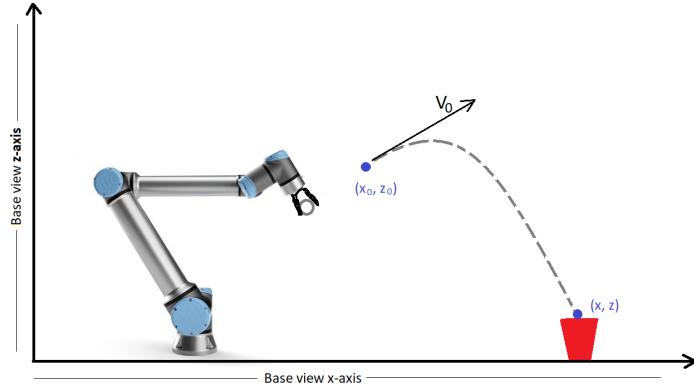


# SEMESTERPROJEKT I

## SIGNALER OG ROBOTSYSTEMER

Udvikling af en kasterobot



### Projektgruppe 5

Alex Ellegaard - ae11e20

Maxim Milenkovic - mamil20 Simon Christensen - simch20

Thomas Therkelsen - ththe20 Victoria Jørgensen - vijoe20

### Vejleder

Thorbjørn Mosekjær Iversen



SYDDANSK | UNIVERSITET

Diplomingeniør Robotteknologi

TEK MMMI

Syddansk Universitet

21/12-2021

## Abstract

This report showcases the process of programming a UR5-series industrial robotic arm to throw a ping-pong ball into a cup.

First, the report introduces the Machine Vision section of the project. Contained in this is both: Computation and correction of lens and image distortion, as well as calibrating the camera's coordinate frame to match the table coordinate frame to ascertain precision, and lastly identifying and locating both the target and projectile for further use in the program, by utilizing specifically the OpenCV C++ library.

Furthermore, the calibration from robot to table coordinate frame and the Physics part of the project is elucidated. Included in that, is the modeling of the throw, as well as transforming that information into a usable movement curve that the robot can move along, which is accomplished utilizing the Jacobian matrix.

In conclusion, it's addressed why SQLite was chosen as the database engine in this project, as well as how it ended up being implemented and used in the main program.

## Forord

Heraf fremgår det i prioriteret rækkefølge hvad det enkelte gruppemedlem primært har ydet og bidraget med til projektet.

Alex og Simon har først og fremmest arbejdet på kamera kalibrering og generelt Machine Vision delen af projektet.

Thomas og Victoria har primært opsat og testet UR RTDE-biblioteket på både UR Simulatoren og på den fysiske robot. Ydermere har Victoria stået for robotkalibreringen

Gruppen har sammen arbejdet på modellering og implementering af kastet.

Thomas og Maxim har arbejdet på database og interface mellem hovedprogrammet og databasen.

Slutteligt har gruppen sammen arbejdet på en sammenkobling af programmerne til et hovedprogram og om udviklingen af denne rapport.

Alex Ellegaard



Alex ME

Maxim Milenkovic



Maxim

Thomas Therkelsen



Thomas T

Simon Bork



Simon B.C.

Victoria Jørgensen



VJ

## Indholdsfortegnelse

<b>1 Indledning</b>	<b>6</b>
1.1 Problemformulering . . . . .	6
1.2 Krav Specifikation . . . . .	6
1.3 Rapportens struktur . . . . .	7
<b>2 Hardware</b>	<b>8</b>
<b>3 Machine Vision</b>	<b>9</b>
3.1 Teori . . . . .	9
3.1.1 Kamera kalibrering . . . . .	9
3.1.2 Homografi . . . . .	9
3.1.3 Hough Circle Transform . . . . .	10
3.2 Implementering . . . . .	10
3.2.1 Beregning af intrinsic parametre og distortion coefficients . . . . .	11
3.2.2 Beregning af Homografi . . . . .	12
3.2.3 Cirkeldetektering . . . . .	13
<b>4 Robotkalibrering</b>	<b>14</b>
4.1 Teori . . . . .	14
4.2 Implementering . . . . .	15
<b>5 Modellering af skrå kast</b>	<b>17</b>
5.1 Teori . . . . .	17
5.1.1 Det skrå kast . . . . .	17
5.1.2 Jakobianten . . . . .	19

5.2	Implementering . . . . .	20
<b>6</b>	<b>Database</b>	<b>22</b>
6.1	Teori . . . . .	22
6.1.1	Database Engine & DBMS . . . . .	22
6.1.2	Eksekvering af queries . . . . .	22
6.1.3	Krav til Databasen . . . . .	22
6.2	Implementering . . . . .	23
6.2.1	Databasens opbygning . . . . .	23
6.2.2	Datastruktur . . . . .	23
<b>7</b>	<b>Det samlede system</b>	<b>25</b>
<b>8</b>	<b>Test</b>	<b>26</b>
8.1	Test af Machine Vision . . . . .	26
8.2	Test af det samlede system . . . . .	28
8.2.1	Første systemtest . . . . .	28
8.2.2	Forbedring af systemet . . . . .	29
<b>9</b>	<b>Diskussion</b>	<b>31</b>
<b>10</b>	<b>Future Work</b>	<b>33</b>
10.1	Automatisk detektering af præcision . . . . .	33
10.2	Automatisk detektering af ramt/ikke ramt . . . . .	33
10.3	Variable vinkelhastigheder ifm. beregning af invers kinematik . . . . .	33
10.4	Gripper timing . . . . .	34
10.5	Forbedringer af Databasen . . . . .	34
10.6	Software Arkitektur . . . . .	35

<b>11 Konklusion</b>	<b>36</b>
<b>12 References</b>	<b>37</b>
<b>13 Appendix</b>	<b>38</b>

## 1 Indledning

Den fjerde industrielle revolution, som mange i vor tid vælger at betegne det, er under konstant udvikling, og det observeres hvordan flere af industriens, såvel, som hverdagens processer, automatiseres. En af de mange fordele ved automation er naturligvis, at de fejl, som mennesket foretager ved den manuelle proces, elimineres.

Med dette i mente, designes og konstrueres der i dag robotter til alverdens forskellige formål og særligt er den 6-ledede robotarm, bl.a. grundet dens universalitet, blevet et uundværligt industrielt værktøj. I netop dette projekt er målet at kunne finde og kaste et objekt og ramme et mål præcist. Her kan man rigtig snakke om at robotten er bedre end mennesket, fordi mennesket ikke bare kan beregne fysikken bag et kast i hovedet og udføre det præcist, hvorimod en robot igennem processorkraft kan regne ud hvordan den skal udføre det og ramme et mål.

### 1.1 Problemformulering

Ud fra ovenstående og den udleverede projektbeskrivelse, udarbejdes en problemformulering hvorfra et robotsystem kan udvikles. Det ønskes at udvikle en robot som kan opsamle en bordtennisbold og derefter kaste og ramme en plastik kop.

- Hvordan oversættes punkter fundet med kameraet, til koordinater i world space og dernæst til robot space?
- Hvordan modelleres et kast, med udgangspunkt i start og slutposition for det kastede objekt, og hvilke faktorer spiller en rolle i timingen af et sådant kast?
- Hvordan kan tovejs databasekommunikation implementeres i systemet, og hvilke data vil være relevante at skrive og læse herfra?

### 1.2 Krav Specifikation

Det færdige robotsystem skal ud fra problemformuleringen opfylde følgende krav:

1. Systemet skal kunne analysere et billede og detektere en bold med diameter på 4cm og en plastik kop med diameter på 10cm.
2. Systemet skal kunne samle bolden op inden for området i bordplan hvor robotten ikke rammer joint limits og safety planes.
3. Robotten skal være i stand til automatisk at udføre en kastebewægelse med lineær acceleration efterfulgt af konstant hastighed i joint space.
4. Bolden skal kunne ramme koppen på alle områder af bordpladen foran punktet hvor bolden slippes, med  $> 80\%$  succesrate.
5. Systemet skal have tovejskommunikation mellem databasen og robotprogrammet, som skal udnyttes til at kunne lagre cellespecifikke informationer ift. både opsætning og statistik om diverse kast.

### **1.3 Rapportens struktur**

Rapporten er struktureret som følgende beskrevet: Først præsenteres hardwarekomponenterne som danner rammen om projektet. Herefter præsenteres hvert teoretiske delkomponent af projektet - Machine Vision, Robotkalibrering, Modellering af skråt kast og database. Til hver af disse kommer først et delafsnit med teori, efterfulgt af den konkrete implementation i projektet.

Disse efterfølges af et afsnit hvor det samlede system præsenteres, og herefter kommer et fælles testafsnit hvori de enkelte delkomponenter testes, efterfulgt af test af det samlede system.

De ovenstående afsnit diskuteres, hvorefter rapporten afrundes med afsnittet Future Work og konklusion.

## 2 Hardware

Tilknyttet semesterprojektet er fire robotceller som er opstillet på balkonen i RobotLab på SDU. Hver celle indeholder et kamera og en robot med en gripper, her er en liste over hardwaren:

- **Basler acA1440-220uc[1] med en Basler C125-0418-5M[2] linse**
- **Universal Robots UR5[3] Robot**
- **Weiss Robotics WSG50[4] gripper**
- **Ethernet switch og kabler**

Robotten og gripperen er begge forbundet til hinanden og til en switch. Herfra løber et Ethernet kabel så man fra en ekstern enhed kan forbinde sig til begge dele samtidig.

Dette gøres ved at konfigurere sin Ethernet adapter til at forbinde til en IP på 192.168.100.*X* adressen hvor man selv vælger *X* som skal være forskellig fra robottens og gripperens *X*, derudover skal man sørge for at have subnet mask sat til 255.255.255.0.

Når man har en subnet mask på 255.255.255.0 betyder det, at de første tre oktetter er Netværksdelen af IP'en, hvor den sidste oktet er Host-delen. Dette betyder at vi på forhånd ved, at de første tre oktetter vil være 192.168.100 hvor den sidste kan være en vilkårlig 8-bit heltalsværdi.

### 3 Machine Vision

I dette afsnit vil skridtene mod vores vision system blive gennemgået i kronologisk rækkefølge. Første skridt mod et samlet vision system er kamerakalibreringen, hvor vi skal finde de parametre der skal til for at udbedre den distortion der finder sted i kameralinsen. Herefter skal der findes en sammenhæng mellem billedet og virkeligheden ved hjælp af homografi. Til sidst skal positionen af bolden og koppen identificeres i billedet.

#### 3.1 Teori

Vi har benyttet os af C++ biblioteket *OpenCV*[5] til al billedbehandling, som hjælper med både at kalibrere kameraet, samt at lokalisere objekter i vores billede. Formålet med Machine Vision delen er først at bestemme intrinsic parameters og lens distortion parameters, som er en information der bruges til at undistorte vores billede. Den anden del af Machine Vision består af udregningen projective transformation, som gør at vi kan mappe punkter i image plane til punkter i table plane.

##### 3.1.1 Kamera kalibrering

Alle kameralinser kommer med en vis forvrængning. Denne forvrængning kan give store problemer i arbejdet med lokalisering af objekter i billedet, da der ledes efter sammenhængende pixels i et mønster. I stedet for at skulle korrigere pixelværdien for hele forvrængningen, var det derfor nødvendigt at undistorte billedet, før vi begyndte at arbejde med det. På denne måde er vi sikre på at der skabes en direkte sammenhæng mellem billedeets koordinater og bordets koordinater.

For at gøre dette skal vi kende kameraets intrinsic parameters. Måden vi får dem er ud fra checkerboard calibration metoden. Denne metode gør brug af et checkerboard, som i vores tilfælde består af 10x7 lige store firkanter. Formålet med checkerboardet er at få taget en mængde billeder, hvor checkerboardet er flyttet rundt i forskellige positioner og orienteringer i forhold til kameraet. Der kan herved gøres brug af *OpenCV* metoden *drawChessboardCorners*[6], som tager billedeerne som input og udregner sammenhængen mellem image- og world koordinater.

Det kan gøres fordi vi har angivet antallet af firkanter på checkerboardet, samt dimensionerne på firkantene, og metoden kan derved beregne alle firkanters positioner. Ud fra sammenhængen mellem image- og world koordinater beregnes kameraets intrinsic parameters i form af en  $3 \times 3$  matrix og distortion coefficients i form af en rækkevektor med 5 kolonner. Vi kan nu bruge disse til at lave en direkte mapping fra virkeligheden til et billede med minimal forvrængning. Denne proces kaldes for rectification.

##### 3.1.2 Homografi

Homografi er en transformation mellem to planer. Da vi i vores tilfælde arbejder på en plan overflade, kan vi anvende homografi til at oversætte punkter fra image plane til world plane. Sammenhængen mellem et koordinat i world plane og et koordinat i image plane kan udtrykkes

på følgende måde, hvor  $H$  er en 3x3 matrix, som er homografi matricen der beskriver sammenhængen mellem world og image plane.

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} \sim H \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix}$$

Måden man beregner denne homografi matrix, er med 4 sammenhængende koordinater i image- og world plane. Dette giver 8 ligninger med 8 ubekendte, som så kan løses. Ligningssystemet er dog ikke et vi selv løser, det bliver gjort igennem metoden *getPerspectiveTransform*[7] som er den *OpenCV* metode der beregner en homografi mellem 4 punkter i image- og world plane.

### 3.1.3 Hough Circle Transform

De to objekter der skal detekteres i projektet, er henholdsvis en bold og en kop. Da begge objekter er cirkulære, vælger vi at benytte os af Hough Circle Transform, som er en specialisering af Hough Transform, hvor der med udgangspunkt i et edge detected billede, bliver fundet cirkler ved hjælp af Hough voting for hver radius. Dette medfører et Hough space, hvor de højeste værdier er der, hvor sandsynligheden er størst for at der findes en cirkel med en given radius. Algoritmen er implementeret i *OpenCV* funktionen *houghCircles*[8]. Denne metode tager som input et gråtone billede, og outputter en *vector<Vec3f>*, hvori alle koordinaterne samt radius for de detekterede cirkler ligger. I metoden kan man angive intervallet for den radius man ønsker at finde, i form af thresholds.

## 3.2 Implementering

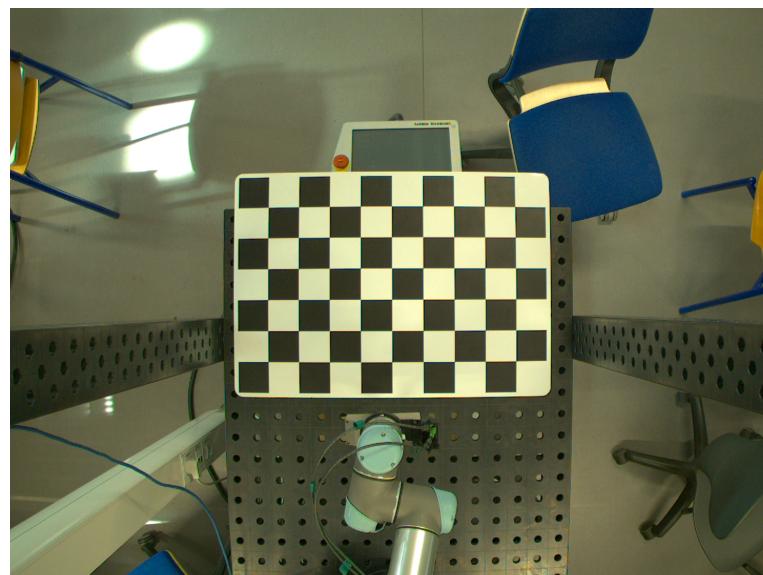
Vores implementation af Machine Vision har taget udgangspunkt i de udleverede programmer *Calibration.zip* og *pylon\_opencv.zip*. Den førstnævnte er et udgangspunkt til et program som har til formål at finde intrinsic parametre og distortion coefficients. Sidstnævnte program indeholder eksempelkode som opretter en liveforbindelse til kameraet. Det er med udgangspunkt i denne eksempelkode at vi skriver vores *MachineVision* klasse. Vores klasse indeholder to metoder. Den ene metode er til at finde bolden og den anden er til at finde koppen. Metoderne hedder henholdsvis *getBall* og *getCup*. Grunden til at vi har to metoder er bl.a. at *getBall* finder cirkler op til en bestem maximum radius og at *getCup* finder cirkler over en bestemt minimum radius. En anden grund er at vi i *getBall* begrænser os til kun at lede efter én cirkel, som opfylder kravet for at være en bold. Grunden til dette er, at vi ikke vil risikere at gribte efter en cirkel, som ikke er en bold. Med denne beslutning begrænser vi os til aldrig at kunne kaste med mere end én bold på bordet. Til gengæld ved vi, at der må være sket en fejl, hvis vi finder mere end én bold med vores vision system. Vi kan derfor returnere en fejl og undgå at gribte ud efter noget uhensigtsmæssigt og dermed risikere at forvolde skade på robotten eller robotcellen. I *getCup* metoden begrænser vi os ikke til kun at finde én kop. Dette er både fordi det ikke er utænkeligt at vi kaster, med flere koppe på bordet, og fordi vi ikke vurderer risikoen til at være høj, hvis vi kaster efter en cirkel, som ikke viser sig at være en kop.

### 3.2.1 Beregning af intrinsic parametre og distortion coefficients

Eftersom kameraet har en vis forvrængning, er det altid vigtigt at starte med kalibreringen af kameraet. Da denne forvrængning er konstant, skal denne kun beregnes én gang for hvert kamera. Vi benytter derfor et separat program til at beregne hvert kameras intrinsic parametre og distortion coefficients. Dette program starter med at behandle 25 billeder af et checkerboard, som består af en  $10 \times 7$  matrix af kvadrater. Gennem metoden *findChessBoardCorners*[9] laver vi en vektor med vektorer  $\mathbf{q}$ , som beskriver placeringen af kvadraterne i  $(x, y, z)$ . Ud fra disse vektorer med koordinater kan man analysere forskellen på de forskellige kvadrater og hermed beregne forvrængningen på kameraet. Efter vi finder koordinaterne, køres metoden *cornerSubPix*[10], som øger nøjagtigheden af kvadraterne til sub-pixel.

Dernæst er det nødvendigt at lave en vektor af vektorer  $\mathbf{Q}$ , som indeholder alle kvadraternes koordinater i checkerboardets koordinatramme. Da checkerboardet er fladt og fordi den beskrives i sin egen koordinatramme, er der ingen dybde i punkterne. Alle  $z$ -værdier bliver derfor sat til 0. Fordi vi bruger den samme størrelse checkerboard for hvert billede, bliver det til en vektor indeholdende lige så mange identiske vektorer, som antallet af billeder vi benytter til kalibreringen.

Efter  $\mathbf{q}$  og  $\mathbf{Q}$  er fundet, giver det nu mulighed for at anvende funktionen *calibrateCamera*[11], som tager disse to vektorer af vektorer, indeholdende sammenhængende punkter fra henholdsvis image- og world plane. Ud fra dette outputter funktionen intrinsic parameters  $\mathbf{K}$  og distortion coefficients  $\mathbf{k}$ . Disse parametre og koefficienter skal benyttes i *machineVision* klassen, og bliver derfor lagt i en database, hvorfra de vil kunne læses fra main programmet. Denne proces bliver beskrevet senere i rapporten i afsnittet der omhandler databasen. Vores cellespecifikke intrinsic parameters og distortion coefficients bliver benyttet i metoderne i *machineVision* klassen til at undistorte de billeder der bliver taget.



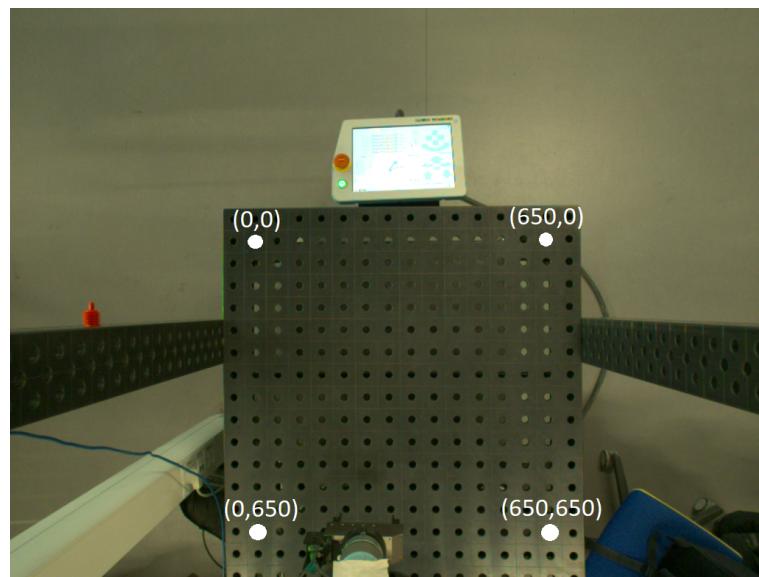
Figur 1: Ubehandlet billede med forvrængning

Figur 1 viser et billede der er blevet taget i main programmet før det bliver undistorted. Det ses

tydeligt på billedet at bordet bliver tyndere i bundet af billedet, da bordets kanter buer ind mod midten. Med denne forvraengning er det vanskeligt at identificere positionen for en bold yderst i billedet, da boldens position også ville være forvrænget. Vi ønsker at bordets kanter på vores undistortede billede er helt lige. Dette vil gøre os i stand til at finde boldens positioner. Få at få et undistorted billede hvor kanterne er helt lige, gælder det derfor om at have præcise intrinsic parameters og distortion coefficients. Disse bliver testet og optimeret i afsnit 8.1.

### 3.2.2 Beregning af Homografi

Med kendskab til disse parametre, er det muligt at begynde med homografin. Første skridt er at mappe nogle punkter i table plane, og finde de tilhørende punkter i image plane. Måden vi gør dette på, er at anvende *houghCircles*, til at finde koordinater for boldens centrum i image plane. Punkterne vi vælger at definere, er yderpunkterne af table plane.



Figur 2: Koordinater i table plane

Som det ses på Figur 2 defineres  $(0,0)$  til at ligge i øverste venstre hjørne, ligesom det gør på et billede. Vi vælger at lægge yderpunkterne 75 mm indenfor bordet, fordi vi vurderede det ville være fornuftigt og mere sikkert ikke at arbejde ude ved kanten af bordet. Dette opdagede vi dog senere ikke var en nødvendighed, da vi alligevel endte med at arbejde tæt på bordets kanter, hvilket medførte koordinater med negativt fortægn. Da det vil have været mere intuitivt at arbejde med positive koordinater burde vi have defineret koordinataksene til at lægge langs bordkanten.

Ud fra disse punkter kan vi nu beregne homografi'en med *getPerspectiveTransform*. Denne metode tager som input en vektor med world koordinater, og en vektor med tilhørende image koordinater. Funktionen outputter en homografi mellem disse to planer, i form af en matrix  $H$ . Denne homografi gør os i stand til at oversætte et vilkårligt punkt fra image plane til world plane. Punkterne vi ønsker at oversætte til world plane er de outputs vi får fra *houghCircles*, som returnerer koordinater og radius for cirkler i image plane.

### 3.2.3 Cirkeldetektering

Hough circle transform algoritmen skal anvendes på et edge detected billede. *OpenCV* implementationen af metoden *houghCircles* har indbygget Canny edge detection. Vi skal derfor bare den et grayscale billede som input. Metoden tager fem væsentlige parametre som vi indstiller på, hvor to af dem hører til den Canny edge detection som er indbygget i metoden. Den første parameter er bin size. Da der kun findes en cirkel i hvert bin, ender denne parameter dermed at styre minimumsdistancen mellem de cirkelcentrum som detekteres. Sættes parameteren for lavt, risikeres der at finde falske cirkler, og sættes den for højt risikeres der samtidigt at misse nogle. I vores tilfælde viste standardværdien på 16, at være den værdi der fungerede bedst. Den næste parameter er det høje threshold for den hysterese der foretages som del af Canny edge detection algoritmen. Sættes denne for lavt vil vi ikke finde alle kanter, og hvis den sættes for højt, vil vi optegne kanter steder hvor de ikke er. Den efterfølgende parameter er det lave threshold. Ved at sætte denne for højt, risikerer vi ikke at få tegnet hele kanten op, og sættes den for lavt fortsætter vi kanten for længe. Vi opnåede de bedste resultater med værdier på henholdsvis 60 og 30. De sidste to parametre er minimum og maximum radius for den cirkel vi ønsker at finde. Måden vi sætter disse parametre, er ved at bruge *houghCircles* metoden på billedet uden en en specifik radius. Dette vil returnere alle tydelige cirkler i billedet, hvorefter cirklen der optegner bolden og koppen kan identificeres og deres radius efterfølgende aflæses. Vi kan nu på baggrund af denne information sætte de thresholds der skal til, for at adskille bolde og kopper i metoderne *getBall* og *getCup*.

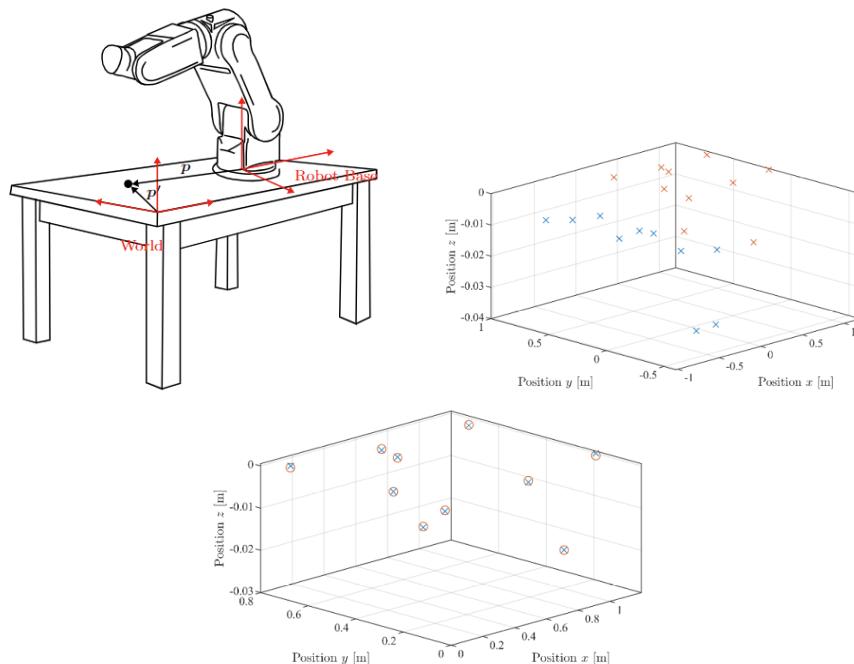
## 4 Robotkalibrering

Følgende afsnit giver en high-level beskrivelse af den lineære kalibreringsmetode, som fremgår i Robot to Table Calibration dokumentet [12], samt en uddybet gennemgang af de valg der er truffet ved implementering af metoden.

### 4.1 Teori

Målet med robot kalibreringen er muligheden for at kunne mappe præcist mellem robottens kartesiske koordinatsystem og de 2D world koordinater som genereres af vision systemet. Robottens koordinatsystem befinner sig i robottens base, med  $x$ - og  $y$ -akser i bord planen, og positiv  $z$ -akse lodret op gennem robotten. World koordinatsystemet er placeret som beskrevet ovenfor i implementeringen af machine vision. Robot kalibreringen tager udgangspunkt i metoden beskrevet i Robot to Table Calibration dokumentet.

På figur 3 gives et overblik over metoden. Her ses det hvordan de 2 koordinatsystemer er placeret i forhold til bordplanen, samt hvordan en teoretisk mapping fra robot- til world koordinater fungerer i cartesian space. Ved nøjagtig brug af fornævnte kalibreringsmetode, outputtes en rotationsmatrix samt en translatorisk vektor, som beskriver mappingsekvensen fra robot- til world koordinater.



Figur 3: Robot og bord i cartesian space

Som vist i udtryk 1 indsættes  $R$ -matrix samt  $T$ -vektor i en transformationsmatrix og ganges på det enkelte robotkoordinat,  $vekp$ , for at få transformationen til world koordinat,  $p'$ . Vi ønsker dog en mapping fra world- til robot koordinater, og ganger derfor den inverse transformations-

matrix på world koordinatet  $\mathbf{p}'$  for at få  $\mathbf{p}$  som vist i udtryk 2.

$$\begin{bmatrix} \mathbf{p}' \\ 1 \end{bmatrix} = \begin{bmatrix} \mathbf{R}_W^R & \mathbf{T}_W^R \\ \mathbf{0}_{1 \times 3} & 1 \end{bmatrix} \cdot \begin{bmatrix} \mathbf{p} \\ 1 \end{bmatrix} \quad (1)$$

$$\begin{bmatrix} \mathbf{p} \\ 1 \end{bmatrix} = \begin{bmatrix} \mathbf{R}_W^R & \mathbf{T}_W^R \\ \mathbf{0}_{1 \times 3} & 1 \end{bmatrix}^{-1} \cdot \begin{bmatrix} \mathbf{p}' \\ 1 \end{bmatrix} \quad (2)$$

## 4.2 Implementering

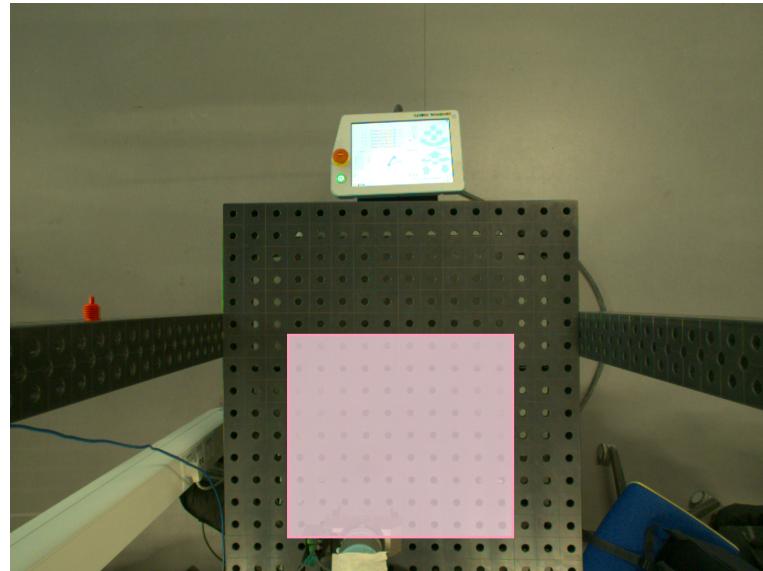
Robotkalibreringen er i vores program implementeret som en separat klasse. Constructoren starter en sekvens af kald af metoder som returnerer henholdsvis world koordinaterne for et givent punkt samt robottens TCP-koordinater, og lagrer herefter disse koordinatsæt som members af klassen. Mere konkret benyttes *machineVision* metoden *getBall*, som returnerer en *vector* med  $x, y$ -world koordinater for centrum af en bold med radius på 4 cm.  $z$ -koordinatet hardcodes til 0, da punktet befinner sig i  $x, y$ -planen. Herefter udskiftes bolden med et 3D-printet objekt, som vist på figur 4. og robotten flyttes hen til "spidsen" på det 3D-printede objekt hvorom, den næsten lukkede gripper, kun lige kan nå ned. Da det lettere kan ses om TCP befinder sig lige over centrum af et smalt objekt end et bredt objekt, er dette med til at sikre så præcis en kalibrering som muligt. Når gripperen er placeret om det 3D-printede objekt kaldes metoden *getActualTCP-Pose* fra Universal Robots' *RTDE Receive Interface* bibliotek, der returnerer robottens aktuelle TCP-pose i kartesiske koordinater [13]. Da kun  $x, y, z$ -koordinaterne af robottens TCP indgår i kalibreringen, låses TCP'et til en fast lodret orientering langs  $z$ -aksen, gennem hele sekvensen hvor punkter returneres.

Alle  $z$ -koordinater i robottens punkter hard codes desuden til 210 mm, og eftersom world koordinaternes  $z$ -værdi er hard coded til 0 mm, vil outputtet blive en præcis mapping af  $z$ -koordinaterne. Dette sørger for at robotten altid vil køre med samme højde på 210 mm hen over et punkt i bordets  $x, y$ -plan. Dette svarer i det komplette system til, at gripperens TCP mappes til et punkt over bolden. Valget er truffet med henblik på at undgå kollision med bolden i bevægelsen hvor robotten flytter hen til det mappede punkt.



Figur 4: 3D-printet objekt til robotkalibrering

Punktretuneringssekvensen kører indtil constructoren får et inputsignal fra terminalen om at stoppe. De  $(3 \cdot X)$ -dimensionelle vektorer  $q$  og  $\dot{q}$  konverteres herefter til Eigen-matricer af typen *Matrix3Xd* [14], hvilket gøres for at minimere fejl samt overskueliggøre debugging-processen ved videre implementering af ovenstående kalibreringsmetode. Grundet de joint limits der er opsat på den fysiske robot, tager kalibreringen kun udgangspunkt i koordinater inden for en mindre del af bordet som vist på figur 5.



Figur 5: Område brugt til kalibrering af robot

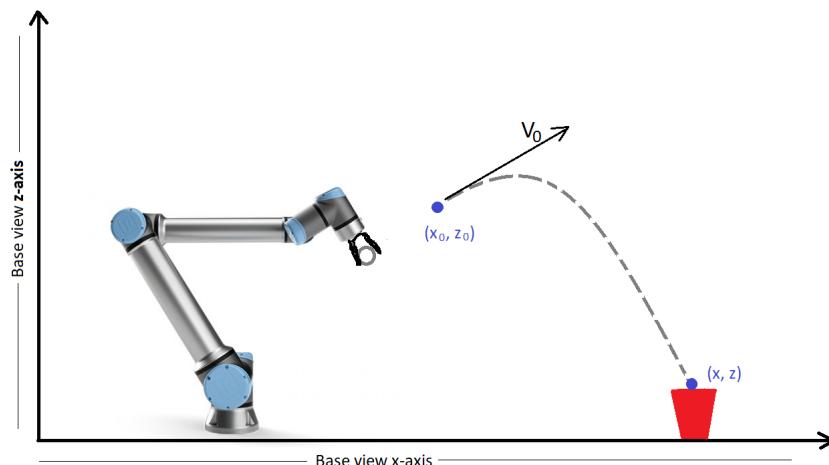
## 5 Modellering af skråt kast

Afsnittet herunder beskriver på teoretisk plan fysikken bag robottens kastebewægelse. Dette indebærer en udledning af hastighedsvektoren for kastet i 3D, samt udledning af joirthastighederne  $\dot{q}$  og startpositionen for kastet  $C$ , i tilfældet hvor der ønskes et kast med lineær hastighed. Herefter præsenteres hvordan teorien er implementeret i praksis, og hvilke overvejelser der har været løbende.

### 5.1 Teori

#### 5.1.1 Det skrå kast

Det fysiske aspekt af kastet tager udgangspunkt i formlen for skråt kast i 2D. På figur 6 fremgår en model af det skrå kast i  $x, z$ -plan, hvor  $(x_0, z_0)$  udgør punktet hvor gripperen slipper bolden (kastets startposition), og  $(x, z)$  udgør punktet hvor bolden skal ramme koppen (kastets slutposition).



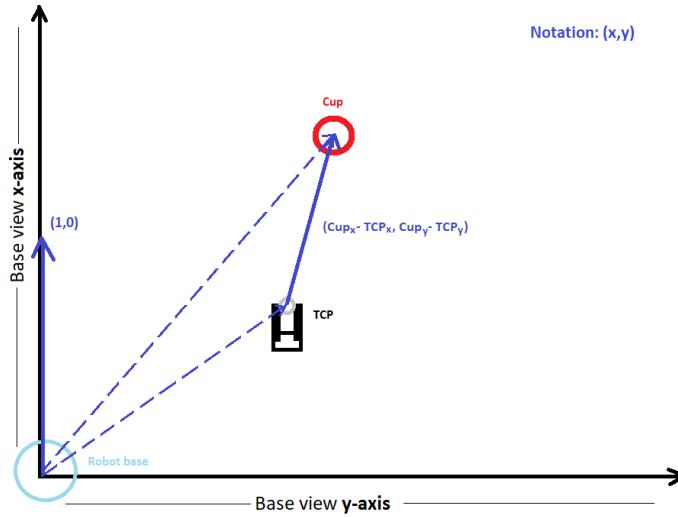
Figur 6: Skrål kast  $x, z$ -plan

Disse værdier benyttes så til udregning af hastighedsvektoren for et skrål kast i 2D, defineret som vist i Appendix A. Her udgør  $t$  tiden hvor bolden er i luften, og  $g$  udgør tyngdeaccelerationen.

$$\mathbf{v}_0 = \begin{bmatrix} \frac{x \cdot x_0}{t} \\ \frac{z - z_0}{t} + \frac{1}{2}gt \end{bmatrix}$$

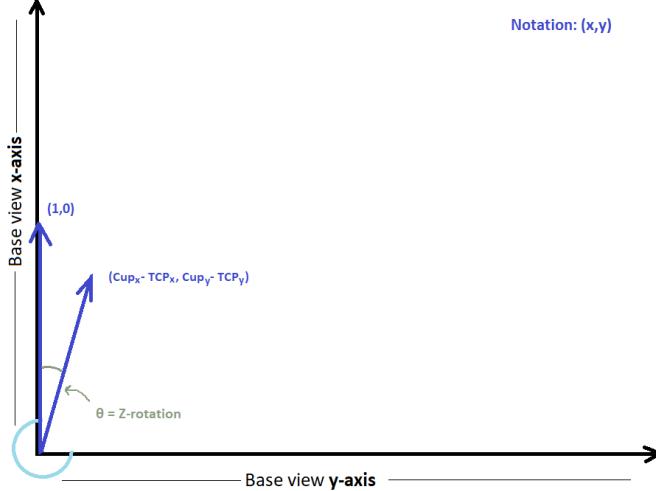
Robotten bevæger sig dog ikke kun i  $x, z$ -plan, men i 3-dimensionalt rum. Dette problem håndteres ved endvidere at tilføje en rotation til den beregnede hastighedsvektor, omkring  $z$ -aksen i robottens kartesiske koordinatsystem. På figur 7 fremgår en teoretisk model af robottens koordinatsystem i  $x, y$ -plan. På modellen er tegnet en enhedsvektor langs  $x$ -aksen, samt to stiplede vektorer fra origo til henholdsvis kop og TCP hvor bolden slippes. Ved at subtrahere de stiplede

vektorer, fås en retningsvektor fra start til slut af kastet.



Figur 7: Skråt kast  $x, y$ -plan

Som det fremgår af figur 8 beregnes vinklen mellem enhedsvektoren langs  $x$ -aksen og retningsvektoren for kastet. Denne vinkel,  $\theta$ , svarer til vinklen omkring  $z$ -aksen hvormed hastighedsvektoren for det skrå kast skal roteres.



Figur 8: Rotation omkring  $z$ , defineret i  $x, y$ -plan

Endeligt fås den ønskede hastighedsvektor for et skråt kast i 3D,  $v$ , ved at multiplicere  $v_0$ , med en  $z$ -rotationsmatrix til vinklen  $\theta$ .

$$v = v_0 \cdot \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

### 5.1.2 Jakobianten

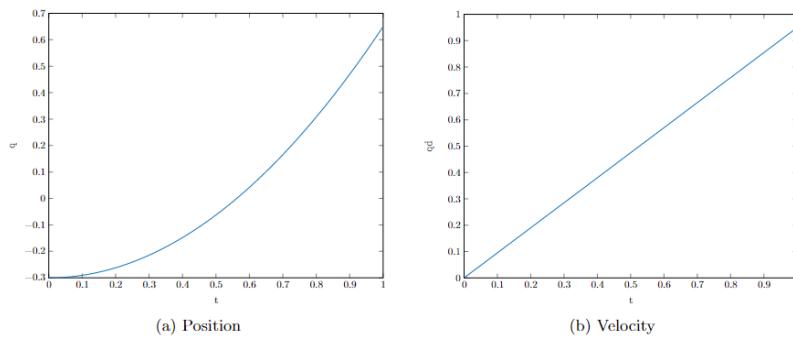
Da end-effectorens vektorielle hastighed i robottens kartesiske koordinatsystem nu er defineret som  $\mathbf{v}$  kan Jakobianten herefter anvendes til at bestemme de individuelle jioin hastigheder som robotten opnår i kastet. Givet er følgende sammenhæng

$$\dot{\mathbf{x}} = \mathbf{v} = \begin{bmatrix} \mathbf{v} \\ \boldsymbol{\omega} \end{bmatrix} \quad (3)$$

$$\dot{\mathbf{x}} = \mathbf{J}(p) \cdot \dot{\mathbf{q}} \quad (4)$$

hvor  $\boldsymbol{\omega}$  udgør end effectorens rotation omkring  $x, y, z$  i robottens kartesiske koordinatsystem.  $\mathbf{q}$  bestemmes til at være et fast defineret position i joint space, hvorfra robotten skal slippe bolden.  $\dot{\mathbf{q}}$  er de jioin hastigheder vi ønsker at finde til vores hastigheder af end-effectoren.

Resultatet af den fysiske modellering bliver en lineær hastighedsvektor, og derfor tager følgende beregninger udgangspunkt i et kast med lineær hastighed og dermed konstant acceleration. På Figur 9 er plottet et grafisk eksempel af henholdsvis position og hastighed, på et sådant kast



Figur 9: Eksempel på kastebane med lineær hastighed og konstant acceleration

Da den vektorielle joint hastighed,  $\dot{\mathbf{q}}$ , netop er defineret som en lineær funktion, med startpunkt i  $(0, 0)$ , gælder følgende:

$$\dot{\mathbf{q}} = \mathbf{a}t + \mathbf{b}$$

$$\dot{\mathbf{q}}(0) = \mathbf{b} = 0$$

$$\dot{\mathbf{q}} = \mathbf{a}t$$

Heraf fremgår endvidere at accelerationen er givet ved koefficienten  $\mathbf{a}$ , da  $\dot{\mathbf{q}} = \mathbf{a}$ . Ved integration af  $\dot{\mathbf{q}}$  bestemmes jointpositionerne,  $\mathbf{q}$ , og robottens jointposition til tiden 0 bestemmes til integrationskonstanten,  $C$ :

$$\mathbf{q} = \frac{1}{2}\mathbf{a}t^2 + \mathbf{C}$$

$$\mathbf{q}(0) = \mathbf{C}$$

Dvs. at startposition for kastet beregnes ved at isolere  $C$ . For at beregne  $C$  må vi kende accelerationen,  $a$ . Denne bestemmes ved at vælge den tid det tager at udføre kastebevægelsen, svarende til tiden fra kastet begynder,  $t = 0$ , til tiden hvor bolden slippes. Denne konstant defineres i det følgende som  $T$ . Heraf må følgende være gældende.

$$\begin{aligned}\dot{\mathbf{q}}(T) &= \mathbf{a}T \\ \mathbf{q}(T) &= \frac{1}{2}\mathbf{a}T^2 + \mathbf{C}\end{aligned}$$

Da  $\mathbf{q}(T)$  er defineret som robottens slutposition i joint space, vil alle konstanter pånær startpositionen,  $\mathbf{C}$ , være kendt, og sluteligt kan  $\mathbf{C}$  bestemmes.

## 5.2 Implementering

Den matematiske beregning af kastet tager udgangspunkt i en slutposition i robottens joints. Derfor har vi til implementering af kastet valgt et fast punkt, hvorfra robotten altid skal slippe bolden. Vi har valgt et kastepunkt der ligger højt over koppen, da det vil kræve mindre opdrift i kastet, at ramme koppen, end hvis kastepunktet var på vandret linje med koppen eller under. Vi ønsker minimal opdrift i kastet, grundet den startposition for kastet,  $\mathbf{C}$ , som beregnes i ovenstående afsnit. Ved kraftigere opdrift vil udgangspunktet for kastet  $\mathbf{C}$ , ligge lavere ad  $z$ -aksen i robottens cartesian space, hvilket medfører risiko for kollision med bordet ( $x, y$ -planen). Af samme grund vælges slutpositionen for kastet med henblik på, at give robotten plads til at skabe fremdrift i kastet. Derfor har vi valgt et kastepunkt som ligger langt fremme i bordplanen. Robotten er naturligvis ikke helt udstrakt i sin slutposition, da dette vil medføre risiko for at ramme en singularitet i selve kastebevægelsen. Modsat ønskes det ikke at robotten skal køre så langt tilbage i bordplanen, til sin startposition for kastet, at det medfører risiko for at ramme en singularitet. Vi vælger altså en slut position som ligger langt fremme i bordplanen, men hvor de individuelle joints stadig har vinkling.

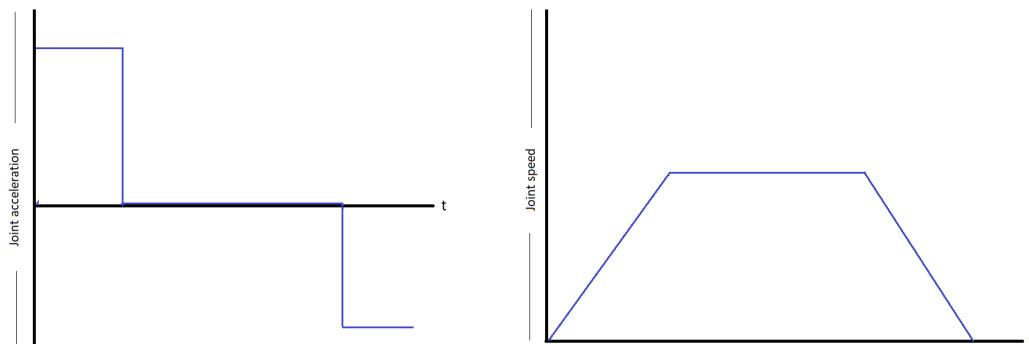
Jakobianten beregnes med slutpositionen  $\mathbf{q}$  som input. Hastighedsvektoren  $\mathbf{v}$  beregnes med udgangspunkt i afsnit 5.1.1, og da vi ikke ønsker nogen rotation af end-effectoren under kastet, sættes  $\boldsymbol{\omega}$  i udtryk 3 til 0. Herefter isoleres  $\dot{\mathbf{q}}$  i udtryk 4, og udledningen af startpositionen,  $\mathbf{C}$ , forløber herefter som beskrevet i afsnit 5.1.2.

Robotten flyttes til beregnede startposition med *moveL\_IK* fra Universal Robots *RTDE API*, og implementeringen af selve kastebevægelsen sker herefter vha. *speedJ* og *speedStop* fra samme bibliotek. Førstnævnte tager robottens joint-hastigheder  $\dot{\mathbf{q}}$  og accelerationen  $\mathbf{a}$  (max  $40\text{ m/s}^2$ ) som input. Outputtet bliver dermed et kast som accelereres lineært indtil der opnås konstant hastighed i joint space, hvorefter accelerationen bliver 0, og den konstante hastighed i joint space forsætter. [13].

*speedStop* er en override-metode der, tager en acceleration (max  $10\text{ m/s}^2$ ) som input, og outputter en negativ lineær acceleration i joint space.

Udover de ovennævnte funktioner benyttes der en fjerde nøglefunktion fra Robotics Library WSG50 API, nemlig *doPrePositionFingers* [15]. Denne tager bredde samt hastighed som parameter, og udfører en bevægelse af gripperfingrene til bestemt bredde (mellem fingrene) med given hastighed. *doPrePositionFingers* anvendes i vores program i *releaseGrip*, som er en metode af *RobotController*-klassen. *releaseGrip* kører først et sleep til kastetiden  $T$  minus en buffer. Dette gøres for at kompensere for den forsinkelse der er fra signalet sendes til gripperen, til bolden slippes i praksis.  $T$  sættes til en værdi der sikrer at max-accelerationen ( $40 \text{ m/s}^2$ ) for  $speed\ddot{J}$  ikke overskrides, jævnfør sammenhængen mellem  $a$  og  $T$  i afsnit 5.1.2 om Jakobianten. Til sidst kaldes *doPrePositionFingers* med maximal hastighed til en bredde større end bolden.

De tre metoder implementeres i mainprogrammet som følgende beskrevet:  
 $speed\ddot{J}$  kaldes med den beregnede  $\dot{q}$  og max acceleration. Samtidig oprettes en tråd som kalder *releaseGrip*, og herefter startes et sleep af main-tråden til tiden  $T + Buffer$ . Når fornævnte sleep har kørt, kaldes *speedStop* med max deceleration.



Figur 10: Ideel acceleration og hastighed på kast i joint space

Når denne kode eksekverer, vil det teoretiske kast i joint space se ud som vist på Figur 10. Der vil først være en konstant acceleration på  $40.0 \text{ m/s}^2$ , hvilket svarer til en lineær stigning i hastighed, i joint space.  $\dot{q}$  vil derefter være opnået inden tiden  $T$  og få en konstant værdi, hvilket medfører at accelerationen bliver 0. Ideelt skal bolden slippes når hastigheden har en konstant værdi, og med henblik på netop at forlænge dette tidsrum, har vi tilføjet en buffer på det sleep der kører i main-tråden. Umiddelbart efter bolden er sluppet, vil der ske en deceleration, samt lineært fald i hastighederne i joint space.

## 6 Database

Det følgende database afsnit indeholder overordnet teori om valg af DBE/DBMS og eksekvering af queries på databasen. Herefter forklares implementationen ift. database-interfacets opbygning samt programmets flow og databasens struktur.

### 6.1 Teori

I dette projekt bruges *SQLite* som database engine og DBMS. Det er et C-baseret bibliotek, som implementerer SQL funktionalitet og bruges her i C++ hvilket er projektets primære programmeringssprog.

#### 6.1.1 Database Engine & DBMS

*SQLite* blev valgt frem for f.eks. *MySQL*, da det har nogle fordele for mindre programmer. *SQLite* tilgår blot en fil i systemet, som den læser fra og skriver til. *MySQL* kræver derimod en forbindelse til en server hvor databasen ligger.

Da dette projekt primært er skrevet i *Qt Creator*, bruges et standard *Qt* bibliotek kaldt *QSqlDatabase*, for at oprette forbindelse til databasefilen gennem *SQLite*.

#### 6.1.2 Eksekvering af queries

Til at eksekvere queries fra interfacet, bruges *QSqlQuery*, som ligeledes *QSqlDatabase* er et standardbibliotek i *Qt*, som tillader at sende queries til en database, man er forbundet til.

##### 6.1.2.1 exec[16]

*exec* er en metode i *QSqlQuery*, som sender en SQL query igennem til DBMS'en, så den bliver eksekveret på databasen. Metoden kan både bruges individuelt med en SQL query som parameter, eller den kan kaldes uden parameter for at eksekvere queries forberedt med *prepare*.

##### 6.1.2.2 prepare[17]

*prepare* metoden bruges til at forberede queries før man eksekverer dem. Det er for eksempel nyttigt hvis man skal indsætte ny data i databasen, da der kan bruges placeholders som eksempelvis kan erstattes med variable fra koden, bundet til de definerede placeholders.

##### 6.1.2.3 bindValue[18]

*bindValue* metoden bruges til at binde en værdi til en placeholder i en query som er forberedt af *prepare*

#### 6.1.3 Krav til Databasen

De krav som gennem projektet er sat til databasen er, at den skal lagre følgende info fra forskellige robotceller, så programmet kan eksekveres på enhver celle:

- IP-adresser på robot og gripper

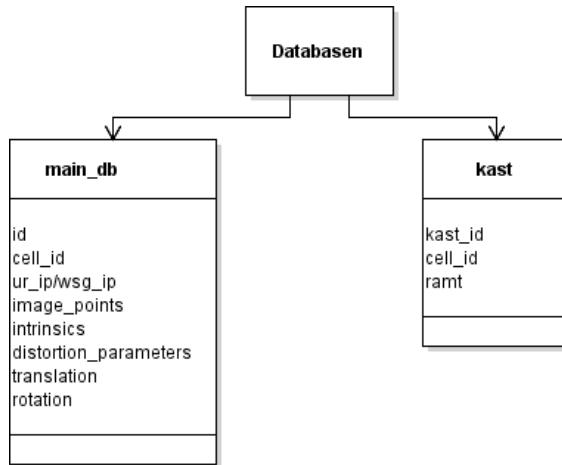
- **Intrinsic parameters & distortion coefficients**
- **Table plane punkter til homografien**

Derudover skal den kunne lagre statistik over robottens ydeevne og præcision.

## 6.2 Implementering

Her beskrives implementeringen af et interface mellem databasen og hovedprogrammet.

### 6.2.1 Databasens opbygning



Figur 11: Databasens opbygning

Som det ses i figur 11, er databasen opbygget primært af to tabeller som er henholdsvis *main\_db* og *kast*. Tabellen *main\_db* indeholder de primære data som bruges til opsætning af programmet ift. hvilken robotcelle man arbejder ved. Hvorimod *kast* tabellen indeholder data omkring performance af robotten ift. om den har ramt et givent kast på en specifik celle. Dette kan bruges til at beregne præcisionen ud for hver celle.

### 6.2.2 Datastruktur

Ift. *main\_db* tabellen blev det valgt at køre primært med strings, udover PK der opbevares som et heltal. Da der i projektet benyttes mange forskellige dataformer, blev det besluttet at indsætte alt i databasen som strings. Grunden til dette er, at der er brug for meget frihed når data behandles, for eksempel når der skal indsættes værdier i rotationsmatricen. Derudover er der også nogle tekniske forhindringer, blandt andet kan *SQLite* kun håndtere få datatyper. På baggrund af dette er det mest effektivt, at lagre alt som strings i databasen og så derefter manipulere med strings for at få den data der er brug for.

I *kast* tabellen opbevares *cell\_id* som en string, mens resten er heltal. Grunden til dette er, at der kun er brug for en boolsk værdi til at beskrive om kastet ramte i koppen eller ej. Her er det derfor mest effektivt at lagre et 1 eller et 0 tal som henholdsvis ramt eller ikke ramt.

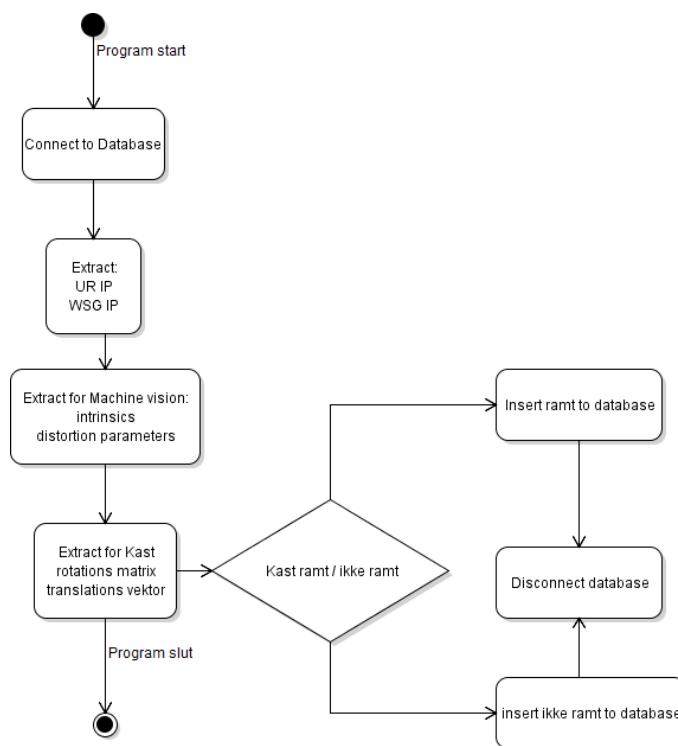
Implementation af interfacet indebærer at lave en klasse som har diverse metoder der bl.a. opretter forbindelse til, læser fra og skriver til databasen.

Databasen implementeres primært ift. opsætning af programmet i forbindelse med eventuelle skift mellem robotceller. Enhver celle har:

- Robot- og grippers IP-adresse
- Intrinsics parametre
- Distortion koefficienter
- Camera-table kalibrering
- Robot-table kalibrering

som alt sammen har potentielt forskellige værdier afhængig af hvilken celle man arbejder med.

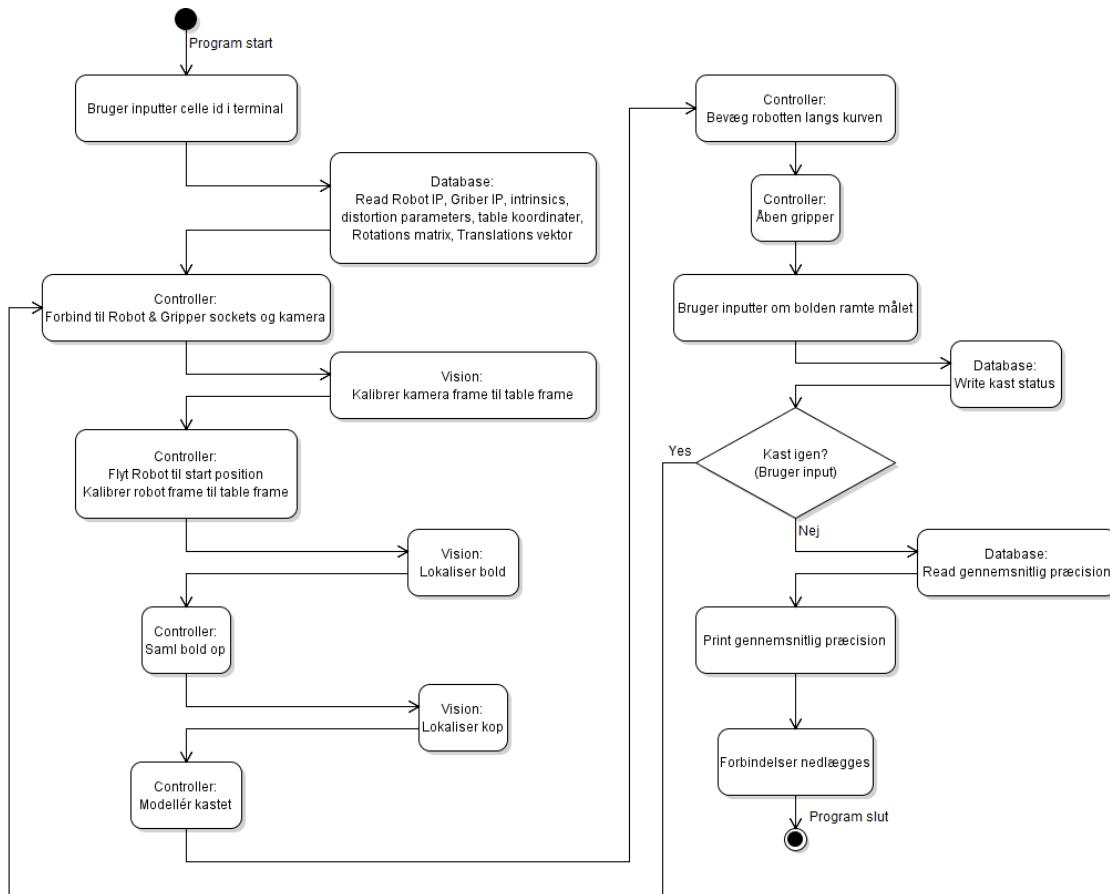
Nedenfor på figur 12 ses databasens flow.



Figur 12: Databasens systemets opbygning

## 7 Det samlede system

Det endelige system starter med at læse de cellespecifikke parametre fra databasen, som bruges til at sætte programmet ordentligt op og kalibrere de tre koordinatrammer med hinanden. Herefter findes og kastes bolden efter koppen, og status efter kast registreres manuelt. Man har som bruger valget om at kaste igen med samme celleparametre, eller om at stoppe og få udprintet en gennemsnitlig kastepræcision for cellen. Programmets flow er illustreret på figur 13 nedenfor.



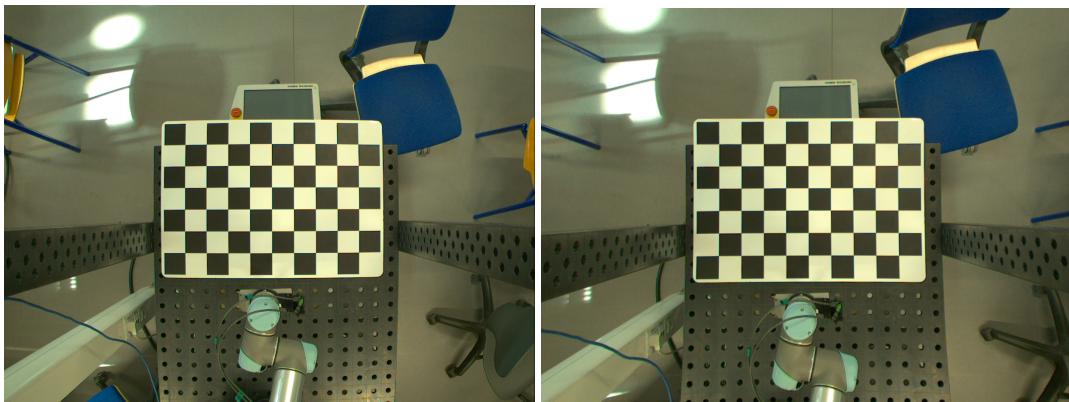
Figur 13: Flow diagram over det samlede system

## 8 Test

I dette afsnit vil der først blive lavet en isoleret test af vision systemet. Da det er besværligt at foretage isolerede tests af de andre elementer, vil test og evaluering af disse indgå i en samlet systemtest som kommer efterfølgende.

### 8.1 Test af Machine Vision

For at teste vores intrinsic parametre og distortion coefficients tager vi et billede gennem vision programmet og benytter dem til undistortion. Resultatet af første test, kan ses på billedet til højre på figur 14.

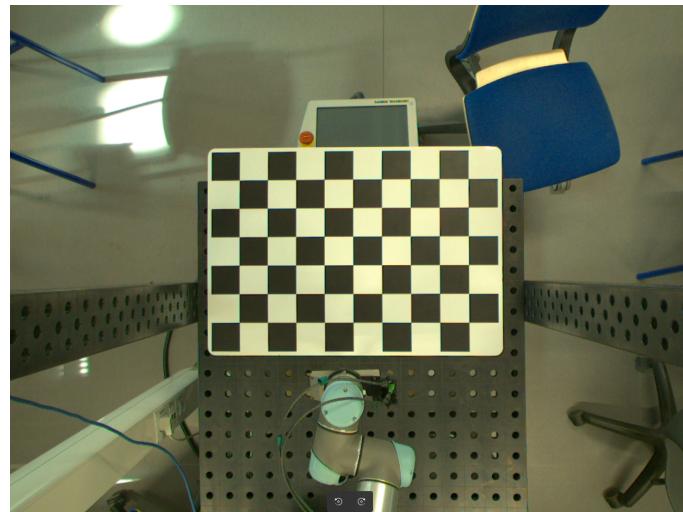


Figur 14: Oprindelig billede til venstre og dårligt undistorted billede til højre

Det ses tydeligt at bordets kanter ikke er rettet ud som ønsket. Vi har derfor en opnået en dårlig undistortion af billedet, da der stadig er en forvrængning, men nu den modsatte vej af det oprindelige billede. Dette kunne skyldes tre væsentlige punkter, som er vigtige i en god kalibrering.

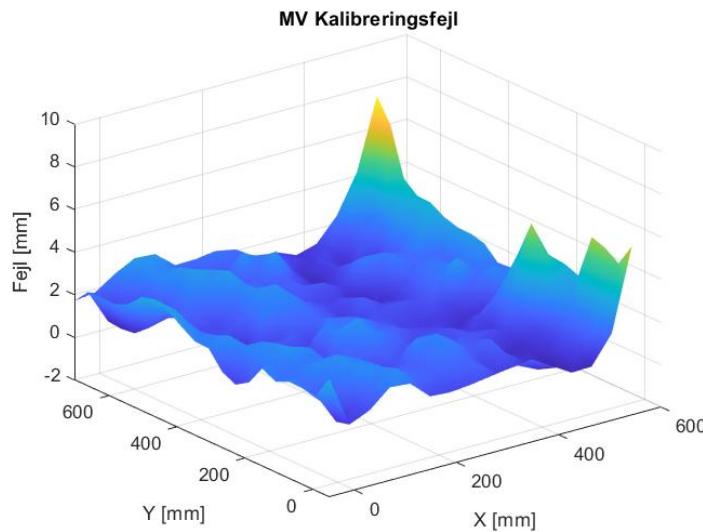
- Jo flere billeder der tages desto bedre kalibrering kan der laves.
- Størrelsen af checkerboardet har en betydning, da kameraet bliver kalibreret bedre jo mere dækkende det er.
- En stor varietet mellem placeringen af checkerboardet, da det både er vigtigt at få dækket alle pixels i billedet, samt at få dækket forskellige værdier af  $z$ -aksen.

Eftersom 20-30 billeder burde være tilstrækkeligt for at opnå den præcision vi ønsker, og fordi vi vurderer at checkerboardets størrelse er passende, formoder vi at den dårlige undistortion skyldes mangel på variation i vores billeder af checkerboardet. Vi tager derfor 30 nye billeder, hvor der lægges fokus på at checkerboardet når at dække alle pixels gennem alle billederne. Samtidigt sørges der for checkerboardet bliver vinklet, så vi opnår nogle forskellige værdier i  $z$ -aksen. Resultatet af det nye undistortede billede kan ses i figur 15. Her ses det tydeligt at bordets kanter er lige, og det vurderes derfor som en god kalibrering. Der er dermed fundet gode intrinsic parameters og distortion coefficients.



Figur 15: Undistortion af billede med gode intrinsic parameters og distortion coefficients

Med vores endelige intrinsic parameters og distortion coefficients fundet, vil vi nu lave en samlet test af vores vision system. Med denne test ønsker vi at teste pålideligheden af de positioner vision systemet mener et objekt har, i forhold til hvor den rent faktisk befinner sig. Måden vi gjorde dette på var at dele bordet op i et grid med 13x15 punkter, hvor hvert punkt havde en afstand på 50 mm mellem sig. Med denne metode ender vi med en test der indeholder 195 datapunkter, hvor vi placerer bolden i hvert punkt, og benytter vision systemet til at finde positionen. På denne måde kunne vi beregne fejlen i vision systemets givne punkter, ved at sammenligne med de kendte punkter i bordet. Resultatet af testen er afbilledet i et spredningsdiagram, som kan ses på figur 16.



Figur 16: Spredningsdiagram der viser fejlen i machine vision, beskrevet i image plane

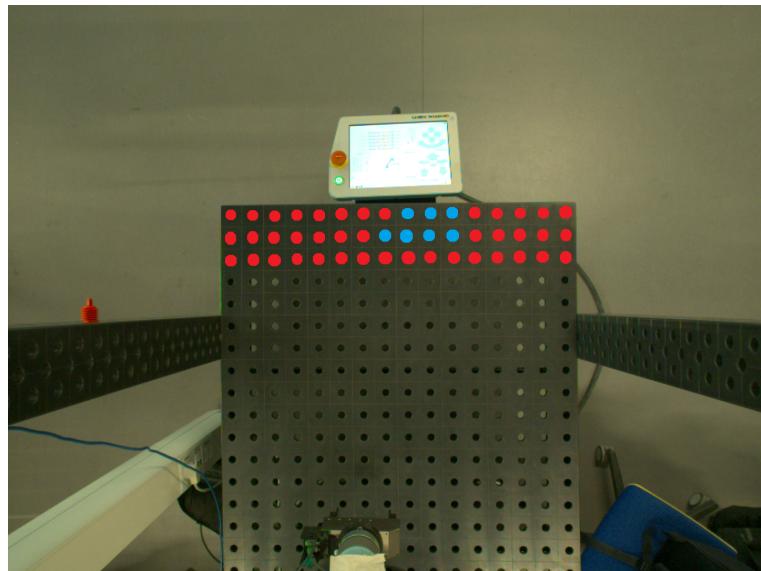
Diagrammet afbilder fejlen i vision systemet, som afstanden der er fra punktet der bliver fundet igennem kameraet til den kendte position af punktet i virkeligheden. Grafens akser er defineret i image plane, hvilket vil sige at punktet (0, 0), er det punkt der ligger øverst til venstre på bordet

jævnfør figur 2. Det ses på grafen at der er konstant minimum fejl på omkring 2 mm, hvor fejlen er mindst når bolden befinner sig lodret under kameraet, og fejlen vokser jo længere væk bolden kommer fra det punkt.

## 8.2 Test af det samlede system

### 8.2.1 Første systemtest

I evalueringen af vores samlede system, opstiller vi en test, der kan illustrere hvor præcist robotten kan ramme koppen i forskellige positioner. Ligesom i testen af vision systemet deler vi bordet op i et grid af firkanter, med 50mm imellem sig. Opstillingen af testen bestod af en bold, som altid var placeret samme sted, og derefter med en kop der løbende bliver placeret på alle firkanter i de tre øverste rækker af griddet. Resultatet af succesfulde kast i testen, kan ses på figur 17. Vi tester alle positionerne af koppen 5 gange. En blå farve på figuren betyder at vi i dette punkt har opnået vores krav om en succesrate på 80%. En rød farve betyder at succesraten ikke er opnået.



Figur 17: Første systemtest hvor der kastes med en fast gripper vinkel

Vi vælger at stoppe testen efter vi har kastet efter koppen på de tre øverste rækker på bordet, da vi allerede på tredje række ikke ramte en eneste kop. Vi antager derfor at vi ikke vil ramme koppen i nogle positioner under denne række, baseret på det generelle offset kastet havde, jo tættere koppen befandt sig på robotten.

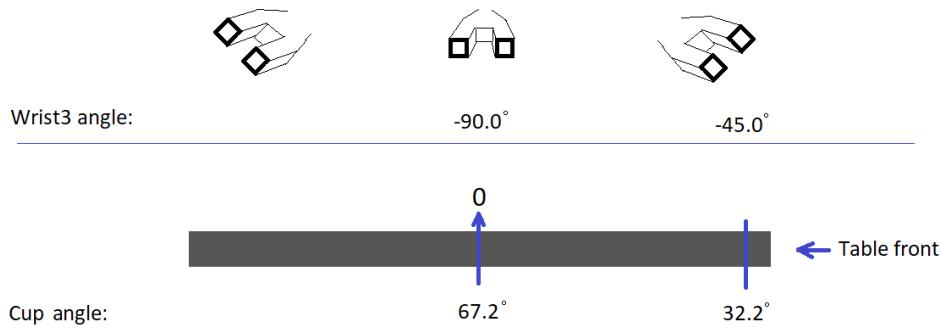
Vi kan se på figuren at det ikke kun er et problem med kast i dybden, men bredden også giver fejl, jo længere væk koppen kommer fra midten af bordet. Under observeringen af testen opdager vi at dette kunne skyldes gripperfingrene. Fordi gripperfingrene altid peger vandret ligeud uden at tage højde for retningen af kastet, ender gripperfingrene med at blokere bolden i dens bane mod koppen. Dette gør det sværere for robotten at ramme, jo længere væk koppen er fra midten.

### 8.2.2 Forbedring af systemet

Grunden til at vi ikke kan ramme koppen i et større område end på Figur 17, kan skyldes flere grunde. På baggrund af Figur 16 vurderer vi ikke at fejlen skyldes vision systemet, da fejlen her ikke lader til at vokse betydeligt ud mod kanten af bordet. Vi formoder at årsagen hænger sammen med den observation, der blev gjort om gripperfingrene i forrige afsnit. Gripperfingrenes blokering af bolden, hænger meget godt sammen med at vi ikke kan ramme koppen i et bredere areal. Dette er fordi vinklen bolden forlader gripperen med, må være større desto længere væk koppen står fra bordets midte, og fingrene dermed blokerer bolden mere.

Vi ønsker at introducere et system, der dynamisk ændrer gripperens vinkel, som funktion af koppens afstand fra bordets midte.

Konsekvensen af at ændre gripperens vinkel til hvert kast, er som udgangspunkt at vi skal ændre Jakobianten hver gang. Det led vi ændrer er wrist 3, hvilket er det yderste joint på robotarmen. Vi har derfor et specialtilfælde hvor en ændring af jointpositionen af yderste led ikke medfører en ændring af TCP, og det er derfor ikke nødvendigt at genberegne Jakobianten. Vores imple-



Figur 18:

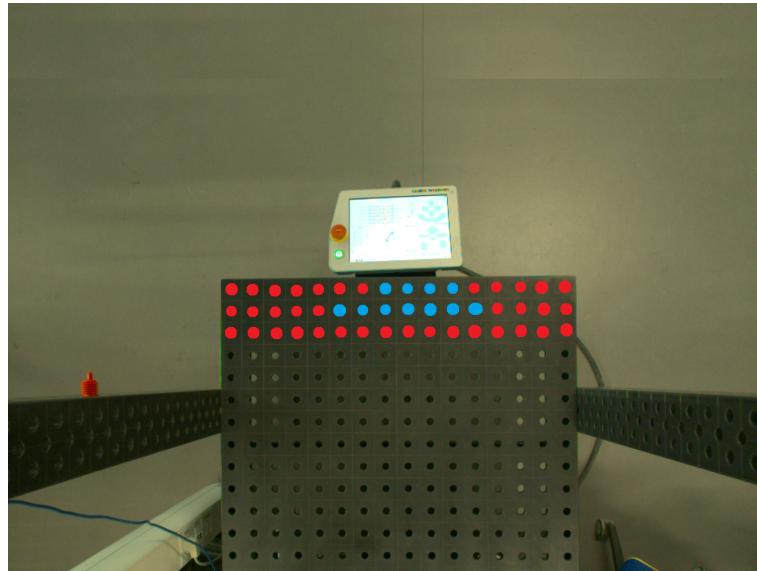
mentation af vinkelændringen er med udgangspunkt i en meget forsimplet model, der bunder i en række antagelser. Modellen tager udgangspunkt i en antagelse om en ønsket vinkling af gripperen til en bestemt afvigelse fra nulpunktet, som kan ses på figur 18. Med disse to vinkler kendt, kan vi så beregne det forhold gripperens vinkel har i forhold til koppens position.

$$\text{vinkelforhold} = \frac{\text{ønsket vinkling af gripperen}}{\text{afvigelse fra nulpunktet}} = \frac{-90-(-45)}{67.2-32.2} = -1.28$$

Vi antager herefter at gripperens vinkel altid skal have dette forhold til enhver position af koppen. Gripperens vinkling i forhold til koppens position, kan derfor udregnes til en vilkårlig position på følgende vis.

$$r = \angle \text{kop} \cdot \text{vinkelforhold}$$

Måden vi vælger at implementere dette på er som sagt meget forsimplet. Grunden til dette er, at vi forholdsvis hurtig kunne implementere det og dermed hurtigere komme til en konklusion af vores påstand omkring gripperfingrenes blokering. Den rigtige måde at løse problemet på er at betragte det som et geometrisk problem. Resultatet af dette kan ses på figur 19.



Figur 19: Anden systemtest hvor der kastes med dynamisk vinklet gripper

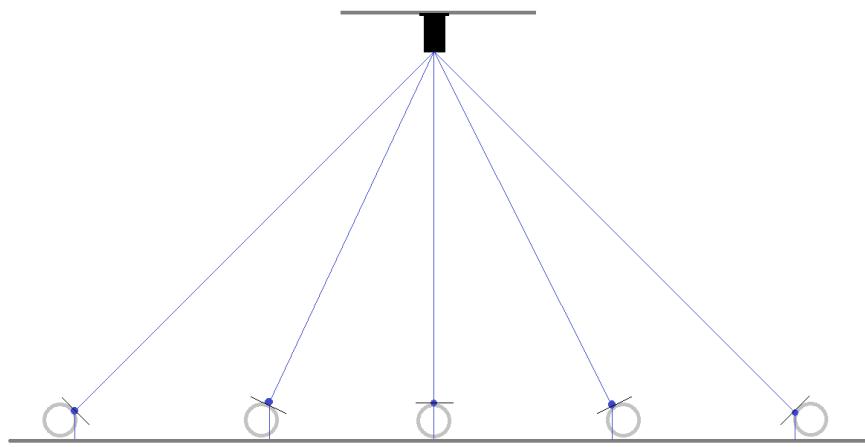
Ud fra resultatet af anden systemtest ses det, at vi har udvidet det område hvor vi kan ramme bolden i bredden, med vores ønskede succesrate. Implementeringen af en dynamisk vinklet gripper, har derfor bekræftet vores antagelse om, at gripperen før blokerede bolden. Vi kan dog også se på billedet, at vi stadig sidder tilbage med det problem, at vi er begrænset til kun at ramme koppen inden for et forholdsvis snævert område i dybden. Dette problem nåede vi dog aldrig at blive færdig med at løse og det falder derfor afsnittet Future Work.

## 9 Diskussion

Efter udarbejdningen og tests af det endelige system, er vi stødt på flere ting, som kunne forbedres.

På figur 20 ses det hvilke koordinater Machine Vision systemet returnerer når bolden er i forskellige positioner på bordfladen. Lige under kameraet vil der være overensstemmelse mellem det returnerede koordinat og boldens reelle position. Derimod vil der i yderpunkterne af bordet være en afvigelse, da det returnerede koordinat er centrum for den cirkel, der ligger tangent på bolden til kameraets vinkel.

Dette skyldes at bolden ikke ses direkte ovenfra på hele bordpladen af kameraet, men med varirende vinkel, som funktion af hvor langt ude i kameraets synspunkt bolden befinder sig.



Figur 20: Offset fra kameravinkel

Offsettet som indgår i kamerakalibreringen, vist på figur 16 bliver ført med videre i robotkalibreringen, hvilket er endnu en potentiel fejlkilde, som påvirker præcisionen af det endelige kast. Dette skyldes at world koordinaterne, som indgår i robotkalibreringen, er produkter af vision systemet. Som beskrevet i afsnit 4.2, figur 5, kan kun få af disse punkter, nemlig punkterne i midten af bordet hvor offset fra vision systemet er lavt, indgå i robotkalibreringen. I den lineære kalibrering af robotten tages der derfor ikke højde for det større offset i yderpunkterne af bordet og her vil robotkalibreringen derfor være upræcis.

Endnu et problem som opstår grundet begrænsninger af joint limits, er sandsynligheden for ikke at være i konstant hastighed når gripperen slipper bolden, som vist på figur 10. Da slutpositionen for kastet er tæt på et safety plane, må der startes en deceleration af systemet relativt kort tid efter den konstante hastighed er opnået. Dette giver en stor udfordring i gripper-timing, da vinduet hvor vi kan kaste med ønsket hastighed bliver lille. Der er derfor en reel risiko for at kaste enten før konstant hastighed er opnået, eller efter decelerationen er påbegyndt.

Ved beregningen af jointhastighederne, som bliver udregnet i ligning 4, sætter vi vinkelhastighederne til 0. Begrundelsen for dette, var på daværende tidspunkt, at vi ikke ønskede en vinkelæn-

dring i løbet kastet, hvilket værdier forskellige fra 0 vil medføre. Vi ønskede sidenhen at ændre orienteringen af end-effectoren. Vi ønsker ikke at beregne vores Jakobiant for hvert kast, og enhver ændring af jointposition der også ændrer TCP, vil resultere i en ny Jakobiant. Vi er derfor begrænset til kun at ændre på vinklen af den yderste joint, wrist 3, da en ændring af denne ikke vil ændre TCP. Med denne tilgang mangler vi derfor friheden til at ændre de to andre wrist led. Dette havde ikke været et problem hvis vi i stedet for at sætte vinkelhastighederne af wrist leddene til 0, havde arbejdet med variable vinkelhastigheder. Grunden til at dette ikke havde været et problem, er at vinkelhastighederne bliver defineret inden den inverse kinematik bliver udregnet i ligning 4. Så ved at definere vinkelhastighederne her, vil resten af kinematikken tage højde for denne ændring, og ændringen af positionen af TCP, vil blive kompenseret for, af resten af robotten i de resterende joints. Dette resulterer i sidste ende i en uændret position af TCP, til trods for at de tre wrist led vil have ændret deres positioner i joint space. Der hvor udregningen bliver mere kompleks, er at ligning 4, er et ligningssystem med 6 ligninger og 6 ubekendte, og introduktion af flere variable vil tilføje kompleksitet til denne beregning. Et implementeringsforslag af denne løsning vil blive præsenteret i Future Works.

I projektet er der brugt en database med to tabeller, der hver har vidt forskellige vigtighed. Den ene tabel har få rækker men vigtig information, hvor den anden har mange rækker men mindre vigtig information. Vi har i denne iteration undgået at lave nogen form for indeksering på databasen, hovedsageligt fordi der ikke er brug for det. Main\_db indeholder celle information hvilket vil sige, at der kun er 5 rækker inklusiv simulationsrobotten og kast tabellen indeholder kun kast information for en celle. Det vil sige at der overordnet ikke er meget data, der skal søges igennem, for at få den ønskede information. Men med skalering af antallet af celler, kan effektiv informationssøgning hurtig blive til et problem. Her er det derfor vigtigt at sørge for at få struktureret data ordentligt. Her kunne der være en indeksering på cell\_id, da der ofte er brug for data til en specifik celle.

## 10 Future Work

Dette afsnit har til formål at introducere løsninger, til de problemer vi står overfor med sidste iteration af robotten. Udover løsninger af problemer, er der også tale om nye systemer, som vi mener vil forbedre robotten. Det er derfor et afsnit hvor der er tale om konkrete løsningsforslag og teori, som vi ikke nåede at få implementeret.

### 10.1 Automatisk detektering af præcision

I vores test af systemet har vi kun mulighed for at teste hvor vi kan ramme og hvor vi ikke kan. Vi kan derfor ikke konkludere noget om hvor meget vi rammer forbi målet. Hvis en test af dette skal laves, kræver det at vi har et system, der kan fortælle os hvor meget forbi målet vi er. Sådan et system skulle laves i machine vision, da det drejer sig om at få aflæst positionen af bolden, når den rammer bordet ved siden af målet. Da vi har at gøre med et system der kun har ét kamera, har vi ikke nogen information om hvor et objekt befinner sig i dybden. For at finde ud af hvornår bolden rammer bordet, skal vi derfor udnytte at boldens radius i image plane, bliver mindre i takt med at boldens afstand til kameraet vokser. En ide til et system der kan detektere boldens fejl til kameraet, ville tage udgangspunkt i netop denne feature.

### 10.2 Automatisk detektering af ramt/ikke ramt

En anden indlysende tilføjelse til vision systemet, vil være et system der automatisk detekterer om koppen er ramt eller ej. Vores idé til implementering af dette system er meget simpel og er med udgangspunkt i metoden *houghCircles*, som bruges til identificering af koppen og bolden. Systemet der automatisk detekterer om vi rammer, vil derfor ligesom *getBall* funktionen, blot skulle finde en bold på bordpladen. Denne gang dog med den ekstra constraint, at centrum af den fundne bold skal ligge inden for koppens cirkel. Findes bolden med denne ekstra constraint, kan vi derfor konkludere, at vi har en fuldtræffer. Kan bolden ikke findes med denne constraint, har vi ramt forbi. Et system som dette vil fungere godt i samspil med databasen, og den vil skulle erstatte den metode, som tager et manuelt input fra brugeren om der er blevet ramt eller ej.

### 10.3 Variable vinkelhastigheder ifm. beregning af invers kinematik

Vi har tidligere introduceret vores ønske om større frihed til at ændre vinklingen af end-effectoren, ved at ændre på alle tre wrist led. Som det fremgik i diskussionen, vil en ændring vinkel af alle wrist, pånær den yderste wrist 3, forsage en ændring af TCP, og dermed en genberegning af Jakobianten. Løsningen på dette vil være at gøre vinklerne for wrist leddene variable, og beregne dem i ligningen for den inverse kinematik. Udfordringen ved dette er dog, at der ved introduktionen af flere variable i ligningen for den inverse kinematik, vil blive skabt et underdetermineret ligningssystem. For at forhindre dette, bliver vi derfor nødt til at ændre en variabel til en konstant, hver gang vi ændrer en konstant til en variabel. Rent praktisk giver dette os muligheden for at låse et led, for hver variabelt wrist led vi indsætter.

Med denne implementation havde det været muligt at låse de joints, der ofte rammer sine joint

limits. Dette kunne f.eks. være joint 2, som ofte stødte ind i dem. For at mindske sandsynligheden for sammenstød mellem boldens kastebane og med gripperfingrene, virker det også som en god ide at fastholde wrist 3 i en fast vinkel på -90 grader, og i stedet styre orienteringen af TCP med de andre to wrist led.

#### 10.4 Gripper timing

Med formodning om at gripper timingen har været en større faktor i præcisionen af vores kast, har vi udtænkt nogle testmetoder som kunne benyttes til at understøtte eller forkaste denne teori. I programmet er der flere variable som bestemmer hvornår gripperens fingre åbnes i kastet. Som tidligere nævnt indebærer det bl.a. tiden  $T$ , som løber fra igangsættelse af kastebevægelsen, indtil gripperen slipper bolden. Derudover har vi implementeret en buffer i main-tråden, der har til funktion, at sikre at kastet forløber ved konstant hastighed i joint space. Den sidste bemærkelsesværdige variabel i systemet, er det offset, som trækkes fra i timingen af *releaseGrip* som beskrevet i afsnit 4.2.

Sidstnævnte variabel bør tilskrives en konstant værdi som baserer sig på outputtet af en isoleret test. Da det ønskes at vide hvor lang tid gripperen er, om at modtage en given kommando, kan en potentiel test udformes som følgende beskrevet: I et tomt main-program oprettes der forbindelse til gripperen. Herefter startes et sleep af main-tråden på eksempelvis 1000 ms, og samtidig kaldes en timerfunktion på en serarat tråd. Efter sleepet i main-tråden eksekveres en *doPrePositionFingers*, og straks efter stoppes timeren i den serarate tråd. Outputtet af timeren vil dermed svare til 1000 ms plus den tid det har taget gripperen at modtage samt eksekvere *doPrePositionFingers*-kommandoen.

For at finde den optimale værdi af de øvrige variable, må der udtænkes individuelle tests af hver variabel. Eksempelvis kunne man tilskrive bufferen i main-tråden en konstant værdi, og herefter teste variablen  $T$ , til forskellige kop-positioner med henblik på at finde en konkret sammenhæng mellem disse. Da førnævnte buffer forlænger vinduet hvor kastet kan foretages ved konstant hastighed, som på figur 10, må bufferen på teoretisk plan være afhængig af  $T$ , og denne teori bør herefter kunne testes.

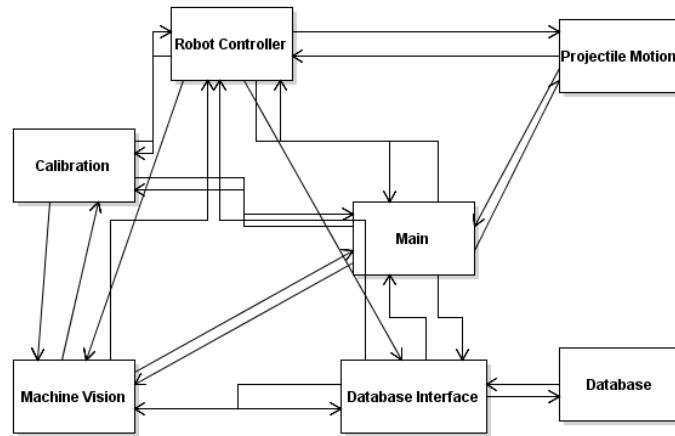
#### 10.5 Forbedringer af Databasen

Selvom databasesystemet virker optimalt og lagrer vigtige informationer, er der altid plads til forbedring. Det nuværende databasesystem lagrer kun informationer til opsætning af kastet, samt det endelige resultat. Her kunne der implementeres yderlige data, blandt andet kunne der lagres køretid af hver celle så man kan se hvor længe hver celle har kørt. Der kunne også lagres data om hvert kast, for eksempel hastigheden af kast eller Jakobianen. En endelig forbedring kunne være at implementere en grafisk brugergrænseflade til databasen, som selvstændigt kunne præsentere dataene. Her menes der, at GUI'en kører parallelt med hovedprogrammet og kun bliver opdateret med databasens indhold og ikke af selve programmet. Alt det ovenstående ville forbedre analysering af cellerne, da man ville have adgang til alle cellers informationer i samme

interface. Her kan der sammenlignes mellem cellerne, for at finde optimale opsætninger. Der kunne også implementeres justering af databasens indhold i GUI'en, så man slipper for at indsætte data igennem metodekald, men hvor man kan redigere dataene direkte gennem GUI'en.

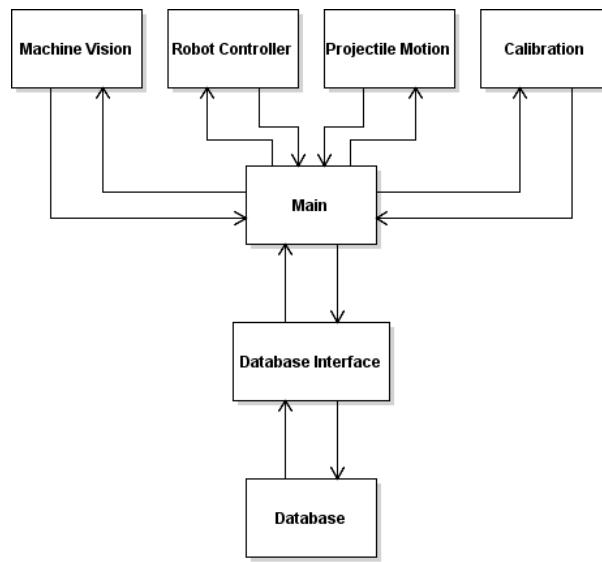
## 10.6 Software Arkitektur

En fremtidig ændring som i høj grad ville forbedre systemet ift. formindskning af kompleksitet, nemmere debugging og skallering af projektet, er forsimpling af softwarearkitekturen. Der foregår udveksling, frem og tilbage, af data mellem stort set alle klasser som vist på figur 21.



Figur 21: Nuværende software arkitektur

Idéelt vil der kun være én klasse som snakker med interfacet til databasen, for at undgå potentielt uendeligt mange forbindelser og query kald på databasen. Et eksempel på det ses på figur 22.



Figur 22: Idéel software arkitektur

Derudover kan navngivning af variable, metoder og klasser forbedres væsentligt. Det skaber stor forvirring omkring debugging og videreudvikling af programmet, at variable og metoder nærmest hedder det samme, og har næsten de samme funktioner.

Generelt bør der fokuseres mere på at lave og opretholde en god software arkitektur.

## 11 Konklusion

I løbet af projektet er det lykkedes at udvikle et velfungerende robotsystem, som har til opgave at udføre et fuldautomatiseret kast.

Mere konkret er der udviklet et kalibreringsprogram, som finder koefficienter til undistortion af billedet. Derefter er der udviklet et visionsystem, som beregner homografien mellem image- og world plane. Med baggrund i denne sammenhæng i kan programmet finde cirkler og returnere positionen af disse i world space.

Koordinaterne for fornævnte objekter konverteres herefter vha. klassen, *Calib*. Klassen tager koordinatsæt inden for et begrænset område af bordet, fra world plane og tilhørende punkter i robottens kartesiske koordinatsystem. Heraf udledes en rotationsmatrix samt translatporisk vektor, som beskriver en mapping mellem robot- og world koordinaterne.

Vi har herefter en *projectileMotion* klasse som, med udgangspunkt i boldens startposition, er i stand til at beregne den nødvendige hastighedsvektor, for at ramme koppen, i den position der er returneret af visionprogrammet. Hastighedsvektoren bliver først beregnet i 2D, hvorefter den med kendskab til koppens vinkel i robottens koordinatsystem, bliver roteret til 3D.

Gennem invers kinematik bliver hele modelleringen omregnet til joint space vha. den inverse Jakobiant. Selve kastet er implementeret i robotsystemet vha. *RTDE* Bibliotekets *speedJ*, som udfører kastebewægelsen fra den beregnede startposition, og forsætter til et tidspunkt efter hastigheden  $\dot{q}$  er opnået. Sideløbende er der udviklet en database som udnytter tovejskommunikation til robotsystemet. Databasen blev opdelt i to tabeller for at optimere dataoverførsel, hvor den ene tabel lagrer informationer, som bidrager til robotten og den anden lagrer informationer om kast. Databasen lagrer information såsom IP-adresser, celle id, rotationsmatrix og om hvorvidt et kast ramte. Det kan konkluderes at vi har udviklet et robotsystem som ikke helt lever op til kravet om at kunne ramme koppen med 80% præcision i hele målområdet, men derimod et system som rammer med 80% præcision i et mere afgrænset område af bordplanen. Det kan skyldes at systemet har sine fejl og mangler, heriblandt det offset som føres med videre fra vision kalibreringen, og ind i robotkalibreringen. Derudover mangler der i høj grad tests og optimering af gripper timing i kastet.

På baggrund af ovenstående og de delvist opnåede krav, kan det konkluderes at projektet i nogen grad, har ramt plet.

## 12 References

- [1] Basler. *acA1440-220uc*. URL: <https://bit.ly/3yMR3Qz>.
- [2] Basler. *C125-0418-5M*. URL: <https://bit.ly/3yMRcn5>.
- [3] Universal Robots. *UNIVERSAL ROBOT UR5e*. URL: <https://bit.ly/3yH1bwE>.
- [4] Weiss Robotics. *WSG Series*. URL: <https://bit.ly/3pc7rac>.
- [5] OpenCV. *OpenCV*. URL: <https://bit.ly/3ec4EaC>.
- [6] OpenCV Docs. *drawChessboardCorners()*. URL: <https://bit.ly/3E2ucSe>.
- [7] OpenCV Docs. *getPerspectiveTransform()*. URL: <https://bit.ly/3oZGXIT>.
- [8] OpenCV Docs. *houghCircles()*. URL: <https://bit.ly/3pYJKBj>.
- [9] OpenCV Docs. *findChessboardCorners()*. URL: <https://bit.ly/3sbiBh1>.
- [10] OpenCV Docs. *cornerSubPix()*. URL: <https://bit.ly/3E4AZL0>.
- [11] OpenCV Docs. *calibrateCamera()*. URL: <https://bit.ly/3IUHALB>.
- [12] Christoffer Sloth. *Robot to Table Calibration*. URL: <https://bit.ly/3E9bEQc>.
- [13] SDU Robotics. *API Reference*. URL: <https://bit.ly/3yKWO12>.
- [14] Eigen Tux Family. *Eigen 3.4.0*. URL: <https://bit.ly/3mker2Q>.
- [15] Robotics Library. *WSG50 API*. URL: <https://bit.ly/321zOzh>.
- [16] Qt Docs. *QSqlQuery::exec()*. URL: <https://bit.ly/3smnaWg>.
- [17] Qt Docs. *QSqlQuery::prepare()*. URL: <https://bit.ly/3J8W9vw>.
- [18] Qt Docs. *QSqlQuery::bindValue()*. URL: <https://bit.ly/3yH7ez3>.

## 13 Appendix

### Appendix A

$$\vec{v}_0 = \begin{pmatrix} |v_0| \cos(\theta) \\ |v_0| \sin(\theta) \end{pmatrix}$$

$$a(t) = \begin{pmatrix} 0 \\ -g \end{pmatrix}$$

$$v(t) = \begin{pmatrix} c_1 \\ -gt + c_2 \end{pmatrix} = \begin{pmatrix} |v_0| \cos(\theta) \\ -gt + |v_0| \sin(\theta) \end{pmatrix}$$

$$v(0) = v_0 \Leftrightarrow \begin{pmatrix} c_1 \\ -g \cdot 0 + c_2 \end{pmatrix} = \begin{pmatrix} |v_0| \cos(\theta) \\ |v_0| \sin(\theta) \end{pmatrix} \Leftrightarrow \begin{array}{l} c_1 = |v_0| \cos(\theta) \\ c_2 = |v_0| \sin(\theta) \end{array}$$

$$p(t) = \begin{pmatrix} |v_0| \cos(\theta) \cdot t + c_3 \\ -\frac{1}{2}gt^2 + |v_0| \sin(\theta) \cdot t + c_4 \end{pmatrix} = \begin{pmatrix} |v_0| \cos(\theta) \cdot t + x_0 \\ -\frac{1}{2}gt^2 + |v_0| \sin(\theta) \cdot t + y_0 \end{pmatrix}$$

$$p(0) = p_0 \Leftrightarrow \begin{pmatrix} |v_0| \cos(\theta) \cdot 0 + c_3 \\ -\frac{1}{2}g \cdot 0^2 + |v_0| \sin(\theta) \cdot 0 + c_4 \end{pmatrix} = \begin{pmatrix} x_0 \\ y_0 \end{pmatrix} \Leftrightarrow \begin{array}{l} c_3 = x_0 \\ c_4 = y_0 \end{array}$$

$$\ddot{x} = 0$$

$$\ddot{y} = -g$$

$$\dot{x} = |v_0| \cos(\theta)$$

$$\dot{y} = -gt + |v_0| \sin(\theta)$$

$$x = |v_0| \cos(\theta) t + x_0$$

$$y = -\frac{1}{2}gt^2 + |v_0| \sin(\theta) t + y_0$$

$$|v_0| \cos(\theta) = \frac{x - x_0}{t}$$

$$|v_0| \sin(\theta) = \frac{y - y_0 + \frac{1}{2}gt^2}{t} = \frac{y - y_0}{t} + \frac{1}{2}gt$$

$$v_0 = \left( \frac{x - x_0}{t}, \frac{y - y_0}{t} + \frac{1}{2}gt \right)$$