OpenCV for Robotics

Unit 3: People-related OpenCV functions:

Introduction

- Summary -

Estimated time to completion: **4-5 hours**

In this course, we will talk about people-related OpenCV functions, and how you can use them with a robot in ROS, in which we will emphasize the following topics:

- Face Detection (Haar cascades)
- People Detection and Tracking. (HOG)

- End of Summary -

In this unit, we will talk about two classic object classifiers: Haar Cascades (we will use them to detect faces on an image) and the Histogram of Oriented Gradients (which we will use to detect people on images); both of them are based on machine learning algorithms. The first one uses AdaBoost (an algorithm formulated by Yoav Freund and Robert Schapire), and the second one uses Support Vector Machine algorithms (developed by Vladimir Vapnik).

But why will we see how to detect faces and people, and track them? Well, both in robotics and artificial vision, the relationship and interaction with humans have been constantly studied and applied for different purposes (such as security, recognition, monitoring, etc.). Both methods are classifiers of objects. The fact that with one we detect faces and with the other bodies does not mean that they are not used for other purposes. On the contrary, it is only a way to diversify and show that there are different methods for carrying out this relationship between robotics, computer vision, and humans.

The idea that runs under both algorithms is to extract some important features from the images that can describe something we want, for example, a car. The idea will be to extract some characteristics that a computer can interpret as a car, and then use these features to train a machine learning algorithm so that it can generalize the abstraction of this desired object.

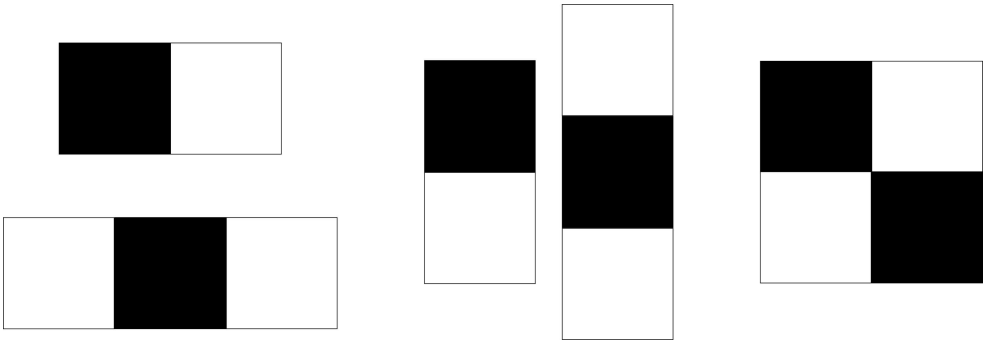
3.1 Face Detection

Haar Cascades

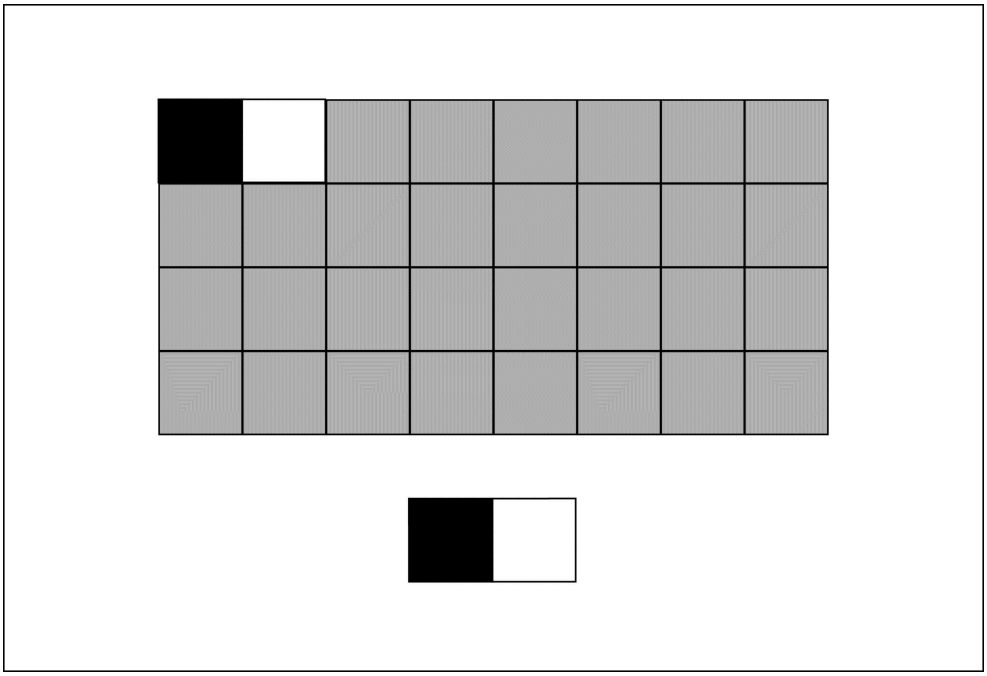
In the computer vision world, one of the most important tasks is classification, where you will want to detect certain objects in an image. For a long time, we always wanted a way to detect people, and nowadays, almost every single camera has a feature that detects people's faces pretty accurately. It´s even used in some social apps, like Snapchat, where an algorithm detects a face and extracts its characteristics.

A pretty good classifier algorithm that OpenCV has is the *Haar Cascade*, which works with Haar Wavelets to analyze image pixels into squares. This was proposed by Viola & Jones (https://ieeexplore.ieee.org/abstract/document/990517) in 2001. This classifier works just like convolution kernels, where we try to extract different features of the image with an *"integral image"*. This uses the AdaBoost learning algorithm, selecting a small amount of features from a large set of images.

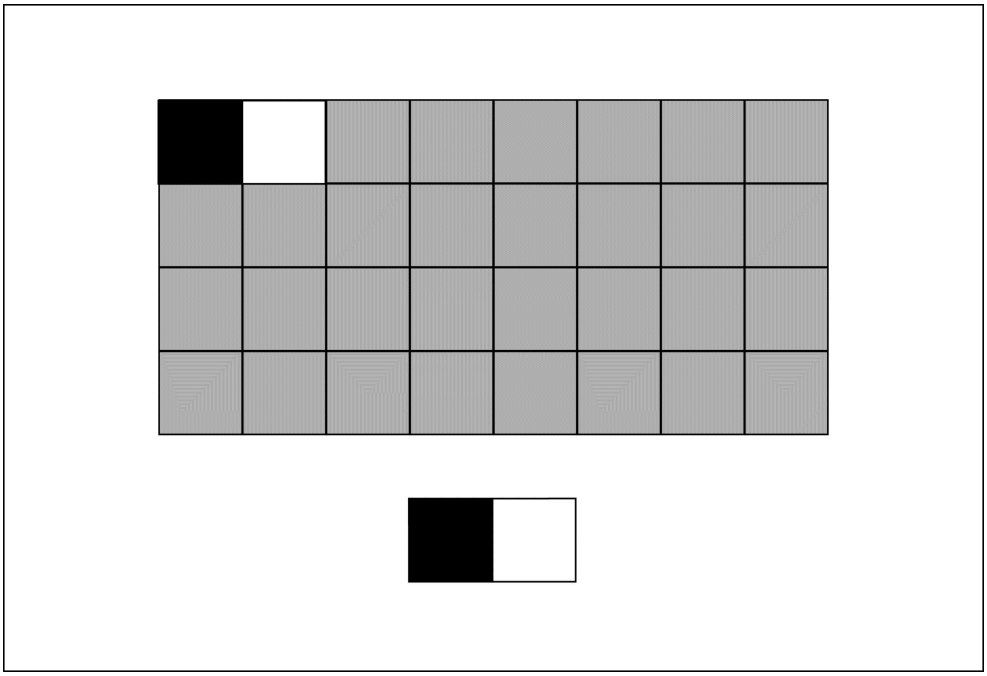
Detection objects with Haar work just like with any other object detection algorithm where you have 2 stages. The first one extracts important features from the images, and the second uses an algorithm to classify this information. Haar works with a group of filters that helps to extract the main features:



These are similar to the sobel operators. Haar uses these filters all over every image to train, and also uses different dimensions, so the algorithm can be scale invariant. This works using different scale filters, for example:



Each of these filters will travel all around the image:



With this algorithm, the number of feature descriptors is very large. This is why the authors use the integral image before, so it can have better descriptors and a faster processing time. This image is formed so that the intensity of the pixels is accumulated, and the intensity value of a pixel will be given by:

$$I_{x,y} = I_{x,y} + I_{x,y-1} + I_{x-1,y}$$

Once we have our Integral Image, we apply all the Haar Filters, and once we obtain the features, we use an AdaBoost (https://codesachin.wordpress.com/tag/adaboost/) algorithm as classifier.

For this training, what we really want is a big data set with positive images (images with the desired object) and negative images (images without the desired object). The main idea is that you can train this model to detect any object you want, similar to convolutional neural networks. In the code ahead, you will learn how to use a trained model to detect faces in images.

Look at the next example:

- Example 3.1 -

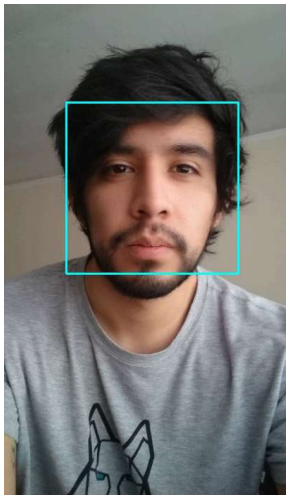
In [ ]:

```
import numpy as np
import cv2

face_cascade = cv2.CascadeClassifier('/home/user/catkin_ws/src/opencv_for_robc

img = cv2.imread('/home/user/catkin_ws/src/opencv_for_robotics_images/Unit_3/C
img_2 = cv2.imread('/home/user/catkin_ws/src/opencv_for_robotics_images/Unit_3

img = cv2.resize(img,(400,700))


gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
gray_2 = cv2.cvtColor(img_2, cv2.COLOR_BGR2GRAY)


ScaleFactor = 1.2

minNeighbors = 3

faces = face_cascade.detectMultiScale(gray, ScaleFactor, minNeighbors)
faces_2 = face_cascade.detectMultiScale(gray_2, ScaleFactor, minNeighbors)


for (x,y,w,h) in faces:

    cv2.rectangle(img,(x,y),(x+w,y+h),(255,255,0),2)
    roi = img[y:y+h, x:x+w]


for (x,y,w,h) in faces_2:
    cv2.rectangle(img_2,(x,y),(x+w,y+h),(255,255,0),2)
    roi = img_2[y:y+h, x:x+w]


cv2.imshow('Face',img)

cv2.imshow('Faces',img_2)

cv2.waitKey(0)
cv2.destroyAllWindows()
```

- End of Example 3.1 -

- Expected Behavior for Example 3.1 -

This is the **Face** image result.



And this is the **Faces** image result.

- End of Expected Behavior for Example 3.1 -

- Explanation for Example 3.1 -

But let's explain each part of this python code.

First, we need to import the libraries we will use and upload the xml classifier, and in this case, read the external images.

In [ ]:
```
import numpy as np
import cv2


face_cascade = cv2.CascadeClassifier('/home/user/catkin_ws/src/opencv_for_robc

img = cv2.imread('face.jpg')
img_2 = cv2.imread('many.jpg')

img = cv2.resize(img,(400,700))
```

Then we convert the image to grayscale.

In [ ]:
```
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
gray_2 = cv2.cvtColor(img_2, cv2.COLOR_BGR2GRAY)
```

The Scale Factor determines just how much the original image is going to be reduced in every scale.

In [ ]:
```
ScaleFactor = 1.2
```

Just like the name says, this will determine the number of neighbors to a higher value. The model will be more selective.

In [ ]:
```
minNeighbors = 3
```

We apply the cascades to our grayscaled images.

In [ ]:
```
faces = face_cascade.detectMultiScale(gray, ScaleFactor, minNeighbors)
faces_2 = face_cascade.detectMultiScale(gray_2, ScaleFactor, minNeighbors)
```

Once the algorithm finishes, we will extract the coordinates of the detections.

In [ ]:
```
for (x,y,w,h) in faces:

    cv2.rectangle(img,(x,y),(x+w,y+h),(255,255,0),2)
    roi = img[y:y+h, x:x+w]

for (x,y,w,h) in faces_2:
    cv2.rectangle(img_2,(x,y),(x+w,y+h),(255,255,0),2)
    roi = img_2[y:y+h, x:x+w]
```

For every coordinate, we will draw a rectangle in the original image.

In [ ]:
```
cv2.rectangle(img,(x,y),(x+w,y+h),(255,255,0),2)
```

This will be a cut of the area of interest.

In [ ]:
```
roi = img[y:y+h, x:x+w]
```

The same for the second image.

In [ ]:
```
for (x,y,w,h) in faces_2:
    cv2.rectangle(img_2,(x,y),(x+w,y+h),(255,255,0),2)
    roi = img_2[y:y+h, x:x+w]
```

And finally show the results.

In [ ]:
```
cv2.imshow('Face',img)
cv2.imshow('Faces',img_2)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

- End of the Explanation for Example 3.1 -

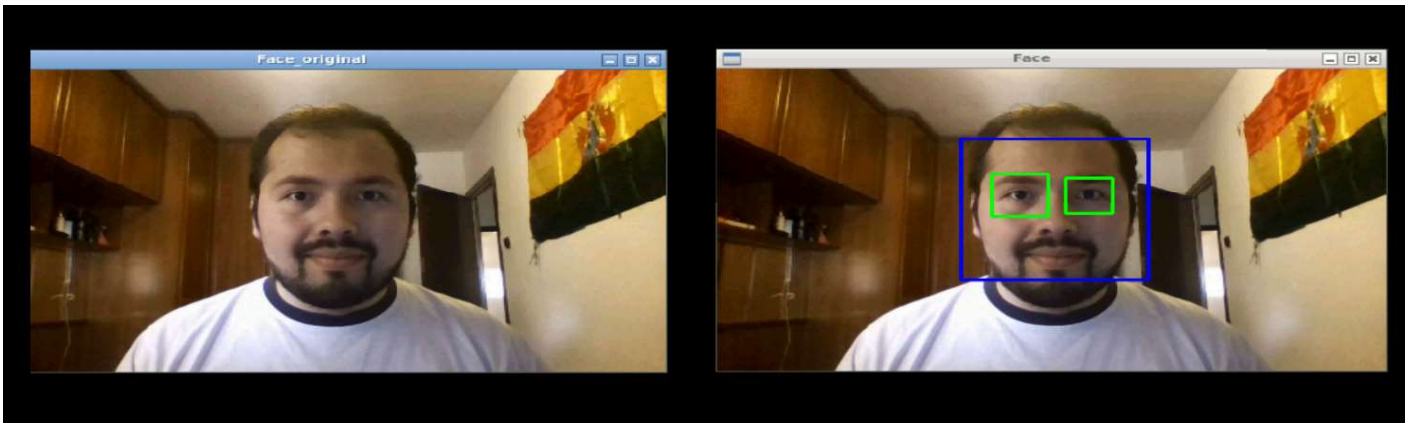
Now you have the explanation of how the code works. Let's apply this code with the same structure we saw in Unit 2.

Recognize a face in an image

- Exercise 3.1 -

- Create a new package in the catkin_ws and call it **unit3_exercises** with rospy as a dependency.
- Create inside this package a new folder and call it **haar_cascades**, and create two files and call them **frontalface.xml** (https://github.com/opencv/opencv/blob/master/data/haarcascades/haarcascade_frontalface_alt.xml) and **eye.xml** (https://github.com/opencv/opencv/blob/master/data/haarcascades/haarcascade_eye.xml). Paste the code of each one inside it.
- Inside the **src** folder, create a new file and call it **exercise_3.1.py**.
- In this file, you have to do the following steps:
  - Read the picture with this path **/home/user/catkin_ws/src/opencv_for_robotics_images/Unit_3/Course_images/chris.jpg**.
  - Resize it to a **500x300** image.
  - Apply the haar cascade of **frontalface** and **eye** detection. You can get an idea by looking at the prior example.
  - Show the result and the original image.
- Execute this program using rosrun command.

- Expected Behavior for Exercise 3.1 -



- End of Expected Behavior for Exercise 3.1 -

- Notes -

Don't forget to give permissions of execution to your python file with the command:

In [ ]:
```
chmod +x python_file
```

- End of Notes -

Good Job! But this algorithm should work with many faces, too. Let's try it.

- Exercise 3.2 -

Based on the previous exercise, you can upload one of your photos and see how you can detect yourself.

- Create a new folder inside your **unit3_exercises** package and call it **my_photo**.
- Here you can upload your photo. remember to use this path to load your photo into the code.
- Follow the same steps as in the previous exercise, but this time, the name of the file will be **exercise3_2.py**, and see the result of your detection code.

- End of Exercise 3.2 -

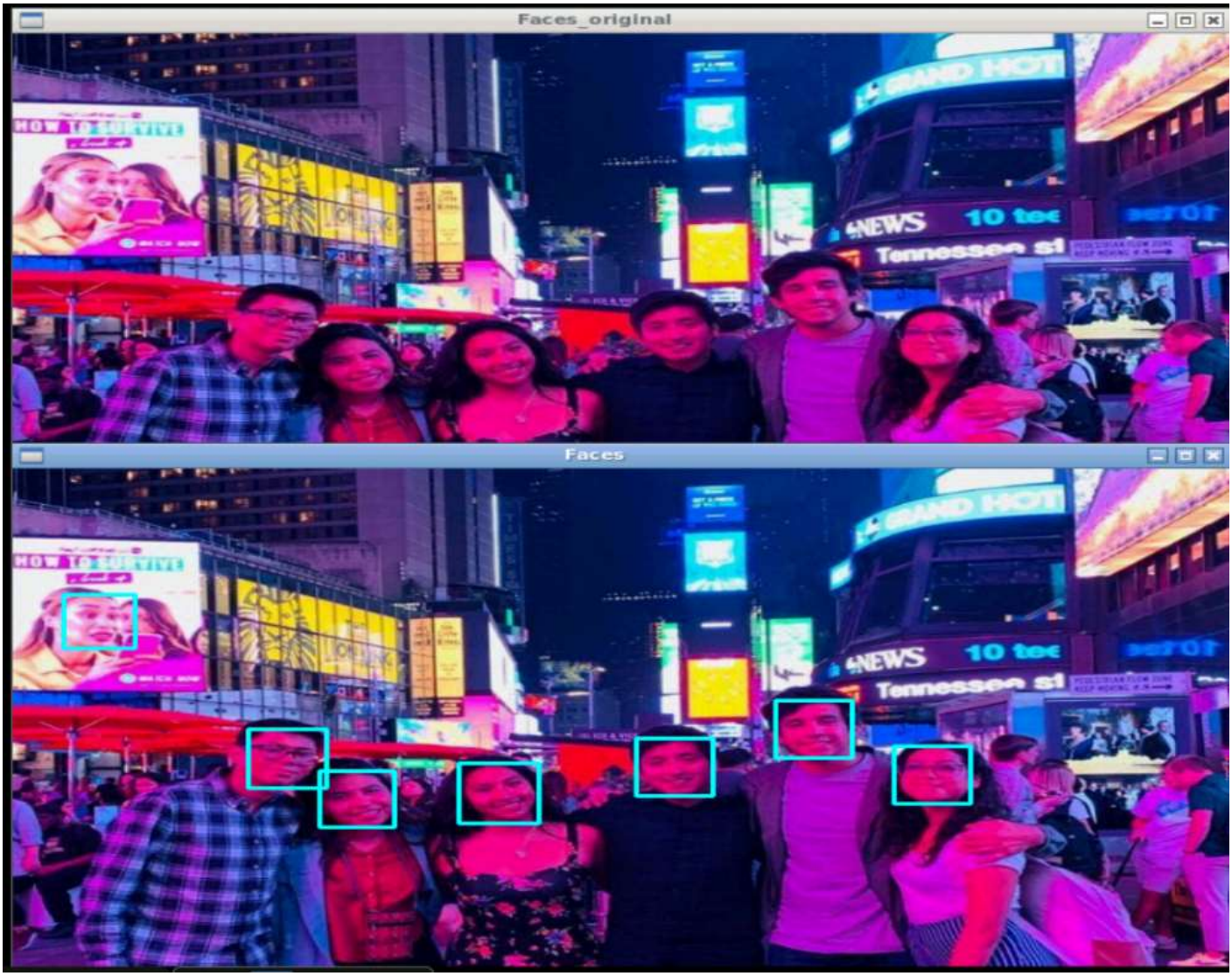
Recognize multiple faces in an image

- Exercise 3.3 -

- Inside the **src** folder of your package *unit3_exercises*, create a new file and call it **exercise_3.3.py**.
- In this file, you have to do the following steps:
  - Read the picture with this path: **/home/user/catkin_ws/src/opencv_for_robotics_images/Unit_3/Course_images/many.jpg**
  - Resize it to a **700x600** image.
  - Apply the haar cascade of **frontalface** detection. You can get an idea by looking at the previous example.
  - Show the result and the original image.
- Execute this program using **rosrun** command.

- End of Exercise 3.3 -

- Expected Behavior for Exercise 3.3 -



- End of Expected Behavior for Exercise 3.3 -

- Notes -

Don't forget to give permissions of execution to your python file with the command:

In [ ]:
```
chmod +x python_file
```

- End of Notes -

Awesome! But, these are external images. If you want to use a robot that works with ROS, well... wait for it... YEAH, now is the time to apply it! And you will do it, so pay attention to the next exercise because you will use and see the results!

Recognize a face in a simulation (Gazebo)

- Exercise 3.4 -

- Inside */catkin_ws/src/unit3_exercises/src*, create two new files and call them **flying_drone3.py** and **exercise3_4.py**.
- Inside the **flying_drone3.py**, copy the following code. It will make the drone fly to a position in order to recognize the person.

In [ ]:

```
#!/usr/bin/env python

import rospy
from geometry_msgs.msg import Twist

from sensor_msgs.msg import Range

class Flying():

    def __init__(self):
        self.hector_vel_publisher = rospy.Publisher('/cmd_vel', Twist, queue_s
        self.hector_sub = rospy.Subscriber("/sonar_height", Range, self.scan_c

        self.cmd = Twist()
        self.ctrl_c = False
        self.rate = rospy.Rate(10) # 10hz
        self.a = 0.0
        rospy.on_shutdown(self.shutdownhook)

    def scan_callback(self, msg):
        self.a = msg.range


    def shutdownhook(self):
        # works better than the rospy.is_shutdown()
        self.ctrl_c = True

    def move_drone(self):
        while not self.ctrl_c:

            if self.a < 1.8:
                self.cmd.linear.z = 0.12

            if self.a > 1.93:
                self.cmd.linear.z = 0.0

            if self.a < 0.39:
                self.cmd.linear.x = -1.3
            if self.a > 0.40 and self.a < 0.50:
                self.cmd.linear.x = 0.7
            if self.a > 0.51 and self.a < 0.60:
                self.cmd.linear.x = -0.2
            if self.a > 0.61 and self.a < 0.70:
                self.cmd.linear.x = 0.7
            if self.a > 0.71 and self.a < 0.80:
```

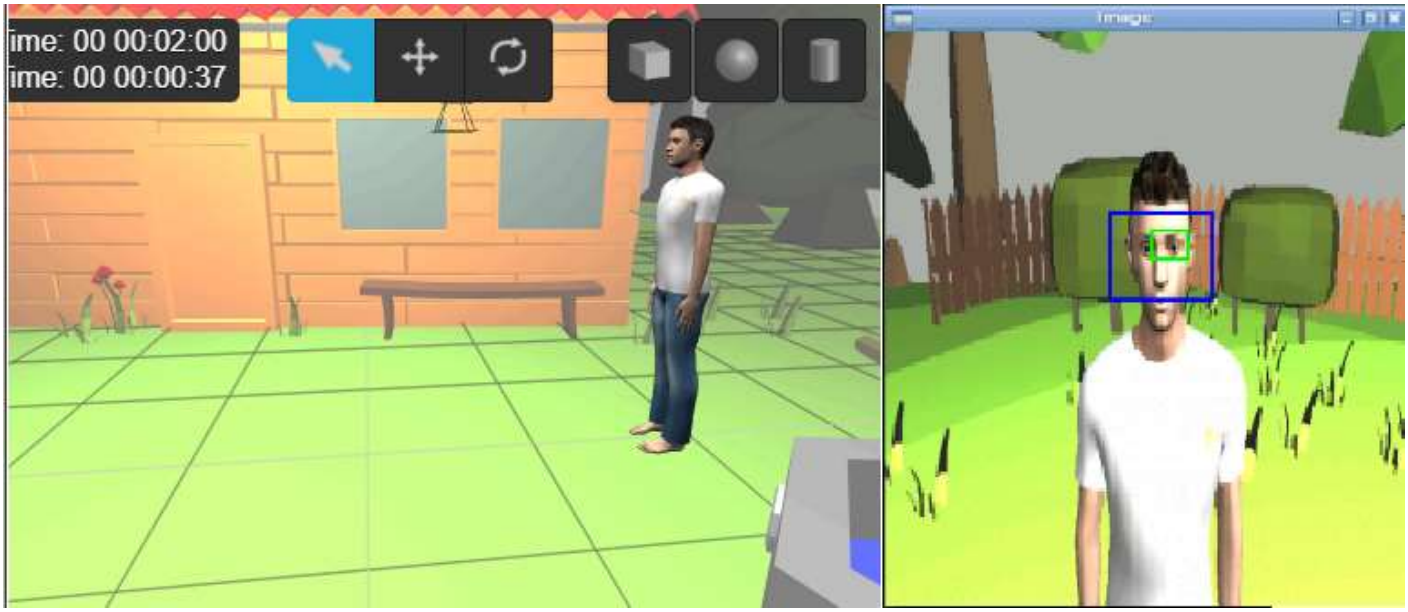
```
                self.cmd.linear.x = -0.2
            if self.a > 0.81 and self.a < 0.95:
                self.cmd.linear.x = 0.9
            if self.a > 0.95 and self.a < 1.00:
                self.cmd.linear.x = -0.2
            if self.a > 1.01 and self.a < 1.10:
                self.cmd.linear.x = 0.5
            if self.a > 1.11 and self.a < 1.20:
                self.cmd.linear.x = -0.2
            if self.a > 1.21 and self.a < 1.30:
                self.cmd.linear.x = 0.5
            if self.a > 1.31 and self.a < 1.40:
                self.cmd.linear.x = -0.34
            if self.a > 1.41 and self.a < 1.50:
                self.cmd.linear.x = 0.72
            if self.a > 1.51 and self.a < 1.60:
                self.cmd.linear.x = -0.29
            if self.a > 1.61 and self.a < 1.78:
                self.cmd.linear.x = 0.71
            if self.a > 1.78 and self.a < 1.98:
                self.cmd.linear.x = 0.08
            if self.a > 1.98 :
                self.cmd.linear.x = 0.0


            self.hector_vel_publisher.publish(self.cmd)




if __name__ == '__main__':
    rospy.init_node('hector_test', anonymous=True)
    hector_object = Flying()
    try:
        hector_object.move_drone()
    except rospy.ROSInterruptException:
        pass
```

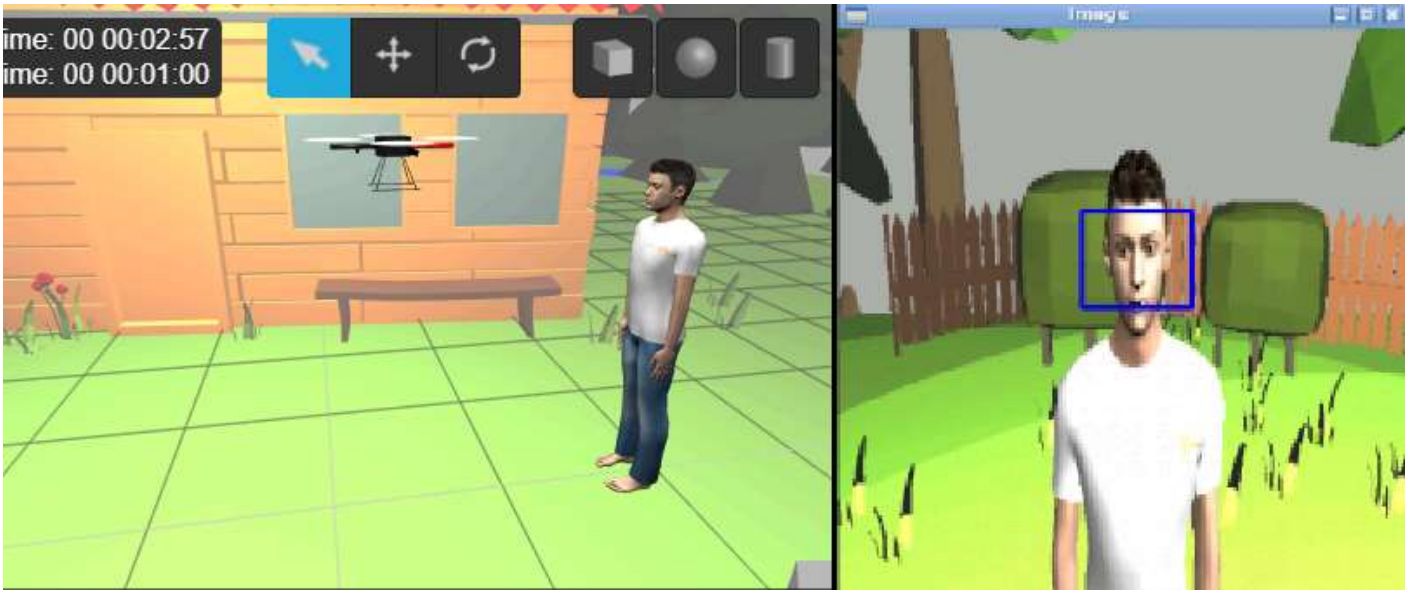
- Now, in **exercise3_4.py**, do the following steps:
  - Apply the haar cascade of frontalface and eye detection. You can get an idea by looking at example 3.1.
  - Resize the image to a 350x550. We resize the image only to reduce data processing.
  - Once a face is detected, take a picture and save it inside the unit3_exercises package. You can choose the name.
  - Show how to detect the face in real time.
- Create a folder inside the unit3_exercises package and call it **launch**.
- Create a new file inside the launch folder and call it exercise3_4.launch, in order to launch these files together.

- End of Exercise 3.4 -

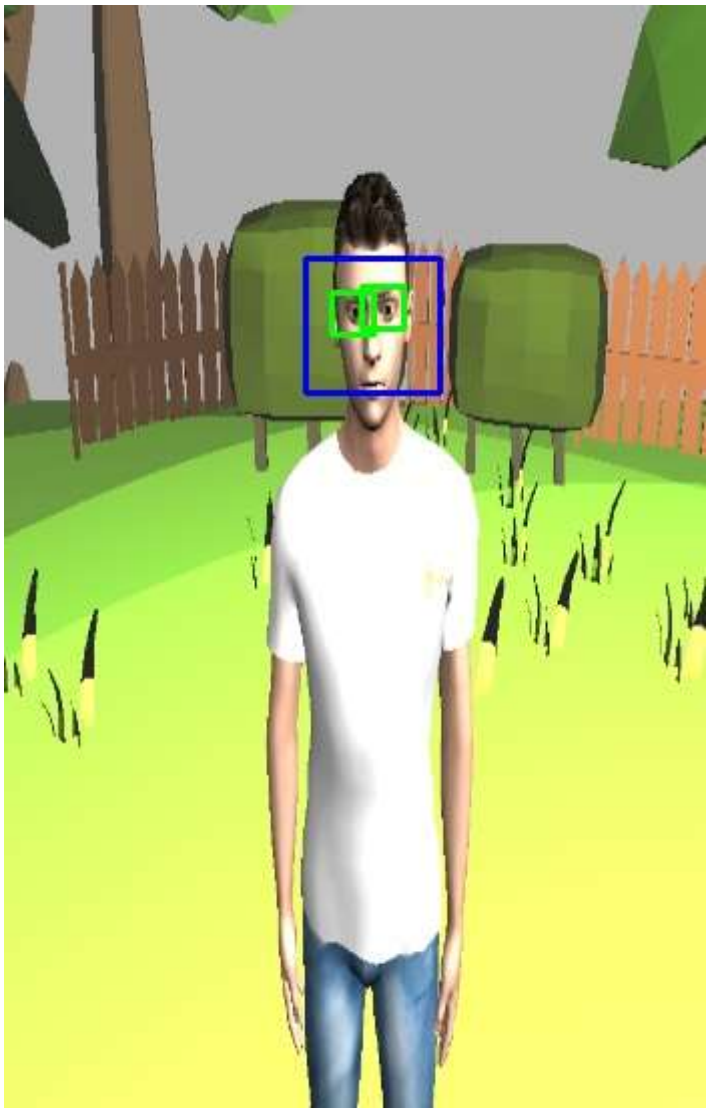
- Expected Behavior for Exercise 3.4 -

It works! You can detect the face even if the haar cascade is trained to find a **REAL face**, and this is a **Simulation**. We have to know that, so if you have results similar to this, you did a good job!





And the picture that you have to take should look like this one.



- End of Expected Behavior for Exercise 3.4 -

- Notes -

Don't forget to give permissions of execution to your python file with the command.

In [ ]:
```
chmod +x python_file
```

- End of Notes -

Tracking a face

But it is not detecting all the time. That's not fair. As we said, it's because it is a simulation. With a real camera and real images, it should detect all the time. Oh, if you don't trust us, well, let's see it! Now you will see this algorithm applied to a real video to see how it works with a real video taken from the drone.

- Example 3.2 -

Let's create another file and call it **example_video.py**.

Inside this new file copy is the following code.

In [ ]:

```
#!/usr/bin/env python

import rospy
from sensor_msgs.msg import Image
from cv_bridge import CvBridge, CvBridgeError
import cv2
import numpy as np


class LoadVideo(object):

    def __init__(self):
        self.ctrl_c = False

        self.bridge_object = CvBridge()

    def shutdownhook(self):
        # works better than the rospy.is_shutdown()
        self.ctrl_c = True

    def video_detection (self,):
        cap = cv2.VideoCapture("/home/user/catkin_ws/src/opencv_for_robotics_i
        face_cascade = cv2.CascadeClassifier('/home/user/catkin_ws/src/unit3_e
        eyes_cascade = cv2.CascadeClassifier('/home/user/catkin_ws/src/unit3_e
        ScaleFactor = 1.2


        minNeighbors = 3
        while not self.ctrl_c:
            ret, frame = cap.read()









            img_original = cv2.resize(frame,(300,200))
            img = cv2.resize(frame,(300,200))



            gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
```

```
            faces = face_cascade.detectMultiScale(gray, ScaleFactor, minNeight



            for (x,y,w,h) in faces:

                cv2.rectangle(img,(x,y),(x+w,y+h),(255,0,0),2)
                roi = img[y:y+h, x:x+w]

                eyes = eyes_cascade.detectMultiScale(roi)
                for (ex,ey,ew,eh) in eyes:
                    cv2.rectangle(roi, (ex,ey),(ex+ew,ey+eh),(0,255,0),2)




            cv2.imshow('Face_original',img_original)



            cv2.imshow('Face',img)



            cv2.waitKey(1)
        cap.release()







if __name__ == '__main__':
    rospy.init_node('load_video_node', anonymous=True)
    load_video_object = LoadVideo()
    try:
        load_video_object.video_detection()
        rospy.oncespin()
    except rospy.ROSInterruptException:
        pass

    cv2.destroyAllWindows()
```

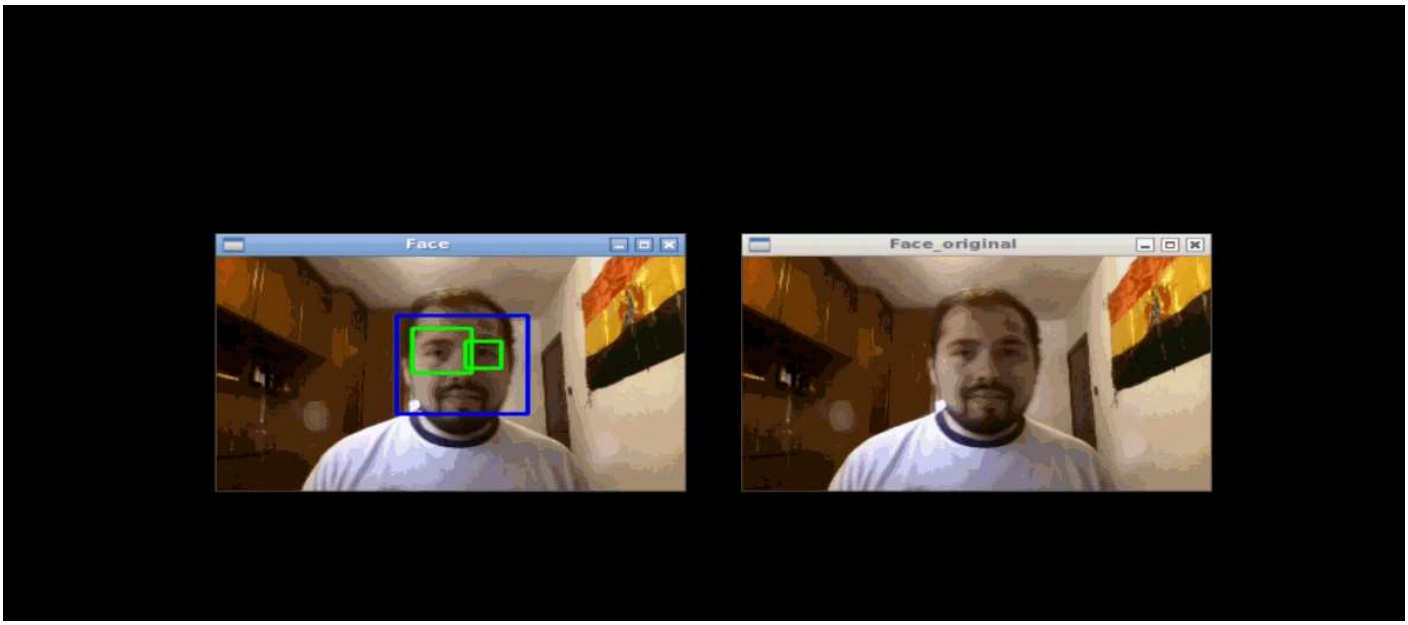
Don't forget to go to the directions of the file and change the permissions of execution.

Now just run it with rosrun command in the **Shell**.

In [ ]:
```
rosrun unit3_exercises example_video.py
```

- End of Example 3.2 -

- Expected Behavior for Example 3.2 -



- End of Expected Behavior for Example 3.2 -

Great! As you can see, it can work properly with real images. It will work pretty similarly in real time. But you know that you can not only detect faces. Let's see what else you can do in this unit.

3.4 People detection and tracking

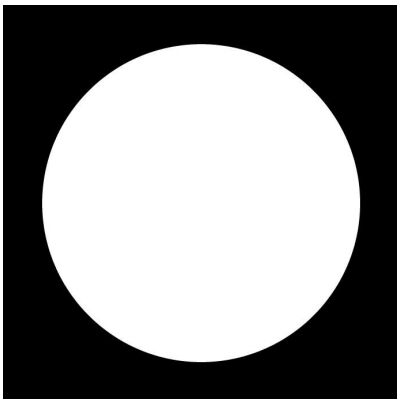
Histogram of Oriented Gradients (HOG)

HOG is one of the most famous descriptor features, which is based on the gradient orientations of images. Do you remember the chapter where we talked about sobel kernels to highlight borders? Well, the main idea of this algorithm is to use these gradients with the orientations to create histrogram boxes of the images, using these histograms as descriptors to train a support vector machine. We will go through this, step by step, so you can understand the functioning of this algorithm.

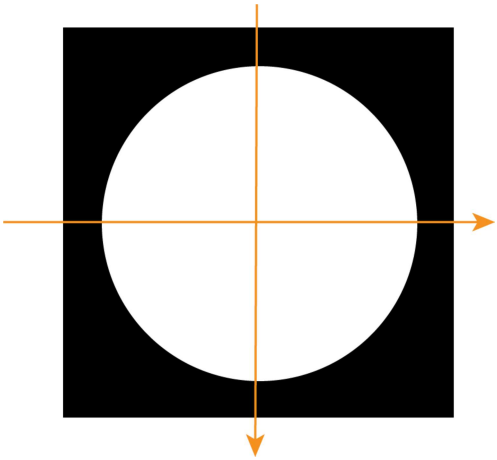
Gradient calculation

So for this part, we will make use of what we have learned in the first chapter, as we already know if we convolve an image with the sobel kernels, we will enhance the borders of the images (this is because the edges denote the gradients of the images). In the parts of the image where the color is almost the same, the gradient will tend to 0. Meanwhile, if there is a big change in the gradient, the magnitude will be higher.

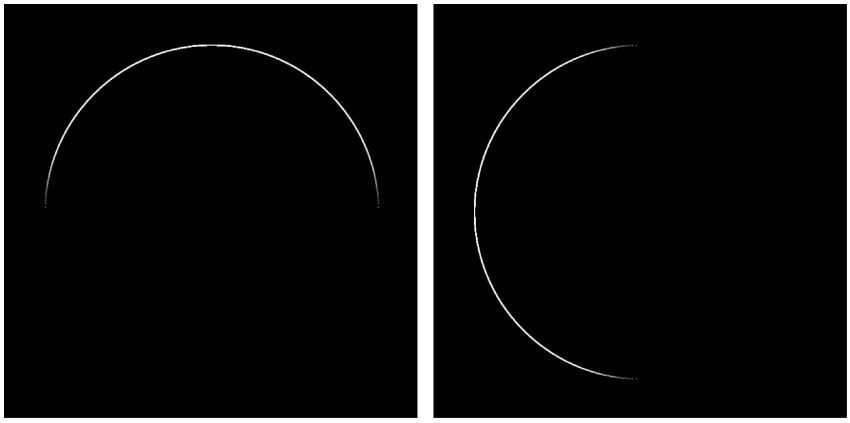
Let's suppose the next image:



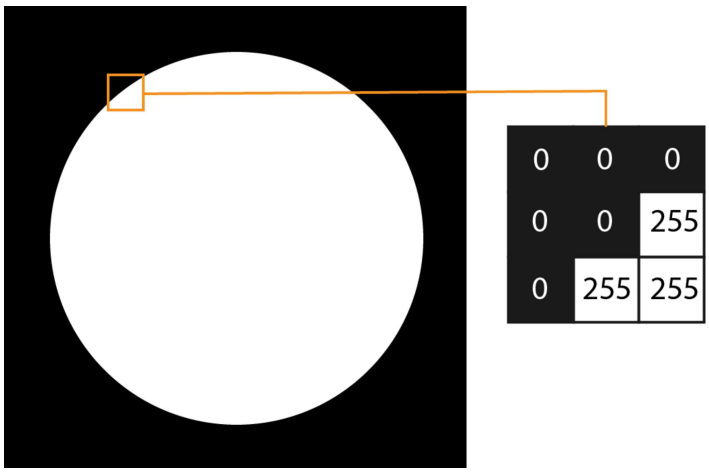
As we can see, we have positive and negative gradients going on. There are changes from black to white and white to black.



If we apply the sobel kernel convolution to this image, we will acquire these gradients for X and Y components.



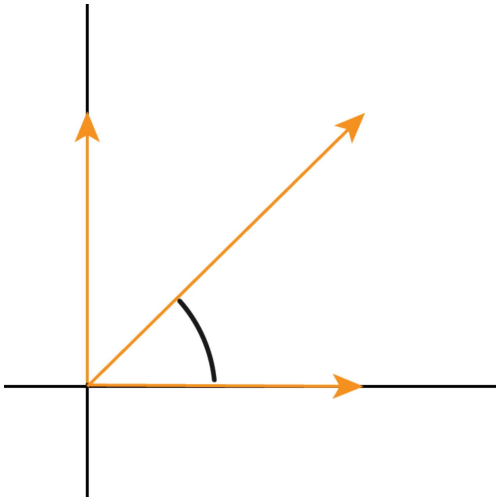
The gradients are positive only for the changes from dark to light. Otherwise, the sign of the gradient is negative. We can calculate this gradient manually, and we can extract a block of 3x3 and use simple substraction between points.



For the ***X*** gradient: ***X = 255 - 0 X = 255***

For the ***Y*** gradient: ***Y = 0-255 Y = -255***

Once we have the components of the gradient, we can calculate its magnitude and orientation.



The magnitude of the gradient:

$$Gm = \sqrt{((255)^2(-255)^2)}$$

$$Gm = 360.6$$

And for the Orientation:

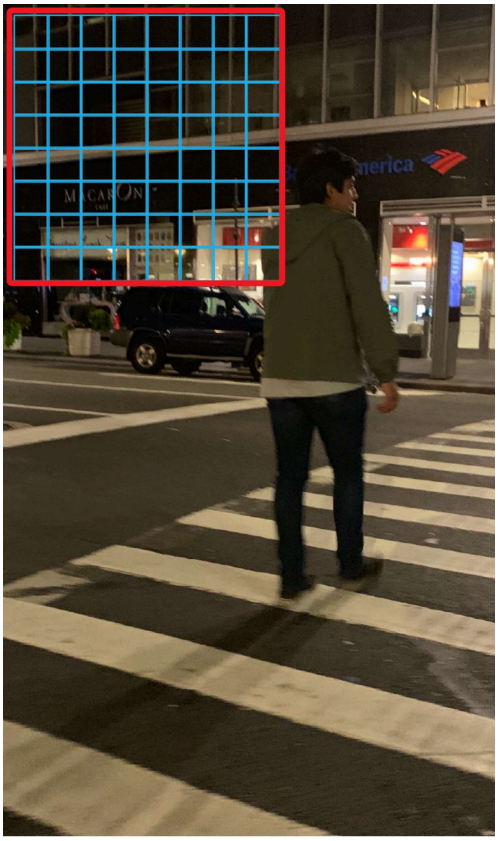
$$\theta = tan^{-1} - (255 / 255)$$

$$\theta = -45^{\circ}$$

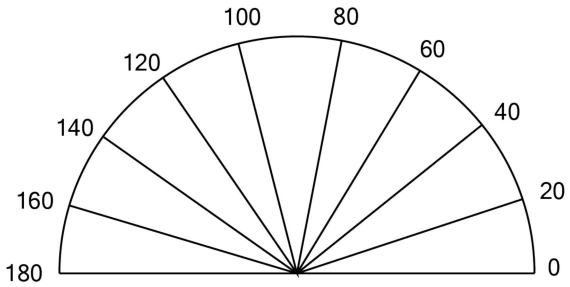
Then let's see how to create the Histogram.

Creating the Histogram

For this part, we need to divide our image in blocks of dimension 8x8. It means we will have 64 gradient calculations per area. The main idea is to get these blocks all over the image. We will have a histogram of magnitude and orientation for each of these blocks.

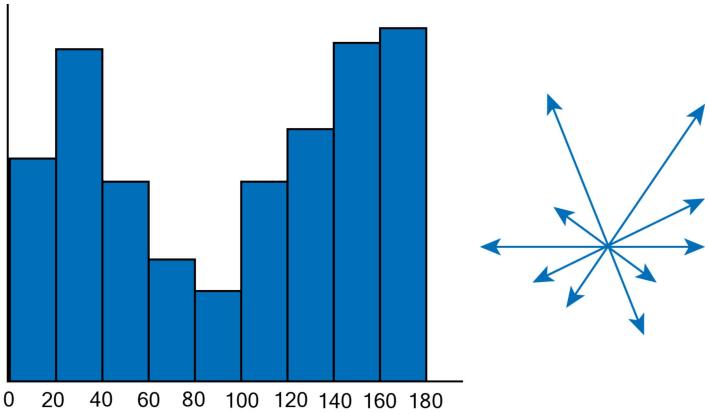


Once we get the blocks of gradients, we need to reduce the number of characteristic features (gradient orientation). For this, instead of saving all the values from 0° to 180°, we will divide it into bins of 20°, so we will get information for only 9 orientation values as shown in the following picture:



Now that we have a smaller segmentation, what we need to do is order our 64 vector magnitudes with their respective orientations into a histogram. For the values that are different from the specified, we need to ponderate them, for example for

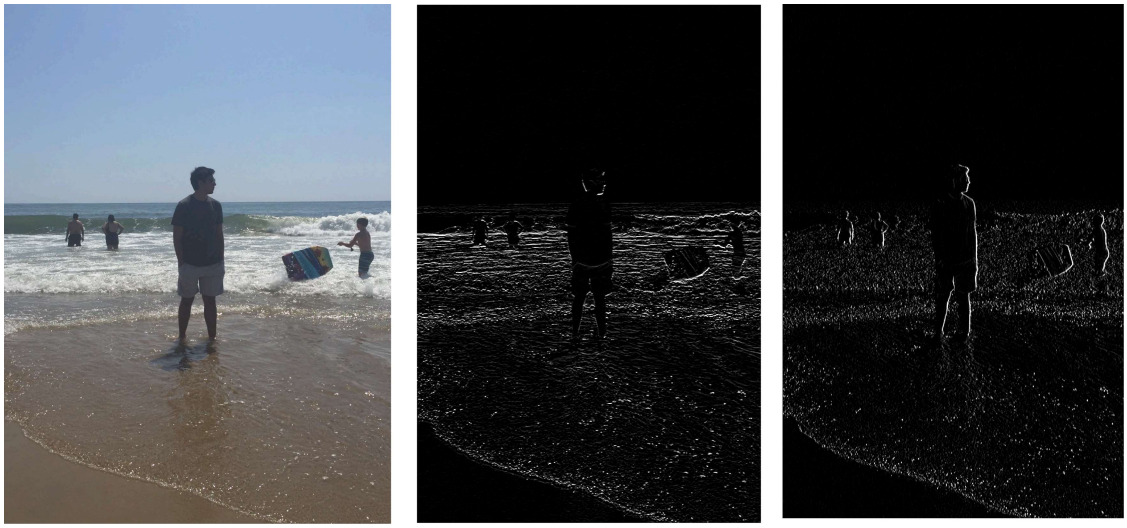
The distance to 40 and 60 is 5 and 15, respectively; therefore, the ratios will be 5/20 and 15/20, so the magnitude assigned will be Gm*1/4 for 40 and Gm*3/4 for 60.



In this way, we can calculate the histogram of oriented gradients for every block of the image.

For every block, we have a vector of dimension 9, which are the main characteristics that we want to extract from the image. Using these descriptors, we can train any machine learning algorithm so that it can help to detect a desired object in an image. A good way to detect people is by using HOG.

Then we will learn how to apply a people detector using the hog algorithm with an opencv trained model. It is pretty accurate. Also, we will use the *Skimage* library, so you can see the computation of the hog descriptors in an image.



The first step would be to calculate the sobel components of the image as shown in the image above; however, opencv helps us with this task. It has an already trained people detector model working under the hog descriptor.

Let's see how it will work with the python code without the structure we use in ROS.

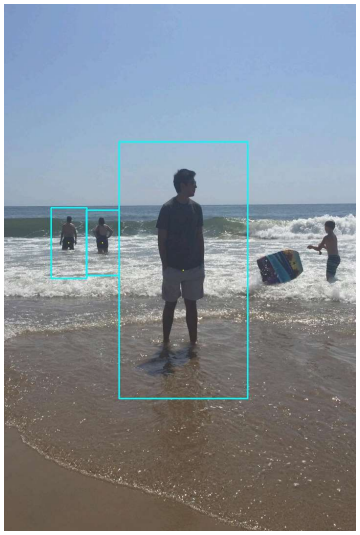
- Example 3.3 -

In [ ]:

```
import numpy as np
import cv2

# Lets initialize the HOG descriptor
hog = cv2.HOGDescriptor()

#We set the hog descriptor as a People detector
hog.setSVMDetector(cv2.HOGDescriptor_getDefaultPeopleDetector())

img = cv2.imread('/home/user/catkin_ws/src/opencv_for_robotics_images/Unit_3/C

#The image is pretty big so we will gibve it a resize
imX = 720
imY = 1080
img = cv2.resize(img,(imX,imY))

#We will define de 8x8 blocks in the winStride
boxes, weights = hog.detectMultiScale(img, winStride=(8,8))
boxes = np.array([[x, y, x + w, y + h] for (x, y, w, h) in boxes])

for (xA, yA, xB, yB) in boxes:

    #Center in X
    medX = xB - xA
    xC = int(xA+(medX/2))

    #Center in Y
    medY = yB - yA
    yC = int(yA+(medY/2))

    #Draw a circle in the center of the box
    cv2.circle(img,(xC,yC), 1, (0,255,255), -1)

    # display the detected boxes in the original picture
    cv2.rectangle(img, (xA, yA), (xB, yB),
                        (255, 255, 0), 2)

cv2.imshow('frame_2',img)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

- End of Example 3.3 -

- Expected Behavior for Example 3.3 -



- End of the expected Behavior for Example 3.3 -

As we mentioned before, we can use the Skimage library to visualize the HOG features. Here is how the python code should be.

- Example 3.4 -

In [ ]:

```
import numpy as np
from skimage import exposure
from skimage import feature
import cv2

img = cv2.imread('/home/user/catkin_ws/src/opencv_for_robotics_images/Unit_3/C

imX = 720
imY = 1080
img = cv2.resize(img,(imX,imY))
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

(H, hogImage) = feature.hog(gray, orientations=9, pixels_per_cell=(8, 8),    c
   visualize=True)

hogImage = exposure.rescale_intensity(hogImage, out_range=(0, 255))
hogImage = hogImage.astype("uint8")

cv2.imshow('features',hogImage)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

- End of the Example 3.4 -

- Explanation for Example 3.4 -

In the first part,

```
(H, hogImage) = feature.hog(gray, orientations=9, pixels_per_cell=(8, 8),    cells_
per_block=(2, 2),
   visualize=True)
```

we define the main parameters of the model, and the number of orientations will be 9 as we explained before. We will have 64 pixels per cell and now we can divide the blocks into cells of 16 *16, so we define blocks as 2 by 2; each one of 8*8 pixels.

- End of the explanation for Example 3.4 -

- Expected Behavior for Example 3.4 -

And here is the result.

As you can see, the silhouette of the person is very highlighted in the hog descriptor. Using these vector values, a support vector machine is trained to abstract what a person is, and is pretty efficient. You can use it to detect multiple objects that are rich in hog descriptors. Also, it would be a good idea to change the learning algorithm to neural networks and compare their performance against the svm.

- End of the expected Behavior for Example 3.4 -

But only theory? That's not our style, is it? Let's do it!.

People detection in an image

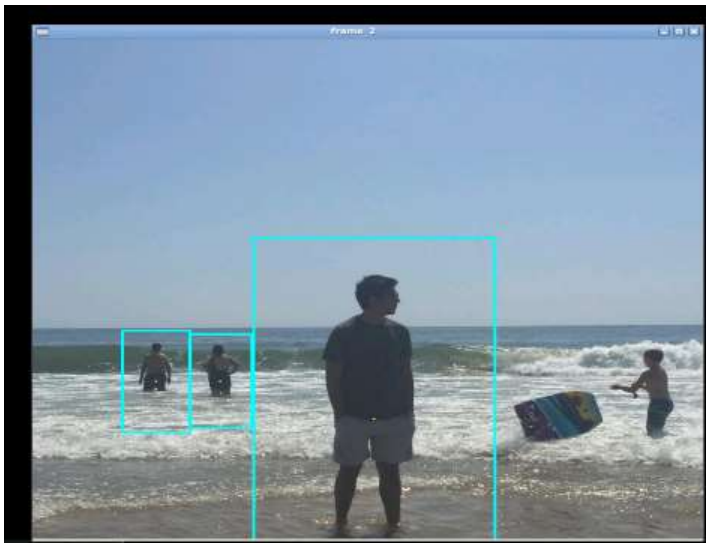
- Exercise 3.5 -

- Create a new file inside *catkin_ws/src/unit3_exercises/src* and call it **exercise3_5.py**.
- Apply the structure that we use in ROS to the **example 3.3** and see the results in the **Graphical Tools**. You can use the same image with the same path as example 3.3.
- Create a new launch file inside *catkin_ws/src/unit3_exercises/launch* and call it **exercise3_5.launch**.

- End of Exercise 3.5-

- Expected Behavior for Exercise 3.5 -

Great! Just like the example. Good job!



- End of Expected Behavior for Exercise 3.5 -

Visualizing the HOG features

Great! Now is the time to see how it works. Well, now it's your turn.

- Exercise 3.6 -

In base exercise 3.5, do the same steps, but this time, applying everything to example 3.4. Go ahead! You can do it!

- Create a new file inside *catkin_ws/src/unit3_exercises/src* and call it **exercise3_6.py**.
- Apply the structure that we use in ROS to the **example 3.4** and see the results in the **Graphical Tools**.
- Create a new launch file inside *catkin_ws/src/unit3_exercises/launch* and call it **exercise3_6.launch**.

- End of Exercise 3.6 -

We hope you did it well. It should be like the following result.

- Expected Behavior for Exercise 3.6 -

It looks great!

- End of the Expected Behavior for Exercise 3.6 -

Yes, all the examples are working well, but you want to use it with the robot! We know that! Now is the time to apply it! Are you ready?

- Exercise 3.7 -

- Inside */catkin_ws/src/unit3_exercises/src*, create two new files and call them **flying_drone3_2.py** and **exercise3_7.py**.
- Inside the **flying_drone3_2py**, copy the following code. It will make the drone fly to a position in order to recognize the person.

In [ ]:

```
#!/usr/bin/env python

import rospy
from geometry_msgs.msg import Twist
from sensor_msgs.msg import Range

class Flying():

    def __init__(self):
        self.hector_vel_publisher = rospy.Publisher('/cmd_vel', Twist, queue_s
        self.hector_sub = rospy.Subscriber("/sonar_height", Range, self.scan_c

        self.cmd = Twist()
        self.ctrl_c = False
        self.rate = rospy.Rate(10) # 10hz
        self.a = 0.0
        rospy.on_shutdown(self.shutdownhook)

    def scan_callback(self, msg):
        self.a = msg.range


    def shutdownhook(self):
        # works better than the rospy.is_shutdown()
        self.ctrl_c = True

    def move_drone(self):
        while not self.ctrl_c:

            if self.a < 1.8:
                self.cmd.linear.z = 0.12

            if self.a > 1.93:
                self.cmd.linear.z = 0.0

            if self.a < 0.5:
                self.cmd.linear.x = -1.0
            if self.a > 0.51:
                self.cmd.linear.x = 0.0



            self.hector_vel_publisher.publish(self.cmd)
```

```
if __name__ == '__main__':
    rospy.init_node('hector_test', anonymous=True)
    hector_object = Flying()
    try:
        hector_object.move_drone()
    except rospy.ROSInterruptException:
        pass
```

- Now in **exercise3_7.py**, do the following steps:
  - Apply the HOG to detect the person in front of you. You can get an idea by looking at exercise 3.5 .
  - Resize the image to a 700 x500. We resize the image only to reduce data processing.
  - Print if the person is on the left side of the robot or on the right side, just one time per move.
  - Show how to detect the person in real time. Shows the image only when the height of the robot is greater than 1.96m and you can move the person using **roslaunch person_sim move_person_standing.launch** in the other shell. Use keys 'j' and 'l' to turn the person, and 'i' to move forward, or 'k' to stop.
- Create a new folder inside the launch folder and call it exercise3_7.launch, in order to launch these files.

- End of Exercise 3.7 -

- Notes -

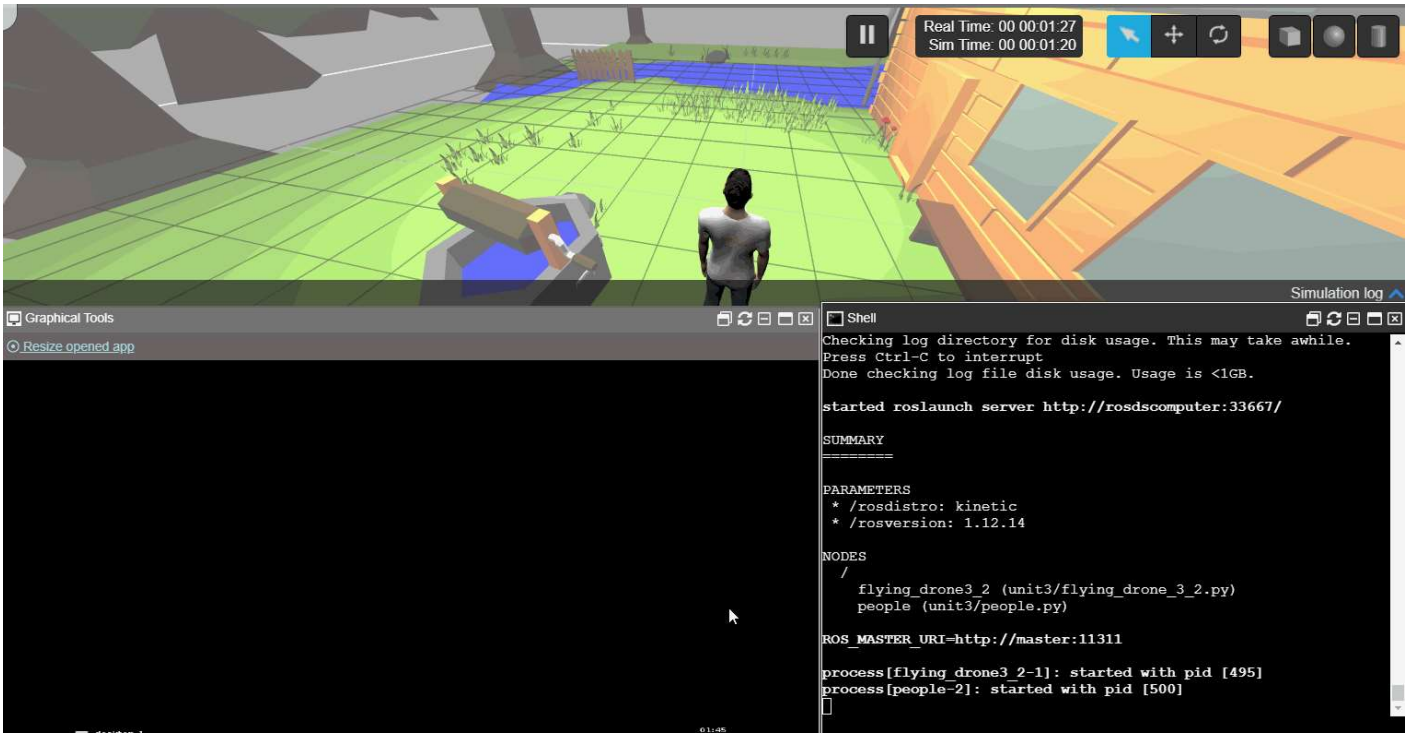
This exercise may run a little slow due to the power of the instance used, however, the code does its proper work.

- End of Notes -

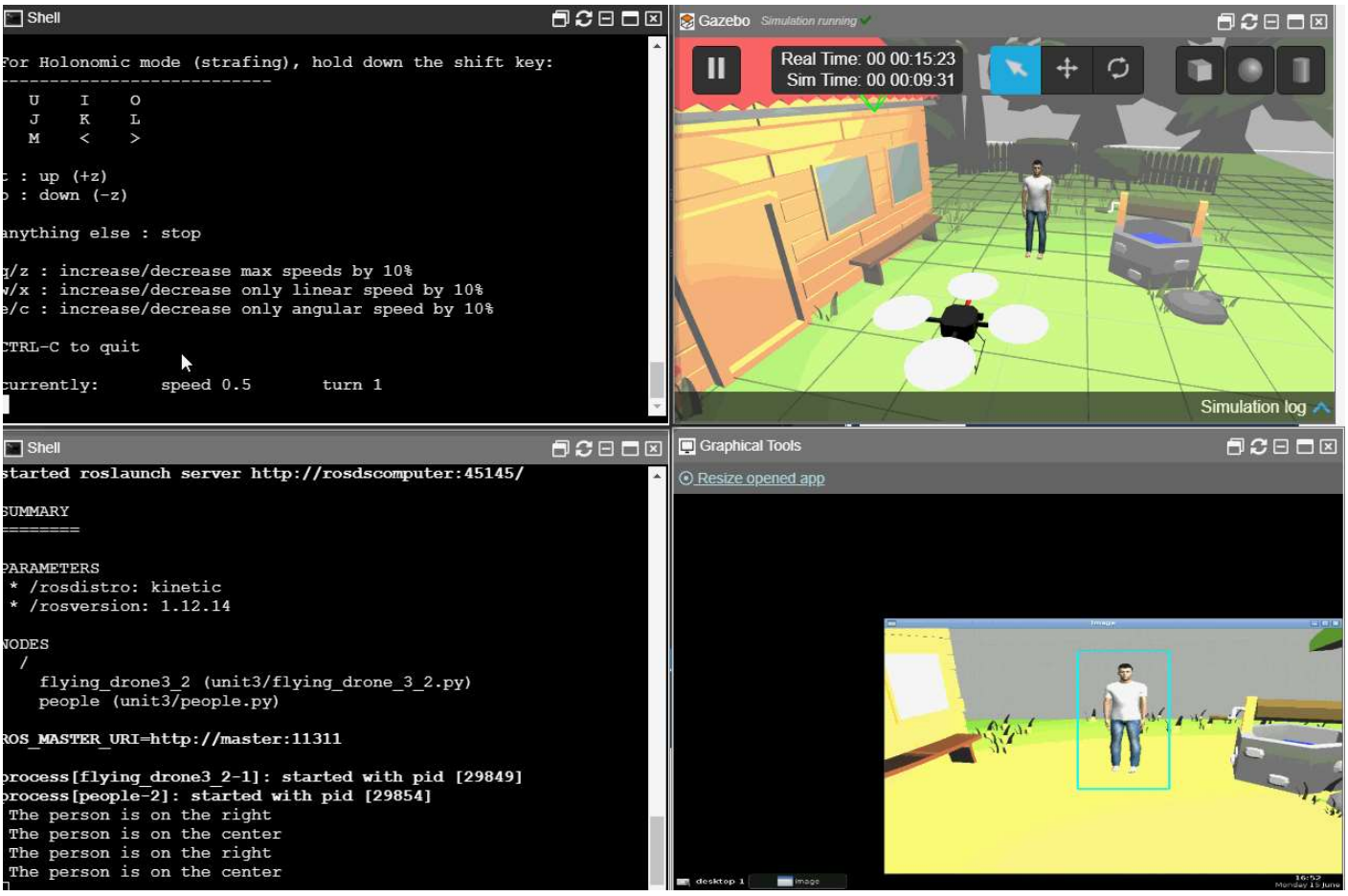
Let's see how it's suposed to be.

- Expected Behavior for Exercise 3.7 -

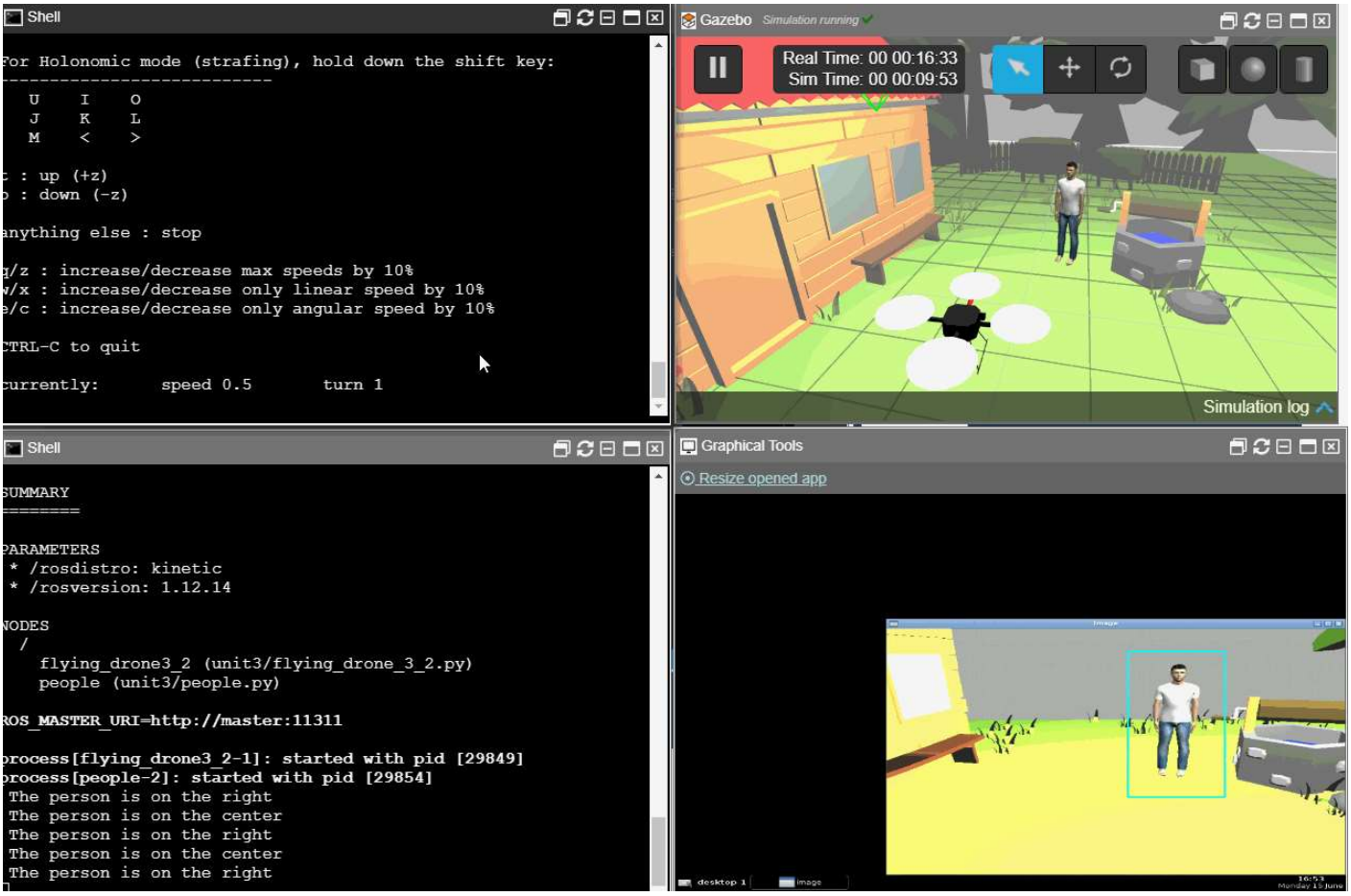
First of all, if you get that, it only shows if the height is greater than 1.95m.



Now, if the person goes to the right



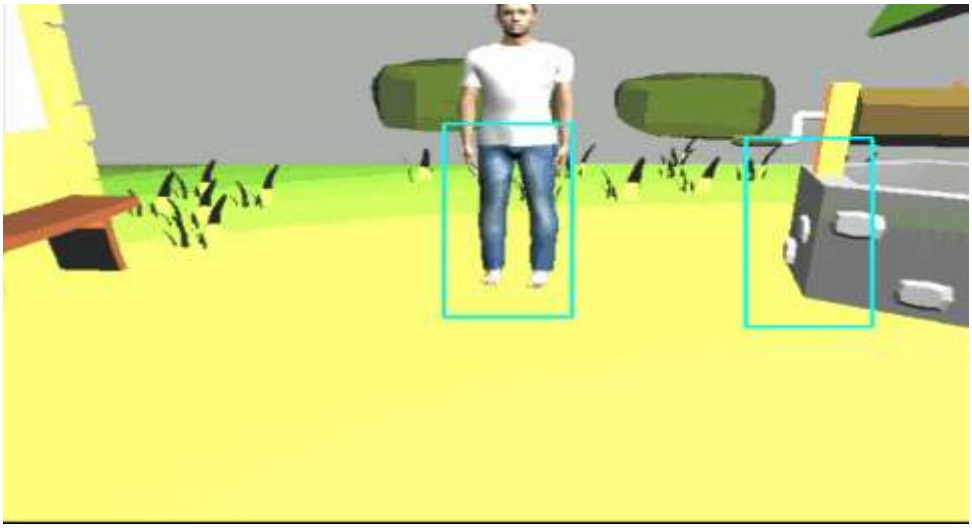
Now to the other side.



- End of Expected Behavior for Exercise 3.7 -

- Notes -

But something weird happen when it was detecting, did you notice? It detects other things.



Sometimes might happen. This error is common, it's called a false positive. As we saw with the face detection with the simulation, the algorithm is not much more precise and optimized; however, it can work well.

- End of Notes -

People tracking

As it could be seen, as in the face detector, the simulation is not always as accurate with this kind of algorithm. However, let's see how it would behave with real images using real video.

- Exercise 3.8 -

Based on **example 3.2 face tracking**, create a file inside *catkin_ws/src/unit3_exercises/src* and call it **exercise3_8.py**. In this file, do the following steps:

- Read the video located in the path **"/home/user/catkin_ws/src/opencv_for_robotics_images/Unit_3/Course_images/chris5-2.mp4"**.
- You have to show the original video and the video with the tracking, but **resize to 300x650**.
- Use HOG in order to track the person in the video.
- Create a launch file inside the **launch folder** in the **unit3_exercises** package in order to launch the node. Call it **exercise3_8.launch**.

- End of Exercise 3.8 -

- Notes -

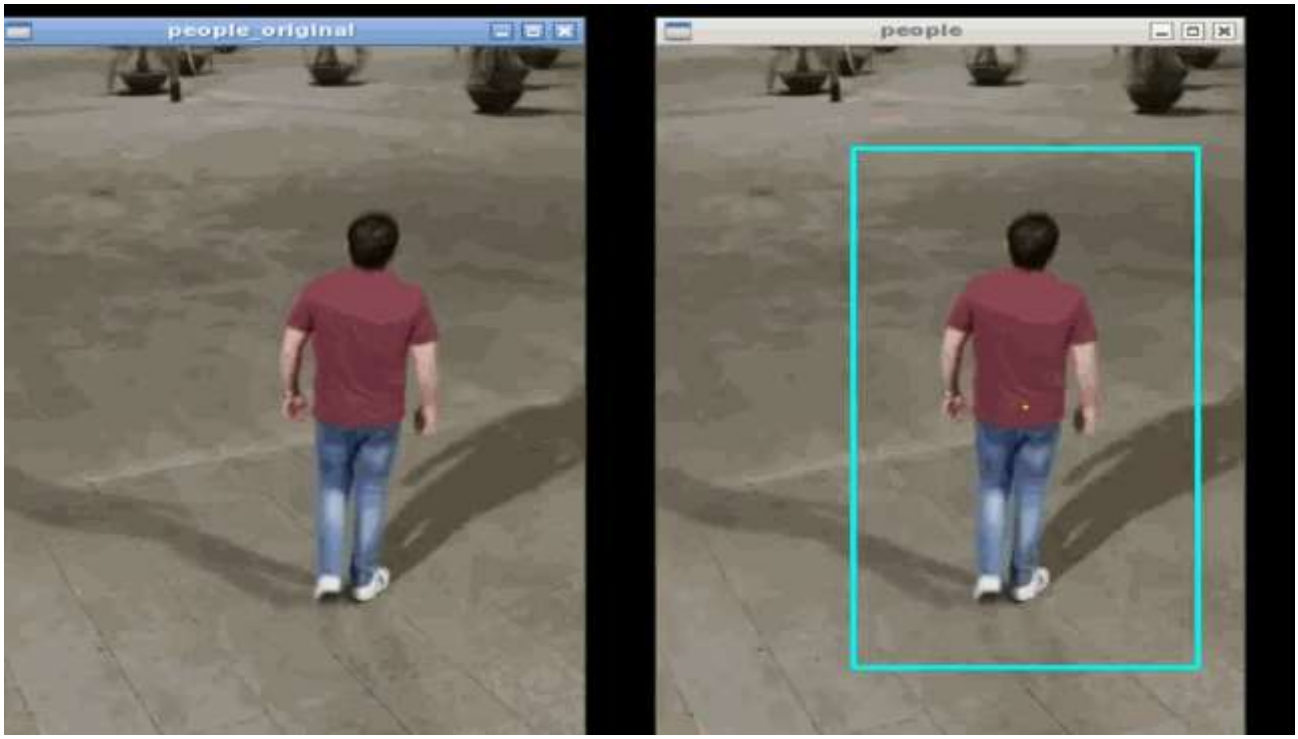
This exercise may run a little slow due to the power of the instance used, however, the code does its proper work.

- End of Notes -

Go ahead! We are pretty sure that you will enjoy the results.

- Expected Behavior for Exercise 3.8 -

If everything is ok, you will see how the person is detected, like the following gif.



- End of Expected Behavior for Exercise 3.8 -

It was very interesting how we learned various algorithms and how we applied them in robotics, right?

GOOD JOB! GO TO THE NEXT UNIT!

English proofread