1. Blockschema över delmodulerna*

## *2.1 Grov beskrivning av produkten*

Det finns fyra olika moder. Tre av moderna inkluderar att roboten ska styras till att slå bollen. I manuell mod anges både vinkel och hastighet och roboten slår bollen enligt detta utan att tänka själv. I uppvisningsmod anges endast vinkel och roboten beräknar med hjälp av modellen en lämplig hastighet för att träffa hålet. I tävlingsmoden då en robot deltar i

tävlingen slumpas en vinkel fram och lämplig hastighet beräknas med hjälp av modellen. I uppvisningsmod och tävlingsmod används en PI-regulator för att förbättra misslyckade slag. I alla dessa moder visas den beräknade bollbanan i grafiken. I träningsmoden används inte roboten utan istället får en användare slå ett slag själv på banan. Bollbanan läses in med hjälp av kameran och visas i grafiken. Användaren kan nu få tips på hur slaget ska förbättras för att bollen ska gå i hålet.

### *2.2 Produktkomponenter*

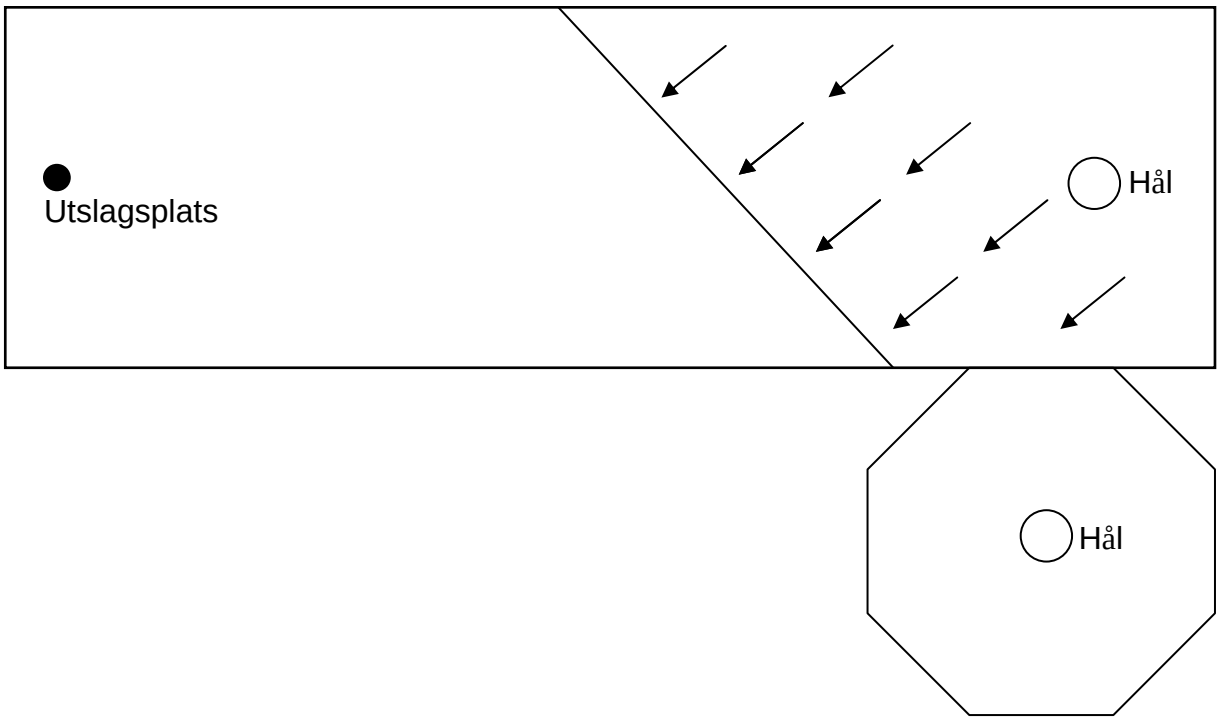
Användargränssnittet och huvudenheten är implementerade i Java och modellen i Matlab. Styrsystemet till roboten är S4C och programmeras i Rapid 3.0. Roboten är av modell ABB IRB1400. Kameran som används är av typen AVT Marlin F-145C2.

### *2.3 Ingående delsystem*

Systemet är uppbyggt av fem moduler: kamera, matematisk modell, grafiskt användargränssnitt, huvudenhet och robot.

### *2.4 Banspecifikation*

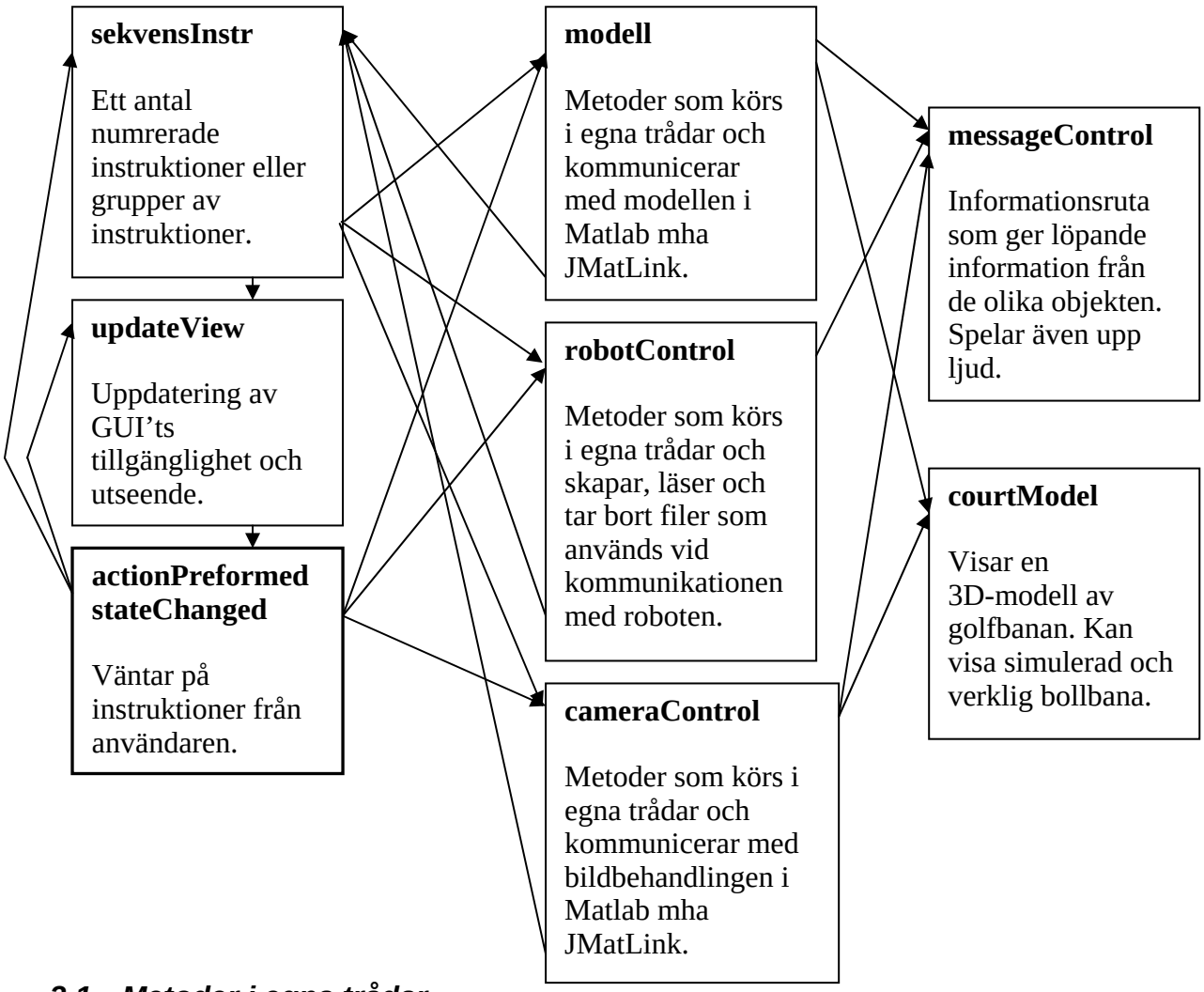
Banan är en minigolfbana som är byggd av ISY vid Linköpings universitet. Banan har en backe som sluttar åt ena kanten upp mot ett hål som leder ned till en green där hålet som bollen ska träffa finns.




*Figur 2. Skiss över banan*

# 3 Huvudenheten

Huvudprogrammet är skrivet i Java. Programmet ligger under körning och väntar på kommandon i funktionerna actionPreformed och stateChanged. Dessa två funktioner lyssnar efter händelser i alla GUI-objekt, d.v.s. alla knappar, menyer och reglar. För att kommunicera med alla moduler finns objekt av klasserna modell, robotControl, cameraControl, courtModel och messageControl. Vissa av metoderna körs i trådar separerade från huvudprogrammets tråd och avslutas eventuellt med att be huvudprogrammet att anropa en ny metod. Huvudprogrammet har även en metod för att uppdatera vyn. Denna metod låser, låser upp, visar och gömmer knappar och reglar.




## *3.1 Metoder i egna trådar*

Objekten i klasserna modell, robotControl och cameraControl har en del metoder som inte är ögonblickliga utan tar en viss tid att utföra. Som exempel kan tas så gott som alla metoder i robotControl-objektet. Från det att en metod anropas för att göra en operation med roboten tills det att roboten är redo för att ta emot en ny order kan det dröja ett flertal sekunder. För att undvika att huvudprogrammet ska låsas medan en sådan metod utförs körs dessa i trådar separerade från tråden som huvudprogrammet körs i. Detta gör att tidskrävande metoder kan anropas av huvudprogrammet som sedan genast kan återgå till att lyssna på kommandon från användaren.

### *3.2 Instruktionssekvenser*

Metoderna som körs i egna trådar startas alltid med ett argument som berättar vad som eventuellt ska hända direkt efter att metoden är färdig. Detta åstadkoms genom att varje metod av denna typ avslutas med att anropa en metod som kallas sekvensInstr i huvudprogrammet. Denna tar som argument ett heltal som representerar en instruktion. En principiell beskrivning av funktionen sekvensInstr ser ut så här:

```
void sekvensInstr(int instr) {
  switch instr {
    case 0:
      break;
    case 1:
      rbt.prepareHitOnTee(velocity,angle,2);
      break;
    case 2:
      rbt.execute(velocity,angle,0);
      break;
    case 3:
.
.
.
  }
  updateView();
}
```

För att köra sekvensen placeBall->prepareHitOnTee->execute körs nu bara kommandot
**rbt.placeBall(court.TRAY, court.TEE, 1);**
I vissa fall anropas även sekvensInstr direkt av huvudprogrammet istället för att gå genom en annan metod.
Observera att detta är en väldigt förenklad och principiell beskrivning. Den egentliga Java-koden ser inte ut exakt så här, men principen är densamma.

### *3.3 Uppdatering av vyn*

Funktionen sekvensInstr avslutas med att metoden updateView anropas. Detta sker även när andra saker händer som ska påverka användarens möjligheter till att göra saker i GUI't. Metoden låser, låser upp, visar och gömmer knappar, reglar och textremsor på lämpligt vis. När användaren t.ex. trycker på en knapp för att byta mod ändras först variabeln som bestämmer vilken mod vi befinner oss i och sedan anropas updateView så att GUI't förändras i enlighet med den nya moden. Metoden frågar även de tre objekten i klasserna modell, robotControl och cameraControl om de är aktiva och låser eventuellt knappar eller reglar enligt detta. T.ex. låses "Slå"-knappen, robot-menyn m.m. så fort robotControl-objektet är aktivt.

## *3.4 Speciella klasser*

### 3.4.1 modell

Modell-klassen gör beräkningar och hämtar information i Matlab med hjälp av JMatLink. Huvudenheten kan anropa följande metoder i denna klass.

| **Kommando** | **Argument** | **Retur** | Förklaring |
|---|---|---|---|
| **simulate** | **vinkel, hastighet, next** | **-** | Simulerar en bollbana med given vinkel och hastighet. Metoden anropar också courtModel-objektet så att visualiseringen av den simulerade bollbanan uppdateras. Startas och körs i en egen tråd och avslutas med att anropa sekvensInstr(next). |
| **calculate** | **vinkel, next** | **-** | Beräknar lämplig hastighet för att bollen ska passera målet. Bollbanan simuleras och visas även med denna metod. Startas och körs i en egen tråd och avslutas med att anropa sekvensInstr(next). |
| **getSimulated** | **-** | **Simulerad bollbana i en matris** | |
| **getAngle, getVelocity** | | **vinkel resp. hastighet.** | |

### 3.4.2 robotControl

Ett objekt i robotControl-klassen skriver, läser och tar bort filer på nätverket. Robotstyrprogrammet väntar på att filerna ska dyka upp och utför sedan kommandon i enlighet med dessa. Det finns att läsa om detta i kapitlet om robotstyrprogrammet (kapitel 5). Huvudenheten kan anropa följande metoder i denna klass.

| **Kommando** | **Argument** | **Retur** | Förklaring |
|---|---|---|---|
| **placeBall** | **från, till, next** | **-** | Flyttar bollen från en position till en annan. Anropas ofta med fasta värden från courtModel, tex TEE eller TRAY. Startas och körs i en egen tråd och avslutas med att anropa sekvensInstr(next). |
| **prepareHitOnTee** | **vinkel, hastighet, next** | **-** | Förbereder ett slag med viss vinkel och hastighet från utslagspositionen. Startas och körs i en egen tråd och avslutas med att anropa sekvensInstr(next). |
| **execute** | **vinkel, hastighet, next** | **-** | Slår bollen med given vinkel och hastighet från utslagspositionen. Startas och körs i en egen tråd och avslutas med att anropa sekvensInstr(next). |
| **hitOnGreen** | **position, next** | **-** | Puttar bollen i hålet på green. Innehåller viss logik för att undersöka om bollen ligger i ett läge där det är olämpligt att slå. Om så är fallet flyttas bollen ut och slås från den nya positionen. Startas och körs i en egen tråd och avslutas med att anropa sekvensInstr(next). |
| **sweep** | - | - | Sveper med klubban över hela utslagsområdet som kameran inte ser. Används för att få bollen att hamna i hämtpositionen. |
| **getSensor1** | | **true/false** | Returnerar värdet som sensor1 (true om bollen ligger på hämtpositionen) fick efter senaste robotoperationen. |
| **getSensor2** | | **true/false** | Returnerar värdet som sensor2 (true om bollen rullat ner till green) fick efter senaste robotoperationen. |

### 3.4.3 cameraControl

cameraControl-klassen ger ordrar till och får information från Matlab med hjälp av JMatLink. Huvudenheten kan anropa följande metoder i denna klass.

| **Kommando** | **Argument** | **Retur** | Förklaring |
|---|---|---|---|
| **calibrate** | **next** | **-** | Gör ett försök till automatisk kalibrering av kameran. Misslyckas detta får användaren upp en ruta med information om hur man kan göra kalibreringen manuellt alternativt försöka igen. Startas och körs i en egen tråd och avslutas med att anropa sekvensInstr(next). |
| **startAcq** | **next** | **-** | Tar in bollbanan vid ett slag. Metoden anropar också courtModel-objektet så att visualiseringen av den verkliga bollbanan uppdateras. Startas och körs i en egen tråd och avslutas med att anropa sekvensInstr(next). |
| **ballInHole** | **-** | **true/false** | Returnerar sant om den senast intagna bollbanan resulterade i att bollen gick i hål och rullade ner till green. Används i träningsmod, då inte sensorerna som hör till roboten kan användas. |
| **getGreenPos** | **next** | | Tar en bild av green och returnerar bollens position om bollen finns på green. Hittas inte bollen returneras origo. Startas och körs i en egen tråd och avslutas med att anropa sekvensInstr(next). |
| **saveImage** | **true/false** | | Sätter variabeln som bestämmer om bilderna som tas med kameran ska sparas eller ej. |

### 3.4.4 courtModel

Objektet i courtModel-klassen är ett visuellt objekt i GUI't. Det visar en modell av banan, en pil som representerar vald vinkel och hastighet, simulerad bollbana och den riktiga bollbanan enligt kameran. Allting visas i 3D och som användare kan man med hjälp av musen vrida, zooma och translatera bilden. Metoderna i detta objekt anropas inte enbart av huvudenheten utan även av modell-objektet och cameraControl-objektet. Här är de metoder som finns i klassen.

| **Kommando** | **Argument** | **Retur** | Förklaring |
|---|---|---|---|
| **setAimStick** | **vinkel, hastighet** | **-** | Ändrar riktningen och storleken på pilen. |
| **setTrajectory** | **bollbana** | **-** | Sätter en ny verklig bollbana. Anropas enbart av cameraControl. |
| **setSimulatedTrajectory** | **bollbana** | **-** | Sätter en ny simulerad bollbana. Anropas enbart av modell. |

### 3.4.5 messageControl

Även objektet i messageControl-klassen är ett visuellt objekt. Detta är en textruta längst ned i GUI't som skriver ut meddelanden från alla olika moduler. Klassen kan även spela upp ljud. Här är metoderna som finns i klassen.

| **Kommando** | **Argument** | **Retur** | Förklaring |
|---|---|---|---|
| **addMessage** | **meddelande** | **-** | Ett meddelande läggs till i rutan med tidsstämpel. Rutan skrollar alltid ner till det sista meddelandet när det läggs in. |
| **soundStart, soundShortBeep, soundLongBeep, soundInstructions, soundCongratulations mm.** | **next** | | Metoder som spelar upp olika ljud. Startas och körs i en egen tråd och avslutas med att anropa sekvensInstr(next). |

# 4 Matematisk modell

Den matematiska modellen har två uppgifter. Den ska dels för given vinkel returnera rätt utgångshastighet för att bollen ska gå i hål samt simulerad bollbana. Dess andra uppgift är att om bollen inte gick i hål reglera slaget med hjälp av kamerans mätningar och räkna ut en ny lämplig hastighet och ny bollbana. Detta har vi löst genom att göra en fysikalisk modell och en PI-regulator.

## *4.1 Modell av banan*

Golfbanan mättes upp av roboten i ett 10*10 centimeters gridnät. Två plan anpassades till banans undre och övre del. Mellan dessa plan anpassades ett tvådimensionellt polynom. I skarvarna blev det små diskontinuiteter som filtrerades bort med ett gaussiskt filter. När förbehandlingen av mätningarna var gjord generades en spline-funktion över hela ytan. Med spline-funktionen kan ytans normaler beräknas godtyckligt tätt.

## *4.2 Gränssnitt*

Följande funktioner används av huvudenheten vid kommunikationen med modellen:

| **Kommando** | **Argument** | **Retur** | Förklaring |
|---|---|---|---|
| **simulate** | **angle, velocity** | **trajectory** | Simulerar systemet. Dvs tar hastighet och vinkel som inargument och returnerar en bollbana. |
| **simulate_hole** | **angle, strength** | **velocity, trajectory** | Simulate_hole tar som inargument den vinkel som man önskar skjuta med. Dessutom anges om man vill skjuta ett löst eller hårt slag. Om det finns en lösning för den angivna vinkeln får man som utargument den hastighet som gör att man träffar hål samt den beräknade bollbanan för vinkeln och hastigheten. |
| **hit_controller** | **trajectory** | - | Slagregulator som ändrar siktet baserat på mätningar från kameran. |

## *4.3 Fysikalisk modell*

Den fysikaliska modellen räknar fram en bollbanan utifrån en godtycklig vinkel och klubbhastighet.

### 4.3.1 Teori

För att ta fram den fysikaliska modellen har vi börjat med att frilägga bollen. Vi har begränsat oss på det viset att vi antar att bollen aldrig glider utan att den hela tiden rullar fram. Detta leder till att vi inte har med någon friktion mot planet, istället har vi valt att ha en hasighetsdämpningsfaktor som verkar i rakt motsatt riktning som hastigheten i bollens centrum.

Vi använder oss av två stycken koordinatsystem. Ett som är fast och som skrivs med versaler och ett som följer med bollens kontaktpunkt med planet och som skrivs med gemener.

$\hat{n} = n_X \hat{X} + n_Y \hat{Y} + n_Z \hat{Z}$

$\overline{F}_N = mgn_Z \hat{n}$

$\overline{P}_O$ *bollens position i centrum*

$\overline{P}_A$ *bollens position i planet*

$\dot{\overline{P}}_O = \overline{V}_O = \overline{\omega} \times r \cdot \hat{n}$ (vi antar ingen glidning)

$\lambda_V$ *= hastighetsdämpningsfaktor*

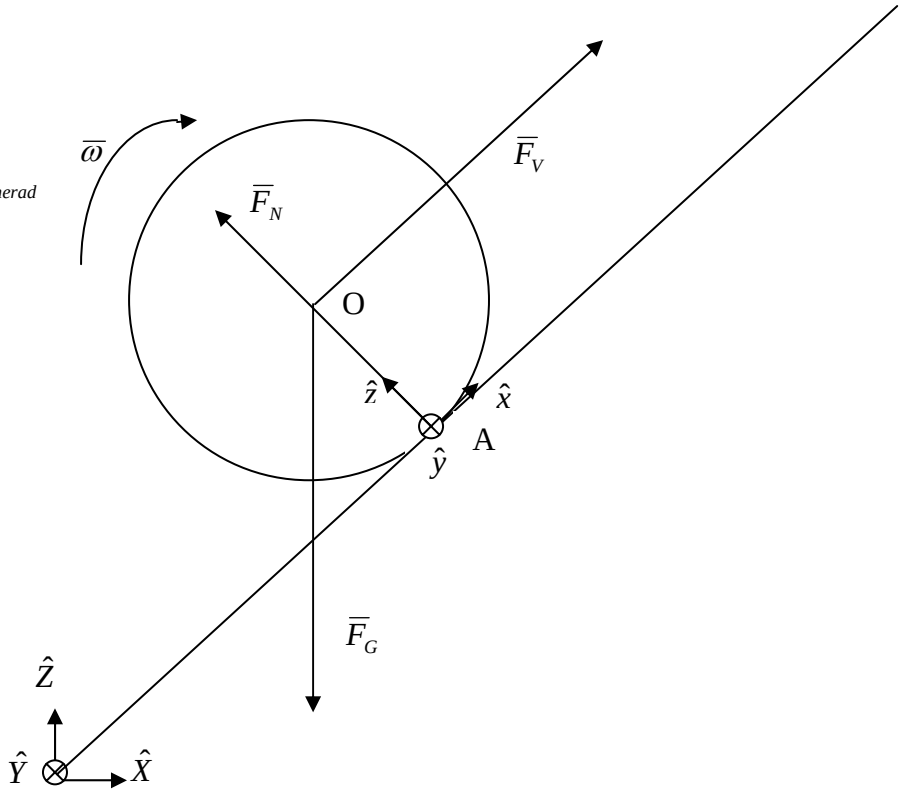
$\overline{F}_V = -\lambda_V \overline{V}$

$\overline{F}_G = mg\hat{Z}$

$\hat{z} = \hat{n}$

$\hat{x} = (1,0,-\frac{n_X}{n_Z})_{normerad}$

$\hat{y} = \hat{z} \times \hat{x}$




Friläggning ger följande moment vid A:

$\overline{F}_0 = \overline{F}_V + \overline{F}_G$

$\overline{M}_A = r \cdot \hat{z} \times \overline{F}_0 = r \cdot (\overline{F}_0 \cdot \hat{x}) \cdot \hat{y} - r \cdot (\overline{F}_0 \cdot \hat{y}) \cdot \hat{x}$

Masströghetsmomenten sett från A blir enligt:

$\overline{I}_x = \overline{I}_y = \overline{I}_z = \frac{2}{5} \cdot m \cdot r^2$

$\overline{x}, \overline{y}, \overline{z}$ är avståndet från A till O dvs. $\overline{x} = 0, \overline{y} = 0, \overline{z} = r$

$$I_x = \bar{I}_x + m(\bar{y}^2 + \bar{z}^2) = \frac{7}{5} \cdot m \cdot r^2$$
$$I_y = \bar{I}_y + m(\bar{x}^2 + \bar{z}^2) = \frac{7}{5} \cdot m \cdot r^2$$
$$I_z = \bar{I}_z + m(\bar{x}^2 + \bar{y}^2) = \frac{2}{5} \cdot m \cdot r^2$$

Vinkelhastighetsvektorn för koordinatsystemet i kontaktpunkten bestäms enligt:

$$\Omega = \frac{1}{2}\{\hat{x} \times \dot{\hat{x}} + \hat{y} \times \dot{\hat{y}} + \hat{z} \times \dot{\hat{z}}\}$$ (4.17 i Föreläsningsanteckningar, Christensen Peter)

Tecknas momentekvationen i A får vi följande rörelseekvationer:

$$\bar{M}_A = \left(\frac{d\bar{h}_A}{dt}\right)_{/xyz} + \bar{\Omega} \times \bar{h}_A, \qquad \bar{h}_A = \bar{I}_A \bar{\omega} = \begin{pmatrix} I_x & 0 & 0 \\ 0 & I_y & 0 \\ 0 & 0 & I_z \end{pmatrix} \begin{pmatrix} \omega_x \\ \omega_y \\ \omega_z \end{pmatrix} = \begin{pmatrix} I_x\omega_x \\ I_y\omega_y \\ I_z\omega_z \end{pmatrix} \Rightarrow$$

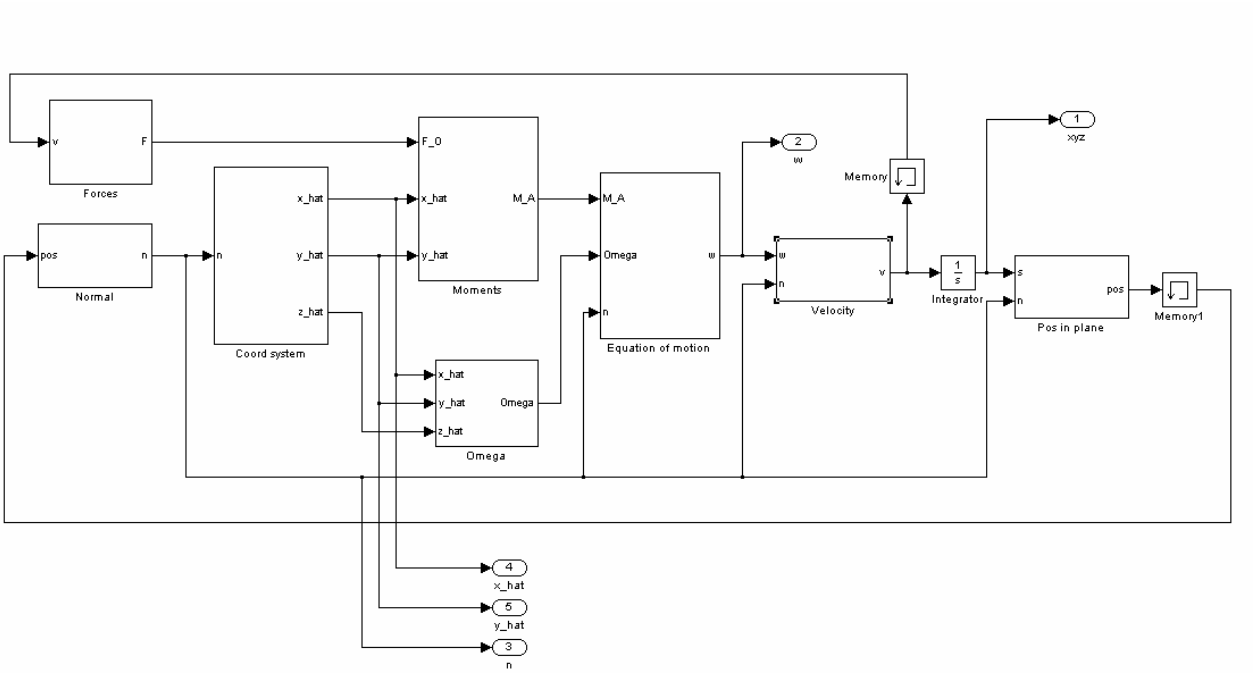
$$\Rightarrow \begin{cases} M_x = I_x\dot{\omega}_x + I_z\Omega_y\omega_z - I_y\Omega_z\omega_y \\ M_y = I_y\dot{\omega}_y - I_z\Omega_x\omega_z + I_x\Omega_z\omega_x \\ M_z = I_z\dot{\omega}_z + I_y\Omega_x\omega_y - I_x\Omega_y\omega_x \end{cases} \Rightarrow \begin{cases} \dot{\omega}_x = (M_x - I_z\Omega_y\omega_z + I_y\Omega_z\omega_y)/I_x \\ \dot{\omega}_y = (M_y + I_z\Omega_x\omega_z - I_x\Omega_z\omega_x)/I_y \\ \dot{\omega}_x = (M_z - I_y\Omega_x\omega_y + I_x\Omega_y\omega_x)/I_z \end{cases}$$

*Notera att $\bar{I}_A$ är konstant eftersom bollen är symmetrisk.

### 4.3.2 Implementering av den fysikaliska modellen

Vi har valt att implementera den fysikaliska modellen i Simulink i Matlab 6.5. Vi har valt att använda en tidskontinuerlig modell och utnyttja Simulinks ODE-lösare. De parametrar som man måste sätta innan simulering är bollens utslagsposition, radie, massa, hastighetsdämpningsfaktor, inledande rotationshastighet samt banans normaler. Modellen arbetar enligt följande: Med banans normaler och bollens position tar den fram koordinatsystemet vid bollens kontaktpunkt. Därefter räknar man fram vinkelhastighetsvektorn $\bar{\Omega}$ och momentet $\bar{M}_A$ i kontaktpunkten. Man tar sedan fram en rotationshastighet $\bar{\omega}$ och därefter en position.

Normalerna till ytan beräknas med en tabell genom att ta bollens position i planet projicerat i XY-planet $\hat{n} = f(P_{AX}, P_{AY})$. Bollens position i centrum $\bar{P}_O$ beräknas genom att integrera $\bar{V}_O$. Positionen i planet fås genom att dra bort radien från $\bar{P}_O$, $\bar{P}_A = \bar{P}_O - r\hat{n}$.




Utöver simulink-modellen har vi även en klubbfunktion. Dess värde beror på vinkeln och klubbans hastighet och gör så att bollen får den utgångshastighet den ska. Det finns även en studsfunktion. När bollen studsar i en vägg kommer vi att ta fram kontaktpunkten med väggen, uppdatera bollens position och dess rotationshastighet och sedan köra simuleringen igen. Bollens nya rotationshastighet kommer att ändras i väggens normalled då den multipliceras med studsfunktionens värde. Studsfunktionen beror på bollens rotationshastighet vid träffen.

## *4.4 Framtagning av rätt hastighet*

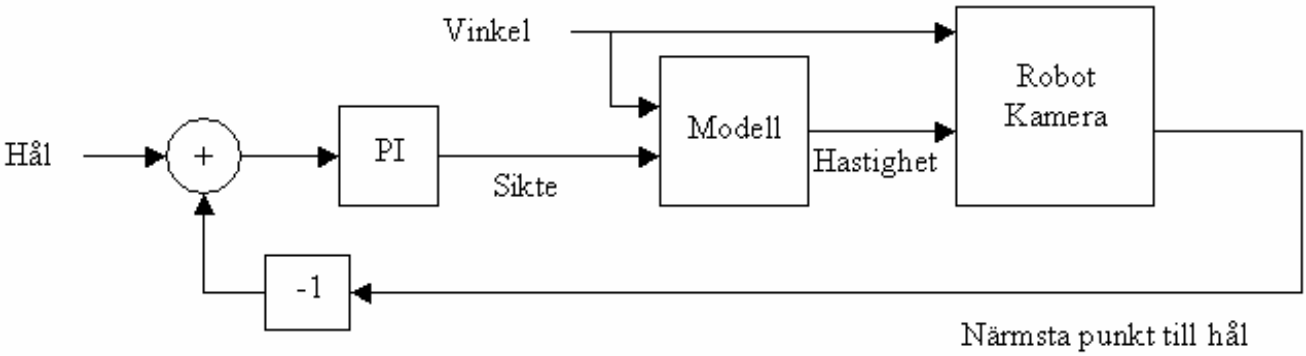
### 4.4.1 Förberäkning av bollbanor

Att göra en simulering med simulink-modellen går snabbt, ska man göra optimering baserat på många olika simuleringar tar det ändå ganska lång tid. Därför görs simuleringar i förhand med en önskad upplösning. En upplösning på 64*64 tar ca 15-20 minuter. För att minska storleken på de förberäknade bollbanorna sparas endast den övre delen av banan som är nära hålet. Dessutom sparas bara bollbanans projektion i XY-planet.

### 4.4.2 Direkt

När modellen får in en önskad vinkel kommer den att söka bland de olika hastigheterna för denna vinkel i de sparade banorna och räkna ut det kortaste avståndet till hålets centrum. Den kommer sedan att välja den hastighet som gör att bollen går i hålet. I vissa vinklar finns det två stycken olika hastigheter som går i hål, en utan studs och en med studs i någon vägg.

### *4.5 PI-regulator*




När roboten slår sitt första slag, siktar den givetvis på hålet. Vid en eventuell miss ser kameran hur nära hålet bollen gick. Genom att bilda felet i XY-planet kan ett nytt sikte beräknas med en PI-regulator. På detta sätt kan missar som beror av modellfel korrigeras på ett effektivt sätt.

$$sikte[0] = hål$$
$$I[0] = 0$$
$$fel = hål - närmsta\ punkt\ till\ hål$$
$$I[n] = I[n-1] + K_I + fel$$
$$sikte[n] = sikte[n-1] + K(fel + I[n])$$

## *4.6 Utvärdering och resultat*

### 4.6.1 Parameterskattning av fysikaliska modellen

De parametrar som vi skattade i modellen var hastighetsdämpningskonstant, studskonstanter mellan klubba och boll, samt mellan väggar och boll. Vi skrev ett program som plottade den uppmätta banan tillsammans med en samplad version av den simulerade banan. Med en samplad version av banan menas att vi samplade ner simuleringen så att kameran och den samplade versionen av banan hade samma samplingstid. Sedan anpassade vi studskonstanterna så att bollbanan fick samma utseende i rum och tid. Det visade sig att studskonstanten mellan klubba och boll inte blev konstant utan beror hyfsat linjärt med klubbans hastighet. En möjlig felkälla kan vara att klubbans hastighet inte är den som vi tror. Studskonstanten med väggarna visade sig bero på bollens rotationshastigheter vinkelrätt och parallellt med väggen samt vilken vägg det var. Vi fann att denna hade en övre och en undre gräns och var linjär däremellan. Hastighetsdämpningskonstanten fick vi till att vara 0.

Hastighetsdämpningskonstant = 0

Klubbfunktion = 0.00355*klubbhastighet-0.0018*vinkel+0.6789

Studsfunktionens värde ligger mellan 0.4 och 1.4 och har anpassats för hastigheter och vinklar som går nära hål. Då vi inte har en korrekt modell för studsarna var det också svårt att få en korrekt studsfunktion. Studsfunktionen vid den högre väggen är konstant 0.4. För vänster väggen ligger den mellan 0.4 och 1.4. Dess värde är högre ju högre andel av hastigheten som kommer vinkelrätt mot väggen jämfört med den som går parallellt. För toppväggen ligger den även där mellan 0.4 och 1.4. Denna beror enbart på hastigheten i x-led och är högre ju högre denna hastighet är.

### 4.6.2 Utvärdering av fysikaliska modellen

Den fysikaliska modellen har vi märkt stämmer väldigt bra med verkligheten. Det vi har sett är att när vi har en bollbana med en eller två studsar så blir överensstämmelsen med verkligheten sämre, dock fortfarande ganska god. Orsaken till detta tror vi beror på att vårt antagande att simulera studsen med en studskonstant inte stämmer med verkligheten. Vid en studs, speciellt med hög fart, studsar bollen upp en liten bit i luften samt glider, något vi inte tar hänsyn till i modellen.

### 4.6.3 Parameterskattning av PI-regulator

Efter ett flertal försök fann vi att följande parametrar gav ett önskat resultat:

P = 0.75

I = 0,25

### 4.6.4 Utvärdering av PI-regulator

PI-regulatorn fungerar bra i alla fall utom ett. Det är när bollen har hög hastighet och går över hål men inte i. Regulatorn kommer då inte reglera särskilt mycket utan vi kommer att slå i stort sett samma slag igen.

### 4.6.5 Förbättringar

Det finns ett par saker som man skulle kunna förbättra i modellen, en av dessa är att kontrollera om banans z-riktning är parallell med gravitationsfältet. En annan sak som skulle göra att modellen stämmer bättre med verkligheten är att införa glidning.

# 5 Robot

## *5.1 Inledande beskrivning av roboten*

Roboten är en industrirobot av modell ABB IRB1400. Den har sex axlar och därmed sex frihetsgrader. Roboten styrs av styrsystemet S4C och har programmerats med hjälp av ABB:s programspråk RAPID 3.0. Verktygen till roboten är utbytbara men under detta projekt har roboten varit bestyckad med en minigolfklubba samt en sugkopp vända 180 grader mot varandra.

## *5.2 Beskrivning av styrprogrammet till roboten*

Styrprogrammet som används är skrivet i RAPID 3.0 och heter GOLFPR05.PRG. Högst upp i programmet finns definitioner av alla arbetsobjekt, verktyg och målpunkter som används. Därefter kommer programmets alla procedurer. Dessa beskrivs här nedan i samma ordning som de står i programmet.

Samtliga positioner samt mått ska anges i millimeter och hastigheter i millimeter/sekund. (-) betyder att värdena på dessa inte spelar någon roll.

| **Kommando** | **Argument** | **Retur** | Förklaring |
|---|---|---|---|
| **hit_on_green** | **get_x (bollens x-position), get_y (bollens y-position), com_ok (=0)** | **com_ok (antar 1 om proceduren lyckats, 0 annars)** | Utför putt av boll på green från angiven bollposition (get_x, get_y). Bollen puttas i hål. |
| **m_hit_on_gr** | **get_x, get_y, put_x (bollens x-position), put_y (bollens y-position), height (bollens diameter), com_ok (=0)** | **get_x (bollens x-position efter proceduren), get_y (bollens y-position efter proceduren), com_ok** | Flyttar ut bollen några centimeter från kanten på green (från positionen put_x, put_y) med hjälp av sugkoppen. Sedan anropas hit_on_green (med returnerade get_x, get_y) för att putta bollen i hål från den nya positionen. |

| | | | |
|---|---|---|---|
| **p_hit_on_tee** | **speed (önskad utgångshastighet), hit_angle (önskad utslagsvinkel), get_x, get_y, com_ok (=0)** | **com_ok** | Flyttar klubban till det läge där svingen för utslag på tee börjar. Denna procedur måste alltså anropas innan hit_on_tee anropas. |
| **hit_on_tee** | **speed (önskad utgångshastighet), hit_angle (önskad utslagsvinkel), get_x, get_y, com_ok (=0)** | **sensor2 (antar 1 om bollen gick i hål på tee, 0 annars), com_ok** | Utför utslag av bollen på tee med angiven vinkel och hastighet. |
| **sug_green** | **get_x, get_y, get_z, put_x, put_y, put_z (tidigare, resp. önskad x-, y- och z-position), height (bollens diameter), com_ok (=0)** | **com_ok** | Utför olika saker beroende på vad get-variablerna är. Om bollen ligger i hålet på green utförs flyttning av bollen från hålet till önskad position (put-variablerna). Om bollen ligger någon annanstans på green utförs flyttning av bollen från denna position till önskad position (put-variablerna). Height-variabeln används inte då bollen ligger i hålet. |
| **place_ball** | **get_x, get_y, get_z, put_x, put_y, put_z (tidigare, resp. önskad x-, y- och z-position), height (bollens diameter), com_ok (=0)** | **com_ok** | Utför olika saker beroende på vad get-variablerna är. Om bollen ligger på green anropas proceduren sug_green. Om bollen ligger någon annanstans flyttas bollen från denna position till önskad position (put-variablerna). |
| **sweep** | | **com_ok** | Utför fösning av boll på den del av banan där kameran inte kan ”se”, d.v.s. på den |

| | | | |
|---|---|---|---|
| | | | plana delen av tee. |
| **read_data** | **comfile, cflag, dfile (namn på filen som ska läsas), command, angle, speed, get_x, get_y, get_z, put_x, put_y, put_z, height** | **command, angle, speed, get_x, get_y, get_z, put_x, put_y, put_z, height** | Läser 10 rader i angiven datafil (dfile) och sparar dessa i variablerna som ges som utargument. |
| **rob_action** | **comfile, dfile, sensor, sensor2, command, angle, speed, get_x, get_y, get_z, put_x, put_y, put_z, height, com_ok** | **com_ok** | Beroende på vad command-variabeln är utförs olika procedurer med de inargument som behövs till respektive procedur. |
| **write_data** | **comfile, dfile (namnet på datafilen vilken ska skrivas till), sensor, sensor2, com_ok** | **-** | Sparar värdet från sensorn i rännan i variabeln sensor. Skriver sedan in värdena på sensor, sensor2 och com_ok på de tre första raderna i angiven datafil (dfile). |
| **check_ready** | **comfile, cflag (namnet på datafilen vilken ska undersökas) , ready** | **ready (antar falskt eller sant)** | Undersöker om angiven datafil (cflag) existerar. Om den gör det returneras sant i variabeln ready, annars falskt. |
| **get_sensordata** | **comfile, dfile (namnet på datafilen vilken ska skrivas till), sensor, sensor2** | **-** | Sparar värdet från sensorn i rännan i variabeln sensor. Skriver sedan in värdena på sensor och sensor2 på andra och tredje raden i angiven datafil (dfile). |
| **main** | | | Huvudproceduren startas då GOLFPR05.PRG exekveras. |

### *5.3 Kommunikation mellan robot och huvudenhet*

Kommunikationen mellan robot och huvudenhet sker i nedanstående sekventiella ordning.

1. Huvudenheten skapar filen data.dat med följande variabler på raderna: command, angle, speed, get_x, get_y, get_z, put_x, put_y, put_z, height. Dessa kommer att användas i roboten enligt kapitel 5.2.

2. Huvudenheten skapar filen commandflag.dat som en flagga på att roboten ska ta emot order.

3. Roboten läser in variablerna som står i filen data.dat.

4. Roboten utför ordern.

5. Roboten skriver över filen data.dat med följande variabler på raderna: sensor, sensor2, com_ok.

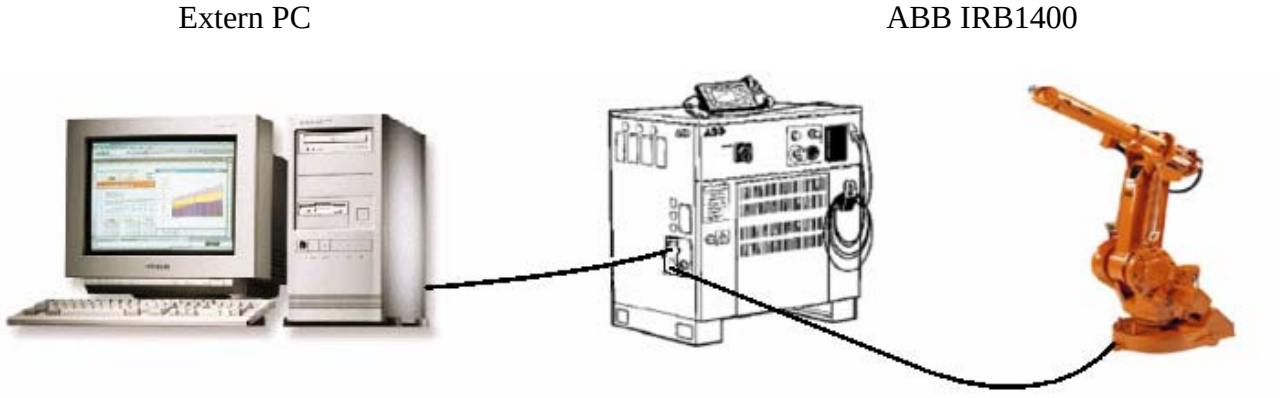
6. Roboten skapar filen robotflag.dat som en flagga på att roboten har utfört ordern.

7. Huvudenheten raderar filerna data.dat, commandflag.dat och robotflag.dat.

### *5.4 Kalibrering av robot*

För att programmet till roboten ska fungera måste tre verktyg vara definierade. Dessa är verktyget *spik*, verktyget *club* samt verktyget *club2*. *Spik* ska vara definierad så att dess *tool center point* är spetsen på spiken. *Club* ska vara definierad så att dess *tool center point* är i den bakre utritade punkten på undre delen av klubban. *Club2* ska vara definierad så att dess *tool center point* är i den främre utritade punkten på undre delen av klubban.

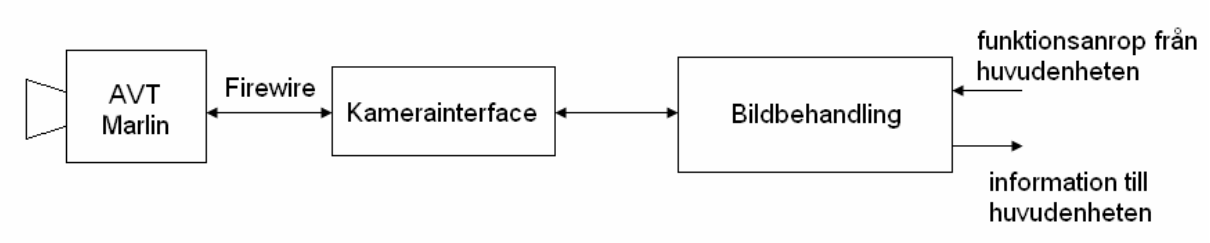
Även banan måste vara definierad för att programmet ska fungera. Denna ska vara definierad som koordinatsystemet *bana2*. På banans kant finns tre stycken skruvar som används för att definiera/kalibrera banan. Skruven i banans nedre högra hörn motsvarar origo och används som punkt x1. Som punkt x2 används skruven i banans högra kant och som punkt y1 används skruven i banans bakkant. Då man ska mäta skruvarnas lägen används lämpligen spiken som verktyg.




*Figur 3. Robot med styrsystem*

# 6 Kamera och bildbehandling

Systemet är utrustat med en digitalkamera av typen AVT Marlin F-145C2. Denna enhet ger information om slaget som huvudenheten ska kunna använda för att förbättra robotens slag. Den identifierar var bollen är på banan så att roboten kan fortsätta slå tills bollen går i hålet på green.




*Figur 4. Översikt bildbehandling och kamera*

Bildbehandlingsenheten får via funktionsanrop förfrågningar från huvudenheten att returnera bollens position eller att leverera en uppsättning koordinater som beskriver bollens bana i x-, y- och z-led i det senaste slaget.

## *6.1 Kamerainterface*

### 6.1.1 Översikt

Kamerainterfacet är en fristående Matlabapplikation som kan användas till att ställa in kameraparametrar, starta och stoppa kameran, samt hämta in bilder som matriser till Matlab.

Det bygger på ett klassbibliotek som följer med kameran kallat FireClass. Biblioteket anropar även en medföljande drivrutin för att kommunicera med kameran via Firewireporten.

### 6.1.2 Egenskaper

Kamerainterfacet har en hel del möjliga inställningar och operationsmöjligher:

- Matlabanvändning parallellt, eftersom alla operationer körs i egna trådar.
- Start och stopp vid valfri tidpunkt.
- Intern bildbuffer med tidsstämpling. Detta gör det möjligt att ha en långsam bildhantering i Matlab utan att riskera att förlora bilder.
- Inställningsbar bildbuffer. Man kan ställa in antalet bilder som ska få plats i buffern, så datorer med olika minnesstorlekar kan användas.
- Två buffermoder. Antingen kan man låta buffern fyllas och sen inte ta in fler bilder, eller så kan man ha en cirkulär buffer som skriver över de gamla bilderna om man vill.
- Snapshot. Man får in den senaste bilden i buffern till Matlab.
- Svartvita eller färgbilder.
- Flera olika bildstorlekar och inhämtningshastigher. Dessa följer DCAM-standardens typer.

- Skalbar bildmod. I denna mod kan man själv ställa in bildstorlek och bildposition. Inhämtningshastigheten beror då på bildstorlek och kan inte väljas själv, dock kan man komma upp i väldigt höga hastigheter vid små bilder (max ca 33 bilder/sekund). Finns både i färg och svartvitt.
- Två inställningstyper för vissa kameraparametrar (slutartid, gamma mm), en avsedd för långsam inhämtning utan rörelser och en för snabba rörelser i bilden. Dessa två typer ska inte blandas samman med bildstorlek och inhämtningshastighet.
- Loggningsmöjligheter för felsökning.

### 6.1.3 Användning

Man anropar interfacet genom att skriva *CameraInterface('kommando', argument)* i Matlabfönstret. Om inget argument ges, blir argumentet implicit 0 (även om den inte används). Ett enkelt användarfall ser ut ungefär så här:

1. Se till att Matlab hittar CameraInterface, antingen genom att gå till den katalogen eller genom set Path.
2. Starta Firewirebussen genom att skriva:
   *CameraInterface('startbus',0)*
3. Sätt in kabeln i kameran (om den inte redan är insatt). Vänta tills bussen har hittat kameran. Du ser det genom att den orangea lampan blinkar till, eller om du skriver
   *CameraInterface('printbufferinfo')*
   Om du får en massa felmeddelanden så har den inte hittat kameran. Vänta och försök sen igen.
4. Starta inhämtningen från kameran:
   *CameraInterface('startacq')*
5. För att hämta in en bild till Matlab, använd ett av följande kommandon:
   *im = CameraInterface('nextimage');*
   *im = CameraInterface('getimage');*
   *im = CameraInterface('getsnapshot');*
6. Visa en färgbild:
   *image(im);*
   Svartvit bild:
   *imagesc(im);*
7. Stoppa inhämtning från kameran genom:
   *CameraInterface('stopacq');*

Inställning av bilderna kan se ut att vara komplicerat, men blir genast enklare om man delar upp det i två fall, icke-skalbara inställningar och skalbara inställningar. För att kunna ändra bildinställningar så måste kamerans inhämtning vara stoppad.

De icke skalbara moderna är enkla att ställa in och man får alla inställningar (bildstorlek, färg och hastighet) direkt. Man ställer in kameran genom att skriva

*CameraInterface('setvidset',vidsetnr);*

Där *vidsetnr* är ett nummer (0-26) som svarar mot den önskade inställningen (se tabell nedan). Man kan inte göra några ytterligare inställningar.

När man använder de skalbara moderna (*vidsetnr* 27 eller 28) kan man ändra bildstorlek hämta in valfri (rektangulär) del av bilden. Man gör detta genom att först lägga kameran i mod 27 eller 28 och sedan

*CameraInterface(setxpos',xpos);*

*CameraInterface(setypos',ypos);*

*CameraInterface(setxsize',xsize);*

*CameraInterface(setysize',ysize);*

Det är inte säkert att man får ut den exakta storleken och positionen, detta är kameraberoende. För AVT Marlin F145B2 är den maximala bildstorleken 1392 i x-led och 1038 i y-led.Även dessa inställningar måste göras när kameran inte hämtar in bilder.

Här följer en fullständig lista över kommandon som fungerar i Matlab. Man anropar dessa genom att skriva *retur=CameraInterface('kommando', argument);*

Om man utelämnar *argument* tolkas det som 0 om det behövs användas alls. *Retur* kan alltid utelämnas.

| **Kommando** | **Argument** | **Retur** | Förklaring |
|---|---|---|---|
| **startbus** | **nummer** | - | Buss 0 är första, och oftast enda, Firewirebussen. |
| **printbusinfo** | - | - | Skriver ut diverse info om bussen. |
| **printbufferinfo** | - | - | Skriver ut antal bilder i buffern / max antal bilder. |
| **getbufferusage** | - | **usage** | Returnerar antal bilder det finns i buffern. |
| **startacq** | - | - | Startar kameran. |
| **stopacq** | - | - | Stoppar kameran. |
| **nextimage** | - | **[im, time]** | Returnerar nästa bild från buffern (**im**) och dess tidsstämpel om **time** har angetts. |
| **getsnapshot** | - | **[im, time]** | Returnerar den bild som senast hämtats från kameran. |
| **getimage** | - | **[im, time]** | Hämtar om samma bild som senast hämtats med **nextimage** eller **getsnapshot**. |
| **gettime** | - | **time** | Hämtar den senast inhämtade bildens tidsstämpel. |
| **clearbuffer** | - | - | Tömmer buffern på alla sina bilder. |
| **setbuffersize** | **size** | - | Sätter bufferstorleken till **size** antal bilder. |
| **enablelog** | - | - | Startar loggning av vad systemet håller på med. |
| **disablelog** | - | - | Stoppar loggningen. |

| | | | |
|---|---|---|---|
| **printfeatures** | - | - | Skriver ut de kameraparametrar som kameran har. |
| **setstandardfeatures** | - | - | Ansätter kameraparametrar optimerat för snabba rörelser. |
| **setcalibrationfeatures** | - | - | Ansätter kameraparametrar för bra ljus och kontrast. |
| **getcolordepth** | - | **colors** | Returnerar färgdjupet , 1 för gråskala, 3 för RGB. |
| **getscalablemode** | - | **mode** | Returnerar 1 om skalbar mod används, 0 annars. |
| **setvidset** | **vidset** | - | Ansätter önskad videoinställning. |

Är man i skalbar mod kan dessutom dessa kommandon användas:

| **Kommando** | **Argument** | **Retur** | Förklaring |
|---|---|---|---|
| **setxsize** | **Xsize** | - | Ansätter önskad X-storlek. |
| **setysize** | **Ysize** | - | Ansätter önskad Y-storlek. |
| **setxpos** | **Xpos** | - | Ansätter önskad X-position. |
| **setypos** | **Ypos** | - | Ansätter önskad Y-position. |
| **getxsize** | - | **Xsize** | Returnerar faktiskt X-storlek. |
| **getysize** | - | **Ysize** | Ansätter X-storlek. |
| **getxpos** | - | **Xpos** | Ansätter X-storlek. |
| **getypos** | - | **Ypos** | Ansätter X-storlek. |

Följande möjliga videoinställningar kan användas genom *Camerainterface('setvidset',nr);*

| **Nr** | FPS | Format | storlek |
|---|---|---|---|
| **0** | 3.75 | YUV-422 | 320*240 |
| **1** | 7.5 | YUV-422 | 320*240 |
| **2** | 15 | YUV-422 | 320*240 |
| **3** | 3.75 | YUV-411 | 640*480 |
| **4** | 7.5 | YUV-411 | 640*480 |
| **5** | 15 | YUV-411 | 640*480 |
| **6** | 3.75 | YUV-422 | 640*480 |
| **7** | 7.5 | YUV-422 | 640*480 |
| **8** | 15 | YUV-422 | 640*480 |
| **9** | 3.75 | Y-8 | 640*480 |
| **10** | 7.5 | Y-8 | 640*480 |
| **11** | 15 | Y-8 | 640*480 |

| | | | |
|---|---|---|---|
| **12** | 3.75 | YUV-422 | 800*600 |
| **13** | 7.5 | YUV-422 | 800*600 |
| **14** | 15 | YUV-422 | 800*600 |
| **15** | 7.5 | Y-8 | 800*600 |
| **16** | 15 | Y-8 | 800*600 |
| **17** | 3.75 | YUV-422 | 1024*768 |
| **18** | 7.5 | YUV-422 | 1024*768 |
| **19** | 3.75 | Y-8 | 1024*768 |
| **20** | 7.5 | Y-8 | 1024*768 |
| **21** | 1.875 | YUV-422 | 1280*960 |
| **22** | 3.75 | YUV-422 | 1280*960 |
| **23** | 7.5 | YUV-422 | 1280*960 |
| **24** | 1.875 | Y-8 | 1280*960 |
| **25** | 3.75 | Y-8 | 1280*960 |
| **26** | 7.5 | Y-8 | 1280*960 |
| **27** | - | YUV-411 | Max 1392*1038 |
| **28** | - | Y-8 | Max 1392*1038 |

Bildformatet Y-8 svarar mot en svartvit bild och YUV-411 samt YUV-422 svarar mot färgbilder. Båda dessa konverteras till en RGB-färgbild i Matlab.

### 6.1.4 FireClass

Det medföljande programpaketet FireClass innehåller alla nödvändiga instruktioner för att kommunicera med en kamera som följer DCAM-standarden, dock kräver den att man installerar en medföljande drivrutin.

Det medföljer även några klasser för att underlätta utveckling av ny programvara. Bland dessa är det främst CFireBus, CFireNode och CCamera som är viktiga. CFireBus tar hand om alla händelser som sker på Firewirebussen, CFireNode är en abstraktion av alla enheter på bussen. CCamera är en underklass till CFireNode som tar hand om alla DCAM-kompatibla kameror som kopplas till bussen. Övriga medföljande klasser och funktioner snabbar på t.ex. bildformatkonvertering. Förklaringar till klasserna finns i [6].

### 6.1.5 CameraInterface

De klasser som har utvecklats följer nedan.

| | |
|---|---|
| **CCameraInterface** | Denna klass har som uppdrag att starta och stoppa arbetstråden. Om man har flera FireWirebussar tar den även hand om byte av buss. |
| **CWorkThread** | Denna tråd tar hand om meddelanden från Firewirebussen som har med kameran att göra. Fyra sorters meddelanden tas omhand; Ny enhet, borttagen |

| | |
|---|---|
| | enhet, ny bild, samt fel. |
| **CCameraThread** | Varje kamera har en egen tråd kopplad till sig där alla beräkningar för deras bilder sker. |
| **CFrameBuffer** | Denna klass vidarebefordrar inställningarna till kameran och mottagning av bilder från kameran. Utöver detta så tidsstämplas varje bild när den kommer in, samt konverterar den till valt bildformat. Bilden läggs sedan in i buffern tillsammans med sin tidsstämpel. |
| **MexStart** | Egentligen ingen klass, utan interfacet till Matlab. Tar hand om alla kommandon som inhämtas från Matlabanvändaren. Utför även eventuell loggning. |

## *6.2 Kalibrering av kamera*

Kameran kan kalibreras automatiskt med funktionen `autoCalibr` eller manuellt genom att skriva `manuellKalibrering`. Följande MATLAB-funktioner används vid kalibrering av denna del av systemet.

### 6.2.1 Funktioner för kalibrering

```
[Cf, Cg, coords, holepos, intStuff]= autoCalibr(img)
```

**Inparametrar**

`img` - En bild av banan som en MxNx3 matris med RBG-lagren i den tredje dimensionen.

**Returvärde**

`Cf` - En 4x3 matris som beskriver hur punkter på fairway i världen förhåller sig till punkter i bilden.

`Cg` - En 3x3 matris som beskriver hur punkter i bilden och punkter på green i världen förhåller sig till varandra.

`coords` - uv-koordinater som används för att skala bilden från kameran när den ska ta en bildsekvens.

`holepos` – Positionerna på de två hålen på banan.

`intStuff` - Interna parametrar och data som används av olika bildbehandlingsalgoritmer (findBall, image2coord).

**Beskrivning:** Funktionen anropar `findRefPoints` för att få koordinater för referenspunkterna. Om detta lyckades och användaren godkänner referenspunkterna anropas funktionen `calibrSyst` för att kalibrera bildbehandlingsdelen av systemet.

```
[Cf, Cg, coord,holepos, intStuff]=manualCalibr(calImg)
```

**Inparametrar**

Se funktionen autoCalibr

**Returvärde**
Se funktionen autoCalibr

**Beskrivning:** Användaren får med musen klicka på de 23 referenspunkterna manuellt. Då detta är klart anropas `calibrSyst` för att kalibrera bildbehandlingsdelen av systemet. Denna funktion anropas lämpligen med scriptet `manuellKalibrering` för att slippa bry sig om inparametrar och returvärden till funktionen.

```
refPoints=findRefPoints(img)
```

**Inparametrar**
`img` - En bild av banan antingen som en MxNx3 matris med RGB värden i de tre lagern i 3:e dimensionen.

**Returvärde**
`refPoints` - Bildkoordinaterna [u v] för de funna referenspunkterna.

**Beskrivning:** Funktionen letar reda på de 23 referenspunkterna för systemet. Först trösklas 8 st typiska punkter fram ur RBG-värdena. Resultatet av dessa trösklingar läggs sedan ihop och detta resultat ”lågpassfiltreras” och trösklas ännu en gång så att det i en 3x3 omgivning måste finnas minst 5 träffar från trösklingen för att det ska räknas som en blå punkt. Roboten, det svarta staketet och banans gröna matta trösklas också fram med lämpliga RBG-trösklar. Resultatet av trösklingen för roboten expanderas sedan vertikalt och åt vänster. Staketet expanderas åt vänster och den gröna mattan expanderas ett par steg med 8-konnektivt strukturelement. Dessa områden läggs sedan ihop och inverteras. Sedan genomförs en logisk OR mellan detta resultat och resultatet av trösklingen av blå punkter så att alla eventuella blå punkter i staketet, på roboten och i mattan försvinner. Resultatet av denna OR operation tunnas sedan konnektivitetsbevarande med 8-konnektivitet så att endast en punkt i varje område blir kvar. Dessa punkter antas vara systemet referenspunkter.

```
[Cf, Cg, coords, holepos, intStuff]=calibrSyst(refPoints, img)
```

**Inparametrar**
`refPoints` - Systemets referenspunkter givna på formen

Övriga inparametrar se funktionen `autoCalibr`.

**Returvärde**
Se funktionen `autoCalibr`.

**Beskrivning:** Funktionen sorterar först referenspunkterna så att de kommer i rätt ordning. I funktionen finns också de uppmätta positionerna för referenspunkterna lagrade. Dessa paras ihop och skickas till `cameraCalibr` för att erhålla de två matriserna `cf` och `cg`. De två områdena där systemet ska kunna detektera bollen beräknas och masker för dessa lagras i `intStuff`. Bakgrundsbilden som ska användas för att detektera bollar på greenen lagas också i `intStuff`.

```
C = camera_calibr(points,u,v)
```

**Inparametrar**

`points` - Matris `[X Y]` eller `[X Y Z]` med punkter i världen där `X`,`Y` och `Z` är kolumnvektorer.

`u` - Vektor med u-värden från punkter i bilden.

`v` - Vektor med v-värden från punkter i bilden.

**Returvärde**

`C` - En 3x3 eller 3x4 matris (beroende på om `[X Y]` eller `[X Y Z]` används) som beskriver hur punkterna i världen och punkter i bilden förhåller sig till varandra.

**Beskrivning:** Funktionen beräknar utifrån de givna punkterna i bilden och i världen en matris som beskriver hur punkterna i bilden och världen förhåller sig till varandra.

## *6.3 Detektering av boll*

Bollens position kan detekteras i de områden på fairway och green som innesluts av referenspunkterna på respektive område. I båda fallen jämförs en bild där bollen kan vara i dessa områden med bakgrundsbilderna som finns lagrade i `intStuff`.

### 6.3.1 Transformation av koordinater från bilden till rummet

Transformationen av koordinater från rummet $(x, y, z)$ till bilden $(U, V)$ kan generellt beskrivas med en 4x3 matris, [8] . Om man antar att världen är platt (vilket görs på green) så reduceras matrisen till en 3x3 matris. Detta gör att följande samband gäller

$$\begin{bmatrix} u & v & t \end{bmatrix} = \begin{bmatrix} x & y & z & 1 \end{bmatrix} \mathbf{C}_f$$
$$\begin{bmatrix} u & v & t \end{bmatrix} = \begin{bmatrix} x & y & 1 \end{bmatrix} \mathbf{C}_g$$

och punkterna i bilden ges av

$$U = \frac{u}{t} \qquad V = \frac{v}{t}$$

I fallet för green så kan 3x3 matrisen inverteras och punkter i rummet kan fås som

$$\begin{pmatrix} \frac{x}{t} & \frac{y}{t} & \frac{1}{t} \end{pmatrix} = \begin{pmatrix} U & V & 1 \end{pmatrix} \mathbf{C}_g^{-1} \Rightarrow \begin{cases} x = \frac{x/t}{1/t} \\ y = \frac{y/t}{1/t} \end{cases}$$

Denna beräkning av rumskoordinaterna är inte exakt utan medför vissa fel. För att försöka minimera dessa fel genomförs en linjär kompensering av kompensering enligt

$$\begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = \begin{bmatrix} kompX1 & kompX2 \\ kompY1 & kompY2 \end{bmatrix} \begin{bmatrix} U \\ V \end{bmatrix}$$

Detta räcker dock inte för sugkoppen som bland annat används för att flytta ut bollen från kanten när roboten ska putta. Detta eftersom sugkoppen måste sluta tätt kring bollen för att roboten ska lyckas suga upp bollen. När bollen ligger nära en kant på green genomförs därför även en korrigering som beror på vilken kant som bollen ligger närmast.

För fairway kan man inte göra förenklingen att världen är platt. Detta medför att man inte kan beräkna rumskoordinaterna genom att invertera avbildningsmatrisen enligt ovan. Rumskoordinater på fairway kan endast estimeras för den del av banan som innesluts av de blå referenspunkterna. På denna yta finns rumskoordinater utspridda med ett mellanrum på 50 mm i x- och y-led lagrade i systemet. Dessa punkter transformeras vid kalibreringen av systemet till punkter i bilden enligt sambandet ovan. Vid estimering av rumskoordinater från koordinater i bilden jämförs sedan vilken av de till bilden transformerade punkterna som ligger närmast. Den uppsättning rumskoordinater som transformerade till den närmaste punkten får sedan utgöra bollens position i rummet. En linjär korrigering motsvarande den som genomförs på greenen genomförs även på fairway.

Då regleringen använder den punkt på bollbanan som är närmast hål har stor vikt lagts vid att få bra estimat på bollens bana kring hålet. Därför finns en offsetjustering av rumskoordinaterna så att detta uppfylls.

### 6.3.2 Funktioner för detektering av boll

```
[u, v]= findBall(area, frame, intStuff)
```

**Inparametrar**

`area` - Anger i vilket området på banan som det ska sökas efter boll 'fairway' eller 'green'.

`frame` - En bildruta ur en slagsekvens (för fairway) eller en stillbild (för green).

`intStuff` - Intern struktur som innehåller parametrar för att kunna detektera bollen. Denna fås som returvärde från funktionen calibr.

**Returvärde**

`u` - u-koordinaten för den detekterade bollen i bilden.

`v` - v-koordinaten för den detekterade bollen i bilden.

**Beskrivning:** Givet en bild och ett område där bollens position ska detekteras returnerar denna funktion koordinaterna i bilden där bollen eventuellt har hittats. Den inkommande bilden maskas och differensen mellan den maskade bilden och motsvarande bakgrund som finns lagrad i `intStuff` beräknas. Denna differens trösklas och ”lågpassfiltreras” så att en 3x3 omgivning måste ha minst 6 vita pixlar för att räknas som att bollen hittats. Medelvärdet i u- och v-led beräknas på det vita området och tas som bollens position.

```
[x, y, z] = image2coord(u,v,C)
```

**Inparametrar**
`u` - u-koordinat i bilen.

`v` - v-koordinat i bilden.

`C` - Matris som beskriver sambandet mellan punkter i bilden och punkter på någon av banans delar banan (fairway) 4x3 och (green) 3x3.

**Returvärden**
`x` - x-koordinat i världen

`y` - y-koordinat i världen

`z` - z-koordinat i världen

**Beskrivning:** Transformerar en punkt i bilden till en punkt i rummet. För mer detaljerad beskrivning hur det går till se avsnittet transformation av koordinater från bilden till rummet.

### *6.4 Övriga funktioner som används av bildbehandlingsdelen*

Följande funktioner används i de flesta av bildbehandlingsenhetens funktioner.

```
imgOut = reArrangeImg(imgIn)
```

**Inparametrar**
`imgIn` - Bild från kameran som ska vändas som en NxMx3 (färgbild) eller en MxN matris (svartvit)

**Returvärde**
`imgOut` - Den rättvända bilden.

**Beskrivning:** Funktionen vänder bilden från kameran till rätta så den stämmer bättre med verkligheten.

```
[out,num] = thin8(in, iter)
```

**Inparametrar**
`in` - Den binära bild som ska krympas.

`iter` - Antalet iterationer. Om iter = 0 körs algoritmen tills den konvergerar.

**Returvärde**
`out` - Den krympta bilden.

`num` - Antalet iterationer som genomförts.

**Beskrivning:** Funktionen tar en binär bild och genomför konnektivitetsbevarande krympning med ett 8 konnektivt strukturelement antingen ett visst antal iterationer eller tills endast en pixel i varje objekt finns kvar.

```
exp = expand(img, conn, n)
```

**Inparametrar**
`img` - Bilden som ska expanderas

`conn` - Anger vilken ”konnektivitet” som ska användas.

| | |
|---|---|
| 4 | 4 konnektiv expansion |
| 8 | 8 konnektiv expansion |
| -1 | Endast vertikal expansion med kärnan ones(11,1) |
| -2 | Endast horisontell expansion med kärnan ones(1,5) |
| -3 | Endast expansion åt vänster |
| -4 | Endast expansion åt höger |
| -5 | Endast expansion uppåt |

`n` - Antalet iterationer i expansionen, måste vara ett positivt heltal

Returvärde
`exp` - Den expanderade bilden.

**Beskrivning:** Givet en binär bild expanderas denna `n` gånger givet konnektiviteten i `conn`.

# 7 Resultat av projektet

Resultatet av projektet är först och främst det självklara; ett fungerande system med en ABB-robot som kan slå en boll i håll på en minigolfbana. Glädjande är att modellen av bollens rörelse med hjälp av kameran gick att justera in så pass bra.

## *7.1 Möjligheter till fortsatt utveckling*

Systemet är i sin helhet uppbyggt av moduler. Om man i framtiden skulle vilja förbättra någon del för sig så skulle detta inte vara några större svårigheter att genomföra utan att ändra hela systemets funktionalitet. Som exempel finns förslag på förbättringar i modellen i avsnitt 4.6.5. Andra exempel på fortsatt utveckling skulle kunna vara att köpa in programvara för att kunna bestämma exakt hastighet på robotens verktyg. Man skulle också kunna tänka sig att utveckla stöd för två kameror antingen för att använda en kamera för green och en för fairway, eller kanske utveckla någon form av stereoseende. Detta för att kunna bestämma bollens position mer exakt.

# Referenser

[1] Svensson, Tomas & Krysander, Christian (2002), *LIPS – nivå 1*. Bokakademin, version 1.0.

[2] Enqvist, Martin (2005), *Projektdirektiv – Golfspelande industrirobot med kamera*.

[3] Tjäder, Mats (2005), *Kravspecifikation – Golfspelande industrirobot med kamera*.

[4] Tjäder, Mats (2005), *Systemskiss – Golfspelande industrirobot med kamera*.

[5] Tjäder, Mats (2005), *Designspecifikation – Golfspelande industrirobot med kamera*.

[6] Zhengyou Zhang: *A Flexible New Technique for Camera Calibration*, Technical Report, MSR-TR-98-71, 1998, Microsoft Research, Microsoft Corporation, One Microsoft Way, Redmond, WA 98052, http://research.microsoft.com/~zhang

[7] Allied Vision Technologies (2004), *Documentation FireClass (CFireBus, CFireNode, CCamera)*.

[8] Forcheimer, et al. Bilder och grafik, TSBB65 2004 del 2, Institutionen för Systemteknik