Ben-Gurion University of the Negev

Faculty of Natural Science

Department of Computer Science

Topics in Computational Vision
202.1.1111

Final Project:
Detect, Reach and Grab a Disposable cup by Komodo Robot

Students: Yaniv Okev ID: 301551842

Iftah Grosvirt ID: 200305639

Yaniv Golan ID: 301508180

Aviv Rachmany ID: 032588733

Michael Petrovitsky ID: 319296927

Date: 27/08/2015

1. Introduction

The general goal of our project is to implement "Detect and Reach" algorithm, applied on Komodo robot (www.robotican.net) using the increasing popular ROS (www.ros.org) — the robot's operating system. Our algorithm uses Komodo's many sensors to scan an open space and spot a disposable cup, reaching it while avoiding occurred obstacles, and finally – grabbing the cup.

While Programming in Python and interfacing OpenCV's libraries with ROS, this project provided us a hands-on and practical experience with issues related to theoretical or applied topics in computational vision and/or human perception.

2. Komodo's Robot background

The Komodo robot is composed of three separate modules which can work independently and are interconnected through mechanical and electrical interfaces. The three modules are as follows**:** **Rover** module, **Sensors** module, and **Arm** module.
The Rover module contains the chassis, the wheels motors and drive units, a controller, the drive battery (the largest one) and a remote control unit. This module can be independently operated as a remote controlled vehicle
The Sensors module contains a computer, Wi-Fi communication and a set of sensors. The sensors are: Asus XTION, Laser scanner, RGB camera, IMU (3 axis accelerometer, 3 axis gyro, 3 axis magnetometer), GPS receiver, and 3 ultrasonic and range sensors. This module can be independently operated as a sensing and computation unit. The Sensors and Rover modules are connected to through internal connector, and 4 screws.
The Arm module contains a 5 DOF (Degrees of Freedom) with two fingers gripper and a battery (the smaller one- the same as the one of the Sensors module). An RGB camera is mounted on the wrist, forward looking on the gripper. This module is connected to the Sensors module through 6 screws and two USB connectors, one for the motors, and one for the camera.

The Komodo Robot Features:

- Dimensions: 59x48x48 cm (23.2” x 19” x 19”)
- Fully ROS compatible http://wiki.ros.org/ric
- 2 x 250W motors
- High resolution encoders (2048 counts per revolution) for closed loop control
- Maximum continuous operation time of 3 hours
- Sensors: IMU, Compass, GPS, camera, RGB-D Camera, IR and Ultrasonic range sensors, Laser scanner, torque and position feedback of the arm joints and gripper
- Arm: 80 cm (31.5 in) long, payload of 0.4Kg (0.9 lb), with camera

3. OpenCV's Haar-Classifier training

In order to recognize something of our choice (something that is not a face – OpenCV's built-in feature) we need a "cascade classifier for Haar features" to point OpenCV at. A cascade classifier basically tells OpenCV what to look for in images. We need a cascade classifier that tells OpenCV how to recognize our disposable cup.

The following instructions are heavily based on Naotoshi Seo's immensely helpful notes on OpenCV haartraining and make use of his scripts and resources he released under the MIT license.

In order to train our own classifier we need samples, which means we need a lot of images that show the object we want to detect, in our work – the cup (positive sample), and even more images without the object (negative sample).

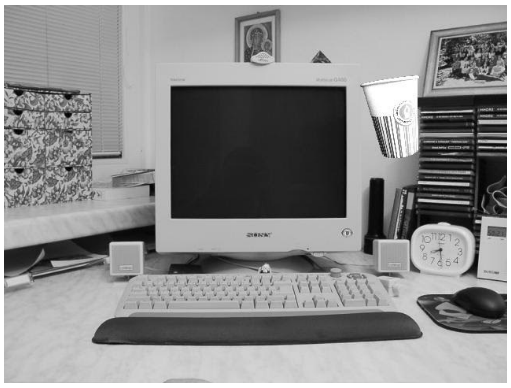
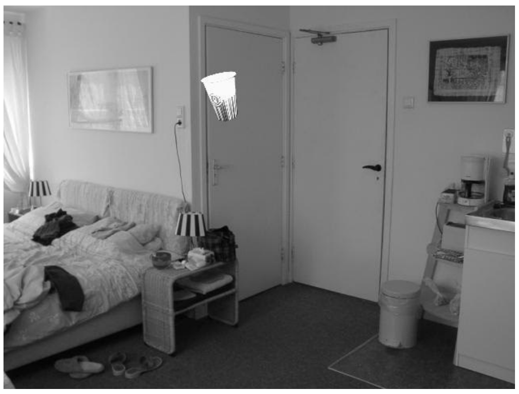
We took about 600 different photos of our BGU-cup and once we have them, we needed to crop them so that only our desired object is visible. We also kept an eye on the ratios of the cropped images, so they shouldn't differ that much.

Since our object is pretty white, we used a black background, and took high-contrast pictures in order to make the next step easier. Also, the pictures don't have to be large because OpenCV will shrink them anyway: ours were 720×1280.

Now we need the negative images, the ones that don't show a cup. In the best case, if we were to train a highly accurate classifier, we would have a lot of negative images that look exactly like the positive ones, except that they don't contain the object we want to recognize.

We've gathered about 2900 negative images that don't contain any cups in them.

Once we had the images, we've processed a script that generated 5 images(for each positive image) by placing a slightly rotated and slightly brighter/darker version of any of our positive images on top of a randomly selected negative image. And because we used a black background when we took our pictures, we specified black as the *background color* making the black on the cropped image transparent, giving us 5 images like these:

If we had 600 pictures, running the previous step on each of them would have produced 3000 pictures of cups floating in random places. What we have to do now is collect all of them into a single .vec file before we can run the training utility. We've done that by using a tool OpenCV gives us: opencv_createsamples. This tool offers several options as to how generate samples out of input images and gives us a *.vec file which we can then use to train our classifier.

opencv_createsamples generates a large number of positive samples from our positive images, by applying transformations and distortions.

The next thing we need to do is to merge the *.vec files we now have in the samples directory. In order to do so, we need to get a list of them and then use Naotoshi Seo’s mergevec.cpp tool.

We can now use the resulting our merged .vec file to start the training of our classifier.

TRAINING THE CLASSIFIER

OpenCV offers two different applications for training a Haar classifier: opencv_haartraining and opencv_traincascade. We are going to use opencv_traincascade since it allows the training process to be multi-threaded, reducing the time it takes to finish, and is compatible with the newer OpenCV 2.x API.

We need to point opencv_traincascade at our positive samples and negative images, and specifying the other parameters (width, height, number of neg/pos samples) tell it to write its output into the classifier directory of our repository.

When the process is finished we’ll find a file called classifier.xml in the classifier directory. This is the one, this is our classifier we can now use to detect cups with OpenCV!

4. Program's Flow Control – Program's Nodes and Psuedo Algorithms

Nodes :

- **Detector**

The "**Detector**" node gets an input image and returns a rectangle (x,y,h,w) for each cup in the image.

The detection uses the OpenCV detectMultiScale function with the cascade classifier we've trained.

Pseudo code:

detector(ROS_image):

```
        CONVERT ROS image to a openCV image
        CONVERT the openCV image to grayscale
        CALL  detectMultiScale with the cascade classifier on the grayscale image
        RETURN list of detected cups
```

- **Control**

The **"Control"** node gets an image from the robot's camera, detects cups in it using the detector node's service, and sends the cup's position to the drive node.

Pseudo code:

RGB_image_callback(RGB_image):

```
    CALL detector node with RGB_image
    IF cup is detected THEN:
        SET X,Y to be the center of the cup's rectangle
        SET deltaX,deltaY to be the offset of X,Y from the center of the image
        SET Z to be the depth at X,Y
        SEND deltaX,deltaY,Z to the Drive node
```

depth_image_callback(depth_image):

```
    SET depth as depth_image
```

- **Drive**

The "**Drive**" node gets cup's position from the control node and attempts to drive towards it while avoiding obstacles. If the cup is within reach of the robot's arm, it stops and sends the position to the Move Arm node to initiate the grabbing sequence.

Pseudo code:

```
Laser_scan_callback(distance_array):
    SET obstacleDistance as the minimum of distance_array
    IF obstacleDistance<MIN_DISTANCE THEN:
        SET canMove to false
        SET forwardVelocity as 0
    ELSE:
        SET canMove to true
        SET forwardVelocity as (obstacleDistance – MIN_DISTANCE)*MAX_VELOCITY
        SET forwardVelocity as the maximum of forwardVelocity and MAX_VELOCITY

Cup_Position_callback(deltaX,deltaY,z):
    IF time passed from last callback is greater than UPDATE_TIME THEN:
        SET cupDetected to true
        IF MIN_REACH < z < MAX_REACH THEN:
            SET canMove to false
        IF canMove  THEN:
            IF the absolute value of deltaX is smaller than FORWARD_ANGLE THEN:
                SET driveForward as true
            ELSE:
                SET driveForward as false
                CALL turnByAngle with deltaX
        ELSE:
            IF MIN_REACH < z < MAX_REACH and !grabMode THEN:
                SET grabMode to true
                SEND deltaX,deltaY,z to Move_Arm node

Drive main():
    SET cupDetected to false
    WHILE NOT cupDetected:
        IF time passed from initialization is greater than WAIT_TO_SCAN:
            SET scanningMode to true
            CALL scanning
    Set scanningMode to false
    WHILE node is not shutdown:
        IF driveForward and canMove:
            SEND forwardVelocity as linear velocity to /cmd_vel
        ELSE:
            SEND 0 as linear velocity to /cmd_vel
```

- **Move Arm**

The "**Move Arm**" node gets the cup position, calculates the required arm joint angles in order to reach it, sends the angles to the robot's arm actuators and picks it up.

Pseudo code:

```
Cup_Position_callback(deltaX,deltaY,z):
    CALCULATE joint angles according to deltaX,deltaY,z
    SEND joint angles to arm actuators
```

Calculating the angles:

In order to pick the cup we need to be able to maneuver the arm to desired points within its reach, based on the data given from the camera: normalized horizontal and vertical offsets from the center of the image, $\boldsymbol{deltaX}$ and $\boldsymbol{deltaY}$, and the depth, $\boldsymbol{z}$**.**

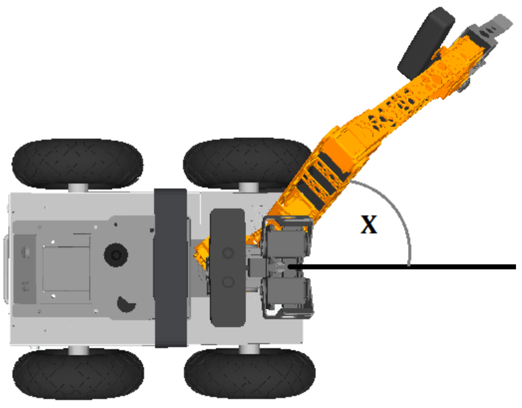
In order to reach a desired point with the arm we need to calculate the required joint angles. The arm is built on a rotating base and includes a shoulder, two elbows and a wrist. To make the calculation easier we chose to keep the first elbow straight. By doing so, all the joints are on the same plane, perpendicular to the ground.

First, we need to find out the horizontal and vertical angles from the camera to the cup. We can do this by multiplying the normalized offsets from center, $\boldsymbol{deltaX}$ and $\boldsymbol{deltaY}$**,** with the corresponding field of view angle:

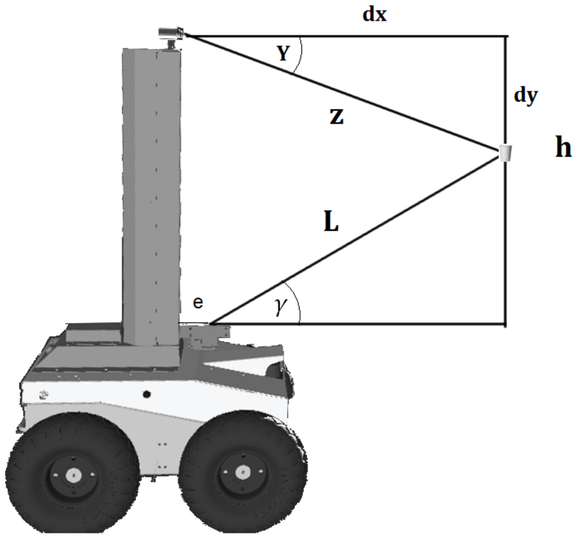
$$\boldsymbol{X = deltaX \cdot hFOV}$$

$$\boldsymbol{Y = deltaY \cdot vFOV}$$

Where $\boldsymbol{hFOV}$ is the horizontal field of view of the camera and $\boldsymbol{vFOV}$ is the vertical field of view of the camera.



We rotate the base of the arm horizontally according to the horizontal angle of the cup, **X**. Now, the problem is simplified into a 2D trigonometric calculation, since the shoulder, elbow and cup are coplanar.



h is vertical displacement of the camera from the arm's base, ***e*** is the horizontal displacement from the camera to the arm's base, is the distance from the camera to the cup.

We calculate the distance and angle from the arm's base to the cup, $\boldsymbol{L}$ *and* $\boldsymbol{\gamma}$, respectively.

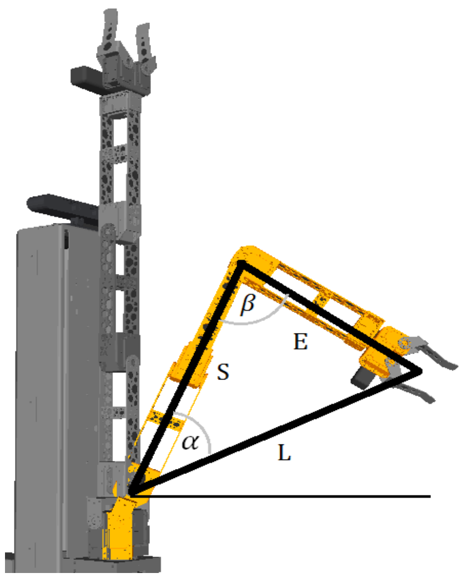
$$\boldsymbol{dx = z \cdot cosY}$$

$$\boldsymbol{dy = z \cdot sinY}$$

$$\boldsymbol{L = \sqrt{(dx - e)^2 + (h - dy)^2}}$$

$$\boldsymbol{\gamma = \arctan\frac{h - dy}{dx - e}}$$

The remaining calculation is solving an SSS triangle for its angles:



$$\boldsymbol{\alpha = \arccos\left(\frac{S^2+L^2-E^2}{2SL}\right)}$$

$$\boldsymbol{\beta = \arccos(\frac{S^2 + E^2 - L^2}{2SE})}$$

Where **S** is the shoulder length, **E** is the elbow length and **L** is the previously calculated distance from arm's base.

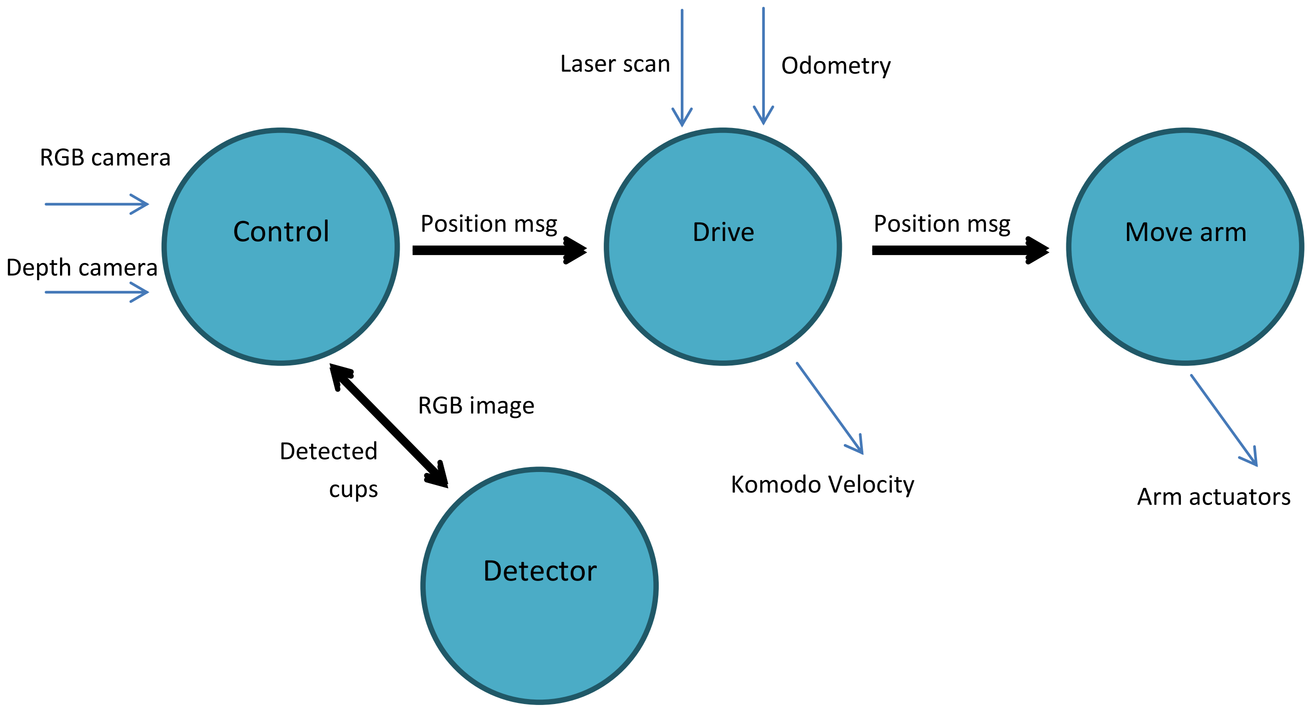
To conclude:

The base rotation angle is: $\boldsymbol{X}$

The shoulder angle is $\boldsymbol{\gamma + \alpha = \arctan\frac{h-dy}{dx-e} + \arccos\left(\frac{S^2+L^2-E^2}{2SL}\right)}$

Elbow angle is $\boldsymbol{\beta = \arccos(\frac{S^2+E^2-L^2}{2SE})}$.

<u>The entire Program Flow:</u>



5. Statistics and Indicators:

Training Stats:

# Positive Images: 600
# Positive Samples: 3000
# Negative Images: 3000
Training type: HAAR Cascade
Training stages: 27
Training estimated time: 4 days on
Maximum detecting range : 3 meters
Success rates(Detecting per frame) : 88%

Run Execution Stats:

Entering Scanning mode after initialization: 14 sec
Scanning each side before continuing: 7 sec
Minimum distance from obstacle: 0.35 meters
Maximum reaching distance: 0.83 meters
Average time from cup detection to grab mode: 53 seconds

6. Difficulties in reaching the goal :

- The RGB camera delays in its new information processing.
- The RGB camera's Depth feature works partially and delays a lot.
- The RGB camera's stand is broken, so computations that relies on it lacks accuracy.
- The Rover messages queue stores old messages even if the queue-size=1 which affects the intended motion.
- Once we went to work wirelessly with the robot, the wireless range worked partially and sometimes did not work at all for long periods.
- There is no extensive online documentation of operating the Komodo robot, and we had to learn it's functionality by trial and error.
- We could not get the robot to detect a large number of cups (Once we tried to train it to identify a number of cups it identified other objects as well as cups). So we decided to focus on identification of only one cup. That change provided us better success rates identifying that specific cup than partially identify any kind of cup.
- Struggled in installing new libraries on Komodo robot – for example "MoveIt" library, which finally we gave up on installing them.

7. Achievements :

- the cup, detect it, reaching it while avoiding the occurred obstacles and finally - grabbing the cup.
- While Programming in Python and interfacing OpenCV's libraries with ROS, this project provided us a hands-on and practical experience with issues related to theoretical or applied topics in computational vision and/or human perception.