Department of Informatics
Chair of Robotics, Artificial Intelligence and Real-time Systems

Computer Vision 2D

Carlo Cagnetta

Marc Hauck

Prashanth Kumar H. R.

Practical Course *Modular Robotics* WS 2021/22

Advisors:	Max Körsten, Jan Ueckmann
Supervisor:	Prof. Dr.-Ing Matthias Althoff
Submission:	17. March 2022

Computer Vision 2D

Carlo Cagnetta
Technical University of Munich
Email: carlo.cagnetta@tum.de

Marc Hauck
Technical University of Munich
Email: ge65qoy@mytum.de

Prashanth Kumar H.R.
Technical University of Munich
Email: prashanth.kumar@tum.de

***Abstract*—In this work, we introduce a 2D computer vision solution that enables a modular industrial robot to detect and grasp specific parts inside its workspace. First, we evaluate different algorithms suited for the use of a given 2D grayscale camera. After choosing a simple and efficient algorithm that is based on a set of assumptions, its consecutive steps and its integration into the robot control and web-interface will be explained in detail. In addition, we also perform associated tasks that were necessary to implement the solution, e.g. the camera calibration and the mechanical design of the camera mounting and gripper. Finally, we validate our approach by successfully implementing a vision-based pick-and-place application.**

I. Introduction

The goal of this project that was part of the practical course "Modular Robotics " was to develop an object detection algorithm, which enables a modular manipulator to detect and grasp an object lying its workspace.

The scope was to create a vision solution that is able to detect and locate parts and that enables the robot to interact with them. Given the rather frequent variation of production environments in which robotic manipulators are used, it was required that the end-user could set up the part detection to flexibly handle various objects in different environments. Moreover, as the robot can already be operated via a web-based user interface based on Vue.JS and NodeRED, the vision solution should be integrated into the same web-interface.

In order to achieve the objective, the group was provided a 2D grayscale camera manufactured by the company Basler shown in fig. 1, together with an infrared and daylight-cutting filter. Besides, the already existing web-interface based on Vue.JS and NodeRED had to be used for integrating the algorithm into the robot control system, utilizing the already existing communication and motion control units.

For the result of the project to be successful, it was required to detect and pick a sample object: a metal bracket shown in fig. 2. In order to achieve this goal, the following tasks were defined: First a new end of arm was designed that could integrate camera and end effector; then an object detection algorithm was developed that could detect the object to be picked as well as transform the 2D pixel coordinate of its center into a 3D real-world coordinate; finally the result was input into the web-interface to move the robot and pick the object.

This report is going to present the work that was done by the group "Computer Vision 2D" throughout the semester in order to accomplish the defined tasks and achieve the required goal. First, different methods to perform object detection will be presented in section II. In section III we will explain the necessary steps to integrate the provided camera into our workflow. At first the choice of the mechanical design for the end of arm will be justified; then we will cover how to perform camera calibration to get undistorted images and perspective calibration to relate 2D image coordinates to 3D real-world coordinates. After that, in section IV, we will discuss the methods of the implemented solution. After setting some underlying assumptions, we will describe the development steps of the object detection and position estimation algorithms. Finally, it will be illustrated how the camera control is integrated into the robot control flow and how everything is combined as a single pipeline into NodeRED. While section V is going to show the results of our solution, these will finally be discussed in section VI, which will also present some possible future work.

Fig. 1. Provided camera: Basler a2A3840-13gmPRO GigE Vision Camera.

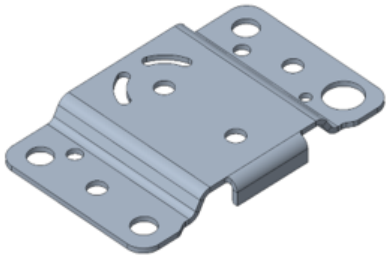

Fig. 2. 3D model of the metal bracket used as sample object.

II. Related Work

Object detection is a very important task in robotics, as it allows a robot to grasp objects without teaching or hard-coding their positions, enabling it to work in more complex and dynamic environments. For robots acting in non-industrial environments, an object recognition system is often obligatory. Even in industrial robotics, where many robots do not need visual perception because of repetitive hard-coded sequences, e.g. in a car production line, there is an increasing demand for robots that can grasp unknown and unordered objects or which can be used flexibly and therefore need visual perception.

As the use of a 2D gray-scale camera was mandatory for the scope of our project, this work focuses on object detection from an input of 2D scene images. The goal of the respective algorithms is to detect and localize objects inside the 2D input image and to transform their 2D locations from pixel coordinates to 3D real-world coordinates. As no spatial information about the scene is available in contrast to 3D methods, object detection from 2D images is a very challenging task. To find an algorithm that is well-suited for the task at hand as well as the scope of our project, we took several algorithms into consideration that can detect objects in 2D images in various different ways:

An early and well-known approach to detect objects in 2D images is presented in [1]. It makes use of the so-called Scale Invariant Feature-Transform (SIFT), a method for image feature generation. To be able to detect objects with the help of image features, all detectable objects have to be available in a 2D image dataset. With these model images, the SIFT features of the objects can be generated offline. Once a scene image has been taken, the features of the model images can then be compared to the features of the scene image, which are generated online. This can be done by using a matching algorithm like FLANN [2]. If enough matching features for an object are found, it can be detected and localized inside the scene image by finding its homography matrix [3], i.e. the transformation matrix from the model image to the scene image up to a scaling factor. Although this approach is easy to implement, SIFT features are prone to changes in viewpoint and illumination. As shown in fig. 3, we were not able to achieve reproducible results with the algorithms provided by OpenCV for metal objects similar to our sample object. While OpenCV suggests the use of a faster and more robust successor of SIFT, Speeded-Up Robust Features (SURF) [4], for feature detection [5], the SURF algorithm is not freely available. To be able to use the code provided by OpenCV, we had to change the function calls for feature detection to make use of SIFT instead of SURF, which could be one reason for the bad performance we observed.

The approach introduced in [6] can detect a wide range of unknown objects in 2D images. In addition to that, the pose and 3D position of the objects can be estimated robustly. To do this, this algorithm makes use of the objectron dataset [7]. In this dataset, many different objects are represented as an annotated video, depicting them inside a 3D bounding box

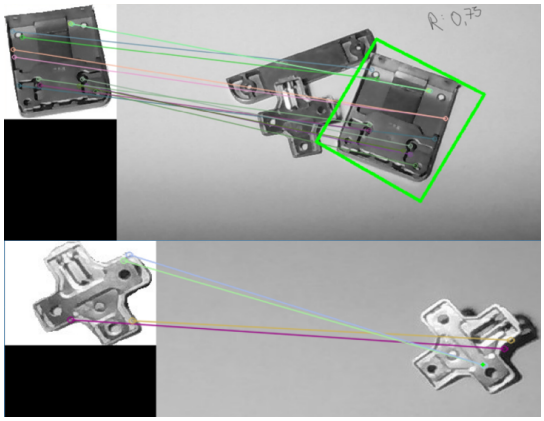

Fig. 3. Test runs of Homography Feature Matching using SIFT [5] and FLANN [2]. Top: successful run with the bounding box indicating the position of the object. Bottom: Failure because of a change in lighting between model and scene image.

from different viewpoints as shown in fig. 4. The underlying algorithm then uses a convolutional neural network (CNN) to recognize and localize the same items online. While the approach shows very good results for many categories of unknown objects that are part of the objectron dataset (e.g. shoes, cups, boxes), it is not able to detect new ones that are not represented in the dataset. Thus, to detect items that were not yet part of the dataset, it would have been required to add them manually and retrain the CNN used for object detection. As this would have been cumbersome without a guarantee for success, this approach was not investigated further in this work.

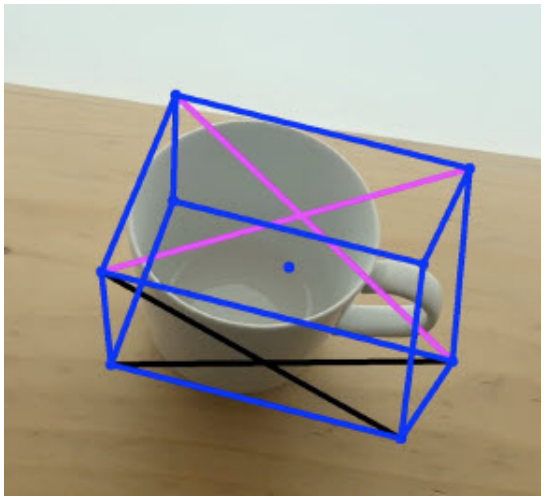

Fig. 4. Example frame of an annotated video that is part of the objectron dataset [7].

Another powerful framework for object detection called DeepIM is presented in [8]. This approach also uses a deep neural network to process the image data and is able to estimate the 3D position and pose of an object from a single

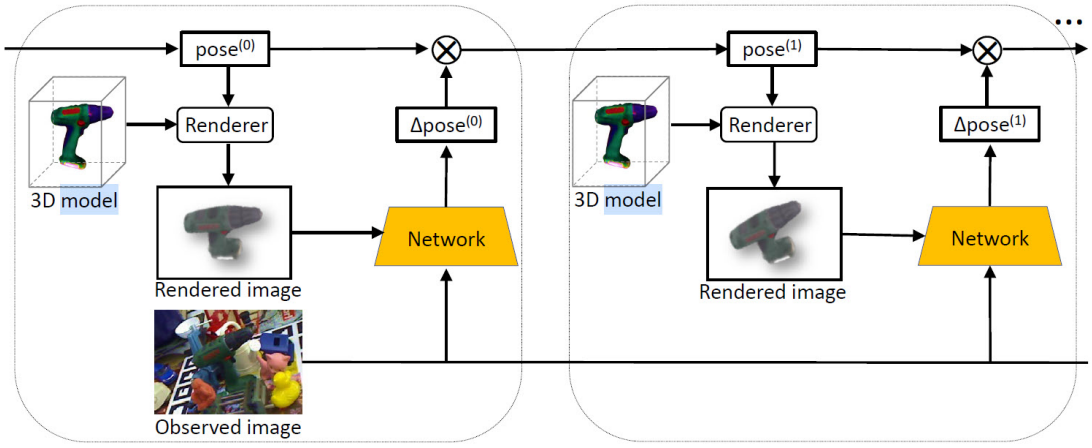

Fig. 5. Method of the DeepIM-framework: Iteratively, a renderer generates a 2D image from a 3D model and compares it to the observed image by using a neural network. [8]

2D image. To do this, this method makes use of a model database where the objects are represented as realistically colored 3D models. These models are fed to a renderer that generates a realistic 2D image of the desired object. The 2D image is then compared to the observed scene image by the neural network. The algorithm outputs a pose estimation that serves as a new input for the renderer. These steps are repeated iteratively to refine the pose estimation in each step and find the pose that creates the greatest similarity between rendered and observed image regarding the desired object. The complete method is shown in fig. 5. The framework can even detect formerly unknown objects as long as they are available as a 3D model, i.e. the network generalizes on unseen data and does not have to be retrained if new objects are introduced. Although this approach is very promising, it was trained with RGB-images as an input, which makes it unsuitable for the use of a grayscale camera. While it should be possible to change the network architecture and retrain it with grayscale images, this would have exceeded the scope of this work.

A simple, yet efficient computer vision solution for 2D images is introduced in [9]. To detect objects, this approach only needs an image of the background instead of models for each object. This background image is then subtracted from the scene image, only leaving areas visible that are different from the background, i.e. that contain the objects to be detected. By further image processing, the pixel coordinates of the object centers can be determined and converted to 3D real-world coordinates. Because of its simplicity, we implemented an object detection algorithm very close to the one presented in [9]. The algorithm and its limitations will therefore be explained in detail in section IV. In addition, associated tasks like necessary calibration steps will be shown in section III.

III. Camera Integration

A. Mechanical design

The main mechanical task of the project was to design both a camera mounting and a gripper for the robot. Regarding the camera, it was either possible to mount it at an external position, where it could oversee most of the workspace, or directly onto the robot arm, right next to the end effector.

Mounting the camera at an external position has the advantage that the camera position is fixed, i.e. it is not part of the kinematic chain and its position does not have to be computed repeatedly. Moreover, as various standard grippers for the robot had already been available, this would have made the gripper design needless. Nevertheless, this also makes the camera application less flexible and depending on the its position, the robot could occlude its own workspace from the viewpoint of the camera.

On the other hand, mounting the camera directly onto the end of arm makes its installation and commissioning more complicated. Its position is crucial for the application as it determines the field of view, but also possibly limits the robot's ability to grasp or move in a specific way if it greatly enlarges the end of arm. In addition, the camera should be mounted close to the gripper, so the robot does not have to alternate between a pose for taking images and a pose for grasping the detected objects. But it should neither be mounted too close, as the gripper would then be inside the camera's field of view, possibly occluding objects of interest in the workspace. As the robot's mechanics, its kinematic and control will always influence the camera position and orientation, inaccuracies that arise in these fields also have to be taken into account. However, in contrast to mounting the camera to a fixed position, the use of it is much more flexible: If an object is occluded or out of the camera's field of view, the camera can be moved to a desired position inside the robot's workspace and take the image from this new position. Furthermore, it even allows to take two or more images of a scene, making it possible to gain depth information.

As flexibility was a key argument for us, we decided to mount the camera to the end of arm. This also implied that we had to design a part which could carry the camera as well as a gripper. For the sake of simplicity we decided to use a suction

cup gripper, which was very suitable to grasp our sample part. To operate it, a vacuum ejector was needed at the end of arm, which implies that the end of arm had to contain mounting holes for it, in addition to those for the camera mounting. The suction cup itself could then be directly mounted to the vacuum ejector using standardized components. Finally, besides carrying the camera and the vacuum ejector together with the gripper, the end of arm had to meet the mechanical requirements of the robot flange and the obligatory connection clamps, including fitting holes for two positioning pins.

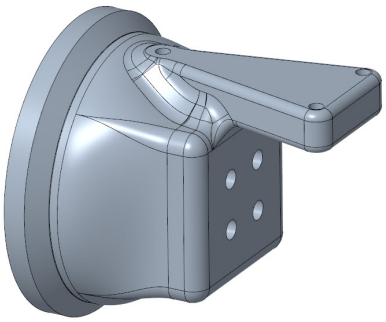

Fig. 6. Isometric view of the end of arm. Top: camera mounting. Right: Vacuum ejector mounting. Left: connection to robot flange.

The part was designed in a way that it could be 3D-printed using fused deposition modeling (FDM) with as few support structures as possible to reduce printing time and material consumption. As FDM-parts shrinks after printing, all mounting and fitting holes had to be re-drilled after printing using a bench-drill.

B. Camera Configuration

The next requirement for integrating the camera into our solution was to be able to control it from the robot control flow. In order to automate the image acquisition, we created a python script which can control the camera and take pictures automatically. The camera control libraries for the Basler camera in use were provided by an open source GitHub repository [10] that is maintained by the camera company itself.

The Basler camera that was used in the project is controlled via an Ethernet connection, which is also responsible for the data transmission from the camera to the computer. Since the Ethernet ports of most laptops are mainly configured for a simple internet connection, a big issue was to configure the Ethernet communication in such a way that big data packages containing the images could be transmitted reliably. First, it was necessary to configure the Ethernet port of the laptop following the instructions for the network adapter properties under [11]. Once the computer connection had been adjusted, it was necessary to optimize the camera configurations in order to get the best image possible. This was done using the Basler camera software called "Pylon Viewer", and using the option of "Automatic Image Adjustment". Moreover, the packet size and the inter-packet delay of the transmitted data had to be tuned so that the images could be sent without missing information. Figure 7 shows an example of two sample images: the one on the left has been taken before tuning the data transmission parameters, and the right one is a tuned image, in which almost no information is missing. To perform object detection, it is necessary that the images are error-free.

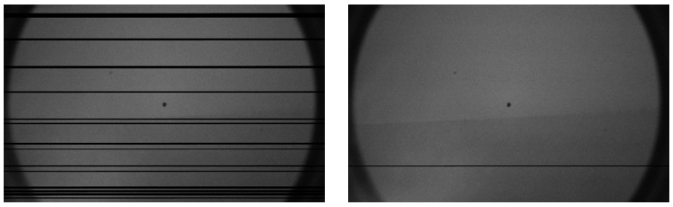

Fig. 7. Left: Image of the background before data transmission parameters tuning, shows many black lines corresponding to missing information. Right: Image of the background after tuning.

To make sure that the camera can be operated under the same conditions each time, a configuration file containing all the optimized parameters is loaded at the beginning of the camera control script. Moreover, the grabbing function to get a picture is called multiple times until a good picture without missing information is successfully grabbed. This way, it is assured that the program always gets the best possible image.

C. Camera Calibration

Before using the camera to capture images, it is necessary to calibrate it and find its intrinsic and extrinsic parameters, such as the focal point (f_x, f_y) and the principal point (c_x, c_y), which are part of the 3x3 *camera matrix (A)* shown in eq. (1).

$$A = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{1}$$

Moreover, some pinhole cameras introduce significant distortion to images. Two major kinds of distortion are radial distortion and tangential distortion. Radial distortion causes straight lines to appear curved, and can be represented as follows [12]:

$$\begin{aligned} x_{\text{distorted}} &= x\left(1 + k_1 r^2 + k_2 r^4 + k_3 r^6\right) \\ y_{\text{distorted}} &= y\left(1 + k_1 r^2 + k_2 r^4 + k_3 r^6\right) \end{aligned} \tag{2}$$

Here r is the distance to an arbitrary center of distortion c and x, y are the real image coordinates of the distorted point, k_1, k_2, k_3 are the radial distortion coefficients.

Tangential distortion happens because the lens of the camera might not be aligned perfectly with the image plane, so some areas might appear closer than other. This is expressed by the following [12]:

$$\begin{aligned} x_{\text{distorted}} &= x + \left[2p_1 xy + p_2\left(r^2 + 2x^2\right)\right] \\ y_{\text{distorted}} &= y + \left[p_1\left(r^2 + 2y^2\right) + 2p_2 xy\right] \end{aligned} \tag{3}$$

Where p_1, p_2 are the tangential distortion coefficients.

These distortions have to be removed from the captured images. To do this, the distortion coefficients of eq. (2) and eq. (3) have to be found:

$$\text{Distortion coefficients} = \begin{pmatrix} k_1 & k_2 & p_1 & p_2 & k_3 \end{pmatrix} \quad (4)$$

The camera calibration algorithm we implemented uses a collection of images as input, with points whose 2D image coordinates and 3D world coordinates are known. After computation, it outputs the camera intrinsic matrix A as well as the rotation and translation of each image relative to the principal points and the distortion coefficients.

In detail, this is done by taking multiple images of a chessboard pattern with known dimensions at many different orientations. An example is shown in fig. 8. Choosing a world coordinate system that is attached to the chessboard, the corners of the chessboard have known 3D world coordinates: since all the corner points lie on a plane, we can arbitrarily choose $Z_w = 0$ for every point, and since the points are equally spaced on the chessboard, the (X_w, Y_w)-coordinates of each 3D point can easily be defined as (0,0), (1,0), (2,0), etc.

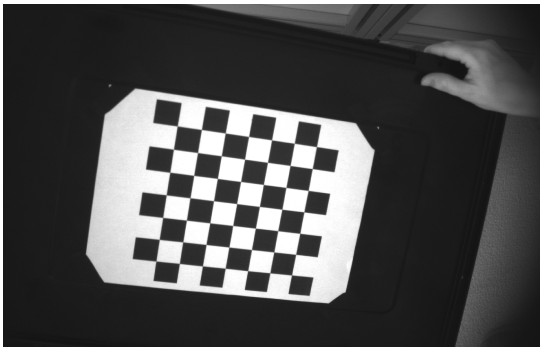

Fig. 8. Sample chessboard image taken during camera calibration.

Next, we need to find the 2D pixel locations of these chessboard corner points. OpenCV provides a built-in function called *cv.findChessboardCorners()* [12] that looks for a chessboard and returns the coordinates of the corners as shown in fig. 9. As final step of calibration we pass these 3D world points and their corresponding 2D pixel locations to OpenCV's *cv.calibrateCamera()* [12] method. This method outputs a camera matrix A and the distortion coefficients, which are saved locally. Since matrix A is used to project the world points onto the image plane, its inverse $\mathbb{A}^{-1}$ can be used to convert back from image points to world points.

D. Perspective Calibration

Perspective calibration uses a different calibration pattern, shown in fig. 10, and a fixed camera position. Its goal is to find the extrinsic parameters of the camera position (later referred to as pre-calibrated position, see section IV-A) and the *scaling factor s)*, which expresses the depth between the camera center and the object plane.

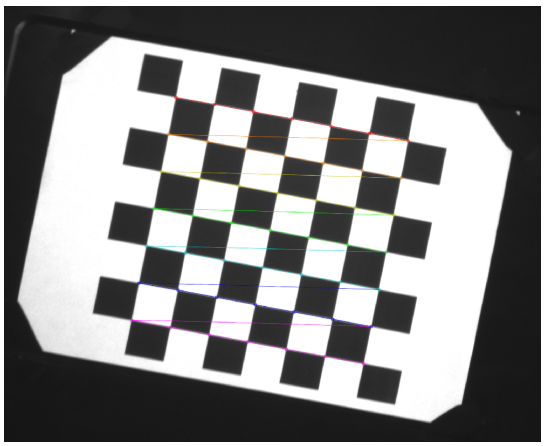

Fig. 9. Undistorted chessboard image with detected corner points.

The calibration pattern has 9 circles. The 3D points (X_i, Y_i, Z_i) of each of these circles are measured manually from the robot frame of reference. Then, the 2D pixel coordinates (u_i, v_i) of the center of each circle are identified. Finally the 3D points and their corresponding 2D pixel coordinates are input into OpenCV's *cv.solvePnP()* [13] method. This method outputs the rotation matrix R^{3x3} , and the translation vector t^{3x1} from the camera reference frame to that of the object plane which corresponds to the center pixel of the image.

$$\left(s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} A^{-1} - t \right) R^{-1} = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \quad (5)$$

Lastly, substituting the above calibrated values into eq. 5, we are able to calculate s, the scaling factor from image plane to object plane. Here A is the camera matrix, t is the translation vector, R the corresponds to the rotation matrix, (u, v) are the 2D image coordinates of the points and (X, Y, Z) are the 3D object coordinates.

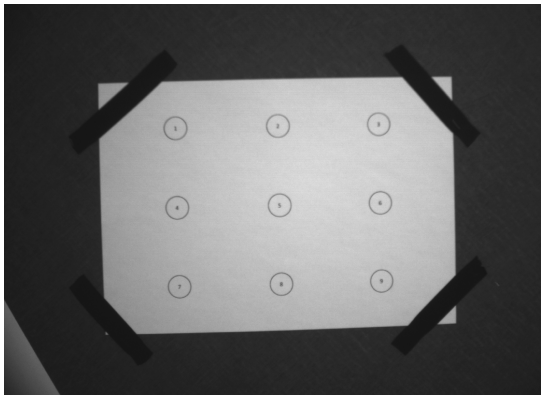

Fig. 10. Image of perspective calibration pattern captured from pre-calibration position.

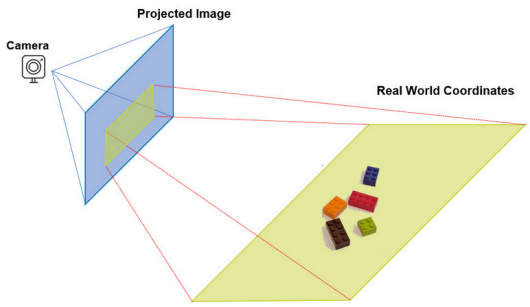

Fig. 11. Camera working environment setting [15]

IV. Implemented Solution

A. Assumptions

For the scope of this project a pinhole camera model [14] is used. In order to simplify the problem, the following assumptions were made:

1) All objects to be detected lie on the same plane.
2) Every image is captured from a pre-calibrated camera position and orientation, i.e. a fixed 6D vector.
3) The distance from the object plane to the camera center is pre-calibrated and remains constant for all images.
4) All objects are separated from each other.

B. Object Detection

Performing robust object detection was the most challenging part of the project. In order to find the best-suited algorithm, we tested different methods as explained in section II. However, many of these proved to be overly complicated and not well-suited for our scope. In particular, feature detection algorithms were failing to recognize the same features on the metal objects because of the shiny reflections on the metal surface. Moreover, the use of a deep neural network was restricting the image recognition to the objects present in the provided dataset in some cases, or required a supplementary training of the network. In both cases, the solution would not have been flexible enough to detect different objects and would have required a rather difficult setup of the object detection by the end-user.

For these reasons, we opted for the more flexible approach of a contour detection algorithm presented in [9]. The object recognition is based on the difference between a template image of the background and the current image of the scene. From this image difference, the algorithm recognizes any contours that were not present in the background template, assigning them an object label. The only requirement for this method to work is to have a fixed background and that the objects are well separated from each other. For the setup it is necessary to provide a template image of the scene without any object. The image processing was programmed in python and makes large use of the OpenCV library.

The process steps of the contour detection algorithm are shown in fig. 12. Since the algorithm works with grayscale images, so that the method is independent from color features, colored images must be converted to black and white in a first step. This was not necessary in our case, since our camera already takes grayscale images. After that, the background image is subtracted from the image of the scene containing the objects to make the objects more easily detectable. This is the crucial step of this method. It removes any noise that was already present in the background and is additionally highlighting the objects. If the subtraction worked well, the background should now be black and empty, and the image should only display the objects present in the scene.

In a third step, the image is blurred adding some Gaussian noise. This step makes the image smoother between pixels and takes some uncertainty into account regarding the contour detection. Smoothing means that the data points are averaged with their neighbors. This has the effect of blurring the sharp edges of the image [16]. The intensity of the blur can be adjusted by tuning the kernel size. The kernel defines the shape of the Gaussian function that is used to take the averaged values. A larger kernel corresponds to a wider variance, thus to a stronger blur [16].

It is also possible to blur the two images before taking the difference, in order to account for some offset of the background noise in the scene. This is the case if the background

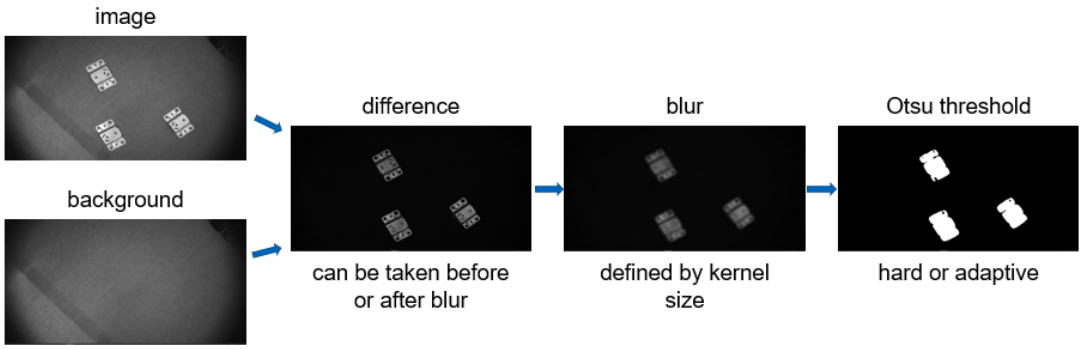

Fig. 12. Process steps of the contour detection algorithm.

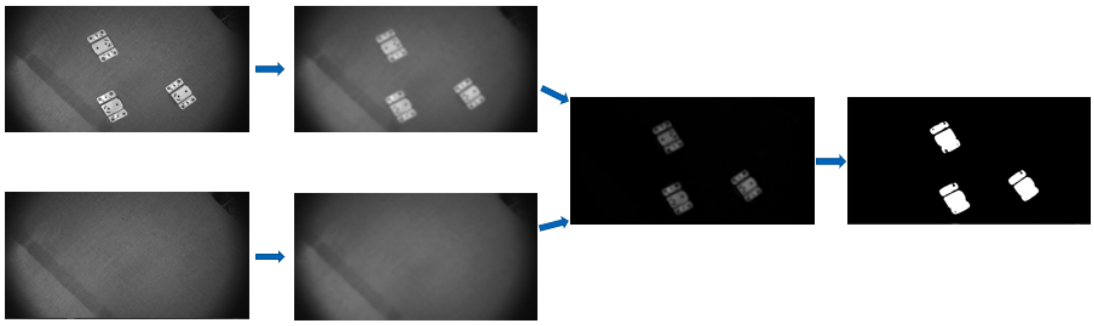

Fig. 13. Result of blurring background and scene images before taking the difference: The contour of a single object is separated into two contours as can be seen in the last image.

contains a lot of noise, and the scene has a small offset from the background image in terms of position. The Gaussian blur would resolve this problem by introducing some uncertainty in the position of the background noise. However, this might also remove some features of the object, leading to a detection of multiple contours and, in the worst case, to the separation of a single object, as shown in fig. 13. This problem might be solved by blurring a second time after taking the difference.

After taking the difference and blurring the image we apply thresholding, which means that the algorithm will decide to place pixels in either the background or foreground (binary). Background pixels will be turned off (black) and foreground pixels will be white. This segmentation is done by the Otsu's method [17], which works by maximizing inter-class variance, which is equivalent to minimizing the intra-class variance, defined by the weighted sum of variances of the two classes:

$$\sigma_w^2(t) = \omega_0(t)\sigma_0^2(t) + \omega_1(t)\sigma_1^2(t) \tag{6}$$

where ω_0 and ω_1 are the weights corresponding to the probabilities of the two classes separated by a fixed threshold t and σ_0^2 and σ_1^2 are the variances of the two classes. In contrast to this hard threshold, OpenCV also provides an adaptive threshold that does not use a fixed value for t over the whole image, but takes into account a set of neighboring pixels and computes a local t. This option can help if blurring does not leverage all of the remaining noise, leading to false positive results in the object detection.

After thresholding, the objects present in the scene are shown in white on the black background. The final steps of the method are shown in fig. 14. The contours of the objects are now very clear, since they correspond to the border at which the pixel color changes from white to black. These can be found by the OpenCV function call *findContours()* [18] and are drawn by a green line inside the original image of the scene.

In the end, it is possible to draw a bounding box around the contours of every object, which has the same orientation as the object, by minimizing the area of the rectangle which contains the contours. Finally, the center of the bounding box can be calculated. The image coordinates of this point will then be transformed into world coordinates relative to the robot base, and will be the grasping point of the object. This process will be further explained in the following section (section IV-C).

C. Object Localization

Detected objects are enclosed by closed contours as shown in Fig:15. Determining the bounding rectangle from the contour is an optimization problem solved using OpenCV's *cv.minAreaRect()* [19] *i.e maximizing the enclosed contour area with minimum bounding box dimensions*. The center of the bounding box can be calculated by finding the intersection of its diagonals. The bounding box and its center are shown in fig. 16.

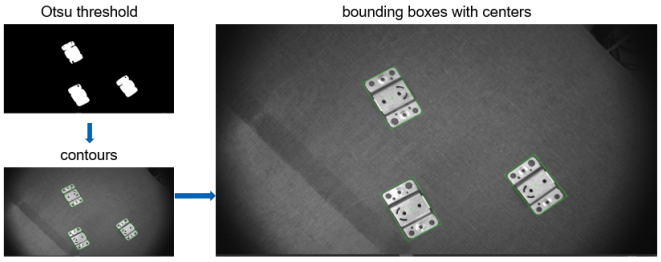

Fig. 14. Final steps of object detection: Contour detection and bounding box generation.

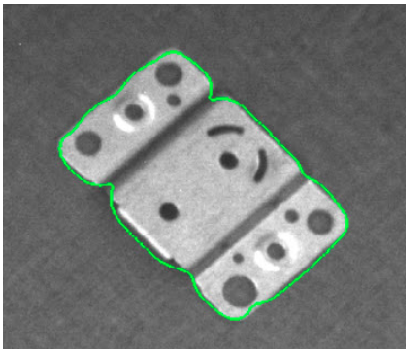

Fig. 15. Detected contour shape.

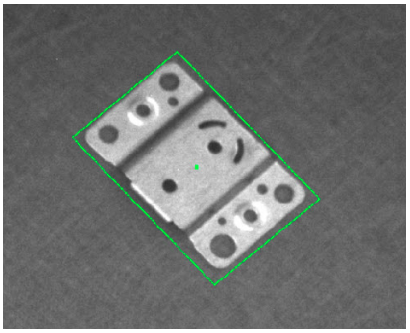

Fig. 16. Bounding box and its center from the detected contour.

Alternatively, the center pixel can also be calculated using the *Image Moments* [19] of the detected contour, but this approach does not provide accurate results. In fact, the center of mass of the object's contour does not always correspond to the object center, since it depends on the output of the thresholding step.

The conversion from image point to the object point is done using eq. (5), where the pixel coordinate *(u,v)* corresponds to the center of the bounding box. The scaling factor s, camera matrix A, translation vector t, rotation matrix R derive from the camera calibration. Upon substituting the values the output (X, Y, Z) is the 3D point corresponding to the center of the bounding box.

D. NodeRED Integration

To integrate the code used for object detection into the NodeRED flow, it is exposed as an HTTP-service as shown in fig. 17. To start the service, we run the *server.py* script, which executes our code as a service under a specific network port. When receiving an HTTP-request, the code is executed sequentially. First, the camera captures the image of the scene as described in section III-B. This captured grayscale image is used by the object detection algorithm. Lastly, the two-dimensional center pixel of the bounding box is converted into the corresponding three-dimensional real-world coordinate as described in section IV-C. This array of real-world coordinates is then sent back as an HTTP- response.

With the HTTP-service running, the NodeRED-interface [20] is used to communicate with the robot and the service. With NodeRED one can control a robot using a graphically programmed flow. Each flow consists of a set of nodes. As shown in fig. 18, the flow we implemented consists of Start Node, Movement Node, HTTP Node, Function Node, and End Node.

- *Start Node*: indicates the start of a robot flow.
- *Movement Node*: moves the robot's end of arm to the pre-calibrated position, so that each image is captured from the calibrated position.
- *HTTP Node*: communicates with the HTTP service (fig. 17). When flow control reaches this node, it sends an HTTP-request to the pipeline, The pipeline is executed sequentially and an array of 3D points is returned back to the HTTP Node.
- *Function Node*: is used to write inline JavaScript code within a flow [21]. Since the Function Node is wired to the robot flow after the Start Node and the HTTP Node, we can access both the robot object and the HTTP-response inside the Function Node. The HTTP-response is an array of 3D points. For each point in the HTTP-response, a set of robot actions is performed: First, the end of arm is moved slightly above the 3D point. After that, suction is initiated. Then, the end effector moves down to grasp the detected object. Once this has been

Fig. 17. Representation of the object detection algorithm exposed as an HTTP-service. After the robot sends an HTTP-request, the object detection pipeline is triggered and outputs the objects' 3D coordinates as an HTTP-response.

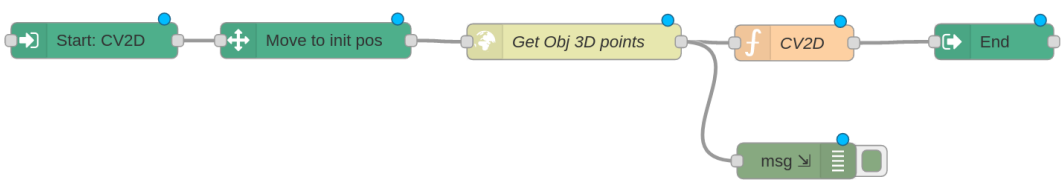

Fig. 18. NodeRED flow to control the robot movements and communicate with the HTTP-service.

done, the robot arm along with the object is moved to the dropping location. Upon reaching the drop location, the suction is released to drop the object. At last, the robot arm is moved back to its pre-calibrated position. This set of actions is then repeated for the next 3D point in the HTTP-response array until all objects have been grasped.

- *End Node*: indicates the end of the flow.

V. Results

Fig. 19. Hardware setting: 5-DoF robot including the end of arm, planar surface with objects and goal box.

To validate our solution, we used a robot with five degrees of freedom (DoF) as shown in fig. 19. This is the minimum number of DoF that are required for the application, but using a robot with more DoF would also be possible. Most importantly, the robot must be able to reach the pre-calibrated camera position and orientation for the application to work. In addition to the self-designed end of arm that was mounted to the last robot joint, pneumatic components had to be integrated to be able to operate the vacuum ejector together with the gripper. We supplied it with pressurized air from a portable compressor and an interposed valve to be able to switch the vacuum generation on and off using a digital output of the robot's control unit. According to the assumptions, the robot was operating on a planar surface where the objects that should be detected were distributed. Lastly, we added a goal box as a final position to place the detected and successfully grasped parts.

After the parts are randomly distributed on the workspace, the robot flow can be started. Once this has happened, the robot performs all consecutive steps autonomously: It first moves to the pre-calibrated camera position and takes an image of the scene. Once that has happened, the object detection algorithm runs and outputs the array containing the 3D positions of all objects. Without a noticeable delay caused by the execution time of the algorithm, the robot starts to move according to the Function Node and for each detected object, it grasps it and puts it into the goal box. After the last object was successfully grasped, the robot moves back to its initial position.

In addition to the general application, we also showed that the robot is able to grasp different parts and is not restricted to a specific orientation of the parts in the workspace. Our sample part, the metal bracket shown in section I, could be grasped in both orientations, i.e. the one that is shown in section IV as well as upside down. The only limitation regarding object orientations is related to the gripper, as the suction cup needs access to a mostly flat surface in order to generate a stable vacuum. As the algorithm does not make use of object models, it is not limited to a certain kind of object. We showed this by also grasping small wooden plates. Figure 20 shows an exemplary scene of objects that could be grasped successfully.

During validation, the weaknesses of the application also became visible. Due to a small offset between the correctly detected object centers and the 3D position of the end effector when attempting to grasp, the robot sometimes was not able to grasp objects, e.g. as the suction cup was placed over a hole in a bracket and could not generate a sufficient vacuum. Using another robot with different kinematic but the same pre-calibrated camera position, this problem did not occur. Therefore, as the offset most likely appeared due to inaccuracies in the robot's mechanics, we could not solve this problem. Inherent to the assumption that all objects have to lie on the same plane, the application is limited to objects with approximately the same height as the distance between the pre-calibrated camera position and the object plane is fixed. By using a flexible suction cup that allows for small variations in the object height, the range of graspable objects can be extended slightly. If objects of arbitrary shape want to be

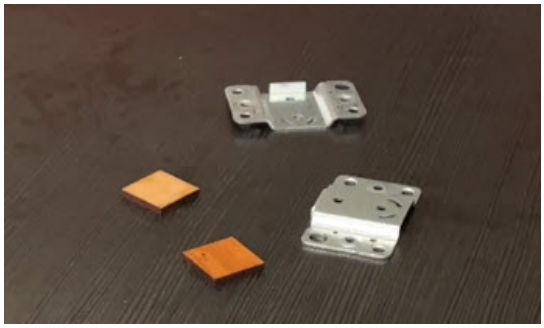

Fig. 20. Exemplary scene in which all objects were grasped successfully, containing the metal bracket in both orientations as well as small wooden plates.

grasped, more advanced object detection algorithms have to be used, as will be explained in the next section.

VI. Conclusion and Future Work

In this work, we introduced a 2D vision application for a modular robot: We implemented an object detection algorithm that works with a single grayscale image as an input, developed the mechanical mounting for a suitable camera and gripper, integrated everything into the robot's web-interface as a pick-and-place application and verified our solution with real-world experiments. For the given assumptions, this application enables a modular robot to detect and grasp unknown objects.

As the unstable image acquisition with the given camera was a major problem, a different camera should be considered for future work, e.g. a USB3-camera that is better suited for use with notebooks. In general, future work should also be focused on releasing as many of those assumptions as possible. The most constraining assumption is the fixed, pre-calibrated camera position as it sets requirements for the robot kinematic and the application in general. To solve this problem, an automatic perspective calibration would be needed. Furthermore, the object detection algorithm is limited as it does not gain any knowledge about the detected objects and cannot truly recognize, i.e. distinguish them. For this reason, occluded or overlapping objects cannot be detected and the algorithm is limited to very specific objects, e.g. objects of approximately the same height and with a feasible grasping point in their center. To overcome these problems, a model-based object recognition algorithm would be suitable. Depending on the algorithm, it would be able to recognize the objects, use information from an object model to get the height of an object or even return a full 3D pose estimation. Lastly, one could even take two images with our 2D camera to gain depth information or directly use a 3D camera to be able to use 3D object recognition algorithms. Obviously, this would leave the scope of this work, which is 2D computer vision, but is mentioned for the sake of completeness.

References

[1] D. G. Lowe, "Object recognition from local scale-invariant features," in *Proceedings of the seventh IEEE international conference on computer vision*, vol. 2. Ieee, 1999, pp. 1150–1157.

[2] "OpenCV FLANN," https://docs.opencv.org/4.x/d5/d6f/tutorial_feature_flann_matcher.html, Accessed: 2022-03-15.

[3] "OpenCV Homography," https://docs.opencv.org/4.x/d7/dff/tutorial_feature_homography.html, Accessed: 2022-03-15.

[4] H. Bay, T. Tuytelaars, and L. V. Gool, "Surf: Speeded up robust features," in *European conference on computer vision*. Springer, 2006, pp. 404–417.

[5] "OpenCV Feature Detection," https://docs.opencv.org/4.x/d7/d66/tutorial_feature_detection.html, Accessed: 2022-03-15.

[6] T. Hou, A. Ahmadyan, L. Zhang, J. Wei, and M. Grundmann, "Mobilepose: Real-time pose estimation for unseen objects with weak shape supervision," *arXiv preprint arXiv:2003.03522*, 2020.

[7] A. Ahmadyan, L. Zhang, A. Ablavatski, J. Wei, and M. Grundmann, "Objectron: A large scale dataset of object-centric videos in the wild with pose annotations," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2021, pp. 7822–7831.

[8] Y. Li, G. Wang, X. Ji, Y. Xiang, and D. Fox, "Deepim: Deep iterative matching for 6d pose estimation," in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018, pp. 683–698.

[9] pacogarciam3, "Lego set - object recognition example," *Kaggle*, 12/12/2018. [Online]. Available: https://www.kaggle.com/pacogarciam3/lego-set-object-recognition-example

[10] GitHub, "pypylon/grab.py at master Â· basler/pypylon," 3/14/2022. [Online]. Available: https://github.com/basler/pypylon

[11] Basler AG, "Network configuration (gige cameras) — basler," 3/7/2022. [Online]. Available: https://docs.baslerweb.com/network-configuration-(gige-cameras)

[12] "OpenCV Camera Calibration," https://docs.opencv.org/3.4/dc/dbb/tutorial_py_calibration.html, Accessed: 2022-03-13.

[13] "OpenCV Pose Estimation," https://docs.opencv.org/3.4/d7/d53/tutorial_py_pose.html, Accessed: 2022-03-17.

[14] "Pinhole camera model," https://docs.opencv.org/3.4/d9/d0c/group__calib3d.html, Accessed: 2022-03-13.

[15] "Prespective Calibration," https://www.fdxlabs.com/calculate-x-y-z-real-world-coordinates-from-a-single-camera-using-opencv/, Accessed: 2022-03-13.

[16] "An introduction to smoothing — tutorials on imaging, computing and mathematics," 5/28/2020. [Online]. Available: https://matthew-brett.github.io/teaching/smoothing_intro.html

[17] N. Otsu, "A threshold selection method from gray-level histograms," *IEEE transactions on systems, man, and cybernetics*, vol. 9, no. 1, pp. 62–66, 1979.

[18] "Opencv: Contours : Getting started," 3/16/2022. [Online]. Available: https://docs.opencv.org/3.4/d4/d73/tutorial_py_contours_begin.html

[19] "OpenCV Contour Features," https://docs.opencv.org/3.4/dd/d49/tutorial_py_contour_features.html, Accessed: 2022-03-13.

[20] "NodeRed Interface - Graphical Programming - Roboco," https://docs.robco.de/sections/node_programming.html, Accessed: 2022-03-13.

[21] "NodeRed Interface - Textbased Programming - Roboco," https://docs.robco.de/sections/text_based_programming.html, Accessed: 2022-03-13.