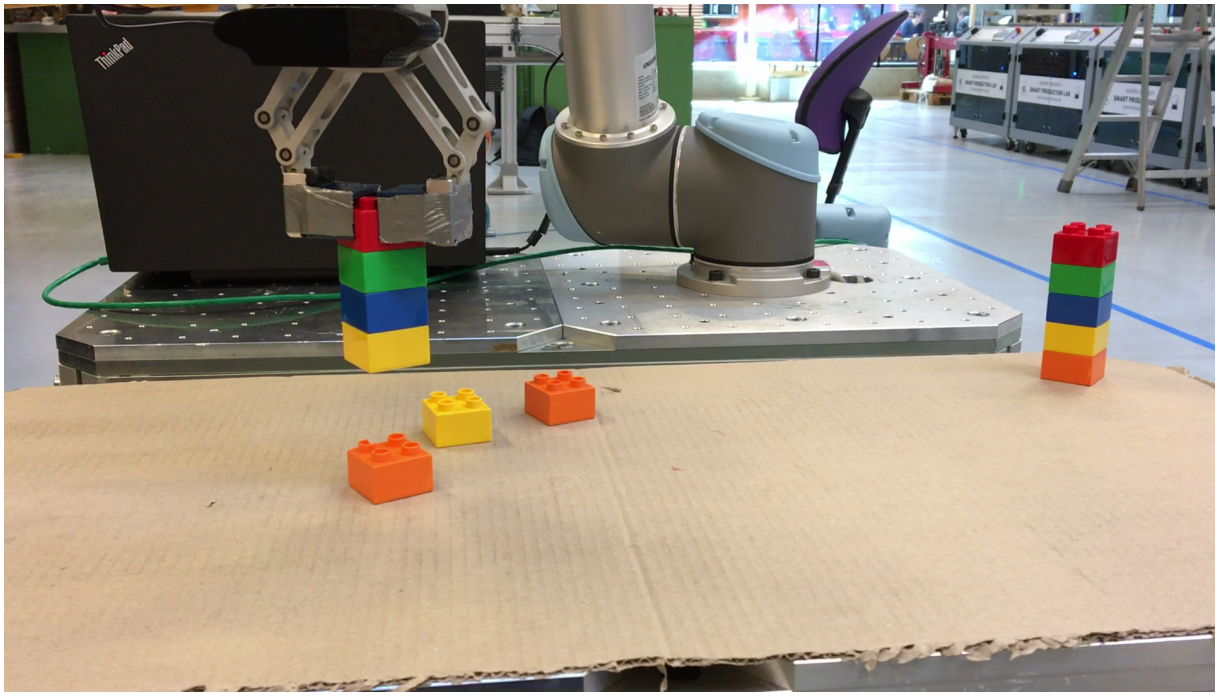
AALBORG UNIVERSITY
STUDENT REPORT

UR-Robot vision-based LEGO stacker



Mini-project Report
Robot Vision course
Group 831
Control & Automation
Spring 2017

Control & Automation
Fredrik Bajers Vej 7C
DK-9220 Aalborg Ø

AALBORG UNIVERSITY
STUDENT REPORT

Title:
Universal Robots vision-based LEGO stacker

Theme:
Robot Vision

Project Period:
2. Semester Control and Automation

Project Group: 831

Participant(s):

Chris Jeppesen

David Romanos

Joan Calvet Molinas

Malte Rørmose Damgaard

Thomas Kølbæk Jespersen

Supervisor(s):
Dimitris Chrysostomous
Kamal Nasrollahi
Sigurd Villumsen

Copies: 8

Page Numbers: 24

Date of Completion:
14. April 2017

Preface:

All ressources developed within the project and mentioned throughout this report, including both MATLAB code and the PolyScope URScript, is available in a GitHub respository: `https://github.com/mindThomas/URRobot-LEGOstacker`

A video demonstration of the working project can be found here: `https://www.youtube.com/watch?v=fLyhVfCzkio`

Contents

1 Introduction

For the mini-project within the Robot Vision the task is to program an industrial robot cell equipped with a camera to pickup LEGO DUPLO® bricks and build Simpson figures with these. The five figures from the Simpson family will consist of either two or three stacked 2x2 DUPLO bricks according to the color schemes listed below, indicating the color from bottom to top.

- Homer: Blue, White, Yellow
- Bart: Blue, Orange, Yellow
- Maggie: Blue, Yellow
- Lisa: Yellow, Red, Yellow
- Marge: Green, Yellow, Blue

Figure 1.1: Example of builded Simpson figures

The DUPLO bricks are located randomly (but not overlapping) on a table next to the robot. A camera located above the table is mounted such that all bricks to be used are within the view of the camera. This allows the camera to be calibrated and used with the robot to identify the color, location and orientation of all the bricks.

A customer can order any set of figures of which the robot should be able to automatically pick up and stack the required bricks in an optimal way. The solution should be be programmed in MATLAB and interface to the chosen robot over either USB or Ethernet.

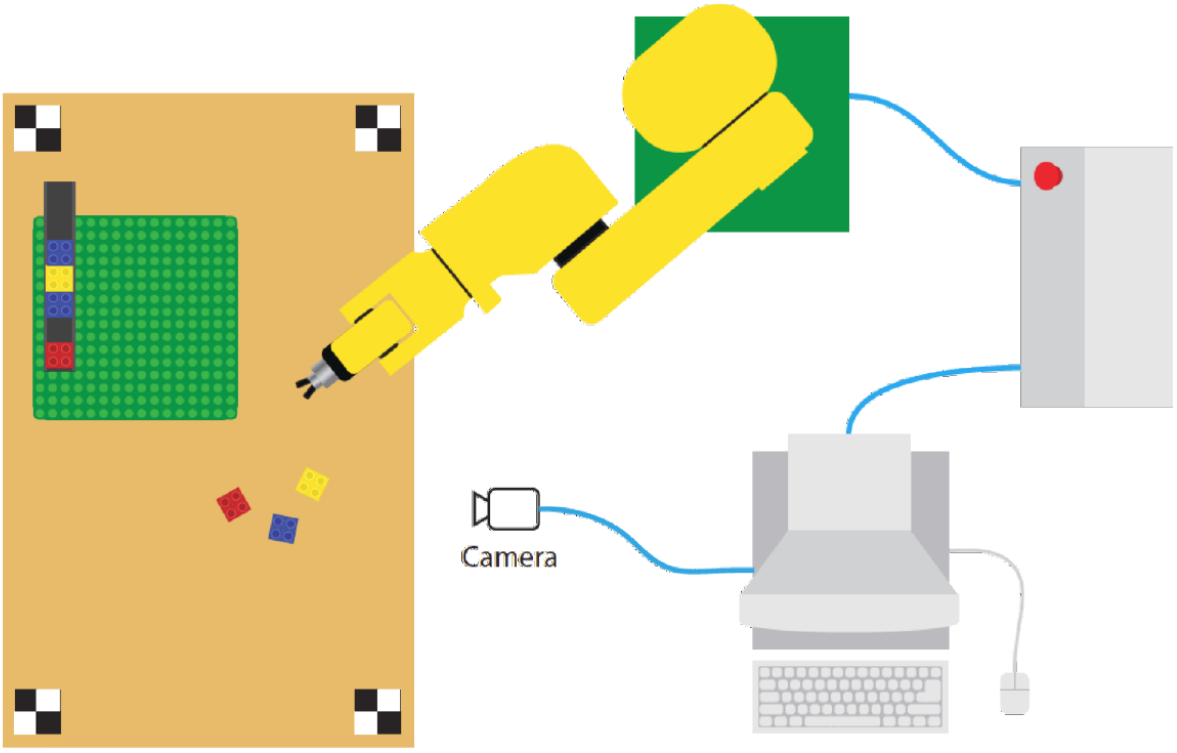


Figure 1.2: Robot cell configuration with camera and station area

In this mini-project report the analysis, design and implementation of the above mentioned solution is described.

In Chapter 2 a short analysis of the problem is given and the Universal Robots UR5 robot cell is chosen. The analysis is concluded in a block diagram showing the structure of the solution to be implemented.

In Chapter 3 the necessary image processing steps to extract the brick colors, locations and orientations are described, including normalization, thresholding, morphological operations and BLOB analysis. An algorithmic approach is used to describe how to implement the processing steps in MATLAB without usage of any existing toolboxes. Further optimizations of the proposed solution can be carried out by usage of existing toolboxes, but this is outside the scope of this report.

In Chapter 4 the interface and control of the chosen Universal Robots UR5 robot through MATLAB is described. Initially the programming of the robot is investigated and the necessary transformation matrices are presented. The resulting MATLAB integration is simulated on a virtual machine running the PolyScope teach pendent operating system [8].

In Chapter 5 the image processing is combined with the MATLAB control interface to the robot involving calibration of the detected camera-based object positions and the real-world positions. Finally the integration includes an analysis and design of the stacking algorithm which is implemented and tested on the robot.

2 Analysis and planning

In this chapter a description of the chosen robot arm, gripper, camera and robot cell layout is given. This results in a set of frames which are used for transformations in later chapters. Based on the chosen architecture, layout and the goal of picking up and stacking LEGO DUPLO bricks, a block diagram is given in the end showing the overall structure of the implemented solution.

2.1 Choice of robot

Picking up a LEGO DUPLO brick requires a robot that is capable of reaching a given 2D location on a table and adjust the orientation of the gripper to match the orientation of the brick. Furthermore the robot should be capable of picking up and stacking the bricks, why a z-dimension is necessary as well. This puts up a requirement of at least 4 degrees of freedom in the robot. If the table is not completely planar a 6-DOF robot would allow any position to be reached in space with any orientation, hence capable of picking up the LEGO brick from any surface.

The robots available in the M-Tech laboratory at Aalborg University [9] for use in this mini-project are all 6-DOF robots and includes robots from KUKA, FANUC, Adept and Universal Robots.

Based on the fact that the Universal Robots is a well-known danish invention used worldwide and highly expanding, it is chosen to work with this type of robot to get experience with their environment for possible use in future career opportunities. Furthermore Universal Robots have a great benefit of being classified as collaborable robots which means they can work together with humans without the need of big fences and closed environments. This will make the development easier as there will be an easy access to the robot and its' working environment, which would be easily adjustable as the robot is not fixed to a specific position in a cage. The available robot from Universal Robots is a UR5 which is chosen for this project.

2.2 Gripper and camera

The chosen UR5 robot came equipped with a two-finger servo-controlled gripper with a gripping force of a few millinewton. With this gripper it was decided that bricks should be picked up with the gripper fingers pointing downwards, hence the gripper tool frame turned upside down.

It is decided to install the camera on the robot arm, as shown in Figure 2.1, as it was not possible to locate any holders for the camera that would allow the camera to be installed in sufficient height above the table without colliding with the robot. By installing the camera on the robot, collisions are avoided completely and furthermore the location and orientation of the camera frame can be exactly known as long as just a mapping from the tool frame of the robot to the camera frame can be determined initially. The determination of this mapping together with the determination of the transformation from camera pixels to camera frame coordinates in meters is described in Section 5.1.

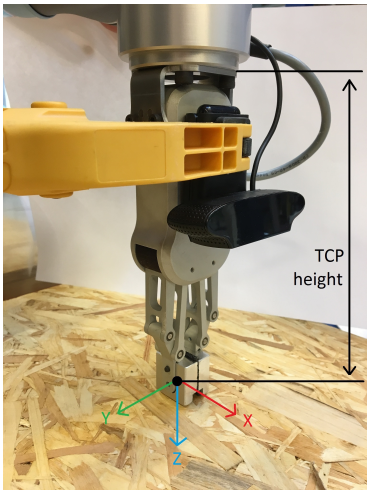


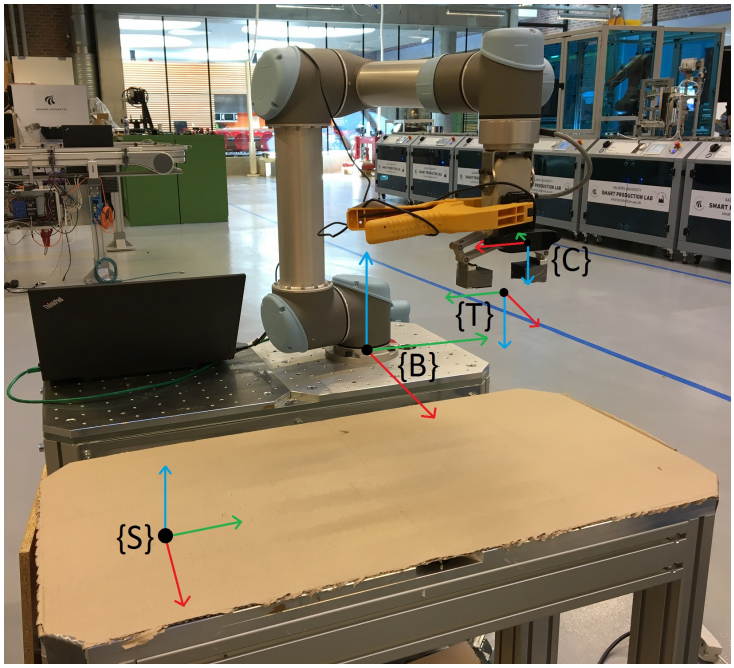
Figure 2.1: Camera mounted on gripper with marked tool center point

2.3 Robot cell layout

Both knowing the chosen 6-DOF robot arm, the gripper type and how the camera is mounted to the gripper allows the robot cell layout to be designed. From the design the following set of frames are placed:

$\{B\}$: Robot base frame
$\{T\}$: Robot tool frame
$\{C\}$: Camera frame
$\{S\}$: Station frame with bricks
$\{S\}$: Station frame for placement

(a) Frame identifiers



(b) Pictured frames on the robot cell layout

Figure 2.2: Robot cell layout with chosen frames

It is chosen to combine the two station frames into one frame indicating the location of the table surface from which the bricks are supposed to be picked up and on which the stacked figures will be placed. The actual layout of the robot with the frames pictured is shown in Figure 2.2.

Frames are used to describe poses which consists of both a translational and rotational part. These parts can be described in different ways, either separately or combined. The translation are given in meters according to the local frame axes, X, Y, Z. The rotation can be given as

Roll, Pitch, Yaw angles, as a quaternion or a rotation matrix. The translational and rotational part can be combined by using the SO(4) description (4-dimensional Euclidean space), being a homogeneous 4x4 matrix representation of the pose. The matrix includes the 3x3 rotation matrix and the 3-dimensional translation vector.

A transformation matrix, ${}^{A}_{B}\mathbf{P}$, describes the transformation from frame $\{B\}$ to frame $\{A\}$ with a translation, T_{3x1}, corresponding to the origo of frame $\{B\}$ in frame $\{A\}$ and rotation, $\mathbf{R}_{3x3}$, corresponding to the rotation of frame $\{A\}$ relative to frame $\{B\}$.

$$ {}^{A}_{B}\mathbf{P} = \begin{bmatrix} \mathbf{R}_{3x3} & T_{3x1} \\ \mathbf{0}_{1x3} & 1 \end{bmatrix} \tag{2.1} $$

Such a transformation is capable of transforming a vector given in frame $\{B\}$ into a vector given in $\{A\}$. A transformation is applied by multiplying the vector given in frame $\{B\}$ with the SO(4) transformation matrix. Notice the use of homogeneous vectors.

$$ \begin{bmatrix} {}^{A}T \\ 1 \end{bmatrix} = {}^{A}_{B}\mathbf{P} \begin{bmatrix} {}^{B}T \\ 1 \end{bmatrix} \tag{2.2} $$

It is decided to use the SO(4) representation to describe all frames and transformations throughout the report.

2.4 Block diagram

Figure 2.3 shows the proposed structure of the system.

First an image is acquired from a camera. Then the image is processed in an image processing part, by which the location and rotation of all bricks in the image are obtained. These positions and rotations is then processed further to convert them into the robot base frame. The positions and rotations obtained from that is then past to a stacker algorithm that chooses which bricks to stack on top of each other, and in which order. These decisions is then past on to the robot controller script, to make the robot stack the bricks.

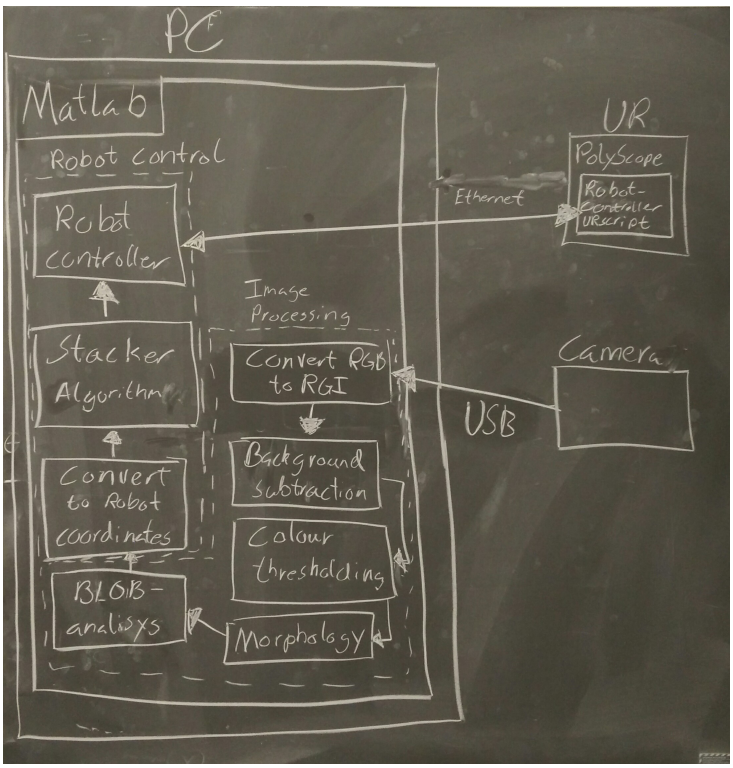


Figure 2.3: Block diagram of proposed solution structure

3 Image processing

The goal of the image processing part of the system is to locate the different bricks in an image. The output of the image processing part should be an XY-coordinate for the centre of the bricks given in the image coordinate system, and a rotation of the bricks. Furthermore the information of the bricks should be sorted by colours.
It has been chosen to divide the image processing into the following steps:

1. Convert the raw RGB image to an RGI image to separate colour information from intensity information
2. Perform background subtraction to remove the static background
3. Perform colour thresholding for each of the different colours of bricks on the RGI image to segment the image and obtain a binary image for each of the brick colours.
4. Perform morphology (neighbourhood processing) to remove noise from the binary images
5. Perform BLOB-extraction on the binary images to separate different bricks of the same colours and remove any noise left.
6. Perform feature extraction to extract information about the different bricks from the BLOBs.

The different steps is described in the following sections, and the image processing is illustrated by the use of the following picture taken while performing the test of the system, as shown in the attached video named 'Universal Robots vision-based LEGO DUPLO stacker.mp4'.

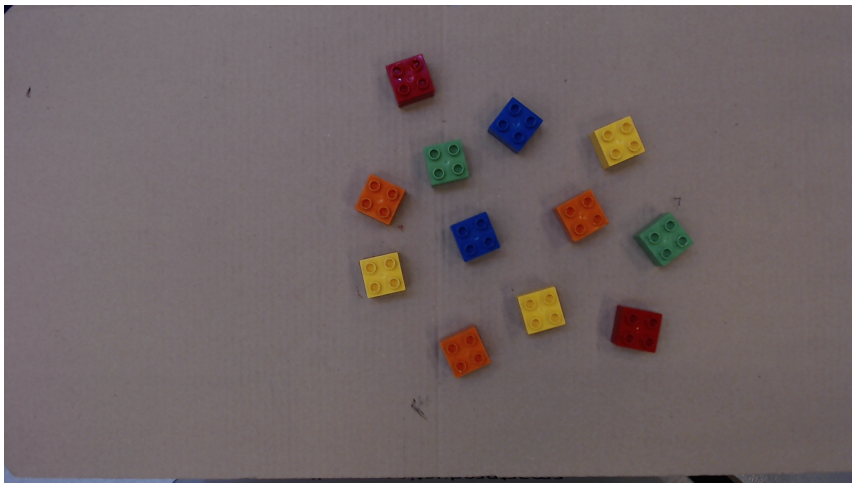


Figure 3.1: Raw RGB image of resolution 1920×1080

3.1 RGB to RGI Conversion

Calculations and functionalities described in this section is is implemented in the function **RGB2RGI2(rgbImage)**.
Before the RGB to RGI conversion is performed all pixels in the RGB image having a value of 255 in any of the colour channels is put to (0,0,0). This is because there is a risk that one of the colour channels of this pixel was saturated in the image acquisition process resulting in a value

of 255. This could lead to incorrect interpretation of the colour later in the image processing, and should thus be discarded. An RGB pixel with values of (255, 250, 250) would be interpreted as white, but could have real values of (10000, 250, 250) and should thus be interpreted as a red colour. Since it is impossible to know the actual value of a saturated pixel, it is better to discard those pixels. Therefore such pixels are put to (0,0,0).

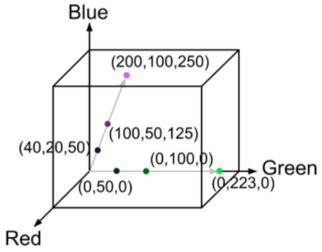


Figure 3.2: Colours represented in the RGB space [3]

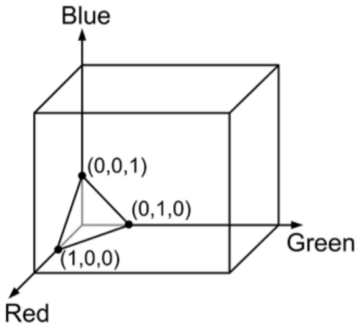


Figure 3.3: The plane in the RGB space that the normalized colours span [3]

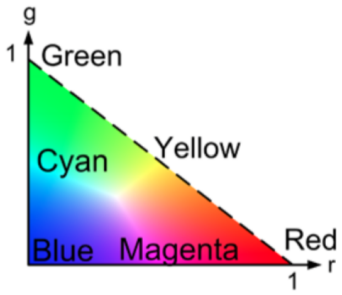


Figure 3.4: The chromaticity plane [3]

In the RGB colour space all colours can be described as vectors with lengths limited by a box, as shown in Figure 3.2. As it is illustrated in Figure 3.2 vectors pointing in the same direction is just different shades of the same colour, with only the illumination being different. By normalizing the vectors describing different colours, new vectors are obtained that all will lie in the triangular plane whose corner points are defined as $(0, 0, 1)$, $(0, 1, 0)$ and $(1, 0, 0)$ as shown in Figure 3.3. Due to the normalization $r_{\text{norm}} + g_{\text{norm}} + b_{\text{norm}} = 1$, only two of the values are needed to fully describe a normalized colour. If r_{norm} and g_{norm} is chosen then the triangle in Figure 3.3 can be represented by the triangle in Figure 3.4 called the chromaticity plane. These normalized colours can be seen as "pure" colours that is not influenced by illumination.
The colour description can be completed with the intensity/illumination. The RGB to RGI conversion is performed with pixel-wise calculation given in (3.1).

$$(r, g, I) = \left(\frac{r}{r+g+b}, \frac{g}{r+g+b}, \frac{r+g+b}{3} \right) \tag{3.1}$$

In theory this RGI representation should allow colour thresholding on any image independent of the illumination level and light intensity. As an example the normalized RGB description of Figure 3.1 is shown in Figure 3.5

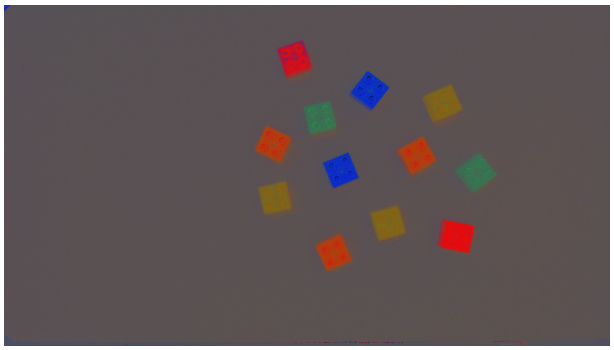


Figure 3.5: The image shown in Figure 3.1 converted to a normalized RGB image. The r_{norm} and g_{norm} channels are calculated like in (3.1), while the b_{norm} channel is only calculated for visualization purposes utilizing the fact that $r_{\text{norm}} + g_{\text{norm}} + b_{\text{norm}} = 1$

3.2 Background Subtraction

Calculations and functionalities described in this section is implemented in the function **BackgroundSubtraction(rgiImage, BG_threshold)**.
To be able to perform a background subtraction an image of the table is taken before any bricks is placed. This background RGB image is then converted into the RGI representation as described in Section 3.1. The RGI image of the background is denoted as B_{RGI}.
To perform the background subtraction the pixel-wise operation shown in (3.2) is applied on the individual r-, g- and I-channel within the RGI image of the table with bricks, I_{RGI}.

$$F(x,y) = I_{RGI}(x,y) - B_{RGI}(x,y) \tag{3.2}$$

Based on $F_r(x,y)$, $F_g(x,y)$ and $F_i(x,y)$, being the resulting channels of the RGI image after subtraction, it is decided if a pixel in I_{RGI} belongs to the background or not. This is done by comparing each entry in $F_r(x,y)$, $F_g(x,y)$ and $F_i(x,y)$ to a threshold, $BG_{\text{threshold}}$. If just one of the thresholds, $|F_r(x,y)| > BG_{\text{threshold}}$, $|F_g(x,y)| > BG_{\text{threshold}}$ or $|F_i(x,y)| > BG_{\text{threshold}}$, is true, then the corresponding pixel in I_{RGI} is kept as a non-background pixel. Otherwise the corresponding pixel in I_{RGI} is put to (0,0,0) to mark the pixel as background.
Performing this background subtraction on the image shown in Figure 3.5 results in the image shown in Figure 3.6

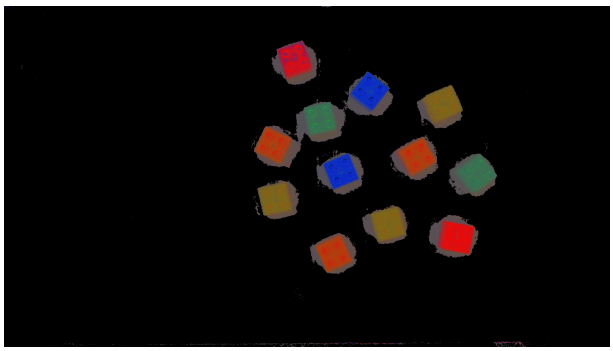


Figure 3.6: The normalized RGB image with removed background.

3.3 Colour Thresholding

Calculations and functionalities described in this section is implemented in the function **colorThresholding(rgiImage, config)**. This function is called for the different colours of bricks separately. Each of the colours of bricks have their own config struct, that among other things defines maximum and minimum threshold values for the r_{norm} and g_{norm} channels, and a minimum threshold value for the i channel in the RGI image. The acquisition of threshold values for the r_{norm} and g_{norm} channels is described in Section 3.3.1.
Before the colour thresholding is performed on the RGI image, the intensity level of each pixel is checked. If a pixel has too low intensity it is put to 0 in the binary output image. This is done because it is hard to distinguish colours when the intensity is low and thus it is better to ignore such pixels. In practice this thresholding on the intensity is performed by comparing each pixels in the I-channel of the RGI image with the minimum threshold value for the intensity, defined in the config struct.
After performing the intensity thresholding, a thresholding is performed on the r_{norm} and g_{norm} channels. In practice this is performed by comparing each pixels in the r_{norm} and g_{norm} channel

of the RGI image with the upper and lower threshold values of the specific colour. If both the r_{norm} and g_{norm} channel satisfies the threshold boundaries the corresponding pixel is put to 1 in the binary output image to mark it as a brick pixel with the given colour. Otherwise it is put to 0.

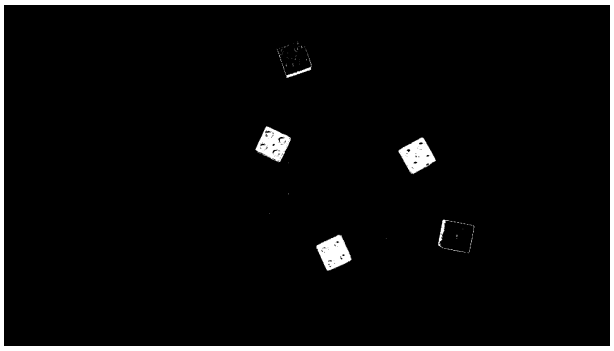


Figure 3.7: Binary image resulting from performing orange colour thresholding on the RGI image shown in Figure 3.6

An example of the binary image resulting from colour thresholding on the RGI image shown in Figure 3.6 with the thresholds defined for the orange brick is shown in Figure 3.7.

3.3.1 Colour Threshold analysis

Calculations and functionalities described in this subsection is implemented in the script **thresholdAnalysis.m**
To define threshold values for the r_{norm} and g_{norm} channels for the different colours of bricks the following is done.

- An RGB image is taken with different coloured bricks on the table.
- The RGB image is converted into RGI representation.
- For each of the different colours of bricks, a region of the RGI image containing that colour is cropped out of the RGI image. This results in 6 small images. One for each of the brick colours being red, green, blue, yellow, white and orange.
- On each of the small images the 5% and 95% percentile is calculated for both the r_{norm} and g_{norm} channels.
- 5% and 95% percentiles calculated for each of the 6 small images are used as the lower and upper threshold values for the respective brick colours.

3.4 Morphology

Calculations and functionalities described in this section is implemented in the function **removeNoise(binaryImage, config)**.
As it can be seen in Figure 3.7 not only actual brick pixels are present within the binary image after colour and intensity thresholding. The binary image also contain some noise. Furthermore there are holes in the interior of some of the bricks, which is inconvenient e.g. when the centre of mass has to be calculated later as part of the feature extraction.

To remove this noise from the image, two consecutive compound operations are performed. First a closing operation is performed to fill holes in the interior of the bricks. Then an opening operation is performed to remove noise. As illustrated in Figure 3.8 a closing compound operation consist of a dilation followed by an erosion operation, where an opening compound operation consist of an erosion followed by a dilation operation.

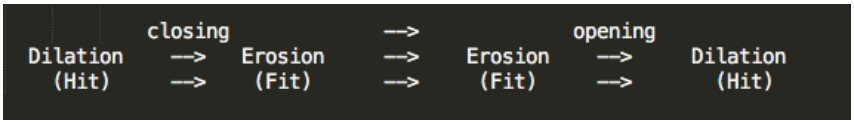


Figure 3.8: Illustration of the operations performed in the morphological process

To perform a dilation operation a structuring element (kernel) is defined. For each pixel in the picture the origin of this structuring element is placed on top. When calculating the output pixel the origin of the structuring element defines the output pixel location. For each of the 1's in the structuring element it is checked if the corresponding pixel in the input image (below the structuring element) is equal to a 1 as well. If this is true for just one of the pixels, then the output pixel is put to a 1. Otherwise it is put to 0. This operation performed for a single pixel is called a Hit operation. An erosion operation is performed in a similar way, except that for the output pixel to be 1, all pixels in the input image covered by a 1 in the structuring element, has to be 1 as well. This operation performed for a single pixel is called a Fit operation.
For this project it is chosen to use box-shaped structuring elements since this tend to preserve sharp edges which a LEGO dublo brick has, and because it is important to preserve the correct form of the brick for the feature extraction.

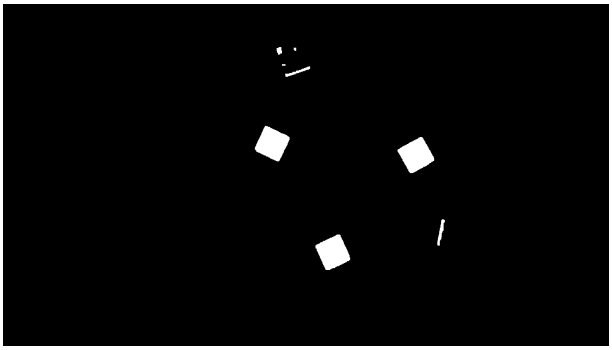


Figure 3.9: Result of performing morphology on Figure 3.7 (binary image of orange bricks) using a 10×10 closing structuring element and 5×5 opening structuring element

The result after performing morphology on the binary image of the orange bricks, resulting from the colour thresholding described in Section 3.3, is shown in Figure 3.9.

3.5 BLOB analysis

Calculations and functionalities described in this section is implemented in the function **blobAnalysis(binaryImage, config)**. The goal of the BLOB analysis in this project is to locate and determine the position of the different bricks in the image and determine their rotation. Before this can be done, the binary images resulting from the morphology described in Section 3.4 has to be converted into a format where all the pixels, which constitute a single brick, is classified and grouped such that they can easily be recognized and distinguished. To do so the recursive grass-fire algorithm can be used. After running the recursive grass-fire algorithm a BLOB is considered to be a brick if its mass is above a certain threshold. If a BLOB is considered a brick then the centre and rotation is calculated. The process is described in further details below.

3.5.1 Recursive grass-fire algorithm

To connect the pixels that constitutes a brick we have chosen to use the recursive grass-fire algorithm with 4-connectivity. The 4-connectivity is chosen since it is faster than 8-connectivity. Furthermore it is assumed that after morphology has been performed on the binary images the pixels constituting one brick will be closely connected, and thus 8-connectivity has no advantages compared to 4-connectivity. The recursive grass-fire algorithm with 4-connectivity is in this project implemented in the function **BLOB_recursive(n, m, obj)**. The output of this algorithm is a new image where each pixel has a number according to the BLOB it belongs to. Pixels not belonging to any BLOB has a value of 0, where pixels belonging to BLOB 9 has a value of 9 and so on.

3.5.2 Centre of Bricks

It has been chosen to use the centre of mass as a measure of the centre position of the bricks. The centre of the bounding box of a BLOB could have been used as an estimate of the centre position, though such an estimate is vulnerable to noise pixels in the boundary of a BLOB. That is a single pixel could potentially move the centre of the bounding box a lot. Thus it has been chosen to use the centre of mass since this is not vulnerable to single noise pixels. The centre of mass is calculated as shown in (3.3).

$$x_m = \frac{1}{N} \sum_{i \in \text{obj}} x_i \qquad y_m = \frac{1}{N} \sum_{i \in \text{obj}} y_i \tag{3.3}$$

This calculation is implemented in the function **find_center_of_mass(obj, image)**, where **image** is the output image of the grass-fire algorithm, and **obj** is the identifier of the object that the centre of mass should be calculated for.

3.5.3 Rotation

After determining the centre of a brick the only thing left is to determine the rotation of the brick. Finding the rotation of a brick is implemented in the function **find_rotation2(j, BLOB_output_img, Center_of_mass)**, where **j** is an identifier of the brick and **BLOB_output_img** is the output image of the grass-fire function. The method of finding the rotation of a brick can be summarized to:

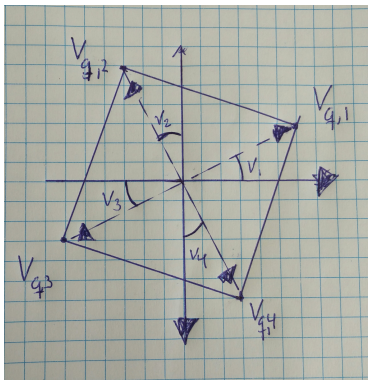


Figure 3.10: Corner-based determination of the rotation using vectors

- Calculate the distance from each pixels constituting a BLOB to the centre of mass of the BLOB.
- Find the pixels in each quadrant being furthest apart from the centre of the BLOB (see Figure 3.10) within a coordinate system whose origin is in the centre of the BLOB.
  - Here we utilize the fact that bricks seen by the camera is square, and thus the pixels furthest away from the centre in each quadrant must be the corners of the brick.
- Define the 4 vectors, $v_{q,1}$,$v_{q,2}$,$v_{q,3}$ and $v_{q,4}$, going from the centre of the BLOB to the pixel in each quadrant being furthest away from the centre.
- Calculate the angles v_1, v_2, v_3 and v_4 as in (3.4).
- If noise is present in the image, then v_1, v_2, v_3 and v_4 will not be exactly the same, and thus the rotation of the BLOB is calculated as the mean of the angles minus 45°, as a horizontal brick should correspond to 0°.

$$\begin{aligned} v_1 &= \tan^{-1}\left(\frac{-v_{q,1,y}}{v_{q,1,x}}\right) & v_3 &= \tan^{-1}\left(\frac{-v_{q,3,y}}{v_{q,3,x}}\right) \\ v_2 &= \tan^{-1}\left(\frac{v_{q,2,x}}{v_{q,2,y}}\right) & v_4 &= \tan^{-1}\left(\frac{v_{q,4,x}}{v_{q,4,y}}\right) \end{aligned} \tag{3.4}$$

Furthermore in the case that a brick is angled close to 45 degrees (approximately between 40° and 50°), there is a risk that there will be two long vectors in the same quadrants, hence making it seem like there is two corners in one quadrant. As there is no control of which of the lengths will be used, one angle might yield a completely different angle than the others, which due to the mean will result in an incorrect estimate of the rotation.
To counteract this, an if statement is included that checks whether just two of the angles v_1, v_2, v_3 and v_4 is higher than 40°. If so the rotation of the brick is put to 45°.
Finally the estimation of rotation is also prone to noise when the angle of a brick is close to 0 degrees (approximately between −5° and 5°) due to the fact that noise pixels at the edge of the brick might get classified as the longest vector, hence affecting the estimate of the rotation.

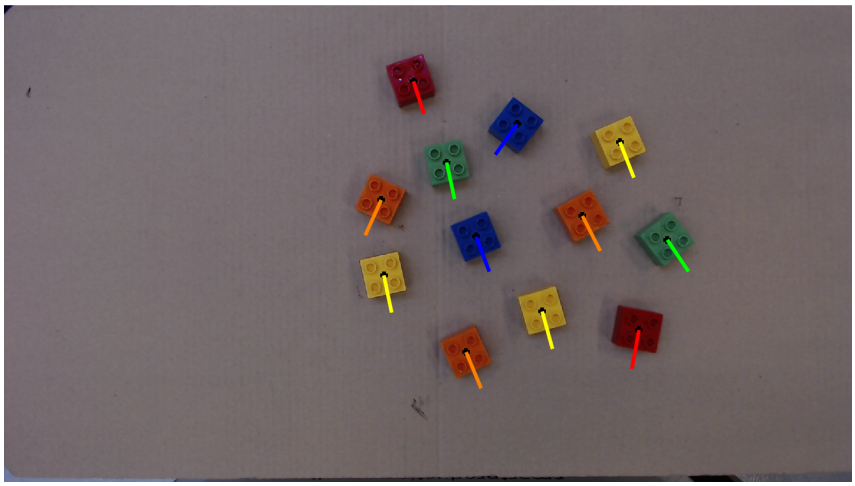


Figure 3.11: Illustration of the bricks found by the image processing. Furthermore the estimated centre of the bricks are marked by a black dot, and the estimate rotations are illustrated by lines pointing in the direction of the rotation. The colour of these lines furthermore indicates what colour each brick has been categorized as.

Figure 3.11 shows the estimated centre and rotation of the bricks found from the binary images shown in Figure 3.9.

4 Robot control

The chosen robot as described in Section 2.1 is a Universal Robots UR5. The robot contains 6 degrees of freedom and is capable of lifting up to 5 kg and reaching up to 850 mm out.

4.1 Universal Robots programming

Universal Robots are programmed through the connected teach pendant including the graphical PolyScope programming interface [7]. The robots can be programmed either visually by moving the robot and setting waypoints manually or they can be programmed in the URScript programming language [6]. Both methods can be typed in through the teach pendant and saved as a program to be executed on the robot.

4.1.1 URScript commands

The URScript commands to be used in this project involves initialization of the robot position by joint movement, moving the robot arm in the Cartesian coordinate system and opening and closing of the gripper.
The joint movement command defined as 'movej' can be used to specify desired joint angles of the 6 robot arm joints, of which the robot will move to by a joint angle interpolation with the desired velocity and acceleration.

movej(*q*, *a*=1.4, *v*=1.05, *t*=0, *r*=0)

Move to position (linear in joint-space) When using this command, the robot must be at standstill or come from a movej og movel with a blend. The speed and acceleration parameters controls the trapezoid speed profile of the move. The t parameters can be used in stead to set the time for this move. Time setting has priority over speed and acceleration settings. The blend radius can be set with the r parameters, to avoid the robot stopping at the point. However, if he blend region of this mover overlaps with previous or following regions, this move will be skipped, and an 'Overlapping Blends' warning message will be generated.

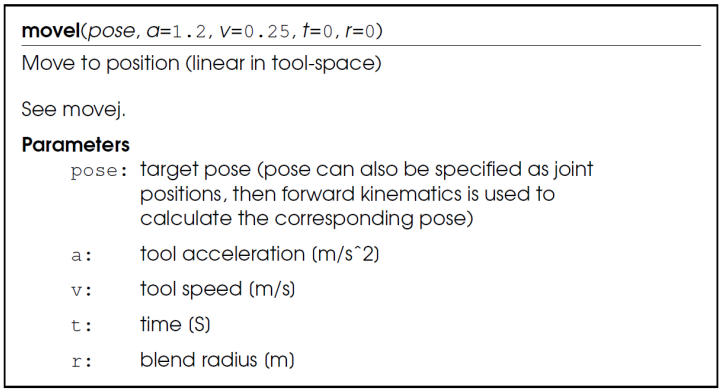
Parameters

- `q`: joint positions (q can also be specified as a pose, then inverse kinematics is used to calculate the corresponding joint positions)
- `a`: joint acceleration of leading axis (rad/s^2)
- `v`: joint speed of leading axis (rad/s)
- `t`: time (S)
- `r`: blend radius (m)

Figure 4.1: Joint movement URScript function [6]

The joint movement is useful if a certain position is to be achieved in a specific configuration (angular configuration). This allows the robot to be forced into a specific configuration where the inverse kinematic used for linear movement would otherwise have forced the robot into a different configuration.

For the linear movement the 'movel' command can be used to specify a desired 3D position and orientation of the tool with origo in the base frame. When calling the linear movement command the robot will move to the specified position and orientation with linear interpolation why it is important to take care of possible singularities.



movel(*pose*, *a*=1.2, *v*=0.25, *t*=0, *r*=0)

Move to position (linear in tool-space)

See movej.

Parameters

`pose`: target pose (pose can also be specified as joint positions, then forward kinematics is used to calculate the corresponding pose)

`a`: tool acceleration (m/s^2)

`v`: tool speed (m/s)

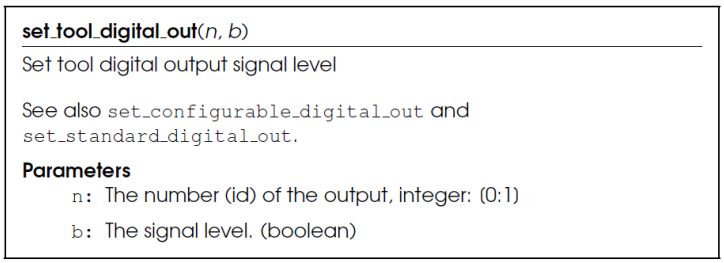
`t`: time (S)

`r`: blend radius (m)

Figure 4.2: Linear (Cartesian) movement URScript function [6]

According to the URScript reference manual [6] Universal Robots use a 6-dimensional vector with floats to describe poses. In a pose, the first 3 coordinates is a position vector and the last 3 an axis-angle. The position elements defines the X,Y,Z coordinates in meters and the axis-angle elements defines an angle of which the rotation of the tool frame is performed and the length indicates the amount to turn. A further description is given in Section 4.2.

To be able to pick up the LEGO DUPLO bricks it should be possible to automate the opening and closing of the gripper mounted to the robot, presented in section Section 2.2. Gripper control can be performed using the 'set_tool_digital_out' command where the output number and signal level is specified, which is the simplest way to control an actuated gripper tool.



set_tool_digital_out(*n*, *b*)

Set tool digital output signal level

See also `set_configurable_digital_out` and `set_standard_digital_out`.

Parameters

`n`: The number (id) of the output, integer: (0:1)

`b`: The signal level. (boolean)

Figure 4.3: Gripper control URScript function [6]

A high signal (true) would close the gripper and a low signal (false) would open the gripper. The gripper mounted on the UR5 robot used in this project can only supply a few millinewton of pressure, why the gripper will automatically stop whenever any form of resistance is met, such as when a brick has been grabbed.

4.1.2 Connecting to the robot over Ethernet

Movement to given waypoints or the execution of individual URScript commands can be connected in a desired sequence by using the graphical PolyScope programming environment. Though instead of using preprogrammed waypoints a computer running MATLAB should be used to send commands to the robot.

The robot comes equipped with both a USB host interface (for USB memory sticks) and an Ethernet interface. To connect the robot to a PC for control, the Ethernet interface would have to be used and the network card of the PC should be configured to be on the same subnet as the robot. The IP address of the robot can be found in the 'About' page on the teach pendant. Configure the computer to be the gateway by making sure that both the computer IP and gateway IP is the same in the network card configuration, preferably ending in "x.x.x.1". Replug the Ethernet cable after configuring.
When the network card has been configured and the Ethernet cable has been replugged the connection to the robot can be verified by pinging the IP address of the robot. Open a terminal and write "ping x.x.x.x" where x.x.x.x should be replaced by the IP address of the robot. If the robot does not respond to the ping requests the network has not been configured appropriately.

The actual MATLAB control of the robot over Ethernet is described in section Section 4.3.

4.1.3 Simulated control

As part of the Robot Vision course the ABB Robot Studio was presented as a possible simulation environment in which ABB robots could be programmed with vision systems being included. Unfortunately there are no similar simulation software for the Universal Robots which would allow both the robot simulation and vision system integration together with MATLAB control.

One possible simulation environment that includes support for the Universal Robots arms is RoboDK [5]. This software allows design of robot cells with several different high-end robot arm brands including simulation of vision or laser-based sensor integration. Furthermore the software allows direct connection to the chosen robot or export of the developed robot program to the specific robot arm language. The actual programming of the robot arm within the simulation environment is done in a Python type of language. Unfortunately the simulated vision images can not be exported to MATLAB directly, why it would not be possible to simulate the actual implementation as it would be with ABB Robot Studio.

Although no complete simulation environment supporting both MATLAB control and simulated vision is available for the UR5, Universal Robots has released a virtual Linux machine called URSim [8], identical to the operating system running on the PolyScope teach pendant. This virtual machine allows URScript programs to be simulated on a PC and even virtual Ethernet connections to be made to allow simulation and test of the PC communication interface and MATLAB control.

4.1.4 Transferring a program over Ethernet

When a PolyScope program has been made, either within the simulation environment or on the physical teach pendant, the program can be copied (or backed up) through a USB memory stick or over Ethernet. To copy a file over Ethernet an SSH connection has to be used to transfer files to/from the robot. For Windows users the program 'WinSCP' can be used as SFTP client to connect to the robot. Within WinSCP enter the IP address of the robot.
When asked about username and password enter 'root' or 'ur' as the username and 'easybot' as the password. The home would usually appear automatically when connected. All saved scripts are stored within '/home/ur/scripts'.

4.2 Conversion of SO(4) transformation matrices

As mentioned in Section 2.3 it has been decided to describe all the frames contained within the robot cell with SO(4) transformation matrices. It should be possible to give the robot arm target as an SO(4) matrix why a conversion to the 6-dimensional UR pose is needed. The UR pose vector is shown in (4.1).

$$p_{UR} = \begin{bmatrix} T_{UR} \\ R_{UR} \end{bmatrix}^T = \begin{bmatrix} x & y & z & r_x & r_y & r_z \end{bmatrix} \tag{4.1}$$

An SO(4) transformation matrix transforming from the tool frame $\{T\}$ to the base frame $\{B\}$ is characterized by the elements shown in (4.2).

$${}^B_T\mathbf{P} = \begin{bmatrix} \mathbf{R} & T \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} r_{xx} & r_{xy} & r_{xz} & x \\ r_{yx} & r_{yy} & r_{yz} & y \\ r_{zx} & r_{zy} & r_{zz} & z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.2}$$

The translational part of the UR pose is identical to the translational part of the SO(4) matrix, T. The conversion from the rotational part, composed of a common 3x3 rotation matrix, to the axis-angle description requires a few steps.

First the rotation matrix is converted to a rotation angle, θ, and a rotation axis, ${}^AK$ [2].

$$\theta = \cos^{-1}\left(\frac{r_{xx} + r_{yy} + r_{zz} - 1}{2}\right) \qquad {}^AK = \frac{1}{2\sin\theta}\begin{bmatrix} r_{zy} - r_{yz} \\ r_{xz} - r_{zx} \\ r_{yx} - r_{xy} \end{bmatrix} = \begin{bmatrix} k_x \\ k_y \\ k_z \end{bmatrix} \tag{4.3}$$

Thereafter the axis and angle is combined into a single 3-dimensional vector, by forming a vector of the axis unit vector whose length is extended to reflect the rotation angle amount.

$$R_{UR} = {}^AK\theta \rightarrow \begin{bmatrix} r_x \\ r_y \\ r_z \end{bmatrix} = \begin{bmatrix} k_x\theta \\ k_y\theta \\ k_z\theta \end{bmatrix} \tag{4.4}$$

Converting back simply requires doing the opposite conversion. First the rotational vector from the UR pose has to be converted into rotation axis and rotation angle.

$$\theta = |R_{UR}| \qquad {}^AK = \frac{R_{UR}}{\theta} \tag{4.5}$$

An axis-angle rotation is performed by starting with a frame $\{B\}$ coincident with a known reference frame $\{A\}$. Frame $\{B\}$ is rotated about the vector ${}^AK$ by an angle θ according to the righthand rule. This yields the rotation matrix shown below where $c\theta = \cos\theta$ and $s\theta = \sin\theta$

$$\mathbf{R} = \begin{bmatrix} k_xk_x(1-c\theta) + c\theta & k_xk_y(1-c\theta) - k_zs\theta & k_xk_z(1-c\theta) + k_ys\theta \\ k_xk_y(1-c\theta) + k_zs\theta & k_yk_y(1-c\theta) + c\theta & k_yk_z(1-c\theta) - k_xs\theta \\ k_xk_z(1-c\theta) - k_ys\theta & k_yk_z(1-c\theta) + k_xs\theta & k_zk_z(1-c\theta) + c\theta \end{bmatrix} \tag{4.6}$$

4.2.1 Configuring the tool center point

The tool center point is defined to be at the tip of the grippers when they are closed. The tool center point location relative to the wrist frame has to be configured through the teach pendant. For the used gripper shown in Figure 2.1, the TCP offset is configured to a z-offset of 213 mm.

4.3 MATLAB interface

There are two approaches to control the robot arm over Ethernet [1]:

- URScript commands sent over TCP/IP port 30002 [6]
- TCP/IP socket connection to own local URScript

Both methods requires a TCP/IP socket to be opened and used to exchange commands and values. Method 1 allows the user to execute all sort of URScript commands as explained in the URScript manual [6]. Method 2 requires a local URScript to be developed and executed on the PolyScope teach pendant which connects to a socket on the host computer from where it can receive messages and react accordingly in a pre-programmed manner. To get knowledge of both teach pendant programming and the MATLAB interface it is decided to use Method 2.

Method 2 consists of the development of a command parser that will parse the incoming messages sent from MATLAB and execute movements and actions accordingly. When a movement has been finished the script should furthermore send a status message back indicating that the movement has finished. This allows the MATLAB script to be programmed to wait for a movement action to finish before continuing with the next. At a bare minimum the local URScript should therefore support movement commands, finish messages and opening and closing of the gripper.

As a special cooperable feature the Universal Robots supports a free-drive mode where the operators interacts with the robot by moving it to a desired position simply by dragging the tool by hand. Teach mode is the programmable version of free-drive and will allow the robot to be moved around by hand in the same way as by pressing the free-drive button on the back of the teach pendant, though the robot will not be able to follow movement commands at the same time. To simplify the camera to robot calibration this teach mode would allow the operator to move the robot to a calibration brick of which the image processing will later calculate the position. Further details are described in Section 5.1. Enabling teach mode by calling the URScript command 'teach_mode()' should also be included in the local URScript.

If the robot position is moved by an operator, eg. in teach mode, it is necessary to know the new pose of the robot. With the 'get_actual_tcp_pose()' command it is possible to get the current robot pose as the 6-dimensional vector which can be sent back to the host computer over the TCP/IP socket.

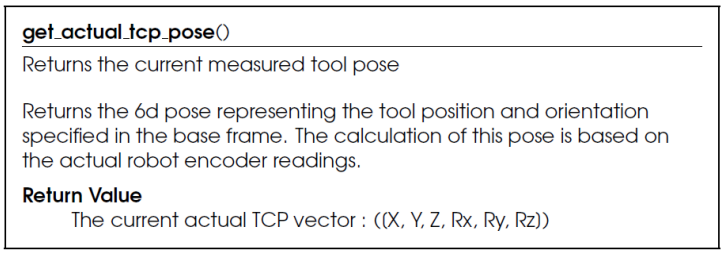


Figure 4.4: URScript function to get current pose of TCP [6]

This concludes the necessary commands developed into the local URScript. To summarize the supported commands are:

MATLAB → Robot

- Move to UR pose
- Open gripper
- Close gripper
- Request current pose
- Enable Teach mode
- Disable Teach mode

Robot → MATLAB

- Moving finished event
- Return current pose

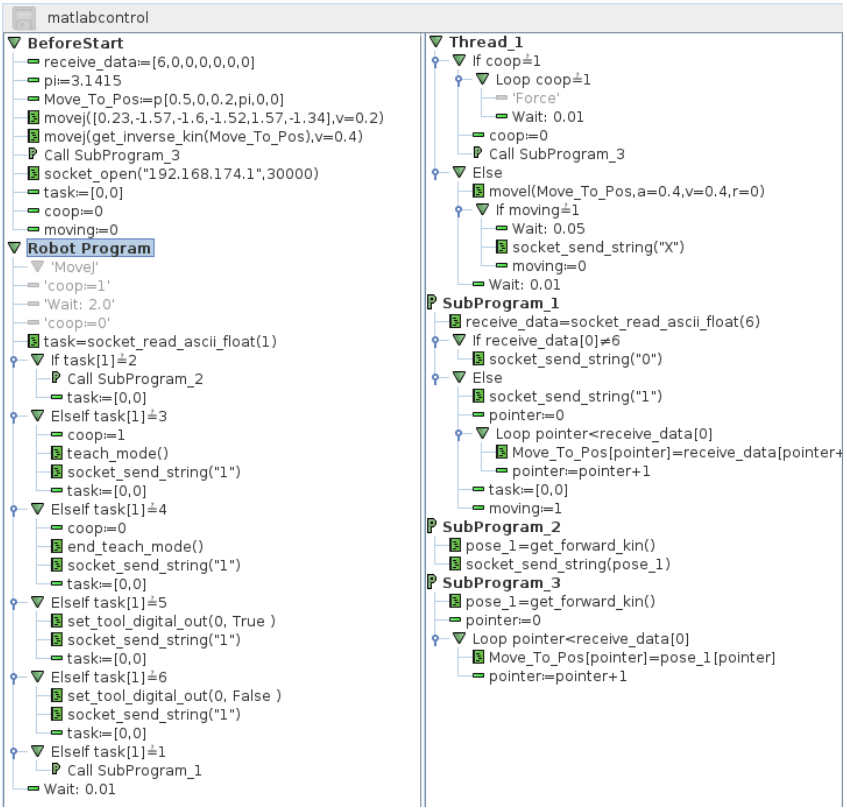


Figure 4.5: Local URScript implementing the necessary features

The URScript implementing these features are shown in Figure 4.5. The marked 'Robot Program' and 'Thread_1' are both threads which run in parallel. 'Robot Program' handles the arrival of new messages from the host computer whereof 'Thread_1' handles the command execution based on the received messages. The script is attached as 'matlabcontrol.script'.

The MATLAB interface is implemented as a common 'tcpip' object which opens a server on the host PC for the robot to connect to. After the robot is connected to the socket the 'tcpip' object can be used as a robot object and passed to the individual functions, eg. 'moverobot(...)', 'OpenGripper(...)' etc.

The MATLAB interface and all functions are simulated using the URSim virtual machine. Instead of programming the robot to connect to the host PC, the simulated robot is programmed to connect to a local IP address of the computer on the network interface shared with the virtual machine, in this case '192.168.174.1'.

All MATLAB files including the robot interface are located in the MATLAB folder.

5 Integration

As a final step the image processing and robot control has to be combined in a single executable MATLAB script to perform the stacking maneuver. The analysis and design of the stacking algorithm is considered in the end of the chapter, resulting in a complete and integrated solution that can be tested in the robot cell.

5.1 Calibration

The image processing results in pixel locations and rotations of the detected bricks. These pixel locations have to be transformed into world coordinates to be integrated with the robot control interface. This procedure is known as calibration and will be separated into two steps: a camera calibration step to find the intrinsic parameters of the camera used and a camera to robot calibration step to find the mapping between normalized coordinates in the camera frame and world positions of the bricks in the base frame of the robot.

5.1.1 Camera calibration

Digital cameras captures the world with a CCD sensor or similar with a lens in front to focus the light beams from the scene onto individual pixel locations. This way of capturing the world with a lens can be modelled with a pin-hole camera model. When working with the pin-hole camera model physical coordinates in the camera frame, x_c, y_c, z_c, are projected into a normalized 2-dimensional coordinate space.

$$p_n = \begin{bmatrix} \frac{x_c}{z_c} & \frac{y_c}{z_c} & 1 \end{bmatrix}^T = \begin{bmatrix} x_n \\ y_n \\ 1 \end{bmatrix} \tag{5.1}$$

Based on tests it has been decided not to correct any distortion as the distortion factor at the operating distance from which the camera will take the picture is less than 1%. Otherwise distortion would have been applied to these normalized pin-hole coordinates. The normalized pin-hole coordinates, x_n, y_n, are transformed into pixel coordinates, u, v using the intrinsic parameters of the camera such as the focal length, principal point and scaling factors. These parameters are contained within a camera calibration matrix, K, which has to be determined.

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = K \begin{bmatrix} x_n \\ y_n \\ 1 \end{bmatrix} = \begin{bmatrix} a_x & 0 & x_0 \\ 0 & a_y & y_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_n \\ y_n \\ 1 \end{bmatrix} \tag{5.2}$$

A camera calibration toolbox made by the Jean-Yves Bouguet from Caltech [4] takes in a sequence of images of a calibration checkerboard with black and white squares of known size. These pictures are then used to calculate both the extrinsic locations of the camera and the intrinsic camera parameters, due to the known physical dimensions of the squares.

The calibration checkerboard is placed in a height from the table matching the height of the bricks. This allows the result of the calibration to be used as a measure of the distance from camera to object, as described in the end of the section.

Figure 5.1: Calibration images taken automatically from different viewing angles

The full camera calibration routine has been automated in the sense that the robot automatically moves to 10 predefined locations and takes a picture. After collecting all images the camera calibration toolbox is automatically used to extract the intrinsic and extrinsic parameters finally stored in the 'Calib_Results.m' file. The automated code is found within the MATLAB script 'Calibration_GeneralCameraCalibration.m'.
The output of the calibration toolbox, found within 'Calib_Results.m' are the parameters needed to form the camera calibration matrix, K. Hereafter the transformation shown in (5.2) can just be reversed to go from pixel coordinates into physical camera frame coordinates. The only requirement here is a known distance/height from the object to the camera centre, z_c.

$$\begin{bmatrix} x_c \\ y_c \\ z_c \end{bmatrix} = z_c K^{-1} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} \tag{5.3}$$

Be aware that this distance is not the height to neither the camera glass or CCD sensor, but rather a measurement of the distance to the pinhole model center.

From the camera camera calibration toolbox the extrinsic parameters, being the translation and rotation of the calibration checkerboard compared to the camera frame, is also determined. As the first image in the sequence of images used to calibrate the camera is taken from the position where the robot will be programmed to take all pictures when detecting bricks, the z-element from the determined extrinsic translation, can be used to define the height from camera center to object. This allows captured brick locations, given in pixel coordinates, to be transformed into camera frame world coordinates, ${}^{C}\mathbf{P}_{\text{brick}}$. This transformation is implemented as part of the function in the script 'PixelToWorldCoordinate.m'.

5.1.2 Camera to robot calibration

For the robot to be able to pick up any of the detected bricks, whose location is now known in world coordinates within the camera frame, it is necessary to convert these coordinates into the base frame of the robot. This requires a transformation between the tool frame and the camera frame to be found.

Assuming that the rotation of the camera is aligned with the both the base frame and tool frame, except for a 90 degree rotation around the z-axis as shown in Figure 2.2, the transformation will only require the determination of a translational offset. This is a reasonable assumption as the tool frame has been programmed to always point downwards when operating and it has been programmed to have 0-rotation around the z-axis when taking pictures of the bricks. As long as the camera is installed carefully, such that the horizontal and vertical alignment matches the tool frame, it is not necessary to determine the rotational part of the transformation.

The complete transformation, ${}^{T}_{C}\mathbf{P}$, from camera frame to tool frame, is hereby described by an unknown translational offset, x_o, y_o, z_o, and a rotation around the z-axis by 90 degree.

$$ {}^{T}_{C}\mathbf{P} = \begin{bmatrix} 0 & -1 & 0 & x_o \\ 1 & 0 & 0 & y_o \\ 0 & 0 & 1 & z_o \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.4} $$

The camera to robot calibration is performed by inserting a LEGO brick into the gripper of the robot and the teach mode of the Universal Robot is enabled. This allows the position of the gripper to be moved on top of a calibration brick.

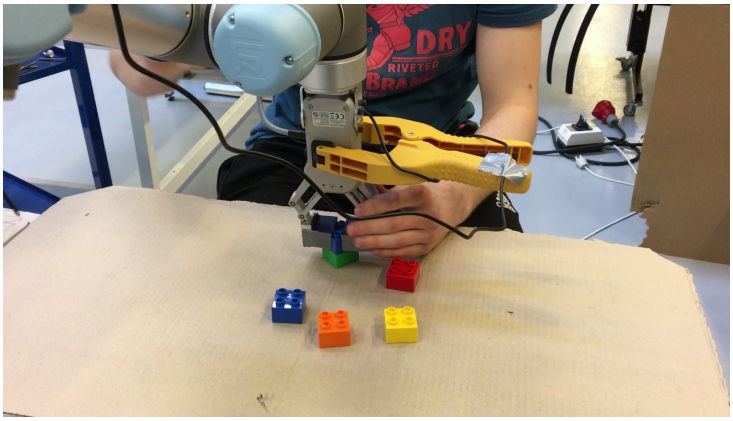


Figure 5.2: Calibration of camera to robot offset

When the gripper has been placed on top of the calibration brick the tool frame position is saved as ${}^{B}_{T}\mathbf{P}_{\text{cal}}$. Then the calibration brick is held firmly while the robot arm moves to the image taking position, ${}^{B}_{T}\mathbf{P}_{\text{image}}$, and take a picture of the brick. This picture is processed by the image processing resulting in a detected brick location which is transformed into world coordinates within the camera frame, ${}^{C}\mathbf{P}_{\text{brick}}$.

The detected brick location within the camera frame is transformed into the base frame by applying the yet unknown transformation, ${}^{T}_{C}\mathbf{P}$, from the camera frame to the tool frame together with the transformation from the tool frame to the base frame in the image taking position. The resulting vector should be equal to calibration position to which the robot was moved manually, that is the translational part of ${}^{B}_{T}\mathbf{P}_{\text{cal}}$ which is extracted using the homogeneous zero-vector, $\vec{0} = \begin{bmatrix} 0 & 0 & 0 & 1 \end{bmatrix}^{T}$.

$$ {}^{B}_{T}\mathbf{P}_{\text{image}} \, {}^{T}_{C}\mathbf{P} \, {}^{C}\mathbf{P}_{\text{brick}} = {}^{B}_{T}\mathbf{P}_{\text{cal}} \, \vec{0} \tag{5.5} $$

The known rotational part of the otherwise unknown transformation to be calibrated, ${}^{T}_{C}\mathbf{P}$ is included as part of the 'PixelToWorldCoordinate.m' script and is applied after the pixel coordinates have been transformed into world coordinates in the camera frame. This allows the transformation, ${}^{T}_{C}\mathbf{P}$, to be split in a known and unknown part, of which the unknown part is denoted as, ${}^{T}_{C'}\mathbf{P}$, and the known rotational part gets included in the calculated brick position in the camera frame now denoted ${}^{C'}\mathbf{P}_{\text{brick}}$.

$$ {}^{T}_{C}\mathbf{P} = {}^{T}_{C'}\mathbf{P} \begin{bmatrix} 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad {}^{C'}\mathbf{P}_{\text{brick}} = \begin{bmatrix} 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} {}^{C}\mathbf{P}_{\text{brick}} \tag{5.6} $$

Using the 'PixelToWorldCoordinate.m' script, which now aligns the rotation of the camera and tool frame in the intermediate frame $\{C'\}$, can then be used to find the translational offset between the two frames, describing the unknown transformation ${}^{T}_{C'}\mathbf{P}$, such that the following equation is fulfilled.

$$ {}^{B}_{T}\mathbf{P}_{\text{cal}}{}^{-1}\ {}^{B}_{T}\mathbf{P}_{\text{image}}\ {}^{T}_{C'}\mathbf{P}\ {}^{C'}\mathbf{P}_{\text{brick}} = \vec{0} \tag{5.7} $$

This translational offset is stored as 'CameraToRobotOffset' in the 'CameraToRobotOffset.mat' file and is applied to the pixel coordinates of every detected brick to determine the brick location in the base frame as found from ${}^{B}_{T}\mathbf{P}_{\text{image}}\ {}^{T}_{C'}\mathbf{P}\ {}^{C'}\mathbf{P}_{\text{brick}}$. Furthermore the z-location of the tool frame in the calibration position is extracted and subtracted one brick height, to be used as a measure of the placement height. This is stored as 'BrickPlaceZ' in the 'CameraToRobotOffset.mat' file.

The implementation of the camera to robot calibration is found within the MATLAB script 'Calibration_CameraToRobot.m'.

5.2 Alignment and gripping issues

In the first tries of the system with the original gripper installed on the robot arm, there were some problems about bricks not being oriented correctly when picked up by the gripper. This error seemed to accumulate while the robot were stacking the bricks, why the robot could not successfully stack all bricks.

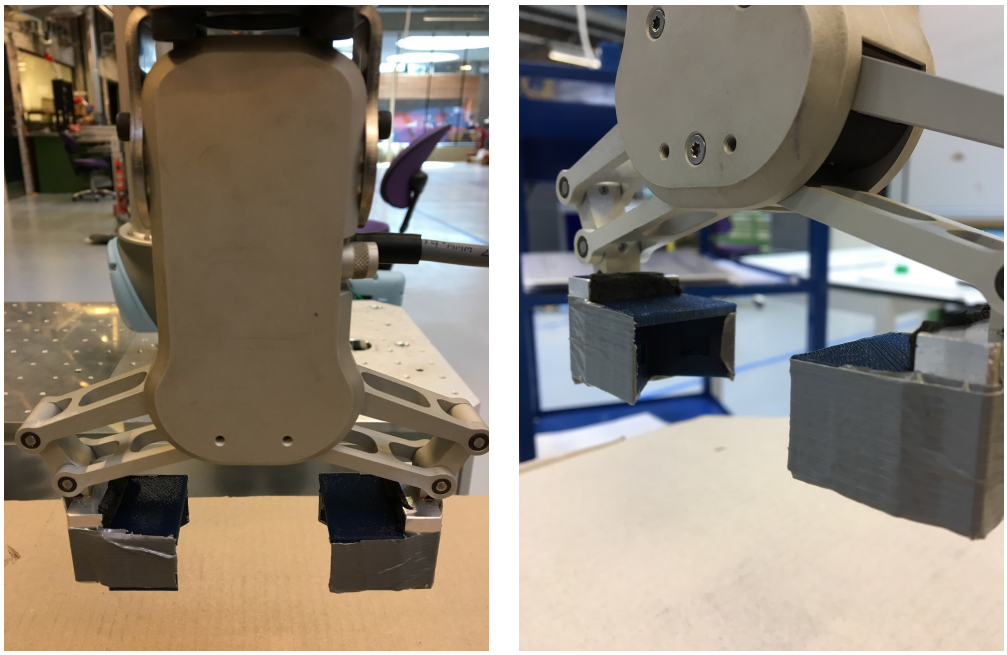


Figure 5.3: Gripper with 3D printed corner-gripper gloves

To solve this issue a fixation tool was 3D modelled, printed, and duct-taped onto the installed gripper. This is shown in Figure 5.3.

5.3 Pickup and stacking order

To save time each ordered figure is stacked completely before being placed aside. This means each figure is stacked from top to bottom by picking up the top brick first. The grippers of the robot will always be holding the top brick and the other bricks will be stacked below this top brick one by one.

It has been decided to stack the bricks in this manner to both save time and also to reduce the likelihood of placing errors. As a stacking or placing plate was not installed on the chosen robot, capable of keeping the position of a placed brick fixed, small placing errors are likely to occur if a brick is put down before a new brick is put on top.

A brick to be picked up is chosen as the first available brick of the necessary color in the brick array from the image processing. When a brick has been picked up it is removed from the brick array hence indicating that it is no longer available. As the bricks in the brick array are numbered and ordered from the top of the image and down, this results in a pickup order where the top-most bricks are used first.

No further optimization has been done of the pickup order, though a likely optimization for future development would be to pick the next brick as the one closest to the current robot position.

5.4 Test and results

As no white bricks were available and the number of different bricks in general were limited due to multiple groups working on the project, it was not possible to implement the actual stacking of the ordered Simpson figures. Instead it has been chosen to stack the bricks in ordered colors: red, green, blue, yellow, orange. If a color is not available it will be skipped and the next possible is chosen. The stacked result is shown in Figure 5.4.

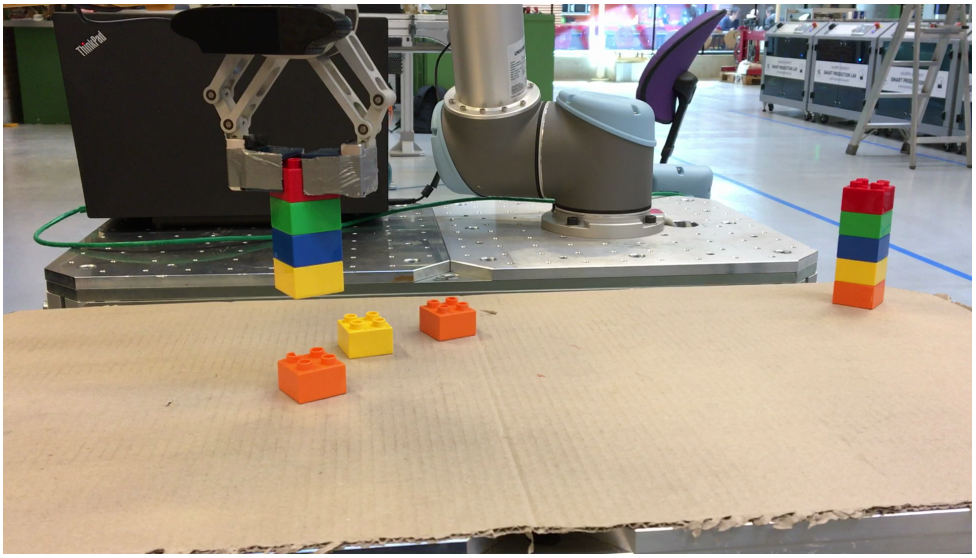


Figure 5.4: Final result stacking bricks in colored order

The final code used in the video demonstration, of which results are mentioned in this chapter, is found within the MATLAB script 'RobotStacker_Final.m'. As it can be seen in the video the system manages to stack the bricks quite well.

Bibliography

[1] Zacobria Universal-Robots community. Universal-Robots Getting Started guide. Last visited 27-03-2017, https://www.zacobria.com/universal-robots-zacobria-forum-hints-tips-how-to/.

[2] John J. Craig. *Introduction to Robotics - Mechanics and Control*. Springer, 2005.

[3] Thomas B. Moeslund. *Introduction to Video and Image Processing*. Springer, 2012.

[4] Jean-Yves Bouguet Ph.D. California Institute of Technology. Camera Calibration Toolbox for Matlab. Last visited 27-03-2017, http://www.vision.caltech.edu/bouguetj/calib_doc/htmls/parameters.html.

[5] RoboDK. RoboDK Simulator. Last visited 27-03-2017, https://www.robodk.com/.

[6] Universal Robots. Manual for The URScript Programming Language. Last visited 27-03-2017, http://www.sysaxes.com/manuels/scriptmanual_en_3.1.pdf.

[7] Universal Robots. PolyScope Manual. Last visited 27-03-2017, http://support.universal-robots.com/foswiki/pub/Downloads/SoftwareUpdates/software_manual_en_Global_1.7beta.pdf.

[8] Universal Robots. URSim simulator virtual machine for Windows. Last visited 27-03-2017, https://www.universal-robots.com/download/?option=27585#section16597.

[9] Aalborg University. M-Tech: Department of Mechanical and Manufacturing Engineering. Last visited 27-03-2017, http://www.m-tech.aau.dk/.