Kurzfassung


Abstract

In robotic the ability of perception is important for many tasks. There are numerous perception packages that were tested in this work. The packages have been analyzed on a PR2. It has been analyzed how the installing process works and the learning and detection of objects. Problems have been written down and solutions were tried to find as well as documented. In the second part of this work packages related to markers have been examined. Markers are good to make complex perception tasks easy. These packages have been tested upon their functionality and applicability. A comparison between packages wasn't possible because the packages didn't work with the same benchmark.
# Contents

List of Figures .......................... 7

1 Introduction ............................ 9
   1.1 Motivation ........................... 9
   1.2 Goal ................................ 9
   1.3 Setup ............................... 10

2 Theory ................................ 11
   2.1 RANSAC .............................. 11
   2.2 SIFT ................................ 11
   2.3 SURF ................................ 12
   2.4 ALVAR ............................... 13
   2.5 PCL ................................ 13
   2.6 OpenCV ............................... 14
   2.7 AR tag ............................... 14
   2.8 Aruco ............................... 15

3 Packages ............................... 17
   3.1 BLORT ............................... 17
   3.2 mlr interactive learner ............ 20
   3.3 Object recognition kitchen ......... 23
   3.4 COB 3D Segmentation ................. 30
   3.5 Find object 2d ....................... 33

4 Marker ................................. 37
   4.1 Ar track alvar ....................... 37
   4.2 aruco_ros ............................ 41
   4.3 ar_sys .............................. 43
   4.4 Summary of all marker packages .... 47

5 Conclusion ............................... 49
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Initial setup of the benchmark</td>
</tr>
<tr>
<td>2.1</td>
<td>Detection of the table top using RANSAC</td>
</tr>
<tr>
<td>3.1</td>
<td>MLR: Selecting the object to train.</td>
</tr>
<tr>
<td>3.2</td>
<td>MLR: The detected object.</td>
</tr>
<tr>
<td>3.3</td>
<td>ORK: Table detection</td>
</tr>
<tr>
<td>3.4</td>
<td>ORK: Coke can detection</td>
</tr>
<tr>
<td>3.5</td>
<td>ORK: Object is detected without learning</td>
</tr>
<tr>
<td>3.6</td>
<td>ORK: wood mesh is present in the database</td>
</tr>
<tr>
<td>3.7</td>
<td>ORK: No wood block is detected</td>
</tr>
<tr>
<td>3.8</td>
<td>ORK: capture_board [17]</td>
</tr>
<tr>
<td>3.9</td>
<td>COB: Segmented Cloud of a table containing objects.</td>
</tr>
<tr>
<td>3.10</td>
<td>FIND3D: Learning</td>
</tr>
<tr>
<td>3.11</td>
<td>FIND3D: The interface</td>
</tr>
<tr>
<td>3.12</td>
<td>FIND3D: Mirrored text on the User Interface.</td>
</tr>
<tr>
<td>4.1</td>
<td>ALVAR: Ar tag for the package ar_track_alvar.[5]</td>
</tr>
<tr>
<td>4.2</td>
<td>ALVAR: Visualization of markers with TF in RVIZ</td>
</tr>
<tr>
<td>4.3</td>
<td>ARUCO: Visualization of arucoRos</td>
</tr>
<tr>
<td>4.4</td>
<td>ARSYS: Aruco marker</td>
</tr>
<tr>
<td>4.5</td>
<td>ARSYS: Error ar_sys package</td>
</tr>
</tbody>
</table>
1 Introduction

1.1 Motivation

Perception plays a huge role in all of robotics, everytime something shall be picked, seen, navigated to and so on the specific object needs to be percepted at first and most of the time also it’s position. There are a lot of different perception packages for ROS that can be used for this. But most of the time it’s relatively hard to find a good package that works for your purpose. Another problem with most of the packages is that they are mostly written for specific robots or camera setups, so before using them you will need to configure them to work for you and also you need to get to know how they are being used. Often there is only a really spare documentation which leads to the need of figuring this out on your own. Therefore in this document we want to test and compare different perception packages that we found to be worth being tested.

1.2 Goal

The purpose of this work is to find different perception packages, get them up and running and try to compare them as good as possible. Actually we wanted to do a benchmark (see figure 1.1) that is equal for all the packages to exactly evaluate them on the same basis and afterwards give a conclusion which of the packages shines at what. This benchmark consisted of a table with objects on it that we thought provided the greatest variety in shapes (simple box, child toy) and also in coloration (plain-colored, colorful texture). We also had objects with different levels of transparency to provide a wide variety of difficulties to test our packages with.

But since it took a lot of effort to get most of the packages running at first and even if we got them running they couldn’t really be used because they only work for very specific tasks or they still didn’t work as said, we decided to evaluate each package individually. Therefore we now provide a short description of each package, the requirements, installation instructions and a conclusion with positive and negative aspects.

To do so we differentiated between packages that work without markers and those relying on markers to percept objects.
1 Introduction

1.3 Setup

Our setup is really simple, it is just a PC running Ubuntu 14.04 with ROS Indigo installed and a Microsoft Kinect attached to it. We also had access to a PR2 of Willow Garage that we could use to test some more specific PR2/Robot related packages. Apart from some packages where we used the PR2, there was just a table with some objects on that we tried to detect with various packages and a kinect attached to a tripod.

In order to use the kinect and see the results, if not already installed, you have to execute the following commands:

- `sudo apt-get install ros-<your ROS distro>-rviz ros-<your ROS distro>-rqt_reconfigure ros-<your ROS distro>-openni`

The distro we used was ROS indigo.
2 Theory

2.1 RANSAC

RANSAC (RANdom SAmple Consensus) by Fischer and Bolle is an algorithm for detecting outliers and gross errors of measurement values. RANSAC is used to find inliers and outliers for any type of model fitting [23].

Algorithm 2.1 RANSAC

1: Randomly select the minimum number of points required to determine the model parameters.
2: Determine model parameters from these points.
3: Choose a subset of measured values (Consensus Set), which distance to the model is less than a certain threshold.
4: Everything that is not an outlier is determined as an inlier. The model is now refitted with only the inliers.
5: Otherwise the algorithm is often repeated but only N times. It is repeated until the desired accuracy is reached. The amount that include the most points is at last selected.

[23] [21]

Object detection with RANSAC algorithm:

The RANSAC algorithm removes all defective points which do not belong to the object. Afterwards the algorithm calculates a connection between the object and the scene. The outcome can then be represented as a point cloud and the objects can be shown in form of color (see figure 2.1) or framed by bounding boxes [25].

2.2 SIFT

SIFT (Scale-Invariant Feature Transform) is an algorithm for detecting objects in an image. It is invariant to image translation, scaling and rotation.
To detect an object an image of the object is needed. Scale-invariant means that feature detection is not affected by rotation or scaling.

The algorithm at first generates a Gaussian pyramid of continuously more smoothed images. Then, the difference between two adjacent pictures is calculated, called the difference of Gaussian. Afterwards potential feature points are determined. This is done by calculating the maxima and minima of a difference-of-Gaussian function. Thereafter, a stability analysis takes place. This will check corners and contrast as possible features. Furthermore, the exact position, the scale and the orientation of the image are determined. The orientation corresponds to the gradient. Thereafter, the determined characteristics are stored into vectors so-called SIFT keys. Each of these points is now assigned to a position. To find the object in an image the SIFT keys of the object are compared to the SIFT keys of the image. This is done by using the nearest neighbor algorithm or binary trees. Both technologies have their advantages and disadvantages. It depends on the situation which method will be used. Only three matches are required to find an object in an image[26].

2.3 SURF

Like the SIFT algorithm, the SURF (Speeded Up Robust Features) algorithm contains both, a detector and a descriptor. SURF uses a fast-hessian algorithm as a detector. Unlike in the case of SIFT, SURF uses box filters to approximate the second order Gaussian derivatives used for computing the Hessian matrix. In order to analyze the scale space, the filter is up-scaled and applied to the original. This is faster than iteratively reducing the image size and can even be done in parallel. To find points of interest, a non-maximum suppression in 3x3 field over 3 scale layers (resulting in a 3x3x3 neighbourhood) is applied. Now the maxima of the
determinant of the Hessian are interpolated in both, scale and image space. The first step the SURF descriptor does is identifying the orientation of the points of interest. This is done by calculating the Haar-wavelet responses in a circle with the radius of 6 times the scale of the filter, at which the point was detected. The wavelet responses are weighted with a Gaussian ($\sigma = 2.5$ times the scale) that is centered at the interest point and represented as vectors. The horizontal response strength is drawn along the abscissa and the vertical response strength is drawn along the ordinate. To find the dominant orientation, the sum of all responses within a sliding orientation window with an angle of $\frac{\pi}{3}$ is calculated. The longest calculated vector defines the dominant orientation. Now a square region with a size of 20 times the scale is constructed around the point of interest. The region should be oriented along the dominant orientation. The extracted region is now split up into smaller 4x4 square sub-regions. In every sub-region the Haar wavelet response is calculated in vertical ($d_y$) and horizontal ($d_x$) direction. The four-dimensional descriptor vector $v$ consists of $\sum d_x$, $\sum d_y$, $\sum |d_y|$, $\sum |d_x|$. Like in SIFT the matching is done using the nearest neighbor algorithm [20].

2.4 ALVAR

ALVAR is a library with the aim to provide tools for creating virtual and augmented reality. This shall be as easy and flexible as possible. The main feature and why ALVAR is mentioned here are its capabilities for marker based tracking. For that use case it especially provides an accurate pose estimation. Also it is possible to bundle multiple markers to use it for full object detection. The library contains several other features but in the robotics case or especially for this work we are only interested in marker detection [3].

2.5 PCL

PCL, which stands for Point Cloud Library, is, as the name says a library for 2d/3d image and pointcloud processing software. It contains a large amount of different algorithms for filtering, feature estimation, surface reconstruction, registration, model fitting and segmentation. There are several use cases that can be done by only using this large and open project. It is for example possible to filter out a plane like a table and then filter all the points on that table. You can also get all the points of the objects on this specific table and if there is some noise in the data it is even possible to filter out all the outliers that may occur during your filtering. PCL is not specifically made to be used in robotics or with the ROS environment. But since it is such a powerful collection of tools a lot of packages for robotics use PCL as the underlying framework or just use some of the possibilities provided by it. PCL is somewhat like a
toolbox for everything that has to do with image and pointcloud processing. Even some of the
algorithms explained before are already implemented in PCL [16].

2.6 OpenCV

OpenCV, where CV stands for "Computer Vision", is another library that contains a lot of
algorithms for image processing. It was developed by Intel, first introduced in 2006, and was
then maintained by Willow Garage until the beginning of 2015. Mainly to make computer
vision accessible to programmers that work on real-time Human-Computer-Interaction or with
mobile robots. The main advantage for using OpenCV lays in the computational speed it
provides and the fact that it doesn’t only provide a huge amount of algorithms but also that
those algorithms include the current state of the art of research. It is split in different modules
starting with a core functionality that defines basic data structures as well as basic functions
that are used by all the other modules. Other packages are image processing, video analysis,
calib3d, which provides calibration as well as for example pose estimation, object detection,
a highgui and gpu accelerated algorithms. These contain all the different algorithms and
can be used to tackle a bunch of different use cases like face detection, object detection and
segmentation to only name a few. Furthermore it even provides the capability to use machine
learning to some extend for example boosting, bayes classification or support vector machines.
OpenCV is programmed in C/C++ such that it can be included into coding directly and is also
cross platform which leads to the possibility to use it on many different operating systems. As
the name says OpenCV is an open source library and free for use under the open-source BSD
license. Only some of the used algorithms that are patented or have limitations for free use are
provided in a nonfree package [14] [15].

2.7 AR tag

Many products in the industry have markers on them, like QR(Quick response)code or a
barcode to carry information. Fiducial markers are useful in robotic as well. If you want to
have the relative position between the camera and some objects in the environment you can
employ a marker to find it. But QR Codes don’t function very well as fiducial markers because
they aren’t meant to be used for fiducial systems and they have problems with a large field of
views and the perspective. Instead of this codes you should use AR (augmented reality) tags.
It is important that the markers transport as little information as possible. Therefore they only
carry an ID to ensure quick detection.
The construction of the AR tags looks as follows. AR tags are bi-tonal planar that means
that they only consist of the colours black and white. Every marker is unique based on its distinguishable ID number. The number is encrypted with robust digital techniques. This encryption creates a low false positive detection rate. It is very important to have a low false positive detection rate so a marker is only recognized when it is available.

Such planar patterns can easily be integrated into the environment and be detected by a camera. There are algorithms to detect such patterns in the environment [22].

### 2.8 Aruco

The aruco markers are also based on a fiducial marker system. All markers based on that system are square, so they have four corners as correspondence points for an easier pose estimation. They also have a wide black border with an inner image, that contains all the information of the marker.

The inner section of the aruco markers are binary codes, which are stored in an automatically generated dictionary. The generation needs two arguments: The number of bits (size) of the marker and the number of markers the dictionary should contain. The generation algorithm maximizes the distance between markers and the distance in the marker itself. The self-distance of a marker ist defined as the distance between the four different rotation-axis of the marker. The compilation of the dictionary is done using a probabilistic algorithm. It is better to have a dictionary with less markers than the maximal possible size of the marker would allow, because the surplus bits are used for error correction. The aruco marker detection algorithm is capable of correcting an error consisting of $\lfloor (\hat{\tau} - 1)/2 \rfloor$ bits, where $\hat{\tau}$ is the minimal distance between markers and the self-distance of the marker.

To detect an aruco marker the contours in the gray-scale image are extracted. After that all contours which can’t be approximated to a 4-vertex polygon are discarded and the inner contours are also discarded because we are only interested in the external contours. Now the inner region of the found polygons are analyzed. The first step here is to remove the perspective projection, after this the resulting image is divided into a regular grid, where each element is assigned 0 or 1 depending on the majority of pixels in it. At this time the first rejection test is performed, which rejects every found polygon that doesn’t have a black boarder. Now the code of the innerpart of the polygon is extracted and rotated in order to gain four different identifiers, one for each rotation. The identifiers are now searched in the dictionary. If no marker is found the distance between the extracted identifiers and all markers in the dictionary is calculated. If a marker has a distance that is equal or smaller than $\lfloor (\hat{\tau} - 1)/2 \rfloor$ this marker is considered as the correct one. In order to estimate the pose of the marker the reprojection error of the corners are iteratively minimized [24].
3 Packages

3.1 BLORT

3.1.1 Requirements

BLORT needs a 3d model with the shape and dimensions of the object you want to detect. The 3d model needs to be in SI units (meter) and the format of the file needs to be *.ply [19]. It uses an image feedback of a single calibrated camera and the camera_info topic of the same camera. It doesn’t need any depth information so a normal webcam is enough to get BLORT running.

For the calculation it uses GLSL, so you need a graphics card if you want to use this package.

3.1.2 Instruction

Learning

In order to detect objects with BLORT you need to learn them first. You can only train one object at the same time. The *.ply file needs to be in the directory defined in the file $(find blort_ros)/config/tracking.ini under the name ModelPath. The name of the file needs to be written down under the Model attribute. You also need to specify a SiftModel, if it doesn’t exist yet it will be created while learning. You also need to edit the $(find blort_ros)/launch/learnsifts.launch file and need to replace the info and image topics with the name of the topics you are using.

To learn objects you need to start the learnsifts.launch. This will show you the image feed of the camera with the model of the object to train in the middle. Instructions on every function of the learn sift program should show up in the console. At the time of writing this the available options are:

- [Space] Save texture face and extract SIFTS.
- [r] Reset tracker to initial pose
- [I] Lock/Unlock tracking
Learning consists of unlocking the models position (l) and saving different texture faces (spacebar) while rotating the object around every axis. If you loose track of the object you lock (l) its position, reset it (r) and unlock it again (l). The initially gray model will get more and more texture while you are learning it. When you have finished you save the *.sift file (return ) and exit the program (q, escape).

Detecting

Before you start the detection you need to specify the objects you want to detect. This is done in the file $(find blort_ros)/config/tracking.ini under the name Model, for the *.ply file and SiftModel for the *.sift file. These are the same options you need to adjust when learning an object, but if you want to detect them you can write down multiple objects whereas while learning you are only allowed to specify one object. After you have specified which objects you want to detect you need to start the tracking node with the command roslaunch blort_ros tracking.launch. This node will output the detected objects via the topic /blort_tracker/detection_result.

3.1.3 Problems

There were no problems getting the package to compile or run. But it is nearly impossible to learn a new object. In order to learn a new object properly you have to align it with the virtual object shown on the screen. This doesn’t work because the tracker tends to lose track of the object immediately after unlocking the virtual object.

3.1.4 Conclusion

BLORT is a package that doesn’t need expensive hardware, a single camera and a GPU is all you need. The downside of this is that a GPU is rarely found on a robot, in fact none of the robots listed on the ROS web page[18] have a GPU. Therefore you will need a separate computer with a GPU in order to use BLORT, which makes it unpractical for usage in productive environment. Besides of the problem of requiring a GPU in order to work, the problem of learning a new object is catastrophic. I wouldn’t suggest to use this package if
you want to detect something other than a coke can, because learning a new object is not working.
3 Packages

3.2 mlr interactive learner

3.2.1 Requirements

The mlr_interactive_learner package requires following topics:

- /kinect_head/rgb/image_color
- /kinect_head/depth_registered/image_rect
- /kinect_head/depth_registered/camera_info

The use of a XBox Kinect is recommended. The mlr_interactive_learner also uses the package

- libopencv-nonfree-dev

[9].

3.2.2 Instructions

Learning

Learning from stream
In order to learn an object from a stream you need to start

- rosrun mlr_interactive_learner learner_node

Be aware that this programm will create a folder named ’train’ in your current directory and will delete an existing folder with this name. A window will open, which shows you the output of the ’/kinect_head/rgb/image_color’ stream. When you see the object you want to learn you need to draw a rectangle with the mouse around it. As soon as you click somewhere in the window the stream will pause and you can take your time to select the object. If you aren’t satisfied with your selection, you can press ’r’ to reset it. Once you are satisfied with your selection, you can press ’s’, in order to save it and then hit ’space’ to resume the stream. Once you have exported enough selections of the object (it should be at least 20 selections), you can start the trainer with the button ’t’. 
Learning from folder
Learning from a folder is only available once you have the ’train’ folder. If you have this folder you can simply run

- `rosrun mlr_interactive_learner learner_node /PATH/TO/FOLDER/`

Detecting
After you have learned an object you can start the classifier with the command

- `rosrun mlr_interactive_learner classifier_node`

[9] This will open a window with a live-stream of the `/kinect_head/rgb/image_color` topic. Once the classifier is confident enough it has detected an object it will draw a rectangle around it and publish its position via the topic `rects`.

---

**Figure 3.1:** MLR: Selecting the object to train.
3 Packages

3.2.3 Conclusion

The camera frames and topics used are hardcoded but this is just a minor inconvenience you have to face if you want to use this package. It is not really fast in detecting an object, only one image is calculated every 2 seconds but the author of this package is working on improving the speed of the detection. Besides of that there is not much more to say about this package, only that it was the best working one we tested.
3.3 Object recognition kitchen

The object recognition kitchen package also ORK is a project that was developed by Willow garage. The purpose of its development was to start several object recognition techniques at the same time. [11]

3.3.1 Requirements

To detect the objects a camera e.g. a kinect is needed for this package. It is also required that ROS is installed on your computer. A table and an object must reside in the area captured by the camera. In addition, stl files of each objects are required. To test the package you need a Coca Cola coke can[12].

3.3.2 Introduction

Installation

To use the package you first need to execute following commands:

- export DISTRO=indigo
- sudo apt-get install libopenni-dev ros-$DISTRO-catkin ros-$DISTRO-ecto* ros-$DISTRO-opencv-candidate ros-$DISTRO-moveit-msgs
- source /opt/ros/$DISTRO/setup.sh

For the package you need:

- sudo apt-get install ros-indigo-object-recognition-core

In addition you need the package ork_tutorials. Clone this package from git into your own workspace.

- https://github.com/wg-perception/ork_tutorials.git

Furthermore the couchapp database must be installed for managing our own objects:

- sudo apt-get install couchdb

If you install the database correctly you can test it:

- curl -X GET http://localhost:5984

Your output should look similar to:
3 Packages

- `% "couchdb":"Welcome","version":"1.0.1"

Afterwards install the following:

- `sudo pip install -U couchapp`

After the installation start:

- `rosrun object_recognition_core push.sh`

The output to the Terminal is a link. On this page the meshes are displayed that are present in the database. At the moment this page is empty.

- `http://localhost:5984/or_web_ui/_design/viewer/index.html`

Learning

All objects that are identified by object recognition kitchen are stored in a database. All steps mentioned in the chapter installation of the database must be run before that.

To be able to recognize the objects, you need a mesh for each of them. It can be in following formats: .stl/obj. We have used the .stl format because it can be built with blender.

The next step that needs to be performed is to add the object to the database. We try to add a Coke can to the database. First step is:

- `rosrun object_recognition_core object_add.py -n coke -d "A universal can of coke" --commit`

After that we get an id of the stored new object. For example:

- `c5d0439fe0b1e7646b6d2cb6e00045e`

With following link you can see whether the new object is stored in the database.

- `http://localhost:5984/_utils/database.html?object_recognition/_design/objects/_view/by_object_name`

The next step is to add the mesh to the created object. To do that you need the id and the .stl/obj. file of the object.

- `rosrun object_recognition_core mesh_add.py YOUR_OBJECT_ID /catkin_ws/src/ork_tutorials/data/coke.stl --commit`

Afterwards the mesh can be viewed under the link:

- `http://localhost:5984/or_web_ui/_design/viewer/meshes.html`
Detecting
We have used object recognition tabletop for detection. The advantage of this is that only the
mesh of object is required for detection. This detection has a rviz plugin to show the results.
The first steps to start the detection are:

- `roslaunch openni_launch openni.launch`
- `rosrun rviz rviz`
- `rosrun rqt_reconfigure rqt_reconfigure`

It is important that the fixed frame in rviz is camera_rgb_optical_frame. If the rqt_reconfigure
is opened then put a hook in depth_registered driver depth_registered.

To detect something that looks like a table you need to start:

- `rosrun object_recognition_core detection -c ‘rospack find object_recognition_tabletop’/conf/detection.table.ros.ork`

In rviz you must add ORKTable and choose the topic /table_array. As a result, our table was
highlighted by a blue frame, as described in the tutorial. It also displays the orientation of the
table with an arrow. The result should look similar to figure 3.3.

Figure 3.3: ORK: Table detection
To detect the coke can and other objects start:

- rosrun object_recognition_core detection -c ‘rospack find object_recognition_tabletop’/conf/detection.object.ros.ork --visualize

First, we only tested it with a coke can. For this ORK object must be added to rviz and the topic /recognize_object_array must be selected [13]. As a result, we get a white label with the name coke on the coke can in the image (see figure 3.4).

Problems
The first problem was to launch the detection. The given commands in the tutorial do not work at all but after several attempts we succeeded to still launch it.

The next problem is that everything that looks similar to a Coke can and a table is detected as such. So even a non-qualified object is recognized as coke can, if it has the similar size (see figure 3.5). Although the simple red box in the picture does not exist in the database it has been recognized as a coke can because it has a similar size and color.
And even walls are recognized as tables by the table detection (see figure 3.5). Other objects can be easily inserted into the database. The problem with this is that they are not recognized by ORK. In Figure 3.6 it can be seen that the wood mesh is present in the database. The next picture (see figure 3.7), unfortunately, shows that we get no output because no white label with name timber is displayed. We have tried to insert the objects in the database with DataCapture. To do so you need the capture_board [17] and place it in the middle of the table and put the object on it. First we need to capture our work space in an image. Start:

- `rosrun object_recognition_capture orb_template -o my_textured_plane`

Press key ’s’ to save the image. Then press ’q’ to exit the window. Start:

- `rosrun object_recognition_capture orb_track -track_directory my_textured_plane`

It opens up a window that should clearly show the current work space. Press ’q’ to quit and try another time. In the next step a bag file is recorded with 36 images. For this you should start following command.

- `rosrun object_recognition_capture capture -i my_textured_plane –seg_z_min 0.01 -o silk.bag`

The problem was that we could not generate this bag file. So we are not able to include the data into the database with this method.
Figure 3.6: ORK: wood mesh is present in the database

Figure 3.7: ORK: No wood block is detected
3.3 Object recognition kitchen

3.3.3 Conclusion

The part of the package ORK for object detection works, but unfortunately only with Coke cans. It is important that if Coke cans shall be properly recognized no objects that look like cans of Coke are located in the same environment. The problem is in fact that everything which looks like a Coke can is recognized as such. The table detection part of the package ORK recognizes the tables properly, but it doesn’t only detect tables as a table which leads to a high unreliability of this function.
3 Packages

3.4 COB 3D Segmentation

3.4.1 Requirements

The COB 3D Segmentation Package doesn’t have a lot of requirements. The only thing that really is needed, is a camera that can capture a depth cloud and works with ROS, e.g. a Microsoft Kinect.

3.4.2 Instructions

Installation
To install COB on the Indigo distribution of ROS you have to execute the following steps:

1. At first you have to install the following packages via apt-get install:
   • libcgal-dev
   • libwxgtk2.8-dev
   • libfftw3-dev
   • ros-indigo-navigation
   • ros-indigo-cob-common
   • ros-indigo-cob-driver
   • ros-indigo-cob-command-tools
   • ros-indigo-cob-extern
   • ros-indigo-cob-perception-common

2. Then you have to clone the following package via git clone:
   • https://github.com/ipa320/cob_environment_perception.git

3. And at last you have to build the package. Don’t forget to source /devel/setup.bash in the git repo.
3.4 COB 3D Segmentation

Problems with the installation
At first we tried to build the whole package on Hydro and didn’t get it build at all. After contacting one of the developers of the package to ask him for help, he told us that we should rather use Indigo instead of Hydro because they stopped developing against Hydro. But even after trying to install on Indigo we weren’t able to install the package and get it running. Luckily the developer was really nice and did help us even more and sent an installation guide, which we now provide in this document.

Usage
To use the package you need to have a topic running that publishes a depth_cloud. You can either use a kinect directly with openni, the camera on a real robot or even a bagfile that you created earlier.

In our case we used a kinect camera and therefore had to start the openni camera driver. In the next step you need to start the 3d segmentation:

```
roslaunch cob_3d_segmentation simple_segmentation.launch point
_cloud_in:=camera/depth/points
```

For the point_cloud_in-parameter you need to set your specific topic. Which you can find by simply using `rostopic list | grep points` and then choose whatever topic fits, if you don’t know it already. Afterwards the segmentation is already running and starts publishing the segmented cloud on the topic `/segmentation/segmented_cloud`. If you for example start RVIZ and add this topic to your visualization you should see something similar to figure 3.9.

After you have your segmented cloud the next step would be to start the object clustering. To do so you need to start the supporting plane extraction:

```
roslaunch cob_3d_mapping_semantics extract_supporting_plane.launch
```

And then to get bounding boxes arround the found objects you need to start tabletop object clustering:

```
roslaunch cob_table_object_cluster tabletop_object_cluster.launch
point_cloud:=/camera/depth_registered/points
shape_array:=/supporting_plane_extraction/shape_array_pub
```

Again with the specific point_cloud-parameter set to your `.../depth_registered/points`-topic. Now COB should be running just fine and you should be able to see bounding boxes being published through the `/bb_marker`-topic.[6]
Problems
Unfortunately, even with the help of one of the developers, we didn’t get COB to function properly. With his help we got it installed and got the segmentation running and as you can see in figure 3.9 our segmented cloud looks quite good. We also get the supporting plane extraction working, the table was found. But still, even though we changed all the topics to what they should be, we never got a bounding box published and it always told us that no objects were found. The developer couldn’t provide further explanation for this issue. So even if everything seemed to be running fine there was no object detection for us. Hopefully that was only an issue on our machine and works for others.

3.4.3 Conclusion
So all in all there is not much left to say. COB seems to be a really nice package, but as with a lot of other packages it is developed for one specific robot and to use it on another system you need to make some modifications which can, in this case, all be made with parameters that can be set when executing the start commands which is really nice. Apart from that the segmentation works really good. But since we couldn’t get the object clustering working we
can only say that like this the package was useless for us because it didn’t do what it should, which is detecting objects. So as already stated in the last section we really hope that this was only a problem for us and it works for everyone else. Therefore you might just give it a try with the risk that it possibly doesn’t work for you as well.

3.5 Find object 2d

This package provides a simple Qt interface to try different OpenCV implementations of SIFT, SURF, FAST, BRIEF and other features detectors. You can also detect objects and publish their positions on a ROS topic. It is the ros integration of the Find-Object application. [8] [7]

3.5.1 Requirements

For normal use with 2D data this package can be used with any camera. But if you use it with a kinect or kinect like camera that provides a pointcloud you can also start a 3D mode that makes it possible to get the 3D positions of the object. Since the package is dependent on OpenCV you will also need to install this package if you do not have it already installed.

3.5.2 Instructions

Installation
You can install this package through ubuntu's apt-get install command:

$ sudo apt-get install ros-indigo-find-object-2d
And that’s it.

Usage
After you have installed the package you will need to start a roscore in a terminal. After starting the roscore you can start your preferred camera node or camera driver. This can either be openni for a kinect or for example the uvc_camera node. And at last to start find_object_2d you need to execute:

rosrun find_object_2d find_object_2d image:=your_camera_node/image_topic
Then a window should pop up that let’s you interact and shows you a video stream with found points. If you want to use a pointcloud you will need to start find_object_3d with the command:

roslaunch find_object_2d find_object_3d.launch
3 Packages

Figure 3.10: FIND3D: On the left: The window that pops up after choosing to add an object. On the right: The next step when you chose an object.

Figure 3.11: FIND3D: The find_object_3d interface after adding an object. The green rectangle shows the detected object and the yellow points are features that are being detected.

Learning
To learn you simple have to choose "Edit"->"Add object" in the menu bar. Then the window on the left in figure 3.10. pops up and you can choose the object from the image. Afterwards the window on the right sight pops up and you can see how many features where selected and after clicking 'end' the object detection starts and you should see something similar to figure 3.11.
3.5 Find object 2d

Problems
There were no real problems but a rather small error we noticed. As it can be seen in figure 3.12 the information text is mirrored.

3.5.3 Conclusion

Find Object was one of the few packages that worked out of the box the only thing we had to do was to set the right parameter for our camera topic. For the testing we did, the package worked really good and with the right setup of algorithms the objection detection worked quite good as well. This package is great for comparing different methods for your specific use case and to play around with those methods. Unfortunately the package isn’t too great for real use in robotic tasks because you can only choose the objects to detect from the interface and have to do this every time. This doesn’t mean that it isn’t usable but we think that other packages might be better for that use case. All in all the package is easy to use and has a lot of different methods that can be tested and with the right setup the detection works quite good. Also the interface is nice and can be used quite easily.

Figure 3.12: FIND3D: Mirrored text on the User Interface.
4 Marker

4.1 Ar track alvar

First some general facts about ar-track-alvar. This package is a wrapper for the ALVAR software library mentioned in the theory part. The two main functions are creating your own marker and detecting them. You can identify the pose of markers and also track them [5].

4.1.1 Requirements

To use this package you need the operating system ROS and a camera for example a webcam or a kinect.

4.1.2 Installation

The installation is quite simple. There exist two methods to do this. The first is to checkout the correct branch of the git repository for example for the operating system indigo the link is:

- https://github.com/sniekum/ar_track_alvar.git.

The other way is to install it directly with:

- sudo apt-get install ros-<distro>-ar-track-alvar

When you have installed or cloned this package successfully you can use it [5].
4.1.3 Create Marker

To create your own marker you have to start:

```
rosrun ar_track_alvar createMarker
```

After that a simple tutorial starts in the terminal which helps you to create a marker. The marker needs an ID number and a size. So you are free to choose these parameters. When closing the terminal it saves the marker as a png file. On the homepage [5] there exists two finished marker png files that can be used. Now you can use the marker and detect it. They should look like in figure 4.1.

4.1.4 Detecting the marker

When you use the ar_track package with the PR2 then you only need to start:

```
roslaunch ar_track_alvar pr2_indiv.launch
```

You have to change this file when you have other camera topics. Furthermore you can add which marker size should be detected and which IDs. If a marker is detected then the node of this package publishes two topics. The first is the visualization_marker topic. The result of this topic can be viewed in rviz. For this you have to start rviz and add the visualization topic:

```
rosrun rviz rviz
```

When the marker is detected rviz shows you the coordinate axes of the markers. It should look similar to figure 4.2. The pose of the marker is published to the topic ar_pose_marker. In the terminal you can show the information e.g. the coordinates of the position of the marker or its
4.1 Ar track alvar

Figure 4.2: ALVAR: Visualization of markers with TF in RVIZ

ID. When you use a kinect as camera then you also get the depth information of the ar_tag [5].

If you want to detect three dimensional objects you can use the multi-bundle detection. This method combines some tags to one unit. For this you have to create a xml file. In this file you have to decide which ar_tag is the master and add the others in relation to it. This detection method is also used to improve the pose etimation of the ar_tags [5].

4.1.5 Positive

The marker package recognizes the markers very quickly and reliably. The output of ID and position of the marker are mostly correct.
4 Marker

4.1.6 Negative

On any object or place that shall be recognized needs a marker attached to it.

4.1.7 Conclusion

The package is reliable and it works fine. Creating ar_tags is simple and well explained through the tutorial. The detected pose of the marker is mostly correct.
4.2 aruco_ros

Aruco_ros is a "Software package and ROS wrapper of the Aruco Augmented Reality marker detector library" [2]. This package determines the 3D position of a marker, which is located in the area.

4.2.1 Requirements

To detect the markers you need a ros operating system such as hydro or higher. Furthermore you need an Aruco marker [1] and a camera for example a kinect to detect them.

4.2.2 Installation

To use this package clone the code from following link in for example your own catkin_workspace.

    https://github.com/palrobotics/aruco_ros

[2]

4.2.3 Detection

The detection works as described under aruco. To detect the marker you need to only start:

    roslaunch aruco_ros single.launch

To show the result of the detection start the follow:

    rosrun image_view image_view image:=/aruco_single/result

[2] After that a window pops up and shows you the result. It should look like in image 4.3. You can see the TF and the ID of the marker.
4.2.4 Positive

The marker is detected quickly and always a correct result is issued.

4.2.5 Negative

A marker needs to be placed on every object or location that shall be recognized.
4.3 ar_sys

ar_sys is a ROS package which uses the ArUco marker boards. With this package the 3D position of the markers in the environment can be estimated [4].

4.3.1 Requirements

To detect markers you need to have such ArUco markers e.g. in printed form. In addition a camera like a kinect or uvc camera and the operating system ros is also needed.

4.3.2 Installation

There exist two ways to get the package. First the direct install:

```
sudo apt-get install ros-<distro>-ar-sys
```

The distro is dependent on your operating system. We use indigo. The other way is to clone the git repository:

```
```

Now you are able to use the package.
4.3.3 Create Marker

There exists a aruco marker generator which generates a HTML page with the marker. This marker is compatible with the aruco marker detection algorithm. On this page [1] you can decide the ID and the size of the marker e.g. MarkerID = 20 and MarkerSize (mm) = 100. The generated marker can be saved as a picture and looks like figure 4.4 [1]. For this package, you need a single board maker which means that multiple markers within a block must be arranged.

4.3.4 Detection Marker

The camera parameters must be adjusted before the single board of markers can be detected. Therefore the camera_calibration.yaml needs to be edited according to your camera data. The information needed to edit the calibration file can be found in the camera_info-topic of your specific camera topic for the kinect this would be /kinect_head/rgb/camera_info. For the kinect we used the file looking as follows:
image_width: 640
image_height: 480
camera_name: camera
camera_matrix:
  rows: 3
cols: 3
data: [525.0, 0, 319.5, 0, 525.0, 239.5, 0, 0, 1]
distortion_model: plumb_bob
distortion_coefficients:
  rows: 1
cols: 5
data: [0.0, 0.0, 0.0, 0.0, 0.0]
rectification_matrix:
  rows: 3
  cols: 3
data: [1, 0, 0, 0, 1, 0, 0, 0, 1]
projection_matrix:
  rows: 3
  cols: 4
data: [525.0, 0, 319.5, 0, 0, 525.0, 239.5, 0, 0, 0, 1, 0]

We have also adjusted the camera topics and the camera node in the file texitsingle_board.launch For tracking start:

\texttt{roslaunch ar.sys single_board.launch} [4]

4.3.5 Positive

The marker is recognized very quickly.

4.3.6 Negative

\texttt{ar.sys} does crash as soon as a marker is recognized. The imageview works fine, and \texttt{ar.sys} is running fine as well, but at the exact moment that we put a marker in camera range \texttt{ar.sys} writes the marker position to the terminal and crashes afterwards. Thus, we had no issue in a viewer or could show the topics in rviz.
Figure 4.5: ARSYS: Error ar_sys package
4.3.7 Conclusion

We tried to solve the error with the developer but without success (look here: https://github.com/Sahloul/ar_sys/issues/3). Therefore, the package has not been usable for us.

4.4 Summary of all marker packages

In summary it can be stated that markers are very suitable to quickly get specific positions. These markers can be used to move the arm of the robot to a certain position or object. ar_track alvar and aruco can both be used to do so. The similarity of the two marker types is that both only carry the ID information. There are no more than ID and size as parameters needed to create the markers. They differ minimally in the output design. The only difference between both packages is that they use different types of markers. The ar_track alvar uses ar_tag markers and the aruco packages the aruco marker.

Unfortunately, the third package ar_sys can not be used because it crashes when it detects a marker.
5 Conclusion

In this work we at first outlined the motivation and goal of it. Then we provided a theoretical background. We explained some of the underlying algorithms used in the packages and also introduced some software libraries being used as well. After that we presented all the perception packages we have tested. We provide the requirements, an installation and usage guide, our problems with the specific package and also a conclusion of our thoughts about it for each package.

The next section provides the same things but only for marker based packages. All in all there is to say, that it was quite hard to find packages that we could get running for our setup. Only the marker packages worked relatively flawless.

Since we are no expert users this might just be the case for users that also have some lack of knowledge but it also shows that the packages are either not well documented or that they are just programmed for specific robots or setups. Which leads to the fact, that actually no package could really be used on the benchmark that we have planned and that out of all the packages only a few worked and even if they worked they only performed well on specific tasks.

This might be caused by the fact that most of the robotic users are experts and tend to program their own packages for the task they have to tackle and don’t publish the packages to be used for everyone. Therefore it might even be better to just try and program your own package with some of the introduced libraries such as PCL or OpenCV.

Concluding it is to say that this work was quite disillusioning because apart from the marker packages there were no packages that worked out of the box or, if they worked, performed well in any way.

The only package that was fine, apart from the marker packages, was the mlr interactive learner, but since it was created at the same institute as this work and therefore used on the exact setup it was programmed on and for, it might have the same problems as every other package for a different setup.
6 Bibliography


[16] **Point Cloud Library.** [http://pointclouds.org](http://pointclouds.org)


All links were last followed on March 17, 2008.
Declaration

We hereby declare that the work presented in this Fachstudie is entirely our own and that we did not use any other sources and references than the listed ones. We have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. We have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

place, date, signature