Stavanger, November 1, 2023

University of Stavanger

Faculty of Science and Technology

ELE610 Applied Robot Technology, autumn 2023

ABB robot assignment 5

This assignment was made and tested by Markus Iversflaten and Ole Christian Handegård as a part of their Bachelor thesis spring 2020.

In this assignment you will be working in both Python 3 and RobotStudio (on Norbert). The goal is to identify the position of several, randomly placed pucks in the work area by capturing one (ore several) image(s) using Python, and then telling RobotWare where the pucks are, so they may be picked and placed by the RAPID program. For this, you will need:

- Python 3.x ($x \geq 6$)
- RobotStudio
- Norbert, with gripper and camera mounted on gripper

Approval of the assignment can be achieved by demonstrating the robot program to the teacher, and then submit a report including Python code on *canvas*.

This assignment is demanding if you insist on making the solution perfect, especially the last part may be difficult. It may also be difficult to get time on the robot when you want it and as much time as you want. It is good to plan your work well on the robot, start by reading all of the assignment. Reserve a two hours slot on Norbert, and use this time effective to capture images from different known positions of camera mounted on robot tool. The scene should be the table with pucks where readable QR-codes are attached. The pucks should be placed in both known, and unknown positions. You will need a RAPID program to move robot tool with the camera to the different positions above the table, this program can be made ready before you go to the laboratory.

When you have a set of many images you can do a lot of work with Python only, not in Robot laboratory. You can derive the vision theory, and then

Karl Skretting, Department of Electrical Engineering and Computer Science (IDE), University of Stavanger (UiS), N-4036 Stavanger. E-mail: karl.skretting@uis.no.

find the transformation matrix from the images where pucks are in known position. The matrix can be tested on some more images where pucks are in known position, and finally also test on some images where the pucks are in unknown positions and check if the findings makes sense.

You should reserve another two hours slot for testing the communication between Python and PC where Python is running. When testing it will be useful to let Python write the position of the loacated puck(s) in the output screen, and to let RAPID write the same positions on the screen on the FlexPendant using `TPWRITE`. Hopefully you will make communication work within this time slot, or if needed reserve and use more time slots. When you understand and master the communication, you can prepare for the final test and make both the Python program and the RAPID program as ready as you can from somewhere not on the robot laboratory, for example on your desk in room E462 or E464. Finally you will need one, ore more, time slots on Norbert again to test and debug and adapt the final solution to work well.

The time limit for this assignment is 40 hours.

5 Control robot from Python

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

5.1 Robot Web Services

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

5.2 Introduction

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

- The Pack and Go file, `UiS_E458_nov18.rspag` ↗.
- `PythonCom.mod` ↗

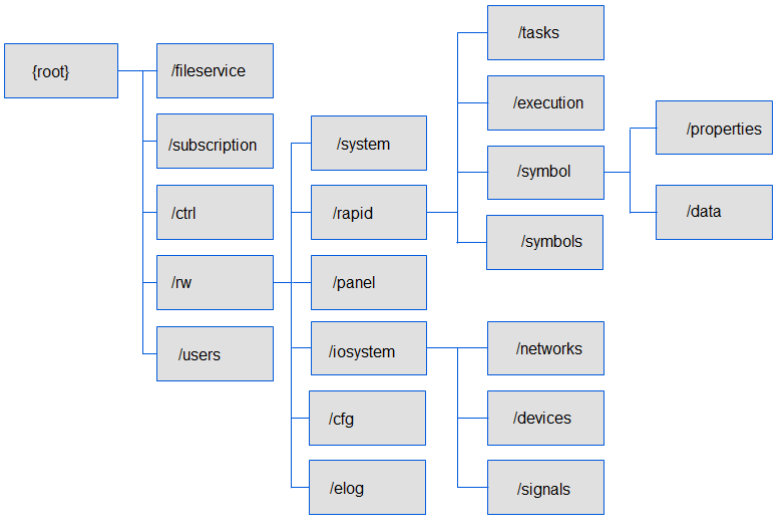

Figure 1: The resource hierarchy in Robot Web Services, retrieved from RobotStudio RWS web page ↗.

- Communication package `rwsuis` ↗
  More information in Bachelor-thesis spring 2020 by Markus H. Iversflaten and Ole Christian Handegård, (in Norwegian)

5.3 Establish connection

Now, you should test communication between Python and RobotWare. This can be done in several ways, but the easiest may be to poll the controller state, which means to check whether the robot's motors are turned **on** or **off**. Your computer should now be connected to the network `E459abb`.

5.3.1 Making a GET request

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

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

Second: Get the right address for the HTTP request. To access Norbert's controller, you will need its IP address, as it is on the local net it is `http://152.94.0.38`.

More detailed explanation:
You can find this IP address by checking the available controllers in RobotStudio. To access the hierarchy of resources (figure 1) for example to poll the controller state, you will need to find the location of the controller state resource. It is located in **RobotWare Services** $\longrightarrow$ **Panel Service** $\longrightarrow$ **Operations on Controller State Resource** $\longrightarrow$ **Get Controller State**. The Sample Call for this resource gives us an idea of how to make the HTTP request. The sample call here is (from OS prompt):

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

`curl` ↗ stands for Client URL (Uniform Resource Locator). The complete address you need is the last part above. When working on a physical robot, `"localhost"` needs to be changed to the IP address of the robot, which you have already found. The address for the controller state will then be:

```
cState = "http://152.94.0.38/rw/panel/ctrlstate"
```

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

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

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

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

To check if the response was successful, you'll need to print the response to the console in Python. This will yield an HTTP status code as text (in brackets), the code can also be returned as an integer: `i = response.status_code`. If the status code is 2xx, then the request was successful, and a connection was established. If the status code is anything else, then something in your request might be wrong.

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

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

This will yield an XML formatted message[1] (which may seem a bit excessive) and somewhere in there, you should find either `guardstop` or `motoron` or `motoroff`. You may also get (store) the message as a string:
`s = response.text`.

[1]The Python standard package `ElementTree` can be used to parse the XML text

It is also possible to receive messages in JSON. This can be done by adding a query string ("json=1") at the end of the resource address:

```
response = requests.get(cState+"?json=1", auth=auth)
```

In principle, this is how all GET requests work. Going forward, you may use our package `rwsuis` ↗, which includes the RWS class. This class has predefined functions for all the requests you will need during this assignment. All the requests in the class pick out only the information needed; e.g. polling the controller state through the RWS class would yield *only* `motoron` or `motoroff`. Check out the documentation on this ABB web page ↗.

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

5.3.2 Making a POST request

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

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

```
VAR robtarget simple_robtarget :=
    [[0,0,100],[0,1,0,0],[-1,0,0,0],[9E+09,9E+09,9E+09,9E+09,9E
    +09,9E+09]];

PROC main()
     WHILE TRUE DO
          MoveJ simple_robtarget, v200, z10, tGripper\WObj:=
    wobjTableN;
     ENDWHILE
ENDPROC
```

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

```
from rwsuis import RWS

```

```
# robot = RWS.RWS("{Robot's IP}")  # as below
robot = RWS.RWS("http://152.94.0.38")  # or https ??
```

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

Requesting mastership in manual mode:

```
robot.request_rmmp()  # Mastership request in manual mode
time.sleep(10)  # Give time to accept manual request
```

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

Requesting mastership in automatic mode:

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

In automatic mode, mastership is automatically given when requested.

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

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

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

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

5.4 Scanning QR codes

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

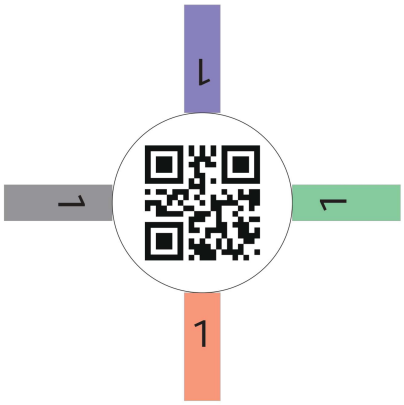

Figure 2: QR code taped on pucks

5.4.1 Capturing an image

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

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

```
import cv2

cap = cv2.VideoCapture(1)

# Change resolution to 1280x960
cap.set(3, 1280)
cap.set(4, 960)

ret, frame = cap.read()
```

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

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

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

5.4.2 Image processing

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

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

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

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

5.4.3 Installing a QR scanner

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

5.4.4 Using the QR scanner

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

```
from pyzbar.pyzbar import decode
...
data = decode(image)
```

The object will look like this:

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

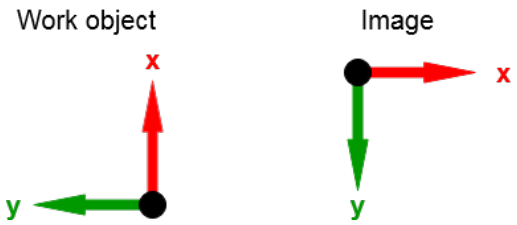

Figure 3: Coordinate systems for work object and image

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

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

5.5 Creating a robtarget

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

First:
Images in Python are usually represented in matrix form. This means that x increases as you go to the right and y increases as you go downwards, as illustrated in figure 3 below. It also means that the origin (0,0) is in the top left, not in the middle as you might be used to. The work object, however, has origin in the middle.

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

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

You must then transform the axes. For figure 3, the transformation would look like this: $\mathbf{x} \longrightarrow \mathbf{y}$, $\mathbf{y} \longrightarrow \mathbf{x}$, $\mathbf{x} \longrightarrow -\mathbf{x}$, $\mathbf{y} \longrightarrow -\mathbf{y}$.

Second:
You must consider the units used in the image taken and the units used by the robot. The image uses pixels as the measuring unit, whereas the robot uses millimeters. A conversion from pixels to millimeters is therefore required.

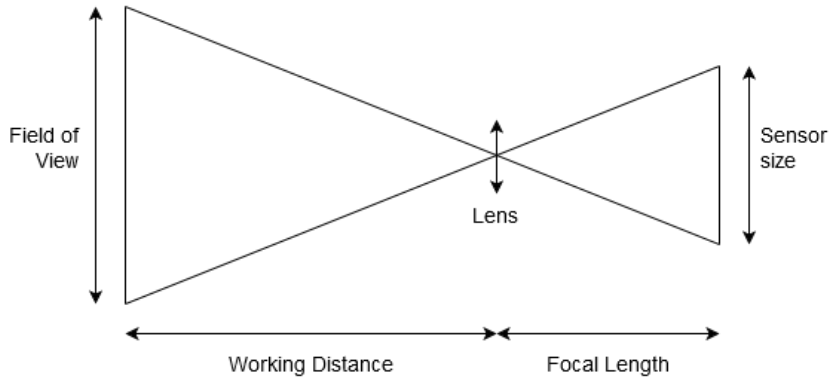

Figure 4: Relation between field of view and sensor size

Finding this conversion requires knowledge of the camera's FOV[2]. The FOV is found by relating it to the working distance and the camera's physical properties. Figure 4 shows this relation.

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

Focal length[3]= 3.7mm
Sensor width = 3.6288mm

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

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

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

$$working\_distance = 400 + 70 - 30 = 440mm$$

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

[2]Field of View

[3]This is a simplification. The focal length also depends on the amount of zoom applied by the lens. This gives a potential error of ±5%. A more correct calculation can be performed here: IDS XS FOV calculator ↗.

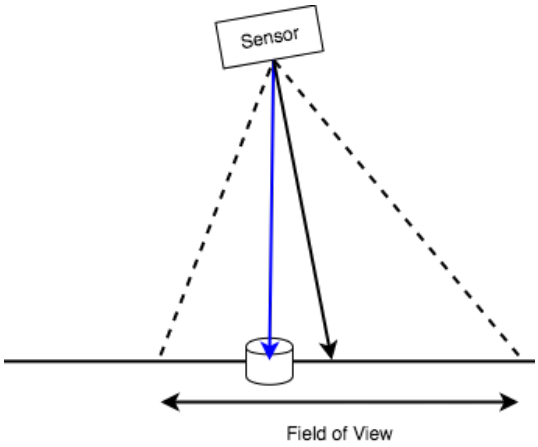

Figure 5: Inaccuracy in camera and mount

$$\frac{FOV\_width}{working\_distance} = \frac{sensor\_width}{focal\_length} \tag{1}$$

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

$$FOV\_width = \frac{sensor\_width}{focal\_length} \cdot working\_distance$$

$$pixels\_to\_millimeters = \frac{FOV\_width}{resolution\_width}$$

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

Third:
The position you've now created through the transformation and conversion is in relation to the camera. To actually pick up a puck, you will have to relate the position to the gripper. As previously mentioned, you will capture an image with the camera in front of the gripper, without any rotation. This means that the camera will be 55mm in front of the gripper. You must relate this to your coordinate system (for figure 3 this would be 55mm in the positive x-direction).

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

Most likely, the mount and USB cable together tilt the camera slightly. In addition, the camera itself has an inaccuracy in the lens, which can provide

further tilt. Figure 5 displays this problem. In some way, the angle of the camera must be found, so it may be compensated for when creating the robtarget.

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

$$comp_x = slope_x \cdot working\_distance$$
$$comp_y = slope_y \cdot working\_distance$$

Let's go through an example!
Used parameters:

- Gripper height = 500mm
- Camera resolution = 1280x960
- Coordinate systems such as in figure 3

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

First, make the center of the image the origin:
$(x, y) = (100 - (1280/2), 150 - (960/2)) = (-540, -330)$

Next, apply the appropriate transformation. With coordinate systems such as in figure 3, the transformation yields a new puck position: $(x, y) = (330, 540)$.

Now, convert the position to millimeters:

$$FOV\_width = \frac{sensor\_width}{focal\_length} \cdot working\_distance = \frac{3.6288}{3.7} \cdot (500+70-30) = 529.6$$

$$pixels\_to\_millimeters = \frac{FOV\_width}{resolution\_width} = \frac{529.6}{1280} = 0.414$$

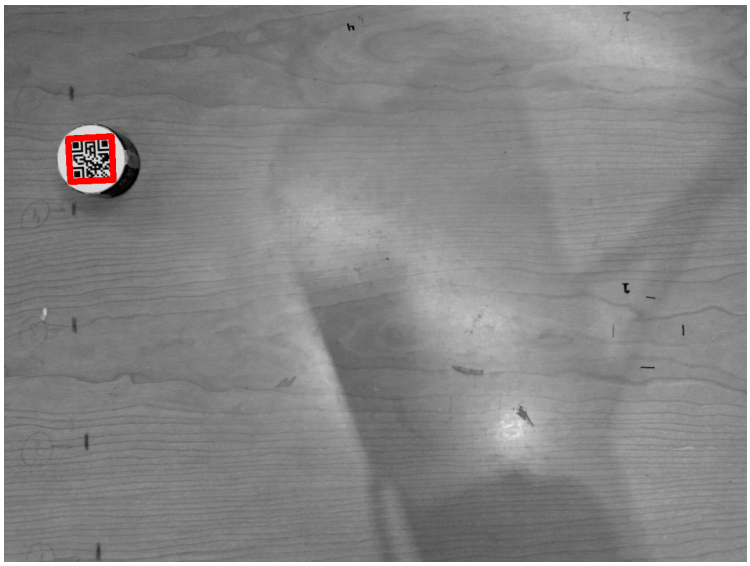

Figure 6: Image of scanned puck

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

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

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

Recall that the camera is mounted 55mm in front of the gripper. In addition, the gripper's position is needed. In this case, it is simply at the origin. In other cases, though, it will differ. Therefore it is best to make a request to RobotWare for the gripper position. This can be done easily through the package:

```
from rwsuis import RWS

norbert = RWS.RWS("http ://152.94.0.38")
gripper_position = norbert.get_gripper_position()

camera_position = [gripper_position[0] + 55, gripper_position
    [1]]
```

```
>>> camera_position
[55,0]
```

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

This position will be *nearly* accurate enough to be used to pick up pucks.

However, as previously mentioned, the camera is likely tilted in some fashion. The tilt is compensated for by using the slope values found through the camera adjustment routine. In this case, the values were:
$slope_x = 0.0229$
$slope_y = -0.0003$

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

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

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

These compensation values must be **subtracted** from the puck position:
Puck position: $(x, y) = (191.6 - 12.366, 223.6 - (-0.162)) =$**(179.2, 223.8)**.

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

5.6 Program structure

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

For example, take two arbitrary script running in parallel: script 1 & script 2, with flag variables 1 & 2, respectively.
When a certain part of a script 1 is finished, flag variable 1 should be set to "TRUE". Once this happens, script 1 should enter a waiting loop, waiting for script 2 to finish whatever it's doing. As flag variable 1 became TRUE, script 2 can continue its execution. Before this happens, though, the flag variable **must be reset** (be set to "FALSE"). Script 2 now continues its execution. Once finished with a certain code block, flag variable 2 should be set to TRUE, and so on...

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

Predefined methods:

```python
# Wait method used in Python
def wait_for_rapid(self, var='ready_flag'):
    """Waits for robot to complete RAPID instructions
    until boolean variable in RAPID is set to 'TRUE'.
    Default variable name is 'ready_flag', but others may be used.
    """

    while self.get_rapid_variable(var) == "FALSE" and self.is_running():
        time.sleep(0.1)
    self.set_rapid_variable(var, "FALSE")
```

```
! Wait method used in RAPID
PROC wait_for_python()
    ! Wait for Python to finish processing image
    WHILE NOT image_processed DO
    ENDWHILE
    image_processed:=FALSE;
ENDPROC
```

5.7 Hints and test of programs

Some help and idea for how a complete test program may be can be found in this short program ↗. Unfortunately, I guess there are some minor(?) errors in this code which need to be corrected: import from `rwsuis`, and use of `RWS` should be as on page 13, how `get_xyx()` is used seems strange, and perhaps some more issues. Make sure to install all dependencies, i.e. the packages that are imported in the project files. When the test programs works as intended you should place a puck on the table. When the program is run, the robot should find the puck and position its gripper directly above the puck, but not try to grip it. If this succeeds, you have made a good step towards the final solution.

5.8 Creating a working program

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

Notes:
You've previously created procedures in RAPID for both picking and placing pucks. These may be reused, with a slight change to picking: The camera should now be approximately centered over the puck before the gripper goes

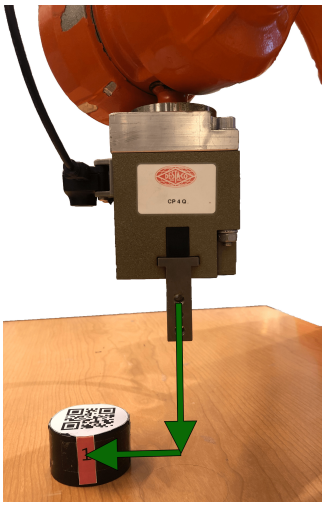

Figure 7: Gripping technique

down to gripping height (10mm above work object). The gripper should then "slide" in toward the puck before gripping it. This technique can be seen in the video made in the 2020 bachelor thesis. This change prevents the puck and QR codes from getting damaged, should the gripper miss its target.

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

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

The four colors down the sides of each QR code (see figure 2) can be used to visualize the puck's rotation. By stacking pucks with the same orientation, these colors will line up nicely. This is also shown in the video ↗. It is possible to find the pucks' orientations through the QR codes. For this, you **must** download and use a `tweaked pyzbar package` ↗, while also uninstalling the previously used pyzbar package. This package makes the corner positions of the QR codes appear in the same order regardless of their orientation. For figure 8, corners 1 & 4 were used to find the puck's orientation.

5.9 Test program

One way to test the program is to place three to five pucks randomly on the desk. Then run the programs, both RAPID program on robot and python-

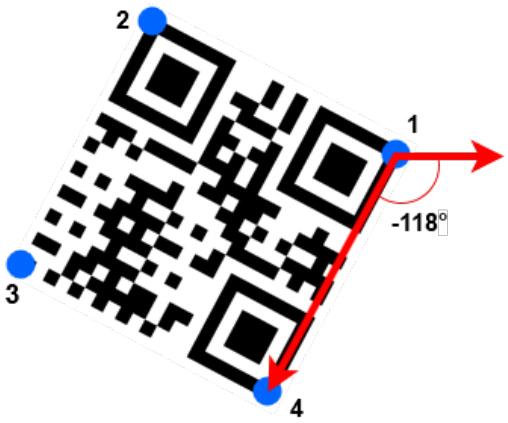

Figure 8: QR code orientation

program on PC. The robot should now locate all puck, pick them one by one and place them in a stack on the center of the table.

To add more demonstration effect to the program it can be extended to also spread the pucks from a centered stack to random positions, that is not to close to each other, on the desk. Then the program should forget where the pucks are, before it starts again from the beginning.