Ville Seppänen

TEOLLISUUSROBOTIN OHJAAMINEN KONENÄKÖSOVELLUKSELLA

Opinnäytetyö
Sähkö- ja automaatiotekniikka

2020

Kaakkois-Suomen ammattikorkeakoulu

Kaakkois-Suomen ammattikorkeakoulu

Tekijä/Tekijät	Tutkinto	Aika
Ville Seppänen	Insinööri (AMK)	Huhtikuu 2020

Opinnäytetyön nimi	
Teollisuusrobotin ohjaaminen konenäkösovelluksella	73 sivua 16 liitesivua

Toimeksiantaja

Kaakkois-Suomen ammattikorkeakoulu

Ohjaaja

Teemu Manninen

Tiivistelmä

Tämän opinnäytetyön aiheena oli liittää ABB:n valmistamaan teollisuusrobottiin konenäkökamera siten, että kameran ottaman kuvan perusteella robotti hakee työalueelta siitä löytyvät kappaleet. Kamera ottaa työskentelyalueesta valokuvan, ja siihen liitetty tietokone analysoi sen sisällön. Kuvasta voidaan tunnistaa tietynväriset kappaleet ja niiden sijainnit, minkä jälkeen ne voidaan lähettää robotille sen ymmärtämässä muodossa. Robotti ottaa vastaan kameralta saadun informaation ja noutaa työskentelyalueelta halutut kappaleet.

Opinnäytetyön toteutukseen käytettiin ABB IRB 120 -teollisuusrobottia, Python-ohjelmointikieltä, OpenCV-konenäkökirjastoa sekä Raspberry Pi -kehitysalustaa ja siihen liitettävää kameraa. Työn lähtökohtana oli, että valmista työtä voitaisiin jatkossa käyttää kurssimateriaalina automaatiotekniikan kursseilla.

Työn käytännön toteutus koostui käytännössä kolmesta pääasiallisesta vaiheesta. Ensin kirjoitettiin konenäkösovellus Raspberry Pi -tietokoneella käyttäen Python-ohjelmointikieltä ja siinä OpenCV-konenäkökirjastoa. Seuraavaksi tietokone ja robotti ohjelmoitiin muodostamaan yhteys keskenään, jotta kommunikointi kameralta saadun datan ja robottisolun välillä oli mahdollista. Viimeinen vaihe oli laitteiston kasaus ja testaus siten, että robottisolu kykeni suorittamaan halutut tehtävät. Jokainen vaihe oli verrattain työläs, etenkin koska aiempaa kokemusta konenäkölaitteista tai teollisuusroboteista ei ollut.

Lopputuloksena oli toimiva konenäkösovellus, jonka avulla robotti kykeni noutamaan halutut kappaleet hyvällä onnistumisprosentilla. Aluksi epävarmalta tuntunut konsepti aiemman kokemuksen puutteessa onnistui lähtökohtiinsa nähden erittäin hyvin.

Asiasanat

konenäkö, teollisuusrobotti, automaatio, hahmontunnistus, Raspberry Pi, OpenCV, Python

South-Eastern Finland University of Applied Sciences

Author (authors)	Degree	Time
Ville Seppänen	Bachelor of Engineering	April 2020

Thesis title

Controlling an industrial robot with a machine vision application

73 pages
16 pages of appendices

Commissioned by

South-Eastern Finland University of Applied Sciences

Supervisor

Teemu Manninen

Abstract

The objective of this thesis was to attach a machine vision camera to an industrial robot manufactured by ABB. The idea is to take a photograph of a working area, analyse the picture and send coordinates of a suitable object for the robot in a form that the robot can understand. Robot then picks up the object and delivers it out of the working area in a position that is based on its colour.

This thesis was made using ABB IRB 120 -industrial robot, Python-programming language, OpenCV-machine vision library and Raspberry Pi -computer with an attached camera module. The results of this thesis are planned to be included in the upcoming automation-courses.

The main part of the thesis consists of three parts. The first was to set up Raspberry Pi and create a Python-script able to find the contours of objects using OpenCV-library. Part two was to create a way to communicate between Raspberry Pi and the robot. The last part was to attach all these parts together so that they would form a working machine vision -guided robot unit.

The requirements set for the robot are well met. The robot would pick up objects with a high success rate.

Keywords

machine vision, automation, industrial robot, pattern recognition, Raspberry Pi, OpenCV, Python

SISÄLLYS

1 JOHDANTO

Tämän opinnäytetyön aiheena on liittää Kaakkois-Suomen ammattikorkeakoulun Mikkelin kampuksella sijaitseva ABB:n valmistama teollisuusrobotti Raspberry Pi -tietokoneella ohjelmoitavaan kameraan siten, että kamera antaa ottamiensa valokuvien analysoinnin jälkeen robotille ohjeita ja robotti toteuttaa annetut käskyt. Kamera tunnistaa työskentelyalueelta eriväriset kappaleet, jotka voidaan sitten robotin toimesta lajitella halutulla tavalla. Vastaavia laitteistoja käytetään teollisuudessa esimerkiksi laaduntarkkailussa, lajittelussa tai mittaustehtävissä. Toimiessaan kyseistä laitteistoa voitaisiin käyttää jatkossa kurssimateriaalina automaatiotekniikan kursseilla havainnollistamaan konenäkölaitteiden toimintaperiaatteita, yhteyskäytäntöjä ja tekniikkaa.

Konenäkökameroita ja itse robotti löytyi koululta jo valmiiksi tähän työhön ryhdyttäessä, mutta vastaavia projekteja niillä ei ollut aiemmin tehty. Tämän vuoksi koin työn aiheen erittäin mielenkiintoisena, koska nyt sain mahdollisuuden tutustua syvällisesti kyseiseen aiheeseen sekä tehdä työn, josta olisi jatkossa hyötyä myös muille opiskelijoille. Robotin käytettävissä oleva työskentelyalue on melko pieni – noin A4-paperia vastaava alue - tämän vuoksi työtä lähdettiin tekemään siitä näkökulmasta, mitä työskentelyalueen puitteissa on mahdollista tehdä. Työn aloitusvaiheessa oli vielä jossain määrin epäselvää siitä, minkälaisia tehtäviä robotin haluttaisiin suorittavan. Sainkin sen vuoksi mahdollisuuden määrittää itse mihin suuntaan aion työtä viedä – toisin sanoen omissa käsissä oli siis päättää, tehdäänkö robotilla lajittelua, laaduntarkkailua vai muita tehtäviä.

Ennen työn aloittamista omat kokemukset teollisuusroboteista ja konenäkökameroista olivat käytännössä olemattomat. Kumpaakaan aihetta ei ollut aiemmilla kursseilla käsitelty, joten käytännössä kaikki tässä työssä käytettävät laitteet ja ohjelmistot ja niiden käyttö oli opeteltava alusta asti itse. Tämän lisäksi konenäkösovelluksen tuottaminen vaatii ohjelmointitaitoja, joita ei ennen työn aloitusta itselläni ollut. En kokenut tätä kuitenkaan ongelmana, koska työ vaikutti erittäin mielenkiintoiselta ja uskon hyötyväni opituista asioista myös jatkossa. Työ oli luonteeltaan hyvin käytännönläheistä, koska suurin osa työskentelyajasta kului laitteiden eri toimintojen kanssa puuhasteluun ja oman työn

tuloksen pystyi helposti toteamaan toimivasta laitteesta tai koodista. Laitteiston testaaminen sitä mukaa, kun uusia ominaisuuksia ja toimintoja saatiin tuotettua, auttoi motivoimaan työnteon aikana.

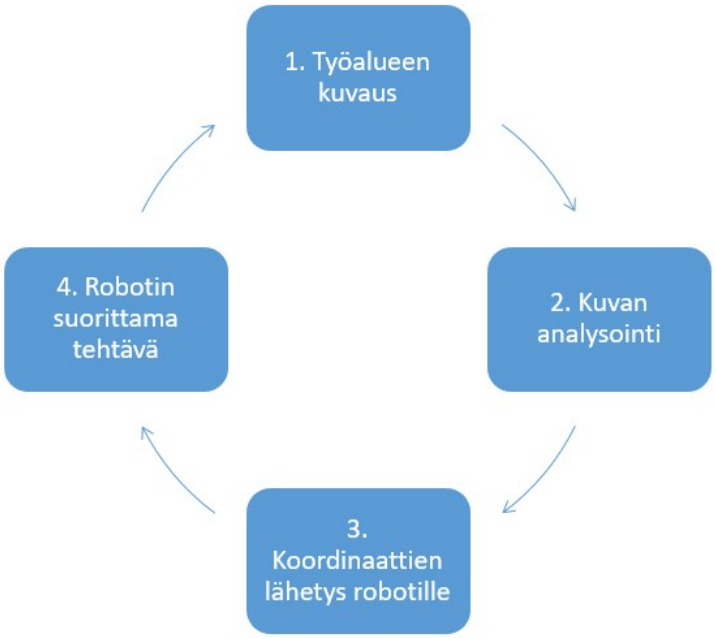
2 TYÖN TAVOITTEET

Opinnäytetyön perimmäisenä tavoitteena oli perehtyä teollisuusrobottien eri toimintoihin, konenäkökameroihin, Raspberry Pi -kehitysympäristöön sekä niiden kanssa käytettäviin ohjelmistoihin ja ohjelmointiin yleensä. Koska työtä ei tehty suoranaisesti millekään yritykselle mihinkään tiettyyn tarkasti rajattuun tehtävään, oli aloitusvaiheessa mahdollista vaikuttaa itse siihen, millainen lopullisesta työstä tulisi. Tähän sisältyy sekä hyviä että huonoja puolia. Robottisolu löytyy valmiiksi koululta, täysin toimintakuntoisena. Robottiin ei itsessään siis tarvinnut tehdä muutoksia, mikä säästi aikaa. Raspberry Pi -tietokoneiden (myöhemmin myös RasPi) aiempia versioita olisi koululta löytynyt, mutta työssä päädyttiin hankkimaan laitteen uusin malli sekä siihen liitettävä kameramoduuli.

Työn suorittaminen vaati myös ohjelmointitaitoja. Niiden puuttuminen työhön lähdettäessä pakotti opiskelemaan myös kyseisiä taitoja. Melko nopeasti ohjelmointitarpeen havaitsemisen jälkeen saatiin kuitenkin selville tarvittava ohjelmointikieli ja siihen liitettävät kirjastot, joista kirjoitettu myöhemmissä osioissa lisää. Ohjelmointitarve koettiin erittäin hyväksi asiaksi jo työhön lähdettäessä, koska sen nähtiin olevan tarpeellinen taito myös tulevissakin projekteissa ja opinnäytetyön myötä olisi hyvät valmiudet jatkaa uusien mielenkiintoisten konenäkö- ja ohjelmointiprojektien parissa.

Ongelmana aluksi oli valmiin, selkeän päämäärän puuttuminen. Kuten aiemmin on mainittu, työtä ei tehty millekään yritykselle vaan enemmänkin ns. demolaitteistoksi, jolloin työn alussa ei tarvinnut määritellä täysin tarkkaa suunnitelmaa lopullisen laitteiston toiminnoista. Olisi selvää, että työn aikana tulisi ilmenemään sellaisia ongelmia ja haasteita, joita olisi vaikeaa tai jopa mahdotonta ennustaa etenkin kokemuksen puuttuessa aiemmista vastaavista töistä. Alussa määritettyä päämäärää voisi joutua työn edetessä muuttamaan ja pohtimaan tarkemmin, mitä ominaisuuksia laitteeseen saataisiin lisättyä. Työn sel-

kärankana pidettiin lähtökohtaisesti kuitenkin sitä, että kameran ohjastama robotti noutaa halutun kappaleen työalueelta. Kaikki muu olisi enemmän tai vähemmän ylimääräistä. Selvää oli, että ilman huolellista suunnittelua voisi edessä olla ongelmia. Alustava karkea työjärjestys ja ohjelmakierto on esitetty kuvassa 1.



Kuva 1. Suunnitelma tarvittavista työvaiheista

Tällä suunnitelmalla työtä alettaisiin toteuttamaan. Tiukkaa aikarajaa toteutukselle ei ollut, joten mahdollisista viivytyksistä ja isommista ongelmatilanteista ei nähty olevan suurta haittaa. Arvio työn valmistumiselle oli noin kaksi tai kolme kuukautta, joskin aiheiden ollessa täysin uusia voisi työn suorittamiseen kulua pidempikin aika – tästä ei työtä aloittaessa ollut kuitenkaan suurempaa huolta. Tärkeintä olisi saada valmis laitteisto aikaiseksi siihen käytetystä ajasta huolimatta.

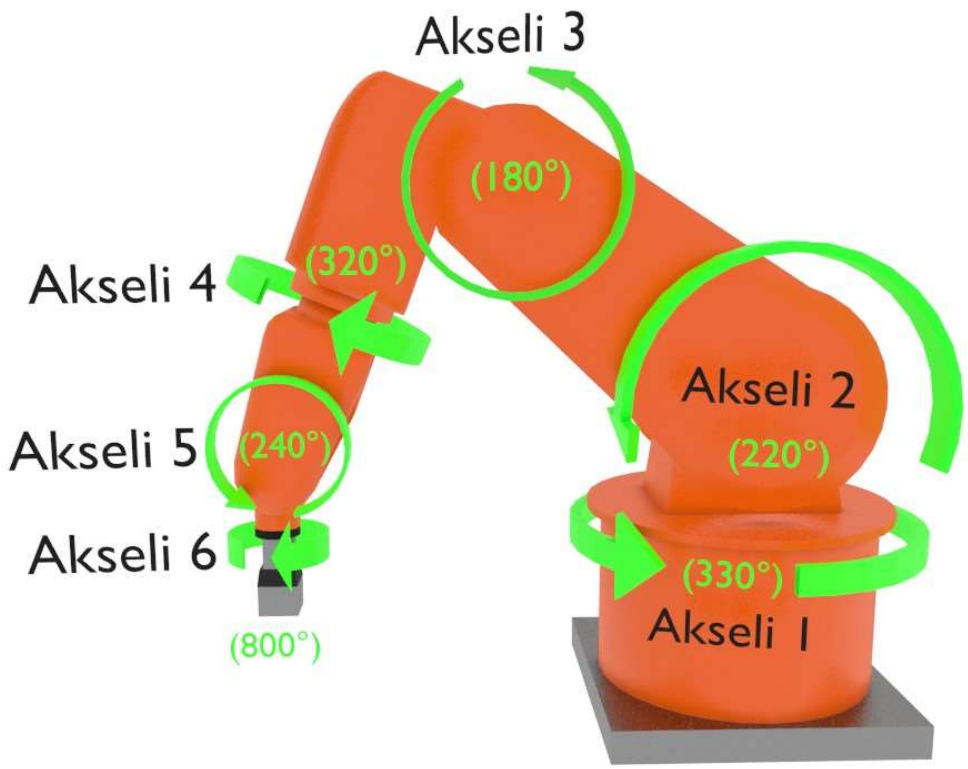
3 KÄYTETTÄVÄT LAITTEET JA OHJELMISTOT

Työn aikana maailmalla vallinnut koronavirusepidemia rajoitti jossain määrin työn suorittamista, etenkin sen loppuvaiheilla. Tämän vuoksi opinnäytetyön teoriaosuudessa on laitteistosta otettujen valokuvien sijaan käytetty runsaasti 3d-mallinnuksella tuotettuja kuvia havainnollistamaan tarpeellisia tekstissä ilmeneviä asioita. Mallinnuksessa robotin mitat eivät ole täysin mittakaavassa

ja mallin karkeuden vuoksi joitain eroja todelliseen robottiin löytyy. Kuvat luotiin Blender v2.82 -sovelluksella.

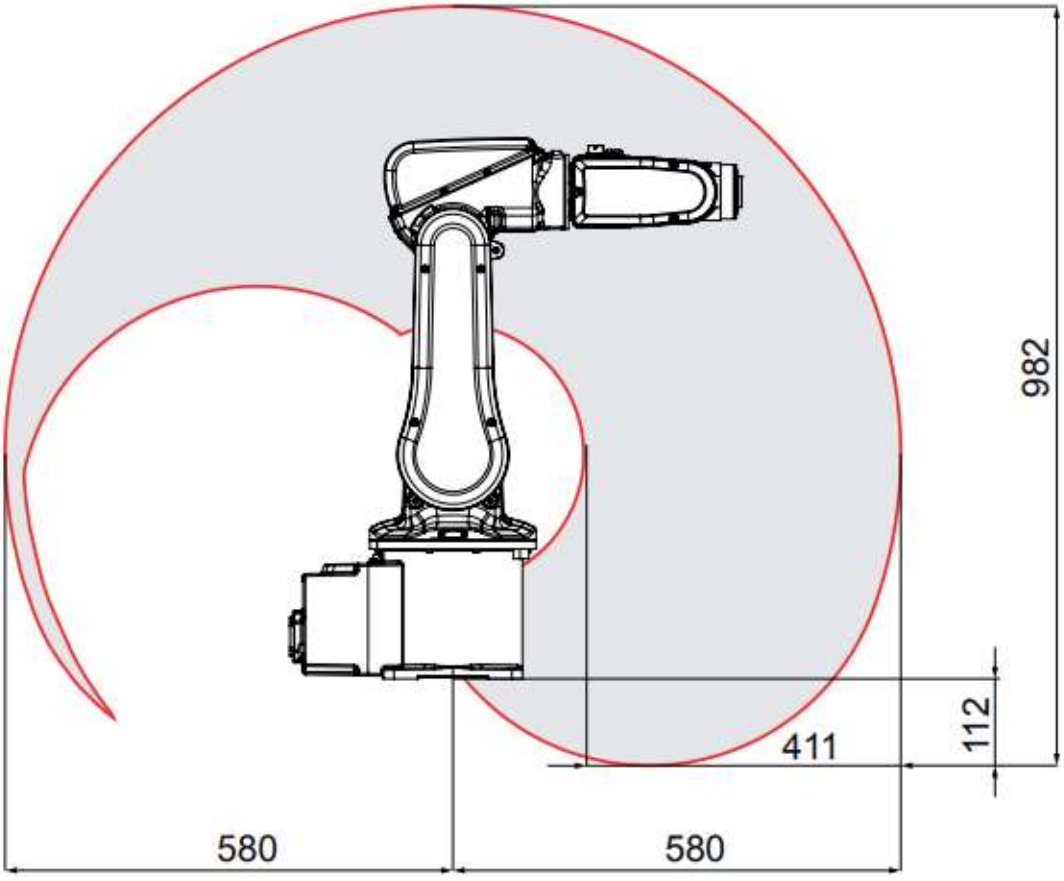
3.1 ABB IRB 120 -teollisuusrobotti

IRB 120 -teollisuusrobotti on yksi pienimmistä ABB:n valmistamista tuotantokäyttöön suunnitelluista roboteista. Pienen kokonsa vuoksi sen yleisimpiä käyttökohteita ovat elektroniikan kokoonpano- ja tarkastustehtävät, ruoka- ja juomateollisuus, koneistus, lääketieteelliset ja yleiset tieteelliseen tutkimukseen liittyvät tehtävät. Pienen kokonsa vuoksi se soveltuu todella hyvin myös oppilaitoskäyttöön harjoituslaitteeksi. Painoa sillä on 25 kilogrammaa, jolloin robotin liikuttelu ja esimerkiksi sijainnin muutos sille rakennetulla pöytäalustalla on helppoa. Maksimi nostokuorma on kolme kilogrammaa ja työkalun ulottuma on maksimissaan 58 senttimetriä. Robotin varsi koostuu kuudesta akselinsa ympäri pyörivästä nivelestä, joiden sijainnit ja suunnat on havainnollistettu kuvassa 2. Tällaista robottimallia kutsutaankin yleisesti nimellä nivelrobotti. [1.]



Kuva 2. Robotin akselien sijainnit ja liikesuunnat havainnekuvassa

Näiden akselien ympäri tapahtuva liike mahdollistaa robotin liikkumisen vapaasti lähes mihin tahansa asentoon ulottuvuutensa puitteissa. Tiettyjä katvealueita syntyy myös, kun robotin kärjellä lähestytään sen ensimmäistä niveltä (akseli 1). Vastaava katvealue jää myös robotin taakse. Laitteesta löytyy hyvät suojausmekanismit, jolloin liian lähelle ajettaessa robotti pysähtyy ja antaa virheviestin, koska sen ei ole mahdollista ajaa itsensä ”läpi”, tai saavuttaa pisteitä, joihin se ei akseleidensa liikesuuntien puitteissa yllä. Nämä katvealueet käyvät esille kuvasta 3. Pystyakselillaan laite kääntyy nollalinjansa ympäri 165 astetta suuntaansa, eli yhteensä 330 astetta. Virtansa laite ottaa normaalista seinäpistorasiasta ja sen tarvitsema teho on noin 240 wattia. [1.]



Kuva 3. Robotin ulottuvuudet

Robotin läpi kulkee paineilmaletkut, jolla paineilmaa voidaan johtaa laitteen työkalulle, mikäli sille on tarvetta. Tässä työssä käytettävään robottiin on asennettu tarttujatyyppinen työkalu, jonka leukoihin johdettavalla paineilmalla sen leuat sulkeutuvat ja täten haluttujen kappaleiden siirtäminen on mahdollista. Robotin työkalun leukojen liike niiden avautuessa ja sulkeutuessa on noin

puoli senttimetriä, eli kohtalaisen lyhyt matka. Sopivankokoisen kappaleen käsittelyyn tämä on kuitenkin täysin riittävästi.

Kaakkois-Suomen ammattikorkeakoululle kyseinen ABB IRB 120 -robotti on hankittu vuonna 2014. Tämän jälkeen sitä ei juurikaan ole käytetty osana automaatiotekniikan kursseja. Kyseisen laitteen hankinnasta ja sen käyttöönotosta on kirjoitettu opinnäytetyö silloiselle Mikkelin ammattikorkeakoululle nimellä ”Teollisuusrobotin käyttöönotto ja ohjelmointi”, tekijänään Tommi Tuunanen. Työssä käydään kattavasti läpi käytettävän robotin toimintoja ja ominaisuuksia. [2.] Tässä työssä samoja aiheita sivutaan ymmärrettävästikin runsaasti, joskin painopiste on enemmän konenäkölaitteistossa ja sen suunnittelussa.

Robottia voidaan ohjata käyttäen sen mukana tulevaa FlexPendant-ohjainta. Kädessä pideltävä ohjain mahdollistaa robotin manuaalisen ohjauksen, pisteiden opettamisen, ohjelmien ajon ja niiden muokkauksen ynnä muita toimintoja. Robottia voidaan siis ohjata ilman erillistä tietokonetta, joskin tietokonetta ja RobotStudio-sovellusta apuna käyttäen on erilaisten ohjelmien luominen tilanteen mukaan monin verroin nopeampaa. FlexPendant-ohjain esitetty kuvassa 4. [3, s. 54.]

Kuva 4. FlexPendant-ohjain

FlexPendant puolestaan on kytketty IRC5-kontrolleriin, joka hoitaa robotin akselien liikuttamisen ja kaikki siihen liittyvät tarvittavat toiminnot. Kontrolleri sisältää kaksi osaa, *Control Modulen* ja *Drive Modulen*. *Control Module* sisältää

kaiken ohjaukseen liittyvän elektroniikan sisältäen ohjaustietokoneen, I/O-moduulit ja flash-muistin. Siihen myös sisältyy robotin käyttämiseen vaadittava ohjelmisto eli RobotWare-järjestelmä. *Drive Module* sen sijaan huolehtii robotin moottoreiden vaatimasta elektroniikasta. Yhdellä *Drive Modulella* voidaan käsitellä kuutta akselia ja robotin mallin mukaan jopa kolmea lisäakselia. Kontrollerista löytyvät kaikki laitteen tarvitsemat liitännät, esimerkiksi verkkoliitäntä, jolla laitteeseen on kytketty reititin langatonta verkkoyhteyttä varten. [3, s. 61.]

3.2 Raspberry Pi

Raspberry Pi on yhdelle piirilevylle rakennettu, noin luottokortin kokoinen tietokone, jota valmistaa isobritannialainen *Raspberry Pi Foundation*. Hyväntekeväisyyteen perustuva järjestö loi RasPin opetuskäyttöön helpottamaan tietojenkäsittelytieteiden (computer science) ja siihen liittyvien aiheiden opiskelua ja tutkimusta, erityisesti jo eri oppilaitosasteilla. [4.] Kuvassa 5 kuvattuna työn kirjoittamisen aikaan RasPin uusin versio *Raspberry Pi 4 Model B*.

Kuva 5. Raspberry Pi 4 Model B

Ensimmäinen versio ilmestyi markkinoille vuoden 2012 helmikuussa, jolloin *Raspberry Pi Model B* tuli myyntiin. Se sisälsi 700 megahertsin suorittimen, *VideoCore IV* -grafiikkayksikön, joka kykeni toistamaan FullHD-laatuista videokuvaa 30 kuvan sekuntinopeudella ja 512 megatavua keskusmuistia. Varsinainen työmuisti (vrt. pöytätietokoneen kovalevy) lisättiin SD-muistikortilla. Lisäksi alkuperäiseen laitteeseen sisältyi kaksi USB 2.0-liitäntää, Ethernet-portti

ja 3,5 millimetrin audioliitäntä. Tässä työssä on käytetty RasPin uusinta versiota, mallinimeltään *Raspberry Pi 4 Model B*. Kuten tietotekniikalle on luonteista, on vuonna 2012 julkistetun ja uusimman version välillä laskentateho luonnollisesti kasvanut merkittävästi ja uusin malli onkin jo varsin tehokas tietokone. Käytetystä mallista löytyy 1,5 gigahertsin neliydinprosessori, *VideoCore VI* -grafiikkayksikkö ja neljä gigatavua keskusmuistia. Käyttömuistia varten laitteesta löytyy microSD-portti. Liitännät käsittävät kaksi kappaletta USB 3.0 -liitäntöjä, kaksi kappaletta USB 2.0 -liitäntöjä, Ethernet-portin ja 3,5 millimetrin audioliitännän. Lisäksi ensimmäisestä versiosta asti mukana ollut liitäntä RasPin omalle kameramoduulille sisältyy. Muut tärkeimmät muutokset löytyvät laitteen virransyötöstä ja näyttöliitännästä. [5.]

Siinä missä alkuperäisessä versiossa – kuten kaikissa versioissa uusinta lukuun ottamatta – on virransyöttö toteutettu microUSB tai GPIO-liitännöillä. Uusimman version kasvanutta virrantarvetta varten on virransyöttö vaihdettu USB-C -liitäntään. Näyttöliitäntä on sen sijaan vaihtunut normaalista HDMI-liitännästä kahteen micro-HDMI-liitäntään, jolloin yhteen laitteeseen voidaan kytkeä myöskin kaksi näyttöä. Muista uudemmista ominaisuuksista tärkein on ehdottomasti Wi-Fi eli langattomien verkkoyhteyksien suora tuki. Langattomuus oli tässä työssä muutoinkin avainasemassa, koska robotin ja RasPin oli muutoinkin tarkoitus toimia samassa langattomassa verkossa. Uusimman ja vanhimman kehitysversion väliin mahtuu ainakin yhdeksän muuta mallia. Muutokset mallien väliltä löytyy lähinnä kasvaneesta tehosta, mutta joukosta löytyy myös huomattavasti pienempikokoinen *Raspberry Pi Zero* ja sen kehittyneemmät versiot. Siinä missä ”perusmallien” koot ovat noin 86 x 56 x 13 millimetriä (leveys, pituus, korkeus), on Zero-malliston vastaavat koot 65 x 30 x 5-10 millimetriä. [5.]

Raspberry Pin roolia nimenomaan kehitysympäristönä tukee siitä löytyvät 40 kappaletta GPIO-pinnejä (general-purpose-input/output). Jokainen pinni voidaan määritellä RasPissa ohjelmallisesti joko sisään- tai ulostuloksi. Näin laitteeseen voidaan helposti lisätä esimerkiksi erilaisia antureita tai vaikkapa ledvaloja. Pinnejä voidaan hyödyntää myös mm. pulssinleveysmodulaatioon. [6.] Suuri liitäntöjen määrä muiden hyödyllisten ominaisuuksien lisäksi mahdollistaa RasPin erittäin monipuolisen käytön niin harrastelu- ja kouluprojekteihin, IoT:hen, kuin myös vaikkapa kattaviin kotiautomaatiojärjestelmiin tai internet-

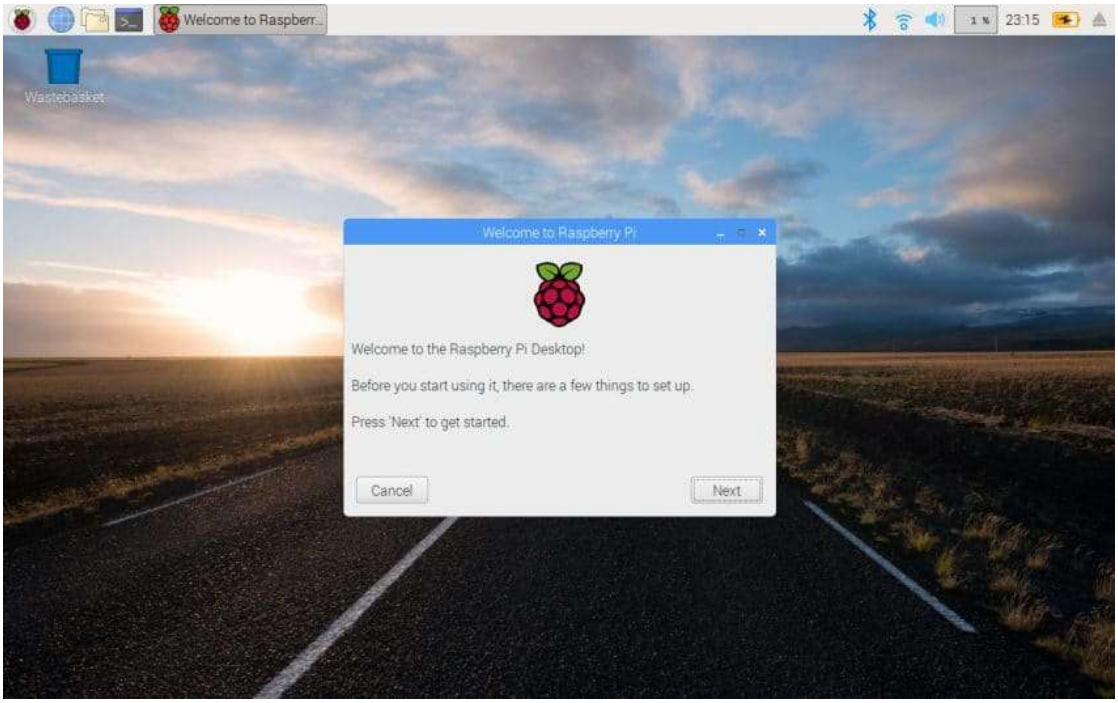
palvelimiin. Tätä tukee myös laitteen huokea hinta. *Raspberry Pi 4 Model B* -mallin neljän gigatavun keskusmuistilla varustettu versio maksoi kirjoitushetkellä noin 70 euroa, jolloin laitteen käytön aloituskustannukset - etenkin verrattuna tavalliseen pöytätietokoneeseen tai kannettavaan laitteeseen - ovat erittäin kilpailukykyiset. Asiaa puoltaa myös etenkin uusimman version laskentakapasiteetin riittävyys normaalissa työpöytäkäytössä, jolloin laitteesta voidaan helposti muokata esimerkiksi mediatoistin, jolla vaikkapa vanhemmastakin ns. ”mökkitelkkarista” saadaan vaivatta luotua äly-tv. Pienen kokonsa vuoksi laitteen asennus hankaliinkin paikkoihin on mahdollista, joka entisestään kasvattaa sen käyttömahdollisuuksia – tämä projekti on siitä hyvä esimerkki.

3.2.1 Raspbian-käyttöjärjestelmä

Raspberry Pille on saatavilla useita eri käyttöjärjestelmiä. Aiemmin mainittu mediatoistin-vaihtoehto voidaan toteuttaa esimerkiksi OSMC-käyttöjärjestelmällä (*Open Source Media Center*), joka on kehitetty varta vasten eri mediatiedostojen toistoon [7]. Linux-pohjaiset käyttöjärjestelmät ovatkin ylivoimaisesti suosituimpia vaihtoehtoja Raspberry Pin kaltaisille laitteille, joskin muitakin vaihtoehtoja tarvittaessa löytyy. Linux-käyttöjärjestelmät ovat pääosin avoimeen lähdekoodiin perustuvia, ja usein ne ovatkin aiemman OSMC-esimerkin tukemana suunniteltu johonkin tarkasti määrättyyn tehtävään, eli ylimääräisiä ominaisuuksia on karsittu runsaasti pois. Näin voidaan saavuttaa kevyempiä käyttöjärjestelmiä, joita voidaan hyödyntää laitteissa, jossa laskentatehoa on hyvin rajallisesti tai suuremmalle määrälle toimintoja ei yksinkertaisesti ole tarvetta.

Linux pohjautuu jo vuonna 1969 alkunsa saaneeseen UNIX-käyttöjärjestelmään [8]. Linuxin pohjalta taasen on kehitetty useita uusia käyttöjärjestelmiä – tunnetuimpia näistä lienevät Ubuntu ja Debian [9, s. 17]. Tässä opinnäytetyössä käytettiin Debian-pohjaista Raspbian-käyttöjärjestelmää, joka on kehitetty nimenomaan Raspberry Pitä silmällä pitäen ja jonka käyttöä *Raspberry Pi Foundation* suosittelee. Raspbian tarjoaa graafisen käyttöliittymän, sekä kattavan kokoelman eri sovelluksia valmiiksi asennettuna. Näistä peruskäyttäjälle tärkein lienee *Google Chrome* -internet-selaimeen pohjautuva Chromium-selain, jota hyödynnettiin satunnaisesti myös tässä työssä. Tärkeimmässä roolissa oli kuitenkin *Thonny Python IDE* -sovellus, jolla Python-ohjelmointi

pääasiassa suoritettiin ja jolla valmista ohjelmakoodia ajettiin. Käytössä oli myös vakiona käyttöjärjestelmän mukana asennettu Python-ohjelmointikielen versio 3.7.3. Kolmas käytettävä sovellus oli *VNC Server* -sovellus ja sen parina pöytätietokoneelle asennettu *VNC Viewer*, jolla RasPin työpöytänäkymä voitiin langattoman verkkoyhteyden yli lähettää pöytäkoneelle. Tällöin RasPi voitiin kiinnittää robottiin kiinni, jolloin erilliselle näyttökaapelille ei ollut tarvetta. Raspbian-käyttöjärjestelmän työpöytänäkymä on esitetty kuvassa 6. Käytettyjä sovelluksia on tarkemmin käsitelty kappaleessa 6.2.



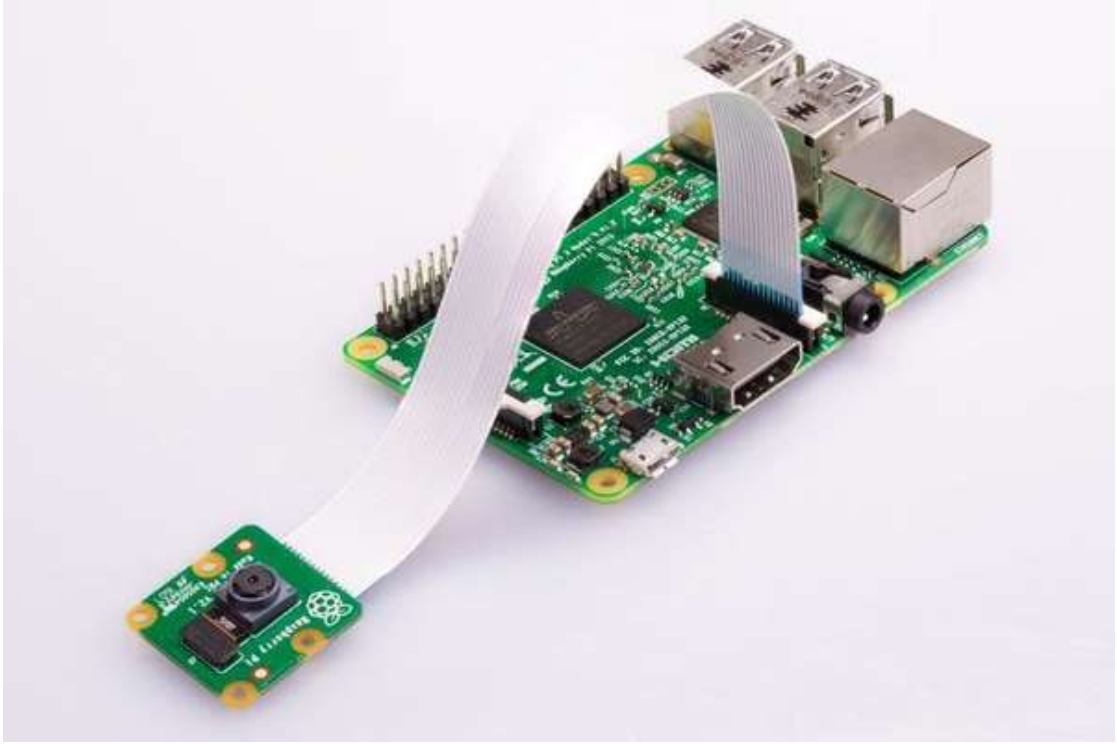
Kuva 6. Raspbian-käyttöjärjestelmän työpöytänäkymä

Kuten edellä olevasta kuvasta voidaan todeta, on kyseisen käyttöjärjestelmän ulkoasu hyvin samantyyppinen verrattuna useimpien tuntemaan Windows-käyttöjärjestelmään. Oletuksena ylhäällä olevan tehtäväpalkin sijainnin voi muuttaa halutessaan alas, jolloin ero nykyisiin Windows-versioihin kapenee entisestään. Huomattavin ero Windowsin ja Raspbianin kaltaisten järjestelmien välillä lienee tapa, jolla uusia sovelluksia asennetaan. Siinä missä Windowsilla asennus tapahtuu esimerkiksi asennustiedoston internetistä lataamalla ja sen käynnistämällä, asennetaan Raspbianilla sovellukset ja niiden päivitykset komentorivin kautta. Esimerkiksi Pythonin asennus tapahtuisi yksinkertaisimmillaan komentoriville kirjoitetulle komennolla ”sudo apt-get install python3”. Komentorivin käyttö saattaa ainakin aluksi tuottaa päänvaivaa,

mutta internet tarjoaa lähes rajattomasti erilaisia oppaita ja apua eri sovellusten ja pakettien asennukseen ja RasPin toimintoihin yleensäkin.

3.2.2 Raspberry Pi Camera Module V2

Konenäkösovellusta kehitettäessä yksi tärkeimpiä yksittäisiä elementtejä on käytettävä kamera. RasPia käytettäessä luonnollinen valinta on laitteen kameraliitäntään tuleva kameramoduuli. *Camera Module V2* on vuonna 2016 myyntiin ilmestynyt, alkuperäisen moduulin korvannut kamera. Se sisältää Sonyn kahdeksan megapikselin IMX219-mallimerkinnältään olevan sensorin, kyeten parhaillaan FullHD-laatuiseen videon kuvaamiseen 30 ruutua sekunnissa. Aiemmassa versiossa megapikseleitä oli viisi, joten kameran valokuvaukseen käytettävän maksimiresoluution voidaan sanoa kasvaneen suhteellisen maltillisesti. Tämä käykin ilmi laitteen teknisistä tiedoista, jotka ilmoittavat alkuperäisen kameramoduulin sensorin maksimiresoluutioksi 2592 x 1944 pikseliä, uuden ollessa suurimmillaan 3280 x 2464 pikseliä. Resoluutio kertoo kuvan mitat pikseleinä alkaen vaaka-akselista, jonka jälkeen on esitetty pystyresoluutio. Megapikselit saadaan kertomalla luvut keskenään, esimerkkinä uuden kameramoduulin megapikselien määrä on 3280 x 2464 = 8 081 920 pikseliä, eli noin kahdeksan megapikseliä. Pienen kokonsa vuoksi laite on helppo asentaa lähes millaiseen käyttöön vain. Ainut tässä työssä ilmennyt ongelma oli kameran mukana toimitettavan liitäntäkaapelin lyhyt pituus. Tähän aiheeseen palataan kuitenkin uudestaan kappaleessa 6.5. Kameramoduulin pieni koko käy hyvin ilmi kuvasta 7. [10.]



Kuva 7. Raspberry Pi Camera Module V2 kiinnitettynä RasPiin

Kameran kiinteä polttoväli on 3,04 millimetriä, jolla saavutetaan horisontaalisesti 62,2 asteen näkökenttä. Tämä niin sanotussa kinovastaavuudessa (esim. täyden kennon järjestelmäkamera 36 x 24 millimetrin kennolla) vastaa noin 30 millimetrin polttovälillä olevaa objektiivia (61,93 astetta). RasPin kameramoduulin aukkoarvo on kiinteä f/2,0. [10.] Suurehko aukko helpottaa kuvaamista huonommassa valaistuksessa, joskin moduulin erittäin pienen fyysisen koon vuoksi kameran muutamien millimetrien kanttiinsa olevalle kennolle osuvien fotonien määrä kuvaa otettaessa on suljinajasta riippumatta auttamatta varsin vähäinen – tämä aiheuttaa kuvaan vääjäämättä kohinaa heti, kun valon määrä kuvattavassa kohteessa laskee. Tämän työn kannalta kohinalla ei ollut juurikaan käytännön merkitystä, mutta moduulin käyttöä esimerkiksi valvontakamerana voi joissain tapauksessa rajoittaa kennon pieni koko ja siitä aiheutuva kohina.

3.3 Python-ohjelmointikieli

Python-ohjelmointikielen ensimmäisen version kehitys sai alkunsa 1980-luvun lopulla, kun hollantilainen Guido van Rossum alkoi harrasteluprojektinaan suunnittelemaan uutta ohjelmointikieltä aiemmin kehitetyn ABC-kielen rinnalle. Python-nimen kyseinen kieli on saanut brittiläisestä Monty Pythonin lentävä sirkus -komediasarjasta. Vuonna 1991 ilmestynyt versio 0.9.0 sisälsi jo muun

muassa tänäkin päivänä mukana olevat lista-, sanakirja- ja teksti-datatyypit. Versio 2.0 ilmestyi lokakuussa 2000, ja tässä työssä käytetty kolmosversio julkaistiin vuonna 2008. [11.] Käytännössä tämän työn toteuttamiseen käytettiin Pythonin versiota 3.7.3, mutta merkitsevää on lähinnä versionumeron kokonaisosa. Versio kolme ei ole taaksepäin yhteensopiva version kaksi kanssa - tämä voikin toisinaan aiheuttaa päänvaivaa, kun koodia haetaan eri lähteistä. Versiolla kaksi kirjoitetut ohjelmat eivät siis lähtökohtaisesti toimi versiossa kolme.

Nykyään Python on TIOBE-indeksin mukaan maailman kolmanneksi käytetyin ohjelmointikieli noin kymmenen prosentin osuudella. Edellä menevät vain Java ja C-kieli. TIOBE-indeksi seuraa eri ohjelmointikielien nimillä tehtyjen hakujen määrää internetin suurimmilla hakukoneilla. [12.] Suosion myötä aloittelevan ohjelmoijan onkin erittäin helppo lähteä opettelemaan Pythonia sen massiivisen käyttäjäkunnan luomien oppaiden ja artikkeleiden vuoksi. Kielen rakenne on tehty mahdollisimman helpoksi lukea ja kirjoittaa, tästä esimerkkinä sen käyttämä tyyli ryhmitellä kirjoitettuja lauseita sisennyksien avulla. Tällöin vältytään esimerkiksi monien muiden ohjelmointikielien usein käyttämiltä, aloittelijalle usein hankalasti ymmärrettäviltä runsaalta sulkeiden ja kaksoispisteiden käytöltä. Yksinkertainen esimerkki sisennyksen käytöstä on esitetty kuvassa 8.

```
1  a = 15
2  b = 10
3  if a > b:
4      print('A on suurempi kuin B')
5  else:
6      print('B on suurempi kuin A')
```

Kuva 8. Yksinkertainen esimerkki Python-koodista

Ohjelma siis suorittaa yksinkertaisen testin vertailemalla luotuja muuttujia ”a” ja ”b” keskenään ja tulostaa sitten vastauksen riippuen vertailun tuloksesta. Kuten kuvasta nähdään, on Pythonilla toteutettu koodi etenkin perustoiminnoissaan hyvin suoraviivaista ja helppolukuista, mikä on kyseisen kielen kohdalla ollut tarkoituskin. Tätä tukee myös Pythonin käyttämä niin sanottu ”dynaaminen tyypitettävyys” [14]. Ohjelman alussa luodulle muuttujalle ei tarvitse

erikseen määritellä sen tyyppiä, vaan tyyppi määritellään muuttujan sisällön perusteella. Samaan muuttujaan voidaan aluksi tallentaa esimerkiksi desimaaliluku, joka ohjelman myöhemmässä vaiheessa muuttuukin vaikkapa string-muotoiseksi tekstimuuttujaksi. Tätä periaatetta on havainnollistettu kuvassa 9.

```
1   oma_muuttuja = 1
2   print(oma_muuttuja)
3
4   oma_muuttuja = 'numero yksi'
5   print(oma_muuttuja)
6
7   oma_muuttuja = 1.0
8   print(oma_muuttuja)
9
10  oma_muuttuja = [1,2,3,4,5]
11  print(oma_muuttuja)
```

Kuva 9. Esimerkki dynaamisesta tyypityksestä

Kuvan mukaisesti muuttuja ”oma_muuttuja” on ensin määritelty kokonaisluvuksi, joka sitten sellaisenaan myös tulostetaan. Seuraavaksi muuttujasta luodaan ns. string-muuttuja, eli tekstiä sisältävä muuttuja. Kolmas muutos tekee muuttujasta desimaaliluvun, eli float-tyyppisen objektin ja viimeiseksi ”oma_muuttuja” muunnetaan vielä listaobjektiksi. Kaikki tämä tapahtuu siis samassa komentosarjassa, joka ajettuaan tulostaa konsoliin sisältämänsä arvot siinä muodossa, mihin ne koodissa riviä ylempänä on muunnettu. Tällainen ominaisuus tekee ohjelman kirjoittamisesta helppoa, kun datatyypin muodosta ei tarvitse erikseen välittää. Virhetilanne syntyy vasta siinä vaiheessa, jos erimuotoisia datatyyppejä yritetään esimerkiksi laskea yhteen. Tekstimuotoista string-objektia ei voida lisätä yhteen esimerkiksi numeerisen kokonaislukuobjektin kanssa, koska se ei ole loogisesti mahdollista. Sen sijaan integer-tyyppisen kokonaisluvun ja float-muotoisen desimaaliluvun yhteen laskeminen onnistuu – tuloksena on tällöin float-objekti, eli desimaaliluku.

Python-kieltä käytetään laajasti niin web-sovellusten kehitykseen, graafisia käyttöliittymiä vaativiin sovelluksiin, tieteelliseen ja numeeriseen laskentaan sisältäen muun muassa koneoppimisprojektit, pelinkehitykseen kuin myös tässä opinnäytetyössä luotavaan konenäkösovellukseen [13]. Käyttötapoja on siis

runsaasti, mikä onkin yksi syy Pythonin suureen suosioon. Erilaisia ohjelmointiin käytettäviä, käyttötarpeen mukaan ohjelmaan lisättäviä kirjastoja on saatavilla tuhansia – esimerkkinä tässä työssä käytettävä OpenCV-konenäkökirjasto. Halutut kirjastot voidaan niiden asennuksen jälkeen lisätä ohjelmakoodiin helposti komennolla ”import kirjaston_nimi”.

3.3.1 OpenCV-konenäkökirjasto

OpenCV-konenäkökirjasto (*Open source computer vision*) on konenäköprojekteja varten luotu kattava, eri ohjelmointifunktioita sisältävä paketti. Alun perin Intelin luoma kirjasto on ollut suuren yleisön saatavilla jo vuodesta 2000, jolloin ensimmäinen alpha-versio julkaistiin. Kirjasto sisältää yli 2500 algoritmia, sisältäen myös koneoppimiseen liittyviä toimintoja. Konenäköön liittyvät funktiot liittyvät kasvojen ja kappaleiden tunnistukseen, kameran ja siinä näkyvien objektien liikkeiden seurantaan, stereokuvasta luotavaan 3d-pistepilveen ja moniin muihin konenäkölaitteistoissa tarvittaviin ominaisuuksiin. Kirjasto on alun perin kirjoitettu C++-ohjelmointikielelle, mutta sittemmin siitä on käännetty omat versionsa myös muun muassa Javalle, Pythonille ja MATLAB:lle. Myös käyttöjärjestelmätuki on kattava sisältäen Windowsin, Linuxin ja macOS:n lisäksi myös monia muita – myös mobiilikäyttöön tarkoitettuja – käyttöjärjestelmiä. Hyvä käyttöjärjestelmätuki mahdollisti tämän työn tekemisen käyttäen Raspberry Pin Raspbian-käyttöjärjestelmää ja OpenCV:n 3.4.6-versiota. [15.] OpenCV-kirjastoa on hyödynnetty myös tieteellisessä tutkimuksessa Raspberry Pin kanssa esimerkiksi alkuvuodesta 2020 ilmestyneessä, maanpinnan valumaa maanvyöryalueella tutkineessa tutkimusartikkelissa [16].

OpenCV-kirjastolla omien konenäkösovellusten luominen ja testaaminen on helppoa selkeiden komentojen, kuin myös kattavan dokumentaation ja internet-oppaidensa vuoksi. Äkkiseltään hankalalta kuulostava aihe selkenee nopeasti, jos vain malttaa aluksi perehtyä opiskelemaan sovellettavia periaatteita ja tekniikoita esimerkiksi tässä työssä käytettävien ääriviivojen tunnistuksen osalta. Yksinkertaisimmillaan ääriviivat voidaan tunnistaa käyttämällä findContours-komentoa, jonka vaatimat parametrit löytyvät saatavilla olevasta dokumentaatiosta.

Käytännön sovelluksia OpenCV:llä toteutetuille konenäkölaitteistoille voivat

olla esimerkiksi Suomen poliisilla jo vuodesta 2014 eteenpäin käytössä olleet REVIKA-laitteet (rekisterikilpien lukulaite, video ja kamera), jotka poliisiautoon sijoitettuna automaattisesti skannaavat ohiajavien autojen rekisterikilpiä suorittaen niillä samalla tarkistuksia esimerkiksi auton verojen maksun ja edellisen katsastusajankohdan suhteen. Laitteiston käytöstä on kirjoitettu opinnäytetyö Poliisiammattikorkeakoululle vuonna 2018, tekijänä Oska Pyykkönen, otsikkona ”Revikan ohjaus ja käyttäminen: Oulun poliisilaitoksen valvonta- ja hälytyssektorilla ja liikennesektorilla” [17]. Vaikkei REVIKA:n käyttämästä kilpien lukutekniikasta saatukaan täyttä varmuutta, voitaisiin vastaava laitteisto kuitenkin rakentaa käyttäen OpenCV-konenäkökirjastoa ja sopivaa kameraa. Kilpien tunnistuksen lisäksi ajoneuvoihin liittyvää konenäkötekniikkaa voidaan käyttää OpenCV:llä vaikkapa niin sanotun kaistavahdin toteuttamiseen, edellä ajavan auton etäisyyden arvioimiseen, pysäköintiavustajaan ynnä muihin. Vaihtoehtoja siis riittää jo pelkästään tieliikennetarpeisiin. Sen lisäksi esimerkiksi liikennevirtojen seuranta onnistuisi automaattisesti laskemalla videokuvasta tunnistettavat ajoneuvot vaikkapa risteysalueella. Turvakamerat (liikkeentunnistus), kasvontunnistus ja monet muut toiminnot antavat mahdollisuuden luoda konenäkölaitteistoja kotiautomaatiojärjestelmän osaksi. Matalan aloituskynnyksen ja etenkin Raspberry Pin tapauksessa sen halvan hinnan vuoksi onkin ymmärrettävää, miksi konenäkö ja siitä pykälää haastavampi aihe koneoppiminen ovat jatkuvasti yleistymässä olevia tekniikoita.

3.3.2 Socket-rajapinta

Pythoniin, kuten lukuisiin muihinkin ohjelmointikieliin, on saatavilla socket-rajapinta, jolla tiedonsiirto kahden erillisen, samassa verkossa olevan laitteen kesken voidaan suorittaa, kun käytettävä IP-osoite ja portti on määritelty. Tässä työssä käytettiin rajapinnan tarjoamaa stream- eli tietovirtapohjaista pakettien lähetystapaa. Tällöin lähtevien pakettien välitys perille asti on varmaa. Jos socketin kautta lähetetään esimerkiksi paketit 1, 2 ja 3, saapuvat ne perille myöskin samassa järjestyksessä. Jos lähetys syystä tai toisesta häiriintyy, saa paketin lähettäjä tästä virheviestin. Socketin käyttäminen on hyödyllistä jo siksikin, että sen avulla tietoa voidaan lähettää ja vastaanottaa, vaikka käytössä olisi kahdella eri ohjelmointikielellä toteutetut Client- ja Server-ohjelmat. Tyypillisesti Client, eli asiakasohjelma pyytää dataa Server- eli palvelinohjelmalta. [18; 19; 20.]

Socket-moduulissa datapaketit siis liikkuvat kahden prosessin välillä. Tässä työssä käytetty tapa hyödyntää TCP/IP-protokollaa, jolla tarkoitetaan usean eri Internet-protokollan muodostamaa pinoa [21]. Perusperiaatteena serverisovellus, eli tämän työn tapauksessa robotin sisältämä RAPID-koodi odottaa yhteyden muodostamisen jälkeen asiakkaalta, eli RasPilta tulevaa dataa. Näin laitteet voivat kommunikoida keskenään ja tarvittava tieto liikkuu jouhevasti kahden eri kielellä toteutetun ohjelman välillä. Client-Server-jaottelusta huolimatta tietoa voidaan lähettää kuitenkin molempiin suuntiin. Pakettien lähetys suoritetaan send-komennolla. Tällöin vastaanottajan tulee olla listen-komennon mukaan kuuntelemassa haluttua IP-osoitetta ja porttia, jolloin saapuva paketti voidaan vastaanottaa. Kuvassa 10 on esitetty yhteydenmuodostukseen vaadittavat koodirivit Pythonin (asiakas) ja RAPIDin (serveri) välillä.

```
PYTHON
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
port = 5555
s.connect(('192.168.0.101', port))

RAPID
SocketCreate server_socket;
SocketBind server_socket, "192.168.0.101", 5555;
SocketListen server_socket;
```

Kuva 10. Socket-yhteyden luonti Python- ja RAPID-koodin välillä

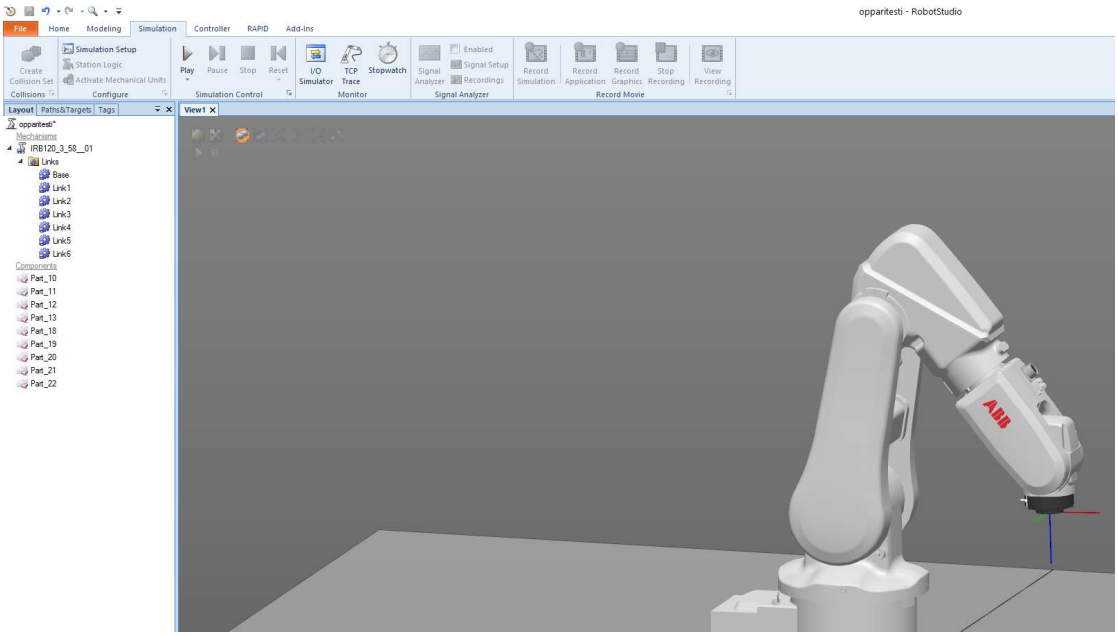
Yhteys muodostuu siis hyvin yksinkertaisilla komennoilla ja pysyy päällä, ellei sitä erikseen katkaista. Yhteyden katkaisu voidaan toteuttaa close-komennolla.

3.4 RobotStudio

ABB IRB -robottien ohjelmointi ja määrittely suoritetaan pääosin käyttäen RobotStudio-sovellusta. Sillä voidaan sekä ohjata oikeaa robottia kuin myös simuloida virtuaalista robottia. [22, s 62.] Simuloinnista on erittäin suuri hyöty laitteen toimintojen testaamisessa, kun testausta voidaan suorittaa ilman pelkoa laitteiden hajoamisesta. Oikeista ympäristöistä voidaan myös luoda 3d-mallleja, jolloin esimerkiksi liukuhihnaympäristöä toteutettaessa voidaan sitä

simuloida ohjelmallisesti jo ennen kuin mitään käydään varsinaisesti rakentamaan, näin mahdolliset riskit voidaan ottaa huomioon jo hyvin aikaisessa vaiheessa. Ohjelmistosta on saatavilla 30 päivän kaikki omaisuudet sisältävä kokeiluversio, jonka jälkeen käyttöön jäävät vain perustoiminnot. Tässä työssä sovelluksen maksullisia toimintoja ei tarvittu, riitti, että simulointi pelkällä robotilla, yhteyden muodostaminen ja RAPID-koodin muokkaaminen onnistui. RobotStudion kautta toteutettu testaus on vaivatonta. Erityisesti simuloinnissa on hyödyllistä, kun käytössä on ohjelman sisältämä virtuaalinen FlexPendant-ohjain, jolla voidaan toteuttaa käytännössä kaikki samat toiminnot kuin oikeallakin robotilla.

Sovelluksen käyttö on varsin vaivatonta sen selkeän käyttöliittymän vuoksi. Esimerkiksi *Microsoft Office* -ohjelmistoa käyttäneelle RobotStudiossa navigointi sujuu luonnostaan niiden yhtenevän ulkoasun kanssa. Käyttöliittymä esitettynä kuvassa 11.



Kuva 11. RobotStudion käyttöliittymää

Uuden projektin tapauksessa työjärjestys on yksinkertainen. File-valikosta valitaan kohta *New – Solution with Station and Virtual Controller*, jonka jälkeen laitteen parametrit voidaan määrittää halutuiksi. Näin on mahdollista luoda esimerkiksi todellisuutta vastaava projekti virtuaalisena, etenkin mikäli ohjelmiston täysversion mallinnusominaisuuksia käytetään hyödyksi. Oikean robotin kanssa RobotStudion käyttö tuo etenkin sen merkittävän edun, että RAPID-

koodia käsitellessä voidaan testivaiheessa koodia muokata tarvittavalla tavalla ja sitten lähettää korjaukset suoraan robotille ja ajaa testi. Kun muutosten lähetys robotille ei käytännössä vie lainkaan aikaa, on testejä helppo tehdä jokaisen pienenkin muutoksen jälkeen. Tällaisia muutoksia varten ohjelmiston RAPID-valikosta löytyy painike *Request Write Access*, jota painamalla voidaan FlexPendantin avulla antaa RobotStudion tehdä suoraan muutoksia robotin sisältämään ohjelmaan.

4 TOTEUTUS

4.1 Työn alkuvaiheet

Alkuperäinen suunnitelma työn toteuttamisesta sisälsi niin sanotusti ”oikean” konenäkökameran käyttämistä. Tässä kohtaa ajatus oli liittää kamera ABB:n RobotStudio-ohjelmistoon, jossa kameraa voitaisiin ohjata sen sisältämän *Integrated Vision* -lisäosan avulla. Vaihtoehtoja eri kameroiksi oli tarjolla useampi kappale. Ensimmäisenä testiin päätyi *Cognex DVT Legend 542C* -konenäkökamera. Kyseinen laite kykenee itse analysoimaan ottamansa kuvat, jolloin laskentaa ei tarvitse tehdä erillisellä tietokoneella. Kamera toimi kuten haluttiin ja siitä löytyi kaikki tarvittavat ominaisuudet työn suorittamiseen. Ongelmaksi muodostui kuitenkin kameran vanha ikä. Kyseiselle laitteelle ei löytynyt enää ohjelmistotukea nykyaikaisille käyttöjärjestelmille (viimeisimpänä Windows XP), minkä vuoksi kuvausohjelmistoa oli ajettava virtuaalikoneella varsinaisen käyttöjärjestelmän päällä. Lähtökohtaisesti se vaikutti liian ongelmalliselta, ja vailla minkäänlaista aiempaa kokemusta aiheesta, päädyttiin lopulta etsimään vaihtoehtoisia ratkaisuja.

Seuraavana oli vuorossa saksalaisen Baslerin valmistama *Pilot pi2400-17gc* – konenäkökamera. Erona aiempaan oli se, ettei kyseinen kamera sisällä omaa laskentayksikköään, vaan kuvien analysointi on tehtävä tietokoneella. Ialtään se oli reilusti aiempaa uudempi – samoja ongelmia ohjelmistojen suhteen ei siis olisi. Kuvaaminen kameralla onnistui, mutta tarvittavan tiedon tuonti kamerasta pois ei tässä vaiheessa ollut vielä tekijällä hallussa. Ensimmäisen kameran kohdalla ei tätä ongelmaa ollut, kun ohjelmisto oli toimiva ja laskenta (esimerkiksi kuvassa näkyvän kappaleen keskikohdan määritys) suoritettiin kamerassa itsessään. Nyt täytyisi siis kehittää keino saada tietoa ulos otetuista kuvista jatkokehitystä ja laskentaa varten.

Kun valmista ohjelmistoa ei ollutkaan käytettävissä, tuli lähestymistapaa työtä kohtaan muuttaa. Vielä tässä kohtaa ei ollut selvillä tarvittavan ohjelmointityön määrä, koska oletuksena oli ollut, että olemassa olevat ohjelmistot suorittavat suurimman työn ja omaksi tehtäväksi jää enimmäkseen tietojen yhdistely eri lähteistä ja muut yksinkertaisemmat tehtävät. Tämän lisäksi RobotStudion *Integrated Vision* -lisäosa toimii vain Cognexin InSight -kameroilla, jolloin kyseistä ohjelmistostakin hyödyttiin vähemmän [22]. Tässä vaiheessa alkoi syntyä ensimmäinen ajatus siitä, että konenäön sekä kameran ja robotin välinen kommunikointi tulisi pystyä hoitamaan täysin ilman valmiita ohjelmistoja. Luonnollisesti robotin sijaitessa koululla tuli työtä tehdä myös siellä. Tämä toi esiin oman käytännön ongelmansa. Robotti sijaitsee Xamkin Mikkelin kampuksen automaatiotekniikan laboratoriossa - luokassa, jossa pidetään usein opetustunteja. Luokassa on oppilaiden käytettävissä tietokoneita, joita käytetään sekä oppituntien aikana että esimerkiksi harjoitustöiden teossa tuntien ulkopuolella. Luokan käyttöaste on siis varsin suuri. Kun työtä käytiin tekemään ja käytettäviin laitteistoihin tutustumaan, kävi varsin nopeasti selväksi se, kuinka paljon aikaa kuluu, kun kaikki tarvittavat ohjelmistot asennetaan kulloinkin käytettävissä olevalle tietokoneelle ja tarvittavat tietoliikenneyhteydet saadaan toimimaan. Nämä hidasteet saivat ajatuksen kääntymään eriytetympiin ratkaisuihin.

Jotta työtä voisi päivittäin jatkaa ilman pitkiä alkuvalmisteluja, täytyi kaikki tarvittava materiaali koota hallitusti yhteen paikkaan. Vaihtoehtona tähän olisi olleet esim. perinteinen kannettava tietokone, joka olisi sitten liitetty koulun verkkoon ja/tai robottiin. Mielenkiintoisempana vaihtoehtona kuitenkin valikoitui Raspberry Pi -tietokone. Selvitystyön jälkeen kävi ilmi, että kyseisellä laitteella olisi mahdollista tuottaa konenäkölaitteisto käyttämällä Python-ohjelmointikieleen tuotettua OpenCV-konenäkökirjastoa. Laitteen pieni koko, edullinen hinta sekä mahdollisuus käyttää laitetta muissa projekteissa tulevaisuudessa oli kaikki erittäin houkuttelevia ominaisuuksia. Näin työssä päädyttiin sivuuttamaan koululta löytyvät konenäkökamerat ja rakentamaan laitteisto Raspberryllä sekä siihen liitettävällä pienellä *Raspberry Pi Camera Module v2*:lla. Tämä toisi merkittäviä muutoksia työn kulkuun sekä kasvattaisi kokonaistyömäärää, mutta tässä tapauksessa sitä ei pidetty lainkaan huonona asiana. Kun käytettävä laitteisto oli kasassa, voitiin varsinaisen konenäkösovelluksen

suunnittelu aloittaa.

4.2 Raspberry Pin esiasetukset

Ennen ohjelmoinnin aloittamista tulisi Raspberry Pihin asentaa vaadittavat työkalut, jotta ohjelmointi sujuisi käytetyn laitteiston suhteen ongelmitta. Raspberry Pihin aiemmin asennettu Raspbian-käyttöjärjestelmä sisälsi valmiiksi *Thonny Python IDE* -ohjelman sekä Python-ohjelmointikielen version 3.7.3, joita käytettäisiin yksinomaan konenäkösovelluksen ohjelmointiin. Käyttöjärjestelmä oli asennettu 32 gigatavun microSD-muistikortille. Raspbianista itsestään oli käytössä 30.9.2019 julkaistu versio. Tätä, kuten muitakaan ohjelmia tai ohjelmointikielten osia, ei työn aikana päivitetty lainkaan. Tähän on useita syitä. Lähtökohtaisesti työssä käytettävän laitteen ei missään vaiheessa olisi tarvetta olla kytkettynä internetiin. Vaatimuksena oli ainoastaan liittyminen robotin kanssa samaan langattomaan verkkoon, mutta tämä ei vaatisi internetyhteyttä. Tällöin ei olisi vaaraa tietoturvapäivitysten, ynnä muiden vastaavien ongelmien suhteen. Toinen syy päivittämättömyydelle oli se, että esimerkiksi Python-kielelle kirjoitetut kirjastot ja moduulit, kuten käyttöön tuleva OpenCV-kirjasto saattaisi päivityksen myötä lakata toimimasta, jos ohjelmistotukea uudelle versiolle ei olisi. Kyseessä olisi joka tapauksessa siinä määrin yksinkertaisia toimintoja vaativa projekti, ettei uusimmille versioille ja päivityksille nähty tarvetta.

OpenCV-kirjastosta laitteeseen asennettiin sen uusin nelosversio käyttäen www.pyimagesearch.comista löytyvää ohjetta [23]. Raspbianiin kirjastoja asennettaessa itse asennus tapahtuu komentorivin kautta, joka voi toisinaan helposti tuoda ongelmia graafisiin käyttöliittymiin tottuneelle käyttäjälle. Muita kirjastoja ei tultaisi lähtökohtaisesti tarvitsemaan.

Työn alkuvaiheessa Raspberry Pi tultaisiin kytkemään HDMI-kaapelilla pöytätietokoneen näyttöön, mutta aikaa myöten työssä tultaisiin siirtymään kokonaisuudessaan *VNC Server*- sekä *VNC Viewer* -nimisiin ohjelmiin. *VNC Server* on Raspbianista valmiiksi löytyvä ohjelma, joka mahdollistaa etäyhteyden muodostamisen langattoman verkon välityksellä toisella koneella olevan *VNC Viewer* -sovelluksen kanssa. Kun sekä Raspberry Pi että satunnainen pöytätietokone olisivat samassa langattomassa verkkoyhteydessä, voitaisiin VNC-

sovelluksia käyttäen näyttää Raspberry Pin työpöytänäkymä pöytäkoneella ja täten käyttää myös pöytäkoneen näppäimistöä ja hiirtä apuna. Tästä olisi suuri hyöty, kun riittäisi, että pöytäkoneen käynnistäisi normaalisti ja Raspberry Pihin kytkisi pelkän virtakaapelin. Laitteen käynnistyttyä ja verkkoyhteyden muodostuttua ilmestyisi Raspberry Pi pöytäkoneella ajettavaan *VNC Viewer* -sovellukseen. Pääosin käytetty yhteys toimi hienosti, joskin aika ajoin luotu yhteys saattoi satunnaisesti tuntemattomaksi jääneestä syystä katketa. Paras arvaus on RasPin jääminen robotin pääosin metallisen rungon ja langattoman reitittimen väliin, joka saattaisi aiheuttaa katkoksen laitteiden välisessä langattomassa yhteydessä, jolloin myös työpöytänäkymä luonnollisesti katkeaa.

Ongelmat olivat kuitenkin siinä määrin harvinaisia, ettei niistä koitunut suurempaa harmia tai viivästystä työn suorittamiseen. Työssä käytettävä Raspberry Pi pidettiin muutoin mahdollisimman puhtaana kaikista ylimääräisistä sovelluksista. Siihen ei asennettu mitään muita uusia sovelluksia tai tehty muutoin muutoksia. Tällä tavoin pyrittiin varmistamaan, ettei työn aikana ilmenisi ongelmia käytettävien laitteiden kanssa esim. käytettävissä olevan muistikapasiteetin tai prosessointitehon rajallisuuden vuoksi. Samasta syystä työn aikana käytetty RasPi suojattiin salasanalla, jottei ulkopuoliset tahot pääsisi tahallisesti tai tahattomasti laitteen asetuksia muuttamaan.

4.3 Ohjelmointivaiheen aloitus

Ensimmäiseksi varsinaiseksi tehtäväksi muotoutui kuvan ottamisen ja sen analysointiin liittyvien osien ohjelmointi. Kameraa ei tultaisi asentamaan itse robottiin kuin vasta paljon myöhemmin – tässä vaiheessa tärkeintä olisi tuottaa toimivaa koodia ja valmistella ohjelmallisia toimintoja tulevia vaiheita varten. Suurin osa sovelluksesta kirjoitettiin joululoman aikana, jolloin ei muutoinkaan ollut käytännön syistä mahdollisuutta viettää aikaa koululla.

Vaikkei valmista sapluunaa työlle ollutkaan, periaatteellinen käsitys ns. ohjelmakierrosta voitiin määrittää jo työn alkuvaiheessa ja tällöin ei ollut niin suurta ongelmaa siitä, missä järjestyksessä eri vaiheet toteutettaisiin. Työ tulisi etenemään *suurin piirtein* seuraavassa järjestyksessä: kappaleen ääriviivojen tunnistaminen, kappaleen keskipisteen laskenta, robotin ja Raspberry Pin väli-

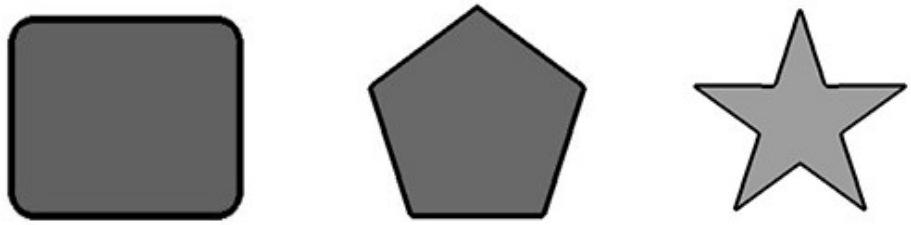
sen yhteyden muodostaminen, koordinaattien muunnokset eri koordinaatistojen välillä, kameran toiminnan testaaminen ja kalibrointi, robotin liikkuminen ja lopulta kaikkien osien huolellinen yhteensovitus, testaus ja säätötoimenpiteet. Äkkiseltään ajatus kameran mukaan tuonnista vasta varsin myöhään projektin aikana tuntui alkuun sikäli huonolta ajatukselta, että mikäli kamera olisi laadultaan tai muilta ominaisuuksiltaan riittämätön kyseisen kaltaiseen projektiin, olisi siitä huomattavaa harmia, mikäli vanhoihin vaihtoehtoihin pitäisi sittenkin palata. Vastaavan kaltaisia konenäköprojekteja oli kuitenkin aiemminkin toteutettu hyvin tuloksin, joten omaan työhön uskalsi luottavaisin mielin ryhtyä.

Ohjelmakoodi tultaisiin kirjoittamaan pääosin Python-ohjelmointikielellä – robotti sen sijaan ymmärtää vain ABB:n kehittämää RAPID-koodia. Kun robottiin ei vielä tultaisi hetkeen koskemaan, oli nyt siis hyvä hetki aloittaa Python-kielen opiskelut. Internet on täynnä ohjelmointiin liittyviä tutoriaaleja ja verkkokursseja, joita seurailemalla taidot varsin nopeasti karttuvat. Vähän kerrallaan koodia oli kertynyt jo siinä määrin, että ensimmäiset kuvantunnistusharjoitukset olivat valmiina. OpenCV-kirjastoa käytettäessä kappaleiden ääriviivoja haetaan findContours-komennolla (kuva 12).

```
cnts = cv2.findContours(img_binary, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)
```

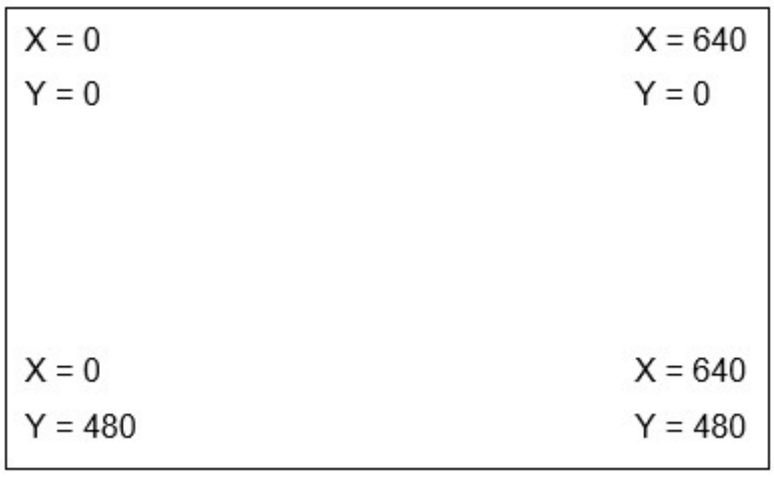
Kuva 12. Esimerkki findContours-komennosta

Kappaleiden tunnistuksen ja niiden ääriviivojen haun harjoittelua varten valmisteltiin digitaalisesti yksinkertaisia, muodoiltaan eriäviä kappaleita, joita koodille sitten tarjottiin tutkittavaksi. Kokeilut onnistuivat hyvin, ja kappaleiden ääriviivat tunnistuivat halutusti. Kappaleen muodolla ei ollut käytännössä merkitystä – ääriviivat löytyivät kerta toisensa jälkeen. Kun testikuvilla harjoiteltiin, tuli silti pitää mielessä se, että harjoittelukuvien ja todellisen tilanteen ero on mittava, ja vaikka testikuvien kappaleet tunnistuivatkin oikein, tämä ei tulisi tapahtumaan yhtä helposti oikealla valokuvalla. Periaatetasolla toimiva ratkaisu oli kuitenkin löytynyt, joten tätä tekniikkaa tultaisiin soveltamaan myös todellisessa tilanteessa. Kuvasta 13 nähdään erimuotoisten kappaleiden tunnistuminen niihin piirrettyine tummempine ääriviivoineen.



Kuva 13. Testikuvia, joissa näkyvillä löydetyt ääriviivat

OpenCV-kirjastolla on mahdollista saada havaituista kappaleista runsaasti erilaista hyödyllistä tietoa. Näitä ovat mm. kappaleen pinta-ala, piirin pituus ja keskipiste [24]. Tässä työssä tärkeimpänä kappaleesta saatavana tietona on sen keskipisteen sijainti kuvassa, joskin myöhemmässä vaiheessa otettaisiin käyttöön myös kappaleen pinta-alan laskenta tulosten suodattamiseksi. Raspberry Pillä otetuissa kuvissa sen nollakohta, eli origo, sijaitsee kuvan vasemmassa yläreunassa, josta x- sekä y-akselin pikseleitä käydään laskemaan. Tästä selvennyksenä kuva 14, jossa kulmapisteiden koordinaatit lisättynä kuvaan.



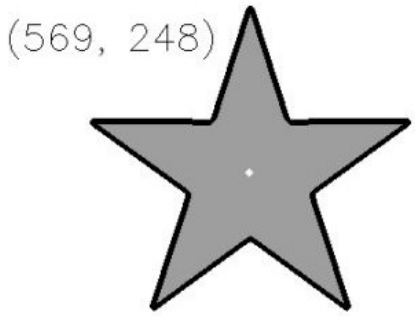
Kuva 14. Kuvan nurkkapisteiden arvot akseleittain 640x480 kokoisessa kuvassa

Kun kappaletta rajaava alue on saatu määritettyä, voidaan OpenCV:n avulla laskea kappaleen keskipisteen sijainti moments-funktion avulla. Python koodissa toimiva moments-funktio esitettynä kuvassa 15.

```
m = cv2.moments(cnts[1])
cx = int(m["m10"] / m["m00"])
cy = int(m["m01"] / m["m00"])
```

Kuva 15. Keskipisteen määritys moments-funktiolla

Koodissa tallennetaan laskettu keskipiste vastaaviin muuttujiin cx, ja cy, jossa cx on x-akselilta laskettu piste ja cy vastaava piste y-akselilla. Tämän jälkeen keskipiste merkitään kuvaan sekä saadut arvot keskipisteen yläpuolelle. Tästä esimerkki kuvassa 16.



Kuva 16. Kappaleen keskipiste ja sen koordinaatit merkittynä esikatselukuvaan

Kuva 17. Mallinnus työssä käytettävistä oikeista kappaleista

Lopullisessa työssä päätettiin käyttää kuvassa 17 näkyviä sylinterin muotoisia, noin senttimetrin korkuisia pyöreitä kappaleita, jotka kokonsa puolesta sopivat täydellisesti robotin tarttujatyökaluun ja täten olisivat oivallisia kappaleita robotin noudettavaksi.

4.4 Kommunikointi robotin kanssa

Koodin ollessa siinä vaiheessa, että testikappaleita voitiin tunnistaa luotettavasti ja tarvittavat tiedot saatiin kuvasta ulos, voitiin aloittaa laitteiden välisen kommunikoinnin ohjelmointi. Tarkoituksena oli luoda yhteys käyttäen socket-protokollaa, jonka avulla voitaisiin luoda verkon yli toimiva yhteys käyttäen laitteiden IP-osoitteita. Näin valmiita koordinaatteja voitaisiin lähettää kameralta robotille langattomasti ja varsin pienellä viiveellä. Kommunikoinnin pohjana käytettäisiin Jatin Goyalin (nimimerkki Jatin1o1) luomaa *ABB-robot-with-python-socket* -nimistä Github.comiin ladattua projektia. Se mahdollistaa Python-koodin ja ABB:n teollisuusrobottien välisen kommunikoinnin käyttäen socket-moduulia. Projektista käytettiin kahta osiota: robottiin ladattavaa server.mod-tiedostoa, sekä Python-koodiin tulevaa yhteydenmuodostus-funktiota. [25.]

Koko opinnäytetyön osalta selkeästi eniten aikaa kului yhteydenmuodostuksessa ilmenneiden ongelmien kanssa – käytännössä useita viikkoja. Tämä

johtui pääosin kokemattomuudesta käytettyjen ohjelmointikielien kanssa. Tällöin pienehkökin virhe koodissa voi aiheuttaa mittavia ongelmia, jos tietotaitoa ohjelmointikielien rakenteista ongelman korjaamiseen ei vielä ole riittävästi. Jos kyseessä olisi ollut tiedon välittäminen kahden Python-koodin välillä, suuremmilta ongelmilta olisi todennäköisesti säästytty. Tässä tapauksessa yhtä aikaa oli kuitenkin käsiteltävänä kahta erityyppistä koodia, jotka molemmat noudattavat omaa ohjelmointilogiikkaansa. Ongelmat vaihtelivat muodostetun yhteyden äkillisestä katkeamisesta aina lähetettyjen ohjauskomentojen vääränmuotoiseen vastaanottoon ja siten yleensä robotin antamaan virhekomentoon, kun vastaanotetut koordinaatit ovat sijaintinsa puolesta mahdottomat saavuttaa.

Aikaa myöten ongelmat kuitenkin lähtivät ratkeamaan, kun ymmärrys aiheesta kasvoi ja testausta oli suoritettu riittävästi. Yhtenä hankalimpana työssä koettuna solmukohtana ollut kommunikoinnin vaikeus ratkesi lopulta iloisesti, kun robotti viimeinkin liikautti niveliään RasPilta lähteneen koordinaatin vastaanoton jälkeen. Tähän pisteeseen asti oli työn valmistumisen suhteen oltu varsin skeptisiä - nyt voitiin ensimmäisen kerran todella uskoa työn valmistuvan ennen pitkää. Lopputuloksena oli siis toimiva socket-yhteys, joka voidaan ohjelman alussa kerran laittaa päälle ja joka sen jälkeen myös pysyy päällä siihen asti, kunnes socket-moduuli erikseen suljetaan tai koko ohjelma pysäytetään. Varsinaisen konenäkösovelluksen lisäksi tehtiin myös toinen lyhyempi ohjelmapätkä, jolla robotille voidaan syöttää käsin halutut koordinaatit – tämä toimi erinomaisesti myöhemmin laitteiston testaus- ja kalibrointivaiheessa, jossa käsin ajoa jouduttiin toistamaan satoja kertoja uudelleen. Kyseisellä testiohjelmalla voitiin myös tutkia helposti erilaisia ongelmia ja niiden ratkaisuja socket-yhteyden kanssa, kun oikeaa kuvaa ei tarvittu, vaan robotille voitiin keksiä haluttu positio aina tarpeen mukaan.

4.5 Kameran ja Raspberry Pin asennus robottiin

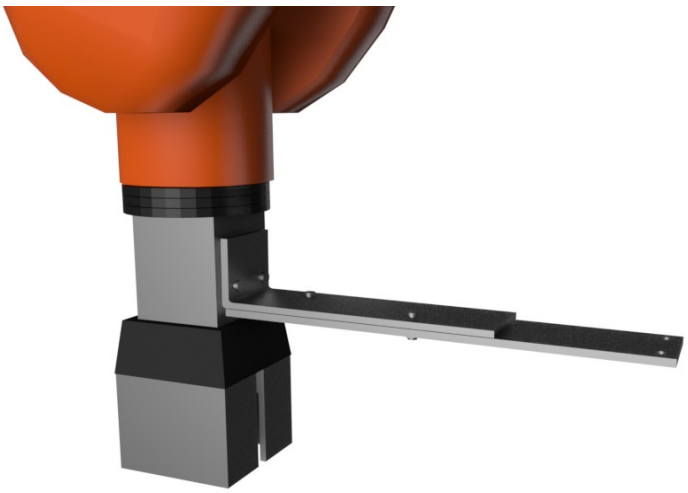
Työn alkuvaiheessa kantavana ajatuksena oli liittää kameramoduuli robotin runkoon siten, etteivät roikkuvat virtajohdot ja kaapelit vaikuta laitteiston toimintaan. Kaikkien robotista erillään olevien kaapeleiden tulisi joka tapauksessa olla tarpeeksi pitkiä ja liikkua siten, että robotti ei omilla liikkeillään pystyisi kaapeleita katkaisemaan tai niitä jännittämään. Tämä olisi ehdottoman

tärkeää, koska kaapelit ja erityisesti Raspberry Pin liitännät ovat siinä määrin heiveröisiä robotin tuottamiin liikevoimiin nähden, että liitoskohtaan kohdistuva veto voisi helposti rikkoa liitännän ja aiheuttaa turhaa viivettä esimerkiksi opinnäytetyön aikataulun suhteen. Ongelman ollessa tiedossa jo ennen asennusta, voitiin se ottaa huomioon ja näin vähentää turhaa riskiä laitteiden vaurioitumisesta.

Kamera oli alun perin tarkoitus kiinnittää robotin työkaluun ja tietokone viedä liitäntäkaapelilla kauemmas sillä ajatuksella, että robotin virheliikkeen seurauksena tietokone ei olisi niin alttiina iskuille. Raspberry Pin kameramoduulin mukana tuleva kaapeli on 15 cm pitkä, ja täten tietokonetta ei saataisi vietyä niin kauas kuin haluttaisiin. Tätä varten tilattiin pidempi kaapeli sekä suojakotelot niin tietokoneelle kuin kamerallekin. Opinnäytetyön teon aikaan maailmalla vaikuttanut koronavirusepidemia kuitenkin hidasti rahtiliikennettä siinä määrin, että tilattujen tuotteiden toimitus peruuntui lopulta kokonaan. Alkuperäisiä suunnitelmia oli siis muutettava ja mietittävä vaihtoehtoisia ratkaisuja.

Tietokone ja kamera tulisi siis liittää robottiin kutakuinkin vierekkäin. Tätä tarkoitusta varten tulisi löytää robottiin sopiva asennuskohta ja -tapa, sekä selvittää kameralaitteiston kiinnitys. Sekä kamerasta että Raspberry Pistä löytyy lähtökohtaisesti niiden kotelointia varten poratut asennusreiät. Näitä reikiä voitaisiin käyttää erityisesti kameran asennuksessa. Kamera tulisi saada asemoitua paikalleen mahdollisimman tarkasti siten, ettei robotin liikkeet tai muukaan ulkoinen voima pääsisi sitä heiluttamaan. Mikäli kamera liikahtaisi asennuspositionstaan, sen toiminta robotin kanssa olisi käytännössä mahdotonta, ennen kuin se kalibroitaisiin uudelleen työalueeseen nähden. Tämä oli yksi tämän opinnäytetyön suurimpia uhkakuvia – laitteisto tulisi olemaan niin herkkä kameran epätoivotuille heilahduksille, että se voisi pahimmassa tapauksessa estää koko työn valmistumisen. Kiinnityksen vahvuuteen tulisi siis panostaa. Kiinnityksessä tulisi myös huomioida kameran polttovälistä aiheutuva kuvan laajakulmaisuus. Jos kamera olisi kiinni liian korkealla työkaluun nähden, voisi työkalun kärki näkyä kuvassa. Testauksen jälkeen todettiin, että kamera tulisi liittää ainakin seitsemän senttimetrin päähän työkalun reunasta, jottei se näkyisi otettavissa kuvissa huomioiden sen suunnitellun asennuskorkeuden työkalun yläpuolella.

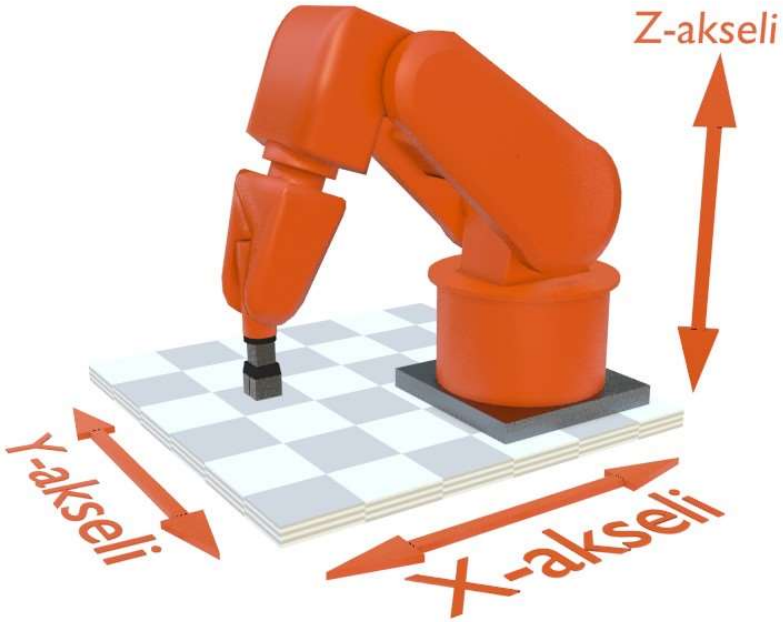
Robotissa on useita mahdollisia kiinnityskohtia. Optimaalisessa tilanteessa kamera kannattaisi asentaa täysin erilliseen tukirakenteeseen turhan liikkeen välttämiseksi, mutta tässä tapauksessa kuitenkin todettiin, että työn harjoitustyyppisen luonteen vuoksi kiinnityksen voisi hoitaa kiinnittämällä laitteet suoraan robottiin. Kameralle paras sijainti löytyi robotin työkalusta, jossa on lisälaitteiden asennusta varten tehdyt kierteelliset reiät. Reikien sijainti työkalussa robotin runkoon nähden olisi sopiva kameran ja tietokoneen asennukselle. Tarkoitusta varten hankittiin rautakaupasta sopivia – yleensä rakennustarvikkeina käytettäviä kulma- ja naulauslevyjä. Sopivilla pulteilla laitteet saatiin asennettua robottiin kiinni. Kuvassa 18 kyseistä kameratukea kuvaava mallinnus.



Kuva 18. Havainnekuva robotille rakennetusta kameratuesta

Kun tukilaite oli asennettu kiinni ja todettu riittävän vakaaksi, voitiin kamera ja Raspberry Pi asentaa siihen kiinni. Sivuttaissuunnassa pultit pitävät laitteet varsin tukevasti paikallaan, ja pelkkä robotin liikkuminen ei vaikuta kameran asentoon. Robotin liikkeet ohjelmoidaan siinä määrin hitaiksi, ettei pelkkä robotin liikehdintä kappaletta noudettaessa kykene muuttamaan kameran asentoa naulauslevyssä. Ongelmana olisikin mahdolliset törmäystilanteet, jolloin riskinä olisi alustana käytettävien teräslevyjen vääntyminen. Itse työkalun asennon ei-toivottu muuttuminen vaikuttaisi myöskin suoraan kameran asentoon, koska kameratuki on asennettu suoraan työkaluun. Edellä mainittujen riskitilanteiden varalta laitteen testaamisessa noudatettiinkin erityisen suurta huolellisuutta, jotta robotti ei liikkeellään ajautuisi törmäykseen alustana toimivan pöydän tai muiden pöydällä olevien objektien – saati itsensä kanssa.

Robotti liikkuu kolmiulotteisessa koordinaatistossa, jossa x- ja y-akselit ovat vaakatasossa robotin alustaan nähden ja z-akseli pystysuuntaan. Tätä on havainnollistettu kuvassa 19. Robotti on asennettu alla olevaan pöytään siten, että x-akselin liike kohdistuu työalueella joko suoraan sen omaa runkoa kohti tai siitä poispäin. Tällöin ohjelmakoodiin voitiin helposti luoda x-akselin koordinaatin varmaa toimintaa varten ns. minimiarvo, jotta robotin työkalu ei missään tilanteessa pääsisi ajautumaan liian lähelle robotin runkoa, jolloin kamera ottaisi kiinni robottiin ja suurella todennäköisyydellä vaurioittaisi kameraa tai sen kiinnitystä. Minimietäisyydeksi asetettiin noin 350 mm, jolloin ohjelman laskemat koordinaatit alkavat aina vähintään arvosta ~350 mm. Tällöin voitiin varmistua siitä, ettei kamera vahingossa vaurioituisi liian pienen x-akselin koordinaatin vuoksi.



Kuva 19. Robotin sijainti pöytätasolla ja sen koordinaattiakselien suunnat

Samankaltainen periaate akselin liikkeen rajoittamisesta tehtiin myös z-akselille. Kappaleentunnistus ei välitä objektin korkeudesta - sillä määritetään vain x- ja y-akseleiden koordinaatit. Täten pystyakselille voitiin asettaa vakiokorkeus objektin noutoa varten. Kun työkalu käy aina samalla korkeudella pöytätasosta, ei törmäysriskiä tässä tilanteessa samalla tapaa ole. ABB:n teollisuusrobotit sisältävät ohjelmallisen SafeMove-toiminnon, jolla turva-alueita voidaan määritellä suoraan robottiin itseensä, mutta koska kyseessä oli tässä

tapauksessa pelkkä kahden akselin hyvin suoraviivainen liikkeiden rajoitustoimenpide, voidaan riskit karsia pois jo ennen kuin koordinaatteja edes lähetetään [26].

Valmis kokonaisuus suunniteltiin toimivaksi ns. jatkuvalla syklillä, jolloin olisi alun perinkin turhaa lähettää riskialttiita koordinaatteja robotille. Tästä aiheutuisi vain turhia keskeytyksiä laitteen toimintaan, kun robotti huomaisi virheellisen koordinaatin ja pysäyttäisi laitteen. Kun kamera ja tietokone saatiin kiinnitettyä ja turvamääritykset tehtyä, voitiin edetä kuvan kalibrointivaiheeseen.

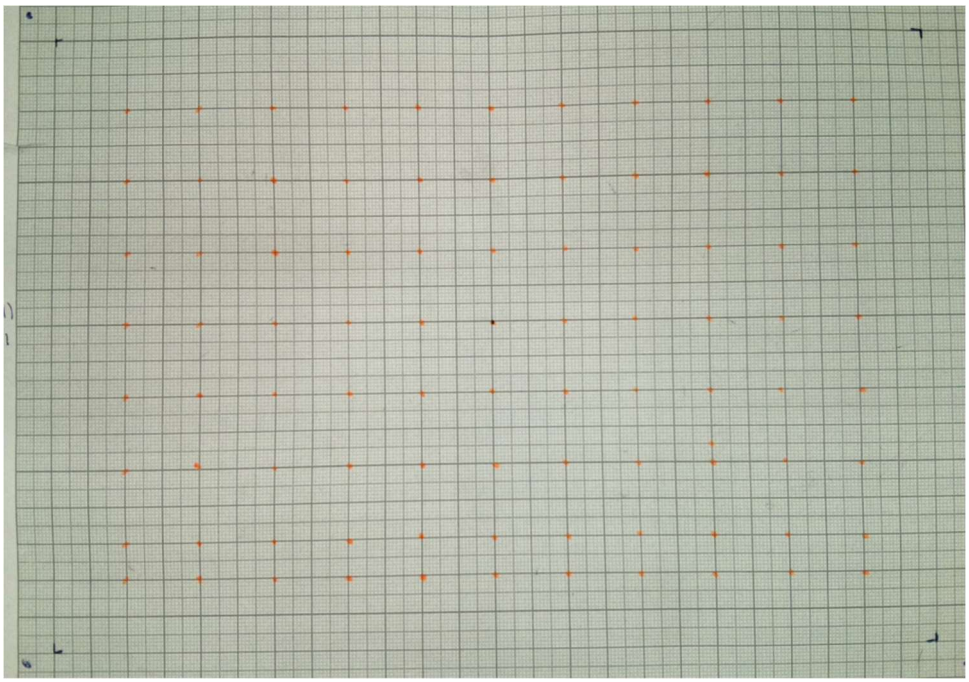
4.6 Kuvan kalibrointi

Robotin noutamien kappaleiden työalueen kooksi määriteltiin alun perin noin A4-paperiarkin kokoinen alue, eli 210 x 297 millimetriä. Periaate oli se, että kamera robottiin asennuksen jälkeen suunnattaisiin siten, että kameran ”näkemä” alue kattaisi koko A4-paperin. Näin työalue pysyisi maltillisen kokoisena ja mahdolliset kameran sijainnin vuoksi perspektiivistä aiheutuvat ongelmat kappaleen tunnistamisen suhteen voitaisiin pitää tarpeeksi pienenä. Tätä asiaa tarkasteltaisiin tarkemmin työn edetessä. Kameran asettelu suoritettiin ajamalla robottia käsiajolla pitäen samalla kameran tuottamaa videokuvaa tietokoneen ruudulla. Lisäksi apuna käytettiin tarkoitusta varten tulostettua millimetripaperia, jolla kohdistus olisi verrattain helppoa tehdä.

Ongelma videokuvan käytössä on siinä, että Raspberry Pin kameramoduuli tukee vain tiettyjä resoluutioita, joiden kuvasuhde ei vastaa A4-arkin kokoa. Tämän vuoksi kameran paikoitus vastaamaan kuvauksen kohteena olevaa paperia olisi liki mahdotonta. Tästä syystä videokuvauksella päädyttiin lopulta vain asettelemaan kamera työalueen keskikohtaan ja säätämään kameran asento mahdollisimman suoraan työalueen tasoon nähden, jotta vältyttäisiin ylimääräisiltä virheiltä koordinaatistojen yhteensovituksessa. Kun video-osuus oli suoritettu, otettiin samasta positiosta yksittäisiä kuvia, joista sitten voitiin rajata haluttu alue työalueeksi. Samalla lopullinen kuvanottopositio määräytyi – tähän pisteeseen robotti ajaisi itsensä aina kuvan ottoa varten. Robotin paikoitustarkkuuden ollessa ± 0,01 millimetriä, voitiin olla luottavaisia siitä, että robotin liikkeet eivät aiheuttaisi ongelmia kameran position suhteen [28]. Myöhem-

min havaittiin, että millimetripaperi ei käytännössä vaikuta kappaleentunnistukseen, joten paperi päätettiin jättää työalueen pohjaksi.

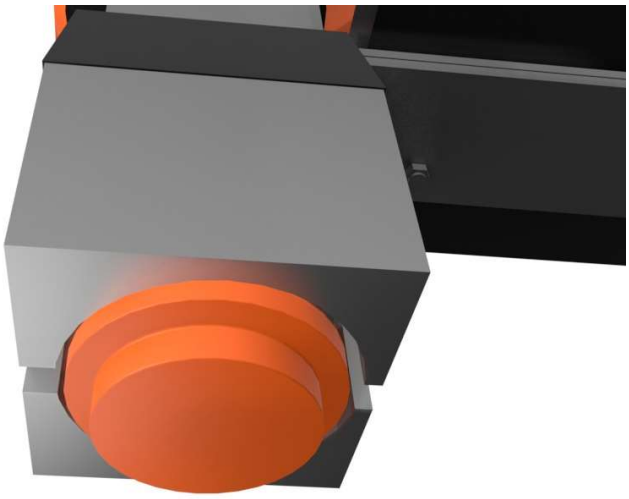
Ongelmana kameran ottamissa kuvissa on niihin aiheutuvat vääristymät. Niitä aiheuttavat mm. optiikan laajakulmaisuudesta aiheutuva tynnyrivääristymä ja mahdollinen kameran virheellinen asento työalueeseen nähden. Tynnyrivääristymä näyttäytyy kuvassa sen reunoilla suorien linjojen ”pullistumisena” [28]. Käytännön testit osoittivat, että tässä tapauksessa tynnyrivääristymästä ei olisi kuitenkaan suurempaan haittaa. Tämä käy ilmi kuvasta 20, jossa tynnyrivääristymää on havaittavissa varsin vähän.



Kuva 20. Kameran kuvaama työalue

Työtä tehdessä piti pohtia virheiden korjauksen menetelmiä. Jokaista erilaista optiikasta, kameran asennosta ja kappaleen tunnistuksesta aiheutuvaa virhettä voitaisiin yrittää korjata yksitellen laskemalla ja testaamalla robotin toiminta korjatuilla arvoilla. Vaihtoehtoinen ratkaisu olisi tehdä yksi isompi arvojen kompensointilauseke, jolla kaikki edellä mainitut ongelmat voitaisiin käsitellä yhdellä kertaa. Robotin työkalun eli tarttujan ollessa ns. auki, sen maksimihalkaisija on joitain millimetrejä suurempi kuin noudettavat kappaleet, joten sataprosenttista tarkkuutta ei välttämättä edes tarvittaisi. Kun kappaleen nou-

dosta ja noudon tarkkuudesta ei vielä tässä vaiheessa ollut käytännön kokemusta, jätettiin ongelma myöhemmin ratkaistavaksi. Kuvassa 21 on havainnollistettu kappaleen ja tarttujatyökalun välisiä mittasuhteita.



Kuva 21. Kappaleen ja työkalun väliin jäävä rako

4.7 Kameran ja robotin koordinaatiston yhteensovitus

Tutkittavan kuvan ja robotin voidaan käsittää toimivan kahdessa eri koordinaatistossa. Kuvassa kappaleen keskipisteen koordinaatit määräytyvät riippuen niiden sijainnista kuva-alueella pikseleissä laskettuna kuvan vasemmasta yläreunasta alkaen. Lopullisen tutkittavan kuvan resoluutioksi asetettiin 1000 x 707 pikseliä, joka noudattaa A4-paperin kuvasuhdetta. Kuvan lyhyemmän sivun pituuden laskenta osoitettu kaavassa 1.

$$297\ x\ 210\ [mm] \approx 1.414\ \rightarrow \frac{1000\ px}{1.414} \approx 707\ px \tag{1}$$

Esimerkkinä tällöin kappaleen koordinaatit sen ollessa kuvan keskipisteessä olisivat x = 500, ja y ≈ 354. Nämä arvot tulisi muuttaa robotin koordinaatistoon siten, että ne vastaisivat kappaleen todellista positiota työalueella. Täten robotti osaisi sille lähetettyjen koordinaattipisteiden avulla noutaa kappaleen pöydältä. Robotin koordinaatistosta tulisi määrittää työalueen äärimmäiset pisteet ja sovittaa koordinaatisto niiden mukaan. Kameran ollessa kuvausposition sillä voitiin ottaa testikuvia kappaleen ollessa pöydällä ja määrittää alue, jolla kappale näkyisi kokonaisuudessaan. Toisin sanoen, vaikka kappaleen todellinen keskipiste sijaitsisi työalueella, mutta kamera näkisi siitä vain

esimerkiksi kolme neljäsosaa, tulisi keskipisteen laskentaan mukaan vain se osuus kappaleesta, joka kameran kuvassa näkyisi. Tällöin kappaleen keskipisteen laskenta tuottaisi vääriä tuloksia. Lopulliseksi työalueen kooksi määritettiin siten 178 x 270 millimetrin kokoinen alue.

Robotille tuli ensiksi määritellä se piste, josta työalueen reuna alkaa, kun akselien suunnat olivat tiedossa. X-akselilla tämä arvo oli 349 mm. Tällöin robotin ollessa x-akselin suhteen pisteessä 349, sen työkalun keskipiste olisi työalueen reunan tasalla. Tämä on siis sama arvo kuin aiemmin mainittu ”turvaetäisyys” robotin rungosta. Tähän arvoon sitten lisättäisiin tarvittava etäisyys, jotta päästäisiin kappaleen kohdalle työalueella. Työalueen ollessa x-akselin suhteen 178 millimetriä pitkä käytettäisiin tätä arvoa etäisyyden laskemiseen. Kameran kuvasta voitaisiin helposti laskea kappaleen keskipisteen etäisyys kunkin akselin reunasta. Näin saataisiin prosenttimuotoinen luku, jolla robotin työalueen maksimiarvoa voitaisiin kertoa. Kappaleen keskipisteen sijainti saadaan siis sijoittamalla tarvittavat arvot kaavaan 2.

$$\textbf{Akselin minimiarvo} \pm (\ \textbf{kappaleen keskipisteen sijainti kuvassa}\ [\%]\ ) \times \textbf{työalueen koko} \quad (2)$$

Esimerkkilasku x-akselin suhteen kappaleen ollessa kuvan keskipisteessä osoitettuna kaavassa 3.

$$\mathbf{349\ mm} + \left(\frac{\mathbf{354\ px}}{\mathbf{707\ px}}\right) * \mathbf{179\ mm}\ \approx \mathbf{439\ mm} \quad (3)$$

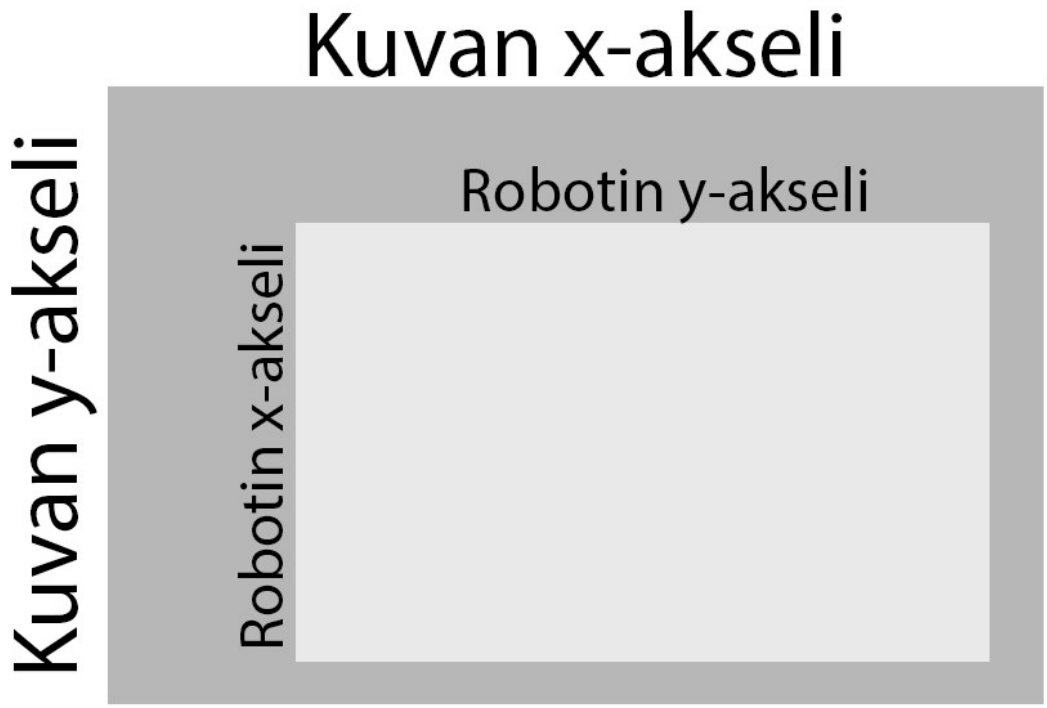
X-akselille laskettu robotin koordinaatistoon sovitettu koordinaattipiste olisi siis arvoltaan 439. Python-koodissa sama laskutoimitus on esitetty kuvassa 22.

```
# Muunnetaan x-koordinaatti robotille
x_koord = int(349+(cy/x_resoluutio)*x_robot)
```

Kuva 22. X-akselin koordinaatin muunnos robotille

Vastaavat arvot laskettiin samaa kaavaa käyttäen myös y-akselin suhteen. Y-akselin alkupisteeksi määriteltiin 72 mm jolloin - toisin kuin x-akselin tapauksessa - kuvasta saatuja arvoja vähennettäisiin alkuarvosta akselin ”suunnasta” johtuen. Laskentaa sekoittaa se seikka, että kuvan x- ja y-akselit ovat

toisinpäin, kuin robotin akselit. Kuvassa kuljettaessa kuvan reunasta ylhäältä alas, kasvavat y-akselin koordinaatit. Kuljettaessa robotilla samaan suuntaan kasvavat x-akselin koordinaatit. Tämä tuli huomioida laskentaa tehdessä. Kuvassa 23 on robotin ja kuvan koordinaatistojen eroja kyetty selkeyttämään.



Kuva 23. Robotin ja kuvan akselien suunnat

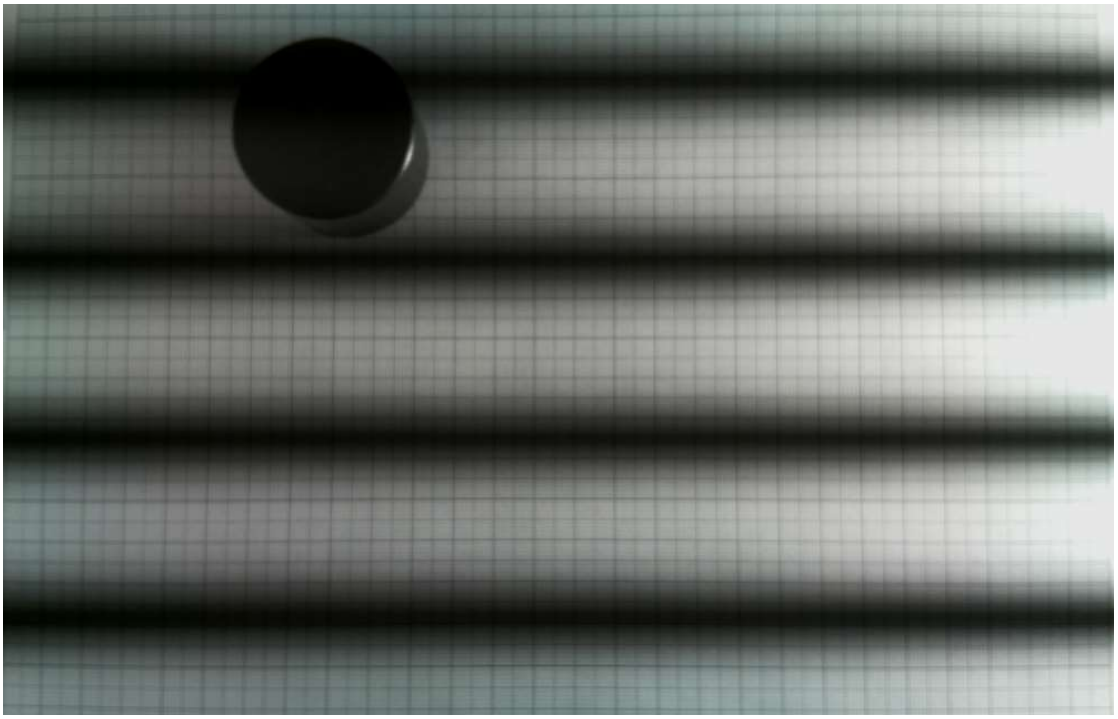
Vaikkakaan lopullisia noutokoordinaatteja ei tässä kohtaa vielä saataisi selville, voitiin nyt kuitenkin jo testata manuaalisesti koordinaatistojen yhteensopivuutta. Huomattiin, että yhteensovitus toimii varsin hyvin – robotti ajaa kutakuinkin oikeaan positioon simuloidun kappaleen päälle. Testejä tehdessä ohjelmakoodi ei vielä automaattisesti lähettänyt kuvasta saatuja arvojaan robotille, vaan kuvan tutkinnan jälkeen koordinaatit syötettiin robotille käsin. Siihen päällimmäisenä syynä oli robotin RAPID-koodin keskeneräisyys ja yleisen varovaisuusperiaatteen noudattaminen. Koordinaatistojen yhteensovituksen jälkeen tuli paneutua tarkemmin kappaleentunnistukseen ja työskentelyyn oikeilla valokuvilla ja objekteilla.

4.8 Valaistus

Konenäkösovelluksissa työalueen valaisulla on erittäin suuri merkitys ohjelman toimivuuden kannalta. Nimensä mukaisesti valokuvia otettaessa valon

määrä ja laatu ovat keskeisimpiä kriteerejä. Tämä työvaihe toi esille mielenkiintoisia ongelmia valaistuksen toteutuksen suhteen, mutta samalla parhaiten toimiva ratkaisu yllättää. Ammattimaisemmin toteutetuissa konenäköprojekteissa voidaan usein käyttää suurempi määrä resursseja valon muokkaukseen kohteessa – tämä on usein vaatimuksena jo laitteiston toimintavarmuudenkin kannalta. Mitä suurempaa tarkkuutta esim. värien tai muotojen analysointiin vaaditaan, sitä suurempi huomio valaistuksen laatuun on kiinnitettävä. Tässä projektissa tärkeintä oli saada tausta erottumaan tarpeeksi hyvin tutkittavasta kappaleesta siten, ettei esimerkiksi kappaleen varjo heikennä ääriviivojen tunnistuksen toimintaa liiaksi. Erillisiä värejä ei käytännössä tarvinnut tunnistaa, vaikka kappaleen lajittelu värin perusteella lopulliseen versioon luotiinkin. Käytetyistä tekniikoista kuitenkin lisää myöhemmässä vaiheessa.

Työn edettyä siihen vaiheeseen, että testikuvista voitiin siirtyä todellisiin kohteisiin, täytyi huomio ensin kiinnittää kohteen valaisuun. Valaistusta mietittäessä tämänkaltaiseen kohteeseen, on tärkein ominaisuus valon tasaisuus – varjot pilaavat kuvan. Ensimmäisenä kokeiluna kohteen valaisussa oli käyttää tavallista led-polttimoilla toimivaa työmaavalaisinta. Työmaavalaisimen hyvänä puolena on runsas valon määrä. 30 W teholla toimivasta valosta riittäisi tarpeeksi valoa pienen työalueen valaisemiseen. Välittömästi kuitenkin huomattiin työmaavalon tuoma ongelma – valon epätasainen jakautuminen työalueelle. Valaisinta ei saatu asennettua kameran yhteyteen robottiin kiinni, jolloin se jouduttiin sijoittamaan kohteen sivulle. Erillisten valaisutelineiden puutteessa valaisinta pidettiin samalla, noin yhden neliömetrin kokoisella pöytätasolla, johon robottikin oli asennettu. Lisäksi ongelmia aiheutti led-valaisimen käyttämän verkkojännitteen taajuudesta (50 Hz) ja kameran suljinajan eroista aiheutuvat, kuvassa erittäin selkeästi näkyvät tummemmat ja vaaleammat juovat. Videokuvassa tämä näkyy erittäin selkeästi havaittavana välkkymisenä. Ongelman laatu käy hyvin ilmi kuvasta 24.



Kuva 24. Led-valon välkkymisestä aiheutuva efekti

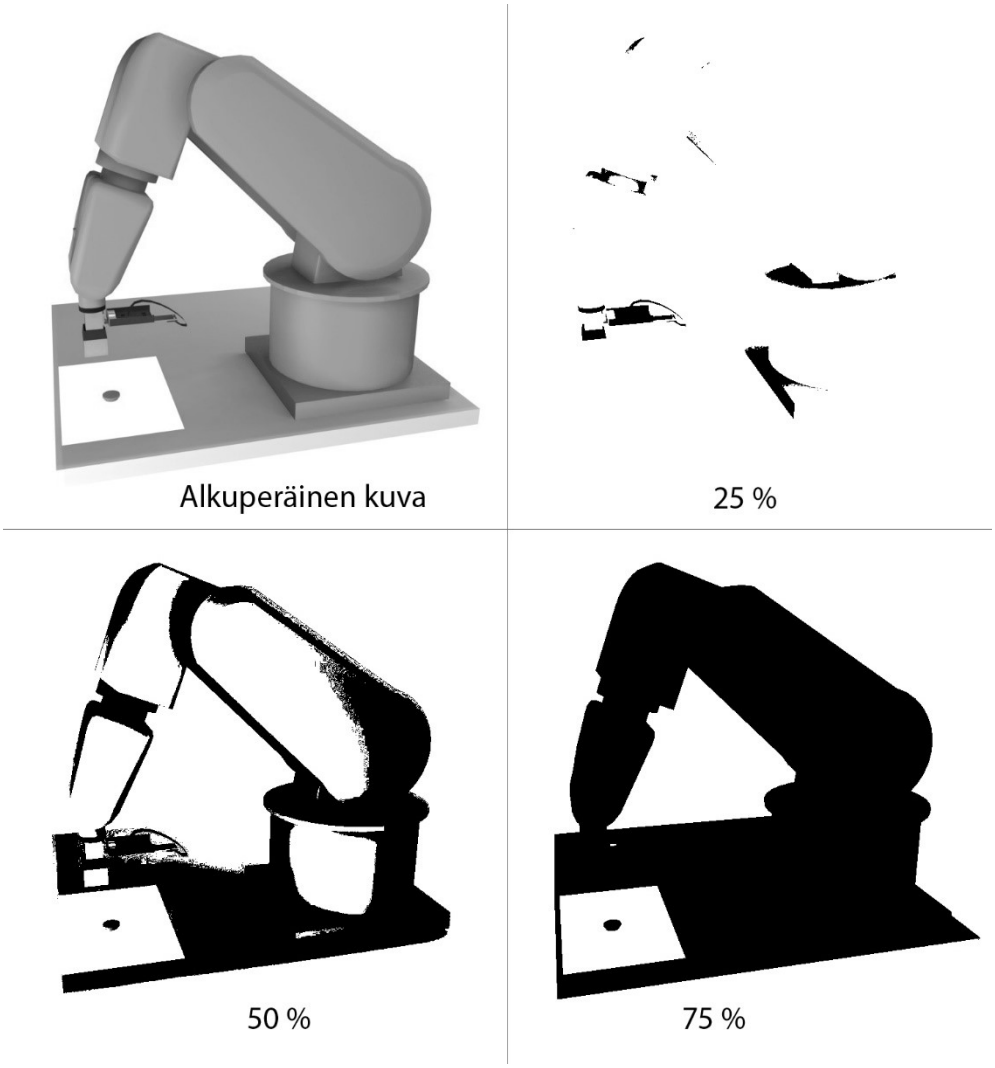
Kuvan ongelma voidaan pääsääntöisesti korjata muuttamalla otettavien kuvien, tai kuvattavan videokuvan suljinaikaa 1/50 sekuntiin tai käyttää luvun 50 kerrannaisia, jolloin sekä kameran suljinaika sekä verkkovirran taajuus ovat ”synkronoituja” keskenään ja ongelma poistuu.

Sivulta valaistessa kuvan valonlähdettä lähempänä oleva reuna ns. ylivalottui toisen reunan jäädessä tummemmaksi. Tällöin kappaleentunnistus ei kykenisi erottamaan kappaletta tummemmaksi jäävästä reunasta, saatikka kappaleen luomasta todella terävästä varjosta. Varjoja ja valon yleistä tasaisuutta saatiin hieman parannettua sijoittamalla valaisimen vastakkaiselle puolelle valoa heijastava levy, mutta riittävän tasalaatuista lopputulosta ei tällä keinoin kyetty saavuttamaan. Apuun otettiin lisäksi halpoja led-retkivalaisimia, joita sijoittamalla eri puolille työaluetta kyettiin saavuttamaan riittävän laadukas työalueen valaistus. Valaisimien valo jakautui kuitenkin niin kapealle alueelle, että näidenkään valaisimien käyttö ei olisi järkevää. Vaikka valoa oli nyt riittävästi saatavilla, sen jakautuminen tuotti päänvaivaa. Tällöin kappaleentunnistusalgoritmi saattaisi toimia kuvan toisessa nurkassa paremmin kuin toisessa, joka ei ollut tyydyttävä ratkaisu. Toistaiseksi parhaita tuloksia oli saavutettu käyttämällä pelkästään luokkahuoneen yleisvalaistusta, mutta kappaleen tunnistuksessa oli vielä parantamisen varaa. Ongelmaa oli lähdettävä ratkomaan toisesta näkökulmasta.

4.9 Kappaleentunnistus todellisella kohteella

Kappaleentunnistus toimi hyvin, kun kyseessä oli valkoiselle pohjalle piirretty suurikontrastinen kappale. Työ oli kuitenkin edennyt jo siihen vaiheeseen, että testikuvia ei enää tarvittu, vaan voitiin ottaa oikeita valokuvia oikeista kappaleista. Aiemmin läpikäyty valaistuksen hankaluus aiheutti harmia, kun aiemmin hyvin toimiva ääriviivojen tunnistus ei tositilanteessa kyennytkään tunnistamaan haluttua kappaletta tai tunnisti kappaleeksi myös sen varjon, jolloin lasketttu keskipiste meni automaattisesti väärin. Tunnistuksen herkkyyttä voitiin säätää, mutta tämäkin keino aiheutti ongelmia, kun toisella puolella kuvaa tunnistus toimi hyvin, toisella huonosti. Uusia keinoja tunnistuksen parantamiseksi piti siis keksiä.

Pythoniin lisättävä imutils-kirjasto mahdollistaa useita kuvalle tehtäviä kuvankäsittelyoperaatioita [29]. Ajatuksena oli, että kuvan kontrastia kasvattamalla kappale erottuisi riittävän selkeästi ääriviivojen täsmällistä tunnistamista varten. Sama ongelma kuitenkin seurasi kuin aiemminkin; siinä missä kappale erottuu valkoista taustaa vasten paremmin, erottuvat paremmin myös sen varjot. Ratkaisu löytyi OpenCV-kirjaston binäärikuvanluontitoiminnosta. Binäärikuvalla tarkoitetaan kuvaa, jossa kaikkia kuvan pikseleitä kuvataan vain kahdella arvolla, ykkösellä tai nollalla (mustalla ja valkoisella). Jokaisen yksittäisen pikselin kirkkaus on arvo väliltä 0–255. Nolla tarkoittaa täysin mustaa ja 255 täysin valkoista [30, s. 25–26]. Kuvaan voitaisiin nyt määritellä se kirkkausarvo, jota tummemmat pikselit muutettaisiin täysin mustiksi ja vastaavasti kirkkaammat täysin valkoiseksi. Muuttamalla tätä raja-arvoa saataisiin kappale näkymään tarpeeksi selkeärajaisena ääriviivojen tunnistusta varten. Raja-arvon muutoksen vaikutusta on havainnollistettu kuvassa 25.



Kuva 25. Havainnekuva binäärimuunnoksen vaikutuksesta raja-arvon muuttuessa

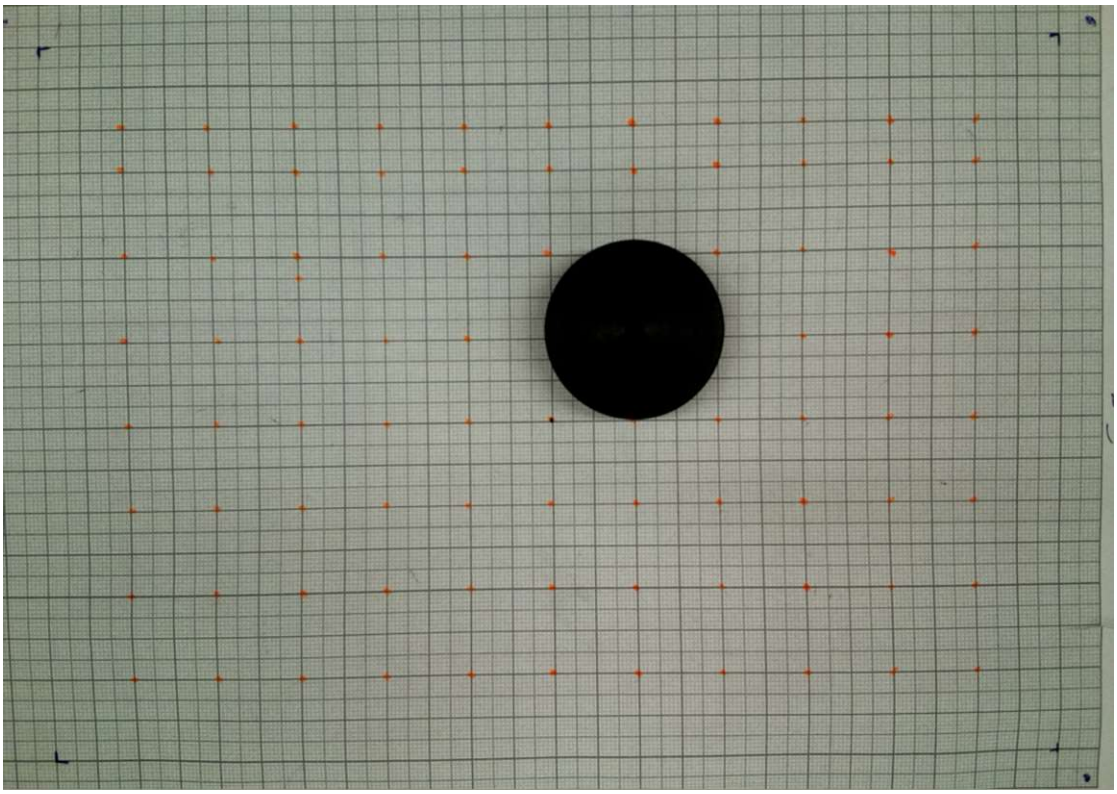
OpenCV-kirjastossa binäärikuvan luonti tapahtuu cv2.threshold-komennolla, joka on esitetty kuvassa 26.

```
img_binary = cv2.threshold(img_grey, thresh, 255, cv2.THRESH_BINARY)[1]
```

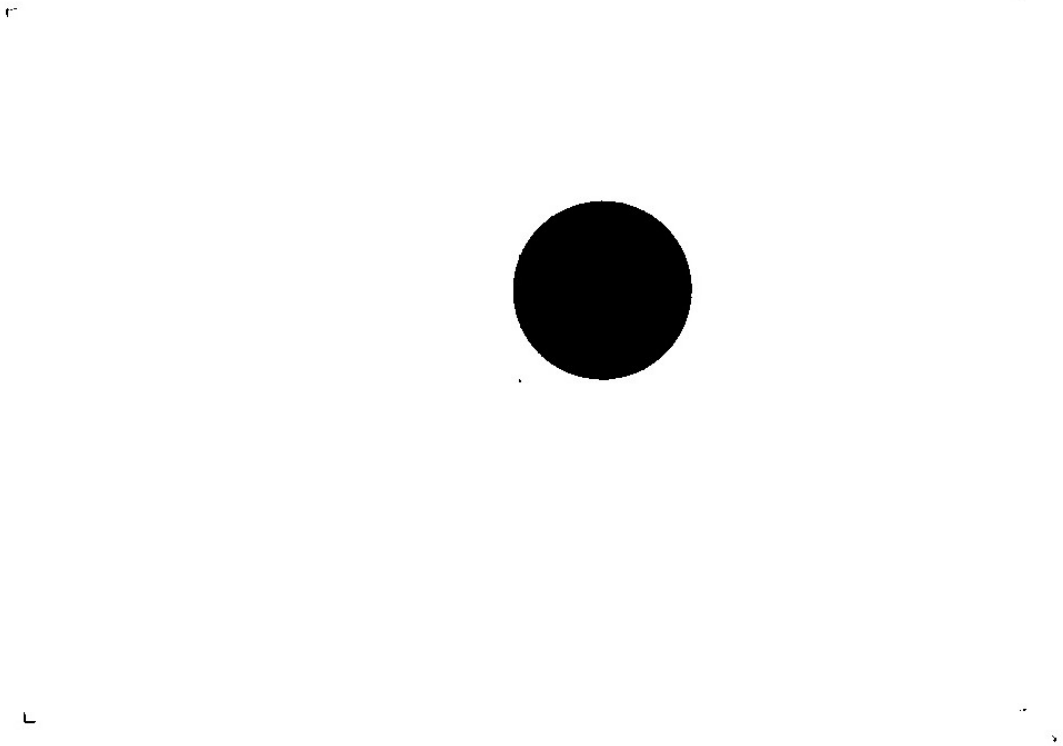
Kuva 26. Binäärikuvan luonti Python-koodissa

Koodiin on aiemmin lisätty muuttuja ”thresh”, johon ylläolevassa koodinpätkässä viitataan. Tätä arvoa muuttamalla voidaan säätää kappaleen näkyvyyttä taustaa vasten. Kokeilemalla saatiin selville, että parhaiten toimiva arvo käyttäen valaistukseen pelkkää luokan yleisvalaistusta oli 20. Tällä arvolla kappaleen ääriviivat tunnistuivat hyvin ja sen varjot jäivät melko hyvin raja-arvon yläpuolelle, jolloin ne eivät vaikuttaneet keskipisteen laskentaan. Pientä vaihtelua työalueella oli, mutta huomattiin, ettei niistä ole haittaa kappaleen keskipisteen

oikeaoppiseen löytymiseen. Kuten kuvista 27 ja 28 voidaan todeta, binäärikuvan muodostamisen jälkeen jäljellä on lähestulkoon täydellisesti piirtyvä kappale, josta ääriviivat voidaan tarkasti paikantaa.



Kuva 27 Noudettava kappale kuvattuna ennen muokkauksia



Kuva 28. Kuva käännettynä binäärimuotoon

Kuten kuvasta 28 voidaan todeta, binäärimuunnoksen jälkeen kuvaan jää pieniä mustia kohtia – selkeimmin näkyviin jäävät työalueen rajoja merkitsevät kulmamerkit. Lisäksi kuvassa voi näkyä roskia yms. ylimääräistä, joka voi aiheuttaa ongelmia ääriviivojen haussa. OpenCV-kirjaston ääriviivojen noutoalgoritmi ei lähtökohtaisesti erottele löytämiään kappaleita muutoin kuin niiden hierarkian mukaan, jolloin binäärikuvasta löytyy tarvittaessa ääriviivat haettuna jokaiselle roskalle ja kulmamerkille. Haluttu hierarkia voidaan määritellä siten, että esimerkiksi kappaleessa, jonka keskellä on reikä, voidaan ulkoreunojen ääriviivat määritellä hierarkiassa ylemmäksi kuin reiän ääriviivat. [31.] Tässä työssä kappaleiden hierarkiaa ei tarvinnut erikseen muuttaa, riittäisi, että roskat ja muut virheelliset havainnot suodatettaisiin pois. Tätä tarkoitusta varten ääriviivat tulisi ensin järjestellä niiden koon mukaan suurimmasta pienimpään. Tämän toteutus näkyy kuvassa 29, jossa contourArea-komennolla voidaan hakea löydetyt kappaleet ja niiden pinta-alat.

```
# Löydetyt alueet järjestellään pinta-alan perusteella
cnts = sorted(cnts, key = cv2.contourArea, reverse = True)[:3]
```

Kuva 29. Löydettyjen alueiden järjestely niiden pinta-alan mukaan

Rivin lopusta löytyvä [:3]-merkintä viittaa siihen, kuinka monta kappaletta tultaisiin tallentamaan muuttujaan ”cnts”. Järjestys ja pinta-alat tallennettaisiin Python-kielen listaan, joista neljä suurinta kappaletta tallennettaisiin muuttujaan. OpenCV:n findContours-komennolla haettaessa suurin löydetty alue on aina koko kuvan alue. Tämä tallentuu järjestelyn jälkeen listan indeksiin nolla. Pythonin tapa hallita listaobjekteja on aloittaa lista numerosta nolla, johon sitten lisätään uusia arvoja kasvavalla indeksiarvolla. Tällöin toiseksi suurin löydetty kappale, joka on aina haettava kappale ja joka löytyy siis tallennetun listan indeksistä yksi, on se kappale, jonka keskipiste halutaan löytää. Toisinaan voi kuitenkin esiintyä tilanne, jossa yhtään kappaletta ei ole pöydällä, jolloin toiseksi suurin objekti kuvassa olisi mahdollisesti jokin kulmamerkeistä. Tällöin robotille lähtisi kulmamerkin keskipisteen koordinaatit, jota sitten virheellisesti yritettäisiin tuloksetta noutaa. Tätä varten koodiin lisättiin tarkastuslauseke, jossa käytetään samaa contourArea-komentoa kuin aiemminkin. Tässä kohtaa sitä vain käytettäisiin suodattamaan pois sellaiset alueet, jotka ovat selkeästi väärän kokoisia oletettuun kappaleen kokoon nähden. Tällöin, mikäli oikean

kokoista kappaletta ei löytyisi tallennetun listan indeksistä yksi (eli listan toiseksi suurin kappale), ohjelmakierto muuttuisi siten, että koodi odottaa viisi sekuntia, jonka jälkeen otetaan uusi kuva ja suoritetaan tarkastus uudelleen. Tätä testausta jatkettaisiin niin kauan, kunnes määrätyn kokoinen kappale löytyisi ja sen koordinaatit voitaisiin lähettää robotille noutoa varten. Ehtolauseke esitetty kuvassa 30.

```
area = cv2.contourArea(cnts[1])
# Tarkastetaan kappaleen koko
if 15000 < area < 23000:
    #print('Kappaleen pinta-ala: ', area)
    pass
else:
        print('Ei kappaleita, yritetään uudelleen viiden sekunnin päästä.')
        time.sleep(5)
        kuvan_otto()
        kuvan_lataus_ja_muokkaus()
        aariviivat()
```

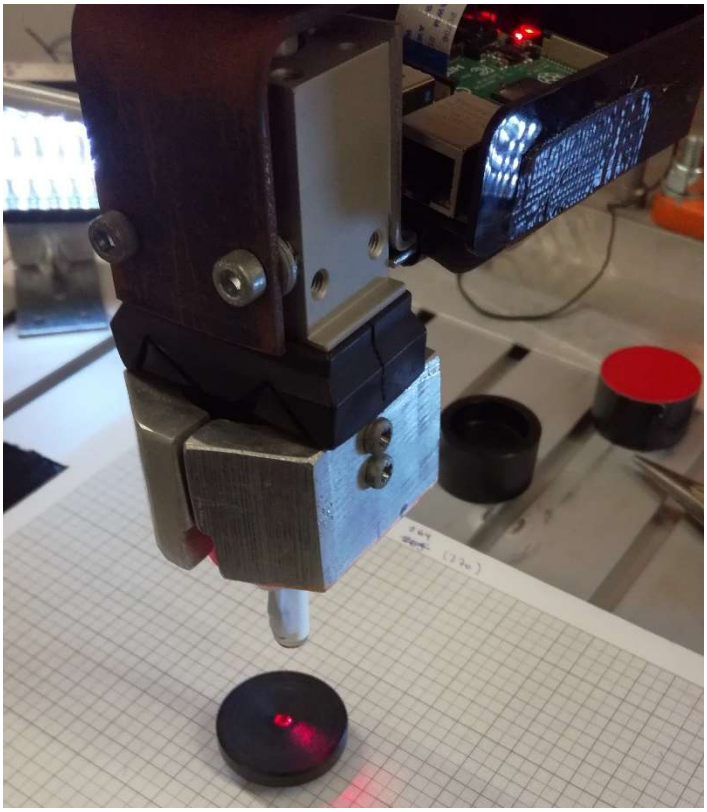
Kuva 30. Kappaleen pinta-alan tarkistus

Kyseinen ehtolauseke pinta-alan tarkkailuun toimi testien aikana käytännössä virheettömästi ja sen avulla työalueelle asetetut tahallaan väärät kappaleet (esim. kynä, kumi) suodatettiin ongelmitta pois, eikä ne siten vaikuttaneet laitteiston toimintaan.

4.10 Virheiden kompensointi

Kuten aiemmassa vaiheessa on todettu, kappaleen tunnistuttua sen koordinaattien avulla käsisyötöllä kohteeseen ajettu robotti pääsi melko lähelle todellista kohdettaan. Virhettä kuitenkin oli akseleittain paikoin reilustikin (± 1 cm), joten sitä pitäisi kyetä tavalla tai toisella kompensoimaan. Virhemarginaalin ollessa noin ± 2 millimetriä pitäisi arvoja joka tapauksessa saada korjattua useita millimetrejä. Aluksi virhe näyttäytyi kasvavana kutakuinkin kuvan keskikohdasta ulospäin. Testit olivat osoittaneet, että virhettä esiintyi enemmän työalueen oikeassa reunassa kuin vasemmassa, joten ajatus työalueen jakamisesta esim. keskikohdan suhteen neljään alueeseen ja laskemalla kompensoinnin jokaiselle eri alueelle voisi olla mahdollinen tapa korjata esiintyvää virhettä. Ennen tarkempia tutkimuksia tämä oli kuitenkin vain arvailua. Tätä varten piti kehittää keino laskea aiheutuva virhe mahdollisimman tarkasti.

Virheen toteamiseksi eri puolilla työaluetta päätettiin alue jakaa yksittäisiin pisteisiin ja laskea virheen määrä kussakin pisteessä. Tähän käytettiin apuna millimetripaperia, johon merkittiin kymmeniä testipisteitä. Apuna virheen mittaamisessa robotin työkaluun kiinnitettiin laservalo, joka sijoitettiin mahdollisimman suoraan työkalun keskikohtaan osoittamaan työalueen pintaa. Haettava kappale asetettiin työalueelle määrättyihin pisteisiin, ja kameralla otetulla kuvalla kappaleen keskipisteen koordinaatit merkittiin ylös ja ajettiin robotti kuvan ilmoittamaan pisteeseen. Tätä toimintaa on havainnollistettu kuvassa 31.



Kuva 31. Laservalon käyttö kompensointitarpeen selvityksessä

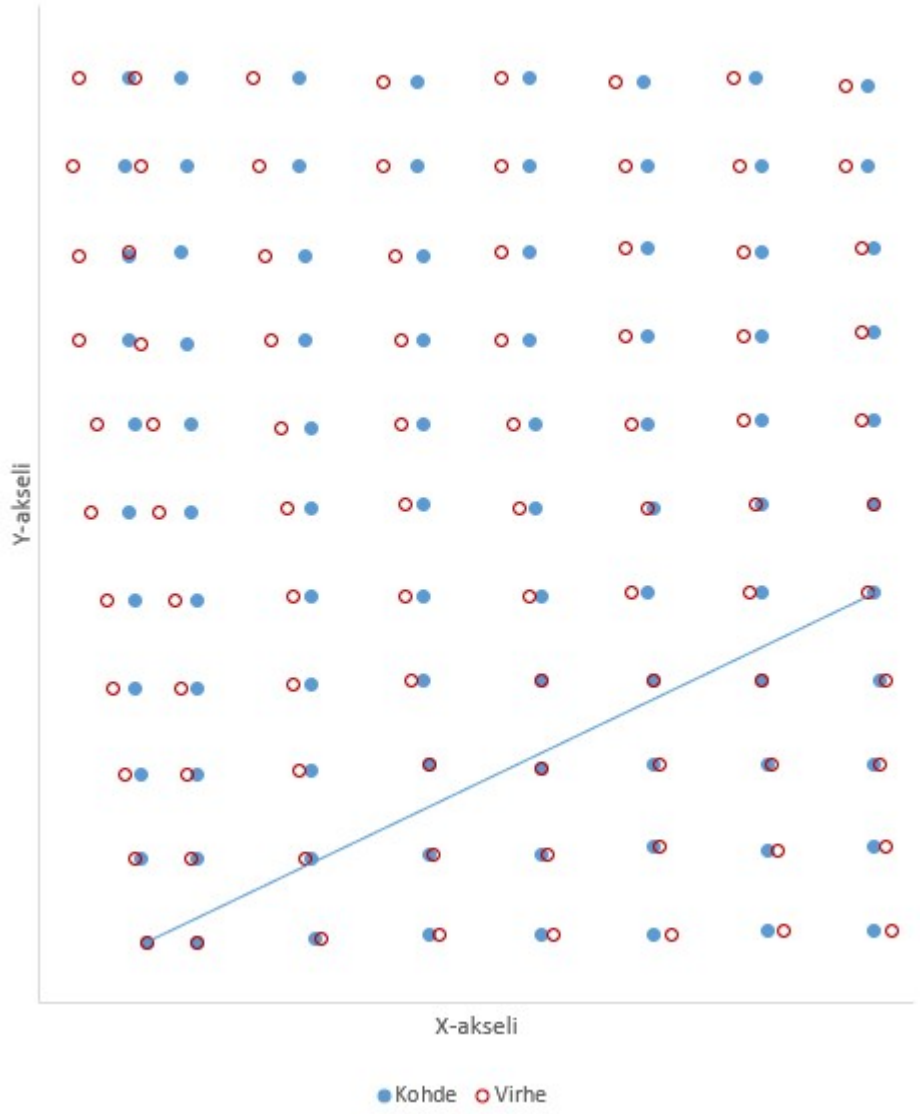
Kappale poistettiin työalueelta ja laservalon osoittaman pisteen perusteella laskettiin kuvan koordinaatin ja robotin ajaman position välinen virhe. Tämä virhe tulisi kompensoida pois, jotta robotti osaisi ajaa täsmällisesti noudettavan kappaleen päälle noudon onnistumiseksi. Sama testirupeama suoritettiin kymmenille eri pisteille työalueella. Tulokset kirjattiin ylös ja lopulta niistä voitiin tehdä kuvassa 32 esitettävä pistemäinen kartta. Selvyyden vuoksi tekstissä puhutaan pisteiden sijainnista kuvan suhteen, ei robotin koordinaatistossa.

Kuva 32. Virheen suunta ja etäisyys

Kuten kuvasta 32 voidaan todeta, pienin virhe ei sijainnutkaan työalueen keskellä, vaan jonkin verran sen alapuolella. Tästä voitiin päätellä, että kameran optiikasta aiheutuneet vääristymät eivät olisi erityisen merkitseviä – tällöin virheen tulisi kasvaa melko lineaarisesti keskikohdasta ulospäin, mutta tässä tapauksessa niin ei käynytkään. Kun virhe jaetaan akseleittain, nähdään, että x-akselin suhteen virhettä löytyy eniten kuvan vasemmasta yläreunasta ollen siellä suurimmillaan noin yhdeksän millimetriä. Sieltä virhe pienenee ja lopulta vaihtaa suuntaansa vasemmalta puolelta oikealle kuvan oikeaa alareunaa lähestyttäessä. Y-akselin suhteen virhe on suurimmillaan sen sijaan kuvan oikeassa yläreunassa ollen noin kuusi millimetriä – suunta vaihtuu jälleen kuvan

alareunaa kohti edetessä. Jotta virheen suunnasta akseleittain saataisiin parempi käsitys, luotiin vastaavat kuvaajat, jossa virhe on jaettu esittämään vain yhtä akselia kerrallaan. Kompensointi tulisi joka tapauksessa tehdä akseleittain, joten tällainen muunnos osoittautui erittäin havainnolliseksi.

Tarkastellessa virheen määrää ja suuntaa x-akselin osalta huomattiin, että virheen leikkauskohta löytyi suurin piirtein kuvan oikean keskikohdan ja vasemman alakulman väliltä. Tämän linjan yläpuolella olevat virhearvot kasvaisivat vasemmalle ja alapuolella olevat oikealle, kuten kuvasta 33 voidaan huomata.



Kuva 33. Virheen määrä ja suunta x-akselin suhteen sekä nollavirhelinja

Tätä akseleittain jakoa voitaisiin hyödyntää kompensoinnin laskennassa. Edessä oli nyt kaksi ongelmaa; ensin pitäisi määrittää virheen suunta, eli se, kummalla puolella ns. nollalinjaa ollaan, sekä varsinainen kompensoinnin määrä kyseisessä pisteessä. Kuvasta nähtiin, että virhe kasvaa kutakuinkin suoraviivaisesti maksimiarvoa lähestyttäessä, tämä tieto helpottaisi laskentaa. Virheen suunta olisi hankalaa määrittää pelkin ehtolausekkein ohjelmoimalla, joten apuna voitaisiin käyttää matemaattista tarkistusta. Tätä tarkoitusta varten tuli ensin määrittää suoran yhtälö nollalinjalle. Kyseessä ollessa normaali numeerinen koordinaatisto saatiin suoran koordinaatit valittua linjalta sieltä, missä virhettä oli vähiten. Virheen havaitsemiseksi luodulle määritystavalle arvioitiin virhemarginaaliksi ± 1 millimetri, joten virheen ollessa välillä -1–1 millimetriä akselilla voitiin virheen uskoa olevan siinä määrin vähäistä, että se menisi mittausepätarkkuuden piikkiin. Tällöin nollalinjalla olevissa pisteissä voisi pientä heittoa olla ilman, että se vaikuttaisi ratkaisevasti kompensointilaskelmien tulokseen. Suora määriteltiin kuvan vasemman alareunan ja oikean reunan viidenneksi ylimmän pisteen välille. Näin saatiin aikaan suoran lauseke x-akselin virheen kompensointia varten. Suoran lauseke esitettynä kaavassa 4. [32.]

$$y = \left(\frac{88}{129}\right) * (-425) \tag{4}$$

Tätä lauseketta käyttämällä kompensointiarvo voitaisiin laskea, kun pisteen etäisyys ja sijainti suoran suhteen tunnettaisiin. Vaikka suoran yhtälö pysyisikin samana puolta vaihdettaessa, pisteen maksimietäisyys ja kompensoinnin etumerkki vaihtuisivat. Tämän vuoksi kompensointilaskuja ei voitaisi yhdistää, vaan ne tuli pitää erillisinä laskuina suoran ylä- ja alapuolella.

Pisteen sijainti suoran suhteen voitiin määrittää ristitulon avulla [33; 34]. Mikäli yhtälön tulos on pienempi (tai yhtä suuri) kuin nolla, piste sijaitsee suoran alapuolella ja toisinpäin. Pisteen sijainti voidaan määrittää yhtälöstä 5, jossa x- ja y-arvot ovat tutkittavan pisteen koordinaatit x1-, x2-, y1- ja y2-pisteiden ollessa kuvaan määritetyn suorayhtälön pisteiden koordinaatit.

$$d = (x - x_1)(y_2 - y_1) - (y - y_1)(x_2 - x_1) \tag{5}$$

Pisteen sijainnin määrityksen jälkeen voitiin ohjelmaan luoda ehtolauseke, jolla tutkittavaa pistettä voitaisiin kompensoida oikea määrä oikeaan suuntaan. Tätä varten tulisi ensin kuitenkin laskea pisteen etäisyys suorasta. Etäisyyden laskenta on esitetty yhtälössä 6.

$$d = \frac{|ax_0 + by_0 + c|}{\sqrt{a^2 + b^2}} \tag{6}$$

Muuttamalla yhtälössä 4 esiintyvä suoran lauseke nollamuotoon saadaan siitä muodostettua yhtälö, joka esitetään kaavassa 7.

$$88x - 129y - 54825 = 0 \tag{7}$$

Kun nämä arvot sijoitetaan yhtälöön 6, voidaan ratkaista pisteen etäisyys x-akselin suhteen ja käyttää saatua arvoa kompensoinnin laskemiseen. Laskennassa tarvitaan myös pisteen maksimietäisyyttä suoran suhteen. Tällä tarkoitetaan työalueelle sijoittuvaa pistettä, joka on mahdollisimman kaukana suorasta (tarkasteltavalla puolella), eli kuvassa 33 tämä tarkoittaisi kuvan vasenta yläreunaa. Kun kyseisen pisteen koordinaatti sijoitetaan yhtälöön 6, saatiin tulokseksi 180,8 millimetriä. X-akselin nollasuoran yläpuolelle jäävän osan maksimivirhe on yhdeksän millimetriä – myös tätä arvoa tarvittaisiin kompensoinnin laskennassa apuna. Lopullinen kompensointiarvo tultaisiin laskemaan yhtälössä 8 esitetyllä tavalla.

$$Koordinaattipiste \pm \left(\frac{Pisteen\ etäisyys\ suorasta}{Pisteen\ maksimietäisyys}\right) * Maksimikorjaus \tag{8}$$

Näin voitaisiin teoreettisesti oikea kompensointiarvo saada laskettua, kun tiedossa oli tutkittava koordinaattipiste sekä pisteen sijainti tutkittavan suoran suhteen. Mikäli piste sijoittuisi x-akselilla suoran alapuolelle, yhtälöt säilyisivät muuten samoina, mutta pisteen maksimietäisyys ja maksimikorjaus muuttuisivat niille laskettuihin vastaaviin arvoihin.

Y-akselilla tilanne on samankaltainen, eli robotin alkuperäisen noutokoordinaatin ja halutun koordinaatin välillä esiintyy virhettä. Y-akselilla virheen

suunta on vain erisuuntaista x-akseliin verrattuna. Tämä käy selväksi kuvasta 34.

Kuva 34. Virhe y-akselin suhteen, sekä sille piirretty nollavirhelinja

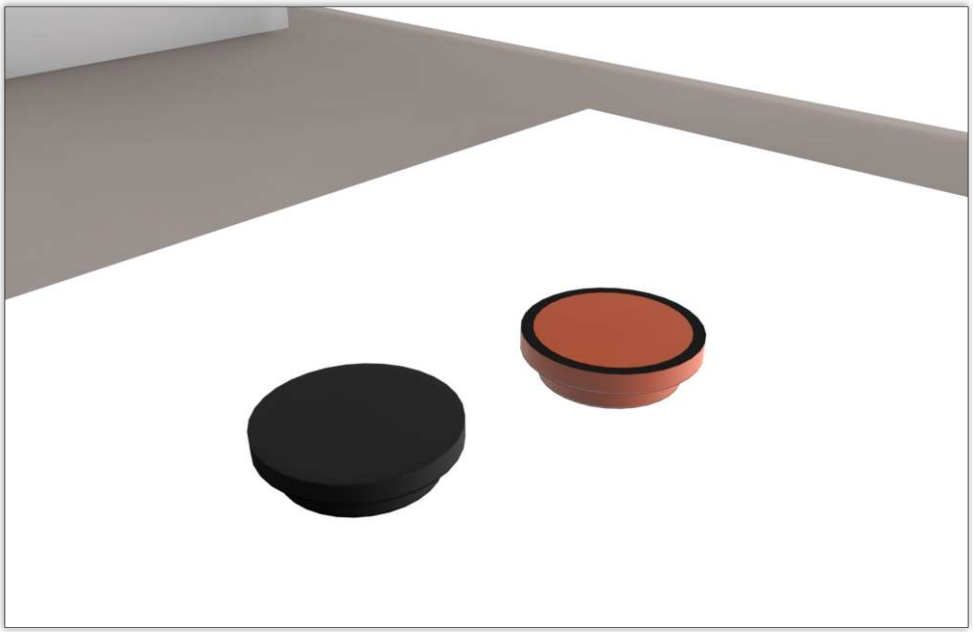
Nollavirhelinja asettuu y-akselilla siis päinvastoin kuin x-akselilla, eli suurin piirtein tämän suoran alueella robotti kulkee y-akselin suhteen suoraan oikeaan pisteeseen, eikä virhettä esiinny. Kun piste etääntyy suoralta, alkaa virhettä syntyä. Suoran yläpuolella, kuvan 34 oikeassa yläreunassa maksimivirheeksi mitattiin kuusi millimetriä. Y-akselille tulisi määrittää – samoin kuin aiemmin x-akselillekin – nollalinjan suorayhtälö, suorayhtälön nollayhtälö sekä maksimietäisyydet puolittain suorasta ja maksimivirheet. Samoja käytäntöjä noudattaen, kuten aiemmassakin vaiheessa x-akselilla, voitiin laskutoimitukset tehdä myös y-akselillekin ja lisätä ohjelmakoodiin. Näiden valmistuttua voitai-

siin kompensointiarvojen vaikutusta päästä ensikerran tositilanteessa kokeilemaan. Yllätys oli varsin suuri, kun huomattiin, että robotti kompensoinnin jälkeen ajaa itsensä lähes täydellisesti haluttuun positioon suoraan noudettavan kappaleen ylle. Hankalasta alkutilanteesta huolimatta kompensointi toimi tarkasti. Tämä oli yllättävää siksi, että kaikki alun mittaukset oli tehty käsin, kappaleen asettelusta virheen laskemiseen.

4.11 Värintunnistus ja lajittelu

Kompensointilaskelmien valmistuttua jäljellä oli käytännössä enää hienosäätöä ja testausta. Robotti haki nyt mustia kappaleita pöydältä suurella tarkkuudella ja toimitti ne pois työalueelta, jonka jälkeen jäätäisiin odottamaan uutta kappaletta. Työhön päätettiin tässä kohtaa toteuttaa mahdollisuuksien mukaan myös lajittelutoiminto kappaleen värin perusteella. Käytössä oli lähtökohtaisesti kolmen värisiä kappaleita – mustia, oransseja ja hopeisia. Työssä päädyttiin rajaamaan lajittelu kuitenkin vain mustiin ja oransseihin. Ainut ongelma oranssien kappaleiden suhteen oli niiden huono ”näkyvyys” kameralle säätämättä aiemmin kappaleessa 6.8 käsiteltyä threshold-arvoa, jolla määritetään binäärikuvan mustien ja valkoisten pikseleiden raja-arvo. Oranssin kappaleen ollessa paljon vaaleampi mustaan verrattuna jäisi se usein binäärikuvamuutoksen jälkeen täysin valkoiseksi. Toisaalta jos threshold-arvoa nostaisi liiaksi, alkaisi kappaleen varjo ja taustan mahdolliset epätasaisuudet ym. häiriötekijät haittaamaan kappaleen reunojen tunnistusta. Tästä syystä kyseinen arvo päätettiin pitää paikallaan ja keksiä ongelmaan vaihtoehtoinen ratkaisu. Lopulta ongelma ratkaistiin värjäämällä oranssin kappaleen ulkoreunat mustiksi, koodihan tarkkailee joka tapauksessa vain *kappaleiden ääriviivoja*. Siksi musta ulkoreuna ei vaikuttaisi värintunnistukseen. Tällainen tilanne ei suurella todennäköisyydellä olisi mahdollista oikeassa, esimerkiksi teollisuuden käyttöön tulevassa konenäkösovelluksessa, mutta koska tässä työssä tällaisia rajoitteita

ei ollut, voitiin hyvin käyttää tällaista keinoa uuden ominaisuuden lisäämiseksi. Kuvassa 35 on esitetty mallinnukset valmiissa työssä käytetyistä kappaleista.



Kuva 35. Noudettavat kappaleet

Värintunnistus tapahtuisi määrittämällä havaitun, oikeankokoisen kappaleen keskipisteen tummuusarvo binäärikuvassa. Mustalla tämä olisi aina nolla, kun taas oranssin kappaleen tumman kehän sisäpuolelle jäävän osuuden arvo olisi aina 255, eli täysin valkoinen. Näin voitaisiin hyvin yksinkertaisella testillä määrittää otetusta ja sitten muokatusta kuvasta sen väri. Kun väri olisi määritetty, kappale lajiteltaisiin robotin x-akselin koordinaattia muuttamalla viemällä se pöydällä sille määrättyyn pisteeseen. Python-koodissa tunnistus ja lajittelu on esitetty kuvassa 36.

```
värintunnistus = cv2.imread('imagebw.jpg', 0)
väri = värintunnistus[cy,cx]
if väri == 0:
    print('Musta palikka')
    x_poisto = 490
else:
    x_poisto = 410
    print('Oranssi palikka')
```

Kuva 36. Värintunnistus ja lajittelu

Koodissa esiintyvä muuttuja ”x_poisto” määrittää robotille lähetettävän x-akselin koordinaatin. Työalueen reunalle määriteltiin sijainnit kummallekin eri värille, johon robotti ne sitten värin havaittuaan ja kappaleen työalueelta noudettuaan toimittaa. Käytännön testeissä värintunnistus toimi sadan prosentin tarkkuudella, eli kappale toimitettiin jokaisella kerralla oikeaan pisteeseen.

4.12 RAPID-ohjelmakoodin tarkastelu

Tähän asti opinnäytetyössä on keskitytty lähes täysin pelkkään Python-ohjelmointikielellä toteutettuun koodiin. Tämä johtuu siitä, että ”työnjako” koodien välillä toimi siten, että Raspberry Pi hoitaa kuvan ottamisen ja koordinaattien laskennan valmiiksi, jolloin robotin tehtäväksi jää ainoastaan muokata saamansa koordinaatit ymmärrettävään muotoon, noutaa kappale ja ilmoittaa, kun uusi kuva voidaan taas ottaa robotin palattua alkuasentoonsa. RAPID-koodin osuus on tästä syystä huomattavasti pienemmässä roolissa. Robotin kontrollerin käsittelemä koodi voidaan karkeasti jakaa muutamiin eri osa-alueisiin, jotka esitettynä kuvassa 37.

Kuva 37. Robotin tehtävät

Ensiksi robotille voitiin määritellä niin sanotut vakiokohteet, joiden koordinaatit eivät ohjelmakierron aikana muutu. Tällaisia ovat muun muassa kuvanottoposition ja muut tarvittavat pisteet. Näitä pisteitä tarvittiin työn aikana laitteen testaamiseen erilaisissa tilanteissa. Lopulliseen työhön ei kuitenkaan lopulta tällaisia vakiopositioita tarvittu kuin aiemmin mainittu kuvanottopositio. Muut pisteet määräytyisivät ainakin osin Raspberry Piltä tulevista koordinaateista. Esimerkki tällaisesta pisteestä on lajittelupositio. Position y- ja z-koordinaatit voitiin määritellä valmiiksi, koska näiden arvot eivät muuttuisi, ainoastaan x-akselin koordinaatti muuttuisi sen mukaan, onko kyseessä musta vai oranssi kappale. Vastaavia positioita, joissa ainakin yhden akselin koordinaatti muuttuisi kappaleen sijainnin perusteella olisi useita. Nämä kohteet odottaisivat koordinaatteja tietokoneelta, jonka jälkeen niihin voitaisiin robotin toimesta ajaa.

Kuten Python-koodissakin, myös RAPID:ssa toimii socket-yhteydenmuodostusmenetelmä. Vakiomuuttujien määrittelyn jälkeen molemmissa koodeissa suoritettaisiin yhteydenmuodostus, jonka jälkeen kommunikointi laitteiden välillä olisi mahdollista. Yhteyden muodostaminen kuului RAPID-koodissa niin sanottuihin alkutoimiin, jonka jälkeen robotti ajaisi itsensä kuvanottopositioon ja vapauttaisi mahdollisesti aiemmin jääneen paineen työkalusta, jolloin se palautuisi auki-asentoonsa. Tätä vaihetta ei enää saman ohjelmakierron aikana tehtäisi uudestaan, vaan seuraavaksi robotin RAPID-koodi siirtyisi seuraavaan osioon, jossa se odottaisi Raspberry Piltä koordinaatteja kappaleen noutoa varten.

Kuvanottopositiossa ollessaan robotti siis odottaisi, että Raspberry Pi ottaisi työalueesta kuvan ja lähettäisi muunnetut koordinaatit robotille. Koordinaatteja saapuisi yhteensä viisi kappaletta. X-, y-, z-, z-ylä- ja x-poisto -koordinaatit kertoisivat robotille kaikki tarvittavat tiedot kappaleen noutoa varten. Z-ylä-koordinaatilla tarkoitetaan robotin työkalun korkeusasemaa kappaleen yllä. Robotti ajettaisiin kappaleen ylle muutamien senttimetrien etäisyydelle ja laskeuduttaisiin suoraan alaspäin noutamaan kappale kyytiin. Tämä parantaisi kappaleen noudon onnistumisprosenttia merkittävästi, kun riskiä siitä, että työkalu siirtäisi noudettavaa kappaletta robotin mennessä sitä hakemaan, ei olisi. X-poisto-koordinaatilla viitataan kappaleen värintunnistuksessa määritettyyn, värin perusteella tehtyyn lajittelupositioon. Saapuessaan koordinaatit tallen-

nettaisiin niitä varten alussa määritettyihin muuttujiin ja lähetettäisiin vastaanotettu arvo takaisin Python-koodille. Näin voitaisiin tarkistaa, että lähetetty arvo on tallentunut varmasti oikein, eikä robotti tämän vuoksi ajaisi itseään väärään paikkaan. Testausvaiheessa esiintyi ongelma, jossa lähetetyt koordinaatit saattoivat tallentua väärin robotin muuttujiin. Esimerkkinä x-koordinaatin arvon ollessa 100 ja y-koordinaatin 200 saattoi robotin x-koordinaatiksi tallentua virheellisesti 1002 ja y-akselille 00, eli Python-koodista tulevat arvot menivät vastaanotettaessa osin ”päällekkäin”. Tämä ongelma poistui, kun lähetysten välille lisättiin yhden sekunnin mittainen tauko. Kun arvot olivat tallennettu muuttujiin, ne muutettaisiin uusiin muuttujiin, jotta robotti pystyisi käyttämään niitä koordinaatteina. Tämä toteutettiin StrToVal-komennolla. Python-koodista koordinaatit saapuivat tekstinä (string), jotka sitten muutettaisiin numeeriseen muotoon (value). Kun kaikki arvot oli muunnettu numeerisiksi, voitiin ne lopulta tallentaa positioiksi robotille.

Nyt kaikki tarvittavat arvot olivat käytettävissä, jolloin jäljellä olisi käytännössä enää kappaleen nouto ja lajittelu. Robotin liikkeet muodostuivat yhteensä neljästä eri positiosta, joiden välillä robotin kaikki liike tapahtui. Kappaletta noudettaessa ajetaan ensin sen päälle, jonka jälkeen liikutaan suoraa alaspäin noutamaan kappale kyytiin. Alhaalla työkalun paineilma kytketään päälle, jolloin sen leuat puristuvat kappaleen ympärille ja sitä voidaan liikutella työkalun mukana. Kappale nostetaan saman pisteen kautta ylös, minkä kautta tultiin alaskin. Tämän jälkeen robotti ajaa lajittelupisteelle, jonka päällä paine vapautetaan ja kappale putoaa värinsä perusteella lajiteltuun pisteeseen. Lopulta robotti ajaa itsensä kuvanottopositioon takaisin. Liikkeen loputtua lähetetään Raspberry Pille ”VALMIS”-viesti, jolloin kierto voidaan aloittaa alusta ja ottaa uusi kuva tutkittavaksi. RAPID-koodin osalta robotin liikkeet on esitetty kuvassa 38.

```
MoveL rtemp_ylä,v300,z100,bintool\WObj:=wobj0;      ! Objektin yllä
MoveL rtemp,v50,fine,bintool\WObj:=wobj0;           ! Objektin nouto
Set DO1;                                            ! Paineilma päälle, leuat kiinni, kappale kyytiin
WaitTime\InPos, 0.3;
MoveL rtemp_ylä,v300,z200,bintool\WObj:=wobj0;      ! Ylös, kappale mukana
MoveL lajittelu,v300,fine,bintool\WObj:=wobj0;      ! Lajittelu
WaitTime\InPos, 0.2;
Reset DO1;
WaitTime\InPos, 0.5;
MoveL kuvan_otto,v300,fine,tool0\WObj:=wobj0;       ! Takaisin alkuun
WaitTime\InPos, 0.2;
SocketSend client_socket\Str:="VALMIS";
```

Kuva 38. Robotin suorittamat liikkeet RAPID-ohjelmakoodissa

Kuvan 25 koodista nähdään kolmannelta riviltä kohta Set DO1. Tällä viitataan robotin työkaluun yhdistettyyn paineilmaliitäntään ulostulossa yksi. Kyseinen ulostulo on muodoltaan digitaalinen, eli se voi saada vain arvon yksi tai nolla. Kun kyseinen ulostulo kytketään SET-komennolla päälle (saa siis arvon yksi), avautuu paineilmaliitäntä, joka sulkee työkalun leuat kiinni. Vastaavasti rivillä kahdeksan kuvattu Reset DO1 tekee päinvastoin, eli paine suljetaan, leuat avautuvat ja kappale tippuu siitä pois. Komennolla WaitTime voidaan halutessaan odottaa paikallaan haluttu aika ennen seuraavaa käskyä tai liikettä.

RAPID-koodin tehtävänä on siis käytännössä kuunnella verkon yli tulevia käskyjä. Kuten aiemmista esimerkeistä voidaan huomata, ei robotin kontrollerille jää juurikaan laskentaa vaativia tehtäviä. Koska kyseessä oli varsin rajattu aihe ja melko tarkkaan määritelty – joskin yksinkertainen – tehtävä, voitiin koodin kirjoittamisessa keskittyä enemmän Pythonilla kirjoitettavaan ohjelmaan ja robotin puolella painopistettä voitiin pitää enemmän käytännön toteutuksessa. Samat kompensaatiolaskut ja koordinaattien muunnosprosessit voitaisiin hyvinkin kirjoittaa RAPID-koodillakin, mutta työn luonteen vuoksi oli mielekkäämpää tuottaa ohjelmakoodia enemmän Raspberry Pillä ja Python-ohjelmointikielellä.

4.13 Ohjelmakierto

Eri ohjelmaosioiden valmistuttua voitiin kaikki aikaansaatu koodi yhdistää, jolloin lopputuloksena oli verrattain tarkasti toimiva konenäkösovellus teollisuusrobotin ohjaamiseen. Kaikkiaan ohjelmakiertoon kuului hieman laskentatavan mukaan 15 eri aliohjelmaa tai toimintoa, jotka vuorollaan olisivat osana toteuttamassa niille asetettua tehtävää. Kuten aiemmista kappaleista on käynyt selville, kuului suurin osa koodin käsittelystä Raspberry Pille robotin kontrollerin ollessa enemmän kuuntelijan roolissa. Ohjelmakierto toteutuisi siten, että sekä Raspberry Pin että robotin koodit tekisivät ”alkutoimensa” vain kerran, jonka jälkeen kierto jatkuisi yhteydenmuodostuksen onnistuttua kuvanottoon ja siitä eteenpäin. Kuvassa 39 on esitetty tiivistettynä konenäkösovelluksen eri vaiheet niiden suoritusjärjestyksessä.

Kuva 39. Ohjelmakierto

Kuvasta poiketen myös robotti suorittaa kappaleessa 6.11 mainitut alkuvalmistelut. Kierto jatkuu ilman virheen ilmenemistä käytännössä loputtomasti, eli robotti noutaa kappaleita pöydältä niin pitkää kuin niitä löytyy. Kuten kuvasta 39 voidaan todeta, niinkin yksinkertainen operaatio kuin kappaleen nouto työalueelta valokuvan perusteella vaatii kuitenkin useita eri työvaiheita, joista jokaista tarvitaan onnistuneen noudon suorittamiseksi. Koska käsittelyssä oli samanaikaisesti kahta erityyppistä koodia, jotka toimivat itsenäisesti rinnakkain, aiheutui tästä toisinaan ongelmia, kun toinen koodeista kerkesi toista ”edelle”. Tästä esimerkkinä tilanne, jossa robotti lähti noutamaan kappaletta työalueelta, jolloin Python-koodi oman ohjelmakiertonsa mukaan ”oletti” robotin olevan jo valmis toimissaan ja otti uuden kuvan, vaikka robotti oli vielä matkalla kappaleen noudossa. Kyseessä oli tilanne, jota ohjelmointivaiheessa ei vielä ollut osattu odottaa, kun testiä jatkuvalla ohjelmakierrolla ei ollut tehty. Ongelma paljastui vasta käytännön testausvaiheessa, minkä uskottiin olevan melko tyypillistä tämänkaltaisissa projekteissa. Ratkaisu löytyi robotin lähettämästä vahvistusviestistä, jota Python-koodi odottaisi ennen siirtymistään uudestaan kuvanottovaiheeseen. Ohjelmointi – niin Pythonin kuin RAPID:nkin – vaatii loogista päättelykykyä, jonka puute paljastuu usein vasta testiajoja tehdessä. Aiemman kokemuksen puutteessa tällaisia tilanteita koodien yhteensovituksen ja robotin liikkeiden ohjelmoinnin kanssa syntyi tuon tuosta. Tämän

työn kannalta olikin erityisen hyvä asia, että työtä voitiin tehdä robotin välittömässä läheisyydessä, jolloin jokaista muutosta ja uutta ominaisuutta voitiin testata runsaasti sitä mukaa, kun koodia valmistui ja ongelmat loogisessa päättelykyvyssä voitiin huomioida ja poistaa jo ennen, kun niistä aiheutui suurempia ongelmia.

5 PÄÄTELMÄT

5.1 Yleisarviointi

Kaiken kaikkiaan tämän opinnäytetyön tuloksena syntynyt konenäkösovellus toimi lähtökohtiinsa nähden erittäin hyvin. Toteutuksen onnistuminen Raspberry Pillä ja Python-ohjelmointikielellä koettiin erityisen positiivisena asiana. Tämä tarjosi hyvän mahdollisuuden opiskella samalla ohjelmointia yleensäkin ja työn innoittamana nähtiin oivana mahdollisuutena jatkaa opiskelua robotin ja konenäön suhteen myös monimutkaisempiin projekteihin ja tekniikoihin - koneoppimisen ollessa niistä vahvimpana ehdokkaana. Lisäksi ohjelmoimalla itse pystyttiin aina olemaan varmoja siitä, mitä sovellus tekee ja eritoten siitä, mistä ongelmien syitä voisi lähteä etsimään virhetilanteissa.

Jatkuvan testauksen ja hienosäädön avulla voitiin lopulta arvioida robotin noutavan kappaleen ja lajittelevan sen onnistuneesti reilusti yli yhdeksänkymmenen prosentin todennäköisyydellä. Siihen nähden, kuinka suuria ongelmia koordinaatistojen ja vääristymien suhteen työn aikana koettiin, oli hienoa todeta laitteen toimivan niinkin tarkasti. On hyvä huomioida, että vaikka robotin suorittama varsin yksinkertainen noutoprosessi ei ole teknisesti erityisen haastava tehtävä toteuttaa, ilman käytännössä minkäänlaisia ennakkotietoja ja kokemusta kyseisistä tekniikoista ja laitteista, tarjosi projekti kuitenkin todella suuren haasteen. Lisäksi käytetyt tekniikat kuvantunnistuksessa ja tietojen lähettämissä eivät välttämättä ole niin sanotusti teollisuuden standardeja, mutta kuten aiemminkin on useampaan kertaan todettu, tämä työ tarjosi mahdollisuuden toteuttaa projekti käyttäen niitä menetelmiä, jotka käytettävien resurssien puitteissa olivat mahdollisia ja mielenkiintoisimpia. Lisäksi, vaikkei kyseessä olekaan teolliseen käyttöön tuleva laite, hyödynnetään työssä kuitenkin samoja toiminnallisia *periaatteita*. Tämä nähtiin positiivisena asiana, etenkin kun lähtökohtana oli kuitenkin toteuttaa työ, jota voitaisiin myöhemmin so-

veltaa automaatiotekniikan kursseilla oppimateriaalina. Oli erittäin mielenkiintoista seurata oman työskentelyn kehittymistä, kun uusia ominaisuuksia valmistui. Siinä missä aluksi ongelmana oli se, miten jokin tietty ominaisuus ohjelmoitaisiin, muuttui tilanne enemmän siihen suuntaan, mitä kaikkea haluttaisiin tehdä. Eli ymmärryksen kasvaessa voitiin keskittyä enemmän käytännön toteutukseen teknisten ongelmien ratkaisun sijaan.

Tuotetun koodin toimintaan voidaan olla siinä määrin tyytyväisiä, että toimiva ohjelma saatiin ohjelmoitua, eikä niin sanotusti turhaa koodia pitäisi juurikaan olla. Tämä siitä syystä, että kokeneemman ohjelmoijan silmissä tuotettu koodi voi hyvinkin näyttää kankealta ja sotkuiselta. Siihen on ymmärrettävästi syynä se, ettei työn aikana vielä keritty opiskelemaan ja sisäistämään kaikkia käytetyn ohjelmointikielen ominaisuuksia, jolloin tiettyjä ohjelmakoodin osia olisi hyvinkin voitu toteuttaa osaavissa käsissä paremmillakin tavoilla. Tiedon järjestely, funktioiden luominen ja käytettyjen menetelmien tyyli kuvastaakin todennäköisesti kutakuinkin täydellisesti aloittelevan ohjelmoijan tyyliä, jossa tärkeintä ei vielä ehkä olekaan tuottaa tyylillisesti ja teknisesti täydellistä ja ongelmatonta koodia, vaan saada laitteisto toimimaan halutulla tavalla. Tämä käy varmasti ilmi tekstissä eri ominaisuuksia auki selostaessakin. Ohjelma ei itsessään ole erityisen monimutkainen ja koodirivejäkin Python-osuuteen tuli vain noin kolmesataa kommentteineen päivineen. Tällöin koodin suorittaminenkaan ei vielä vaadi niin suurta määrää laskentatehoa, että kankean koodin takia laitteiston suorituskyvyssä nähtäisiin huomattavaa hidastumista. RAPID-koodin osalta aiempi kokemattomuus näkyy vielä selvemmin. Valtaosa koodista on toki tuotettu Pythonilla, mutta ilman toimivaa koodia robotinkin päässä olisi työn suorittaminen ollut mahdotonta. Tähän auttoi kuitenkin eniten Github.comista löydetty valmis pohja, jonka tyyliä seuraillen saatiin tarvittavat toiminnot toteutettua. Kuten aiemmin onkin mainittu, RAPID-koodin osuus oli valmiissa työssä pääsääntöisesti vain tiedon vastaanottamista ja sen välittämistä robotille.

Kokonaisuutena työstä jäi tekijälleen hyvin positiivinen vaikutelma. Suurempia vikoja tai puutteita ei juurikaan valmiiseen työhön jäänyt, näitä ja muita kehitysmahdollisuuksia käsitellään kuitenkin seuraavassa kappaleessa. Puutteisiin voitiin vaikuttaa myös rajaamalla tehtävää tarpeen mukaan – mahdollisuudet

tällaisissa projekteissa ovat kuitenkin lähes rajattomat. Työn ennakkoasetelmiin nähden katsottiin, että työn tavoitteet saatiin toteutettua ja ainakin lähestulkoon kaikki ominaisuudet saatiin toimimaan siten, kuin ne oli suunniteltukin. Tästä esimerkkinä värintunnistus-osio. Alkuperäinen suunnitelma oli toteuttaa värintunnistus havaitsemaan niin monta eri väriä kuin vain teknisesti olisi mahdollista, mutta lopulta päädyttiin toteuttamaan työ vain kahdella erisävyisellä kappaleella. Teknisesti useamman värin tunnistus ei olisi minkäänlainen ongelma, se vaatisi vain enemmän ohjelmointia. Kun tehtävänä oli kuitenkin keskittyä enemmän kappaleen noutoprosessiin, voitiin värintunnistus toteuttaa kevennettynä vain kahdelle värille. Tähänkin ongelmaan perehdytään tarkemmin vielä kappaleessa 7.2.2.

5.2 Kehitysmahdollisuudet

Vaikka työ saatiinkin suoritettua tavoitteisiinsa nähden oikein hyvin, jää aina kuitenkin jotain parannettavaa sekä ajatuksia jatkokehityksen suhteen. Opinnäytetyön luonteen voidaan käsittää yleensä olevan jokin melko tarkasti rajattu tehtävä, kuten tässäkin tapauksessa. Työn edetessä tuli silti esille useita ajatuksia siitä, kuinka työtä voitaisiin jatkossa kehittää ja mitä puutteita valmiiseen työhön jäi. Vaikka konenäkösovellus tekeekin sen, mitä siltä pyydetään, on seuraavissa kappaleissa esitetty joitain työn aikana ilmenneitä verrattain pieniä puutteita sekä mahdollisia jatkokehitysideoita.

5.2.1 Havaitut puutteet ja ongelmat

Kappaleessa 6.10 on käsitelty robotin koordinaatiston ja kameran optiikan aiheuttamia ongelmia kappaleen sijainnin määrityksessä. Vasta kompensointilaskelmien valmistumisen jälkeen kävi ilmi, että robotin akselien varsin mittava virhe johtui robotin kalibroimattomuudesta. Tämä ongelma olisi poistunut robotin kalibroinnilla, jolloin kompensoinnissa olisi todennäköisesti tarvinnut ottaa huomioon ainoastaan kameran virheellisestä asennosta johtuva kompensointi sekä optiikasta ja perspektiivistä aiheutuvat virheet. Tällaisessa tapauksessa kompensointilaskut olisivat olleet huomattavasti yksinkertaisempia kokonaisuudessaan. Voidaan silti todeta työn onnistuneen hyvin suuresta virheen kompensoinnista huolimatta, eli vaikka robotin akselien koordinaatistossa olisikin virheitä, saatiin työ kuitenkin valmiiksi.

Robotin paikoitustarkkuuden ollessa ± 0,01 millimetriä, voi sen jatkuvasta liikuttelusta aiheutua aikaa myöten ongelmia laitteistossa, jossa konenäkökamera on kiinni robotin rungossa. Tällöin kappaleen noudon onnistumisprosentti vääjäämättä ajan myötä laskee. Tämän vuoksi paras vaihtoehto olisi asentaa kamera erilleen robotista omalle tukevalle jalustalleen siten, ettei sen asento pääse missään tilanteessa muuttumaan. Kiinnityksen pitäisi olla myöskin siinä määrin tarkasti mitoitettu, että kameran voisi tarvittaessa poistaa esim. huoltotöitä varten. Tällöin robotille voitaisiin asettaa turva-alue, jolla kamera sijaitsisi, jolloin robotti ei rikkoisi tahattomasti kameraa. Nyt rakennetussa laitteistossa kameran ollessa kiinni robotin heikoimmassa kohdassa, eli sen työkalussa, on kameralaitteisto jatkuvassa vaarassa – ei pelkästään kameraan suoraan kohdistuvista iskuista – vaan myös työkalun vastaanottamista iskuista. Vaikkei kamera työkaluun kohdistuvista iskuista menisikään rikki, olisi suurempana ongelmana kameran asennon muuttuminen, jolloin kameralaitteisto olisi tavalla tai toisella kalibroitava takaisin oikeaan asentoonsa.

Valmiin laitteiston yksi puute ilmene virheenkorjauksessa tai toisin sanoen sen puutteessa. Mikäli noudettava kappale ei esimerkiksi syystä tai toisesta tartu työkalun leukoihin sitä noudettaessa, ei robotilla ole tässä tilassaan kykyä tätä ongelmaa tunnistaa. Robotti suorittaa tehtävänsä loppuun ”normaalisti”, oli sillä kappale mukanaan tai ei. Virhetilanteen syntyessä ohjelmakoodit eivät myöskään osaa tilannetta itse purkaa. Mikäli robotti törmää liikkeessään liian suurella vauhdilla esimerkiksi pöytään ja pysäyttää itsensä suojellakseen itseään vaurioitumiselta, ei Raspberry Pille kulkeudu tästä minkäänlaista tietoa, vaan ohjelma jää loputtomiin odottamaan robotilta tulevaa käskyä ottaa uusi kuva. Myöskään muista koodissa tapahtuvista virheistä ei sillä itsellään ole kykyä palautua, vaan molempien laitteiden koodit on pysäytettävä ja sitten käynnistettävä uudelleen.

5.2.2 Kehitysmahdollisuudet

Kuten aiemmassa vaiheessa on todettu, tämänkaltaisilla projekteilla on käytännössä loputtomasti potentiaalia ja robotti yhdistettynä konenäkökameraan tarjoaa valtavan määrän mahdollisia harjoitustyyppisiä projekteja. Koulun tiloissa oleva vapaasti oppilaiden käytettävissä oleva harjoituslaitteisto ei myöskään sido robottia mihinkään tiettyyn tehtävään, koska tarvittaessa robotin

kontrollerin sisältämän ohjelman voi helposti vaihtaa, mikäli samaan aikaan kehitteillä olisi useampia projekteja. Raspberry Pin kamera tarjoaa hyvät edellytykset konenäkökameraa vaativiin projekteihin, mutta yhtenä mahdollisuutena olisi kameramoduulin korvaaminen ”oikealla” konenäkökameralla, mahdollisuuksien mukaan sellaisen liittäminen myös Raspberry Pihin voisi onnistua. Tätä mahdollisuutta ei kuitenkaan tässä opinnäytetyössä tutkittu, kun laitteeseen hankittiin Raspberry Pin oma kameramoduuli, joka täytti tehtävänsä riittävän hyvin.

Kappaleiden värintunnistus toimii vain kahdella erivärisellä kappaleella, mustalla ja oranssilla. Näiden kappaleiden erottaminen toisistaan ohjelmallisesti on hyvin helppoa johtuen niiden sävyeroista, jolloin binäärikuvamuunnoksessa toinen näyttäytyy mustana ja toinen valkoisena. Periaatteessa lajittelun voisi suorittaa useammalla värillä sillä periaatteella, että lajittelu tapahtuisi kategorioiden ”musta” ja ”värikäs” välillä, jolloin kaikki mustasta eroavat kappaleet tunnistettaisiin ja lajiteltaisiin tämän uuden jaon perusteella. Mikäli kuitenkin varsinainen värin perusteella tapahtuva lajittelu haluttaisiin toteuttaa, täytyisi käytettyä binäärimenetelmää muuttaa, koska alkuperäisessä menetelmässä mahdollisia vaihtoehtoja on vain kaksi. Tämä olisi ohjelmallisesti helppoa toteuttaa. Alkuperäistä kappaleesta otettua kuvaa muokataan useaan kertaan ennen sen värimäärittelyä. Ensimmäisessä vaiheessa otettu kuva kuitenkin tallennetaan normaalina värikuvana, jolloin tätä muokkaamatonta kuvaa voitaisiin helposti hyödyntää myöhemmässä vaiheessa värintunnistamiseen muiden toimintojen pysyessä silti muuttumattomina. Kun noudettavasta kappaleesta tunnistetaan väri, tapahtuu tunnistus kappaleen keskipisteen väriarvon perusteella. Värikuvassa jokaisen pikselin väriarvo on tallennettu RGB-muodossa siten, että kaikkien värikanavien arvot muuttuvat välillä 0–255. Esimerkiksi täysin punainen väri saa arvot R = 255, G = 0 ja B = 0 ja esimerkiksi keltaiselle arvot ovat R = 255, G = 255 ja B = 0. Näin kappaleen ääriviivojen havaitsemisen jälkeen sen keskipisteen väriarvo voitaisiin merkitä muistiin (tallentaa muuttujaan) ja kirjoittaa koodiin ehtolause, joka lajittelisi eriväriset kappaleet ja antaisi niille väriä vastaavan lajittelukoordinaatin. Koska kappaleen väriarvo saattaisi hieman muuttua riippuen sen sijainnista työalueella, voisi ehtolausekkeeseen määritellä tietynväriselle kappaleelle väriarvojen vaihteluvälin, jotta pienet heijastumat ynnä muut epätarkkuudet eivät häiritsisi värintunnistusta lii-

kaa. Huomioitava seikka on, että vaikka uusia värejä voitaisiin tunnistaa aiempaa laajemmalla skaalalla, ei se poistaisi kappaleen tunnistukseen liittyvää ongelmaa, eli värillisiin kappaleisiin tulisi silti merkitä sen reunat mustalla värillä tai muutoin parantaa tunnistustekniikkaa. Värintunnistuksen voisi myös osoittaa esimerkiksi vastaavan värisen led-valon syttymisellä Raspberry Pissä. Tämänkaltaista valoindikointia voisi hyödyntää työssä muutenkin esimerkiksi koodin eri suoritusvaiheissa. Tietojen lähetyksen aikana voisi palaa yksi valo, niiden onnistuneen vastaanoton jälkeen toinen laitteen odottaessa robotin suorittamaa tehtäväänsä kolmas ja niin edelleen. Tällaiset valot havainnollistaisivat noutoprosessia varsin hyvin, etenkin henkilölle, jolla ei ole konenäkölaitteista ja teollisuusroboteista aiempaa kokemusta.

Lajitteluprosessia olisi mahdollista kehittää muutoinkin kuin pelkän värin perusteella. OpenCV-konenäkökirjasto kykenee tunnistamaan pyöreiden kappaleiden lisäksi myös käytännössä minkä tahansa muunkin muodon, jolloin lajittelua voitaisiin kehittää myös muun muotoisiin kappaleihin. Tämä olisi teknisesti hieman hankalampaa, joskaan ei mahdotonta toteuttaa. Tällöin tulisi myös huomioida robotin tarttujatyökalun mahdolliset muutostarpeet. Pyöreiden kappaleiden käytössä tämän työn suhteen oli yksi selkeä etu muihin muotoihin – pyöreää kappaletta voidaan kääntää pystyakselinsa ympäri miten päin vain ilman, että sen profiili (mittasuhteet) kameran suuntaan muuttuu. Tällöin esimerkiksi suorakulmion muotoisella kappaleella myös työkalun asentoa tulisi muuttaa sen kyytiin saamiseksi. Ominaisuuksia lajittelulle olisi siis useita, robotin kannalta suurimman haasteen tuonee tarttujan ominaisuudet ja erityisesti sen muotoilu, joka palvelee ehdottomasti parhaiten profiililtaan pyöreitä kappaleita.

Työssä rakennettu konenäkökameralaitteisto sallisi myös jossain määrin erilaiset laaduntarkkailutehtävät. Kameraa voitaisiin käyttää myös tunnistamaan kappaleista erilaisia virheitä tai puutteita. OpenCV-kirjasto tarjoaa niin suuren määrän hyödyllisiä ominaisuuksia, että sillä voitaisiin vaikkapa tarkistaa kappaleesta löytyvän tekstin oikeinkirjoitus, sijainti, väri ynnä muita ominaisuuksia. Näin robotti voisi lajitella hyvät ja huonot kappaleet erikseen – teollisuudessa konenäkökameroita käytetäänkin runsaasti juurikin laaduntarkkailutehtäviin. Mikäli tällaisia muutoksia työhön tehtäisiin, olisi todennäköistä, että myös valaistukseen pitäisi panostaa enemmän. Mitä pienempiin kohteisiin

mennään, sitä tärkeämpää on huolehtia riittävän laadukkaasta valaistuksesta.

Kuten aiemmin tässä kappaleessa on mainittu, noutoprosessia olisi mahdollista havainnollistaa enemmän esimerkiksi led-valoin ilmoittamaan prosessin eri vaiheista ja mahdollisista virhetilanteista. Toinen kehitysmahdollisuus havainnollistamiseen ja työn selkeyttämiseen olisi luoda Raspberry Pille kirjoitetusta ohjelmakoodista oikea sovellus, jonka kautta käyttö tapahtuisi. Sovellus voisi esittää otetun kuvan, tietoja siitä ja esimerkiksi tallentaa useampia aiemmin otettuja kuvia muistiin, joita sitten voisi selata sovelluksessa. Pythonilla kirjoitettuja koodeja voi muokata ohjelmamuotoon esimerkiksi TkInter-nimisellä lisäosalla, jota voidaan käyttää graafisten käyttöliittymien luomiseen [35]. Tätä vaihtoehtoa suunniteltiin työn aikana, mutta ajatuksesta luovuttiin, kun laitteiston toiminnot olivat kuitenkin siinä määrin suppeita, ettei kyseiselle ominaisuudelle nähty riittävää tarvetta.

Laitteiston testauksessa laskettiin keskiarvot ohjelmakierron käyttämälle ajalle. Ajat ovat summittaisia, mutta antavat silti hyvän kuvan eri toimintoihin kuluvasta ajasta. Kun Python-koodi käynnistetään ja robottiin muodostetaan yhteys samalla, kun kuva otetaan, vei onnistunut kappaleentunnistus ja sen koordinaattien lähetys robotille testijaksoissa aikaa kahdeksasta yhdeksään sekuntia. Seuraavilla kierroksilla, kun yhteys oli jo muodostettu, meni kuvan käsittelyvaiheeseen keskimäärin hieman alle kahdeksan sekuntia. Noutoprosessin aika vaihtelee eniten, koska kappaletta haetaan eri puolilta työaluetta ja toimitetaan lajitteluun robotin nopeuden pysyessä vakiona. Tähän meni testeissä keskimäärin 13,5 sekuntia. Noutoprosessin alku on laskettu siitä hetkestä, kun robotti on vastaanottanut koordinaatit ja lähtee liikkeelle. Nouto päättyy sillä hetkellä, kun robotti on saapunut takaisin kuvanottopisteeseen ja uusi kierto alkaa. Näin ollen kokonaiskiertoajaksi saadaan keskimäärin 30 sekuntia. Testeissä robotin nopeus on kuitenkin asetettu varsin hitaalle liikkeelle, joten erityisesti noutoaikaa olisi helppo nopeuttaa, jos laitteiston toiminnasta ollaan muutoin varmoja siten, että törmäysvaaran riski on saatu minimoitua. Ymmärrettävästi törmäystilanteessa suurempi vauhti kasvattaa myös laitteiden – erityisesti mukana kulkevan RasPin – vaurioitumisriskiä. Myös ohjelmakierrosta voisi löytyä hiomista, ja yksi selkeä tarkemman tutkimuksen paikka

olisikin koordinaattien lähetysprosessissa, jossa viestien lähetyksen välille jätettiin tarkoituksella pitkä, sekunnin mittainen viive viestien varman toimituksen varmistamiseksi.

LÄHTEET

1. IRB 120 ABB's 6 axis robot – for flexible and compact production. ABB. 2019. PDF-dokumentti. Saatavissa: https://library.e.abb.com/public/7aa0711a20fa41c49a8fdf3bbc3d5bb0/IRB120-Rev.J-ROBO149EN_D.pdf?x-sign=LJaHKRy/4i2ZKYT7HvybnSGI+eV-Jzc8Ibuj7gPd1s1kO+b7vIxHBeQVI4JgFc20M [viitattu 16.3.2020].

2. Tuunanen, T. 2014. Teollisuusrobotin käyttöönotto ja ohjelmointi. Mikkelin ammattikorkeakoulu. Sähkötekniikka. Opinnäytetyö. PDF-dokumentti. Saatavissa: http://urn.fi/URN:NBN:fi:amk-201405076268 [viitattu 18.3.2020].

3. Käyttäjän opas IRC 5 ja FlexPendant. ABB Robotics. 2013. PDF-dokumentti. Saatavissa: https://docplayer.fi/510174-Abb-robotics-kayttajan-opas-irc5-ja-flexpendant.html [viitattu 18.3.2020].

4. About us. Raspberry Pi Foundation. 2020. WWW-dokumentti. Saatavissa: https://www.raspberrypi.org/about/ [viitattu 20.3.2020].

5. Raspberry Pi models comparison. SocialCompare.com 2020. WWW-dokumentti. Saatavissa: https://socialcompare.com/en/comparison/raspberrypi-models-comparison [viitattu 20.3.2020].

6. Raspberry Pi – GPIO. Raspberry Pi Foundation. S.a. WWW-dokumentti. Saatavissa: https://www.raspberrypi.org/documentation/usage/gpio/ [viitattu 30.3.2020].

7. About. OSMC. 2020. WWW-dokumentti. Saatavissa: https://osmc.tv/about/ [viitattu 20.3.2020].

8. UNIX. Wikipedia. 2020. WWW-dokumentti. Saatavissa: https://fi.wikipedia.org/wiki/Unix [viitattu 29.3.2020].

9. Smith. R. 2012. Linux Essentials. E-kirja. Indianapolis, Indiana: John Wiley & Sons, Inc. Saatavissa: https://ebookcentral.proquest.com/lib/xamk-ebooks/detail.action?docID=817722 [viitattu 16.3.2020].

10. Raspberry Pi – Camera Module. Raspberry Pi Foundation. S.a. WWW-dokumentti. Saatavissa: https://www.raspberrypi.org/documentation/hardware/camera/ [viitattu 1.4.2020].

11. History of Python. GeeksforGeeks. S.a. WWW-dokumentti. Saatavissa: https://www.geeksforgeeks.org/history-of-python/ [viitattu 25.3.2020].

12. Tiobe Index for April 2020. 2020. TIOBE Software BV. WWW-dokumentti. Saatavissa: https://www.tiobe.com/tiobe-index/ [viitattu 6.4.2020].

13. Applications for Python. Python Software Foundation. S.a. WWW-dokumentti. Saatavissa: https://www.python.org/about/apps/ [viitattu 31.3.2020].

14. Why is Python a dynamic language and also a strongly typed language. Python Software Foundation. 2012. WWW-dokumentti. Saatavissa: https://wiki.python.org/moin/Why%20is%20Python%20a%20dynamic%20language%20and%20also%20a%20strongly%20typed%20language [viitattu 6.4.2020].

15. About. OpenCV. 2020. WWW-dokumentti. Saatavissa: https://opencv.org/about/ [viitattu 29.3.2020].

16. Chen, I, Ho, S-C, Su, M-B. 2020. Computer vision application programming for settlement monitoring in a drainage tunnel. *Elsevier* Volume 110, 1-9. Helmikuu 2020. Tieteellinen artikkeli. Saatavissa: https://www.sciencedirect.com/science/article/abs/pii/S092658051930603X [viitattu 29.3.2020].

17. Pyykkönen, O. 2018. Revikan ohjaus ja käyttäminen : Oulun poliisilaitoksen valvonta- ja hälytyssektorilla ja liikennesektorilla. Poliisiammattikorkeakoulu. Poliisi (AMK). Opinnäytetyö. PDF-dokumentti. Saatavissa: http://urn.fi/URN:NBN:fi:amk-201802272843 [viitattu 2.4.2020].

18. Saikkonen Riku. 2012. Ohjelmoinnin peruskurssien laaja oppimäärä, Luento 4: Verkko-ohjelmointi, tapahtumapohjainen ohjelmointi, lisää ohjelmien suunnittelusta. PDF-dokumentti. Aalto-yliopisto. Päivätty 15.2.2012. Saatavissa: https://wiki.aalto.fi/download/attachments/63548818/luento4.pdf?version=1&modificationDate=1329258090084 [viitattu 1.4.2020].

19. What is a socket? Tutorials Point. S.a. WWW-dokumentti. Saatavissa: https://www.tutorialspoint.com/unix_sockets/what_is_socket.htm [viitattu 30.3.2020].

20. Rohan Murty, Hitesh Ballani. 2004. Socket Programming. Cornell University. PDF-dokumentti. Päivitetty 8.2.2004. Saatavissa: http://www.cs.cornell.edu/courses/cs519/2004sp/519-fa04-03-sockets-v0.pdf [viitattu 22.3.2020].

21. The TCP/IP Protocol Stack. TechnologyUK. S.a. WWW-dokumentti. Saatavissa: http://www.technologyuk.net/computing/computer-networks/internet/tcp-ip-stack.shtml [viitattu 6.4.2020].

22. Integrated Vision, Product specification. ABB Robotics. 2018. PDF-dokumentti. Saatavissa: https://search-ext.abb.com/library/Download.aspx?DocumentID=3HAC046868-001&LanguageCode=en&DocumentPartId=&Action=Launch [viitattu 27.3.2020].

23. Install OpenCV 4 on your Raspberry Pi. Pyimagesearch. 2018. WWW-dokumentti. Saatavissa: https://www.pyimagesearch.com/2018/09/26/install-opencv-4-on-your-raspberry-pi/ [viitattu 28.3.2020].

24. Contour Features. OpenCV. S.a. WWW-dokumentti. Saatavissa: https://docs.opencv.org/3.4/dd/d49/tutorial_py_contour_features.html [viitattu 20.3.2020].

25. ABB-Robot-with-python-socket. Jatin Goyal. 2019. Tietolähde. Saatavissa: https://github.com/Jatin1o1/ABB-Robot-with-python-socket [viitattu 13.3.2020].

26. SafeMove, Robot Safety Option. ABB Robotics. S.a. PDF-dokumentti. Saatavissa: https://library.e.abb.com/public/cf49d4f8bed37f67c125772e00518e8e/SafeMove%20ROB0088EN_C.pdf [viitattu 26.3.2020].

27. ABB's smallest robot – IRB 120. ABB. 2012. PDF-dokumentti. Saatavissa: https://library.e.abb.com/public/4ba04345ce481405482579800011cbfa/Product%20presentation%20IRB%20120%20Revision%20E.pdf [viitattu 28.3.2020].

28. Barrel distortion. ScienceDirect. 2018. WWW-dokumentti. Saatavissa: https://www.sciencedirect.com/topics/engineering/barrel-distortion [viitattu 3.4.2020].

29. Imutils. PyPi. 2019. WWW-dokumentti. Saatavissa: https://pypi.org/project/imutils/#description [viitattu 2.4.2020].

30. Binary Image Processing. University of South Florida. S.a. PDF-dokumentti. Saatavissa: https://www.cse.usf.edu/~r1k/MachineVisionBook/MachineVision.files/MachineVision_Chapter2.pdf [viitattu 6.4.2020].

31. Contours Hierarchy. OpenCV. S.a. WWW-dokumentti. Saatavissa: https://docs.opencv.org/master/d9/d8b/tutorial_py_contours_hierarchy.html [viitattu 4.4.2020].

32. Suoran suunta. Jyväskylän yliopisto. S.a. WWW-dokumentti. Saatavissa: http://www.math.jyu.fi/matyl/propedeuttinen/kirja/index-49.html [viitattu 28.3.2020].

33. Ristitulo. Helsingin yliopisto, Matemaattis-luonnontieteellinen tiedekunta. S.a. PDF-dokumentti. Saatavissa: https://www.cs.helsinki.fi/u/jhasa/kurssit/lm1_kesa13/lm1_luku14.pdf [viitattu 5.4.2020].

34. Calculate on which side of a straight line is a given point located? StackExchange. 2013. Keskusteluryhmän artikkeli. Saatavissa:

https://math.stackexchange.com/questions/274712/calculate-on-which-side-of-a-straight-line-is-a-given-point-located [viitattu 19.3.2020].

35. TkInter. Python Software Foundation. 2019. WWW-dokumentti. Saatavissa: https://wiki.python.org/moin/TkInter [viitattu 29.3.2020].

KUVALUETTELO

PYTHON-KOODIN KÄSITTELY

Tässä liitteessä käydään läpi Python-ohjelmointikielellä tehty ohjelma, jolla konenäkökameraa ohjataan ja jolla koordinaattien lähetys robotille tapahtuu. Koodi avataan enemmän tai vähemmän rivi riviltä, joskin toistuvat komennot ynnä muut ilmiselvät rivit jätetään huomiotta, kun ne on ensimmäisen kerran käsitelty. Kuvissa näkyvät #-alkuiset rivit ovat Pythonissa kommentointia varten, eli niillä riveillä ei ole alkumerkin jälkeen mahdollista toteuttaa mitään ohjelmallista toimintoa, ts. koodia. Kahdella kolmen heittomerkin sarjalla (’’’, ’’’) voidaan kommentoida useita riviä tekstiä kerralla, jolloin kaikki näiden merkkien väliin jäävä teksti tulkitaan kommentiksi. Kommentoinnilla voidaan helposti myös testata toimintoja ja löytää mahdollisia virheitä koodissa – riviä ei tarvitse kokonaan poistaa, vaan se voidaan kommentoida esim. testauksen ajaksi ”piiloon”. Tässä liitteessä ei juurikaan keskitytä syihin, jotka johtivat kunkin koodirivin kirjoittamiseen (eli miksi näin tehtiin), vaan siihen miten koodi toimii. Syitä on pohdittu aiemmin tämän opinnäytetyön Toteutus-osiossa. Kyseiseen osioon onkin hyvä tutustua periaatteet ymmärtääkseen, mikäli tätä opastusta haluaa käyttää.

```
28  '''----------------Tuodaan tarvittavat kirjastot--------------------'''
29  import cv2                                  # OpenCV
30  from picamera import PiCamera               # RaspBerryn kamera
31  import imutils                              # kuvan kääntö
32  import socket                               # TCP/IP-yhteys
33  import time                                 # Odotus
```

Ensimmäisenä koodissa tuodaan projektiin sen vaatimat kirjastot, kuvan kommenteissa avattuna kunkin kirjaston tehtävä.

```
35  '''-------------Muodostetaan yhteys robottiin-------------'''
36  # Luodaan socket HUOM! socket luodaan kierrossa vain yhden kerran
37  s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
38  # Määritetään portti
39  port = 5555
40  # Määritetään haluttu ip-osoite
41  s.connect(('192.168.0.101', port))          #robotti
42  #s.connect(('192.168.0.100', port))         #simulointi (Kommentoi aina toinen pois)
43  print("Yhdistetään robottiin")
```

Yhteydenmuodostus alkaa luomalla s-niminen muuttuja, joka sisältää socket-moduulin ja sen tarvitsemat vakiomuuttujat AF_INET ja SOCK_STREAM. Lisätietoja osoitteesta https://docs.python.org/3/library/socket.html. Riville **39** Port-arvoksi voidaan määritellä mikä tahansa vapaa portti väliltä 1024-65536. Mikäli porttia muuttaa, on muutos muistettava tehdä myös RAPID-koodiin. Alle arvon 1024 olevat portit ovat ”varattuja” portteja, esim. internet-liikenteelle käytetään porttia 80, jota kaikki nettiselaimet kuuntelevat. IP-osoitteeksi laitetaan robotin kontrollerin IP-osoite. Kontrollerin IP-osoitteen löytää FlexPendantin *System info* -valikosta kohdasta *Network Connections – Service Port* tai *WAN*, riippuen siitä, kumpaan porttiin robotissa verkkokaapeli on kytketty. Riveillä **41** tai **42** täytyy toisessa olla aina kommenttimerkki edessä riippuen siitä, halutaanko koodilla ohjata suoraan robottia vai RobotStudiossa pyörivää simulaatiota. ”Väärä” vaihtoehto kommentoidaan pois. Rivillä **43** tulostetaan konsoliin teksti ”Yhdistetään robottiin” havainnollistamaan ohjelman etenemistä.

```
45  # Pyydetään serveriltä dataa
46  def data_pyyntö():
47      data = s.recv(4096)
48      if len(data) > 0:
49          res=str(data)
50          print(res[2:len(res)-1])            # Karsitaan viesteistä turhat merkit pois
51  # (b'ROBOTILTA SAAPUNUT VIESTI') --> ROBOTILTA SAAPUNUT VIESTI
```

Ensimmäinen funktio määritellään seuraavaksi ”def funktion_nimi():” -komennolla. Tällöin kyseistä funktiota voidaan kutsua uudelleen tarvittaessa pelkästään sen nimeä käyttäen. Rivillä **47** voidaan recv-komennolla asettaa koodi vastaanottamaan tietoa socketista, sulkeissa käytettävän puskurin koko. Puskurin koon on oltava mielellään suhteellisen pieni numero kahden potenssi, esimerkiksi tässä käytetty 4096. Rivillä **48** vastaanotetun datan pituus merkkeinä tarkistetaan ja mikäli pituus on vähintään yksi merkki, tulostetaan saapunut viesti konsoliin. Rivillä **49** on luotu res-niminen muuttuja, jossa aiemmin luotu data-muuttuja muunnetaan *stringiksi,* eli tekstiksi. Rivillä **50** tapahtuu lopullinen tulostus, kun saapuneesta viestistä on karsittu ylimääräiset merkit

pois. Rivillä **51** esimerkki saapuvan viestin siistimisestä. Halutessaan vaikutuksen voi todeta lisäämällä viimeisen rivin **50** jälkeen uuden rivin ja kirjoittamalla riville print(res) ja ajamalla koodin.

```
55  '''----------------------Kameran asetukset ja kuvan otto-----------------------'''
56  def kuvan_otto():
57      camera = PiCamera()             # Määritetään kamera
58      camera.resolution = (1000,707)  # Resoluutio (ÄLÄ MUUTA, kuvasuhde sama kuin A4)
59      camera.contrast = 25            # Kontrasti [-100 - 100]
60      camera.exposure_mode = 'auto'
61      camera.capture('image.jpg')     # Tallennetaan kuva projektikansioon
62      print('Otetaan kuva')
63      camera.close()
```

Seuraavana otetaan kuva. Jälleen määritellään kuvan otolle oma funktionsa kuvan_otto(), jonka jälkeen rivillä **57** määritetään käytettävä kamera, tässä tapauksessa käytetään Raspberry Pin omaa kameramoduulia. Seuraavilla kolmella rivillä määritetään kuvan resoluutio (x,y), kontrasti ja käytettävä valotusmoodi. Sivulta https://www.raspberrypi.org/documentation/raspbian/applications/camera.md löytyy komennot eri moodien käyttöön. Valotusmoodia valittaessa kannattaa huomioida tekstiä ympäröivät heittomerkit. Rivillä **61** capture-komennolla otetaan kuva ja tallennetaan se samaan kansioon, missä projektitiedosto sijaitsee. Halutessaan tähän voidaan määritellä myös eri kansio antamalla kansion polku ennen kuvan nimeä, nämä kaikki kirjoitetaan heittomerkkien sisään. Rivillä **63** kameramoduuli suljetaan kuvan oton ja tallennuksen jälkeen.

```
67  '''-------------------Kuvan lataus ja muokkaus------------------'''
68  def kuvan_lataus_ja_muokkaus():
69      global img_grey, img_binary, x_resoluutio, y_resoluutio
70      img_grey = cv2.imread('image.jpg', cv2.IMREAD_GRAYSCALE) # Luetaan MV-kuva muistiin
71      img_grey = imutils.rotate(img_grey, 180)                 # Käännetään kuvaa 180 astetta
72      ''' Thresh-muuttuja määrittää binäärikuvan mustan ja vaalean raja-arvon,
73      jos esim valaistus muuttuu ja tunnistus ei onnistu, kokeile ensin säätää
74      tätä arvoa'''
75      thresh = 20
76      # Binäärikuvan luonti
77      img_binary = cv2.threshold(img_grey, thresh, 255, cv2.THRESH_BINARY)[1]
78      cv2.imwrite('imagebw.jpg',img_binary)         # Tallennetaan mv-binäärikuva
79      x_resoluutio, y_resoluutio = img_grey.shape   # Kuvan resoluutiomuuttujat
```

Funktion määrittelyn jälkeen asetetaan neljä kyseisessä funktiossa käytettävää muuttujaa globaaleiksi. Tällä tarkoitetaan sitä, että näitä muuttujia voidaan tarvittaessa kutsua myöhemmissä vaiheissa. Ilman tätä määrittelyä muuttujat toimivat vain saman funktion sisällä. Img_grey -muuttujaan kirjoitetaan rivillä **70** imread-komento, joka on OpenCV-kirjaston tapa ladata haluttu kuva projektin muistiin. Sulkeissa haluttu kuva, eli sama kuva joka hetki sitten otettiin. Tämä kuva voi olla myös mikä tahansa projektikansiosta löytyvä kuvatiedosto (ainakin .jpg tai .png käypiä). Samalta riviltä löytyvä IMREAD_GRAYSCALE-käsky viittaa kuvan lukemiseen mustavalkoisena. Vaihtoehdot tälle löytyvät osoitteesta https://docs.opencv.org/3.4/d4/da8/group__imgcodecs.html#gga61d9b0126a3e57d9277ac48327799c80ae29981cfc153d3b0cef5c0daeedd2125. Rivillä **71** käytetään imutils-kirjastoa kääntämään otettua kuvaa rotate-komennolla, suluissa haluttu kuva ja asteet. Binäärikuvan muodostusta varten luotiin thresh-muuttuja, joka määrittää mustan ja valkean raja-arvon lopullisessa tutkittavassa kuvassa. Tätä arvoa muuttamalla voidaan haarukoida sopiva arvo, jonka perusteella kaikki tätä arvoa tummemmat alueet näkyvät mustina ja vaaleammat valkoisena. Vaihteluväli 0-255. Rivillä **77** luodaan binäärikuva käyttäen apuna edellisellä rivillä luotua thresh-arvoa. Threshold-komennon sisältämät parametrit ovat seuraavat: ensin sulkeiden sisällä on muutettava kuva, tässä tapauksessa img_grey -muuttuja, eli mustavalkoiseksi aiemmin tallennettu kuva. Sen jälkeen on raja-arvo, tämä voi olla numeerisessa muodossa tai muuttujana, kuten tässä tapauksessa. Luku 255 viittaa niihin pikseleihin, jotka jäävät raja-arvon yläpuolelle. Tällä arvolla kaikki kirkkaammat pikselit muutetaan täysin valkoisiksi. Lopputuloksena siis mustaa ja valkoista sisältävä kuva. Neljäs parametri viittaa kuvan tallennukseen binäärimuodossa. Binäärimuutos voidaan suorittaa usealla eri tavalla hieman eri lopputuloksin. Eri vaihtoehdot nähtävissä täällä: https://docs.opencv.org/master/d7/d4d/tutorial_py_thresholding.html. OpenCV-kirjaston tapa tallentaa kuva tapahtuu imwrite-komennolla., jota käytetään rivillä **78**. Parametrit järjestyksessä: tallennettavan tiedoston nimi, tallennettava kuva tai muuttuja (https://docs.opencv.org/2.4/modules/highgui/doc/reading_and_writing_images_and_video.html#bool%20imwrite(const%20string&%20filename,%20InputArray%20img,%20const%20vector%3Cint%3E&%20params)). Rivillä **79** tallennetaan kuvan resoluution arvot omiin muuttujiinsa. Tämä toteutetaan

shape-komennolla. Lopputuloksena kuvan x-akselin resoluutio tallentuu x_resoluutio -muuttujaan ja y-akselin vastaava y_resoluutio -muuttujaan.

```
83  '''------Ääriviivojen tunnistus-------'''
84  def aariviivat():
85      global cnts, toisto
86      cnts = cv2.findContours(img_binary, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)
87      cnts = imutils.grab_contours(cnts)
88      # Löydetyt alueet järjestellään pinta-alan perusteella
89      cnts = sorted(cnts, key = cv2.contourArea, reverse = True)[:3]
90      toisto=bool(1)
91      area = cv2.contourArea(cnts[1])
92      # Tarkastetaan kappaleen koko
93      if 15000 < area < 23000:
94          #print('Kappaleen pinta-ala: ', area)
95          pass
96      else:
97              print('Ei kappaleita, yritetään uudelleen viiden sekunnin päästä.')
98              time.sleep(5)
99              kuvan_otto()
100             kuvan_lataus_ja_muokkaus()
101             aariviivat()
```

Ääriviivojen tunnistusfunktiossa luodaan aluksi tarvittavat globaalit muuttujat. Cnts-muuttujaan rivillä **86** tallennetaan kappaleesta löytyvät ääriviivat. Parametrit: tutkittava kuva tai muuttuja, ääriviivojen hierarkia, käytettävä menetelmä. Hierarkialla viitataan ääriviivojen tallentamisen periaatteeseen. Esimerkki. Mikäli tutkittavassa kappaleessa on reikä tai reikiä, voidaan löydettyjen ääriviivojen hierarkia määritellä sen perusteella, mikä viivoista on *uloimpana*. Tällöin uloimpana olevat ääriviivat ovat hierarkiassa korkeammalla, kuin sisemmät. Tätä hierarkiaan perustuvaa jakoa voidaan tarvittaessa muuttaa. Cv2.RETR.TREE -komento tekee juuri tällaisen hierarkian, jossa uloimmat ääriviivat ”lasketaan” ensin ja sisemmät sen jälkeen. Tässä työssä hierarkialla ei ollut juuri merkitystä, koska kappaleet ovat yhtenäisiä ja niistä käytetään vain tarkkaan valittuja ääriviivoja https://docs.opencv.org/3.4/d9/d8b/tutorial_py_contours_hierarchy.html. Menetelmänä on käytetty CHAIN_APPROX_SIMPLE -menetelmää. Tällöin kaikista vaaka-, pysty- ja diagonaalilinjoista tallennetaan vain niiden päätepisteet, eli suorakulmaisesta kappaleesta tallennettaisiin vain neljä pistettä. Pyöreällä kappaleella pisteitä on huomattavasti enemmän, koska suorat linjat puuttuvat. Vaihtoehdot kuvattuna

https://docs.opencv.org/2.4/modules/imgproc/doc/structural_analysis_and_shape_descriptors.html. Rivillä **87** imutils-kirjastolla tallennetaan löydetyt ääriviivat grab_contours -komennolla aiemmin luotuun cnts-muuttujaan. Jotta kappaleen koon tunnistus toimisi jatkossa, lajitellaan ääriviivat rivillä **89** sorted-komennolla. Ensimmäisenä parametrina on haluttu muuttuja, eli cnts. ”Key” tarkoittaa tässä järjestyksen tapaa, tyyliä tai ominaisuutta, jonka mukaan järjestely halutaan toteuttaa. Tässä kohtaa järjestelyperusteena on OpenCV:n contourArea-komento, eli ääriviivojen sisään jäävä pinta-ala. Reverse-parametrilla määritetään järjestelyn suunta, työssä käytettiin suurimmasta pienimpään lajittelua. Viimeisenä hakasuluissa on merkit ”:3”. Tällä valmista pinta-alat järjestyksessä olevaa listaa rajataan neljään ensimmäiseen tulokseen. Jättämällä tämän pois tallentuu cnts-nimisen muuttujan sisältämään listaan kaikki kuvasta löydetyt alueet pinta-aloineen. Rivillä **90** luotu muuttuja ”toisto” toimii ns. tyhjänä muuttujana, sitä käytetään vain ohjelmakierron lopussa käynnistämään kierto uudelleen. Rivin **91** area-muuttujaan tallennetaan cnts-muuttujan sisältämän listan toiseksi suurin objekti – tämä ilmenee hakasuluissa olevasta numerosta. Toiseksi suurin siksi, että Pythonin tapa tallentaa listat alkavat aina indeksistä nolla ja löydetyistä ääriviivoista (ainakin näillä asetuksilla) suurin on aina koko kuvan alue ja täten aina suurempi kuin haettavan kappaleen pinta-ala.

Area-muuttujan määrittelyn jälkeen luodaan ehtolause riviltä **93** alkaen, jolla suodatetaan vääränkokoiset kappaleet tuloksista pois. Näin voidaan pääsääntöisesti varmistua siitä, että vain noudettavat kappaleet päätyvät tarkemmin tutkittavaksi. Tutkittavien kappaleiden pinta-alat olivat testauksessa aina kooltaan 19000 ja 21000 väliltä, jolloin muutaman tuhannen yksikön marginaalit suuntaansa varmistavat, että suodatus toimii luotettavasti. Mikäli listan toiseksi suurimman kappaleen pinta-ala jää annettujen arvojen välille, jatkuu ohjelmakierto normaalisti (pass). Muussa tapauksessa tulkitaan, että kappaleita ei ole työalueella, jonka jälkeen time.sleep(5)-komennolla rivillä **98** ohjelma odottaa viisi sekuntia, kunnes kierto aloitetaan alusta uuden kuvan otolla.

```
105 '''--------------Keskipisteen laskenta ja näyttö--------------'''
106 def keskipisteen_laskenta_ja_naytto():
107     global cy, cx
108     m = cv2.moments(cnts[1])
109     cx = int(m["m10"] / m["m00"])        # Keskipisteen laskenta akseleittain
110     cy = int(m["m01"] / m["m00"])
111     koordinaatit = cx, cy
112     # Piirretaan keskipiste objektille
113     cv2.circle(img_binary, (cx, cy), 1, (150, 0, 150), 3)
114     # Lisätään keskipisteen koordinaatit kuvaan
115     cv2.putText(img_binary, '{}'.format(koordinaatit) , (cx - 20, cy - 20),
116     cv2.FONT_HERSHEY_SIMPLEX, 1, (150, 0, 150), 1)    # Kaytetty fontti
117     print('------Tulokset-------')
118     print('Objektin keskipisteen koordinaatit: {} x {}'.format(cx, cy))
119     cv2.imshow('imagebw.jpg', img_binary)
120     cv2.waitKey(2000)                                 # Kuvan näyttö [ms]
121     cv2.destroyAllWindows()                           # Sulje ikkuna
```

Rivillä **108** alkaa keskipisteen määritys halutulle kappaleelle. Moments-komennolle annetaan käsittelyyn cnts-muuttuja ja sen toiseksi suurin objekti. Riveillä **109** ja **110** lasketaan keskipisteiden koordinaatit niille määriteltyihin muuttujiin (https://docs.opencv.org/2.4/modules/imgproc/doc/structural_analysis_and_shape_descriptors.html?highlight=moments#moments). Kappaleelle lasketaan ns. massakeskipiste, jolloin kappaleen ei tarvitse olla symmetrinen keskipisteen laskemiseksi. Rivillä **111** koordinaatit-muuttujaan tallennetaan lasketut keskipisteen koordinaatit akseleittain myöhempää käyttöä varten. Rivillä **113** circle-komennolla piirretään pieni piste (ympyrä, jonka säde on yksi pikseli) tutkittavaan kuvaan visuaalista tarkastusta varten. Parametrit: tutkittava kuva, halutun pisteen koordinaatit, säde, väri, viivanleveys. Rivillä **115** ja **116** putText-komennolla lisätään keskipisteen oheen myös sen koordinaatit. Parametrit: tutkittava kuva, lisättävä teksti, halutun tekstin vasemman alareunan sijainti, käytettävä fontti, fontin skaalaus sen peruskokoon nähden, tekstin väri, tekstin paksuus. On hyvä huomioida, että putText-komento siis jatkuu seuraavallekin riville. Rivin **119** imshow-käskyllä voidaan näyttää haluttu kuva tietokoneen ruudulla. Näyttö kestää rivillä **120** olevan waitKey-komennon sulkeissa olevan ajan [ms]. Lopulta rivillä **121** luotu kuvannäyttöikkuna sulkeutuu destroyAllWindows-komennolla.

```
125  '''--------------------Värintunnistus ja lajittelu----------------------'''
126  def värintunnistus_ja_lajittelu():
127      global x_poisto
128      värintunnistus = cv2.imread('imagebw.jpg', 0)
129      väri = värintunnistus[cy,cx]
130      if väri == 0:
131          print('Musta palikka')
132          x_poisto = 490
133      else:
134          x_poisto = 410
135          print('Oranssi palikka')
```

Värintunnistus alkaa riviltä **128**, jossa luodaan samanniminen muuttuja kuvan muistiin lukua varten. Indeksi nolla sulkujen lopuksi määrittää kuvan lukemisen harmaasävykuvana. Seuraavaksi rivillä **129** luotava väri-muuttuja sisältää aiemmin lasketun kappaleen keskipisteen sijainnin, jonka arvoa sitten tutkittaisiin. Riviltä **130** alkaa ehtolause, jossa kappaleen väri tunnistetaan. Musta kappale saa aina arvon nolla, muussa tapauksessa kappaleen oletetaan olevan *jotain muuta kuin musta*. Tässä tapauksessa siis oranssi. Väri tunnistetaan aiemmin luodusta binäärikuvasta, jossa ei siis ole muita värejä, kuin mustaa ja valkoista. X_poisto -muuttujalla määritellään robotille lähtevä kappaleen x-akselin lajittelukoordinaatti työpöydällä.

```
140  '''-------Koordinaattien muutos robotille---------'''
141  def koordinaattien_muutos():
142      ''' Muunnetaan arvot robotin koordinaatistoon seuraavalla kaavalla:
143      Akselin minimiarvo +/- ( keskipiste % kuvan reunasta ) * työalueen koko '''
144      global x_robot, x_koord, y_robot, y_koord
145      x_robot = 178     # Työalueen leveys robotin x-akselilla
146      # Muunnetaan x-koordinaatti robotille
147      x_koord = int(349+(cy/x_resoluutio)*x_robot)
148      # Muunnetaan y-koordinaatti robotille
149      y_robot = 270  # Työalueen leveys y-akselilla
150      y_koord = int(72-(1-(cx/y_resoluutio))*y_robot)
151      print('Muunnetut koordinaatit:')
152      print('X-koordinaatti: {}'.format(x_koord))
153      print('Y-koordinaatti: {}'.format(y_koord))
```

Kuvassa esitetyssä työvaiheessa ei juurikaan uusia komentoja ilmene. Periaatteet laskuille voi halutessaan tarkistaa kappaleesta 6.7. Riveillä **147** ja **150**

riviltä löytyvät int()-komennot määräävät muuttujan tuloksen tallentamisen integer, eli kokonaislukumuotoon. Riveillä **152** ja **153** on käytetty string-muotoisen tekstin sisään syötettävää muuttujan arvoa format-komennolla. Esimerkki: luodaan ”kellonaika”-niminen muuttuja, joka näyttää sen hetkisen ajan. Halutaan tulostaa konsoliin teksti ”Kello on [kellonaika]” (Eli esim. ”Kello on 12.25”). Tämä voidaan toteuttaa komennolla print(’Kello on {}’.format(kellonaika)). Aaltosulkeet string-muotoisen tekstin seassa ja .format() -yhdistelmä antaa toteuttaa kyseisen komennon. Sama voidaan toteuttaa muinkin keinoin, mutta harjoituksen vuoksi päätettiin toteuttaa kyseisen rivin tulostus näin.

```
157  '''----------------Virheiden kompensointi---------------'''
158  def virheiden_kompensointi():
159      global x_korjattu, y_korjattu
160      ''' Suoran yhtälö x-akselin korjaukselle:
161        y=(88/129)x-425 TAI 88x-129y-54825=0 '''
162      # x-akselin korjaussuoran pisteet, älä muuta näitä arvoja!
163      x2 = 374
164      x1 = 503
165      y2 = -170
166      y1 = -82
167      # Ratkaistaan pisteen sijainti suoran suhteen ristitulon kaavalla
168      v1 = (x2-x1, y2-y1)                    # Vektori 1
169      v2 = (x2-x_koord, y2-y_koord)          # Vektori 2
170      x_risti = v1[0]*v2[1] - v1[1]*v2[0]    # Ristitulo
171      # Suoran funktiosta saadut arvot
172      fx_x = 88          # Näitä ei saa muuttaa!
173      fy_x = -129
174      fc_x = -54825
```

Virheiden kompensointiosio on käyty varsin perinpohjaisesti läpi jo aiemmin. Kyseinen osio sisältää käytännössä vain matemaattisia lausekkeita ja numeroarvojen pyörittelyä. Rivillä **170** nähtävät v1[0], v2[1], v1[1] ja v2[0] -arvot viittaavat aiemmalla kahdella rivillä näkyviin laskuihin. V1[0]-viittaa rivin **168** sulkulausekkeen laskuun x2-x1, v1[1] taasen laskuun y2-y1 ja niin edelleen. Tässä nähdään jälleen esimerkki Pythonin tavasta järjestää listassa olevat luvut alkamaan indeksistä nolla. Pilkku toimii listassa erottimena.

```
176     # Lasketaan korjaus riippuen pisteen sijainnista suoran suhteen
177     if x_risti <= 0:     # X suoran alapuolella
178         # Lasketaan etäisyys suorasta
179         et = (abs(fx_x*x_koord+fy_x*y_koord+fc_x))/((fx_x**2+fy_x**2)**0.5)
180         et1 = int(et)
181         # Lasketaan korjaus
182         max_etäisyys = 75
183         max_korjaus = 2
184         x_korjaus = x_koord-(et/max_etäisyys)*max_korjaus
185         x_korjattu=int(x_korjaus)
186         x_korjaus1=(et/max_etäisyys)*max_korjaus
187         print('Korjattu x: ', x_korjattu)  # x_korjattu = tämä robotille
188     else:   # X suoran yläpuolella
189         et = (abs(fx_x*x_koord+fy_x*y_koord+fc_x))/((fx_x**2+fy_x**2)**0.5)
190         et1 = int(et)
191         max_etäisyys = 180.8
192         max_korjaus = 9
193         x_korjaus = x_koord+(et/max_etäisyys)*max_korjaus
194         x_korjattu=int(x_korjaus)
195         x_korjaus1=(et/max_etäisyys)*max_korjaus
196         x11 = round(x_korjaus1)
197         print('Korjattu x: ', x_korjattu) # x_korjattu = tämä robotille
```

Samaan tapaan tässäkään osiossa ei ole matemaattisten lausekkeiden lisäksi mitään aiemmasta poikkeavaa, ohessa kuitenkin ehtolausekkeiden osalta selvennys. Python käyttää ehtolauseiden jälkeen sisennystä erottamaan ne rivit, jotka if-lauseeseen sisältyvät. Seuraavassa kuvassa havainnollistava esimerkki.

```
1  x = 1
2  if x == 1:
3      print('X on yksi')
4  else:
5      print('X ei ole yksi')
6  y = 2
```

Tässä tapauksessa if-lauseen sisään jäisi siis vain rivi **3**. Else-kohdan alle jäisi vain rivi **5**. Viimeinen rivi olisi ehtolausekkeiden ulkopuolella, eikä näin siis vaikuttaisi tarkasteluun. Aiemmin esitetty kompensointiosio on identtinen y-akselin osalta, joten sitä ei tässä liitteessä ole tarvetta tarkastella.

```
239 '''---------------Tietojen lähetys robotille----------------'''
240 def tietojen_lähetys():
241     # Halutessaan voi lähettää myös seuraavat testiarvot:
242     x_akseli=330     # X-akselin testiarvo  (normaalisti kuvasta)
243     y_akseli=-150    # Y-akselin testiarvo
244 
245     ''' Robotin ongelmallisen koordinaatiston vuoksi noutokorkeutta (z-akseli)
246     pitää säätää y-akselin koordinaatin perusteella; jos kappale on keskikohdan
247     oikealla puolella --> mennään alemmas'''
248     if y_korjattu >= -58:
249         z_akseli= -193
250     else:
251         z_akseli = -192
252 
253     z_akseli_ylä = -150      # Z-akselin ylä-asento
254     value4 = 0               # X - euler
255     value5 = 0               # Y - euler
256     value6 = 0               # Z - euler
257     # Muunnetaan numeeriset arvot tekstiksi 123 -> '123'
258     num1=str(x_akseli)
259     num2=str(y_akseli)
260     num3=str(z_akseli)
261     num4=str(value4)
```

Oheisilta riveiltä voidaan huomioida rivit **258** eteenpäin. Komennolla str() voidaan muuntaa, tässä tapauksessa kokonaislukumuodossa olevat muuttujat tekstityyppisiksi. Tällöin esimerkiksi muuttujan z_akseli_ylä -muuttuja muunnetaan muodosta -150 muotoon ”-150”. Tämä muunnos vaaditaan robotille lähetystä varten. Kaikki lähtevät koordinaatit muunnetaan siis tekstimuotoon.

```
269      # Valmiiden koordinaattien lähetys robotille
270     print('-------Lähetys--------')
271     print('Lähetetään arvot robotille')
272     s.send(bytes(num8, 'utf-8'))       # (koordinaatti, merkistökoodaus)
273     # Lähetysten välissä odotetaan hetki, jottei arvot mene sekaisin
274     time.sleep(1)
275     print('X -->: {}'.format(num8))
276     data_pyyntö()
277     s.send(bytes(num9, 'utf-8'))       #testiarvo num2, y_koord=num9
278     time.sleep(1)
279     print('Y -->: {}'.format(num9))
280     data_pyyntö()
281     s.send(bytes(num3, 'utf-8'))
282     time.sleep(1)
283     print('Z -->: {}'.format(num3))
```

Rivillä **272** käytetään socket-moduulin send-komentoa koordinaattien lähetykseen robotille (https://docs.python.org/3/library/socket.html). Tässä kohtaa aiemmin tekstiksi muunnetut muuttujat muutetaan jälleen kerran, tällä kertaa tavuiksi (byte). Send-komento itsessään ottaa tässä kohtaa kaksi parametria: lähetettävä muuttuja, merkistökoodaus (https://fi.wikipedia.org/wiki/Unicode#UTF-8). Koordinaatin lähetyksen jälkeen odotetaan yksi sekunti, jonka jälkeen tulostetaan sekä lähtenyt, että robotilta palannut koordinaatti, kuten tehty rivillä **276**. Tässä kohtaa ajetaan ohjelman alussa luotu data_pyyntö -funktio, jolloin koodi odottaa tietoa robotilta saapuvaksi. Tässä tilanteessa vastauksena on sinne juuri aiemmin lähetetty koordinaatti. Näin voidaan tarkkailla lähteneitä ja saapuneita koordinaatteja ja havaita mahdolliset ongelmat tiedonsiirrossa.

```
306     print('Arvot lähetetty, odotetaan vahvistusta robotilta')
307     data_pyyntö()
308     ''' Kun arvot on lähetetty, koodi odottaa 'VALMIS'-komentoa robotilta
309         ennen kuin uusi kuva otetaan'''
```

Kun kaikki koordinaatit on lähetetty robotille, alkaa robotti ne saatuaan noutaa kappaletta työalueelta, jonka jälkeen se palaa alkuperäiseen kuvausasentoonsa ja lähettää perille saapumisestaan ilmoituksen takaisin Raspberry Pille. Tämän aikaa Python-koodi odottaa rivin **307** mukaan varmistusta robotilta, jonka jälkeen uusi ohjelmakierto voidaan aloittaa.

```
314 while toisto == True:
315     print('Aloitetaan alusta')
316     kuvan_otto()
317     kuvan_lataus_ja_muokkaus()
318     aariviivat()
319     keskipisteen_laskenta_ja_naytto()
320     värintunnistus_ja_lajittelu()
321     koordinaattien_muutos()
322     virheiden_kompensointi()
323     tietojen_lähetys()
```

Viimeisenä toiminnassa olevana ominaisuutena käytetään while-komentoa. Rivillä **90** luotu, aina TRUE-tilassa oleva toisto-muuttuja aloittaa ohjelmakierron alusta, kun robotti on lähettänyt kuittauksen päästessään aiemmasta kappaleenhausta perille.

```
325 def sulje_socket():
326         # Socketin sulkeminen
327         s.close()
```

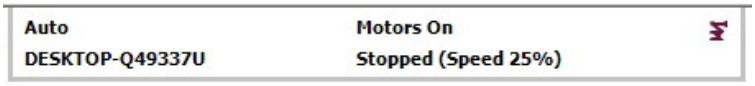
Loppuun on muodostettu varsin yksiselitteinen funktio socket-yhteyden sulkemiselle. Rivin **327** close-komennolla muodostettu yhteys katkaistaan. Tätä funktiota ei lopullisessa työssä käytetty lainkaan, koska robotti toimii jatkuvalla kierrolla, ellei ohjelmaa katkaista erikseen tai synny esimerkiksi törmäystilanne, jolloin robotin liike automaattisesti lakkaa. Tätä funktiota voidaan kuitenkin halutessaan käyttää laitteen jatkokehityksessä.

LIITE 2/1

LAITTEISTON LYHYT KÄYTTÖOPAS

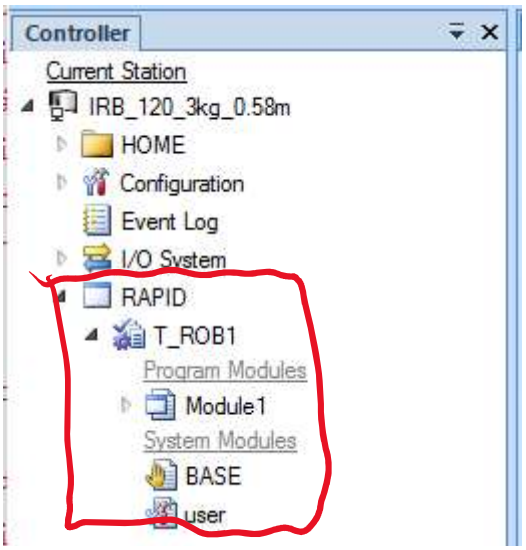
ROBOTTI JA ROBOTSTUDIO

- Laitetta voidaan käyttää joko suoraan Flexpendantista tai vaihtoehtoisesti RobotStudio-sovellusta käyttäen
- **Robotin ohjelman on aina oltava ensin käynnissä, ennen kuin RasPissa olevaa koodia voidaan käynnistää**
- Robotti on oltava automaattiajolla, sekä moottorien oltava päällä, jotta Flexpendantissa oleva ohjelma voidaan ajaa



- **FlexPendantia** käytettäessä riittää, kun *Production Window* -kohdassa on painettu ”***PP to main***” -nappia. HUOM. Tämä täytyy tehdä jokaisella kerralla, kun ohjelma aiotaan ajaa, jotta kursori palaa ohjelman alkuun ja socket-yhteys voidaan muodostaa!
- **Start**, ohjelma käynnistyy

- **RobotStudion** kautta ajettaessa tulee ensin käynnistää jokin luokassa oleva pöytäkone ja yhdistää se robotin kanssa samaan langattomaan verkkoon. Vakiona: dlink-2E7F tai ROBOTTI-nimellä, salasana: sahkolabra
- Käynnistä RobotStudio
- Jos robotti löytyy verkosta, voi sen lisätä projektiin kohdasta File – Online
- Valitse välilehti RAPID ja sieltä painike ***Request Write Access***, hyväksy toiminto tarvittaessa FlexPendantista
- JOS konenäkösovellus ei ole ladattuna FlexPendantiin, tulee RAPID-koodi kopioida (esim. serveri.txt tiedoston sisältö) sivupalkin RAPID – T_ROB1 – Module1 -kohtaan. Tuplaklikkaa Module1-tekstiä, jolloin RAPID-editori aukeaa ja kopioi teksti sinne. Kts. kuva alla



- Kun oikea sovellus on sisällä, paina **APPLY**-painiketta
- RAPID-valikosta ***Program Pointer – Set Program Pointer to Main in all tasks***
-
- Robotin nopeus FlexPendantissa on vakiona 100%, tämä kannattaa turvallisuussyistä vaihtaa pienempään arvoon, testauksessa käytetty arvoa 25%
- Paina **Start**, mikäli ohjelmassa ei ole virheitä, robotti käynnistyy ja ajaa itsensä alkupositioonsa ja RAPID-koodi jää odottamaan yhteydenottoa RasPilta
- Mikäli ohjelma antaa socket-aiheisen virheen, on RAPID-koodissa olevat IP-osoitteet todennäköisesti väärin, robotin IP-osoitteen löytää FlexPendantin *System info – Controller Properties – Network Connections*-kohdasta, muuta täältä löytyvä IP-osoite RAPID-koodiin kaikkialle, mihin se on merkitty (avaa RAPID-koodi ja paina Ctrl + F, tee haku ”192.”, jolloin tarvittavat sijainnit löytyvät)

  HUOM.
  Kun kumpi tahansa koodeista (Python/RAPID) on keskeytetty tai laite on virhetilanteen vuoksi pysähtynyt, voidaan laite käynnistää uudelleen VAIN seuraavalla tavalla:

- Robotstudio / FlexPendant, ***Program pointer to main*, Start**

RASPBERRY PI

- Jonkin luokan pöytäkoneen ollessa käynnissä JA samassa langattomassa verkkoyhteydessä robotin kanssa, kytke RasPin virtajohto
- Käynnistä Windowsissa VNC Viewer-sovellus
- RasPi käynnistyy muutaman minuutin, jonka jälkeen siihen saa yhteyden VNC Viewerillä, oletuksena RasPin IP-osoite on 192.168.0.101 (jos ei tällä löydy, kokeile muuttaa IP-osoitteen viimeistä numeroa esim. 100,103 ym. tai etsi RasPin IP-osoite IP scannerilla)
-

- Kun RasPi on yhdistettynä VNC Viewer -ohjelmaan, avaa Thonny Python IDE ja lataa siihen konenäköskripti (ESIM. ”Konenäkö – ABB IRB 120.py”)
- Tarkista skriptistä, että yhteydenmuodostus-osiossa simulointirivi on kommentoituna pois (#) ja ’robotti’-rivi ei. Tarkista myös robotin IP-osoite, porttia ei tarvitse muuttaa

```
41  s.connect(('192.168.0.101', port))        #robotti
42  #s.connect(('192.168.0.100', port))       #simulointi (Kommentoi aina toinen pois)
```

- Robotissa oleva ohjelma tulee käynnistää aina ensin, jotta yhteydenmuodostus onnistuu
- Kun robotin ohjelma on käynnissä, paina Thonny Idessä painiketta **START**, ohjelma käynnistyy, yhteydenmuodostus alkaa ja sen onnistuessa kamera ottaa ensimmäisen kuvan

HUOM.

Muista tarkkailla robotin liikettä jatkuvasti sen ollessa käynnissä. Noudata erityistä varovaisuutta, jos siirrät haettavia kappaleita takaisin työalueelle ennen uuden kuvan ottoa.