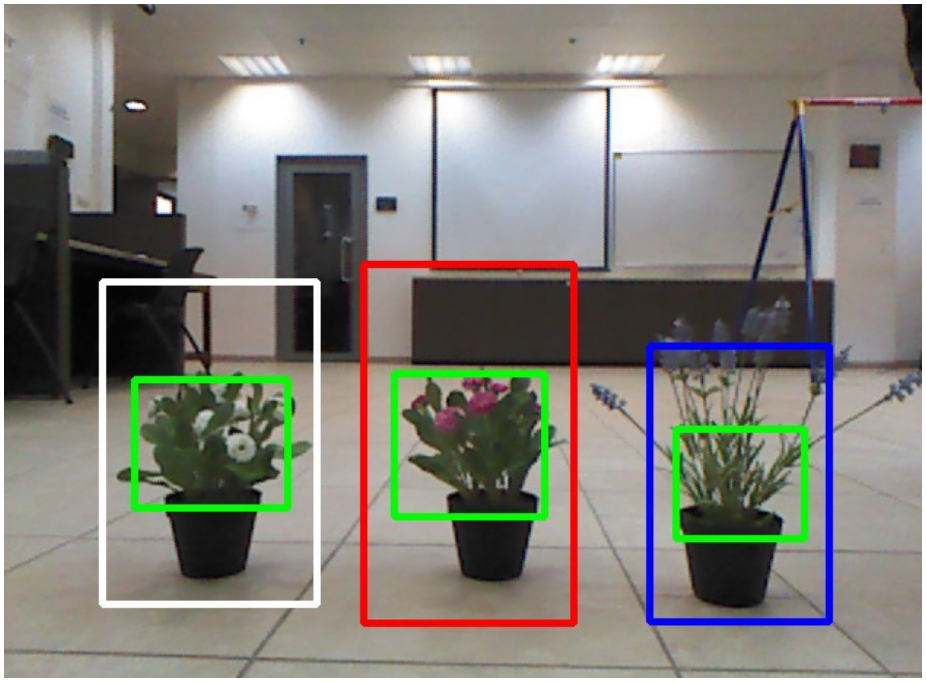
Project Flower

Vardan Papyan
CS Department
Technion - Israel
vardanp91@gmail.com

Emil Elizarov
CS Department
Technion - Israel
semil123@t2

April 20, 2014

Instructor: Mr. Amir Geva
Prof. in charge: Prof. Ehud Rivlin



1 Introduction

A new robot is being developed in the robotics lab. The robot's goal is transfering specific flowers in a greenhouse from one place to another.
The robot is being developed on a Turtlebot paltform which is equipped with a Kinect and runs on ROS.
Our goal was to create a flower detector and reconginer as a ROS node, which can be be easily used by the team who developes the robot.

1.1 Brief description of terms

ROS Robot Operating System (ROS) is a software framework for robot software development, providing operating system-like functionality on a heterogeneous computer cluster. ROS provides standard operating system services such as hardware abstraction, low-level device control, implementation of commonly used functionality, message-passing between processes, and package management.[1]

Kinect Kinect is a camera with a depth sensor.

Figure 1: Microsoft's Kinect

Turtlebot TurtleBot combines popular off-the-shelf robot components like the iRobot Create, Yujin Robot's Kobuki and Microsoft's Kinect into an integrated development platform for ROS applications.
The robot is supposed to detect the flower using the Kinect, recognize the flower (and by that deciding whether to move the flower or not). The movement of the flower will be through two robotic arms which will be connected to the robot, they would hold the flower and move it from spot A to spot B.[2]

Figure 2: Turtlebot 2

Opencv library OpenCV (Open Source Computer Vision Library) is an open source computer vision and machine learning software library. OpenCV was built to provide a common infrastructure for computer vision applications and to accelerate the use of machine perception in the commercial products. Being a BSD-licensed product, OpenCV makes it easy for businesses to utilize and modify the code.[3]

OpenNI The OpenNI (short for Open Natural Interaction) framework is an open source SDK used for the development of 3D sensing middleware libraries and applications.[4] If you want to use a Kinect you should use OpenNI[4].

1.2 The flowers

The goal of the project is to detect and recognize these three types of flowers:

(a) The white flower

(b) The pink flower

(c) The purple flower

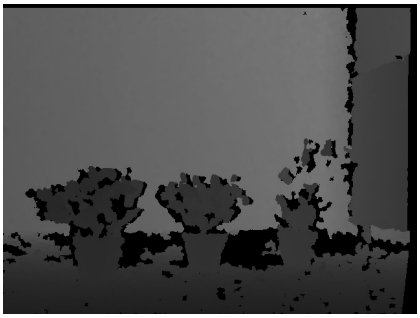
Figure 3: The three flowers which we have to detect and recognize

2 The Algorithm

2.1 The input

Our kinect provides us with two streams of images:

(a) Input RGB image



(b) Input depth image

2.2 Depth filtering

Our initial step is filtering the RGB image by depth. We choose only the pixels which are approximately 1 meter away from the robot.
This is a good opportunity to discuss about the properties of the depth image: The meaning of a depth image pixel getting (0,0,0) value (i.e. it is a black pixel) is that the camera is unable to determine wheter the pixel is close to the camera or far from the camera. Therefore we prefere to take the minumum amount of these pixels into into consideration when we analyzing the frame. The only scenario in which we take these pixels into considaration is when an object is

too close to the camera - in this case the depth image has zero values in all the pixels of the object. This is not our desire when a plant is right in front of the camera! The solution we came up with is to determine a threshold of zero value depth pixels, when the amount of zero value depth pixels exceedes the threshold then we do not discard any zero value pixel.

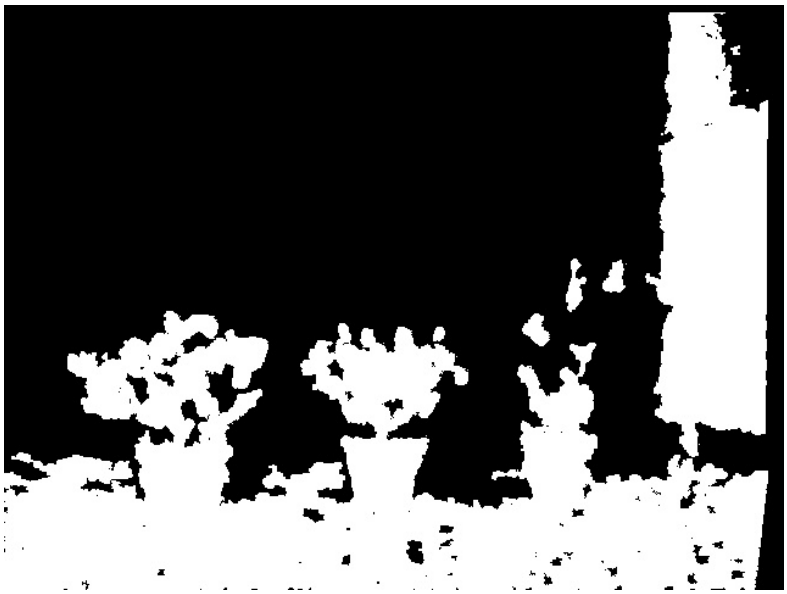


Figure 5: Depth mask

In the mask, the white pixels represent close pixels whereas the black ones represent the pixels beyond approximately one meter.
As you might see the white and the pink plants are "masked" well enough, but this is not the case with the purple plant - we noticed that the purple plan often gets a very minimal mask which is not enough to classify it correctly. For this reason we have decided to dialate the depth mask. By this we filter less of the purple plant, and the result is as follows:

Figure 6: Dialated depth mask

2.3 HSV Filtering

The next step is filtering all the green pixels. Our goal is to detect big green blobs which correspond to plants. Since we want to filter the green color and desire illumination invariance, we chose to work with the HSV colorspace. We sampled many green pixels from images we had captured and found the best HSV range for the plants' green color. The resulting green mask is:

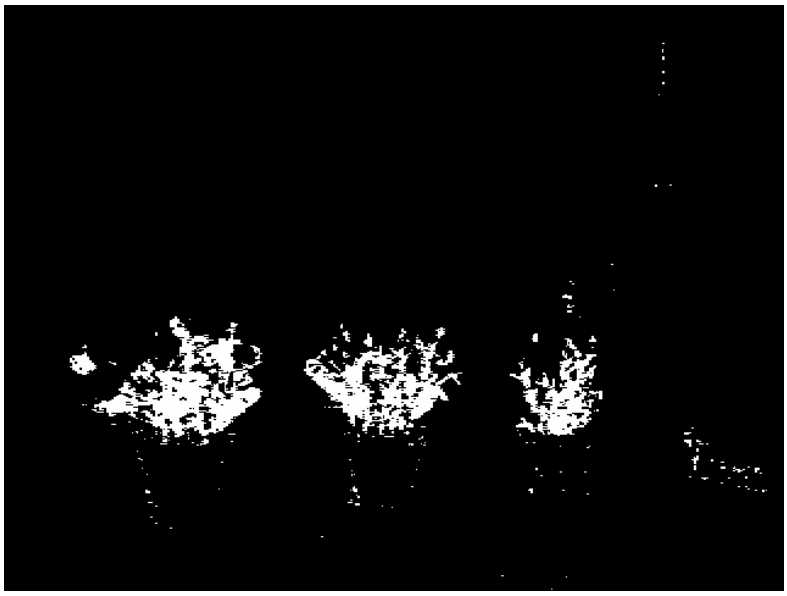


Figure 7: The green mask

2.4 Morphological Operations

Opening The opening operation is used for filtering noise from black white masks. It first uses erosion operator to shrink every connected component and then it uses dilate operation to restore the connected components to their original size. If a connected component was too small then it vanishes after the erosion and therefore after the opening operation we are left with all the large connected components. The resulting mask after the opening operation is:

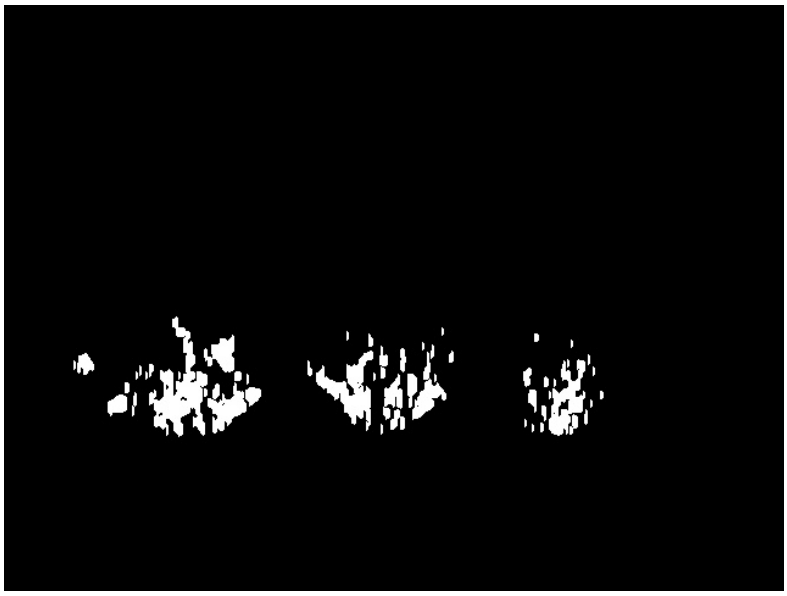


Figure 8: The mask after opening operation

Dilate The next step we are interested in is finding connected components which, as previously mentioned, correspond to plants. Our current RGB image has three connected components, however our mask has many connected components. Therefore we want to use dilate operator in order to inflate every pixel and find the correct three connected components. In order to do this dialation we use a series of different dialation operations. The resulting mask is:

Figure 9: The mask after dilate operation

We can now show the RGB image after masking it with the last mask:

Figure 10: The RGB image after all the masks

2.5 Finding The Connected Components

Our goal is finding connected components in our mask. The finding connected components algorithm works as follows:

1. Set all the white pixels with the flag "1" and the black one with the flag "0".
2. Set X:=2.
3. Iterate over all pixels:
   (a) If the pixel's flag value is equal to "1":
      i. Mark the pixel with X.
      ii. Recursively mark all pixels in this connected component with X.
      iii. X:=X+1.

Now we would like to show a demonstration of the above algorithm:
At first, the mask has zeros in the background (the black areas) and ones in the connected components (the white areas).
Assume we have this image as an input:

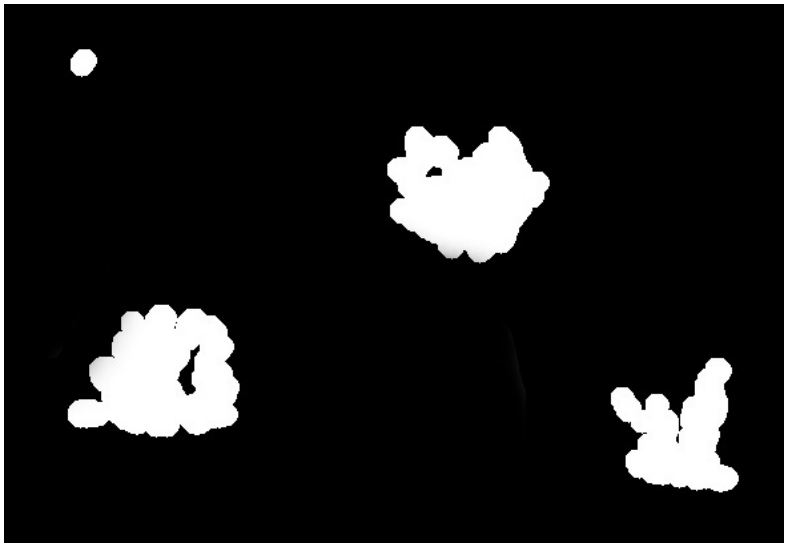


Figure 11: Initial mask

We iterate over the pixels and find the first pixel which has the value one. We then fill the connected component with the value 2. We will use different colors to represents areas with different value of X.

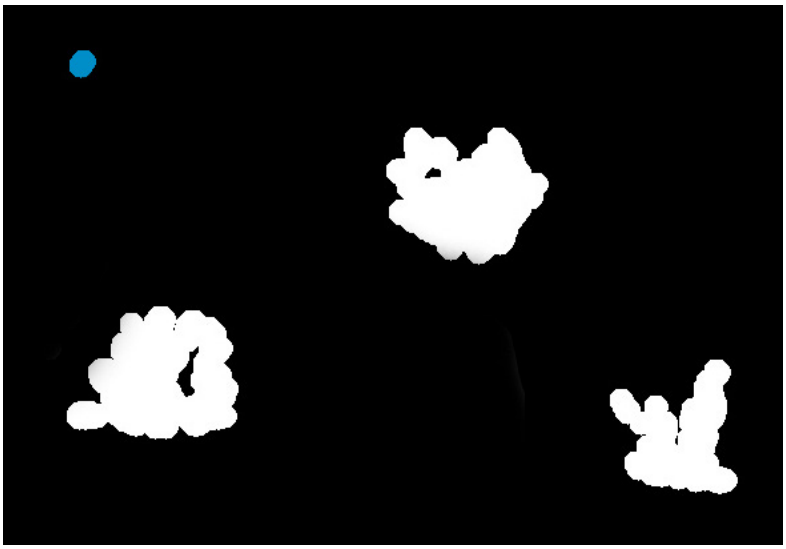


Figure 12: The mask after the first connected component filled

We now find another pixel which has the value one. We fill its connected component with the value 3.

Figure 13: The mask after the second connected component filled

The final mask will have zeros in the background, the mask won't have ones at all, and each one of the connected components will be marked with different number:

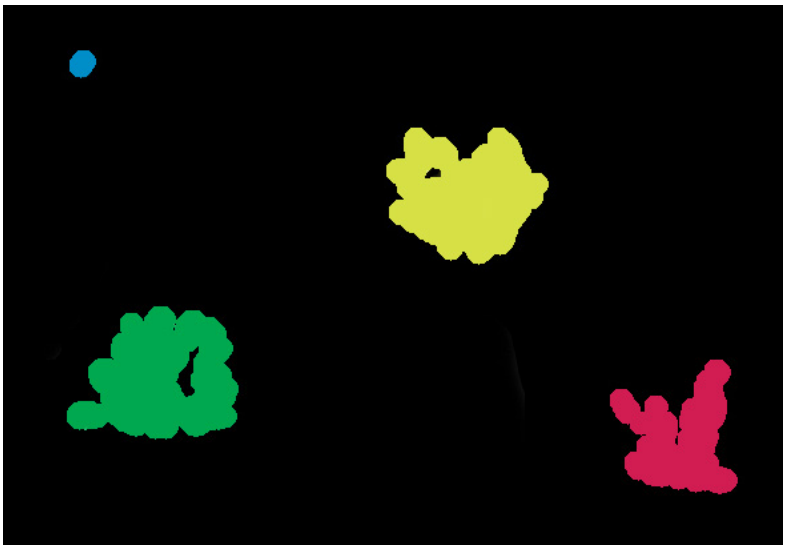
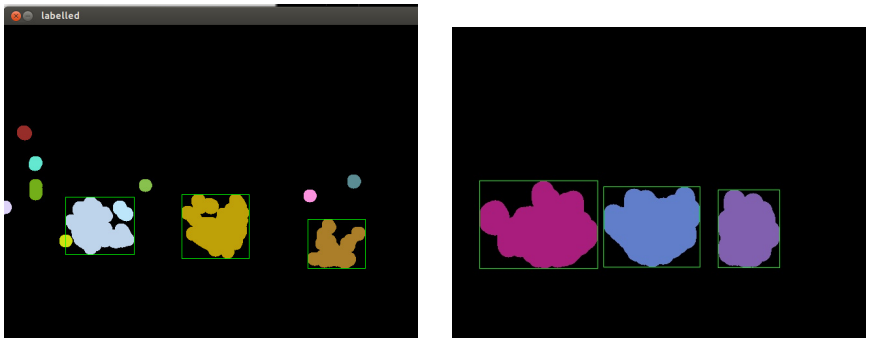


Figure 14: The final mask

Once we obtain our connected components we can bound them using rectangles.

2.6 Filtering Rectangles

The obtained connected components can be noisy, namely they are small connected components which do no correspond to any plant. We therefore must filter them. In order to do so we find the area (number of pixels) of the largest connected component and filter all the components smaller than some constant times this area. In addition we have a "hard threshold" which defines the smallest area of a connected component which can correspond to a plant. After filtering the components we finally get the blobs which represent the plants.



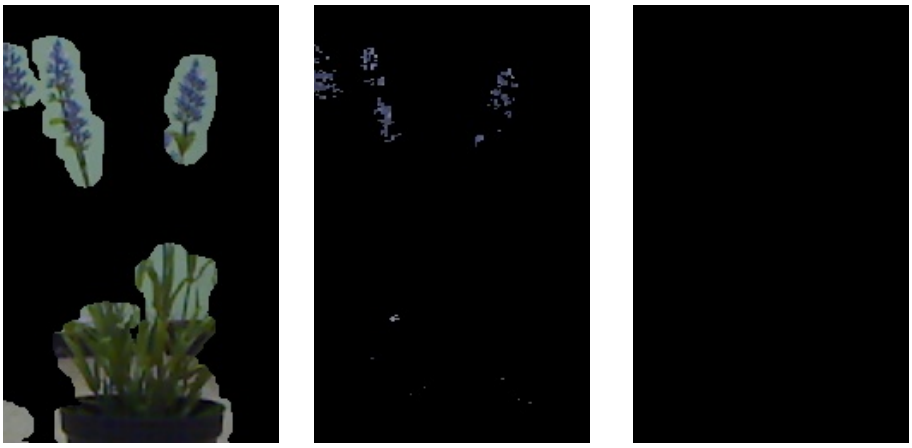
(a) A case with noise (b) Noiseless case

Figure 15: The blobs representing the plants bounded by thin green rectangles

In the beginning we had often a very noisy detection, but in a later stage after we did the "series of dialation" and fixed the green ranges the noise has reduced significantly.

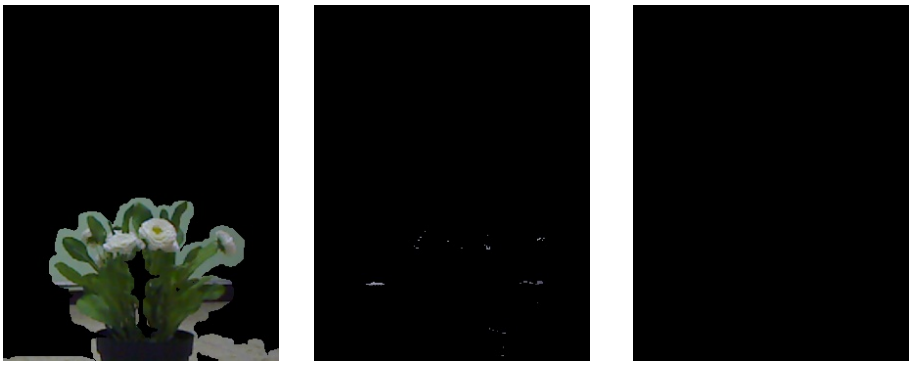
2.7 Plant Recognition

In order to recognize the plant, we must analyze the colors of the plants. One can not escape from using colors, since the pink and white flowers are of same structure and differ only by color. We therefore enlarge the rectangle which surrounded the green parts of the plants and try to find the percentages of the colors: pink and purple. We once again turn to the HSV colorspace in order to filter the pixels which are pink and purple. We have examined a lot of images and have found the best HSV ranges for the purple and the pink colors, and by using these ranges we have created two filtered images, one which consists of pink pixels and another which consists of purple pixels:



(a) Original purple plant (b) Purple pixels only (c) Pink pixels only

Figure 16: The purple plant and its colors



(a) Original white plant (b) Purple pixels only (c) Pink pixels only

Figure 17: The white plant and its colors

(a) Original pink plant (b) Purple pixels only (c) Pink pixels only

Figure 18: The pink plant and its colors

Once we've created our pink and purple masks we calculate the percentage of pink pixels in the pink mask and percentage of the purple pixels in the purple mask. If the absolute value of the difference between those two percentrages is smaller than a defined threshold then we classify the plant as white - the reason for this is that the white color turned to be extremely delicate to illumination changes, for istance this is a close up of a bright white (!) page:

(a) The white page is under the closet (b) A close up on the page

Figure 19: The white illumination problem

As you can see the page is purple, the same happenes with the white flowers. For this reason we have decided to classify the white flower in the presented method.
Now, if the flower is not classified as white in the previous stage, then the flower can be either purple or pink. We have trained a SVM to classify between purple and pink based on the hue histogram (will be explained in the following subsection).

2.8 The SVM

We have created two datasets of plants: one for purple plants and one for pink plants, each one of the datasets contains pairs of image and its mask, for example:

(a) The image (b) The mask

In the begining of our algorithm we train a linear SVM over the adjusted hue histograms of the dataset's plants:
Foreach image we create a hue values histogram (after discarding all the black pixels of the mask), for example the histogram of the above image is: (note that in OpenCV the hue values are between 0 to 179)

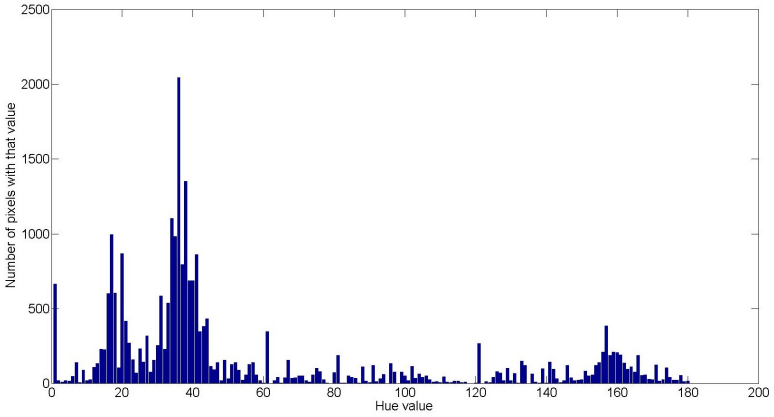


Figure 21: The immidiate hue histogram

Afterwards we nullify all the bars which are not in the hue range of the purple or the pink colors:

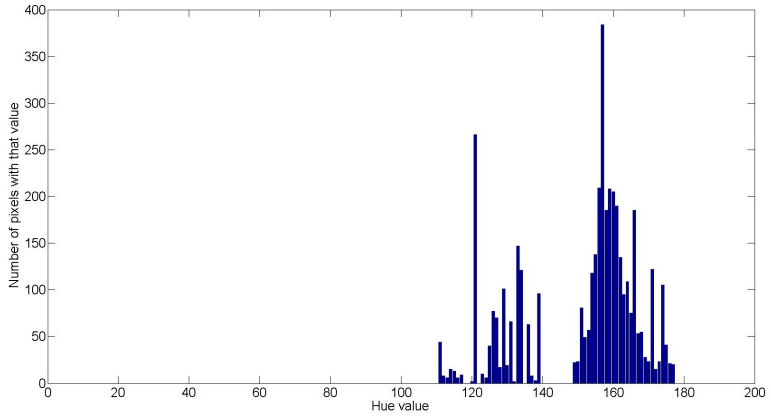


Figure 22: Only purple and pink hue histogram

And finally we normilize the histogram:

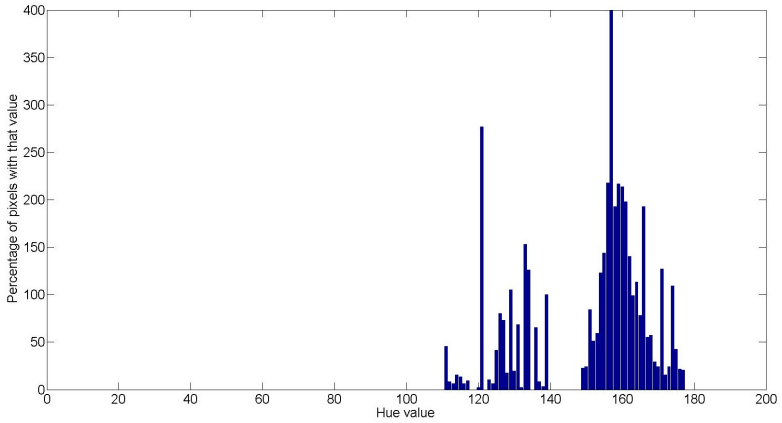


Figure 23: The final normilized histogram

We preprocess the whole dataset as mentioned above and then train the resulting histograms with the linear SVM in order to classify the purple and the pink classes. Afterwards when our algorithm detects a plant which is not white we calculate the histogram as explained above on the large rectangle that surround the plant (with its corresponding mask) and then let the SVM do the classification. We wish to mention that we do some "correcting" to the SVM's classification, e.g. when it classifies a plant to be pink but the plant has a small percentage of pink pixels then we decided that the plant will be purple and not pink as the SVM classified. Although the mentioned example is a quite rare, but it happens here and there, and the "correction" makes the algorithm more robust.

2.9 The final result

The results are as follows:

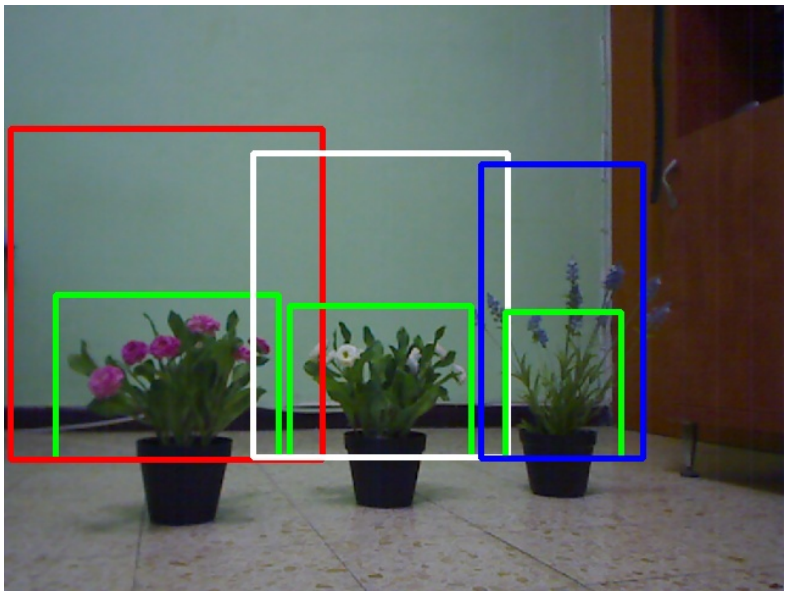


Figure 24: The detected plants

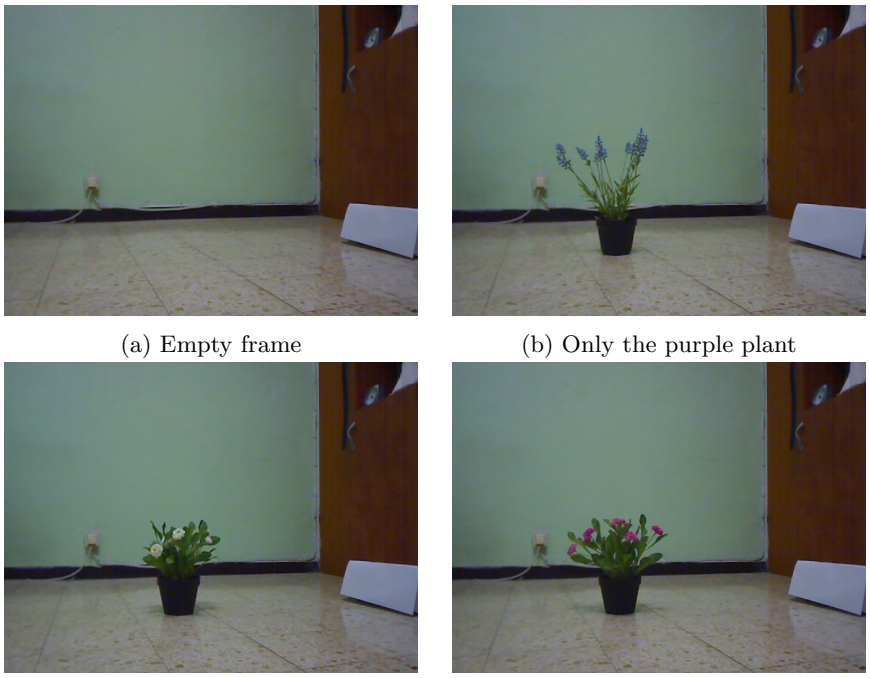
We would like to emphasize that the histograms and the colors percentages are calculated over the big rectangles and not over the green rectangles.

3 Research

In the following subsections we will show the tests we did on our algorithm, in these tests we wish to test the robustness and stability of our algorithm.

3.1 Detection and Recognition Stability

We would like to know how stable our algorithm is. In order to do that we placed the plants, one by one, in front of the camera and recored in a row for around 1.3 minutes the frames the camera captured. We sampled a BGR frame and depth frame every half a second. The first quarter of frames were with no plants at all, the second were with only the purple plant, the third with only the white and the fourth with only the pink plant. We took two of those datasets - one with a stable illumination, and one with a more dynamic illumination, and then we have created a confusion matrix of what was detected on each frame:



(a) Empty frame (b) Only the purple plant

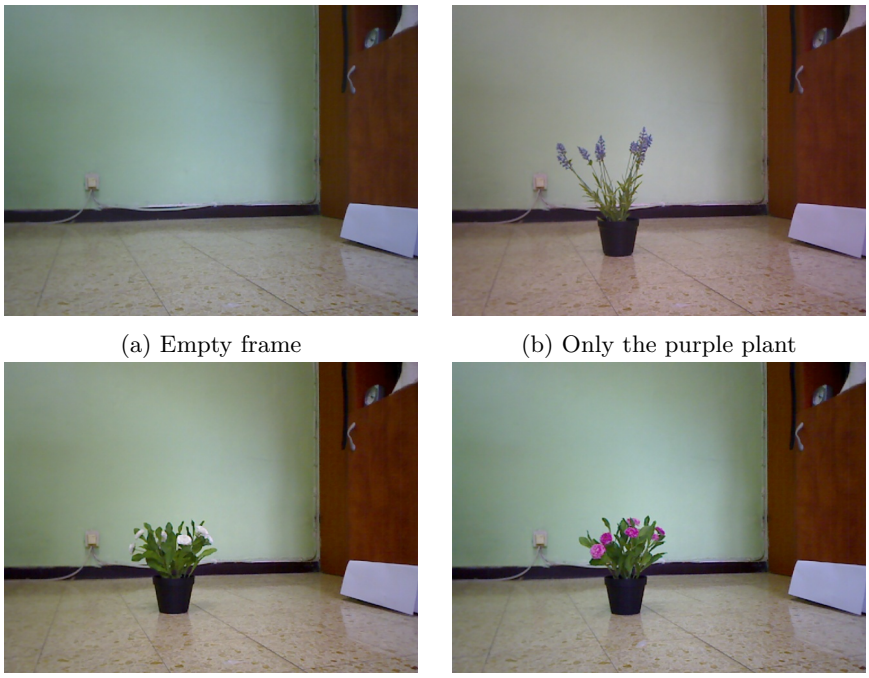
(c) Only the white plant (d) Only the pink plant

Figure 25: Examples of the dataset with the stable illumination

The results in a confusion matrix were:

		Detected Class				
		No plant	**Purple**	**White**	**Pink**	**Sensitivity**
Expected Class	**No plant**	0.2	0	0	0	**1**
	Purple	0	0.26667	0.008333	0	**0.9697**
	White	0	0	0.26833	0	**1**
	Pink	0	0.0016667	0	0.255	**0.99351**
	Precision	**1**	**0.99379**	**0.96988**	**1**	**Accuracy: 0.99**

Table 1: confusion matrix on stable illumination



(a) Empty frame (b) Only the purple plant

(c) Only the white plant (d) Only the pink plant

Figure 26: Examples of the dataset with the more dynamic illumination

The graph was:

Expected/Detected	**No plant**	**Purple**	**White**	**Pink**	**More than one**	**Sensitivity**
No plant	0.24619	0	0	0	0	**1**
Purple	0	0.25272	0	0	0.0087146	**0.96667**
White	0	0	0.23965	0	0.0043573	**0.98214**
Pink	0	0	0	0.24837	0	**1**
More than one	0	0	0	0	0	**1**
Precision	**1**	**1**	**1**	**1**	**0**	**Accuracy: 0.98693**

Table 2: Confusion matrix on unstable illumination

It can be concluded that the algorithm becomes less accurate when the illumination becomes more unstable. In addition, the algorithm is quite stable and has relatively few false positives.

3.2 Different Positions

We did three experiments, each experiment was on a different plant. In each experiment we took a series of photos on 6 different locations, i.e. we have

placed the plant in 6 different locations, each 20 cm from the other such that the fartest location was 1.2 meters from the camera and the closest location was 20 cm from the camera. On each location we turned around the plant 8 times (on steps of 45 degrees) and took at least 3 photos of each rotation, ultimately we had around 32 photos for each location per a plant and a total of 200 photos for each plant. For example the 6 locations were:

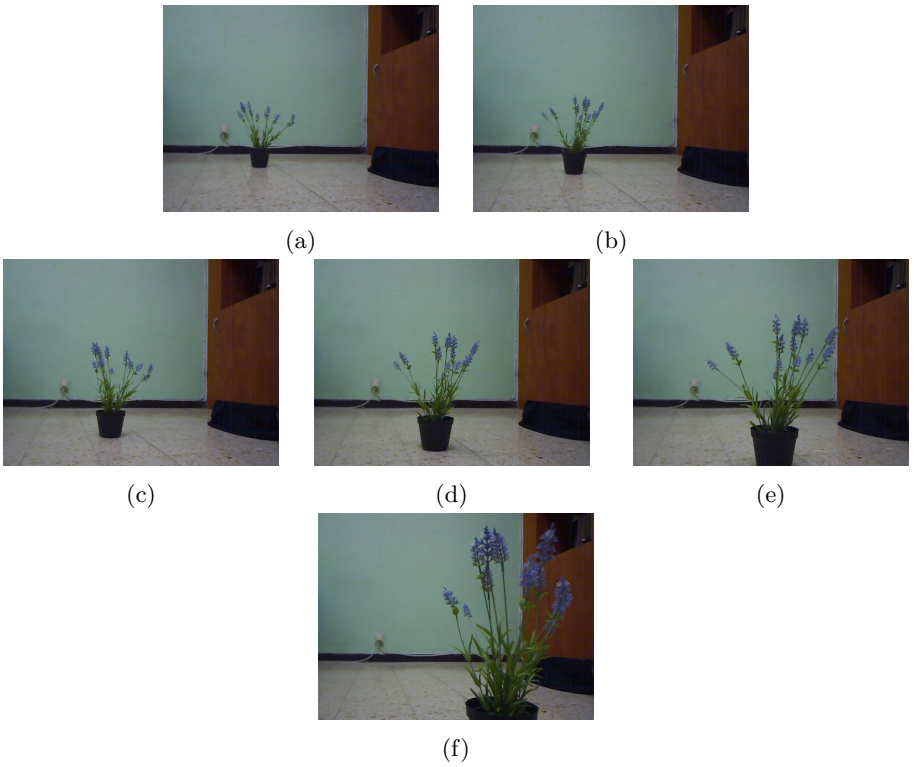


Figure 27: Examples of the six different locations distances from the camera

And an example of the rotation in each location:

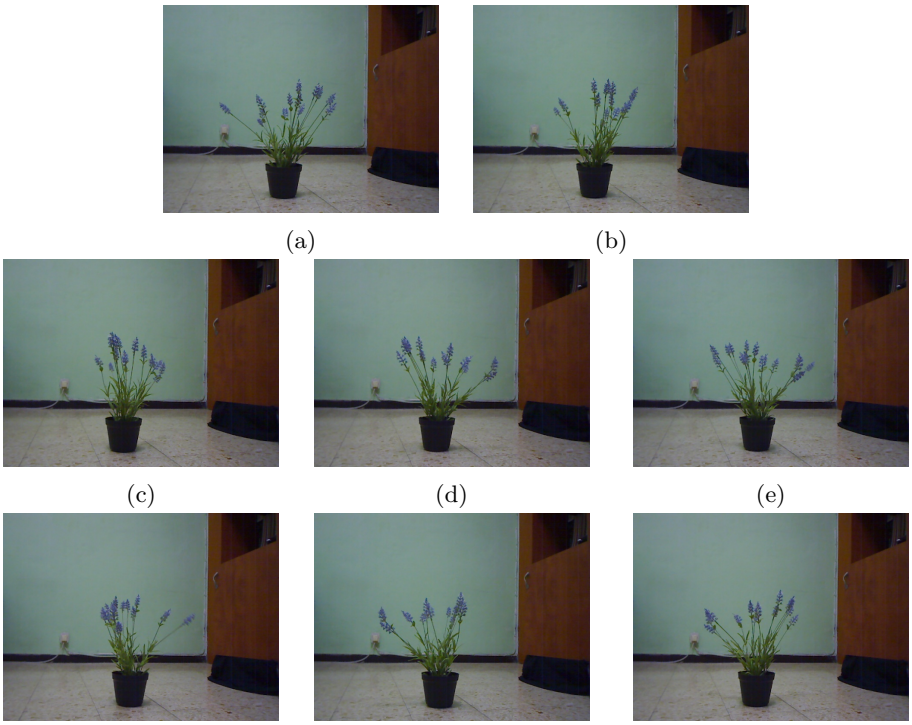


Figure 28: Examples of the eight different rotations in each location

We took those three datasets and created graphs, the results are:
(Note: the first frames are the furthest locations and the latest frames are the closest locations to the camera.)

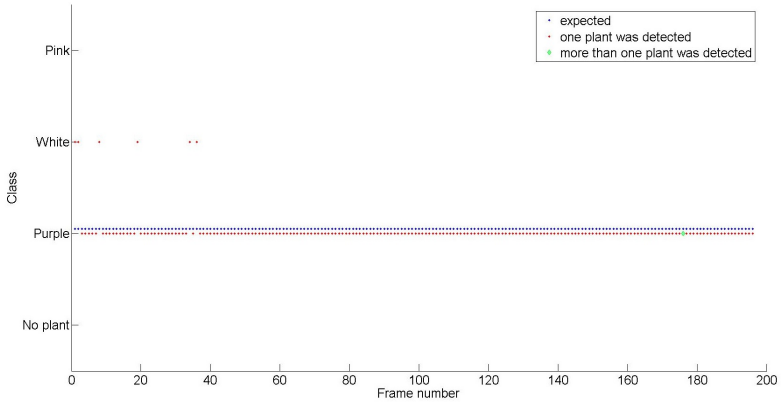


Figure 29: Graph of the purple plant

The same graph in a table:

Detected Class				
No plant	**Purple**	**White**	**Pink**	**More than one**
0	0.96429	0.030612	0	0.005102

Table 3: Table of the purple plant

As expected, we can see that the main errors were in the furthest locations, we assume that the reason for this is the fact that the purple flowers are not detected so well in the depth frame when the plant is far from the camera.

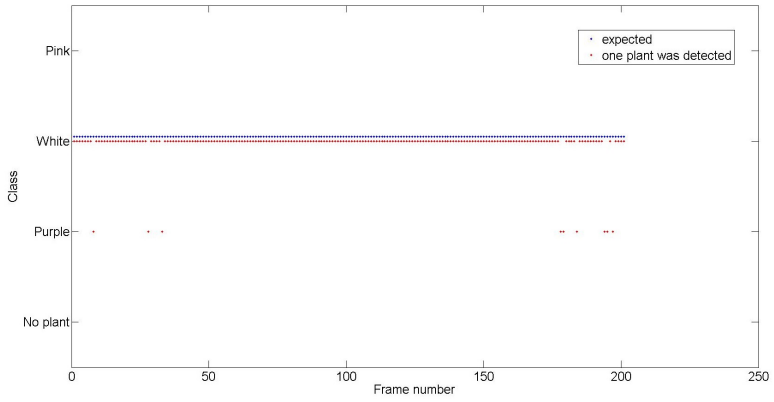


Figure 30: Graph of the white plant

The same graph in a table:

Detected Class			
No plant	**Purple**	**White**	**Pink**
0	0.044776	0.955224	0

Table 4: Table of the white plant

The errors accured when we were too close or too far from the camera, maybe the reason for this is because the white color tends to be more 'purplish' in those situations.

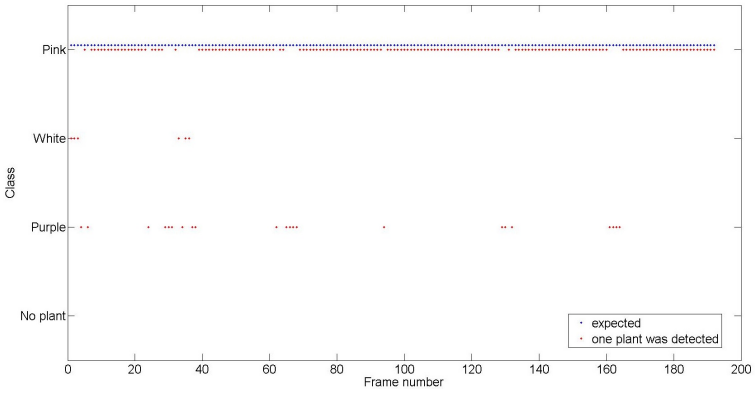


Figure 31: Graph of the pink plant

The same graph in a table:

Detected Class			
No plant	**Purple**	**White**	**Pink**
0	0.11458	0.03125	0.85417

Table 5: Table of the pink plant

It is evident that our proposed algorithm prevailed the treacherous test sets. Although mistaken sometimes, its accuracy is quite impressive.

4 Future Work

- One of the main obstacles encountered during our research was the sensativity to illumination. The hsv colorspace was worse than we've initially imagined and a more robust illumination invariant algorithm could be proposed.
- The purpose of our work was to classify the three flowers presented, it is possible to add more types of flowers to the svm. The flower detection framework remains the same since it relays on the green color of the plants.
- A much easier approach to classifying flowers in a greenhouse would be the following:
  Put a sticker on each flower which corresponds to the appropriate type, the classification becomes trivial and an added bonus is easy pose estimation.

References

[1] `http://en.wikipedia.org/wiki/Robot_Operating_System`.

[2] http://www.turtlebot.com/.

[3] http://opencv.org/.

[4] http://www.openni.org/.

Appendix

During our development of Project Flower we have encountered a severe problem. In this appendix we will describe the problem and its solution (or rather workaround).

Problem

Our program, as part of our suggested algorithm pipeline, showed the images sent from the Kinect. After several frames, a flickering of the images shown occurred and eventually no images were broadcasted.

Explanation

We will try to explain the cause of the problem since we can only assume what the real problem is.

After the Kinect sends frames to our application, the frames are stored in a buffer. This buffer is probably of finite size. Since the computer, which processes the frames, is slow, the callbacks to the frames received are delayed. Eventually, too many frames are stored in the buffer and as result, the buffer overruns.

Solution

We have tried reducing the frequency of the frames sent using a ROS command. However, this solution only prolonged the runtime of our program and has not resolved the issue. We have therefore looked for another solution and found the one true solution to the problem. In the main function of our algorithm, every k-th iteration we have unsubscribed to the Kinect topic and then subscribed once again.

The code, which solved the problem, is attached below:

```
int main(int argc, char** argv)
{
 // initialization

 // main loop
 for(;;){
        if(iterations % k == 0)
        {
                iterations = 0;

                // unsubscribe
                delete ic;
                ic = NULL;

                // resubscribe
                ic = new ImageConverter();
        }
        iterations++;

 // process frame
}
```