



Universitetet
i Stavanger

DET TEKNISK-NATURVITENSKAPELIGE FAKULTET

BACHELOROPPGAVE

Studieprogram/spesialisering:

Automatisering og elektronikkdesign

Vårsemesteret, 2020

Åpen / Konfidensiell

Forfattere: Markus H. Iversflaten, Ole Christian Handegård

Fagansvarlig: Ståle Freyer

Veileder(e): Karl Skretting, Morten Mossige

Tittel på bacheloroppgaven: ABB-robot med maskinsyn

Engelsk tittel: ABB robot with machine vision

Studiepoeng: 20

Emneord: Lab, Maskinsyn, ABB, IRB-140, Robot, uEye XS, RobotWare, Robot Web Services, REST API, Python.

Sidetall: 53

+ vedlegg/annet: 16 +
maskinsyn_vedlegg.zip

Stavanger, 29.05.2020
dato/år

ABB-robot med maskinsyn

Markus Iversflaten, Ole Christian Handegård

29. mai 2020

Forord

Forfatterne av denne rapporten er Markus H. Iversflaten og Ole Christian Handegård, som studerer til automatisering og elektronikkdesign bachelor ved Institutt for Data- og Elektroteknikk (IDE) ved Universitetet i Stavanger. Begge kommer rett fra studiespesialisering på videregående skole, og har jobbet de siste somrene hos Aktieselskabet Saudefaldene i Sauda innerst i Ryfylkefjorden.

De siste månedene har vi stort sett tilbrakt på robotlaboratoriet ved UiS. Vi har her hatt privilegiet av å jobbe med en av industrirobotene på campus. Dette har gjort oppgaven både spennende og lærerik.

Vi vil rette en takk til våre veiledere Ståle Freyer, Karl Skretting og Morten Mossige, som har gitt verdifulle innspill gjennom hele oppgaveprosessen.

Vi vil også takke Bart Verboven fra IDS Imaging for tips til både kamera-innstillinger og bildefangst. I tillegg vil vi takke Sammy Alaoui Soulimani ved UiS for hjelp til å forstå kvaternioner.

Sammendrag

Denne bacheloroppgaven har som mål å lage et robotsystem som er i stand til å lokalisere, identifisere, orientere og stable klosser som er tilfeldig plassert i robotens arbeidsområde. Med utgangspunkt i dette systemet, skal et demonstrasjonsprogram lages samt en laboratorieoppgave i faget ELE610 utformes.

Systemet er i hovedsak bestående av bildefangst og -behandling i Python, lokalisering av klosser ved lesing og analyse av QR-koder, HTTP-kommunikasjon mellom Python og RobotWare, og plukking og plassering av klosser ved bruk av en industriell robotarm.

Demonstrasjonsprogrammet som ble utformet inneholder tre hovedruter som viser ulike egenskaper ved systemet. Disse inkluderer plukking og plassering av en valgt kloss, stabling og orientering av klosser, og et evighetsprogram som viser systemets robusthet. Rutinene fungerer også som et avansert løsningsforslag til laboratorieoppgaven.

Laboratorieoppgaven består av flere deloppgaver som skal løses, før et hovedprogram skal utvikles. Hovedprogrammet skal være en forenklet versjon av demonstrasjonsprogrammet. Det er laget et rammeverk som skal gi et godt utgangspunkt for løsning av oppgavene. I rammeverket inkluderes et Python-bibliotek, *rwsuis*, som skal sørge for at kommunikasjonsdelens kompleksitet begrenses. Som et tillegg, er all kode i systemet dokumentert grundig.

Til sist diskuteres ulike valg tatt underveis i oppgaveløsningen, samt mulige framtidige utvidelser av systemet.

Det er laget en video som viser systemet i aksjon.

Lenke: <https://vimeo.com/420278023>

Innhold

1	Introduksjon	1
1.1	Oppgavetekst	1
1.2	Bakgrunn	1
1.2.1	Robot	2
1.2.2	Robot Web Services	4
1.2.3	Kamera	4
1.2.4	Python-pakker	5
1.2.5	JSON	5
2	Konstruksjon	6
2.1	Overordnet system	6
2.1.1	Robotens operasjonsmodi	7
2.1.2	Filstruktur i Python og RAPID	8
2.1.3	Gripeteknikk	9
2.2	Kamera	10
2.2.1	<i>Camera</i> -klassen	10
2.2.2	Bildefangst	11
2.2.3	Bildeoppløsning og -format	11
2.2.4	Innstilling av fokus	12
2.2.5	Eksponeringstid	13
2.2.6	Korrigering av tiltet kamerasensor	14
2.2.7	Lagring av kamerainnstillinger	15
2.2.8	Kontinuerlig videovisning	16
2.3	Lesing av QR-koder	16
2.3.1	Bildebehandling	16
2.3.2	Programvare for lesing av QR-koder	18
2.3.3	<i>Puck</i> -klassen	20
2.4	Posisjonskalkulering	21
2.4.1	Transformasjon av koordinater	21
2.4.2	Konvertere fra piksler til millimeter	22
2.4.3	Kompensere for vinklet kamera	24
2.4.4	Griper- og kameraposisjon	25
2.4.5	Sammensatt eksempel	25
2.5	Kommunikasjon mellom Python og RobotWare	28
2.5.1	Robot Web Services	28
2.5.2	Requests-biblioteket	29
2.5.3	<i>RWS</i> -klassen	30

2.5.4	Brukte HTTP-metoder	31
2.5.5	Lesing av JSON	32
2.5.6	Kommunikasjon under programutførelse	33
2.5.7	Testing av kommunikasjon	34
2.6	Kodedokumentasjon	35
3	Resultat	37
3.1	Demonstrasjonsprogram	37
3.1.1	Plukke og plassere én valgt kloss	38
3.1.2	Stable klosser	40
3.1.3	Evhightsprogram	43
3.2	Laboratorieoppgave	44
3.3	Responstider ved HTTP-forespørslar	45
3.4	Repeterbarhetstest av systemet	46
3.5	Video	47
4	Diskusjon	47
4.1	Valg av kamera	47
4.1.1	uEye XS	48
4.2	Valg av bildeoppløsning	48
4.3	Rotert kameralinse	48
4.4	Forvrengning i kamera	49
4.4.1	Kalibrering	49
4.5	Programvare for bildefangst	49
4.5.1	Problemer med autofokus	49
4.5.2	Problemer med automatisk eksponeringstid	50
4.6	Kameraverktøy i RobotStudio	50
4.7	Valg av QR-leser	50
4.8	Plukking av klosser i høyden	51
4.8.1	Bredde på QR-koder	51
4.8.2	Annerledes kameraorientering	51
4.9	Stemmestyring av prosessen	51
4.10	Grafisk brukergrensesnitt	52
4.11	XML kontra JSON	52
4.12	Lagring av klossinformasjon i Python	53
4.13	Konklusjon	53
Referanser		i
A Vedlegg		iv
A.1	Laboratorieoppgave	v

Figurer

1	<i>Norbert</i> ved UiS' robotlaboratorium	2
2	Arbeidsobjektet i virkeligheten sammenlignet med RobotStudio	3
3	Verktøyet i virkeligheten sammenlignet med RobotStudio	3
4	Kloss med festet QR-kode	4
5	uEye UI-1007XS-C [5]	4
6	Robotoppsett og nærbilde av kameramontasje	6
7	Kommunikasjon mellom Python og RobotWare	7
8	Filstruktur i Python	8
9	Filstruktur i RAPID	9
10	Gripeteknikker	9
11	Automatisk lyskontroll ved ulike bakgrunner	13
12	Vinklet sensor, illustrert med vinkel på 15°	15
13	Vindu med kontinuerlig videovisning	16
14	Effekten av to ulike kantbevarende filter på støyfullt bilde	17
15	Normalisering av bilde i OpenCV	18
16	QR-kode i stående posisjon, med standard orientering (0°)	19
17	Nummerering av hjørner før og etter endring i <i>pyzbar</i> , her med QR-kode vinklet -118°	20
18	Eksempler på klar og hindret griperbane	21
19	Koordinatsystem til arbeidsobjekt i RAPID og bilde i OpenCV	22
20	Sammeheng mellom synsvidde og sensorstørrelse	22
21	Illustrasjon av overskyt i x- og y-retning	24
22	Illustrasjon ved rotasjon av griper med påmontert kamera	25
23	Illustrert eksempel av posisjonskalkyle	27
24	Ressurshierarkiet i Robot Web Services.[4]	29
25	En vedvarende HTTP-forbindelse har hastighetsfordel over flere forbindelser [27]	30
26	Forside på Read the Docs	35
27	Resultat av dokumentasjon i Sphinx og RST	36
28	Flytskjema over oppstart av demonstrasjonsprogram	37
29	Flytskjema over plukking og plassering av én kloss	39
30	Flytskjema over stabling av klosser	41
31	Typisk resultat fra stabling av klosser	42
32	Flytskjema over evighetsprogram	43

Tabeller

1	Tabell med oppløsninger og tilhørende “Format-ID” for uEye XS. [18]	12
2	HTTP-forespørslar ved første kommunikasjonstest	34
3	HTTP-forespørslar ved andre kommunikasjonstest	35
4	Responstider ved vanlige HTTP-forespørslar	45
5	Konfidensintervall ved vanlige HTTP-forespørslar	45
6	Responstider ved start/stopp av motorer	46
7	Konfidensintervall ved start/stopp av motorer	46

Forkortelser

API	Application programming interface
AUTO	automatisk modus
bps	bilder per sekund
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
MANR	manuell modus med redusert hastighet
OpenCV	Open Source Computer Vision Library
REST	Representational state transfer
TCP	tool center point
URI	uniform ressursidentifikator
URL	nettverksadresse
WPW	What Python Wants
XML	Extensible Markup Language

1 Introduksjon

I emnet ELE610 - Prosjekter i robotteknikk ved UiS skal antall oppgaver med ABB-robotene økes. Siden bildefangst allerede er en del av pensum i faget, anses bildebehandling i robotsystemer som å være en naturlig utvidelse til dette. Programmering i Python er sentralt i emnets andre deler, og det kan være fordelaktig å implementere dette også i ABB-delen av pensum.

Roboter med maskinsyn er veldig relevant i dagens industri. For å realisere dette er det nødvendig å etablere kommunikasjon mellom kameraets programvare og roboten. ABB har utviklet et grensesnitt, Robot Web Services, som gjør dette mulig.

1.1 Oppgavetekst

Oppgaven går ut på å sette et kamera på griperen til en ABB-robot av typen IRB 140 som gjør systemet i stand til å lokalisere, identifisere, orientere og stable klosser som er tilfeldig plassert i arbeidsområdet (på en pult ved siden av roboten).

Dette skal realiseres ved:

- Bildeoverføring fra kamera til PC
- Bildebehandling/-analyse i Python 3.7 med OpenCV
- Kommunikasjon mellom Python og RobotWare ved bruk av Rest API

Ved fullførelse av systemet, skal en laboratorieoppgave i faget ELE610 utformes. Denne skal inneholde eksempelkode som gir et rammeverk og utgangspunkt for løsningen. Det skal også utvikles et program egnet for demonstrasjon av robotsyn og som kan benyttes som (et avansert) løsningsforslag for laboppgaven.

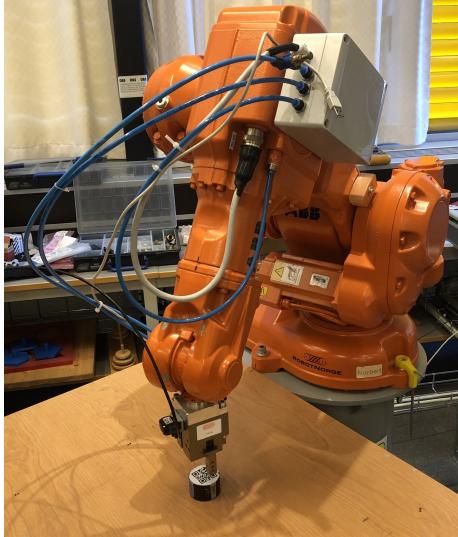
Videre, om tiden tillater, kan oppgaven utvides med stemmestyring av prosessen. Dette vil illustrere en robot med både “syn og hørsel”.

1.2 Bakgrunn

For å løse oppgaven, tas det utgangspunkt i en del ferdiglaget maskin- og programvare. Disse nevnes og forklares i dette kapittelet.

1.2.1 Robot

Universitetet i Stavanger har en robotlab med to roboter navngitt *Norbert* og *Rudolf*. Robotene er av typen ABB IRB 140 som begge kan utstyres med ulike verktøy etter behov. I denne oppgaven brukes roboten *Norbert*, som er utstyrt med en griper som muliggjør plukking av klosser.



Figur 1: *Norbert* ved UiS' robotlaboratorium

RobotWare er en samlebetegnelse for all programvare som er installert på robotsystemet for å styre roboten.

RobotStudio er et simulerings- og programmeringsverktøy, laget av ABB, som muliggjør testing og programmering mot både reelle og virtuelle roboter. [1] RobotStudio har en innebygget editor som brukes til å skrive programmer i programmeringsspråket RAPID. Her kan det defineres arbeidsobjekt, verktøy og variabler. [2]

FlexPendanten er et betjeningspanel. Den er utstyrt med en berøringsskjerm og styreknapper for kontrollering av robot. Panelet er tilknyttet den fysiske robotkontrolleren til den respektive roboten. Det finnes også en virtuell versjon av denne i RobotStudio. [3]

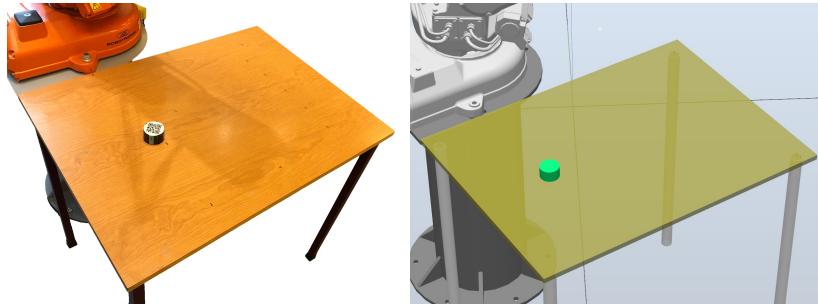
Robotkontrolleren er styreskapet med elektronikk og programvare som styrer roboten. [3]

Arbeidsobjektet som er benyttet i oppgaven var ferdig laget i RobotStudio fra

faget ELE610 ved UiS. [3] Arbeidsobjektet er *wobjTableN* og er definert som:

```
1 wobjTableN:=[FALSE ,TRUE ,"" ,[[150 ,-500 ,8] ,
2 [0.707106781 ,0 ,0 ,-0.707106781]] ,[[0 ,0 ,0],[1 ,0 ,0 ,0]]];
```

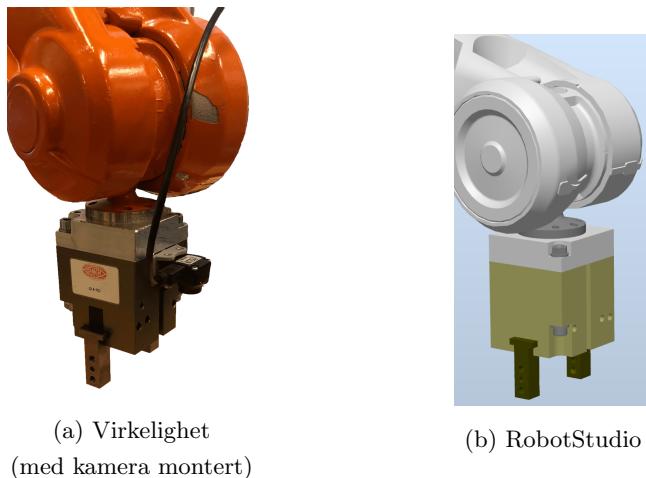
Arbeidsobjektet er bordet vist i figur 2. Dimensjonene på bordet er 800x600mm.



Figur 2: Arbeidsobjektet i virkeligheten sammenlignet med RobotStudio

Verktøyet som er benyttet i oppgaven var både ferdig montert på roboten og laget i RobotStudio fra faget ELE610 ved UiS. [3] Verktøyet som er brukt var *tGripper* og er definert som:

```
1 tGripper:=[TRUE ,[[0 ,0 ,114.25],[0 ,0 ,0 ,1]] ,
2 [1 ,[-0.095984607 ,0.082520613 ,38.69176324],[1 ,0 ,0 ,0] ,0 ,0 ,0]];
```



Figur 3: Verktøyet i virkeligheten sammenlignet med RobotStudio

Et *robtarget* er en dataklasse som brukes i RAPID. Den beskriver en posisjon og ønsket orientering som roboten kan bevege verktøyet til, relativt til arbeidsobjektet. [3] Lokaliserte klosser representeres ved *robtargets*.

Klossene som er benyttet i oppgaven var ferdig 3D-printet fra faget ELE610 ved UiS. Klossene har en diameter på 50mm og en høyde på 30mm.



Figur 4: Kloss med festet QR-kode

1.2.2 Robot Web Services

Robot Web Services er et sett med flere grensesnitt basert på REST-prinsippet (“Representational state transfer”). Gjennom disse tjenestene er det mulig å sende HTTP-forespørsler (“Hypertext Transfer Protocol”) mot RobotWare, som muliggjør kommunikasjon mellom RobotWare og for eksempel Python. [4]

1.2.3 Kamera

Kameraet som brukes i oppgaven er “UI-10007XS-C” (uEye XS), levert av IDS Imaging (se figur 5). Kameraet er lite (26.5 x 23 x 21.5 mm) og veier 12 gram. Det er enkelt å ta i bruk via USB-tilkobling til PC. Kamera har en maksoppløsning på 2592 x 1944. Det har også flere funksjoner som autofokus, fargekorreksjon og automatisk lyskontroll. [5]

Kameraet sitter fast i en 3D-printet montasje (laget av Ståle Freyer) som kan festes på gripermodulen på robotarmen (se figur 6b). Som et resultat, er kameraet posisjonert 55mm foran og 70mm over griperens *TCP* (“tool center point”).



Figur 5: uEye UI-1007XS-C [5]

1.2.4 Python-pakker

Flere ulike bibliotek i Python blir benyttet i denne oppgaven. Noen av de mest kjente er *NumPy*, *OpenCV* (“Open Source Computer Vision Library”) og *configparser*. [6] [7] [8]

ZBar brukes for å lese QR-koder i oppgaven. Den siste versjonen (versjon 0.10) av ZBar ble utgitt i 2009 og fungerer bare i Python 2. For å bruke denne programvaren, benyttes derfor *pyzbar* som er et wrapper-bibliotek rundt ZBar og som støtter Python 3. [9] [10]

PyuEye er et bibliotek med Python-bindinger for IDS Imaging’s *uEye API* (“Application programming interface”). Denne API-en gir støtte for de ulike kameramodellene som produseres av IDS. Gjennom *PyuEye* er det mulig å ta full kontroll over IDS sine uEye-kamera i Python. [11] [12]

Requests er et brukervennlig HTTP-bibliotek for Python som gjør det enkelt å sende HTTP-forespørsler. All kode for kommunikasjon mellom Python og RobotWare, realiseres med funksjoner fra *Requests*-biblioteket. [13]

Locust er et brukervennlig hjelpemiddel for å teste brukerbelastning og respons-tid. Responstidene måles i millisekunder og kan leses ut og lastes ned ved hjelp av grensesnittet over nettleseren. Locust bruker *Requests*-biblioteket for HTTP-forespørsler. [14]

Sphinx benyttes for å skrive kodedokumentasjon for Python. Nøkkelfunksjoner inkluderer kryssreferanser, kodeutheving, hierarkistruktur og mulighet for både HTML- (“HyperText Markup Language”) og PDF-format. *Sphinx* bruker *reStructuredText* (RST) som markeringsspråk. [15] [16]

1.2.5 JSON

JSON (“JavaScript Object Notation”) brukes for å parse og lese svar på HTTP-forespørslene som sendes via *Robot Web Services*. JSON er et lettleselig data-utvekslingsformat som er bygd på to hovedstrukturer: navn-verdi-par (ordlister i Python) og sorterte lister med verdier (lister i Python). [17] Formatet er uavhengig av programmeringsspråk og kan brukes i Python gjennom standardbiblioteket.

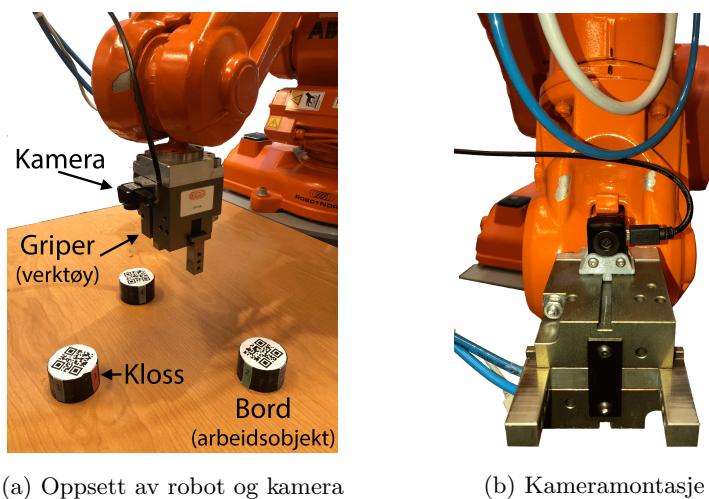


2 Konstruksjon

Konstruksjonsdelen beskriver hvordan demonstrasjonsprogrammet ble laget. Det gis innledningsvis et innblikk i systemets overordnede struktur i form av maskin- og programvare. Hvert moment i oppgaveløsning gjennomgås også i detalj. Dette inkluderer bruk av kamera, lesing av QR-koder, posisjonskalkulering, og kommunikasjon mellom Python og RobotWare.

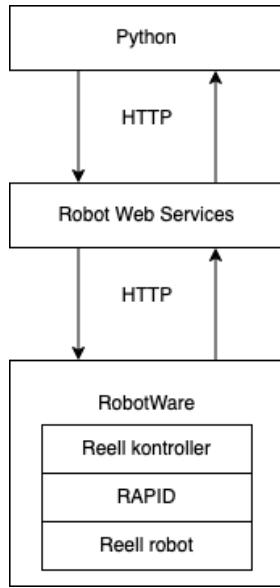
2.1 Overordnet system

Systemet er bygget opp av en ABB IRB 140 robot med en griper, et trebord og 3D-printede klosser med QR-kode. Kameraet er plassert på fremsiden av griperen ved hjelp av en 3D-printet montasje, og ser ned på bordet med et fugleperspektiv.



Figur 6: Robotoppsett og nærbilde av kameramontasje

For å kunne styre roboten ved bruk av Python brukes Robot Web Services som er bygget på arkitekturstilen REST. Kommunikasjonsflyten er vist i figur 7 under.



Figur 7: Kommunikasjon mellom Python og RobotWare

Kommunikasjonen er nettverksbasert og utføres med HTTP. Denne kommunikasjonsflyten er enkel å forholde seg til, da en sender én forespørsel for hver ressurs en ønsker å gjøre noe med. Dette gjøres ved hjelp av en nettverksadresse (URL) og et verb som sier noe om hva som skal utføres på ressursen. Kommunikasjonen blir forklart i mer detalj i kapittel 2.5.

2.1.1 Robotens operasjonsmodi

Roboten har flere operasjonsmodi og to av disse, *MANR* (manuell modus med redusert hastighet) og *AUTO* (automatisk modus), ble brukt under oppgaveløsningen.



Manuell modus utmerker seg til testing og feilsøking, siden roboten enkelt kan stoppes og hastigheten har et lavt maksimum.

Når roboten styres fra Python, er det hensiktsmessig å la den stå i automatisk modus. I automatisk modus er det ikke nødvendig å bruke FlexPendanten i det hele tatt. Motorer kan slås av og på, RAPID-program kan startes, og mye mer kan gjøres direkte fra Python. På grunnlag av dette kjøres demonstrasjonsprogrammet hovedsakelig i *AUTO*.

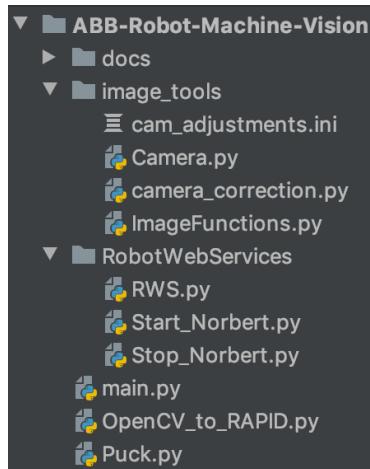
2.1.2 Filstruktur i Python og RAPID



Filstrukturen i Python og RAPID preges av et ønske om å utføre flest mulige operasjoner i ett språk. På denne måten vil kommunikasjon begrenses til et minimum. Som et resultat av dette, vil lesbarhet øke og feilsøking forenkles.

Python er et meget utbredt programmeringsspråk og har gode hjelperessurser på nett. Dette gjør at språket er enkelt å bruke, og har bred funksjonalitet. RAPID, til sammenligning, har et begrenset utvalg av slike ressurser. I flere sammenhenger, fører dette til at språket er vanskelig å bruke. I tillegg har det flere begrensninger enn Python. Det er derfor nødvendig å utføre for eksempel bildefangst og -behandling i Python.

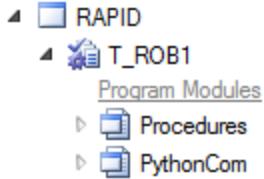
Med bakgrunn i denne sammenligningen, velges det å hovedsakelig bruke Python som programmeringsspråk.



Figur 8: Filstruktur i Python

I figur 8, over, vises filstrukturen fra Python-prosjektet. *image_tools* inkluderer alle funksjoner som omhandler kamera, bildefangst, og lesing av QR-koder. Mappen *RobotWebServices* inneholder det som er nødvendig for å kommunisere med roboten. Her inkluderes også funksjoner for start og stopp av *Norberts* motorer. *OpenCV_to_RAPID.py* inneholder transformasjoner og kompenseringer som er nødvendige for kalkulering av *robtargets*. *Puck* inneholder *Puck*-klassen som brukes for å lagre nøkkelinformasjon om hver enkelt kloss. Python-delen av demonstrasjonsprogrammet ligger i *main.py*. Alle delene i filstrukturen gjennomgås i ulike deler av rapporten. Den uåpnede mappen, (*docs*), gjelder dokumentasjon.

mentasjon av koden og blir presentert i kapittel 2.6.

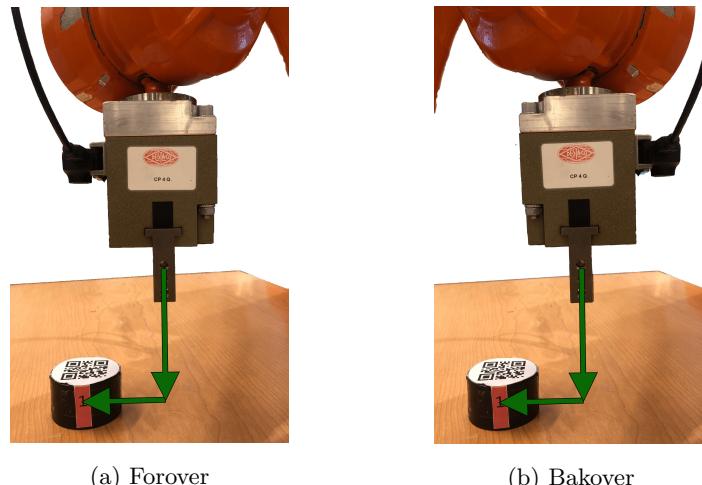


Figur 9: Filstruktur i RAPID

Prosjektet i RAPID har en enkel struktur. Den er bestående av to moduler, *Procedures* og *PythonCom*, som inneholder funksjoner og hovedprogram henholdsvis.

2.1.3 Gripeteknikk

Marginen mellom griper og kloss er 1mm på hver side. Med bakgrunn i dette brukes en bestemt gripeteknikk for å plukke klosser. I stedet for å la griperen gå ovenfra og ned mot klossene, går den først ned og deretter inn mot klossene fra siden. Med andre ord når klossens posisjon er bestemt, går griper til gripehøyde (10mm over arbeidsobjektet) enten 55mm foran eller bak klossen, før den “glir” inn mot klossen og plukker. Ved små avvik vil det dermed være liten fare for harde kollisjoner. Ved å plukke på andre måter risikeres dette.



Figur 10: Gripeteknikker

Gripeteknikken er illustrert ovenfor, legg spesielt merke til kameraets posisjon i forhold til klossen. I begge tilfeller er posisjonen til klossen allerede hentet

ut. Griperetningen blir bestemt gjennom kollisjonshindringen i kapittel 2.3.3. Gripeteknikken i figur 10a er den mest vanlige. Her beveger griperen seg, som nevnt tidligere, først ned mot bordet, og så inn mot klossen. I figur 10b har griperen allerede bevegd seg over klossen og er klar for å gå ned til gripeposisjon, for å så “rygge” bakover mot klossen.

2.2 Kamera

Kamera er montert på gripermodulen på enden av robotarmen. Det er dermed mulig å ta bilder i forskjellige høyder og posisjoner. Roboten har en begrenset rekkevidde, og med gripermodulen koblet på, kan den maksimalt nå en kamerahøyde på ca. 670mm når kameraet er sentrert over arbeidsobjektet.

Det er viktig at bildefangsten foregår med riktige kameraparametre. Dette innebærer å stille riktig brennvidde og riktig eksponeringstid på kameraet. uEye XS har automatisk kontroll for begge disse, men i oppgaven stilles de manuelt etter behov for å øke systemets robusthet. Dette omtales i kapittel 2.2.4 & 2.2.5.

uEye XS har en viss toleranse for posisjonsnøyaktigheten til sensoren i kamerahuset. Sensoren kan være både tiltet og rotert, som kan føre til avvik i kalkulerte klossposisjoner. Disse omtales i kapittel 2.2.6 & 4.3 henholdsvis.

2.2.1 Camera-klassen

Kameraet brukes i Python ved hjelp av IDS sin egen API, *uEye*. [11] Denne er skrevet i C++. Derfor brukes Python-biblioteket *PyuEye*, som inneholder Python-bindinger for *uEye*. [12]

Det er laget en klasse i Python, *Camera*, som gjør nytte av dette biblioteket på ...inneholder det nødvendige for å bruke kameraet. Her brukes alle metodeene klassen inneholder:

```
1 # Initialization
2 >>> cam = Camera()
3 >>> cam.init()
4 >>> cam.set_parameters()
5 >>> cam.allocate_memory()
6 >>> cam.capture_video()
7
8 >>> image = cam.get_image()
9
10 >>> cam.exit_camera()
```

I funksjonen *set_parameters* blir formatet satt, automatisk fokus og eksponering deaktivert og en ny spesifikk eksponeringsverdi satt. Funksjonen *get_image* kan også brukes i en løkke for å vise video. Det tar vanligvis 5-6 sekunder å initialisere kameraet. Derfor gjøres det kun én gang, ved oppstart.

2.2.2 Bildefangst

Som nevnt kan roboten oppnå en maksimal kamerahøyde på ca. 670mm. Ved ligning 1, gir dette et horisontalt synsfelt lik ca. 657mm og vertikalt synsfelt lik 493mm. Arbeidsobjektet har dimensjoner 800x600mm, som betyr at det ikke er mulig å vise hele bordet i ett bilde. Det er dog mulig å dele opp bordet i flere bilder, slik at alt dekkes.

Selv med en maksimal kamerahøyde på ca. 670mm, er det valgt å plassere kamera 570mm over arbeidsobjektet. Dette gir et horisontalt synsfelt lik ca. 559mm og vertikalt synsfelt lik 419mm. En økning i høyde utover dette gir merkbar reduksjon i QR-leserens suksessrate. Dette kan løses ved å øke bildeoppløsning (se tabell 1), men det ble ikke sett på som hensiktsmessig. Årsaken diskuteres i kapittel 4.2.

For å bestemme klossenes generelle beliggenhet, tas bilde(r) med kamera 570mm over arbeidsobjektet. For å nærmere bestemme klossenes posisjon, tas alltid et nærbilde av hver enkelt kloss som skal plukkes. I nærbildeposisjon, er kamera 100mm over den gjeldene klossen. Dette bidrar til å øke nøyaktigheten i klossposisjonen til et tilfredsstillende nivå.

2.2.3 Bildeoppløsning og -format

Kameraet som brukes støtter flere format og har et sett med ulike oppløsninger som vist i tabellen over. Kameraets sensor er 4:3 og for å utnytte dette brukes naturligvis en oppløsning med 4:3-format. I oppgaven brukes en oppløsning på 1280x960, men andre 4:3-oppløsninger er også tilgjengelige, som vist i tabell 1. Denne oppløsningen tilfredsstilte våre krav med tanke på QR-deteksjon.

Format-ID	Oppløsning	Navn
4	2592 x 1944	5M
5	2048 x 1536	3M
6	1920 x 1080	Full HD 16:9
8	1280 x 960	1.2M 4:3
9	1280 x 720	HD 16:9
12	800 x 480	WVGA
13	640 x 480	VGA
20	1600 x 1200	UXGA
31	640 x 480	VGA 30 fps
32	800 x 480	WVGA 30 fps

Tabell 1: Tabell med oppløsninger og tilhørende “Format-ID” for uEye XS. [18]

2.2.4 Innstilling av fokus

For å oppnå et mest mulig robust resultat, er løpende autofokus ikke brukt, og slås av når parametre for kamera settes ved oppstart. I stedet benyttes manuelt fokus så langt det er mulig. I tilfeller hvor manuelt fokus ikke kan brukes, trigges autofocus én gang for å ta bilde.

For å bruke manuelt fokus, må korrekt brennvidde være kjent. Korrekt brennvidde avhenger av arbeidsavstanden (avstand fra kameralinse til objekt). Det er derfor mulig å lage en karakteristikk som binder disse sammen. Ifølge IDS Imaging, er korrekt brennvidde ikke nødvendigvis lik i ulike kamera av samme modell. [19] Det er derfor ikke mulig å lage en *generell* karakteristikk. Det ble derfor i stedet laget en *spesifikk* karakteristikk for kameraet som brukes i løsningen.

For å lage karakteristikken, ble det tatt bilder med arbeidsavstand fra 620mm til 20mm, med inkrementer på 15mm (totalt 40 bilder). For hver arbeidsavstand, ble autofocus trigget én gang for å få korrekt brennvidde. Deretter ble verdiene lagret som verdipar: [arbeidsavstand, brennvidde].

Alle uEye XS har fokusverdier fra 112 til 240, hvor 112 gir minst mulig brennvidde i kamera. Ved bildefangst fra standard høyde over arbeidsområdet (570mm), kreves en fokusverdi på rundt 204.

Om nødvendig, kan en “tvinge” kameraet til å refokusere med:

```

1 >>> ueye.is_Focus(cam.handle(), ueye.
    FOC_CMD_SET_ENABLE_AUTOFOCUS_ONCE, None, 0)

```

Dette unngås om mulig, siden det både er tregt og kan gi fokus på annet enn objektet.

2.2.5 Eksponeringstid

Av samme grunner som i kapittel 2.2.4, brukes ikke automatisk eksponeringstid i løsningen.



(a) Vanlig bakgrunn

(b) Blå duk som bakgrunn

Figur 11: Automatisk lyskontroll ved ulike bakgrunner

Endring i lysforhold og/eller bakgrunn kan gi opphav til vanskeligheter i lesing av QR-koder. For å kunne være sikker på at leseren oppfatter QR-kodene ble det implementert en rutine som bestemmer tilstrekkelig eksponeringstid. Denne rutinen bør kjøres før kjøring av hovedprogrammet om lysforhold eller bakgrunn endres.

I rutinen tas det bilder ved alle eksponeringstider. Bildene sendes til QR-leseren, som returnerer en liste over alle QR-koder som ble lest. Deretter kan eksponeringstid og antall leste QR-koder lagres sammen i en liste, “*value_pairs*”. Det tas bilder med eksponeringstid fra 2-66ms, med inkrementer på 2ms.

Når alle bilder er tatt, blir en vektet sum regnet ut, og den riktige eksponeringstiden bestemmes:

```

1 >>> weighted_sum = 0
2 >>> pucks_found = 0
3 >>> for value_pair in value_pairs:
4 ...     pucks_found += value_pair[1]
5 ...     weighted_sum += value_pair[0] * value_pair[1]
6

```

```
7 >>> exposure = str(int(weighted_sum / pucks_found)) # Correct
      exposure
```

Rutinen er gjort tilgjengelig som en funksjon, *find_correct_exposure*, i Python-prosjektet. Denne vil bli kjørt dersom lysforholdene eller bakgrunnen endres under kjøring av evighetsprogrammet (omtalt i kap. 3.1.3). QR-leseren ikke finner en kloss etter å ha tatt et forhåndsbestemt antall bilder, så vil rutinen bli utført slik at eksponeringstiden endres og programmet kan fortsette.

2.2.6 Korrigering av tiltet kamerasensor

Både kameraets sensor og kamerahusets montasje (figur 6b) er vinklet en ukjent mengde. Sensoren kan være vinklet og toleransen er $\pm 1^\circ$. [20] Det må også korrigeres for avvik fra rette vinkler i montasjen. Denne vinkelen er det ikke funnet grenseverdier for. Dette gir et maksimalt feilbidrag i klossposisjon fra sensor på:

$$h \cdot \tan^{-1}(1^\circ + \angle_{montasje}),$$

hvor h er kameraets høyde over objektet. Ved oversiktsbilde vil denne vinkelen være irrelevant, ettersom nøyaktig posisjon ikke kreves. Ved nærbilde (100 mm over klossen), derimot, kreves maksimal nøyaktighet. Maksimalt feilbidrag fra sensors vinkel ved nærbilde er:

$$(100 \text{ mm}) \cdot \tan^{-1}(1^\circ + \angle_{montasje}) > 1.745 \text{ mm}.$$

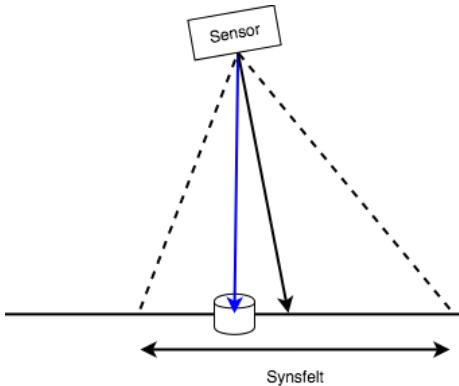
Med en feilmargin på 1mm ved plukking av klosser, må dette feilbidraget fjernes.

Det er laget en justeringsroutine, *camera_adjustment*, for å bestemme i hvilken grad kameraets sensor er vinklet. Justeringsruten bestemmer den totale vinkelen mellom sensoren og arbeidsområdet, uttrykt som stigningstall i x og y . Stigningstallene bestemmes ved sammenligning av to bilder av samme kloss fra ulik høyde. Fra disse to bildene lages to robttargets, ved metoden i kapittel 2.4. Ved vinkel $\neq 0^\circ$ mellom sensor og arbeidsområde, vil disse to robttargets være ulike. Ut fra disse, bestemmes Δx og Δy . Δh behøves også:

$$\Delta h = h_{høy} - h_{lav}$$

$$\Delta x = x_{høy} - x_{lav} \longrightarrow stigning_x = \frac{\Delta x}{\Delta h}$$

$$\Delta y = y_{høy} - y_{lav} \longrightarrow stigning_y = \frac{\Delta y}{\Delta h}$$



Figur 12: Vinklet sensor, illustrert med vinkel på 15°

I figur 12 er den blå pilen z-aksen til arbeidsområdet, mens den sorte pilen viser til midten av synsfeltet i kameraet. Selv om klossen står direkte under kamera, vil den oppfattes som å være til siden av kamera.

Denne rutinen er gjort tilgjengelig som en funksjon, *camera_adjustment*, i Python-prosjektet. Rutinen bør kjøres hver gang kamera byttes ut eller montasjen justeres. Dersom det oppleves at griperens nøyaktighet er lav, bør den også kjøres.

2.2.7 Lagring av kamerainnstillinger

I oppgaven brukes en INI-fil (konfigurasjonsfil), *cam_adjustments.ini*, som inneholder eksponeringstid funnet ved metoden i kapittel 2.2.5 og stigningstall funnet ved metoden i kapittel 2.2.6. Slik kan typisk *cam_adjustments.ini* se ut:

```

1 [EXPOSURE]
2 exposure = 34.0
3
4 [SLOPE]
5 slopex = 0.0243
6 slopey = 0.0043

```

Ved endrede lysforhold eller ny oppkobling av kamera, bør en initialiseringsroutine kjøres før oppstart av hovedprogrammet. Denne rutinen bestemmer riktig eksponeringstid og finner korrigeringsfaktoren til kameraet. Ved lange kjøretider, kan det også være nødvendig å oppdatere eksponeringsverdien under kjøring. For eksempel ved endring i lysforhold under kjøring. Initialiseringsrutinen blir da automatisk startet mens hovedprogrammet går i pause.

For å lese og skrive til denne filen, lages et ConfigParser-objekt som finnes i *configparser* fra Pythons standardbibliotek. [8] Ved bruk av dette kan verdiene i INI-filen oppdateres og de nye verdiene brukes både før og under kjøring.

2.2.8 Kontinuerlig videovisning

Under kjøring av demonstrasjonsprogrammet, vises alltid video i et separat vindu. Videoen hentes fra kameraet som er montert på griperen, slik at brukeren kan se hva som foregår fra robotens perspektiv. Bildene som vises blir sendt til QR-leseren slik at de behandles og QR-kodene kan markeres. Dette brukes både for å skape et visuelt appellerende program, samt for å feilsøke under programutførelse.

Med en bildeoppløsning på 1280x960 er det mulig å ta opp video med 15 bilder per sekund (bps). Videoen vises gjennom OpenCV's *imshow*-funksjon i en evig *while*-løkke.



Figur 13: Vindu med kontinuerlig videovisning

2.3 Lesing av QR-koder

Klossenes posisjoner bestemmes ved å lese QR-koder. Disse leses i Python gjennom *pyzbar*-biblioteket. For å sikre en høy suksessrate i QR-lesing, behandles bildene med OpenCV-funksjoner på forhånd.

2.3.1 Bildebehandling

Målet med bildebehandling er å fjerne støy og øke kontrast i bilder, uten å redusere QR-kodenes lesbarhet. Resultatet av dette er en høyere suksessrate ved QR-lesing økes.

For å bevare lesbarheten i QR-koden, er det bare enkelte støyfjerningsmetoder som kan brukes på bildet. Ved bruk av feil type filter kan kantene i bildet bli uskarpe og QR-koden ødelegges. Filteret som skal brukes må derfor være kant- og hjørnebevarende. Spesielt to funksjoner i OpenCV's repertoar møter kriteriet om å være kantbevarende: *bilateralFilter* og *medianBlur*. Andre vanlige filtre, som *GaussianBlur*, møter ikke dette kriteriet. [21]

Av funksjonene som fremstår i OpenCV, er det spesielt en som utmerker seg i hjørnebevaring, nemlig *bilateralFilter*.

I figur 14 sammenlignes de to kantbevarende filtrene *bilateralFilter* og *medianBlur*. Parametrene som er brukt under sammenligning er:

```

1 medianBlur = cv2.medianBlur(src=noisy_image, ksize=3)
2 bilateralFilter = cv2.bilateralFilter(src=noisy_image, d=3,
sigmaColor=75, sigmaSpace=75)
```



Figur 14: Effekten av to ulike kantbevarende filter på støyfullt bilde

Sammenligningen mellom disse to filtrene bekrefter at *bilateralFilter* bevarer både kanter og hjørner. *medianBlur*, derimot, har bare bevart kanter, men ikke hjørner. Bildet som er brukt, er tatt fra vanlig oversiktsposisjon, med kamera 570mm over arbeidsobjektet. I tillegg brukes minst mulig styrke i *medianBlur*, som er **ksize=3** ("kernel size"). Det er dermed ikke mulig å redusere filtreringsstyrken i *medianBlur*. *bilateralFilter*, derimot, har flere frihetsgrader og kan, hvis ønskelig, reduseres i styrke. Med basis i sammenligningen brukes *bilateralFilter* for å fjerne støy i bilder.

For å øke kontrast i bildene, brukes *normalize*. Denne funksjonen normaliserer pikselverdiene (fargen) i bildet.



Figur 15: Normalisering av bilde i OpenCV

2.3.2 Programvare for lesing av QR-koder

Kun én funksjon fra *pyzbar* er nødvendig for å lese QR-koder, nemlig *decode*:

```

1 >>> from pyzbar.pyzbar import decode
2 ...
3 >>> data = decode(image)
4 >>> data
5 Decoded(
6     data=b'Puck#1', type='QRCODE',
7     rect=Rect(left=27, top=27, width=145, height=145),
8     polygon=[Point(x=27, y=27), Point(x=27, y=172), Point(x=172, y
9         =172), Point(x=172, y=27)])

```

decode returnerer et *Decoded*-objekt. Det inneholder informasjonen fra QR-koden, i “*data*”-parameteren, samt koordinatene til alle dens hjørner, i “*polygon*”-parameteren.

Informasjonen fra QR-kodene representeres i *bytes* og må konverteres til en streng slik at klossnummeret kan hentes ut:

```

1 >>> data.data
2 b'Puck#1'
3 >>> data_string = str(data.data, "utf-8")
4 >>> puck_number = int(''.join(filter(str.isdigit, puck_string)))
5 >>> puck_number
6 1

```

Punktene i *polygon*-parameteren brukes for å bestemme QR-kodenes midtpunkt og orientering i bildet:

```

1 >>> data.polygon
2 [Point(x=27, y=27), Point(x=27, y=172), Point(x=172, y=172), Point(x
3     =172, y=27)]
3 >>> x1 = data.polygon[0][0]

```

```

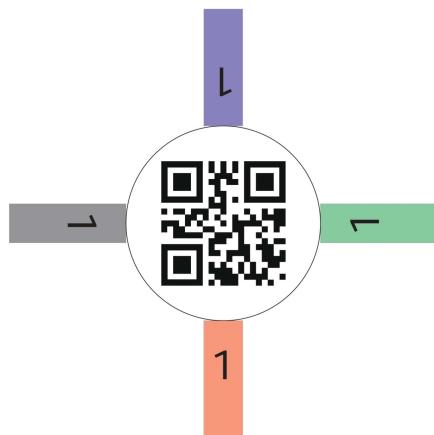
4 >>> y1 = data.polygon[0][1]
5 >>> x2 = data.polygon[3][0]
6 >>> y2 = data.polygon[3][1]
7
8 >>> angle = np.rad2deg(np.arctan2(-(y2 - y1), x2 - x1))
9 >>> angle
10 0.0
11
12 >>> x = [p[0] for p in points]
13 >>> y = [p[1] for p in points]
14 >>> position = [sum(x) / len(points), sum(y) / len(points)]
15 >>> position
16 [99.5, 99.5]

```

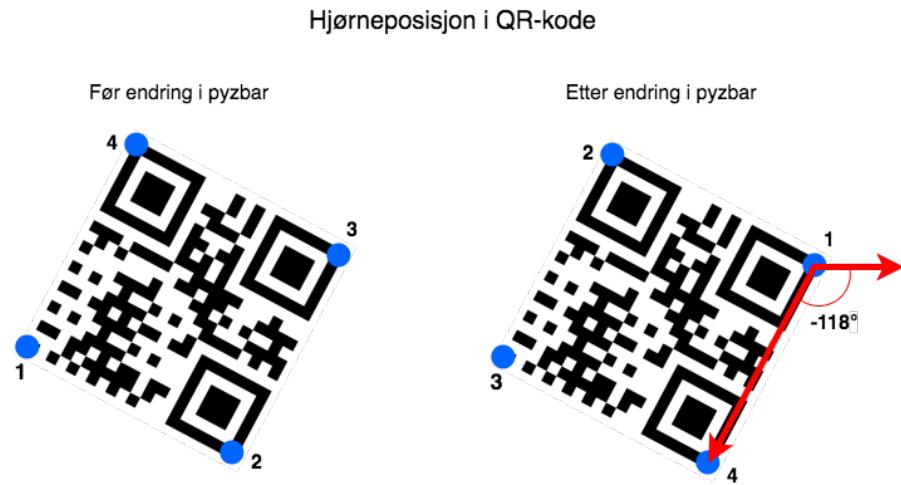
For å relatere QR-kodens posisjon til midten av bildet, trekkes halve bildeoppløsningen fra posisjonen, slik at origo flyttes fra øvre venstre hjørne, til midten av bildet.

For hver QR-kode som leses, skapes et *Puck*-objekt (ref. kap. 2.3.3). Etter alle QR-koder i bildet er lest, returneres en liste med alle skapte *Puck*-objekter.

pyzbar gir posisjonen til hvert hjørne av QR-koden. Disse posisjonene blir gitt i en ordnet rekkefølge: hjørnet lengst til venstre først, og deretter resten av hjørnene mot klokka. Dette gjør det umulig å hente ut orienteringen til koden. Dermed må en endring gjøres i *pyzbar*-biblioteket, slik at det samme hjørnet alltid gis først av *pyzbar*. [22]



Figur 16: QR-kode i stående posisjon, med standard orientering (0°)



Figur 17: Nummerering av hjørner før og etter endring i *pyzbar*, her med QR-kode vinklet -118°

2.3.3 *Puck*-klassen

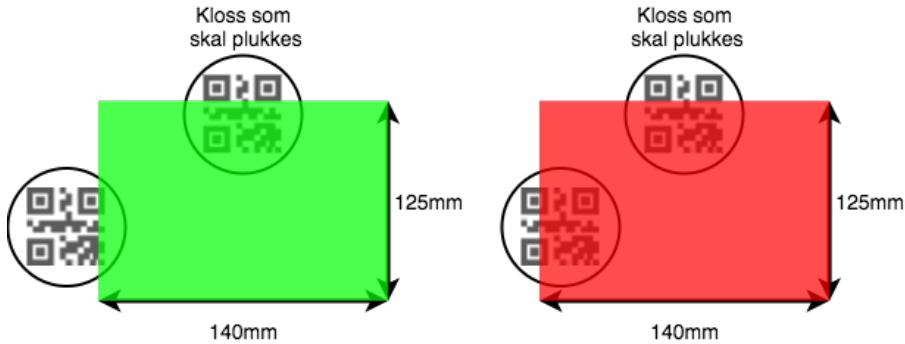
Gjennom skaping av *Puck*-objekter i Python, lagres nøkkelinformasjon om klossene. Dette omfatter nummer, posisjon, orientering og høyde. På denne måten er det enklere å forholde seg til de ulike klossene på bordet.

Alle parametre bortsett fra høyden må initialiseres ved skaping av et nytt objekt. Hvis en kloss allerede er initialisert, kan du ved hjelp av klargjorte funksjoner enkelt oppdatere posisjonen, orienteringen eller høyden. [23]

I tillegg inneholder *puck*-klassen funksjonalitet som skal hindre kollisjon mellom griper og andre klosser på bordet når en kloss skal plukkes. Med metoden som beskrives i kapittel 2.1.3 skapes en “bane” for griperen. Banen må være fri for klosser for å unngå kollisjon. Funksjonen som brukes er *check_collision*, som tar inn en liste med klosser. I funksjonen skapes et rektangel med utgangspunkt i klossen som skal plukkes. Rektangelet er 140mm bredt og 125mm høyt og angir størrelsen på griperbanen. Størrelsen på rektangelet er bestemt basert på størrelse på griper, lengden på griperbanen og størrelsen på klossene. Dersom andre klosser enn den som skal plukkes befinner seg i banen, justeres banen i funksjonen. Alle klosser roteres rundt klossen som skal plukkes, helt til banen er kollisjonsfri. Den første vinkelen som gir kollisjonsfri bane lagres i funksjonen.

Vinkelen som returneres bestemmer hvordan klossen skal plukkes. Ved vinkler mellom $[-90, 90]$ gripes klossen på vanlig måte. For vinkler mellom $[-90, -180]$ og

[90,180] beveger griperen seg først forover og glir inn mot klossen bakover. Vinkelet oppdateres slik at klossen fortsatt treffes (se siste del i videoen i kapittel 3.5). Slik begrenses rotasjon til et minimum.



Figur 18: Eksempler på klar og hindret griperbane

Griperbanen defineres som fri så lenge ingen klossposisjoner befinner seg innenfor det definerte rektangelet. Klossposisjonen regnes som sentrum i klossen. Det som dermed avjør om en bane er hindret eller ikke, er om sentrum av en kloss er innenfor rektangelet. Dette vises i figur 18.

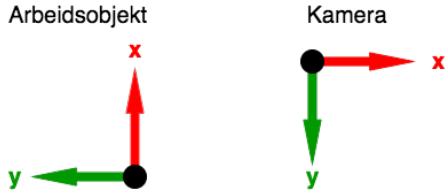
2.4 Posisjonskalkulering

Denne delen beskriver metodene som fremstår i Pythonfilen *OpenCV_to_RAPID.py*. En funksjon i denne filen, *create_robtarget*, bruker alle disse metodene for å konvertere koordinater fra OpenCV (som funnet ved lesing av QR-koder, se kap. 2.3) til koordinater på arbeidsobjektet (*robtargets*).

Posisjonskalkylen er både posisjons- og orienteringsuavhengig. Det betyr at griperen kan ha en vilkårlig posisjon i arbeidsområdet, samt være rotert en vilkårlig mengde rundt sin egen z-akse uten at posisjonskalkylen kompromitteres.

2.4.1 Transformasjon av koordinater

Bilder i OpenCV representeres i matriseform, og ”koordinatsystemet” i OpenCV følger samme struktur. Det betyr at aksene i OpenCV er som vist i figur 19 og med origo i øvre venstre hjørne. Arbeidsobjektet, *WobjTableN* i RAPID, bruker et annet koordinatsystem. For å relatere posisjoner i bilder til arbeidsobjektet, må derfor koordinatene transformeres.

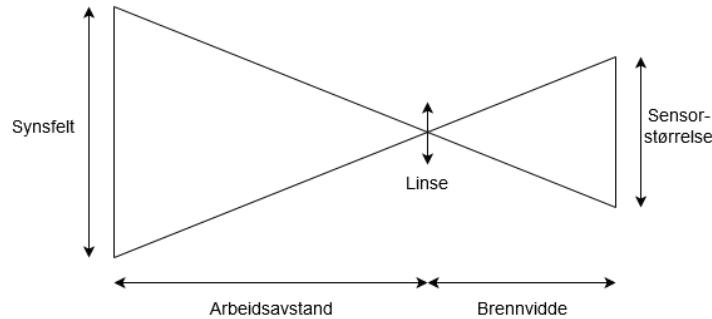


Figur 19: Koordinatsystem til arbeidsobjekt i RAPID og bilde i OpenCV

Transformasjonen koordinatene må gjennomgå, er følgende: $x \rightarrow y$, $y \rightarrow x$, $x \rightarrow -x$, $y \rightarrow -y$. Dersom griperen (og dermed kameraet) er rotert, må koordinatene også roteres en tilsvarende mengde. Rotasjonsmengden (θ_z) finnes ved metoden i kapittel 2.4.4 og formel 2 & 3.

2.4.2 Konvertere fra piksler til millimeter

Posisjonen på klossene som leses ut fra bildene, bruker bildepiksler som skala. Før disse posisjonene sendes til RobotWare, må de transformeres til millimeterskala. Transformasjonen avhenger av høyden fra kamera til arbeidsobjektet og synsvinkelen til kameraet. Disse kan relateres slik:



Figur 20: Sammeheng mellom synsvidde og sensorstørrelse

$$\frac{\text{sensorbredde}}{\text{synsvinkel_bredde}_{mm}} = \frac{\text{brennvidde}}{\text{høyde}} \quad (1)$$

Med utgangspunkt i ligning 1, kan synsvinkelen i millimeter regnes ut. Forholdet mellom denne synsvinkelen og synsvinkelen i piksler brukes for å konvertere fra piksler til millimeter:

$$\text{piksler_til_millimeter} = \frac{\text{synsvinkel_bredde}_{mm}}{\text{oppløsning_bredde}}$$

uEye XS har en brennvidde på ca. 3.7mm ($\pm 5\%$)¹ og sensorbredde på 3.6288mm (oppgett av produsenten på e-mail).

$$pixel_to_mm = FOV \cdot h,$$

hvor h er høyden fra kamera til arbeidsområdet.

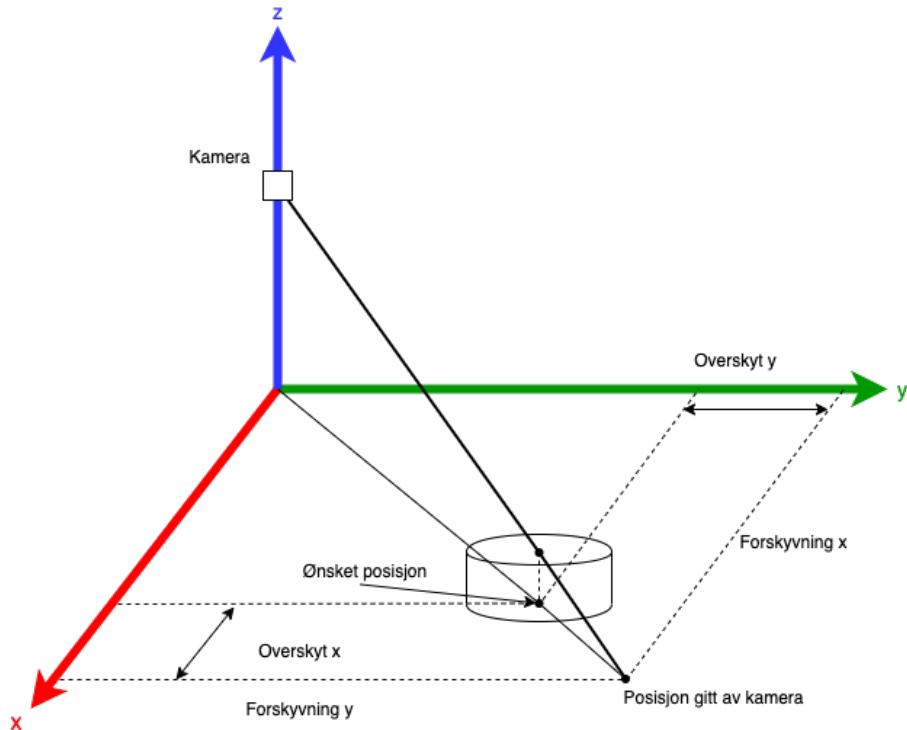
Som nevnt, er det valgt å bruke høyden fra kameraet til arbeidsobjektet som basis for konvertering fra piksler til millimeter. Som illustrert i figur 21, fører dette til overskyt ved bestemmelse av klossenes posisjoner. For å hindre dette, brukes en funksjon, *overshoot_comp*, som kompenserer for overskytet som følge av konverteringen. Funksjonen tar inn høyden til griperen og et *Puck*-objekt.

```

1  def overshoot_comp(gripper_height, puck):
2
3      compensation = [x * puck.height / (gripper_height + 70) for
4          x in puck.position]
5
6      puck.set_position(position=list(map(lambda x, y: x - y,
7          puck.position, compensation)))

```

¹I oppgaven brukes 3.7mm som mål på brennvidden, selv om denne varierer ved ulike fokusnivå. En korrekt kalkulator for kameraets synsvinkel finnes på IDS sine nettsider. [24]



Figur 21: Illustrasjon av overskyt i x- og y-retning

2.4.3 Kompensere for vinklet kamera

Som nevnt i kapittel 2.2.6 må feilbidraget fra kameratasorens vinkel fjernes. Etter justeringsrutinen i samme kapittel (2.2.6) er kjørt, oppdateres *cam_adjustments.ini* med stigningstall som skal brukes for å kompensere for vinkling i kamerasenoren.

Hver gang en posisjon skal bestemmes ved bilde, må disse stigningstallene brukes for å kompensere for vinkel i kamera. Dette gir:

$$kompensering_x = stigning_x \cdot h$$

$$kompensering_y = stigning_y \cdot h,$$

hvor h er kamerasenorenets høyde over objektet.

Disse kompenseringsverdiene skal trekkes fra posisjonskoordinatene. Før dette gjøres, sjekkes griperorienteringen. Dersom griperen er rotert, må kompenseringsverdiene også roteres en tilsvarende mengde. Rotasjonsmengden (θ_z) finnes ved metoden i kapittel 2.4.4 og formel 2 & 3.

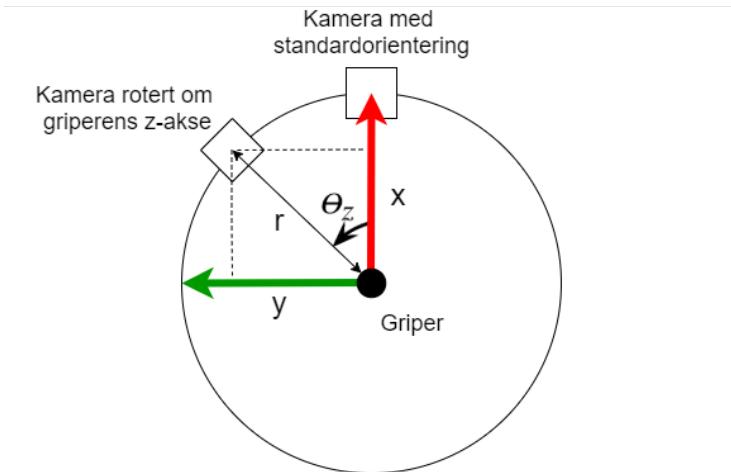
2.4.4 Griper- og kameraposition

Klossenes posisjoner må relateres til griperverktøyet, *tGripper*, i RobotStudio for å kunne knyttes til arbeidsobjektet, siden et kameraverktøy ikke er laget (mer om dette i kapittel 4.6). Som nevnt i kapittel 1.2.3, er kameraet plassert 55mm foran griperen. Denne lengden brukes som en radius i en sirkel hvor griperens *TCP* er sentrum og kameraet beveges langs sirkelbuen, se figur 22.

Standardorientering er definert som 0° , og rotasjon mot klokka som positiv. For å finne retningen på forskyvningen, brukes griperens orientering. Den hentes ut gjennom en GET-forespørsel i funksjonen *get_gripper_position*, inkludert i *rwsuis*-biblioteket. Orienteringen returneres på kvaternionsform. [25] I Python brukes kvaternionen til å bestemme griperens rotasjon i grader om z-aksen:

$$\text{kvaternion} = (w, x, y, z) \quad (2)$$

$$\theta_z = \tan^{-1} \left(\frac{2(w \cdot z + x \cdot y)}{1 - 2(y^2 + z^2)} \right) \quad (3)$$



Figur 22: Illustrasjon ved rotasjon av griper med påmontert kamera

θ_z anvendes deretter for å dekomponere forskyvningen i x- og y-komponenter:

$$x_{\text{forskyving}} = \cos(\theta_z) * r$$

$$y_{\text{forskyving}} = \sin(\theta_z) * r,$$

2.4.5 Sammensatt eksempel

Her vises et sammensatt eksempel av en posisjonskalkyle. Figur 23 illustrerer dette videre.

Utgangspunkt: Posisjonen til griperen er (-50,100,500) og den er rotert 45° med klokka om sin egen z-akse. En griperhøyde på 500mm tilsvarer en kamerahøyde på 570mm. I denne posisjonen tas et bilde av en del av arbeidsområdet som sendes til QR-leseren. Bildet har en oppløsning på 1280x960. Et *Puck*-objekt returneres med posisjon (250,-150).

Posisjonen funnet i bildet må nå gjennomgå de nødvendige transformasjonene og justeringene for å bli et *robtarget*. Først transformeres koordinatene i posisjonen: $x \rightarrow y$, $y \rightarrow x$, $x \rightarrow -x$, $y \rightarrow -y$. Dette gir en ny posisjon lik (150,-250).

Siden griperen var rotert, må også koordinatene roteres. En rotasjon på 45° med klokka tilsvarer en rotasjon på -45°. Etter koordinatene roteres gis en ny posisjon lik (-70.7,-282.8).

For å finne konverteringstallet *piksler_til_millimeter*, må først synsvinkelen i millimeter bli funnet.

$$\text{synsvinkel_bredde}_{mm} = \frac{\text{høyde}}{\text{brennvidde}} \cdot \text{sensorbredde} = \frac{570}{3.7} \cdot 3.6288 = 559$$

Ved å sammenligne denne med bildeoppløsningen, finnes *piksler_til_millimeter*:

$$\text{piksler_til_millimeter} = \frac{\text{synsvinkel_bredde}_{mm}}{\text{opplosning_bredde}} = \frac{559}{1280} = 0.437$$

Ved å konvertere koordinatene fra piksler til millimeter, gis en ny posisjon på (-30.9,-123.6).

Som vist i figur 21 må overskytet kompenseres for. Det gjøres ved å multiplisere koordinatene med en faktor lik:

$$\text{kompenseringsfaktor} = \frac{\text{klosshøyde}}{\text{kamerahøyde}} = \frac{30}{570} = 0.053$$

Multiplikasjon med kompenseringsfaktoren gir et overskyt på (-1.64,-6.55). Dette overskytet fjernes fra koordinatene og gir en ny posisjon på (-29.3,-117.1).

For å kompensere for vinkelen i kamera og kameramontasjen, hentes stigningstall i x og y fra *cam-adjustments.ini*. I dette tilfellet var de følgende:

$$\begin{aligned} \text{stigningstall}_x &= 0.023 \\ \text{stigningstall}_y &= 0.009 \end{aligned}$$

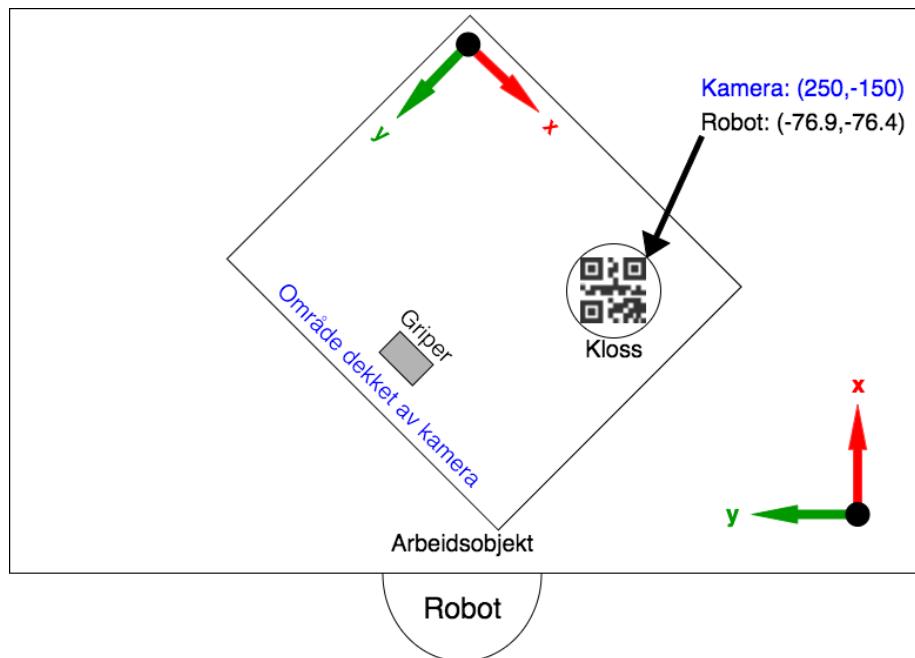
Som vist i kapittel 2.2.6 finnes kompenseringsverdiene ved å multiplisere stigningstallene med høyden til kamera over objektet. Objektet er her klossen (eller QR-koden) som har en høyde på 30mm over arbeidsobjektet. Høyden som brukes er derfor 540mm.

$$kompensering_x = stigning_x \cdot h = 0.023 \cdot 540 = 12.42$$

$$kompensering_y = stigning_y \cdot h = 0.009 \cdot 540 = 4.86$$

For å relatere disse verdiene til klossens posisjon, må de først roteres i like stor grad som griperen (-45°). Etter denne rotasjonen er de faktiske kompenseringsverdiene på $(12.21, -5.34)$. Disse trekkes så fra så fra klossposisjonen og gir ny posisjon på $(-41.5, -111.8)$.

Til sist må kameraposisjonen trekkes fra klossposisjonen for å relatere den til midten av arbeidsobjektet. Kameraposisjonen finnes ved å først finne gripperposisjonen, og deretter legge til forskyvningen mellom griper og kamera. Gripperposisjon er som nevnt $(-50, 100, 500)$ og orienteringen er -45° . Lengden mellom griper og kamera er 55mm og kan ses på som en radius slik som vist i figur 22. I standardorientering (med gripervinkel 0°) er forskyvningen mellom griper og kamera $(55, 0)$. Denne må roteres -45° . Dette gir en forskyvning på $(35.4, -35.4)$. Når denne trekkes fra klossposisjonen gis en endelig posisjon på $(-76.9, -76.4)$.



Figur 23: Illustrert eksempel av posisjonskalkyle

Figur 23 illustrerer eksempelet som er gjennomgått. Griper og kamera er rotert 45° med klokka og gir en klossposisjon på (250,-150) i bildet. Etter posisjonskalkuleringen, gis et *robtarget* med koordinater (-76.9,-76.4).

2.5 Kommunikasjon mellom Python og RobotWare

Robot Web Services muliggjør kommunikasjon mellom Python og RobotWare. Arkitekturen i disse tjenestene tas i bruk gjennom Python-biblioteket *Requests* og et selvlaget bibliotek, *rwsuis* i Python. I dette kapittelet vises det eksempler på bruk av disse ressursene samt en metode for testing av denne arkitekturen.

2.5.1 Robot Web Services

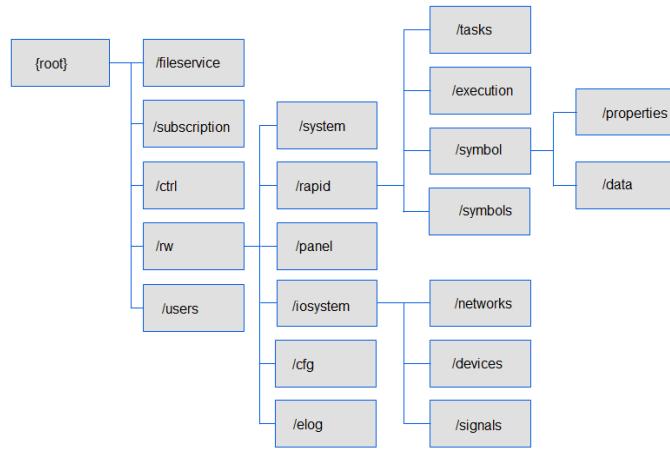
Kommunikasjonen mellom PC og RobotWare er gjort mulig gjennom ABB’s API; Robot Web Services, som baserer seg på REST. [4] Dette åpner for både henting og endring av ressurser i RobotWare. REST deler referanser til ressurser i RAPID i stedet for å dele ressursene direkte. Dermed må brukeren sende forespørsel til riktig uniform ressursidentifikator (URI) for å få tilgang på ressursen. [26] Eksempelvis vil det å velge en RAPID-variabel se slik ut:

`http://localhost/rw/rapid/symbol/data/RAPID/T_ROB1/`

etterfulgt av variabelnavnet. I sammenheng med REST brukes vanligvis HTTP for selve overføringen av data. Etter en forespørsel til RobotWare, gis svaret i enten XML- (“Extensible Markup Language”) eller JSON-format (RWS støtter begge²). Som standard returnerer Robot Web Services svar i XML-format. For å velge JSON, må en sette spørreparameter til “json=1” i forespørselen:

`http://localhost/rw/iosystem?json=1`

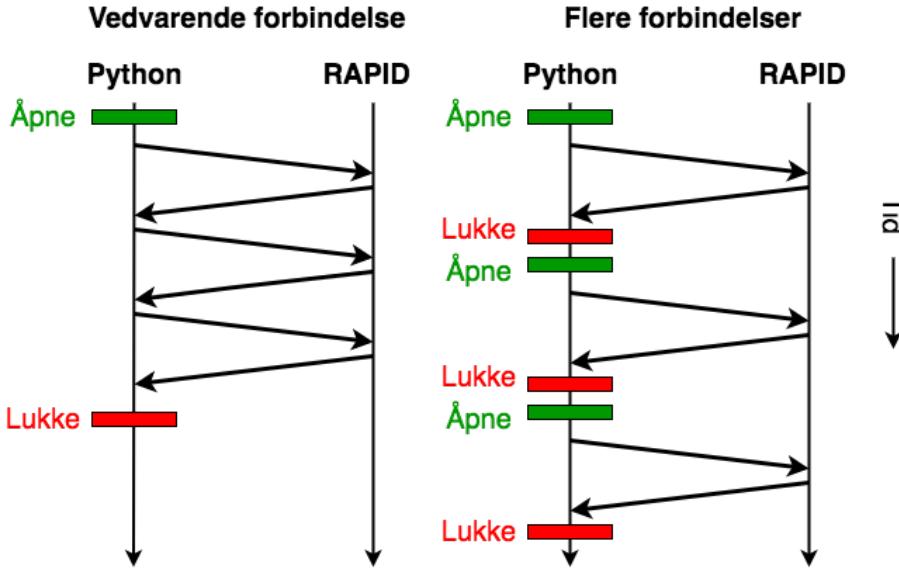
²OPTIONS-forespørsler og abonnement på ressurser støtter bare XML, men brukes ikke i oppgaven.



Figur 24: Ressurs hierarkiet i Robot Web Services.[4]

2.5.2 Requests-biblioteket

For å forenkle kommunikasjonen brukes Python-biblioteket *Requests*. “*Requests* er et elegant og enkelt HTTP-bibliotek, bygget for mennesker.” [13] Denne har ferdige funksjoner for de ulike HTTP-metodene. For å ta nytte av disse på en effektiv måte, brukes en selvlaget klasse, *RWS* (se kapittel 2.5.3), med funksjoner for ulike typer forespørsler mot RobotWare. Når et *RWS*-objekt lages, opprettes også en vedvarende HTTP-forbindelse, i form av et *requests.Session*-objekt. Fordelen med dette illustreres i figur 25.



Figur 25: En vedvarende HTTP-forbindelse har hastighetsfordel over flere forbindelser [27]

For å få tilgang på ressurser i RobotWare, må brukeren være *autorisert*. Dette avgjøres av serveren ved å sjekke brukernavn og passord sendt i “headeren” av hver HTTP-forespørsel. Disse sendes og sjekkes via *digest access authentication*. I motsetning til *basic access authentication*, sendes brukernavn og passord ikke i ren tekst, men i *hashet* form. [28] *Digest access* må brukes siden forespørslene mot RobotWare sendes over HTTP, og ikke dens sikre utvidelse, HTTPS.

Standard brukernavn og passord i RobotWare er “Default User” og “robotics”. *Requests* inneholder en funksjon *HTTPDigestAuth*, som bruker *digest access*:

```
1 >>> from requests.auth import HTTPDigestAuth
2
3 >>> auth = HTTPDigestAuth('Default User', 'robotics')
```

Session-objektet fra *Requests* gir mulighet for å lagre autentiseringsinformasjonen i selve objektet, slik at forespørsler automatisk inneholder denne i headeren:

```
1 >>> session = requests.Session()
2 >>> session.auth = HTTPDigestAuth('Default User', 'robotics')
```

2.5.3 RWS-klassen

RWS er en selvlaget klasse som forenkler kommunikasjon mellom Python og RobotWare. Denne er inkludert i Python-biblioteket *rwsuis*, som vi har gjort offentlig tilgjengelig for de som skal utføre laboratorieoppgaven. [29] Foruten Python's

standardbibliotek, avhenger den bare av *Requests*-biblioteket. Den inneholder funksjoner med alle de ulike forespørslene som brukes i demonstrasjonsprogrammet. Det er dermed mulig å hente og endre status, posisjon, variabler og andre ressurser på kontrolleren. [30] For å koble seg til *Norbert*, må en instans av denne klassen lages med IP-addresen til *Norbert* (152.94.0.38), samt brukernavn og passord. Det er også mulighet for å lage en instans av denne klassen som kan kommunisere med *Rudolf*, ved å endre IP-addresen.

```
1 >>> from rwsuis import RWS
2 >>> norbert = RWS.RWS('152.94.0.38', 'Default User', 'robotics')
```

Ved GET-forespørslene, parses responsen automatisk, og all vesentlig informasjon returneres i funksjonen. JSON brukes eksklusivt som datautvekslingsformat i *RWS*-klassen. Parsing av slike responser omtales videre i kapittel 2.5.5.

Enkelte forespørslene aksepteres bare dersom robotkontrolleren er i en spesifikk tilstand. Dersom slike forespørslene (sendt med funksjoner fra *RWS*-klassen) feiler, gis mulige grunner for dette i Python-konsollen. Funksjonen *start_RAPID* spør om å starte RAPID-programmet som er lastet inn på FlexPendanten. Dersom dette feiler, gis responsen som vist:

```
1 >>> norbert.start_RAPID()
2 Could not start RAPID. Possible causes:
3 * Operating mode might not be AUTO. Current opmode: AUTO.
4 * Motors might be turned off. Current ctrlstate: motoroff.
5 * RAPID might have write access.
```

Ved hjelp av responsen, er det tydelig at robotens motorer må slås på før *start_RAPID* prøves igjen.

2.5.4 Brukte HTTP-metoder

Blant de eksisterende HTTP-metodene [31], er GET og POST mest relevante for denne oppgaven.

GET er en *trygg* metode som kun kan hente en representasjon av den spesifiserte ressursen. Metoden er trygg siden den ikke kan endre noe på serveren. Det er dermed mulig å bruke GET-metoden uten *mastership*³. Informasjonen ved en veldig forespørsel representeres enten i XML eller JSON. Dette bestemmes av brukeren ved å inkludere "?json=1" som spørreparameter ("query string") eller la det stå tomt (XML er standard). Informasjonen som returneres må aksesseres på riktig måte. Dette kan gjøres gjennom parsing av XML eller JSON i Python.

³skrivetillatelse mot robotkontroller

POST ber om tillatelse til å endre på en ressurs på serveren. Dette er dermed ikke en *trygg* metode og krever derfor *mastership* fra kontrolleren for å utføres. For å få *mastership* over kontrolleren, må en POST-forespørsel også sendes. Dersom kontrolleren står i manuell modus, må forespørselet manuelt aksepteres på FlexPendanten. I automatisk modus, gis *mastership* automatisk ved forespørsel. Forespørsler for *mastership* i automatisk og manuell modus finnes begge i *rwsuis*-biblioteket:

Automatisk modus:

```
1 >>> norbert.request_mastership()
```

Manuell modus:

```
1 >>> norbert.request_rmmp()
```

2.5.5 Lesing av JSON

Ved vellykket GET-forespørsel til RobotWare, returneres en JSON-formattert streng som parses i Python med `json.loads`. [32] Etter parsing, fremstår denne strengen som en kombinasjon av lister og ordlister i Python. Informasjon kan dermed hentes ut av denne strukturen som vanlig i Python. Ved forespørsel av “execution state”, som enten er “running” eller “stopped”, ser responsen slik ut:

```
1 >>> session = requests.Session()
2 >>> session.auth = HTTPDigestAuth('Default User', 'robotics')
3 >>> response = session.get("http://152.94.0.38/rw/rapid/execution?")
4 >>> response.text
5 '{'
6     "_links": {
7         "base": {
8             "href": "http://152.94.0.38:80/rw/rapid/"
9         }
10    },
11    "_embedded": {
12        "_state": [
13            {
14                "_type": "rap-execution",
15                "_title": "execution",
16                "ctrlexecstate": "stopped",
17                "cycle": "once"
18            }
19        ]
20    }
21 }'
```

For å hente ut hvilken status roboten er i, brukes JSON slik:

```

1 >>> json_string = response.text
2 >>> _dict = json.loads(json_string)
3 >>> execution_state = _dict["_embedded"]["_state"][0]["
4   ctrlexecstate"]
5 >>> execution_state
5 'stopped'

```

Det er også mulig å lage en JSON-streng i Python ved å bruke `json.dumps` på et Python-objekt. [32] Siden *Requests* brukes i denne oppgaven, kreves det ikke å lage JSON-strenger.

2.5.6 Kommunikasjon under programutførelse

Ved kjøring av program som inkluderer både Python- og RAPID-filer, er kommunikasjonen mellom disse asynkron. Utførelsesrekkefølgen har dermed stor betydning for oppførselen til programmet. Demonstrasjonsprogrammet er et eksempel hvor dette gjelder.

Generelt ønskes det at en kodeblokk kjøres i en fil, før en kodeblokk kjøres i den andre filen, og så videre. Dette oppnås med bruk av flaggvariabler og ventemetoder i både Python og RAPID. Flaggene er boolske variabler. Når en ønsket kodeblokk er gjennomgått, endres den relevante flaggvariabelen til å være sann (*TRUE*).

Ventemetodene vises under:

```

1 # Wait method used in Python
2 def wait_for_rapid(self, var='ready_flag'):
3     """Waits for robot to complete RAPID instructions
4     until boolean variable in RAPID is set to 'TRUE'.
5     """
6     while self.get_rapid_variable(var) == "FALSE" and self.
7         is_running():
8             time.sleep(0.1)
9             self.set_rapid_variable(var, "FALSE")
10
11 ! Wait method used in RAPID
12 PROC wait_for_python()
13     ! Wait for Python to finish processing image
14     WHILE NOT image_processed DO
15     ENDWHILE
16     image_processed:=FALSE;
17 ENDPROC

```

Ved mangel på eller feil bruk av flaggvariabler, vil programutførelsen ikke bli som forventet. I demonstrasjonsprogrammet innebærer dette for eksempel at bilder tas i feil posisjon, eller at programmet havner i en evig venteløkke.

I demonstrasjonsprogrammet velges hvilken del av RAPID-programmet som skal kjøres gjennom sending av en variabel med navn *WPW* (“What Python Wants”). Før et program er valgt, kjøres en evig løkke i RAPID som kontinuerlig sjekker etter endringer i *WPW*.

2.5.7 Testing av kommunikasjon

Gjennom bruk av Python-biblioteket *Locust*, testes responstider ved vanlige HTTP-forespørsler i demonstrasjonsprogrammet. Det er mulig å simulere flere ulike brukere av en API, for å kjøre belastningstester av denne. Kommunikasjon mellom Python og RobotWare skjer med én bruker om gangen, og det er derfor ikke nødvendig å skape mer enn én virtuell bruker gjennom *Locust* i testen.

Hensikten med testen er å tallfeste hastighetene som oppnås ved bruk av Robot Web Services, og å vurdere API-ens robusthet.

Det ble avholdt to separate tester av ulike grupper HTTP-forespørsler. Testene ble skapt ved hjelp av *Locust*-dokumentasjonen. [33] I første test, sendes forespørsler som er hyppig forekommende under kjøring av demonstrasjonsprogrammet. Disse vises i tabell 2.

Type	Navn
GET	Gripper position
GET	Execution state
POST	Robtarget
GET	RAPID variable
POST	RAPID variable

Tabell 2: HTTP-forespørsler ved første kommunikasjonstest

Det velges en tilfeldig ventetid mellom 10-15ms mellom hver HTTP-forespørsel. Alle testede forespørsler blir tilfeldig valgt, men har lik prioritet, slik at de kjøres ca. like mange ganger. Det ble totalt sendt 100 000 forespørsler under testen.

I neste test, sendes forespørsler som forekommer ved start og stopp av demonstrasjonsprogrammet, nemlig av- og påslåing av robotens motorer. Disse sendes vanligvis ikke midt i programutførelsen, men er likevel interessante å analyse- re. For å gi et realistisk bilde på responstidene, må disse forespørslene kjøres i sekvens. Dersom motorene er påslått når det sendes forespørsel om å slå på motorene, senkes responstiden betraktelig. Forespørslene vises i tabell 3 under.

Type	Navn
POST	Motors on
POST	Motor off

Tabell 3: HTTP-forespørsler ved andre kommunikasjonstest

Det blir brukt en konstant ventetid på 2 sekunder mellom hver forespørsel. Det ble totalt sendt 50 forespørsler under testen.

Resultatene av kommunikasjonstesten presenteres i kapittel 2.5.7.

2.6 Kodedokumentasjon

ABB Robot with Machine Vision

This software is mainly directed toward engineering students and staff at the University of Stavanger. It was first created as a Bachelor thesis work.

User Guide

- [License](#)
- [Installation](#)
 - [\\$ pip install](#)
 - [Get the source code](#)
- [Robot Web Services](#)
 - [RWS Class](#)
- [Image Tools](#)
 - [Camera Class](#)
 - [Capturing Images and Video](#)
 - [Scanning QR Codes](#)
- [OpenCV_to_RAPID.py](#)
- [Puck Class](#)
- [Need Help?](#)

Indices

- [Index](#)
- [Search Page](#)

Figur 26: Forside på Read the Docs

Det er brukt en betydelig mengde tid på å dokumentere kode både i kildekoden og på *Read the Docs*. [34] Dette er gjort med et håp om at kildekoden kan brukes og vedlikeholdes i fremtiden ved UiS, både som laboratorieoppgave og som demonstrasjonsprogram.

Dokumentasjonen på Read the Docs er laget ved bruk av Sphinx og reStructuredText. [15] [16] Under vises et brukseksempel av disse.

```
1 Scanning QR Codes
2 ~~~~~
3
4 .. py:function:: QR_Scanner(img)
5
6     Filters and normalizes the input image. The processed image
7     is decoded using pyzbar_.
8     For every QR code detected, a :py:class:`'Puck'` object is
9     created.
10
11    :param ndarray img: An image
12
13    :return: A list of :py:class:`'Puck'` objects
14
15    .. _pyzbar: https://pypi.org/project/pyzbar/
```

Scanning QR Codes

`QR_Scanner(img)`

Filters and normalizes the input image. The processed image is decoded using `pyzbar_`. For every QR code detected, a `Puck` object is created.

Parameters: `img (ndarray)` – An image

Returns: A list of `Puck` objects

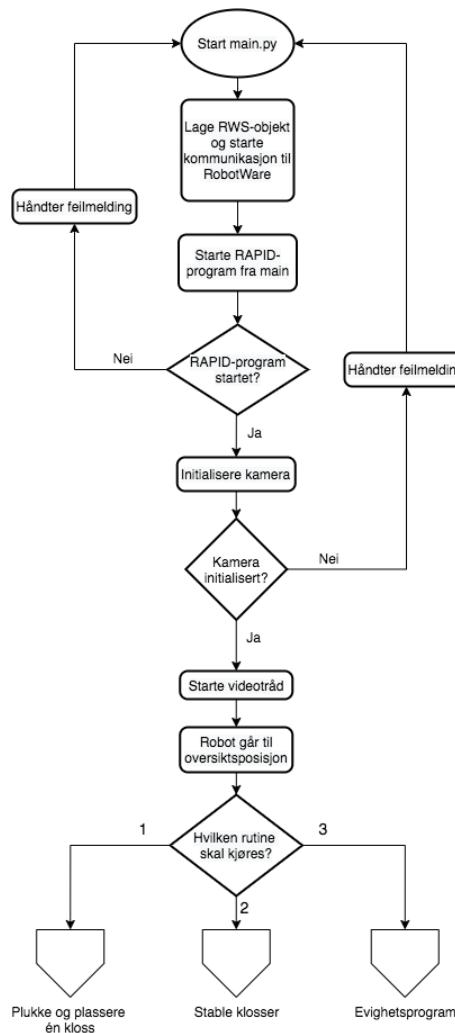
Figur 27: Resultat av dokumentasjon i Sphinx og RST

3 Resultat

I dette kapittelet presenteres demonstrasjonsprogrammet, samt andre resultater som alle har utgangspunkt i dette. Dette inkluderer en laboratorieoppgave, resultat av kommunikasjonstester og en repeterbarhetstest av systemet, samt en video som viser programmet i aksjon.

3.1 Demonstrasjonsprogram

Det er laget et demonstrasjonsprogram som består av tre ulike rutiner i Python og RAPID. Programmet knytter sammen alt i konstruksjonsdelen til et samlet resultat.



Figur 28: Flytskjema over oppstart av demonstrasjonsprogram

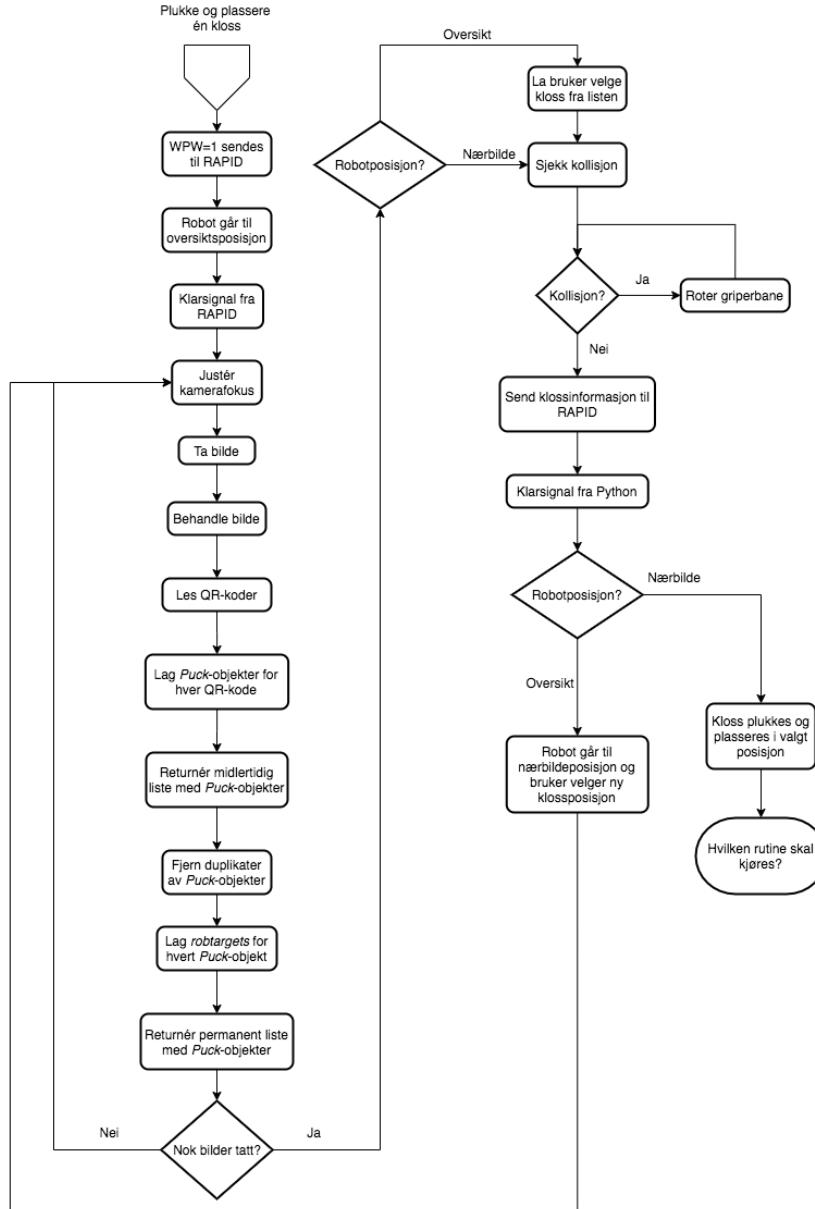
Felles for disse rutinene er oppstarten av demonstrasjonsprogammet. Før programmet kan startes, må roboten stå i automatisk modus, og motorene slås på. Etter at automatisk modus er aktivert, kan motorene slås på via en POST-forespørsel. Et enkelt program som gjør dette er lagret som en satsvis fil på skrivebordet til PC-en på laboratoriet, med navn *Start_Norbert.bat*. Når motorene er slått på, kan demonstrasjonsprogrammet startes. Her åpnes og initialiseres kamera, slik som vist i 2.2.1. I tillegg startes en separat tråd i Python som viser kontinuerlig video fra kameraets perspektiv. Deretter åpnes kommunikasjon mot Norbert, *mastership* etterspørres, og programmet i RAPID startes. Python venter så til roboten er i startposisjon, med kamera 570mm midt over arbeidsobjektet.

Etter initialiseringen, kan ønsket rutine velges i Python. Valgene representeres i Python-konsollet:

1. Plukke og plassere én valgt kloss
2. Stable klosser
3. Evighetsprogram
4. Finne riktig eksponeringstid (ved metoden i kap. 2.2.5)
5. Kameraajustering (ved metoden i kap. 2.2.6)
0. Avslutte program

3.1.1 Plukke og plassere én valgt kloss

I denne rutinen kan det være et vilkårlig antall klosser i robotens arbeidsområde. Brukeren velger selv hvilken kloss som skal plukkes og hvor den skal flyttes.



Figur 29: Flytskjema over plukking og plassering av én kloss

For å starte plukking og plassering av én kloss i RAPID, oppdateres *WPW* (What Python Wants) til riktig verdi (1). I startposisjon tas det fem bilder av arbeidsområdet. Hvert bilde sendes til QR-leseren som lagrer alle QR-koder som *PuckPuckrobtargets* som kan sendes til RAPID.

Brukeren får nå presentert alle klosser som ble funnet, og må velge hvilken

av disse som skal plukkes. Dersom brukeren velger et ugyldig alternativ, gis en relevant feilmelding før valgene presenteres på nytt. Når et gyldig alternativ velges, finner Python en klar bane for griperen ved metoden i kapittel 2.3.3. Deretter sendes både posisjon og orientering til griperen, og Python gir klarsignal til RAPID. Griperen beveges nå til nærbildeposisjon. Nærbildeposisjonen preges av resultatet i kollisjonshindringen. Dersom griperbanen må roteres, vil denne orienteringen brukes også når nærbilde tas.

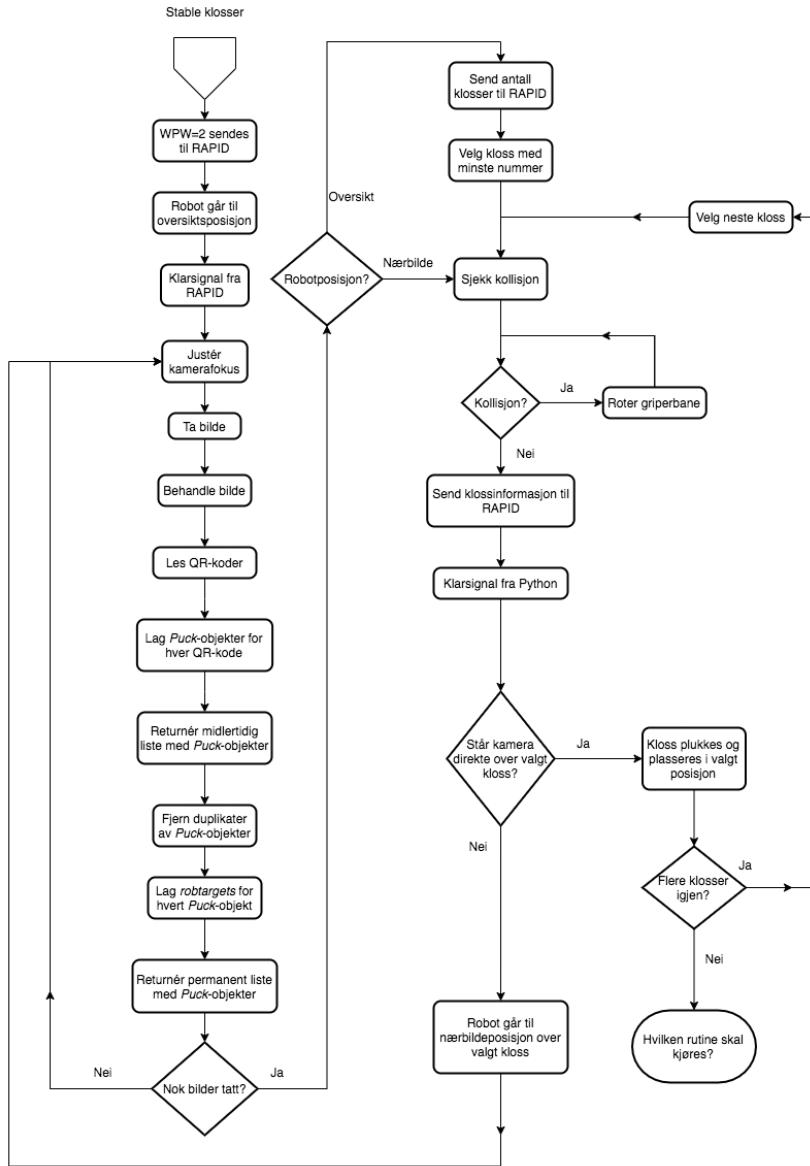
Før nytt bilde tas, får brukeren velge hvor den bestemte klossen skal flyttes. Etter valgt posisjon, tas nærbilder helt til klossen igjen er funnet (dette skjer vanligvis etter ett forsøk). Den nye klossposisjonen skal være mer nøyaktig enn den forrige. Denne brukes på ny for å sjekke kollisjon med andre klosser, for å være helt sikker på at banen er kollisjonsfri.

Den endelige posisjonen sendes nå til RAPID. I tillegg sendes det informasjon om hvorvidt griperen skal gå framover eller bakover for å plukke klossen. Et klarsignal sendes så til RAPID slik at klossen kan plukkes og plasseres i ønsket posisjon.

Samtidig som klossen flyttes, roteres den slik at den resulterende orienteringen er stående, slik som i figur 16.

3.1.2 Stable klosser

I denne rutinen kan det være et vilkårlig antall klosser i robotens arbeidsområde. Disse plukkes i stigende rekkefølge (basert på klossnummer) og plasseres i en posisjon definert i RAPID.



Figur 30: Flytskjema over stabling av klosser

For å starte stabling av klosser i RAPID, oppdateres *WPW* (What Python Wants) i til riktig verdi (2). I startposisjon tas det fem bilder av arbeidsområdet. Hvert bilde sendes til QR-leseren som lagrer alle QR-koder som *Puck*-objekter. Disse returneres i en liste med *Puck*-objekter. Deretter transformeres alle klossposisjonene til *robtargets* som kan sendes til RAPID.

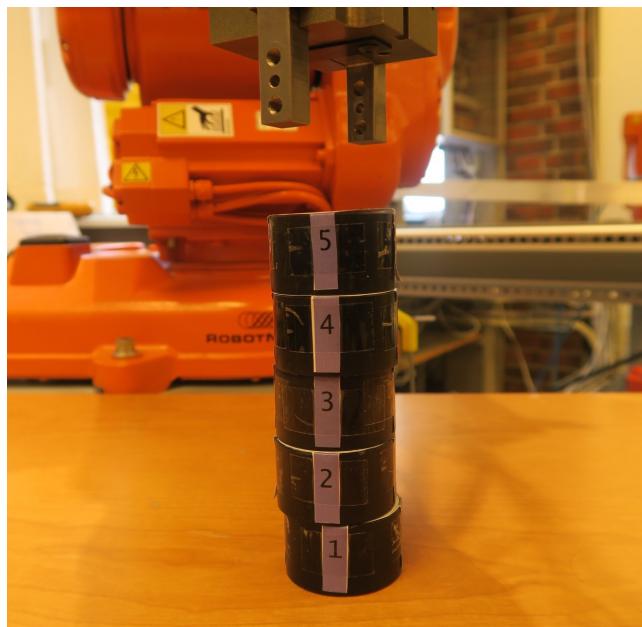
Antall klosser funnet blir sendt til RAPID for å bestemme lengden på for-løkken i programmet. Python kjører også en for-løkke med samme lengde.

I for-løkken velges alltid klossen med lavest nummer. Python finner så en klar bane for griperen ved metoden i kapittel 2.3.3. Deretter sendes både posisjon og orientering til griperen, og Python gir klarsignal til RAPID. Griperen beveges nå til nærbildeposisjon. Nærbildeposisjonen preges av resultatet i kollisjonshindringen. Dersom griperbanen må roteres, vil denne orienteringen brukes også når nærbilde tas.

Nærbilder tas så til klossen igjen er funnet. Den nye klossposisjonen skal være mer nøyaktig enn den forrige. Denne brukes på ny for å sjekke kollisjon med andre klosser, for å være helt sikker på at banen er kollisjonsfri.

Den endelige posisjonen sendes så til RAPID. I tillegg sendes det informasjon om hvorvidt griperen skal gå framover eller bakover for å plukke klossen. Et klarsignal sendes så til RAPID slik at klossen kan plukkes og plasseres i ønsket posisjon. For hver kloss, økes høyden i den ønskede posisjonen, mens x- og y-koordinater forblir like. På denne måten stables klossene i høyden.

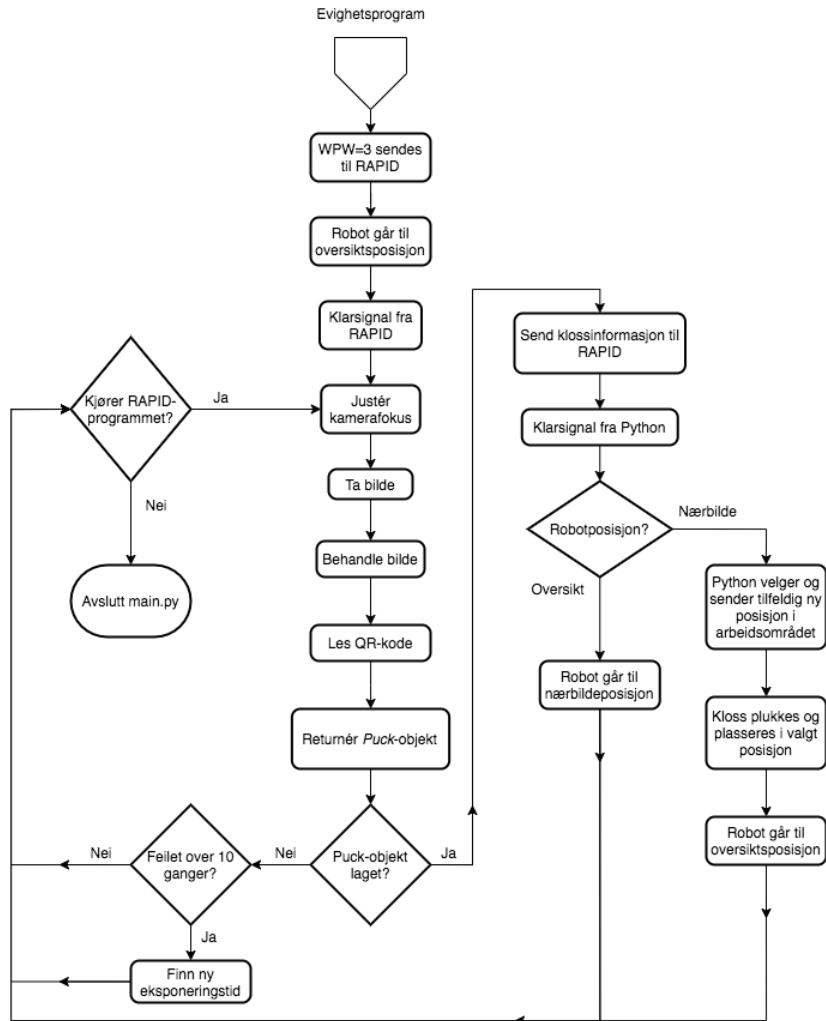
Samtidig som klossene flyttes, roteres de slik at den resulterende orienteringen er 0° , slik som i figur 16. Ved stabling av klosser gir dette et visuelt resultat som bekrefter at klossene roteres riktig, som vist i figur 31.



Figur 31: Typisk resultat fra stabling av klosser

3.1.3 Evighetsprogram

I denne rutinen kjører roboten helt til den slås av. Det kan bare være én kloss i robotens arbeidsområde. Den samme klossen plukkes og plasseres i en tilfeldig posisjon innenfor et definert område på ubestemt tid.



Figur 32: Flytskjema over evighetsprogram

For å starte evighetsprogrammet i RAPID, oppdateres *WPW* (What Python Wants) til riktig verdi (3). I startposisjon tas det bilder som sendes til QR-leseren helt til klossen oppdages. Dette skjer vanligvis ved første bilde. Et *Puck*-objekt lages deretter fra informasjonen i QR-koden. Deretter transformeres klossposisjonen til et *robtarget* som kan sendes til RAPID.

Etter posisjonen sendes til RAPID, beveger roboten seg til nærbildeposisjon over klossen. Det regnes ikke ut en kollisjonsfri bane ettersom denne allerede skal være uhindret.

I nærbildeposisjon lager Python en tilfeldig valgt ny klossposisjon som klossen skal flyttes til. Dette skjer gjennom *random*-biblioteket i Python. Denne nye posisjonen sendes til RAPID før et nytt bilde tas.

Nærbilder tas nå helt til klossen igjen oppdages. Når posisjonen er funnet, sendes denne til RAPID sammen med ny klossposisjon. I evighetsprogrammet går alltid griperen framover for å plukke klossen. Klossen plasseres deretter i den nye tilfeldig valgte posisjonen.

Roboten returnerer så til oversiktsposisjon og starter en ny runde.

Et resultat av en lang kjøring (16 timer) av evighetsprogrammet vises i kapittel 3.4.

3.2 Laboratorieoppgave

Ved fullførelse av systemet, ble en laboratorieoppgave til faget ELE610 på UiS utformet. Utformingen av oppgaveoppsettet og kapittelinndelingen er inspirert av Karl Skretttings tidligere laboratorieoppgaver i faget [35], men selve oppgaveteksten er selvlaget. Oppgaven består av flere deloppgaver som skal løses før et hovedprogram kan påbegynnes. Hovedprogrammet skal være en enkel versjon av systemet utviklet i denne bacheloroppgaven.

Deloppgavene består av følgende:

- Etablere kontakt mellom Python og RobotWare. Det gis her en introduksjon til HTTP-forespørslar, *Requests*-biblioteket og *rwsuis*-biblioteket.
- Bildefangst og lesing av QR-koder. Det presenteres ulike valg av programvare som kan brukes for å løse dette, samt praktiske eksempler.
- Posisjonskalkulering. Det vises her en meget grundig gjennomgang av de ulike stegene som bidrar til å lage et *robtarget* som kan brukes i RAPID.

For å konstatere at alt av maskin- og programvare er riktig konfigurert, kan et eksempelprogram inkludert i laboratorieoppgaven benyttes. Eksempelprogrammet tar bilde av arbeidsområdet, kalkulerer et *robtarget* og beveger seg over en kloss uten å plukke den. Fullføres dette med suksess, kan en konkludere med at

systemet er riktig konfigurert.

Det er gitt flere filer, både i Python og RAPID som skal gi et utgangspunkt for løsningen. Det gis også tilgang til *rwsuis*-biblioteket slik at kommunikasjonsdelens kompleksitet ikke er urimelig høy.

Laboratorieoppgaven er inkludert i rapporten som vedlegg A.1.

3.3 Responstider ved HTTP-forespørsler

Resultatene fra kommunikasjonstesten i kapittel 2.5.7, fremstår i tabellene under.

Type	Navn	Forespørsler	Feil	Gj.snitt (ms)	Maks (ms)	Gj.snitts innholds-størrelse	Forespørsler/s
GET	GripperPos	20042	0	3	14	918	12.20
GET	Execution	20168	0	2	13	191	12.28
POST	RobTarg	19947	0	4	11	0	12.14
GET	Variable	20042	0	3	11	131	12.20
POST	Variable	19951	0	3	16	0	12.15
	Totalt	100150	0	3	16	248	60.97

Tabell 4: Responstider ved vanlige HTTP-forespørsler

Tabell 4 viser resultatet etter testing av de mest brukte forespørslene i demonstrasjonsprogrammet. Det ble sendt 100 000 forespørsler i løpet av 27 minutter. Dette resulterer i ca. 61 forespørsler per sekund. Gjennomsnittlig responstid ligger på rundt 3ms. Maksimum responstid er målt til å være 16ms og kom ved oppdatering av en variabel.

Konfidensintervall							
Type	Navn	90%	95%	98%	99%	99.9%	99.99%
GET	GripperPos	4	4	5	5	7	9
GET	Execution	4	4	4	4	6	10
POST	RobTarg	5	5	5	5	7	12
GET	Variable	4	4	4	5	6	11
POST	Variable	4	4	5	5	6	15
	Total	4	5	5	5	7	11

Tabell 5: Konfidensintervall ved vanlige HTTP-forespørsler

Et mer detaljert resultat av testen gis i ulike konfidensintervall for hver forespørsel

og vises i tabell 5. Resultatet viser at responstider på over 10ms fremstår svært sjeldent (ca. 1 av 10 000 forespørslar).

Type	Navn	Forespørslar	Feil	Gj.snitt (ms)	Maks (ms)	Forespørslar/s
POST	MotorsOn	25	0	745	783	0.21
POST	MotorsOff	25	0	12	14	0.21
	Total	50	0	379	783	0.42

Tabell 6: Responstider ved start/stopp av motorer

Responstidene ved start og stopp av robotens motorer vises i tabell 6. Det ble sendt 50 forespørslar i løpet av 2 minutter. Responstiden ved start av motorer er vesentlig høyere enn ved stopp av motorer. Dette skyldes påslåing av selve elektronikken i roboten.

Konfidensintervall					
Type	Navn	75%	80%	90%	95%
POST	MotorsOn	760	780	780	780
POST	MotorsOff	13	13	13	13
	Total	740	760	780	780

Tabell 7: Konfidensintervall ved start/stopp av motorer

Ved hjelp av konfidensintervallet i tabell 7, ser vi at responstidene varierer i liten grad. Det er mulig at flere forespørslar ville avslørt et høyere maksimum i responstid. Dette ble ikke sjekket for å begrense eventuell last på robotens elektronikk.

3.4 Repeterbarhetstest av systemet

Systemets robusthet ble testet gjennom kontinuerlig kjøring av evighetsprogrammet i kapittel 3.1.3.

Parametre og forhold for test:

- Driftsmodus: Automatisk
- Maks hastighet: v1000
- Akselerasjon: AccSet 100, 50, \FinePointRamp:=30;

- Lysforhold: Kontrollerte. Minimalt med sollys. Konstant belysning fra taket.
- Kamera: IDS UI-1007XS-C
- Bildeoppløsning: 1280x960
- Arbeidsobjekt: Trebord, se figur 2a
- Arbeidsområde: $x : (-50 \rightarrow 150)$, $y : (-150 \rightarrow 150)$. Areal: 600cm^2
- Tidsrom: 17:10 - 09:15 (16 timer, 5 minutt)

I tidsrommet testen ble avholdt, gjennomførte roboten 14 000 runder med plukking og plassering. Det tilsvarer en gjennomsnittlig rundetid på:

$$\text{Rundetid} = \frac{\text{Tid}[s]}{\text{Runder}} = \frac{16 \cdot 3600 + 5 \cdot 60}{14000} = 4.14s$$

Rundetiden kan om ønskelig reduseres ved økning i hastighet og akselrasjon. Parametrene som er brukt under repeterbarhetstesten er valgt for å minimere både slitasje på robotens ledd og støy/vibrasjoner som følge av raske rykk i robotens bevegelser.

3.5 Video

For å illustrere hvordan repeterbarhetstesten og stabling av klosser blir gjennomført ble det laget en video som viser roboten under kjøring. I tillegg kan man se alt fra robotens perspektiv, i sanntid.

Videoen har også en nytteverdi for studenter som skal utføre laboratorieoppgaven. Her vises blant annet gripeteknikk og teknikk ved bildefangst.

Lenke til videoen: <https://vimeo.com/420278023>

4 Diskusjon

I denne delen diskuteres viktige valg tatt med hensyn på oppgaveløsning og mulige fremtidige utvidelser til systemet.

4.1 Valg av kamera

Oppgaven skulle i utgangspunktet løses ved bruk av dybdekameraet *Intel RealSense Depth Camera D435*. [36] Med dybdeinformasjon, er muligheten for programutvidelser større. Det kan for eksempel enkelt plukkes klosser i høyden.

Intel har programmert et bibliotek for Python, *pyrealsense2*, som kan kontrollere dybdekameraet. [37] Dette biblioteket ble forsøkt brukt uten suksess.

Ingen gode løsninger ble funnet i sammenheng med dybdekameraet. Med bakgrunn i oppgavens tidsramme, ble det derfor bestemt å forkaste dette kameraet. I stedet ble uEye XS brukt.

4.1.1 uEye XS

Det var mest naturlig å velge uEye XS som et alternativ til dybdekameraet. XS-kameraet har vi tidligere erfaring med fra faget ELE610. Laboratorieoppgaven vil dermed også ha en naturlig sammenheng med eksisterende laboratorieoppgaver i ELE610, nemlig “Image Acquisition”-delen, som også bruker IDS sine kameras.

4.2 Valg av bildeoppløsning

I demonstrasjonsprogrammet ble det valgt å bruke en bildeoppløsning på 1280x960. Som vist i tabell 1, har uEye XS en rekke ulike muligheter for andre oppløsninger. Enkelte av disse ble forsøkt brukt, men ga ikke tilfredsstillende resultater. Høyere oppløsninger ble opplevd som krevende for Python å jobbe med. Dette førte til lengre programutførelse og videovisning med lavt antall bps. Lavere oppløsninger førte til en nedsatt suksessrate i QR-leseren.

4.3 Rotert kameralinse

Kameralinsen kan være rotert opptil $\pm 2^\circ$. Dette er et feilbidrag som ikke er tatt høyde for i oppgavens løsning. En tilstrekkelig nøyaktighet i systemet er allerede oppnådd, og støttes av resultatet i kapittel 3.4. I tillegg vil en eventuell rotasjon av kameralinsen utgjøre en minimal feil i posisjonskalkulering ettersom kameraet i nærbildeposisjon står tilnærmet direkte over klossens sentrum. Det anses derfor ikke som hensiktsmessig å bruke tid på å finne en metode som kompenserer for dette feilbidraget.

Som et tillegg, må det påpekes at andre kamera av typen uEye XS kan være påvirket i større grad av feilbidrag enn det vårt kamera var i oppgaven. Det er derfor ikke mulig for oss å garantere at disse vil ha en tilstrekkelig nøyaktighet.

4.4 Forvrengning i kamera

Alle optiske systemer introduserer forvrengning av ulik grad. Forvrengingen kommer av at det er enklere og billigere å produsere en sfærisk linse enn en linse som er matematisk. I tillegg er det kostbart å posisjonere sensor nøyaktig. Radiell og tangentIELL forvrengning gjør størst utslag på grunn av dette. [38]

4.4.1 Kalibrering

For å fjerne forvrengning i et optisk system, kan kameraet kalibreres. Dette innebærer å estimere parametrerne til linsen og bildesensoren i kameraet. Disse parametrerne kan brukes til å korrigere for linseforvrengning, måle størrelsen på et objekt i verdensenheter eller bestemme plasseringen av kameraet i scenen. [39] I enkelte tilfeller er kamerakalibreringen sentral, uten denne kan vi i værste fall risikere at all informasjon vi får ut fra kamera er feil og unøyaktig.

I vårt system, er kamerakalibrering ikke kritisk. Forvrengingen i kameraet er liten, og gir ikke merkbart utslag på nøyaktigheten ved lesing av QR-koder. Det ble derfor valgt å ekskludere kamerakalibrering fra oppgaveløsningen.

4.5 Programvare for bildefangst

I første løsning av oppgaven, ble OpenCV anvendt til bildefangst og endring av kameraparametre. Denne løsningen var kort og enkel å bruke. Ved bruk av OpenCV's funksjoner, kan bilde tas med USB-kamera på bare to linjer med kode:

```
1 >>> cap = cv2.VideoCapture(1)
2 >>> ret, frame = cap.read()
```

Problemet med denne løsningen var hovedsakelig mangel på funksjonalitet. OpenCV har mange implementerte funksjoner for endring av kameraparametre, men for uEye XS er disse svært begrenset. Det er for eksempel ikke mulig å kontrollere fokus i uEye XS ved OpenCV. Uten slik funksjonalitet, må kameraets standardparametre brukes. Dette inkluderer løpende autofokus og automatisk lyskontroll.

4.5.1 Problemer med autofocus

Bruk av løpende autofocus går bekostning av robusthet. I visse tilfeller fokuserer ikke kameraet tilstrekkelig, og i andre tilfeller ikke i det hele tatt. Dette fører til stopp i programutførelse, dersom ingen QR-koder oppfattes av leseren. Som nevnt er det ikke mulig å kontrollere fokus i OpenCV.

For å løse problemet med autofokus, ble det først foretatt skarphetstester av alle bilder. Ved utilfredsstillende skarphetsverdier, ble bildet forkastet. Én idé var å arbeide rundt dette problemet ved å estimere skarpheten i hvert bilde, slik at uskarpe bilder kan forkastes. Resultatet var fortsatt ikke tilfredsstillende.

Som forklart i kapittel 2.2.4 om innstilling av fokus falt løsningen på en *spesifikk* manuell fokus karakteristikk for kameraet.

4.5.2 Problemer med automatisk eksponeringstid

Den automatiske eksponeringen i kameraet bruker et interesseområde (AOI (Area of Interest) på IDS sine sider) som dekker en stor del av bildet. Derfor, ved oversiktsbilde, vil dette interesseområdet i hovedsak bestå av arbeidsområdet. Ved bruk av standard trebord som arbeidsområde, er automatisk lyskontroll tilfredsstillende. Ved andre bakgrunner (slik som i figur 11b), strekker denne ikke til. Med mulighet for å manuelt endre eksponeringstid unngås slike problem, og løsningen vil være både mer robust og dynamisk.

4.6 Kameraverktøy i RobotStudio

Underveis i oppgaveløsningen ble det tydelig at det burde blitt laget et “kameraverktøy”. I den endelige løsningen brukes griperen eksklusivt som verktøy (*tGripper*) i RobotStudio. Dette har skapt unødvendige kompleksiteter i forbindelse med utregning av forskyvninger mellom griper og kamera (for eksempel som vist i kapittel 2.4.4).

Det vil være svært hensiktsmessig å lage et slikt kameraverktøy med tanke på laboratorieoppgaven i fremtiden.

4.7 Valg av QR-leser

Det er flere ressurser tilgjengelig for å oppdage og dekode QR-koder i Python. Våre fremste kandidater var OpenCV sin egen QR-kodeleser, *ZXing* og *ZBar*. Med et ønske om å gjøre det meste av bildebehandling i OpenCV, ble denne leseren først brukt. Den ble opplevd som vanskelig å bruke samt dårlig i ytelse. Den ble derfor forkastet. Valget falt på *ZBar*, som viste seg å være både raskere og mer robust enn både OpenCV og *ZXing*. *ZBar* er et eksternt bibliotek, som betyr at det ikke kan brukes direkte i Python. Vi har brukt *pyzbar* som baserer seg på *ZBar*. *ZBar* gjør det mulig å hente ut all nødvendig data (f.eks. posisjon) fra QR-kodene på en elegant måte.

4.8 Plukking av klosser i høyden

Bildefangst foregår med kamera over arbeidsobjektet, slik at det får et fugleperspektiv. Dette gjør det utfordrende å få informasjon om hvor høy en stabel med klosser er. Først og fremst var tanken å bruke et dybdekamera, men som forklart i kapittel 4.1 ble ikke dette realisert. Det ble derfor sett på andre metoder for å plukke klosser i høyden.

4.8.1 Bredde på QR-koder

Med forbehold om at alle QR-kodene er identiske i størrelse, er det være sannsynlig at QR-koder plassert i høyden er større enn de som er plassert nede på bordet.

For å approksimere høyden på en stabel ble det laget en enkel funksjon med grenseverdier for QR-kodenes bredde. Det ble tatt flere bilder i flere høyder over QR-koden hvor både høyden og QR-kodens bredde ble lagret. En kunne da relativere bredden på en QR-kode med en høyde i funksjonen. Funksjonen returnerer den approksimerte arbeidsdistansen.

Dette fungerte ikke bra nok til å inkluderes i demonstrasjonsprogrammet. En utfordring med plukking i høyden var at størrelsen på QR-kodene varierte med plassering til klossene på arbeidsobjektet. En kloss i midten av bordet er tilsynelatende større enn en kloss som er i kanten av kameraets synsfelt. Dette ble nedprioritert til fordel for andre utvidelser, som kollisjonshindring (i kapittel 2.3.3).

4.8.2 Annerledes kameraorientering

En annen mulig løsning, som ikke ble forsøkt realisert, innebærte å ta bilde på skrå eller fra siden av bordet. Dette ble nedprioritert på lik linje med idéen i kapittel 4.8.1.

4.9 Stemmestyring av prosessen

En mulig utvidelse til oppgaven var å implementere stemmestyring av prosessen, for å illustrere en robot med både “syn og hørsel”. Det ble, gjennom bruk av Python-biblioteket *SpeechRecognition*, utviklet en enkel måte å styre demonstrasjonsprogrammet på. [40] Likevel er ikke stemmestyring inkludert i prosjektets endelige løsning. Grunnen til dette hadde bakgrunn i vårt ønske om å skape et robust system. I et demonstrasjonsprogram som i denne oppgaven, er det avgjørende at alt fungerer nøyaktig som det skal. Dersom det ikke er tilfelle, vil

systemet virke mindre imponerende, og formålet med det hele falmer.

Talegenkjenning er riktignok i konstant utvikling, men det er fortsatt mange faktorer som påvirker teknologiens suksessrate. Disse inkluderer teknologiske faktorer (som mikrofonkvalitet og opptakskvalitet), miljøfaktorer (som bakgrunnsstøy eller flere som prater om gangen) og språkutfordringer (som homofoner, personlig uttale og aksenter). [41] Ved å lage et større rammeverk rundt talegenkjenningen med en ekstensiv unntakshåndtering vil suksessraten kunne økes, men det ble ikke sett på som en fornuftig bruk av tid.

Dersom oppgavens tidsramme hadde vært større, ville det være naturlig å utvide demonstrasjonsprogrammets brukergrensesnitt. Stemmostyring av prosessen ble i oppgavens tidlige stadier sett på som en god kandidat for dette. I senere tid ble dette sidestilt til fordel for en idé om et grafisk brukergrensesnitt. Dette omtales i mer detalj i kapittel 4.10.

4.10 Grafisk brukergrensesnitt

Demonstrasjonsprogrammet bruker et veldig enkelt grensesnitt som består av *prints* og *inputs* i Python-konsollen. Systemet har potensial for å bli visuelt imponerende samt meget interaktivt, dersom det utvides med et grafisk brukergrensesnitt. Vi mener at et slikt grensesnitt kan være både mer robust og mer passende enn en stemmostyrt prosess (ref. kap. 4.9).

Idéen vår baserte seg på et vindu som viser video fra kameraet. For hver QR-kode som ble lest, skulle en knapp skapes “bak” hver kloss i bildet. Posisjonen til knappen bestemmes fra posisjonen til QR-koden i bildet (før posisjonskalkulering). Ved trykk på en kloss, vil den velges og plukkes av griperen. I tillegg ville det være mulig å markere på bildet hvor klossen skal plasseres igjen. Dette kan muliggjøres ved å hente ut pekerposisjonen i bildet og lage et *robtarget* ved metoden i kapittel 2.4.

4.11 XML kontra JSON

Å hente ut informasjon fra XML i Python er generelt mer krevende enn JSON. Derfor unngås XML i demonstrasjonsprogrammet og *rwsuis*-biblioteket.

Ved første løsning av oppgaven, ble XML brukt i enkelte GET-forespørsler hvor JSON tilsvynelatende ikke var støttet. I senere tid viste det seg at JSON støttes i alle RobotWare-ressurser som ble brukt i oppgaven. Det er uklart om dette er vår egen skyld (i form av tidligere dårlig syntaks i spørreparametene) eller om

det skyldes utbedringer av Robot Web Services, som har vært under videreutvikling samtidig som oppgaven ble løst.

Robot Web Services mangler fortsatt JSON-støtte for OPTIONS-forespørsler og abonnementstjenesten (*subscription service*). En naturlig utvidelse til denne oppgaven er å ta i bruk abonnementstjenesten for å abonnere på ulike ressurser i RobotWare. Ressurser som er nyttige å abonnere på inkluderer *controller state* og *operation mode* som viser om motorene er av eller på og om kontrolleren står i automatisk eller manuell modus. Abonnementstjenesten støtter bare XML-respons, og dette må parses i Python. I vår tidligere løsning, ble *Elementtree* fra Pythons standardbibliotek brukt for å parse XML. [42]

4.12 Lagring av klossinformasjon i Python

I demonstrasjonsprogrammet er det nødvendig å lagre informasjon om flere klosser samtidig. I oppgavens første løsning ble det brukt en global ordliste som inneholdt nødvendig informasjon som klossnummer, -posisjon og -orientering. Denne ordlisten ble initialisert i en oppstartsfil, *config.py*, som ble importert fra *main.py* og dermed kjørt. Siden ordlistene er foranderlige (“mutable”) i Python, ble ordlisten aldri lagt inn som innargument i funksjoner eller returnert fra funksjonene. For hver utvidelse av programmet ble det vanskeligere å følge programflyten. Det ble tydelig at dette ikke var god programmeringspraksis.

Det ble senere laget en *Puck*-klasse (kap. 2.3.3) som ble brukt for å lage *Puck*-objekter for hver QR-kode. Disse ble plassert i en liste som ble initialisert i *main.py* uten å brukes globalt.

4.13 Konklusjon

I denne oppgaven har vi laget et system som er i stand til å lokalisere, identifisere, orientere og stable klosser som er tilfeldig plassert i arbeidsområdet. Dette er gjennomført med kamera, QR-koder og kommunikasjon mellom Python og robot. I oppgaveløsningen ble det valgt å prioritere robusthet og effektivitet i robotsystemet. Vi mener at resultatene i repeterbarhetstesten (kapittel 3.4) viser at vi har lykkes med dette.

Gjennom en omfattende kodedokumentasjon og et nøyne valg av programstruktur, kan det utviklede systemet med stor sannsynlighet dras nytte av i flere år fremover ved UiS. Som en ytterligere konsekvens av dette, har systemet et stort utvidelsespotensial. Dette reflekteres spesielt i Python-biblioteket *rwsuis*. [29]

Referanser

- [1] ABB. *RobotStudio*. 2016. URL: <http://new.abb.com/products/robotics/robotstudio>. (sjekket: 20.05.2020).
- [2] Henrik Berlin. *Text based robot programming made easy*. 2012. URL: <http://www.abb.com/blog/gad00540/1DDE6.aspx?tag=RAPID%5C%20programming>. (sjekket: 12.02.2020).
- [3] Karl Skretting. *ABB robot assignment 1*. Stavanger, jan. 2020. URL: <http://www.ux.uis.no/~karlsk/ELE610/rs1.pdf>.
- [4] ABB. *Robot Web Services*. Versjon 1.0. URL: <https://developercenter.robotstudio.com/api/rwsApi/>. (sjekket: 16.05.2020).
- [5] IDS Imaging Development Systems GmbH. *uEye XS*. URL: <https://en.ids-imaging.com/store/xs.html>. (sjekket: 16.05.2020).
- [6] Travis E. Oliphant. *NumPy*. URL: <https://numpy.org/>. (sjekket: 18.05.2020).
- [7] Open Source Computer Vision Library. *OpenCV*. URL: <https://opencv.org/about/>. (sjekket: 18.05.2020).
- [8] Python Software Foundation. *configparser — Configuration file parser*. URL: <https://docs.python.org/3.7/library/configparser.html>. (sjekket: 18.05.2020).
- [9] Lawrence Hudson. *pyzbar*. Versjon 0.1.8. URL: <https://pypi.org/project/pyzbar/>. (sjekket: 16.05.2020).
- [10] Jeff Brown. *ZBar bar code reader*. URL: <http://zbar.sourceforge.net/index.html>. (sjekket: 16.05.2020).
- [11] IDS Imaging Development Systems GmbH. *IDS Software Suite*. Versjon 4.93. URL: <https://en.ids-imaging.com/manuals/ids-software-suite/ueye-manual/4.93/en/index.html>. (sjekket: 25.02.2020).
- [12] IDS Imaging Development Systems GmbH. *PyuEye*. Versjon 4.90.0.0. URL: <https://pypi.org/project/pyueye/>. (sjekket: 25.02.2020).
- [13] Kenneth Reithz. *Requests: HTTP for Humans*. URL: <https://requests.readthedocs.io/en/master/>. (sjekket: 16.05.2020).
- [14] Jonatan Heyman mfl. *What is Locust?* URL: <https://docs.locust.io/en/stable/what-is-locust.html>. (sjekket: 16.05.2020).
- [15] Georg Brandl mfl. *SPHINX Python Documentation Generator*. URL: <https://www.sphinx-doc.org/en/master/>. (sjekket: 16.05.2020).
- [16] David Goodger. *reStructuredText Markup Specification*. URL: <https://docutils.sourceforge.io/docs/ref/rst/restructuredtext.html>. (sjekket: 18.05.2020).

- [17] Douglas Crockford. *Introducing JSON*. URL: <https://www.json.org/json-en.html>. (sjekket: 20.02.2020).
- [18] IDS Imaging Development Systems GmbH. *Sensor data UI-1007XS — IDS Software Suite*. Versjon 4.93. URL: <https://en.ids-imaging.com/manuals/ids-software-suite/ueye-manual/4.93/en/sensor-data-ui-1007xs.html>. (sjekket: 22.05.2020).
- [19] IDS Imaging Development Systems GmbH. *Setting the focus — IDS Software Suite*. Versjon 4.93. URL: <https://en.ids-imaging.com/manuals/ids-software-suite/ueye-manual/4.93/en/ui-1007xs-xs-cockpit-set-focus.html>. (sjekket: 8.05.2020).
- [20] IDS Imaging Development Systems GmbH. *Position accuracy of the sensor USB uEye XS*. URL: <https://en.ids-imaging.com/manuals/ids-software-suite/ueye-manual/4.93/en/sensor-position-usb-ueye-xs.html>. (sjekket: 29.02.20).
- [21] Alexander Mordvintsev og Abid Rahman K. *Smoothing Images*. 2013. URL: https://opencv-python-tutorials.readthedocs.io/en/latest/py-tutorials/py_imgproc/py_filtering/py_filtering.html. (sjekket: 21.02.20).
- [22] GitHub-bruker: sushil-bharati. *fix: in-odered polygon #39*. URL: <https://github.com/NaturalHistoryMuseum/pyzbar/pull/39>. (sjekket: 29.05.2020).
- [23] Markus H Iversflaten og Ole Christian Handegård. “*Puck Class*” i *ABB Robot with Machine Vision*. Mai 2020. URL: <https://abb-robot-machine-vision.readthedocs.io/en/latest/puck.html>. (sjekket: 29.05.2020).
- [24] IDS Imaging Development Systems GmbH. *XS WORKING DISTANCE AND FIELD OF VIEW*. URL: <https://en.ids-imaging.com/xs-fieldofview.html>. (sjekket: 16.05.2020).
- [25] Grant Sanderson og Ben Eater. *Visualizing quaternions*. URL: <https://eater.net/quaternions>. (sjekket: 19.02.2020).
- [26] Wikipedia.org. *Representational state transfer*. URL: https://en.wikipedia.org/w/index.php?title=Representational_state_transfer&oldid=950802505. (sjekket: 13.04.2020).
- [27] Wikipedia.org. *HTTP persistent connection*. URL: https://en.wikipedia.org/w/index.php?title=HTTP_persistent_connection&oldid=918823577. (sjekket: 19.02.2020).
- [28] Henning Klevjer, Audun Jøsang og Kent Varmedal. «Extended HTTP Digest Access Authentication». I: apr. 2013, s. 3–5. DOI: [10.1007/978-3-642-37282-7_7](https://doi.org/10.1007/978-3-642-37282-7_7).

- [29] Markus H Iversflaten og Ole Christian Handegård. *rwsuis*. Versjon 0.2. Mai 2020. URL: <https://pypi.org/project/rwsuis/>. (sjekket: 27.05.2020).
- [30] Markus H Iversflaten og Ole Christian Handegård. “*Robot Web Services*” i *ABB Robot with Machine Vision*. Mai 2020. URL: <https://abb-robot-machine-vision.readthedocs.io/en/latest/robotwebservices.html>. (sjekket: 29.05.2020).
- [31] Roy Fielding og Julian Reschke. *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content*. RFC 4180. Jun. 2014, s. 21–33. URL: <https://tools.ietf.org/html/rfc7231>.
- [32] Python Software Foundation. *JSON encoder and decoder*. URL: <https://docs.python.org/3.7/library/json.html>. (sjekket: 16.05.2020).
- [33] *Locust Quick Start*. URL: <https://docs.locust.io/en/stable/quickstart.html>. (sjekket: 16.05.2020).
- [34] Markus H Iversflaten og Ole Christian Handegård. *ABB Robot with Machine Vision*. Mai 2020. URL: <https://abb-robot-machine-vision.readthedocs.io/>. (sjekket: 29.05.2020).
- [35] Karl Skretting. *ABB robot assignment 4*. Stavanger, jan. 2020. URL: <http://www.ux.uis.no/~karlsk/ELE610/rs4.pdf>.
- [36] Intel Corporation. *Depth Camera D435*. URL: <https://www.intelrealsense.com/depth-camera-d435/>. (sjekket: 31.03.2020).
- [37] Intel Corporation. *pyrealsense2*. Versjon 2.33.1.1382. URL: <https://pypi.org/project/pyrealsense2/>. (sjekket: 31.03.2020).
- [38] Wikipedia.org. *Distortion (optics)*. URL: [https://en.wikipedia.org/w/index.php?title=Distortion_\(optics\)&oldid=951242587](https://en.wikipedia.org/w/index.php?title=Distortion_(optics)&oldid=951242587). (sjekket: 17.04.2020).
- [39] MathWorks. *What Is Camera Calibration*. URL: <https://se.mathworks.com/help/vision/ug/camera-calibration.html>. (sjekket: 16.05.2020).
- [40] Anthony Zhang. *SpeechRecognition*. Versjon 3.8.1. URL: <https://pypi.org/project/SpeechRecognition/>. (sjekket: 11.03.2020).
- [41] John Levis og Ruslan Suvorov. «Automatic Speech Recognition». I: nov. 2012, s. 4. DOI: 10.1002/9781405198431.wbeal0066.
- [42] Python Software Foundation. *xml.etree.ElementTree — The ElementTree XML API*. URL: <https://docs.python.org/3/library/xml.etree.elementtree.html>. (sjekket: 10.02.2020).

A Vedlegg

Vedleggene til rapporten er innbakt i PDF-en som en komprimert mappe.

Siden laboratorieoppgaven er en sentral del av rapporten, er denne inkludert som et synlig vedlegg.

A.1 Laboratorieoppgave

ELE610 Robot Technology

Markus Iversflaten, Ole Christian Handegård

May 28, 2020

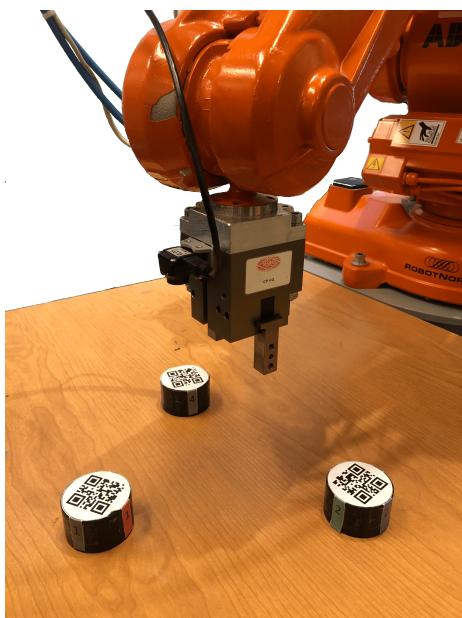


Figure 1: Robot setup

ABB Robot Assignment X

In this assignment you will be working in both Python 3 and RobotStudio. The goal is to identify the position of several, randomly placed pucks in the work area by capturing an image in Python, and telling RobotWare where the pucks are, so they may be picked and placed. For this, you will need:

- Python 3
- RobotStudio
- Norbert, with gripper and camera mounted to gripper

1 Control robot from Python

You may start from the same Pack-and-go file used earlier, it contains a station similar to the actual laboratory in E458. If you haven't already done so, download the Pack-and-go-file, [UiS_E458_nov18.rspag](#), and make sure that file extension is rspag.

1.1 Robot Web Services

Communication will happen through the **Robot Web Services API**, which is made by ABB. Through this API, you gain access to all RobotWare resources. Accessing these resources from Python is quite simple. It is important that you familiarize yourself with some of these resources, namely the **Mastership**, **Panel** and **RAPID** services, which are listed under **RobotWare** services.

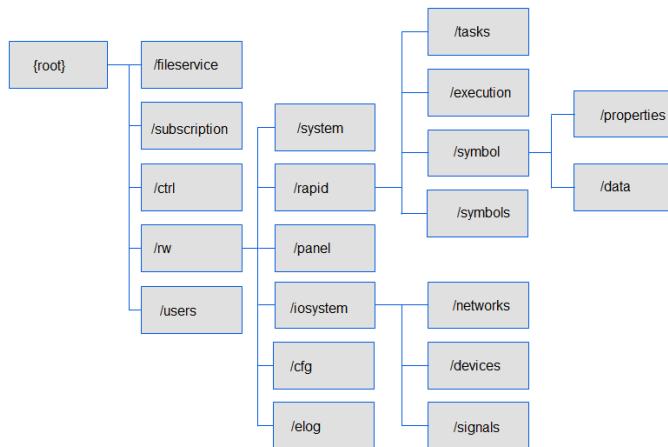


Figure 2: The resource hierarchy in Robot Web Services, retrieved from <https://developercenter.robotstudio.com/api/rwsApi/>

1.2 Introduction

In this assignment, there are several files you need. These will hopefully simplify the laboratory work.

- [UiS_E458_nov18.rspag](#), the Pack-and-go file.
- [PythonCom.mod](#)
- [Communication package \(rwsuis\)](#)

1.3 Establish connection

Now, you should test communication between Python and RobotWare. This can be done in several ways, but the easiest might be to poll the controller state, which means to check whether the robot's motors are turned **on** or **off**.

1.3.1 Making a GET request

Polling the controller state is done through a **GET request**. GET requests are safe methods, which means that they *don't change* any resources. To make a GET request, you will need a few things.

First: A way to make HTTP requests in Python. You may use **urllib** or **Requests**, which are both libraries for Python. We strongly recommend using Requests, which is very easy to use. The solution is also written using Requests.

Second: Get the right address for the HTTP request. To access Norbert's controller, you will need its IP address. You can find the IP address by checking the available controllers in RobotStudio. To access the hierarchy of resources (figure 2), the address will then be "http://152.94.0.38". To poll the controller state, you will need to find the location of the controller state resource. It is located in **RobotWare Services** → **Panel Service** → **Operations on Controller State Resource** → **Get Controller State**. The **Sample Call** for this resource gives us an idea of how to make the HTTP request. The sample call here is:

```
curl -digest -u "Default User":robotics "http://localhost/rw/panel/ctrlstate".
```

The complete address you need is marked in bold. When working on a physical robot, "localhost" needs to be changed to the IP address of the robot, which you have already found. Your address will then be:

"**http://{Robot's IP}/rw/panel/ctrlstate**".

Third: The username and password to access the robot's controller. By default in all RobotWare controllers, the username is "**Default User**" and the password is "**robotics**". To enter these credentials, you will need to use digest access authentication. This is already included in the Requests package, so you don't have to worry about it. In Python, the username and password will be used like this:

```
1 auth = requests.auth.HTTPDigestAuth("Default User", "robotics")
```

After all these things are found, you are ready to make your first GET request. Here is how to poll the controller state of Norbert in Python:

```
1 response = requests.get("http://152.94.0.38/rw/panel/ctrlstate",
    auth=auth)
```

To check if the response was successful, you'll need to print the response to the console in Python. This will yield an HTTP status code. If the status code is 2xx, then the request was successful, and a connection was established. If the status code is anything else, then something in your request might be wrong.

To see the actual content in the response, you may print the message delivered:

```
1 print(response.text)
```

This will yield an XML formatted message (which may seem a bit excessive) and somewhere in there, you should find either “motoron” or “motoroff”. It is also possible to receive messages in JSON. This can be done by adding a query string (“json=1”) at the end of the resource address:

```
1 response = requests.get("http://152.94.0.38/rw/panel/ctrlstate?json
    =1", auth=auth)
```

In principle, this is how all GET requests work. Going forward, you may use our package **rwsuis**, which includes the RWS class. This class has predefined functions for all the requests you will need during this assignment. All the requests in the class pick out only the information needed; e.g. polling the controller state through the RWS class would yield *only* “motoron” or “motoroff”. Check out the documentation [here](#).

The RWS class also contains several POST requests. POST requests are not safe methods, as they *change* resources. This means that they require mastership from the robot controller.

1.3.2 Making a POST request

You will now attempt to change the translational data (x,y,z) of a robtarget variable in RAPID through a POST request. This is very similar to making GET requests.

First:

You will need to create a simple RAPID module where you declare a robtarget. The robtarget **cannot** be a constant (CONST in RAPID). It must be declared as a variable (VAR or PERS) robtarget in order to be modifiable (this in fact goes for *any* variable you want to change in RAPID from Python). You will also need to make a simple main procedure which will run in a “WHILE TRUE DO”-loop.

```

1 VAR robtarget simple_robtarget :=
    [[0,0,100],[0,1,0,0],[-1,0,0,0],[9E+09,9E+09,9E+09,9E
    +09,9E+09]];
2
3 PROC main()
4     WHILE TRUE DO
5         MoveJ simple_robtarget, v200, z10, tGripper\WObj:=
        wobjTableN;

```

Second:

You will need to create a Python script which does two things: requests mastership of the controller and makes a POST request to change a robtarget's translational data. To do this, you will first create an **RWS** object, from the package:

```

1 from rwsuis import RWS
2
3 robot = RWS.RWS("{Robot's IP}")

```

With an RWS object, you gain access to all methods in the RWS class. You will now use some of these to change the robtarget variable in RAPID. First, you will use it to request mastership, as such:

Requesting mastership in manual mode:

```

1 robot.request_rmmr() # Mastership request in manual mode
2 time.sleep(10) # Give time to accept manual request

```

When requesting mastership in manual mode, you must accept the request on the FlexPendant within 10 seconds.

Requesting mastership in automatic mode:

```

1 robot.request_mastership() # Mastership request in automatic mode

```

In automatic mode, mastership is automatically given when requested.

When you have acquired mastership, you may try to send new translational data to the robtarget in RAPID. This should be done using **set_robtarget_translation**, as shown below:

```

1 new_robtarget = [0, 0, 400]
2 robot.set_robtarget_translation("{variable name in RAPID}",
        new_robtarget)

```

Note: The program in RAPID has to be running *before* you start your Python script. You should then be able to modify the robtarget in RAPID through Python code. Try it!

You've now been through the communication basics which you will use to create your main program later on.

1.4 Scanning QR codes

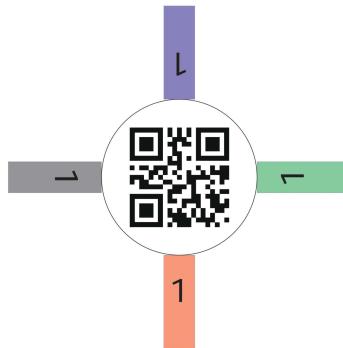


Figure 3: QR code taped on pucks

To locate the pucks in the work area, you should scan the QR codes on top of them. This will reveal the pucks' positions. To do this, you must first capture an image with the camera in Python, and then put the image through a QR scanner. It may also be a good idea to process the image before passing it to the QR scanner. This will make the image easier to decode, which in turn will make your solution more robust.

1.4.1 Capturing an image

To capture images in this assignment, you will need to use a **uEye XS** camera. You can capture the image either through **OpenCV** or **PyuEye**.

Using OpenCV:

The easiest way to retrieve an image from the camera in Python is through OpenCV. OpenCV has a general API for capturing images, and thus, has only general functionality. This means that many of the cameras parameters will be left untouched.

Here is how to capture an image in OpenCV:

```
1 import cv2
2
3 cap = cv2.VideoCapture(1)
4
5 # Change resolution to 1280x960
6 cap.set(3, 1280)
```

```
7 cap.set(4, 960)
8
9 ret, frame = cap.read()
```

Setting the image resolution can also be done in OpenCV, as shown. This will prove useful later.

Using PyuEye:

IDS has created their own API, **PyuEye**, for controlling their cameras. With this, you gain access to all functionality within the XS camera through Python. This also means that the API is harder to use than OpenCV.

For this assignment, OpenCV should be sufficient. If you still would like to use PyuEye, then you may find some help in the functions provided in the **IDS Software Suite**.

1.4.2 Image processing

The QR scanner might not be able to decode images with too much noise and/or too little contrast. You should therefore try to reduce noise and increase the contrast in the image.

Reducing image noise:

Reduced image noise can be achieved through image filtering. Some of the most common filters are Gaussian blur, Median filter and Bilateral filter. These all exist in the OpenCV library. You must choose one of these to use, by reading about them [here](#) or elsewhere on the web.

Note: when working with QR codes, some filters are definitely better than others. Try to find the best one!

Increasing image contrast:

Increasing the contrast in the image will make QR codes stand out more. There are several ways to achieve a greater image contrast. Methods worth checking out: **cv2.normalize**, **cv2.equalizeHist** and **basic linear transform**.

1.4.3 Installing a QR scanner

There are several QR scanners for Python. In our experience, **ZBar** is both fast, easy to use, and has all the functionality you will need. It should already be installed on the laboratory computers. To use ZBar in Python 3, you will need to install **pyzbar**. This can easily be done by: `pip install pyzbar`

1.4.4 Using the QR scanner

The only function you will need from pyzbar is **decode**. decode only takes an image, and returns a **Decoded object** (if the image contains QR codes).

Basic usage:

```
1 from pyzbar.pyzbar import decode  
2 ...  
3 data = decode(image)
```

The object will look like this:

```
1 Decoded(  
2     data=b'Puck#1', type='QR CODE',  
3     rect=Rect(left=27, top=27, width=145, height=145),  
4     polygon=[Point(x=27, y=27), Point(x=27, y=172), Point(x=172, y  
=172), Point(x=172, y=27)]  
5 )
```

To access any information from the QR code, simply navigate through the different object parameters:

```
1 >>> data.polygon  
2 polygon =[Point(x=27, y=27), Point(x=27, y=172) , Point(x=172, y  
=172), Point(x=172, y=27)]
```

1.5 Creating a robtarget

The **Decoded** object from the QR scanner contains positional data of all detected QR codes. This is found in the “polygon” parameter. You will need to use this to pinpoint the center of the QR code. It is important to know that this position **cannot** be used directly in RAPID. There are several steps to creating a robtarget that is usable in RAPID.

First:

Images in Python are usually represented in matrix form. This means that x increases as you go to the right and y increases as you go downwards, as illustrated in figure 4 below. It also means that the origin (0,0) is in the top left, not in the middle as you might be used to. The work object, however, has origin in the middle.

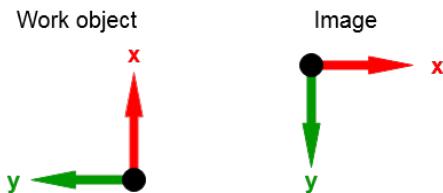


Figure 4: Coordinate systems for work object and image

To relate a position in the image to a position on the work object, they must be in the same reference frame. Therefore, you will have to transform the coordinates found in the image to the coordinate system used in the work area.

First, you'll need to make the center of the image the origin. To do this, simply subtract half of the height and width of the image from the coordinates.

You must then transform the axes. For figure 4, the transformation would look like this: $x \rightarrow y$, $y \rightarrow x$, $x \rightarrow -x$, $y \rightarrow -y$.

Second:

You must consider the units used in the image taken and the units used by the robot. The image uses pixels as the measuring unit, whereas the robot uses millimeters. A conversion from pixels to millimeters is therefore required. Finding this conversion requires knowledge of the camera's FOV¹. The FOV is found by relating it to the working distance and the camera's physical properties. Figure 5 shows this relation.

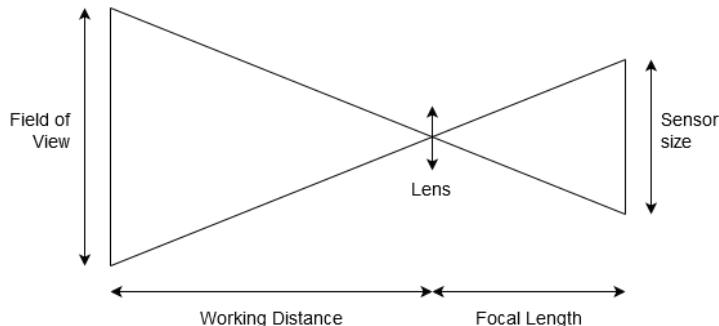


Figure 5: Relation between field of view and sensor size

The focal length and sensor size of the camera are already known. The working distance is the height from the camera's lens to the subject, and will therefore vary.

$$\text{Focal length}^2 = 3.7\text{mm}$$

$$\text{Sensor width} = 3.6288\text{mm}$$

The working distance is found by first relating the camera's position to the

¹Field of View

²This is a simplification. The focal length also depends on the amount of zoom applied by the lens. This gives a potential error of $\pm 5\%$. A more correct calculation can be performed here: [IDS XS FOV calculator](#).

gripper's. The lens of the camera is approximately 70mm above and 55mm in front of the gripper (in relation to the tool data in RobotStudio). Therefore, by knowing the gripper's position, you also know the position of the camera. It is important to note that the position of the camera is in relation to the work object (the table).

The subject should be chosen to be the QR codes. As you should know, the pucks are 30mm in height, which means that the QR codes will be 30mm above the work object.

Using this information, one can find the working distance for any gripper height. A gripper height of **400mm** will for example yield a working distance as such:

$$\text{working_distance} = 400 + 70 - 30 = 440\text{mm}$$

With this knowledge, it is possible to use shape similarity to find the FOV:

$$\frac{\text{FOV_width}}{\text{working_distance}} = \frac{\text{sensor_width}}{\text{focal_length}} \quad (1)$$

Using equation 1, you can find the field of view and the conversion from pixels to millimeters:

$$\text{FOV_width} = \frac{\text{sensor_width}}{\text{focal_length}} \cdot \text{working_distance}$$

$$\text{pixels_to_millimeters} = \frac{\text{FOV_width}}{\text{resolution_width}}$$

If working with a resolution of 1280x960, your resolution width would be 1280 pixels.

Third:

The position you've now created through the transformation and conversion is in relation to the camera. To actually pick up a puck, you will have to relate the position to the gripper. As previously mentioned, you will capture an image with the camera in front of the gripper, without any rotation. This means that the camera will be 55mm in front of the gripper. You must relate this to your coordinate system (for figure 4 this would be 55mm in the positive x-direction).

Finally:

You must take into consideration the inaccuracy of the placement of the camera.

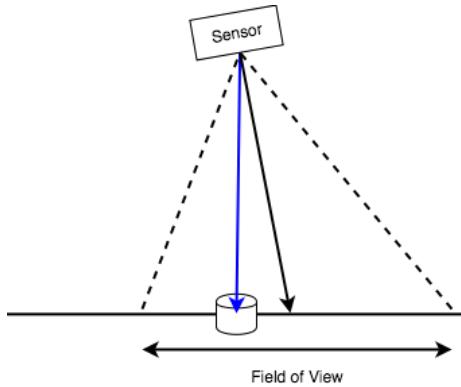


Figure 6: Inaccuracy in camera and mount

Most likely, the mount and USB cable together tilt the camera slightly. In addition, the camera itself has an inaccuracy in the lens, which can provide further tilt. Figure 6 displays this problem. In some way, the angle of the camera must be found, so it may be compensated for when creating the robtarget.

The easiest way to do this is to compare results from two different images captured at different heights. To do this, all previous steps **must first be completed**. At this point, you may use the template `cam_adjust_lab.py`. With this routine, you are meant to place one puck in the view of the robot. You will need to fill in the missing code with your own functions, found in the previous steps. After the routine has finished, you will be given slope values for x and y which shall be used to compensate for the tilted camera. These should be used together with the working distance to find the compensation values.

$$comp_x = slope_x \cdot working_distance$$

$$comp_y = slope_y \cdot working_distance$$

Let's go through an example!

Used parameters:

- Gripper height = 500mm
- Camera resolution = 1280x960
- Coordinate systems such as in figure 4

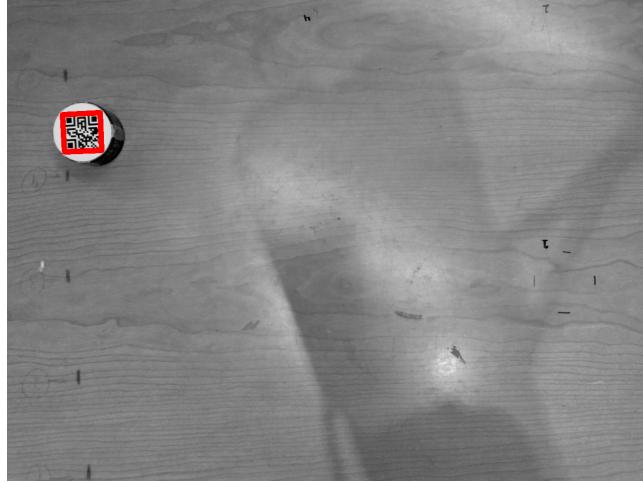


Figure 7: Image of scanned puck

The scanned image yields a puck position: $(x, y) = (100, 150)$.

First, make the center of the image the origin:

$$(x, y) = (100 - (1280/2), 150 - (960/2)) = (-540, -330)$$

Next, apply the appropriate transformation. With coordinate systems such as in figure 4, the transformation yields a new puck position: $(x, y) = (330, 540)$.

Now, convert the position to millimeters:

$$FOV_width = \frac{sensor_width}{focal_length} \cdot working_distance = \frac{3.6288}{3.7} \cdot (500 + 70 - 30) = 529.6$$

$$pixels_to_millimeters = \frac{FOV_width}{resolution_width} = \frac{529.6}{1280} = 0.414$$

Now, multiply the position coordinates with the conversion number. This can be done elegantly in Python through **list comprehension** (note that `puck.position` only contains x- and y-coordinates in this case):

```
1 puck.position = [x * pixels_to_millimeters for x in puck.position]
```

This yields a new puck position: $(x, y) = (136.6, 223.6)$.

Recall that the camera is mounted 55mm in front of the gripper. In addition, the gripper's position is needed. In this case, it is simply at the origin.

In other cases, though, it will differ. Therefore it is best to make a request to RobotWare for the gripper position. This can be done easily through the package:

```

1 from rwsuis import RWS
2
3 norbert = RWS.RWS("http://152.94.0.38")
4 gripper_position = norbert.get_gripper_position()
5
6 camera_position = [gripper_position[0] + 55, gripper_position[1]]
```

$$\begin{aligned} \text{1 } &>>> \text{ camera_position} \\ \text{2 } &[55, 0] \end{aligned}$$

Using this information, it is simple to add the position of the camera to the puck position: $(x, y) = (191.6, 223.6)$.

This position will be *nearly* accurate enough to be used to pick up pucks. However, as previously mentioned, the camera is likely tilted in some fashion.

The tilt is compensated for by using the slope values found through the camera adjustment routine. In this case, the values were:

$$slope_x = 0.0229$$

$$slope_y = -0.0003$$

The amount of compensation depends on the slope values and the working distance:

$$comp_x = slope_x \cdot working_distance = 0.0229 \cdot (500 + 70 - 30) = 12.366$$

$$comp_y = slope_y \cdot working_distance = -0.0003 \cdot (500 + 70 - 30) = -0.162$$

These compensation values must be **subtracted** from the puck position:

Puck position: $(x, y) = (191.6 - 12.366, 223.6 - (-0.162)) = \underline{\underline{(179.2, 223.8)}}$.

The acquired puck position should now be accurate enough to be used for picking up the puck! However, to be completely sure that the accuracy is high enough, you **should always** capture two images of the same puck. The first image should be captured from an overview position (e.g. with gripper height of 500mm), and the second image from a closer view (e.g. with gripper height of 60mm).

1.6 Program structure

Order is everything, when running several scripts in parallel. Generally, some block of code should be run in one of the scripts, before a block of code from the other script is run, and so on. To ensure that the scripts are executed in the required order, you should use *flag variables*, both in Python and RAPID. Flag variables are usually **booleans**.

For example, take two arbitrary script running in parallel: script 1 & script 2, with flag variables 1 & 2, respectively.

When a certain part of a script 1 is finished, flag variable 1 should be set to “TRUE”. Once this happens, script 1 should enter a waiting loop, waiting for script 2 to finish whatever it’s doing. As flag variable 1 became TRUE, script 2 can continue its execution. Before this happens, though, the flag variable **must be reset** (be set to “FALSE”). Script 2 now continues its execution. Once finished with a certain code block, flag variable 2 should be set to TRUE, and so on...

In the predefined methods, the flag variables are called `image_processed` and `ready_flag` in Python and RAPID respectively.

Predefined methods:

```
1 # Wait method used in Python
2 def wait_for_rapid(self, var='ready_flag'):
3     """Waits for robot to complete RAPID instructions
4     until boolean variable in RAPID is set to 'TRUE'.
5     Default variable name is 'ready_flag', but others may be used.
6     """
7
8     while self.get_rapid_variable(var) == "FALSE" and self.
9         is_running():
10            time.sleep(0.1)
11            self.set_rapid_variable(var, "FALSE")
```

```
1 ! Wait method used in RAPID
2 PROC wait_for_python()
3     ! Wait for Python to finish processing image
4     WHILE NOT image_processed DO
5     ENDWHILE
6     image_processed:=FALSE;
7 ENDPROC
```

1.7 Final preparation

To make sure that all hardware and software is correctly configured, you should try [this short program](#). Make sure to install all dependencies, i.e. the packages that are imported in the project files. You should place a puck on the work object. When the program is run, the robot should find the puck and position its gripper directly above the puck. If this succeeds, you are ready to start building the main program.

1.8 Creating a working program

At this point, you should have sufficient material to create working Python and RAPID scripts. To start, you may use these template files:

[PythonCom_template.mod](#) and [main_template.py](#).

Notes:

You've previously created procedures in RAPID for both picking and placing pucks. These may be reused, with a slight change to picking: The camera should now be approximately centered over the puck before the gripper goes down to gripping height (10mm above work object). The gripper should then "slide" in toward the puck before gripping it. This technique can be seen in the video [here](#). This change prevents the puck and QR codes from getting damaged, should the gripper miss its target.



Figure 8: Gripping technique

As previously mentioned, you should always capture two images of the same puck. To capture the second image (from a closer view), you will need to position the camera above the acquired robtarget. The second image should have a much greater accuracy than the first.

Before any image is captured, there must be a pause. The pause should ensure that the robot is perfectly still, and that the camera has had time to adjust its autofocus correctly. This can be done in Python through `time.sleep(secs)`.

Optional:

The four colors down the sides of each QR code (see figure 3) can be used to visualize the puck's rotation. By stacking pucks with the same orientation, these colors will line up nicely. This is also shown in the [video](#). It is possible to find the pucks' orientations through the QR codes. For this, you **must** download and use a [tweaked pyzbar package](#), while also uninstalling the previously used pyzbar package. This package makes the corner positions of the QR codes appear in the same order regardless of their orientation. For figure 9, corners 1 & 4 were used to find the puck's orientation.

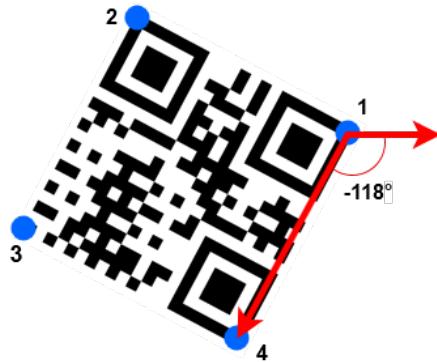


Figure 9: QR code orientation