

2013•2014
FACULTEIT INDUSTRIËLE INGENIEURSWETENSCHAPPEN
master in de Industriële wetenschappen: energie

Masterproef

Vision-based control of robotic arm with 6 degrees of freedom

Promotor :
ing. Eric CLAESEN

Promotor :
ing. ROMAN CAERMK

Wim Versleegers

Proefschrift ingediend tot het behalen van de graad van master in de industriële wetenschappen: energie

Gezamenlijke opleiding Universiteit Hasselt en KU Leuven

2013•2014
Faculteit Industriële
ingenieurswetenschappen
master in de Industriële wetenschappen: energie

Masterproef

Vision-based control of robotic arm with 6 degrees of freedom

Promotor :
ing. Eric CLAESEN

Promotor :
ing. ROMAN CAERMK

Wim Versleegers

Proefschrift ingediend tot het behalen van de graad van master in de industriële wetenschappen: energie

Foreword and acknowledgements

This thesis is written as a completion for my final master year at UHasselt (University of Hasselt) at the department of FIIW, the Faculty of Industrial Engineering Sciences. My master year specializes in Energy and Automation.

The research was performed at Západočeská Univerzita v Plzni (the University of West Bohemia) in Czech Republic, during the Erasmus Exchange program. The thesis was done at the Mechanical and Cybernetics department.

The performed research consists of the control of a vertical articulated robotic arm with six degrees of freedom. The most important part of this thesis was the ability to be able to connect an USB webcam to the robot, so that vision applications were possible with the robot.

The research has started in February 2014. The topic was a very interesting and educational process, because it is easy to see the practical applications of my research. When I started this project my knowledge of programming robots was very limited. This assignment has taught me a lot of the problems occurring with programming robots. The vision part of the project has given me a lot more insight of what is necessary to receive useful signals from a camera.

Even with all the problems I've encountered while programming the robot, I am satisfied with the booked results. Therefore, I would like to thank my supervisors Eng. Roman Čermák, Ph.D from the University of West Bohemia and Eng. Eric Claesen from the University of Hasselt. They helped me achieving this goal and came with a lot of great suggestions. Also I would like to thank mister Edwin van Baar, the technical account manager Industrial Automation from Mitsubishi Benelux. He provided me with useful information regarding the communication with the robot and without him I would not have been able to become these results.

Abstract

This paper studies the procedure to program a vertically articulated robot with six degrees of freedom, the Mitsubishi Melfa RV-2SD, with Matlab. A major drawback of the programming software provided by Mitsubishi is that it barely allows the use of vision-based programming. The amount of useable cameras is limited and moreover, the cameras are very expensive. Using Matlab, these limitations could be overcome. However there is no direct way to control the robot with Matlab. The goal of this project is to set up a serial connection between the robot and Matlab. The procedure is investigated for a case in which the robot has to pick up geometrical objects and sort them. The camera, connected via a USB-connection to the PC, provides an image of the working field. This image is processed in Matlab. The resulting data is transformed to a trajectory with the different passing points. These points are converted to the correct format to send to the robot in a command through the serial connection. The results are simulated and performed with the actual robot.

Abstract in Dutch

Deze paper bestudeert de procedure om een verticaal gelede robot met zes vrijheidsgraden, de Mitsubishi Melfa RV-2SD, met Matlab te programmeren. Een grote tekortkoming van de software van Mitsubishi is dat de mogelijkheden omtrent visie erg beperkt zijn. Er zijn maar enkele camera's mogelijk om te gebruiken en deze zijn bovendien erg duur. Door gebruik te maken van Matlab is het wel mogelijk beeldverwerking met eender welke camera te doen, maar er is geen rechtstreekse methode om de robot aan te sturen met Matlab. Het doel van dit project is om een seriële connectie tussen de robot en Matlab tot stand te brengen. De hele procedure is onderzocht voor het geval waar de robot geometrische objecten moet opnemen en deze moet sorteren. De camera, verbonden via een USB-connectie met de PC, biedt een beeld van het werkveld aan. Dit beeld wordt verwerkt in Matlab tot een traject met de verschillende tussenpunten die de robot moet doorlopen. Deze punten worden dan geconverteerd naar een formaat dat kan worden doorgestuurd naar de robot via de seriële connectie. De resultaten worden gesimuleerd en uitgevoerd met de echte robot.

Table of contents

Foreword and acknowledgements	1
Abstract	3
3 Introduction.....	15
3.1 Problem statement.....	15
3.2 Objectives.....	16
3.3 Materials and methods	16
4 Hardware	17
4.1 Robot.....	17
4.2 Environment.....	18
4.2.1 Robot setup.....	18
4.2.2 Camera frame.....	18
5 Software	21
5.1 Matlab.....	21
5.1.1 Image Acquisition Toolbox.....	21
5.1.2 Digital Image Processing Toolbox.....	21
5.1.3 Peter Corke Robotics Toolbox.....	22
5.1.4 Mitsubishi Melfa Toolbox.....	23
5.2 RT Toolbox2.....	23
6 Image acquisition.....	25
7 Image processing	27
7.1 Preprocessing of the image	27
7.1.1 Crop the image	27
7.1.2 Convert to black and white.....	27
7.1.3 Enhance the image.....	28
7.2 Shape recognition	29
7.2.1 Bounding box and extent	30

7.2.2	Compactness	31
7.3	Conversion coordinates.....	32
7.4	Applied to the test setup	33
7.4.1	Preprocessing.....	33
7.4.2	Shape recognition	35
8	Trajectory planning	39
8.1	Determining the z-coordinates.....	39
8.2	Determining other points of trajectory	39
8.3	Applied to the test setup	41
8.4	Converting to joint coordinates.....	42
8.4.1	Orientation.....	42
8.4.2	Inverse kinematics.....	42
8.5	Wrist downward singularity	43
9	Robot definition.....	45
9.1	Joint limits.....	45
9.2	Denavit-Hartenberg parameters.....	45
10	Communication	49
10.1	Serial communication	49
10.2	Bi-direcional communication.....	49
10.3	RS-232 protocol.....	49
10.4	Protocol for communication with robot.....	50
10.4.1	Transmit data.....	50
10.4.2	Receive data	54
11	Simulation in Mitsubishi Melfa Toolbox.....	55
11.1	Workflow communication	55
11.1.1	Open connection.....	55
11.1.2	Turn on control	56
11.1.3	Turn on servo.....	56
11.1.4	Set speed.....	56

11.1.5	Check if the servo is on.....	56
11.1.6	Perform movements	58
11.1.7	Close connection.....	59
11.1.8	Applied to test setup	59
11.2	Results.....	60
12	Simulation with RT Toolbox2.....	61
12.1	Communication workflow.....	61
12.1.1	Write program.....	62
12.1.2	Send program to controller.....	63
12.2	Results.....	64
13	Results simulations.....	65
14	Robot.....	69
14.1	Add serial port to Matlab.....	69
14.2	Workflow communication robot.....	70
14.3	Results.....	71
15	Conclusions.....	73
15.1	Communication with robot via serial connection.....	73
15.1	Shape recognition.....	73
15.2	Wrist singularity	74
15.3	Further works	74
16	References.....	75
17	Attachments	77

1. List of tables

Table 1: Joint limits (adapted from [1]).....	45
Table 2: Joint limits.....	48
Table 3: Simulations test setup.....	67
Table 4: Settings serial communication.....	69
Table 5: Serial commands for communication	70
Table 6: Results robot.....	72

2. List of figures

Figure 1: Mitsubishi Melfa RV2SD (adapted from [1])	17
Figure 2: Setup robot	18
Figure 3: Camera height.....	19
Figure 4: Interference robot and camera	19
Figure 5: Parents and holes (adapted from [6]).....	30
Figure 6: Origin transformation	32
Figure 7: Test setup	33
Figure 8: Cropped image	34
Figure 9: Black and white image	35
Figure 10: Boundaries	36
Figure 11: Marked image.....	37
Figure 12: Flowchart trajectory.....	39
Figure 13: Trajectory test setup	41
Figure 14: Wrist downward singularity: solution	44
Figure 15: Denavit–Hartenberg parameters (adapted from [8])	45
Figure 16: Determination DH parameters.....	47
Figure 17: Serial communication (adapted from [13])	50
Figure 18: Communication with robot (adapted from [14]).....	50
Figure 19: Command according to R3 protocol (adapted from [14])	52
Figure 20: NMEA checksum	53
Figure 21: Flowchart simulation Mitsubishi Melfa Toolbox.....	55
Figure 22: Simulation with RT Toolbox2	61

3 Introduction

Nowadays, robots are more important than ever before. A lot of robotic arms replace the simple labor. They are, for example, used for welding, spraying, assembly lines, etc. This results in faster and much safer labor, because less people have to work in the potential danger zones of the other machinery. On top of that, robots are independent of stress, illness and other human factors.

My final project work aims to program and control a vertical articulated robotic arm with six degrees of freedom to pick up and sort certain objects with Matlab. The big problem is that the Mitsubishi software limits the usage of vision-based control. Only Melfa vision cameras are supported. Not only does this limit the amount of cameras, they are also very expensive. The software is perfectly suited for fast programming in an industrial environment, but for the university, the robot is used for research. Therefore they want to be able to use cheap cameras. Matlab provides a lot of easy and powerful image processing functions and allows using every camera.

I will do this project work at the University of West Bohemia (Západočeská univerzita v Plzni). This university is located in Pilsen in the Czech Republic. The project work will be done at the department of Mechanical engineering and the department of Cybernetics.

The robotic arm is the MELFA RV-2SD from Mitsubishi and the programming will be done mainly in Matlab.

3.1 Problem statement

For a lot of robotic applications cameras are necessary to perform the desired tasks. For example, a robot needs to remove bad products on a product line. You need an image to see whether there are certain defects and eventually whether the product is bad and has to be removed.

The problem described in this thesis exists of a field that contains square, rectangular and circular objects that need to be sorted. The robotic arm should be able to pick up these objects and bring them to the correct place. This means that a procedure should be developed to localize and identify the objects on the field with a camera. When these properties of the objects are found, they must be sent to the program that handles the planning of the trajectory for the robotic arm.

To let the robot perform these moves, a communication between the robot and the computer, running Matlab, must be created, so that the necessary actions can be transferred to the robot and the robot can perform them.

3.2 Objectives

The main goal is to establish a communication between Matlab and the robot with a serial connection, because Matlab makes it possible to create vision applications with almost every camera.

In this case, the desired application is to let the robot pick up square, rectangular and circular shapes and to sort them.

The first objective is to place the camera in a good way to get a complete picture of the field. A method must be developed to process the image and recognize the shapes of the objects and to get their position. Next, their position can be used in the program to plan the trajectory. With the knowledge of the shapes, the robot knows where to bring the objects.

When the trajectory is known, the necessary movements can be performed. To do this, we have to establish a good communication between the robot and the computer, running Matlab. There are a few options to do this, like serial line, USB connection, Ethernet... In this case, our aim is to use a serial line connection.

3.3 Materials and methods

The main tools we need are:

- a robotic arm with a electromagnet attached to the end-effector;
- the camera to recognize and locate the different shapes and to check the position of the robot;
- the software to process the images coming from the camera;
- The software to control the robotic arm.

For the robotic arm we are using the robot from Mitsubishi, the MELFA RV-2SD. The robot is equipped with an electromagnet at the end of the arm, to pick up the metal objects.

For the camera we will use a simple USB connected webcam. This webcam is attached to a frame above the working area and is independent from the robot.

The images will be processed with Matlab and a toolbox called Image Processing Toolbox.

To communicate with the robot we will also use a toolbox for Matlab called Robotic Toolbox from Peter Corke. As an extra, there is the possibility to use RT Toolbox 2 instead, provided by Mitsubishi themselves. In that case, a connection between Matlab and the toolbox must be established. The communication will be done using a serial port.

4 Hardware

4.1 Robot

The robot used in this project is the Mitsubishi Melfa RV2-SDB. This robot is a vertical articulated robot with six degrees of freedom.

The robot has these six degrees of freedom because of the six rotating joints. The first three degrees of freedom determine the position of the end effector. The three remaining degrees of freedom determine the orientation of the end effector.

[Robot's Outer Dimensions and Motion Space]

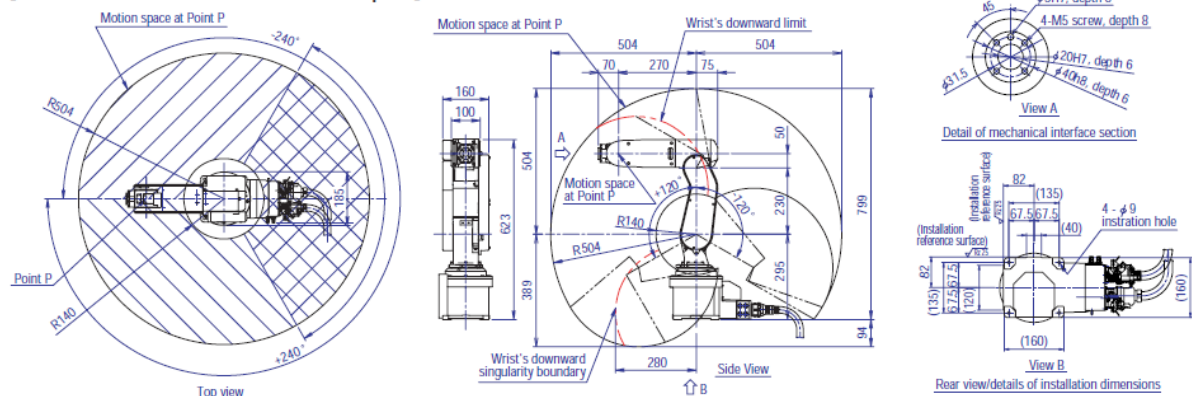


Figure 1: Mitsubishi Melfa RV2SD (adapted from [1])

The working area of the robot is visible in Figure 1, indicated by the outer black lines. In the horizontal plane (visible in the top view) the working area is described as a partial circle with a radius of 504mm. The robot is unable to rotate 360° degrees around joint 1, the vertical rotating axis through the base point. It is limited between +/-240°. Also the robot cannot reach near its base in the circle with radius 140mm without encountering a wrist's downward singularity. This means when the wrist is faced downwards and the end effector is send near the base of the robot, the 4th joint has to rotate 180° all of a sudden, for the robot to be able to reach closer to the base while keeping the wrist downwards.

In the vertical plane, the robot can only move between the partial circles with a radius of 140mm and 504mm. The second joint is limited between +/-120° in relation to the vertical axis. The red lines in the side view of the robot in Figure 1 indicates the limits of the working area where the wrist singularity occurs.

The position repeatability is $\pm 0.02\text{mm}$, which means that when the robot is sent to the same position multiple times, the difference in positions is maximum 0.02mm.

The robots rated mass load capacity is 2kg. However, with the wrist positioned downwards, the maximum load capacity is 3 kg.

4.2 Environment

4.2.1 Robot setup

The robot is mounted on a table, as shown in visible in Figure 2.

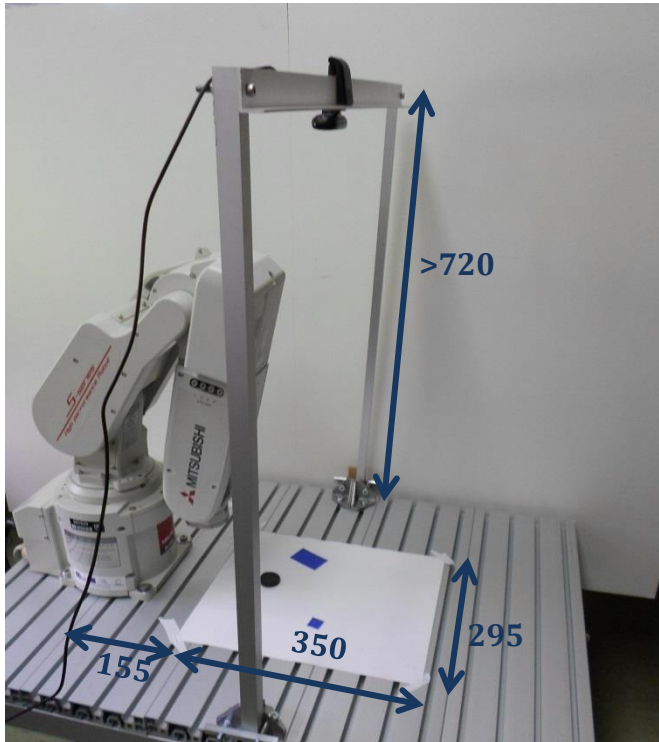


Figure 2: Setup robot

The working field has a width of 295mm and a height of 350mm. This corresponds to the white area on the table. The distance in the x-direction between the robot base and the start of the working field is 155mm. In the y-direction the working field is centered.

4.2.2 Camera frame

The camera has to take a picture so that the field of view matches the working field as close as possible. However, the camera should be out of the reachable zone of the robot.

The USB camera used is the Logitech C170. This camera has a diagonal field of view of 58° . The aspect ratio is 4:3. The minimum height of the camera, when it is centered with the working field, can be calculated. With Figure 3, this is determined.

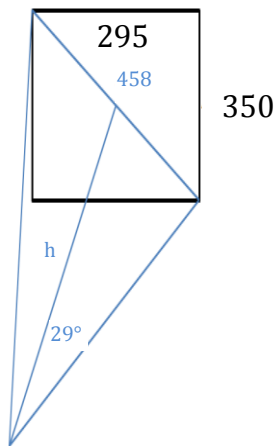


Figure 3: Camera height

$$2a = \sqrt{295^2 + 350^2} = 458$$

$$a = 229\text{mm}$$

$$h = \frac{a}{\tan(29^\circ)} = \frac{229}{\tan(29^\circ)}$$

$$h = 413\text{mm}$$

Thus, the minimum height of the camera to capture the complete working area is 413mm.

Figure 4 shows the robot in its stretched position with the end-effector at the same x-position as the camera frame.

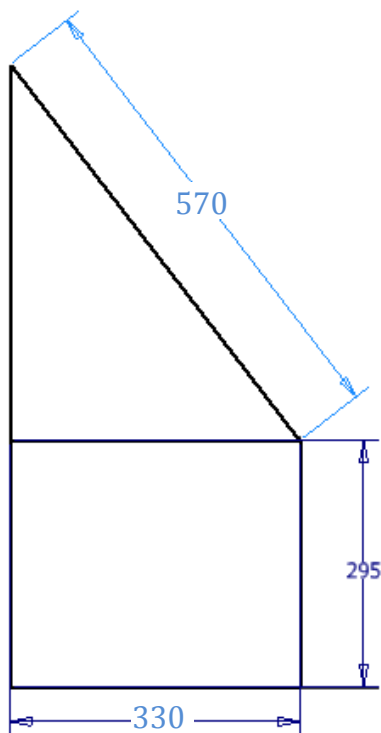


Figure 4: Interference robot and camera

Since the camera is centered above the working area it is

$$155 + \frac{350}{2} = 330mm$$

away from the robot's origin. When the elbow is stretched, the robot's arm is 570mm long. The offset of 50mm after the elbow of the robot is neglected.

The maximum height of the robot reached at this point is:

$$295 + \sqrt{570^2 - 330^2} = 760mm > 413mm$$

Thereby, the frame of the camera should be at least 720mm high.

5 Software

5.1 Matlab

MATLAB® is a high-level language and interactive environment for numerical computation, visualization, and programming. Using MATLAB, you can analyze data, develop algorithms, and create models and applications. The language, tools, and built-in math functions enable you to explore multiple approaches and reach a solution faster than with spreadsheets or traditional programming languages, such as C/C++ or Java™ [2]

For this project Matlab is used to do the image processing and for controlling the robot. The camera will be connected to the computer with USB and the robot will be connected with a serial line. Matlab will control these communications with the help of some additional toolboxes, which are described in paragraphs 5.1.1 - 5.1.4.

5.1.1 Image Acquisition Toolbox

To import the data from the camera to the Matlab software, the Image Acquisition Toolbox is used. This toolbox makes it possible to import pictures or videos, so they can be further processed with the Digital Image Processing Toolbox.

5.1.2 Digital Image Processing Toolbox

The Digital Image Processing Toolbox for Matlab makes it possible to process the images to useful data. The images provided from the camera come with both useful and useless data. For this application, the only useful data is where the objects are and what their shape is. The colors, background, etc. is all useless information which slows down the program. That is why it is important to process the image to the information that is needed.

5.1.3 Peter Corke Robotics Toolbox

“The Toolbox has always provided many functions that are useful for the study and simulation of classical arm-type robotics, for example such things as kinematics, dynamics, and trajectory generation. The Toolbox is based on a very general method of representing the kinematics and dynamics of serial-link manipulators.

These parameters are encapsulated in MATLAB® objects - robot objects can be created by the user for any serial-link manipulator and a number of examples are provided for well know robots such as the Puma 560 and the Stanford arm amongst others. The Toolbox also provides functions for manipulating and converting between datatypes such as vectors, homogeneous transformations and unit-quaternions which are necessary to represent 3-dimensional position and orientation.

...

Advantages of the Toolbox are that:

- the code is quite mature and provides a point of comparison for other implementations of the same algorithms;
- the routines are generally written in a straightforward manner which allows for easy understanding, perhaps at the expense of computational efficiency. If you feel strongly about computational efficiency then you can always rewrite the function to be more efficient, compile the M-file using the Matlab compiler, or create a MEX version;
- since source code is available there is a benefit for understanding and teaching. “ [3]

5.1.4 Mitsubishi Melfa Toolbox

The Mitsubishi Melfa Toolbox is created at the Czech Technical University in Prague at the Faculty of Electrical Engineering. The website gives an explanation of the functions, which are shown below, but translated to English.

“This toolbox provides access to control robots from Matlab using a single interface for Mitsubishi Melfa robots. It moves the articulated robot in Cartesian coordinates.

In order to use the toolbox to control the robot, a definition file must be available. The robot definitions for the Mitsubishi Melfa RV-6S and the Mitsubishi Melfa RV-6SDL are provided. For others it is necessary to have their own definition.”
[4]

The Mitsubishi Melfa Toolbox uses functions of the Peter Corke Robotics Toolbox. It provides a simulation and a way of connecting between the PC and the robot. The big advantage is that all the other possible inputs of Matlab can be used, especially cameras.

5.2 RT Toolbox2

RT Toolbox2 from Mitsubishi is simple software coming with the robot. It allows to connect the robot quickly and to make programs and send them to the controller of the robot. The big problem with this software is, as mentioned before, that the software only allows vision applications with a limited amount of cameras.

The software also allows to do some quick simple tests of reachability. The jog function allows to go in straight motions in the Cartesian space. In this way, the limits of the robot can be found experimentally.

6 Image acquisition

To be able to localize and identify the objects in the working field, a USB camera is used. This camera is attached horizontally to a grounded frame and pointed down.

The data is imported from the camera to the Matlab software with the image acquisition toolbox. The code used to do this is as follows:

```
vid = videoinput('winvideo', 1); %connect the camera
set(vid, 'ReturnedColorSpace', 'rgb');
img = getsnapshot(vid); %saves the picture
```


7 Image processing

7.1 Preprocessing of the image

The main goal of the preprocessing of the image is to eliminate all the useless data. The desired resulting image is an image of the working field without the surroundings with black background and white objects with the least possible distortion.

In order to achieve this goal, the following steps are performed:

1. Crop the image;
2. Convert to black and white;
3. Enhance the image.

7.1.1 Crop the image

The webcam makes a picture which contains more than just the working field. To adjust the picture so that it matches the working field completely, the picture needs to be cropped and maybe rotated, in the case that the camera is mounted crooked. The cropping of the image is performed with the *imcrop* function provided by the Digital Image Processing Toolbox. The image gets cropped along with the border of the working field. With these two actions, the image matches the working field perfectly.

```
c = imrotate(c, angle);  
c = imcrop(img, [topLeftX topLeftY ...  
               bottomRightX bottomRightY]);
```

7.1.2 Convert to black and white

Since the colors are unimportant for this task and image enhancing is much easier with black and white images, the image gets converted to black and white

The function *im2bw* allows to do this.

```
bw = im2bw(c, level);
```

With *level* the threshold level, a value between 0 and 1, with 0 black and 1 white. Everything above this level becomes white and vice versa. The level depends on the application. If the objects are brighter than the background, the image also needs to be inverted.

When the contrast between the fore- and background is high enough and the distortion of the image is limited, it is possible to calculate the level automatically with the function *threshold*. For this function we need to convert the image to gray scale first.

```
bw = im2bw(c, graythresh(rgb2gray(c)));
```

7.1.3 Enhance the image

This part of the preprocessing differs for each application, but mostly consists of a noise removal- and sharpening operation.

7.1.3.1 Noise removal

There are several methods to reduce noise. In this case, most of the noise is 'Salt&Pepper-noise'. An effective way to remove this kind of noise is to use a Medianfilter. A medianfilter takes the median value (in this case ones and zeros, since the image is in black and white) of all the surrounding pixels, defined by a structural element.

For example the structural element could be a 5x5 matrix with only ones. The filter takes the 5x5 matrix around the current pixel to calculate the median and then replaces the current pixel with this value.

The result is that a single white pixel or small white zone in a black surrounding gets removed and vice versa.

The command in Matlab is *medfilt2(image, SE)* with SE the structural element.

7.1.3.2 Sharpening

There are several reasons why the images can be unsharp, for example lens defects. By using combinations of erosion and dilation, the image can get sharpened. The functions *imopen* and *imclose* from the Digital Image Processing Toolbox makes use of this principle.

“IM2 = imopen(IM,SE) performs morphological opening on the grayscale or binary image IM with the structuring element SE. The argument SE must be a single structuring element object, as opposed to an array of objects. The morphological open operation is an erosion followed by a dilation, using the same structuring element for both operations.” [5]

“IM2 = imclose(IM,SE) performs morphological closing on the grayscale or binary image IM, returning the closed image, IM2. The structuring element, SE, must be a single structuring element object, as opposed to an array of objects. The morphological close operation is a dilation followed by an erosion, using the same structuring element for both operations.” [6]

By using these two functions, the shapes will match their real shape better. Forgotten pixels are added again and pixels that should not be part of the shape get deleted again. This makes the borders of the shape sharper.

7.2 Shape recognition

Now that the image is completely converted to a useable binary image, it is possible to determine the borders of each shape with the function *bwboundaries*. This function is provided by the Digital Image Processing Toolbox.

“B = bwboundaries(BW) traces the exterior boundaries of objects, as well as boundaries of holes inside these objects, in the binary image BW. *bwboundaries* also descends into the outermost objects (parents) and traces their children(objects completely enclosed by the parents). BW must be a binary image where nonzero pixels belong to an object and 0 pixels constitute the background. The following figure illustrates these components.

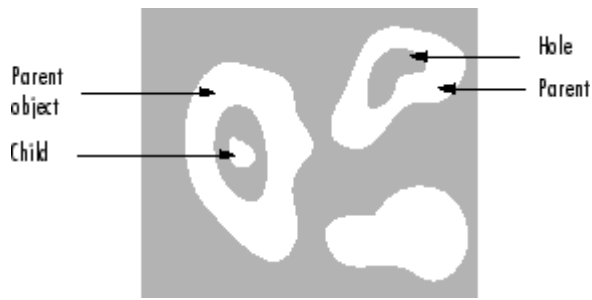


Figure 5: Parents and holes (adapted from [6])

`bwboundaries` returns B , a P -by-1 cell array, where P is the number of objects and holes. Each cell in the cell array contains a Q -by-2 matrix. Each row in the matrix contains the row and column coordinates of a boundary pixel. Q is the number of boundary pixels for the corresponding region." [7]

In this case, the holes need to be suppressed, since the objects do not contain any holes and they can affect the weighted center of the objects.

Once the boundaries are determined, all the objects are recognized as regions. With the function `regionprops`, a list with the properties of these regions is returned.

With these properties it is possible to determine whether the shape is a square, rectangle or circle and also what the coordinates of the center are.

There are several ways of determining what shape the object has. According to the properties returned by `regionprops`. In this thesis, two different methods are examined.

7.2.1 Bounding box and extent

The first possible solution is to use the `BoundingBox` function. This function creates the smallest rectangle that contains the whole shape. It returns the coordinates of the left-top corner and the width and height of the shape. If the width and height are equal, then it is a square or circle. If not it is a rectangle.

When the width and height is equal, the next test is to compare the area of the shape and the area of the bounding box. This can be done with the `Extent` function, provided by the Digital Image Toolbox. It returns a scalar between zero and one. If the returned value is one, the bounding box has the exact same shape as the region. This is in case of a square. In case of a circle, the extent has the next value:

$$extent = \frac{area\ circle}{width(bb) * height(bb)} = \frac{\frac{\pi * d^2}{4}}{width(bb)^2}$$

$$d = width(bb)$$

$$extent = \frac{\pi}{4} \approx 0,79$$

This means that if the returned value is higher than 0,79, it is a square, regarding that there are no other shapes in the field as rectangles, squares and circles. A safer border is that if the extent is higher than 0,95 it is a square.

A big limitation of this method is that the standard *BoundingBox* function can only create a rectangle which is oriented horizontally or vertically. This is a problem because the shapes are oriented randomly.

There is the possibility to create an advanced Bounding Box method, which uses the *Orientation* function to find the orientation of the rectangle. With this information it finds a Bounding Box that can also be rotated. Another solution would be to rotate the shapes in the image so that they are horizontal and then use the *BoundingBox* function.

7.2.2 Compactness

The incompactness of a 2D object is defined as the ratio between the square of the perimeter and the area.

$$c = \frac{P^2}{A}$$

A circle is the most compact 2D figure that exists. This means that the value for c is the lowest.

$$\begin{aligned} c &= \frac{P^2}{A} = \frac{(2 * \pi * r)^2}{\pi * r^2} \\ &= \frac{4 * \pi^2 * r^2}{\pi * r^2} \\ &= 4 * \pi \end{aligned}$$

With r the radius of the circle.

A square is the most compact figure of the quadrangles. It can be calculated as follows:

$$\begin{aligned}
 c &= \frac{P^2}{A} = \frac{(4 * a)^2}{a^2} \\
 &= \frac{16 * a^2}{a^2} \\
 &= 16
 \end{aligned}$$

With a the length of the edge of the square. The result of this, is that all the rectangles and other quadrangles have an incompactness bigger than 16.

7.3 Conversion coordinates

The coordinates of the objects acquired by the image processing, described in the previous chapter, are not the Cartesian coordinates according to the origin of the robotic arm. The positions are expressed in pixels and not in millimeters. The coordinates should be converted to the robot coordinates in millimeters as follows:

$$x_{ar}[mm] = x_a[pixels] * \frac{\text{width working field [mm]}}{\text{width image [pixels]}}$$

$$y_{ar}[mm] = y_a[pixels] * \frac{\text{height working field [mm]}}{\text{height image [pixels]}}$$

Next, the translation and rotation between the origin of the robot and the origin of the image should be compared.

As visible in Figure 6, the origin should be rotated 90° counter-clockwise and the y-axis should be inverted.

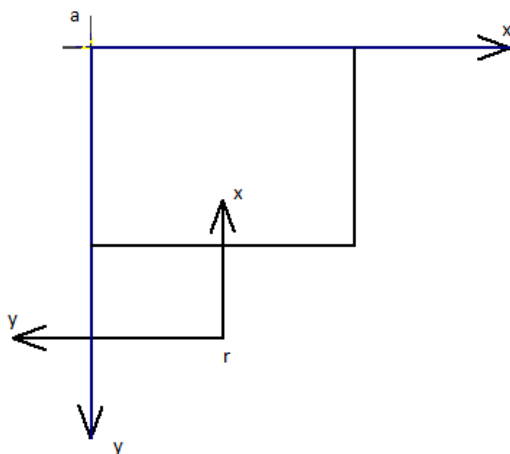


Figure 6: Origin transformation

$$x_{a''}[mm] = -y_{a'}$$

$$y_{a''}[mm] = -x_{a'}$$

Next it should be translated

$$x_r[mm] = x_{a''} - x_{tr}$$

$$y_r[mm] = y_{a''} - y_{tr}$$

With x_{tr} en y_{tr} the distance of the translation. y_{tr} is $\frac{1}{2}$ of the width of the image and x_{tr} the height plus the offset between the origin of the robot and the image.

7.4 Applied to the test setup

7.4.1 Preprocessing

For testing the image processing, a test setup was built where the webcam makes the following picture shown in Figure 7.

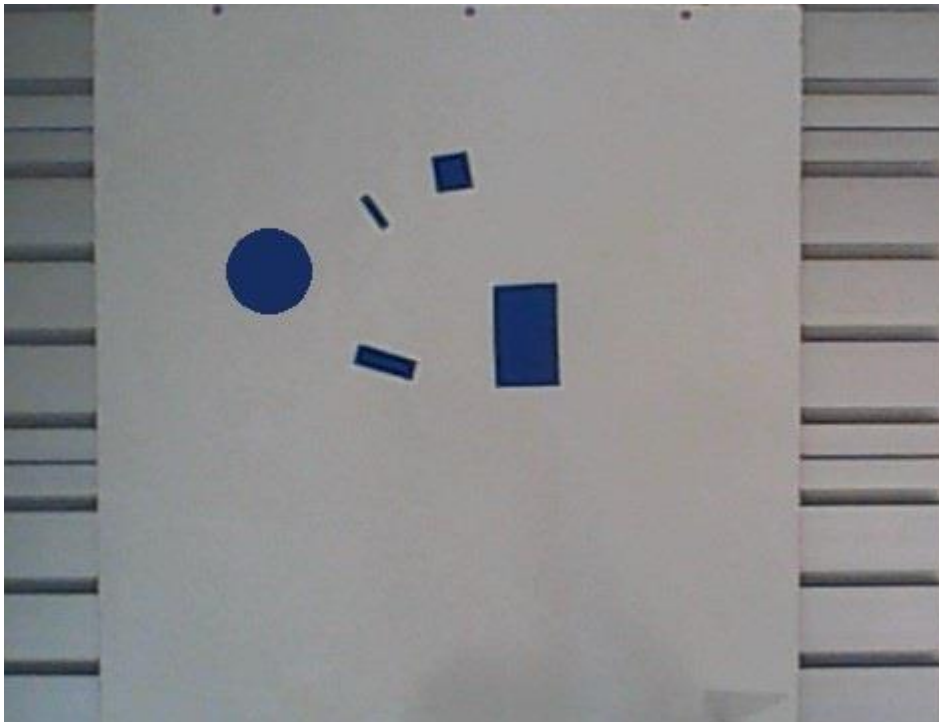


Figure 7: Test setup

Next, the image gets cropped to the actual working field. The following code is used.

```
c = imcrop(img, [302.5 180.5 227 191]);  
figure, imshow(c);
```

The results are shown in Figure 8.

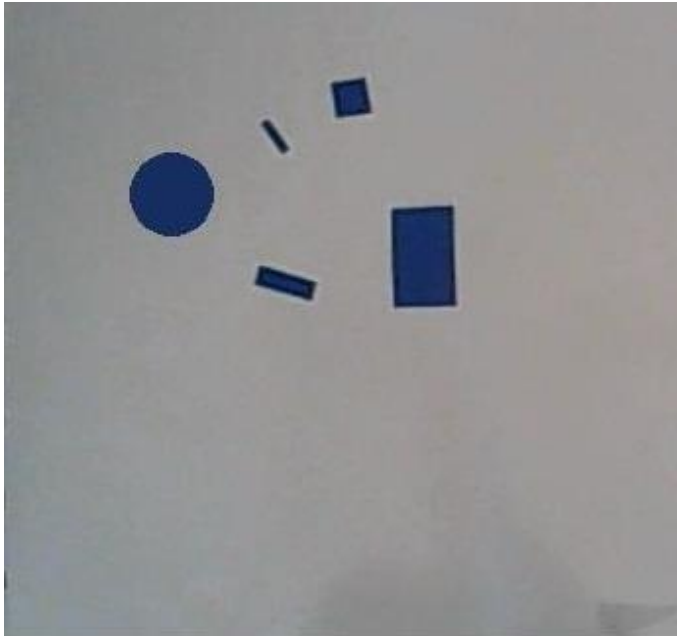


Figure 8: Cropped image

Note that the size of the working field is unimportant, since this is just a test setup. When used with the robot, the cropped image needs to have the correct dimensions measured in millimeters.

This image should be converted to black and white. Since the contrast of the fore- and background is high, it is possible to use the *im2bw* command without any extra parameters. The function uses the standard level of 0,5.

Because the objects are darker than the background, the resulting image should be inverted, i.e. black becomes white and vice versa.

```
bw = im2bw(c);  
bw = ~bw;
```

To remove the remaining noise in the background, a median filter is applied. The structure element is a 5x5 matrix filled with ones.

```
bw = medfilt2(bw, [5, 5]);
```

Since the edges of the objects are blurred, also an *imopen* en *imclose* filter is applied. The result is that when a pixel of the border is missing or pixels around the border get respectively added or deleted, which results in a sharper image of the objects.

```
SE = ones(3); %create a structure element of 3x3 filled with
ones
bw = imopen(bw, SE);
bw = imclose(bw, SE);
imshow(bw);
```

These actions result in the image shown in Figure 9.

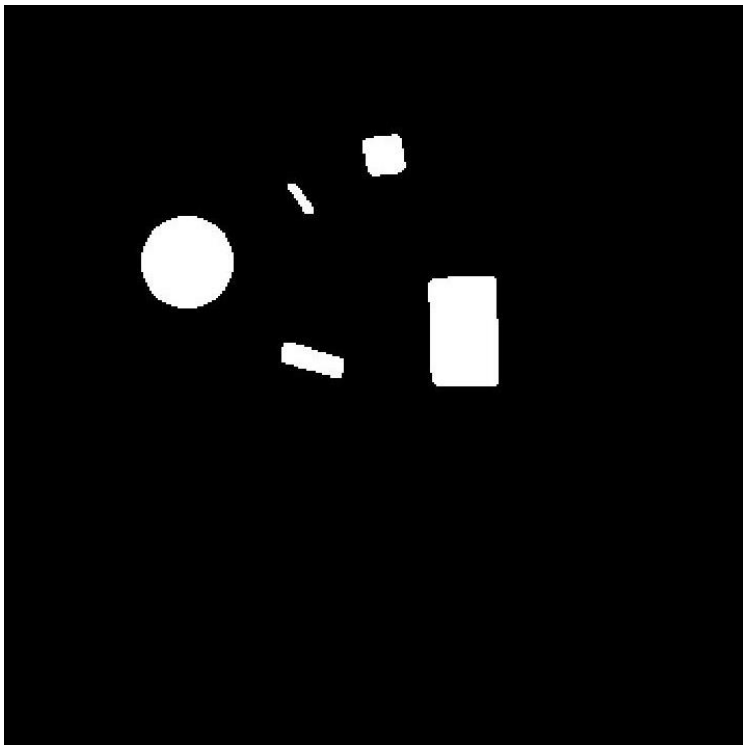


Figure 9: Black and white image

7.4.2 Shape recognition

Due to the limitation of the standard *BoundingBox* property and the complexity of the advanced bounding box method with rotation possibilities, the shape recognition will be performed with the *incompactness* function.

The program will search for the boundaries first. It does this with the function *bwboundaries*. This function defines the white regions in the black background. The 'noholes' argument in the function means that if the white objects contain black holes, they are ignored. Figure 10 shows the boundaries of the regions.

```
[B,L] = bwboundaries(bw, 'noholes');
```

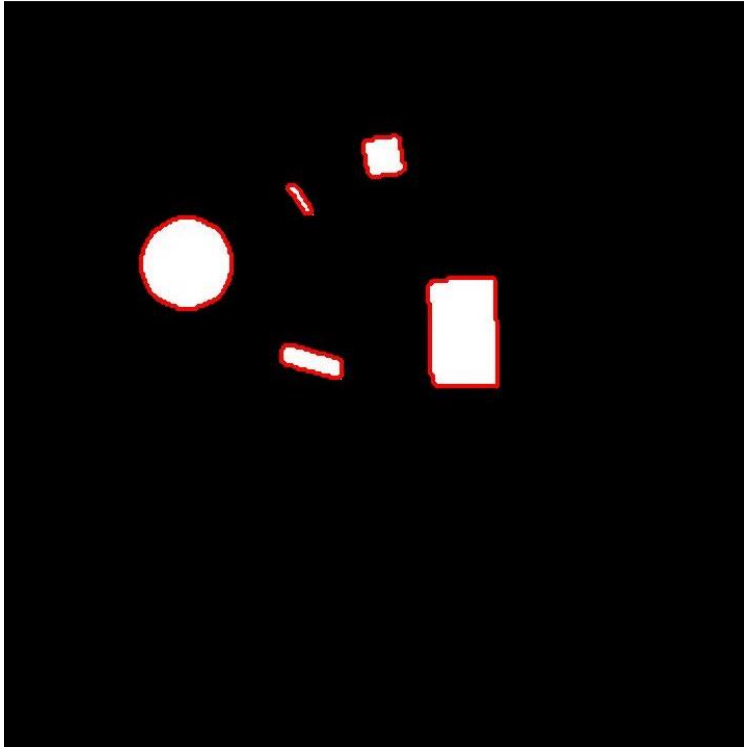


Figure 10: Boundaries

The width and height of the working field are set and the function *calculateObjectProperties* gets called.

```
%Set field width and height in mm.  
width = 295;  
height = 350;  
  
%Function to find all the objects and their properties.  
props = calculateObjectProperties(L, bw, width, height);
```

The function *calculateObjectProperties* first calls the resolution of the image. Then it puts the properties of the labels in the matrix *properties*.

```
%get image resolution  
[y, x] = size(bw);  
properties = regionprops(L, 'all'); %Call all the  
properties of the objects
```

Next, the shape gets recognized. A while loop is used to go through the *properties* matrix. The compactness of every object gets calculated and, according to the result, the shape gets assigned. The position is also saved to the props matrix.

```

for k = 1:length(properties)
    compactness(k) = properties(k).Perimeter^2 ...
        / properties(k).Area;           %calculate compactness

    if compactness(k)                       %check if circle
        props(k,1) = 1;                   %1 means circle

    elseif compactness(k) > 14 & ...
        compactness(k) < 17               %check if square
        props(k,1) = 2;                   %2 means square

    else                                     %else rectangle
        props(k,1)=3;                     %3means rectangle
    end

    props(k,2) = properties(k).Centroid(1); %x-position
    props(k,3) = properties(k).Centroid(2); %y-position
end

```

Figure 11 shows the cropped image with the regions marked on their center.

- X for rectangle;
- □ for squares;
- O for circles.



Figure 11: Marked image

The coordinates of the positions need to be converted to coordinates in mm and according to the origin of the robot, as explained in chapter 7.3.

```
%convert pixels to mm
props(:,2) = width / x * props(:,2);
props(:,3) = height / y * props(:,3);

%translating to the correct origin
props(:,2) = props(:,2) - width/2;
props(:,3) = height + 150 - props(:,3);

%rotating and mirror axis and create the z-coordinates
props(:, [2,3]) = props(:, [3,2]);
props(:,3) = -props(:,3);
```

The results are also displayed in the next matrix:

```
props =
    1.0000    399.1134    75.2201
    3.0000    350.0660    25.9410
    3.0000    431.0931    30.4635
    2.0000    452.5810    -2.4149
    3.0000    364.4579   -33.8775
```

With the first column the property of the shape. This is 1 for a circle, 2 for a square and 3 for a rectangle. The second and third column returns respectively the x- and y-coordinate in pixels from the center of the object.

8 Trajectory planning

8.1 Determining the z-coordinates

Since all the objects are placed on the working table, all the objects have a z-position of 0mm. However, since the objects have a certain thickness, the z-position of the end-effector had to be *thickness* mm.

8.2 Determining other points of trajectory

Figure 12 shows the flowchart for the trajectory that the end-effector of the robotic arm has to follow.

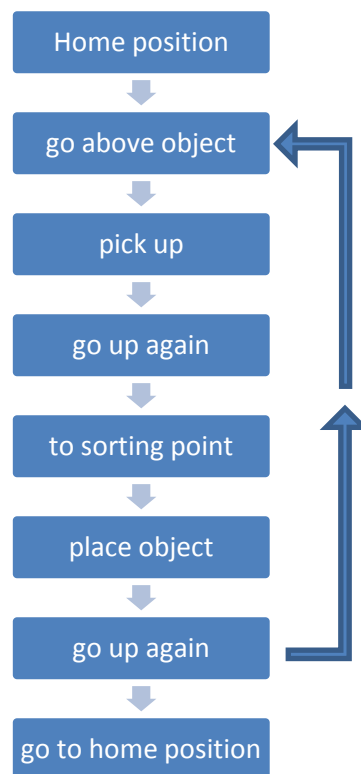


Figure 12: Flowchart trajectory

From the home position, the end-effector of the robotic arm should move to exactly above the first object. The coordinates from this point is $(x_i, y_i, thickness + offset)$. Then it should go down to grab the object with coordinates $(x_i, y_i, thickness)$. It should move up again, to the same point as before.

Subsequently it should move to exactly above the sorting point. The coordinates of this point depends on the shape of the object *i*. The z-coordinate again equals to *thickness + offset*. Next it goes down. The z-position now equals to *amount * thickness* with *amount* the number of objects that are already placed on the corresponding sorting point.

The function *calculateTrajectoryAndSpeed* is the function that calculates the trajectory points.

The documentation of the function is shown below.

This function creates a $3 \times (n \times 6)$ matrix defining the trajectory with *n* the amount of objects.

Arguments:

```
props:      n x 3 matrix with in the first
             column the shape, second the x position
             and third y position in the robots origin
T1:         1x2 matrix with x and y position
             for circles
T2:         1x2 matrix with x and y position
             for squares
T3:         1x2 matrix with x and y position
             for rectangles
offset:     height (z value) to move above objects.
heightField: height of the surface with the shapes on top
thickness:  thickness of object in mm.
```

Return values:

```
path:      The (n*3) x 3 matrix that contains all the
             xyz-coordinates of the points of the
             trajectory.
```

Basically, what this function does, is running a for-loop for each object. The next six points of the trajectory gets calculated for each object, according to Figure 12

The complete code can be found in 17.1.3.

8.3 Applied to the test setup

For the test setup, the matrix *props* from 7.4.2 is used. The offset is set to 100mm, the thickness of the objects is set to 1mm.

The coordinates for the sorting points are as followed:

- T1 = [150, 250] for circles;
- T2 = [150, -250] for squares;
- T3 = [150, 0] for rectangles

The results of the trajectory with the test setup, discussed in 7.4, are plotted in Figure 13. A clearer overview can be found in the chapters 11 and 12 with the simulations.

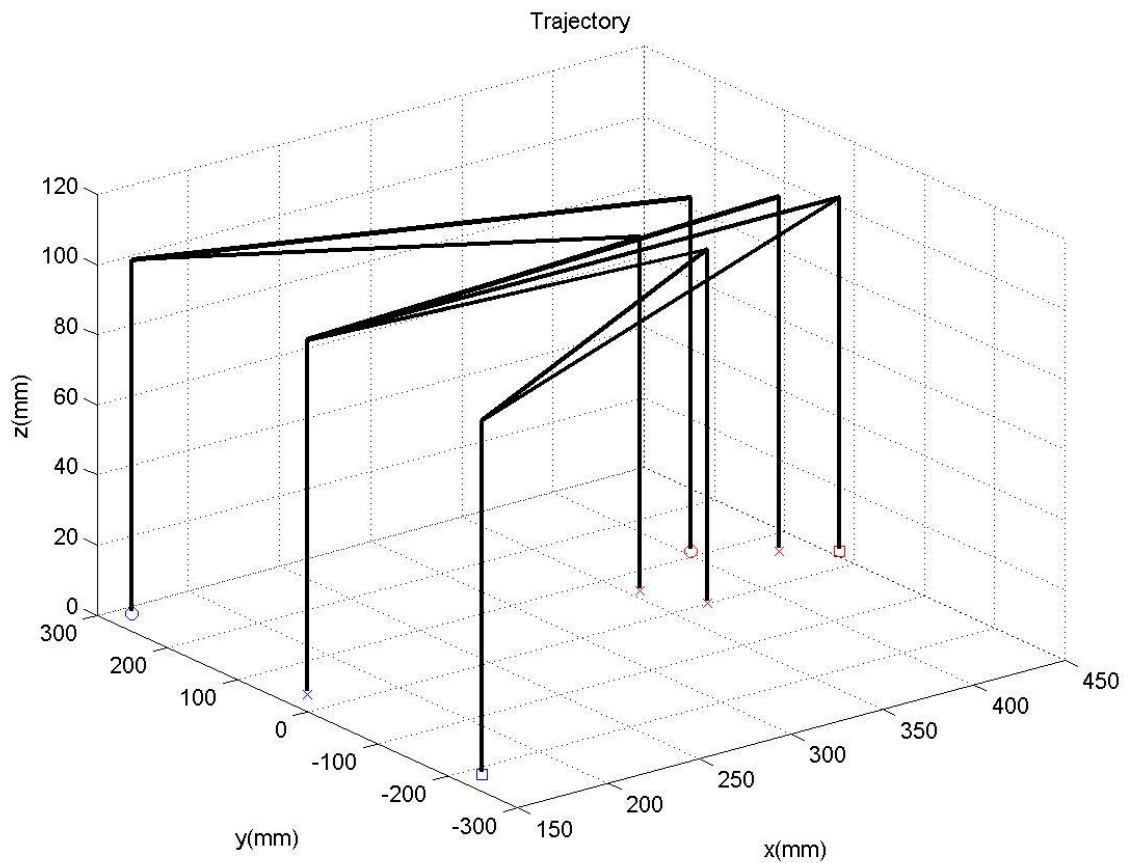


Figure 13: Trajectory test setup

8.4 Converting to joint coordinates

Until now, the trajectory only contains the x-, y- and z-coordinates. However for sending command to the robot, the six joint coordinates have to be known. The function *createConformations* performs this task. The complete code can be found in attachment 17.1.4.

Converting to joint coordinates consists of two steps. First we need to have the orientation of the end effector in the Cartesian space. Then we use the inverse kinematics function, provided by the Mitsubishi Melfa Toolbox, to create the correct joint coordinates.

8.4.1 Orientation

Besides the Cartesian xyz-coordinates, the rotation of the end effector is also required. For this application it is best to keep the end effector vertical at all times. This means

$$\alpha = 0^\circ = 0 \text{ rad}$$

$$\beta = 180^\circ = \pi \text{ rad}$$

$$\gamma = r \text{ rad}$$

The γ rotation is actually random. One option is to keep it 0 all the time, but it is also a good possibility to keep the 4th and the 6th joint 0 rad all the time, since the orientation of the shape while sorting does not matter. This keeps the 4th and the 6th joint unused and is also the option used further in this experiment.

8.4.2 Inverse kinematics

The Mitsubishi Melfa Toolbox provides a function to convert Cartesian coordinates to joint coordinates. This function is used and then the 4th and the 6th joint are overwritten by zero.

8.5 Wrist downward singularity

Overwriting the values of the 4th and 6th joint does come with a problem. The robot is not capable of moving near its base while keeping with the end-effector horizontally. At some point it will not be able to move closer to the base due to the joint limits. This is known as the wrist downward singularity.

The robot has to deal with a wrist singularity when $\sqrt{x^2 + y^2} < 270\text{mm}$.

The value of 270mm is determined with the jog XYZ function in RT Toolbox2. When the arm goes from x = 400 to 150 and y remains 0, it reaches the singularity and stops moving at 260mm. When x goes up from 150 and y = 0, it reaches the singularity at 280mm. Every value between these two values is a good limit.

The solution to the problem is to change the value of the 4th and 6th joint by respectively 180° and -180° when the objects are within the wrist singularity area. The reason that they are both overwritten is that when only the 4th joint value is overwritten, the object will rotate quickly when hanging on the arm. If the 6th joint is also overwritten, the end effector will keep its orientation during the movement.

```
if path(1,n)^2 + path(2,n)^2 < 235^2 %wrist singularity check
    b(6) = 180;
    b(4) = 180;
else
    b(6) = 0;
    b(4) = 0;
end
```

The results are visible in Figure 14.

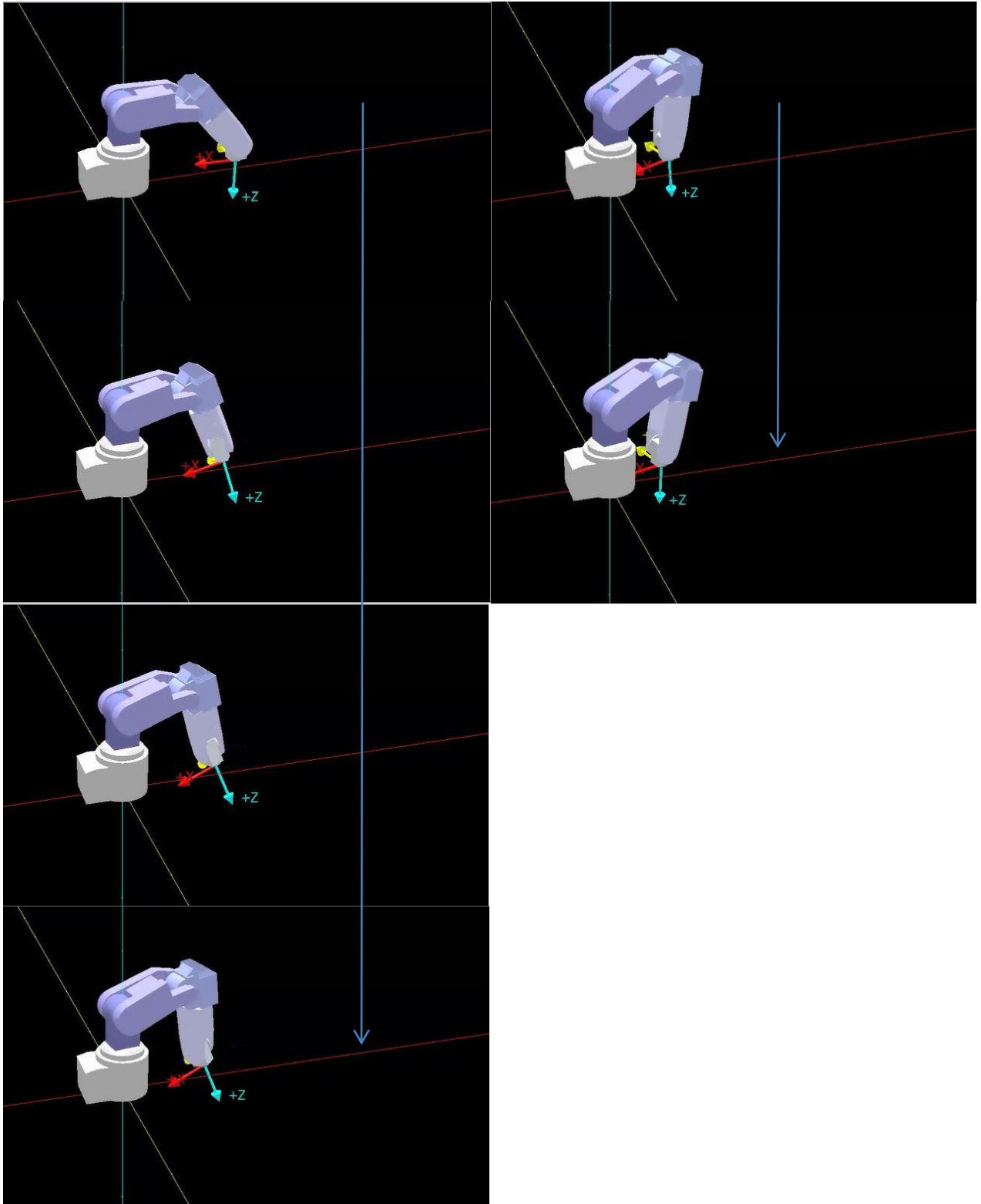


Figure 14: Wrist downward singularity: solution

9 Robot definition

To be able to simulate the robot and to do some calculations such as direct and inverse kinematics with the Melfa Toolbox, the robot has to be defined. The toolbox provides a template file for defining the robot. The only extra data that is needed is the Denavit-Hartenberg parameters and the joint limits.

9.1 Joint limits

The joint limits are given in the datasheet of the robot.

Operating range	J1	deg	480 (-240 to +240)
	J2		240 (-120 to +120)
	J3		160 (0 to +160)
	J4		400 (-200 to +200)
	J5		240 (-120 to +120)
	J6		720 (-360 to +360)

Table 1: Joint limits (adapted from [1])

9.2 Denavit-Hartenberg parameters

Figure 6 shows the joints of the robot in a stretched position. With this position and the shown origins of each different joint, the Denavit-Hartenberg parameters are determined.

The parameter identification procedure requires a specific definition of reference frames attached to the links of the kinematic chain. Fig. 15 illustrates this definition.

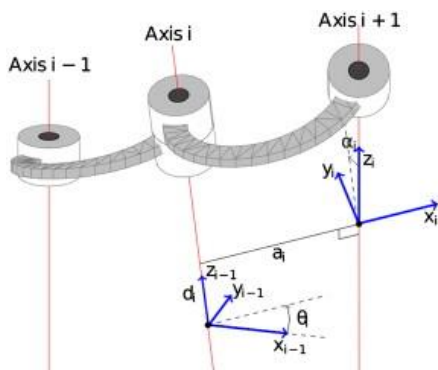


Figure 15: Denavit-Hartenberg parameters (adapted from [8])

The procedure is formed by the following steps:

1.

Identify links and joints: Links are numbered from 0 (base) to n (end-effector). Joints are numbered from 1 to n . In this version of the procedure, joint i connects links $i-1$ and i .

2.

Define the reference frames for the internal links: Locate \mathbf{z}_i axis along the axis of joint $i+1$. The origin of the frame \mathbf{O}_i is positioned along joint $i+1$ axis. If the \mathbf{z} axes are parallel \mathbf{O}_i is arbitrarily chosen. Otherwise, it is located in the intersection between \mathbf{z}_i and the common normal to \mathbf{z}_{i-1} and \mathbf{z}_i . \mathbf{y}_i axis is chosen to compose a right-hand frame.

3.

Define the reference frames for the extremities links: \mathbf{z}_0 is located along the axis of joint 1. \mathbf{x}_0 and \mathbf{y}_0 are arbitrary. \mathbf{x}_n axis is normal to the joint n axis, while \mathbf{y}_n and \mathbf{z}_n are arbitrarily defined.

4.

Identify the D-H parameters for each link: a_i is the distance between \mathbf{z}_{i-1} and \mathbf{z}_i . d_i is the distance between \mathbf{x}_{i-1} and \mathbf{x}_i . α_i is the angle between \mathbf{z}_{i-1} and \mathbf{z}_i measured along \mathbf{x}_i , while θ_i is the angle between \mathbf{x}_{i-1} and \mathbf{x}_i , measured along \mathbf{z}_i .

5.

Determine the homogeneous transformation matrices for each joint.

6.

Determine the overall homogeneous transformation matrix by premultiplication of the individual joint transformation matrices. [9]

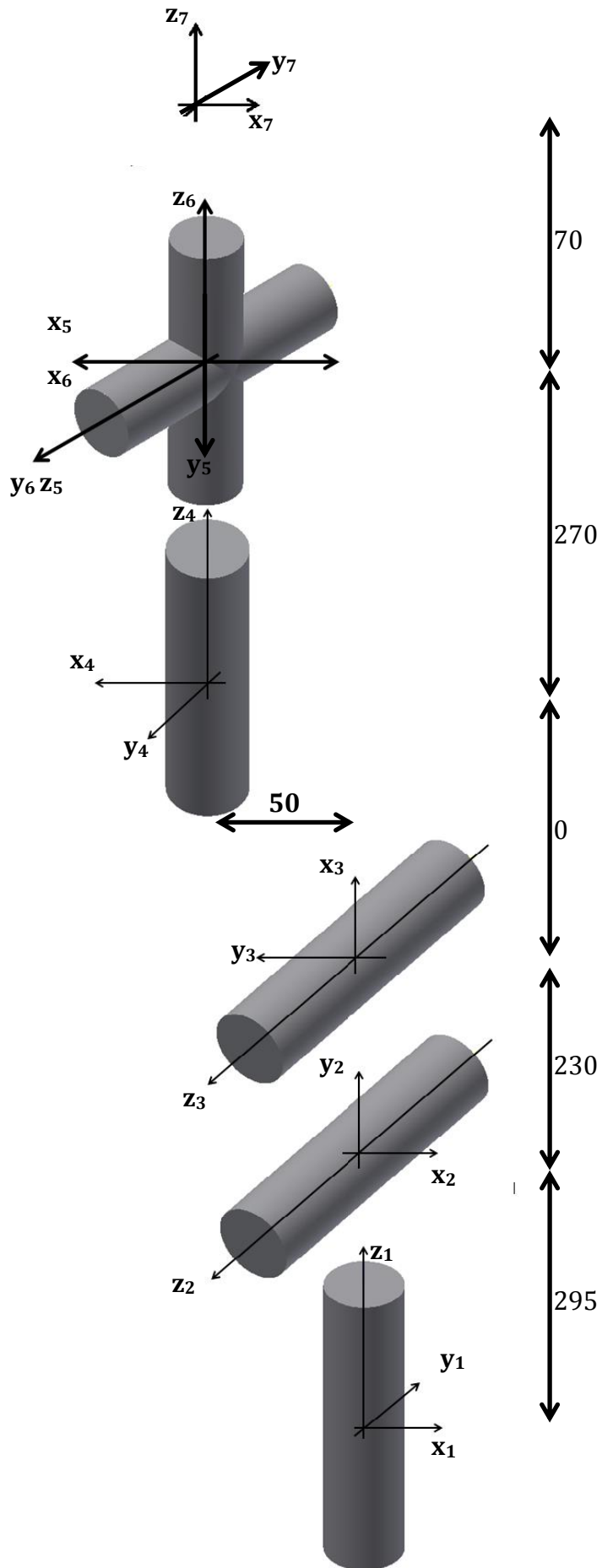


Figure 16: Determination DH parameters

Figure 16 shows this applied to the Melfa RV-2SD. The results are shown in Table 2: Joint limits

i	1	2	3	4	5	6
α_i	$-\pi/2$	0	$-\pi/2$	$\pi/2$	$-\pi/2$	0
a_i	0	230	50	0	0	0
θ_i	0	$-\pi/2$	$-\pi/2$	0	0	π
d_i	295	0	0	270	0	70

Table 2: Joint limits

This data is entered in the template form of the robot definition. The complete robot definition can be found in attachment 17.1.4.

10 Communication

To communicate with the robot, a serial connection is used.

10.1 Serial communication

Serial communication is defined as follows.

In telecommunication and computer science, serial communication is the process of sending data one bit at a time, sequentially, over a communication channel or computer bus. This is in contrast to parallel communication, where several bits are sent as a whole, on a link with several parallel channels. *[10]*

10.2 Bi-directional communication

According to the website Taltech, Bi-directional communication means that the device can send and receive data at the same time. In this case, there are separate lines for transmitting and receiving data. *[11]*

This kind of communication is used in this application, because the robot responds on every command sent to it. For example the PC sends a command to get the current state of the robot. The robot responds with the current state.

10.3 RS-232 protocol

RS232 stands for “Recommended Standard 232”.

“RS-232 is a standard communication protocol for linking computer and its peripheral devices to allow serial data exchange. In simple terms RS232 defines the voltage for the path used for data exchange between the devices. It specifies common voltage and signal level, common pin wire configuration and minimum, amount of control signals.” *[12]*

It is important that both the devices know the amount of data bits. The communication starts when the start bit is sent (low signal). This is visible in Figure 16. Then the actual data bits are sent and finally a stop bit, which is high. As long as the signal stays high the communication is paused.

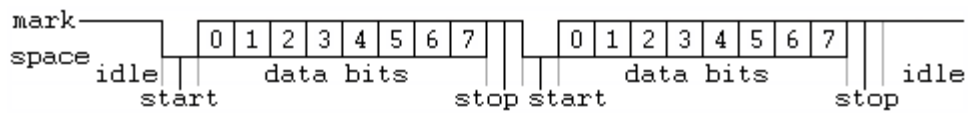


Figure 17: Serial communication (adapted from [13])

10.4 Protocol for communication with robot

10.4.1 Transmit data

Figure 18 shows the flowchart of the communication with the robot.

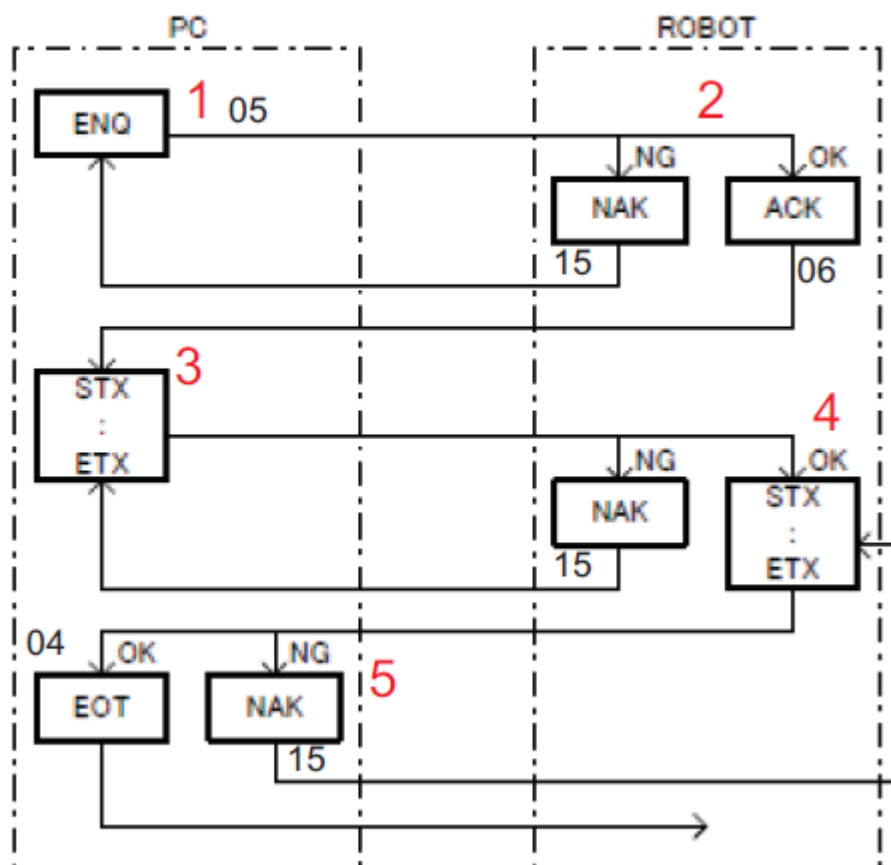


Figure 18: Communication with robot (adapted from [14])

10.4.1.1 Send ENQ

The first step of the communication with the robot is sending the ENQ byte. This byte asks if the device, which in this case is the robot, if it is ready to receive data. It responds with ACK, which means it is ready.

10.4.1.2 Command

Next, the command is sent. The command is built to the R3 protocol. It looks as follows:

<Robot No.>: The robot number to be operated is specified. (0, 1, 2 or 3)

It is possible to omit it. Omitting it is 1.

There are commands that influences all robots if 0 is specified.

< Slot No >: The slot number to be operated is specified. (0, 1 - 33)

...It is possible to omit it. Omitting it is 1...

< Command >< Argument >: It differs in each command;" [15]

Subsequently, the command is preceded by the ASCII character R

It now looks like:

R<Robot No.>;<Slot No>;<Command><Argument>

Since the robot needs to know how long the data is going to be, the command is preceded by two hexadecimal numbers, which determine the amount of bits. For example, if the command is 12 characters long, the command is preceded by 0B.

Next, we add D0 at the beginning of the command.

It is also necessary to allow the robot if the command is sent correctly. To allow this, a NMEA checksum is performed and the result is added to the end of the command. This checksum is done by converting every ASCII character to its hexadecimal form. Next, the XOR logical function is applied to all the hexadecimal numbers. The result is another hexadecimal value. These two characters are added at the end of the command.

To complete the command, it is preceded by STX and ended with ETX. This is to indicate the beginning and ending of the command. An overview is shown in Figure 19.

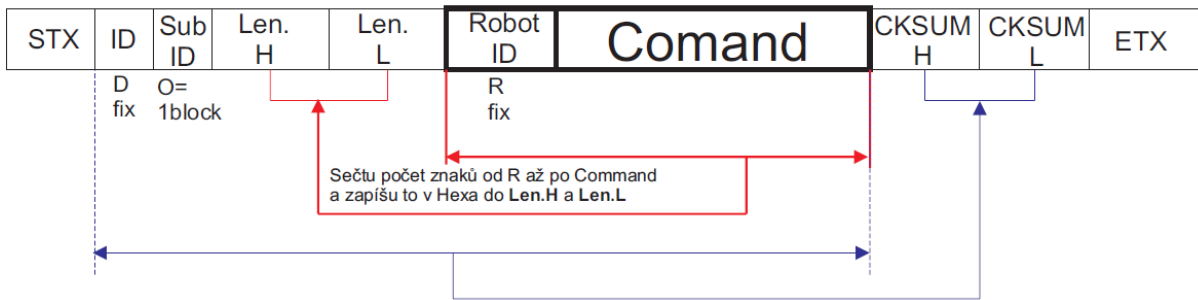


Figure 19: Command according to R3 protocol (adapted from [14])

10.4.1.3 Send EOT

After a command, the EOT, or End Of Transmission command is sent to the robot. This indicates the end of the communication.

10.4.1.4 Example

As an example, the data to send to the COM port is given to turn on the servo motors of the robot.

According to the R3 protocol, the command for opening the communication with the robot is *SRVON*.

The robot and slot number are 1:

1;1SRVON

Preceded by R:

R1;1;SRVON

The amount of characters is 10. In hexadecimal digits this is 0A.

0AR1;1SRVON

Preceded by D0:

D00AR1;1;SRVON

Converting all the ASCII characters to its hexadecimal form:

44 30 30 41 52 31 3b 31 3b 53 52 56 4f 4e

Taking the XOR of all the characters:

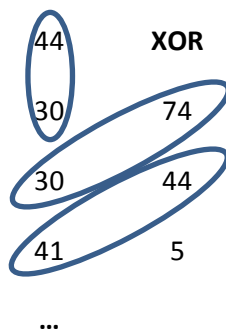


Figure 20: NMEA checksum

Eventually, the result is *01*. The command becomes:

D00AR1;1;SRVON01

Since the command has to start with STX and end with ETX, the command is converted to hexadecimal, since there are no physical ASCII characters for STX and ETX, but there are hexadecimal values.

44 30 30 41 52 31 3b 31 3b 53 52 56 4f 4e 30 31

Now, add the STX and ETX:

02 44 30 30 41 52 31 3b 31 3b 53 52 56 4f 4e 30 31 03

This is the command send to the robot after the command ENQ and before the command EOT is sent. These commands are respectively *05* and *04* in hexadecimal.

Concluded:

05 02 44 30 30 41 52 31 3b 31 3b 53 52 56 4f 4e 30 31 03 04

10.4.1.5 Transform commands in Matlab

Since it is time consuming to do all these steps manually, a function is created to do these steps automatically, given the original command with its argument. First R, the robot no. and the slot no. are added.

```
command = strcat('R1;1;',command); %add R1;1; to command
```

Next the length of the command is added in hexadecimal value and the result is preceded with D0.

```
front = dec2hex(numel(command)); %precede with length
command
if length(front) ==1
    front = strcat('0',front); %hexadecimal part needs 2
digits
end
command = strcat('D0',front,command); %preceed with D0
```

Finally, the NMEA checksum is performed and added.

```
command1 = strcat('$',command,'*'); %Needed for NMEA
checksum
command = strcat(command,nmeachecksum(command1)); %add
checksum at end
```

10.4.2 Receive data

The received data looks as follows:

```
"QoK<Answer>
or
QeR<Error No.>
< Answer >: It differs in each command..
< Error No.>: It replies the error number when the command
cannot be executed." [15]
```


11 Simulation in Mitsubishi Melfa Toolbox

11.1 Workflow communication

Figure 21 shows the workflow of the communication that is used to simulate the movements of the robot. Note that there is no program written into the controller. Only the joint coordinates of the next point of the trajectory is sent together with a command to perform this movement directly.

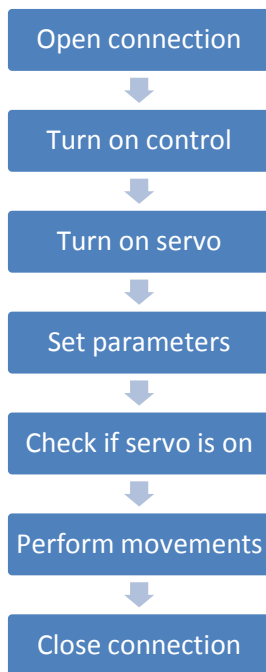


Figure 21: Flowchart simulation Mitsubishi Melfa Toolbox

11.1.1 Open connection

To open the connection, the function *mmOpenVirt* is sent to the virtual robot. This command creates the virtual robot and opens the communication with it. The commands are sent in the same way as with a real robot, but never get sent to the real serial port.

11.1.2 Turn on control

Turning on the control of the robot enables the computer to set parameters and to make the robot move. To do this, the command 'CNTLON' must be send through the serial connection. The function *mmCntlOn* from the Melfa Toolbox performs this task. It sends 'EXECNTLON' to the robot and receives an answer. The 'EXEC' is necessary, because the command has to be executed directly and is not saved into a program which is executed later. It also checks if there is an error with sending and receiving the command.

11.1.3 Turn on servo

To enable the robot to move, the servos must be turned on. This is done with the command 'SRVON'. The function *mmServoOn* performs this task. Besides the command, the function is identical to the *mmCntlOn*.

11.1.4 Set speed

The desired speed is set while the robot is turning the servos on, since this takes a while. The jog speed can be set with the command OVRD n, where n means the relative speed according to its maximum value. The toolbox provides a function called *mmJogSpeed* to do this.

What the functions also does, is checking if the entered speeds or accelerations are valid. If the entered value for the speed (or acceleration) is invalid, the command does not get executed and the function returns an error.

11.1.5 Check if the servo is on

The function *mmWaitForServoOn* validates that the servos are on. The command DSTATE is sent to the robot. It replies with the stop state of the robot.

“[Function]

The stop state is read.

[Format]

DSTATE

[Answer]

QoK<Run sts.><Stop sts.><Error no.>;<Step no.>;<Mech no.>

<Run sts.> Run status by 2 HEX number fixation

00000000B 0 / 1

_____1 Cycle / Repeat

_____1_ Cycle stop ON / OFF

_____1_ MLOCK OFF / ON

_____1__ Auto / Teach

_____1___ Running of Teach mode

_____1___ Servo OFF / ON

_____1_____ STOP / RUN

1_____ Operation disable / enable

<Stop sts.> Stop status by 2 HEX number fixation

00000000B

_____1 EMG STOP

_____1_ STOP

_____1_ WAIT

_____1__ STOP signal ON / OFF

_____1___ Program select enable

_____1_____ (reserve)

_____1_____ Pseudo input

<Error no.> Error number. (0:No error)

<Step no.> Execution step number “ **[15]**”

By checking the sixth bit of the run state byte, the function confirms that the servo is on or not. However, after three seconds, the function will return an error, because it takes too long to get the response.

11.1.6 Perform movements

Now the robot is ready to perform its movements. The command *mmMovSafe* has three tasks.

First it checks if the movement that should be performed is valid. If not, it returns an error.

Next, the movement string is created. This string is the command that is sent to the robot through the communication line. The command exists of two parts. The first part is to send the coordinates to the robot together with the coordinate system. There are three kinds of coordinate systems.

- 'J' for joint coordinates;
- 'P' for XYZ coordinates;
- 'X' for XYZ coordinates and the rotation in joint coordinates.

The second part of the command is 'MOV', followed by 'J', 'P' or 'X', depending on the type of coordinates.

These commands are preceded by EXEC, for the same reason as before.

Finally the function calls the current state of the robot. It checks if the robot returns an error and displays this error.

With each performed movement there should also be an alert when the movement is finished, so the robot does not perform the next command before the robot has reached the desired position. The function *mmWaitForStop* performs this task. It sends the command to get the state of the robot and keeps doing this every n seconds (n is an optional parameter, standard value is 0,5 s) until the bit for STOP/RUN is 0, until an error occurs or until the adjustable maximum time is reached.

11.1.7 Close connection

With the function *mmClose* the connection gets closed. It sends the command CLOSE through the com port.

11.1.8 Applied to test setup

The code used for the function to communicate between the PC and the robot is as follows:

```
function executeProgram_RV2SD(conformations, speed, wait)
```

First the connection with the robot RV2-SD gets opened. In case a virtual simulation is started *mmOpenVirt* is used. Otherwise the normal command *mmOpen* is used.

```
%open connection  
robot1 = mmOpenVirt('RV2SD');
```

Next, the command is sent to give Matlab permission to send commands to the robot.

```
%Turn on control  
mmCntlOn(robot1);
```

Subsequently, the servo motors get turned on.

```
%turn on servomotor  
mmSrvOn(robot1);
```

Next, a for-loop iterates through all conformations, i.e. move to the consecutive points of the trajectory. With each move, the speed is entered and next the move gets executed. The parameter 'conformations' is a cell matrix. Each element contains a 6x1 matrix with the next point of the trajectory, expressed in joint coordinates.

The 'r' returned from the *mmWaitForServoOn* function is 1 when an error is returned and zero if not. When an error occurs, the connection gets shut down with the *mmClose* function.

'r1' and 'r2' have a similar meaning. The connection gets closed when one of the two is high.

```

[~ amount] = size(conformations);

%iterate through each conformation
for j=1:amount
    %move in jog coordinates
    mmSetJogSpeed(robot1, speed(j));

    [~, r1] = mmMovSafe(robot1, 'J', conformations{j} );

    %wait until finished
    [r2] = mmWaitForStop(robot1);

    %quit if error
    if ( r1 || r2 )
        mmClose(robot1);
        error('Robot is in error state. ');
    end
end
end

```

When all the moves are performed, the communication line gets closed.

```

%all done, close communication
mmClose(robot1);

```

11.2 Results

In the simulation it is visible that the trajectory is followed correctly. The test setup discussed earlier is executed and in Table 3: Simulations test setup in chapter 13 the results of the simulation are visible.

12 Simulation with RT Toolbox2

12.1 Communication workflow

RT Toolbox2 does not allow the option to communicate with Matlab or with external cameras. Therefore, another method should be used to enable the virtual robot to perform the trajectory. Instead of sending each movement as a different command, the complete program will be written to the controller of the virtual robot and afterwards the program will be ran in the RT Toolbox2 software.

The program that will be written to the controller will be created by Matlab, so that all the previous steps concerning the image acquisition, image processing and trajectory planning will remain the same. Figure 22 shows an overview of the method.

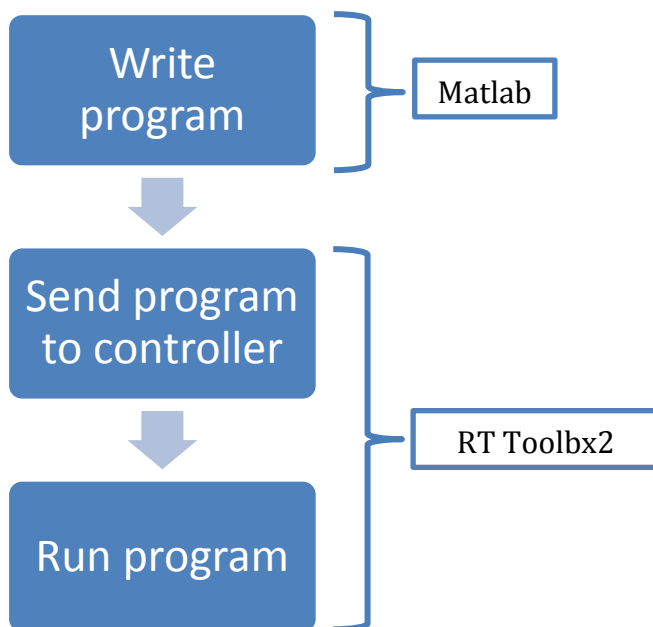


Figure 22: Simulation with RT Toolbox2

12.1.1 Write program

For the (virtual) robot to be able to perform the complete trajectory by running a program, the program must contain:

1. Defining all the points
2. The order of the points

At the beginning of the program the speed will be set. The program also has to end with the command end.

All the commands are preceded with its line number.

Concluded, the program has to look like this:

```
1 J1=(j1,j2,j3,j4,j5,j6,j7,j8)
2 J2=(j1,j2,j3,j4,j5,j6,j7,j8)
...
n+1 MOV J1
n+2 MOV J2
...
2*n+1 END
```

With j_1, j_2, \dots the joint coordinates of the corresponding point of the trajectory.

The file for a program is a *.prg* file. However, its content is just text. By letting Matlab write strings to a *.prg* file, it is possible to write a program in Matlab that can be opened and sent to the robot with RT Toolbox2.

The first step is to make an array where all the lines can be saved in string format

```
stringArray = cell(amount*2+3,1);
```

Next, the first line is entered to set the speed

```
stringArray{1} = sprintf('1 Ovrd 25');
```

A for-loop is started which goes through all the points of the trajectory. For each element the lines for describing and defining the corresponding point of the trajectory are written in the array.


```

for n = 1:amount
    b = conformations{n};
    tmp = ...

sprintf('J%d=(%0.3f,%0.3f,%0.3f,%0.3f,%0.3f,%0.3f,0.000,0.00
0)'...
    ,n, b(1), b(2), b(3), b(4), b(5), b(6));
stringArray{n + 1} = tmp;
tmp = sprintf('%d Mov J%d', n, n);
stringArray{n+amount + 1} = tmp;
end

```

Finally the end command is entered in the array.

```
stringArray{amount*2 + 2} = sprintf('%d end', amount*2+3);
```

Now the complete program is written to the array. This array should now be converted to the *.prg* file. For doing this, the file gets opened first.

```
fid = fopen('...\TSTSETUP.prg', 'wt');
```

Now we write each line.

```

for (i=1:amount*2 + 3)
    fprintf(fid, '%s\n', my_cell{i});
end

```

Finally the file gets closed again.

```
fclose(fid);
```

The complete program can be found in attachment 17.1.7.

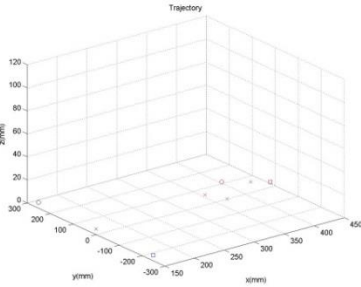
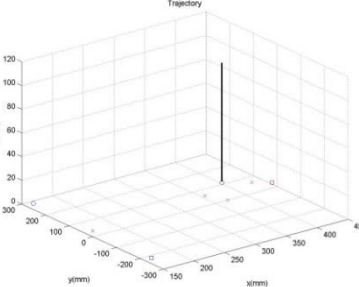
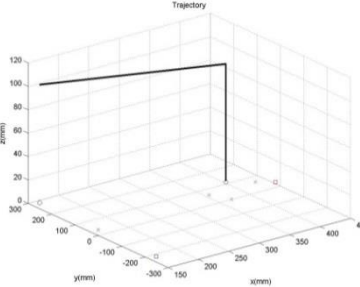
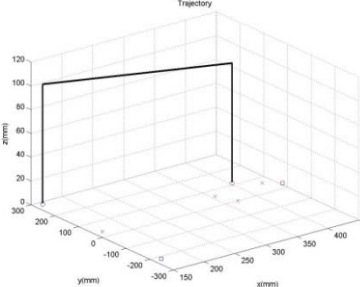
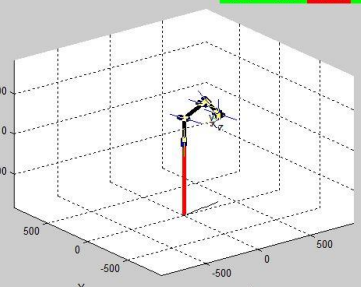
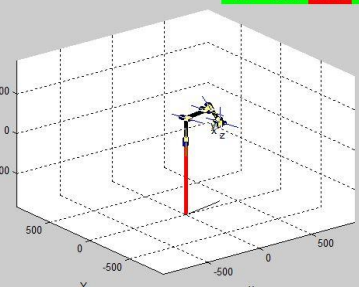
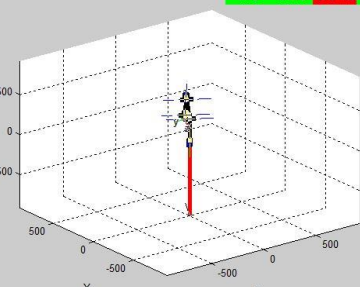
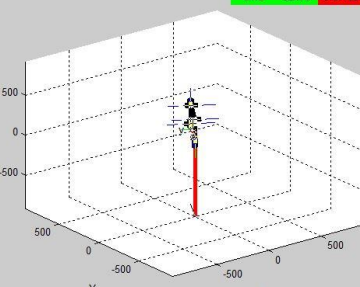
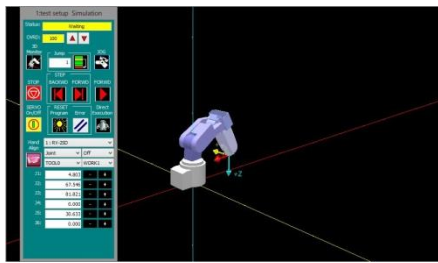
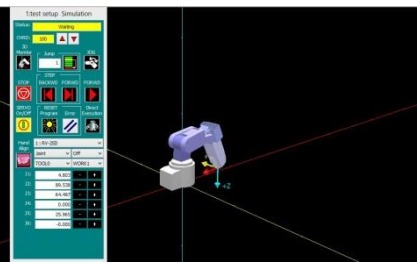
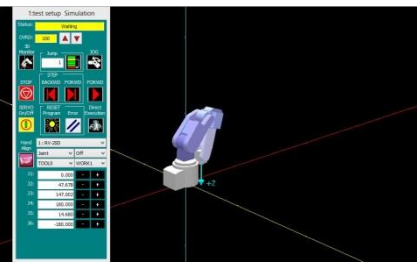
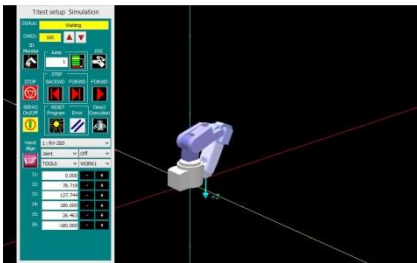
12.1.2 Send program to controller

After the program is written by Matlab, it is possible to open this program in the RT Toolbox2. Next this whole program is sent to the virtual robot of the RV-2SD, which matches the exact same specifications of the real robot. If the robot is now possible to perform the trajectory, it is certain that the actual robot will also be able to perform it, and that no joint limits will be reached.

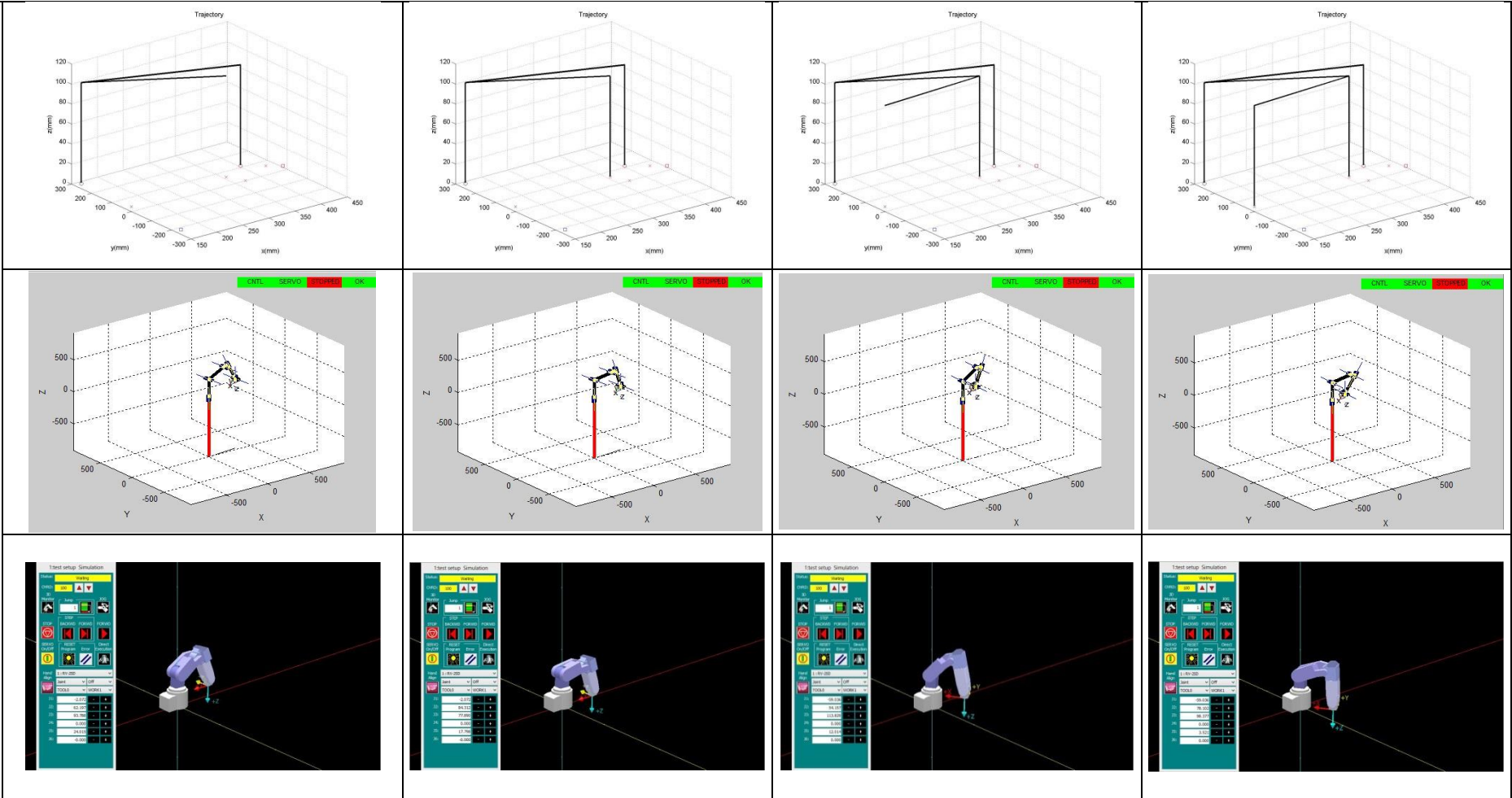
12.2 Results

The program is started and in Table 3: Simulations test setup in chapter 13 the results are shown together with the trajectory and the results of the Mitsubishi Melfa Toolbox Simulation.

13 Results simulations

Obj. Nr.	To object	Down & up	To SP	Down & up
1				
				
				

2



3				

Table 3: Simulations test setup

14 Robot

According to the manual of the Mitsubishi Melfa Toolbox, the communication can be performed the same way as the simulation done in chapter 11. The only difference would be to change *mmOpenVirt* to *mmOpen* in the program found in attachment 17.1.6. However, the toolbox only supports the robots Mitsubishi Melfa RV6S and RV6SDL. The communication with the Melfa RV2SD works differently than described in chapter 11 with these robots. Also, in the R3 protocols command list, there is nothing listed to perform direct movements. As a result, a new way to perform the movements of the trajectory had to be found.

The solution used in this project is similar to the method used to perform the simulation with the RT Toolbox2.: First, write all the lines of the program to the controller's memory. Next, the command to run the robot will be sent.

14.1 Add serial port to Matlab

Before the serial port can be used by Matlab for sending commands, it needs to be opened and set up the same way as the robot. Therefore the following properties of the serial port must be set to the settings shown in Table 4: Settings serial communication

Baud rate	9600
Terminator	Carriage return
Databits	8
Stopbits	2
Parity	even

Table 4: Settings serial communication

These are the settings that are needed to communicate with this specific robot.

After this step it is possible for the robot to understand the commands sent through the serial port.

Next, the COM1 port should be opened, so Matlab gets permission to the serial port from the PC.

```
s=serial('COM1')
set(s, 'BaudRate', 9600);
set(s, 'Parity', 'even');
set(s, 'DataBits', 8);
set(s, 'StopBits', 2);
set(s, 'Terminator', 'CR');

fopen(s);
```

14.2 Workflow communication robot

The workflow to communicate with the robot, according to the used method is described in Table 5: Serial commands for communication

	Action	Command	Explanation
1.	Open connection	OPEN=TOOLBOX	Establishing connection between PC and robot
2.	Turn on control	CNTLON	Give the PC permission to write in controller
3.	Load program	LOAD=TRAN	Load a program on the controller
4.	Write program	EDATA<line content>	Write a line into the program
5.	Save program	SAVE	Save and close the program
6.	Turn servo on	SRVON	Turn the servo of the robot on
7.	Run the program	RUNTRAN;1	Run the program. The 1 is to tell the robot to only perform the program once

Table 5: Serial commands for communication

In step 4, the line content is the same as in chapter 12 for the simulation with RT Toolbox2.

Each command is sent to the serial port as follows:

```
data = char2hex(command); %convert to hex  
  
data = hex2dec(data); %convert to decimal  
fwrite(s,uint8(data)); %write to serial port
```

The function *char2hex* not only converts all the characters to their hexadecimal equivalent, but it also adds the ENQ and STX command at the beginning and ETX and EOT command at the end.













The command should also be converted to decimal values, because Matlab is not able to send pure hexadecimal commands.

At the end of the program the serial port gets closed again as follows:

```
fclose(s);
```


14.3 Results

The results of sending the trajectory to the robot are shown in Table 6.

Obj. Nr.	To object	Down & up	To SP	Down & up
1				
2				
3				

4				
Home				

Table 6: Results robot

15 Conclusions

With the described method it is possible to do a vision-based control of the robot with Matlab. The robot recognizes the shapes picks them up and sort them out. The robot is able to grasp near the base thanks to the solution for the wrist singularity, without letting the end-effector, holding the objects, making sudden and useless rotating movements which can cause the object to fall.

15.1 Communication with robot via serial connection

With the established communication between Matlab and the robot, described in chapter 14 it is possible to do vision-based control of the robot with Matlab. All the commands have to be according to the R3-protocol for the robot to be able to understand them. The commands needs some additions for letting the robot know the length of the command and for the robot to be able to do an error check thanks to the checksum.

Every command has to start with a STX and end with an ETX command. Before the communication can start, an ENQ command has to be sent and after the transmission the EOT command must be sent.

When all these requirements are fulfilled, Matlab can control the robot.

15.2 Shape recognition

The shape recognition was possible thanks to the property *incompactness*. With this method circles, squares and rectangles can be distinguished. However, if more kind of shapes has to be sorted, this method needs some additional steps to be able to work.

15.3 Wrist singularity

When the robot needs to go near its base, the robot has to pass a wrist singularity border. By setting joint coordinates J4 and J6 to zero when the end effector is further away than this border and by setting them to respectively 180 and -180 degrees, when closer than this border, the robot can work without any problems of the wrist singularity.

15.4 Further works

A problem with the described control of the robot with Matlab is that it writes the complete program at the start of the cycle. This results in a long time to send the trajectory to the robot.

A faster approach would be to send only the next move to the controller of the robot. By doing so, the next movement can be calculated while the robot performs its current move. Another advantage of this method would be that moving objects can also be picked up.

16 References

- [1] Mitsubishi. (2010, June) Melfa RV-2SDB/RV-2SQB Series. Datasheet.
- [2] Mathworks. mathworks.com. [Online].
<http://www.mathworks.com/products/matlab/>
- [3] Peter corke. PeterCorke.com. [Online]. http://petercorke.com/Robotics_Toolbox.html
- [4] Martin Meloun. Mitsubishi Melfa Robot Control Toolbox for Matlab - Instructions for use. [Online].
https://cw.felk.cvut.cz/wiki/help/common/robot_mitsubishimelfa_toolbox
- [5] Mathworks. Mathworks. [Online].
<http://www.mathworks.com/help/images/ref/imopen.html>
- [6] Mathworks. Mathworks. [Online].
<http://www.mathworks.com/help/images/ref/imclose.html>
- [7] The MathWorks, Inc. <http://www.mathworks.com>. [Online].
<http://www.mathworks.com/help/images/ref/regionprops.html#bqkf8id>
- [8] Fauske KM., "An introduction to sketch 3d for pgf and tikz users," 2009.
- [9] C.P. Tonetta, A. Dias C.R. Rocha, "A comparison between the Denavit–Hartenberg and the screw-based methods used in kinematic modeling of robot manipulators," *Robotics and Computer-Integrated Manufacturing*, pp. 723–728, 2011.
- [10] Wikipedia. (2014, april) Wikipedia. [Online].
http://en.wikipedia.org/wiki/Serial_communication
- [11] Taltech. Taltech. [Online].
http://www.taltech.com/datacollection/articles/serial_intro
- [12] Bijal Parikh. Engineers garage. [Online].
<http://www.engineersgarage.com/articles/what-is-rs232>
- [13] Wikipedia. Wikipedia. [Online].
http://en.wikipedia.org/wiki/Asynchronous_serial_communication
- [14] AutoCont Controlsystems. Komunikace RS232 - ROBOT RV-2SD. Manual.
- [15] "Connection with personal computer[RS-232C/Ethernet]," 2009.

17 Attachments

17.1 Matlab code

17.1.1 Main program code

```
clear all
close all
% % read in image
vid = videoinput('winvideo', 1, 'YUY2_640x480'); %connect the camera
set(vid, 'ReturnedColorSpace', 'rgb');
img = getsnapshot(vid); %saves the picture
% img = imread('C:\Users\wim\Pictures\Camera-album\afbeelding029.jpg');
img = imrotate(img,180);

c = imcrop(img,[150.5 20.5 315 385]);
figure,imshow(c);

bw = im2bw(c, graythresh(rgb2gray(c)));
bw = ~bw;
bw = medfilt2(bw,[5,5]);
SE = ones(3); %create a structure element of 3x3 filled with ones
bw = imopen(bw,SE);
bw = imclose(bw,SE);
% imshow(bw);
[B,L] = bwboundaries(bw,'noholes');

%Set field width and height in mm.
width = 295;
height = 350;

%Function to find all the objects and their properties.
[props, propsOld] = calculateObjectProperties(L, bw, width, height);

%Function to mark the borders in red in the black and white image.
plotBorders(B,bw);
markImage(c, B, propsOld);

%Set the height off the working area and thickness.
thickness = 1;
heightTable = 5;

%Set the height above the objects.
offset = 100;

%set the coordinates of the sorting points
T1 = [150, 250]; %Circles
T2 = [150, -250]; %Squares
T3 = [150, 0]; %Rectangles

%Calculate the trajectory, the speed for each movement and
%whether it should be a linear movement or joint movement.
path = ...
    calculateTrajectory(props, T1, T2, T3, offset, heightTable, thickness);
```

```
%Function to plot the Trajectory.
% plotTrajectory(T1, T2, T3, path,props, heightTable);

%load the robot
robot = mmRobot_RV2SD;

%make the matrix with the path in the correct matrix format
conformations = createConformations(path);

length = size(conformations);
length = length(2);

%Create program for the simulation with RT Toolbox2
createRtProgram(conformations, length);

%start the Mitsubishi Melfa Toolbox simulation
simulation(conformations);

%send all the commands to the robot
s = Serial_TRAN(length(2), conformations);
```


17.1.2 Determine properties of objects code

```
function [props, propsOld] = calculateObjectProperties(L, bw, width, height)

    %get image resolution
    [y, x] = size(bw);
    properties = regionprops(L, 'all'); %Call all the properties of the
objects

    for k = 1:length(properties)
        compactness(k) = properties(k).Perimeter^2 ...
            / properties(k).Area; %calculate compactness
        if compactness(k) < 14 %check if circle
            props(k,1) = 1; %1 means circle
        elseif compactness(k) > 14 & ... %check if square
            compactness(k) < 15.5 %2 means square
            props(k,1) = 2; %else it is a rectangle
        else %3 means rectangle
            props(k,1)=3;
        end

        props(k,2) = properties(k).Centroid(1); %save x-position
        props(k,3) = properties(k).Centroid(2); %save y-position
    end

    propsOld = props;

    %convert pixels to mm
    props(:,2) = width / x * props(:,2);
    props(:,3) = height / y * props(:,3);

    %translating to the correct origin
    props(:,2) = props(:,2) - width/2;
    props(:,3) = height + 155 - props(:,3);

    %rotating and mirror axis
    props(:, [2,3]) = props(:, [3,2]);
    props(:,3) = -props(:,3);
    props;
end
```


17.1.3 Trajectory planning code

```
function path =...
    calculateTrajectory(props, ...
        T1, T2, T3, offset, heightField, thickness)
% This function creates a 3x(n*6) matrix defining the trajectory
% with n the amount of objects.
% Arguments:
%   props:      n x 3 matrix with in the first
%               column the shape, second the x position
%               and third y position in the robots origin
%   T1:         1x2 matrix with x and y position
%               for circles
%   T2:         1x2 matrix with x and y position
%               for squares
%   T3:         1x2 matrix with x and y position
%               for rectangles
%   offset:     height (z value) to move above objects.
%   heightField: height of the surface with the shapes on top
%   thickness:  thickness of object in mm.
%
%
% Return values:
%   path:       The (n*3) x 3 matrix that contains all the
%               xyz-coordinates of the points of the trajectory.

[s ~] = size(props);

%amount of each kind of objects that are picked up
squares = 0;
rectangles = 0;
circles = 0;

%working field is lower than top of objects.
heightField = heightField + thickness;

for i = 1:s
    path(1 + 6*(i-1), :) = [props(i,2) props(i,3) ...
        heightField + offset]; %to object

    path(2 + 6*(i-1), :) = [props(i,2) props(i,3) ...
        heightField]; %grab

    path(3 + 6*(i-1), :) = [props(i,2) props(i,3) ...
        heightField + offset]; %up

    %to designated place
    if props(i,1) == 1
        path(4 + 6*(i-1), :) = [T1(1) T1(2) ...
            heightField + offset];

    elseif props(i,1) == 2
        path(4 + 6*(i-1), :) = [T2(1) T2(2) ...
            heightField + offset];

    elseif props(i,1) == 3
        path(4 + 6*(i-1), :) = [T3(1) T3(2) ...
            heightField + offset];
    end
end
```

```

%down
if props(i,1) == 1
    path(5 + 6*(i-1), :) = [T1(1) T1(2)...
        heightField + thickness*circles];
    circles = circles + 1;

elseif props(i,1) == 2
    path(5 + 6*(i-1), :) = [T2(1) T2(2)...
        heightField + thickness*squares];
    squares = squares + 1;
elseif props(i,1) == 3
    path(5 + 6*(i-1), :) = [T3(1) T3(2)...
        heightField + thickness*rectangles];
    rectangles = rectangles + 1;
end

%back up

if props(i,1) == 1
    path(6 + 6*(i-1), :) = [T1(1) T1(2)...
        heightField + offset];
elseif props(i,1) == 2
    path(6 + 6*(i-1), :) = [T2(1) T2(2)...
        heightField + offset];
elseif props(i,1) == 3
    path(6 + 6*(i-1), :) = [T3(1) T3(2)...
        heightField + offset];
end
end
path(6*s + 1, :) = [T3(1) T3(2) heightField + offset];
end

```

17.1.4 Robot definition code

```
classdef mmRobot_RV2SD < mmRobotDef
%MMROBOT_RV-2SDB Robot specification: Mitsubishi RV-2SDB
%
% MITSUBISHI MELFA TOOLBOX v1.4
% (C) Martin Meloun
%
% Mitsubishi Melfa Toolbox is free software: you can redistribute it and/or
modify
% it under the terms of the GNU Lesser General Public License as published by
% the Free Software Foundation, either version 3 of the License, or
% (at your option) any later version.
%
% Mitsubishi Melfa Toolbox is distributed in the hope that it will be useful,
% but WITHOUT ANY WARRANTY; without even the implied warranty of
% MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
% GNU Lesser General Public License for more details.
%
% You should have received a copy of the GNU Lesser General Public License
% along with Mitsubishi Melfa Toolbox. If not, see
<http://www.gnu.org/licenses/>.
%
% Author: Revision: Date: Reason:
% Martin Meloun 3 4.8.2011 16:08:25 Corrected
joint 6 theta parameter
% Martin Meloun 2 12.10.2010 16:10:32 Corrected
DH notation
% Martin Meloun 1 25.5.2010 15:28:05 Initial
revision
```

methods

```
function robot = mmRobot_RV2SD()

% Name
% -----
robot.name = 'RV-2SD';
robot.description = 'Mitsubishi Melfa RV-2SD';
robot.series = 'Melfa V';
robot.pincode = 0;

% Properties
% -----

robot.DOF = 6;
robot.joints = 'RRRRRR';

robot.denavitHartenberg = [ -pi/2, 0, -pi/2, pi/2, -pi/2, 0; %alpha
                             0, 230, 50, 0, 0, 0; %a
                             0, -pi/2, -pi/2, 0, 0, pi; %theta
                             295, 0, 0, 270, 0, 70];%d

robot.denavitHartenbergParameters = [ 0, 0, 0, 0, 0, 0; %alpha
                                       0, 0, 0, 0, 0, 0; %a
                                       1, 1, 1, 1, 1, 1; %theta
                                       0, 0, 0, 0, 0, 0]; %d

robot.base = eye(4);
robot.tool = eye(4);
```

```

robot.jointLimits = [-240, -120, 0, -200, -120, -360; %minimums
                    240, 120, 160, 200, 120, 360]; %maximums
robot.A76 = mtzTranslate([0; 0; robot.denavitHartenberg(4,6)]);
robot.ikt = @mmIkt_RV6S_Series;
robot.testKinematics = @mmTestKinematics_RV6S_Series;
robot.getflags = @mmGetFlags_RV6S_Series;

% PC Control
% -----

robot.com = [];
robot.comtype = 'Serial';
robot.portname = 'COM9';
robot.controller = 'CR1DA-771';
robot.robotNo = 1;
robot.slotNo = 1;
robot.comprops = mmComprops_Serial(9600, 8, 'even', 2, 'on', 'on', 'CR');
robot.topVariableSign = '>';

% Debugging
robot.verbose = 0;
robot.thrError = 0;

% HardHome
% -----

robot.hhflag = 2;

% Userdata
% -----

robot.userdata = [];

% Security (none - base definition)
% -----

robot.safetyData = [];
robot.secureMovement = 0;
robot.secureSpeed = 0;
robot.secureAccel = 0;

end

end

end

```

17.1.5 Code to create conformations

```
function conformations = createConformations(path)
    path = path'; %
    robot = mmRobot_RV2SD;
    amount = size(path, 2);
    conformations = cell(1, amount);

    for n=1:amount
        %take the inverse kinematics of all the positions of trajectory
        conformations{n} = mmIkt(robot, [path(:,n); 0; 180; 0]);
        b = conformations{n};
        if path(1,n)^2 + path(2,n)^2 < 235^2 %wrist singularity check
            b(6) = -180;
            b(4) = 180;
        else
            b(6) = 0;
            b(4) = 0;
        end
        s = size(b);
        if s(2) > 1 %more solutions, take the first one
            b = b(:,1);
            %Save the joint positions in cell with strings
        end
        conformations{n} = b;
    end
end
```


17.1.6 Mitsubishi Melfa Toolbox Simulation code

```
function simulation(conformations)

%open connection
robot1 = mmOpenVirt('RV2SD');

%Turn on control
r = mmCntlOn(robot1);

%turn on servomotor
r = mmSrvOn(robot1);

%wait until the servomotor is on
r = mmWaitForServoOn(robot1);

amount = size(conformations);
%check for error
if (r)
mmClose(robot1);
error('Failed to turn on servo motors.');
```

```
end

%iterate through each conformation
for j=1:amount(2)
    %move in jog coordinates
    mmSetJogSpeed(robot1, 25);

    [~, r1] = mmMovSafe(robot1, 'J', conformations{j});

    %wait until finished
    [r2] = mmWaitForStop(robot1);

    %quit if error
    if ( r1 || r2 )
        mmClose(robot1);
        error('Robot is in error state.');
```

```
end
end

%all done, close communication
mmClose(robot1);
end
```


17.1.7 Code to create program for RT Toolbox2 Simulation

```
function createRtProgram(conformations, amount)

stringArray = cell(amount*2+3,1);
stringArray{1} = sprintf('1 Ovrdr 25');

for n = 1:amount
    b = conformations{n};
    tmp = ...
    sprintf('J%d=(%0.3f,%0.3f,%0.3f,%0.3f,%0.3f,%0.3f,0.000,0.000)'...
        ,n, b(1), b(2), b(3), b(4), b(5), b(6));
    stringArray{n + 1} = tmp;
    tmp = sprintf('%d Mov J%d', n, n);
    stringArray{n+amount+1} = tmp;
end

stringArray{amount*2 + 2} = sprintf('%d end',amount*2+3);

%open RT2 program
fid = fopen('C:\Users\wim\Documents\School\Masterproef\fotos
testopstelling\RTT2\Test setup\test setup\Program\TSTSETUP.prg','wt');

%write code in it
for (i=1:amount*2 + 3)
    fprintf(fid, '%s\n',stringArray{i});
end

%close it again
fclose(fid);

end
```


17.1.8 Communication with robot

```
function s = Serial_TRAN(length, conformations)
% Script runs TRAN.PRG program in the control unit
% Uses Command Creator functions

s=serial('COM1')
set(s, 'BaudRate', 9600);
set(s, 'Parity', 'even');
set(s, 'DataBits', 8);
set(s, 'StopBits', 2);
set(s, 'Terminator', 'CR');

fopen(s);

sendCommand(s, 'OPEN=USERTOOL');

sendCommand(s, 'CNTLON');

sendCommand(s, 'LOAD=TRAN');

command = 'EDATA1 OVRD 20';
sendCommand(s,command);

for i = 2:length+1 %length+2:length*2+1
    b = conformations{i-1};
    command = sprintf('EDATA%d
J%d=(%0.3f,%0.3f,%0.3f,%0.3f,%0.3f,%0.3f,0.000,0.000)', i,i-1, b(1), b(2),
b(3), b(4), b(5), b(6));
    sendCommand(s,command);
end

for i = length+2:length*2+1
    command = sprintf('EDATA%d MOV J%d',i,i-length-1);
    sendCommand(s,command);
end

command = sprintf('EDATA%d END',length*2 + 2);
sendCommand(s,command);

sendCommand(s, 'SAVE');

sendCommand(s, 'SRVON');

sendCommand(s, 'RUNTRAN;1');

fclose(s);
```


17.1.9 Function to send commands

```
function sendCommand(s,command)

command = strcat('R1;1;',command); %add R1;1; to command

front = dec2hex(numel(command)); %preced with length command
if length(front) ==1
    front = strcat('0',front); %hexadecimal part needs 2 digits
end

command = strcat('D0',front,command); %preced with D0

command1 = strcat('$',command,'*'); %Needed for NMEA checksum
command = strcat(command,nmeachecksum(command1)); %add checksum at end

data = char2hex(command); %convert to hex

data = hex2dec(data); %convert to decimal
fwrite(s,uint8(data)); %write to serial port

end
```


17.1.10 Function to convert ASCII characters to hex

```
function hex = char2hex(string)
characters = {' '; 'A'; 'B'; 'C'; 'D'; 'E'; 'F'; 'G'; 'H'; 'I'; 'J'; 'K'; 'L'; 'M'; ...
             'N'; 'O'; 'P'; 'Q'; 'R'; 'S'; 'T'; 'U'; 'V'; 'W'; 'X'; 'Y'; 'Z'; '0'; '1'; '2'; '3'; ...
             '4'; '5'; '6'; '7'; '8'; '9'; '='; '('; ')' ; '.'; '-' ; ',' ; ' ' };

hexvalues = {'3B'; '41'; '42'; '43'; '44'; '45'; '46'; '47'; '48'; '49'; '4A'; ...
            '4B'; '4C'; '4D'; '4E'; '4F'; '50'; '51'; '52'; '53'; '54'; '55'; '56'; '57'; ...
            '58'; '59'; '5A'; '30'; '31'; '32'; '33'; '34'; '35'; '36'; '37'; '38'; '39'; ...
            '3D'; '28'; '29'; '2E'; '2D'; '2C'; '20'};

hex = cell(length(string)+2,1);
hex{1} = '05';
hex{2} = '02';
hex{length(string) + 3} = '03';
hex{length(string) + 4} = '04';
for i=1:length(string)
    for j = 1:length(characters)
        a = sprintf(string(i));
        b = sprintf(characters{j});
        if strcmp(a,b)
            hex{i+2} = hexvalues{j};
        end
    end
end
end
```

Auteursrechtelijke overeenkomst

Ik/wij verlenen het wereldwijde auteursrecht voor de ingediende eindverhandeling:
Vision-based control of robotic arm with 6 degrees of freedom

Richting: **master in de industriële wetenschappen: energie-automatisering**
Jaar: **2014**

in alle mogelijke mediaformaten, - bestaande en in de toekomst te ontwikkelen - , aan de Universiteit Hasselt.

Niet tegenstaand deze toekenning van het auteursrecht aan de Universiteit Hasselt behoud ik als auteur het recht om de eindverhandeling, - in zijn geheel of gedeeltelijk -, vrij te reproduceren, (her)publiceren of distribueren zonder de toelating te moeten verkrijgen van de Universiteit Hasselt.

Ik bevestig dat de eindverhandeling mijn origineel werk is, en dat ik het recht heb om de rechten te verlenen die in deze overeenkomst worden beschreven. Ik verklaar tevens dat de eindverhandeling, naar mijn weten, het auteursrecht van anderen niet overtreedt.

Ik verklaar tevens dat ik voor het materiaal in de eindverhandeling dat beschermd wordt door het auteursrecht, de nodige toelatingen heb verkregen zodat ik deze ook aan de Universiteit Hasselt kan overdragen en dat dit duidelijk in de tekst en inhoud van de eindverhandeling werd genotificeerd.

Universiteit Hasselt zal mij als auteur(s) van de eindverhandeling identificeren en zal geen wijzigingen aanbrengen aan de eindverhandeling, uitgezonderd deze toegelaten door deze overeenkomst.

Voor akkoord,

Versleegers, Wim

Datum: **6/06/2014**