MKI59: Robotlab practical

Practical 3: Vision

November 30, 2017

Last lesson, we explored several modalities (touch, sonar, speech, vision) for building a reactive NAO. For example, remember that you made your NAO react on detected faces? The NAO used vision algorithms to find faces in images acquired through its camera. In this third lesson, we will start to go beyond "just" using the pre-cooked NAO modules. In particular, we will explore the vision channel and use a well-known state of the art vision library called *OpenCV*. If you never worked with OpenCV before, don't worry, you will be able to follow this tutorial. However, you will need OpenCV on your computer. In this tutorial, you will:

- learn how to capture images and process images or streams of images
- use OpenCV to perform relatively simple image processing routines
- make your NAO find blue balls

1 Computer vision and NAO

The NAOqi library has some built in modules to process its images captured with the NAO's camera. However, many of these modules are rather limited and do not suffice if you want to do something more advanced. In this practical we will move beyond these limitations as we teach you how to capture video information with the NAO and how to process it using OpenCV.

1.1 OpenCV and Numpy

OpenCV is an open-source computer vision package (it's all in the name). It has a lot of useful functions concerning computer vision, allowing you to make cool applications with only a couple of lines code. The library can be downloaded here: https://sourceforge.net/projects/opencvlibrary/files/. We will be using the latest version, version 3.3.1. You are allowed to use a different version, but we might not be able to help you accordingly if you run into trouble. Installation isntructions can be found here: http://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_setup/py_setup_in_windows/py_setup_in_windows.html
To fully make use of OpenCV we are also using NumPy. NumPy makes using arrays in Python a lot easier and faster (both in use and in run time) and has a lot of functions for linear algebra. You will notice that as you are starting to write more complex applications in Python, you cannot go without NumPy. NumPy is already included in many Python distributions (like Anaconda), so chances are you already have this library installed. If not, you an get it here:

http://www.scipy.org/scipylib/download.html Now lets check if you installed both the libraries by running this script:

Code snippet 1: Testing OpenCV and numpy

```
import cv2
import numpy as np

print "cv version" + cv2.__version__
print "Numpy version" + np.__version__
```

If everything is installed correctly, running code snippet 1 will print what version of OpenCV and Numpy you are running.

1.2 Capturing a single frame

All control of the NAO video camera happens though the ALVideoDevice module. To use it we first have to create a proxy of the ALVideoDevice module and subscribe a camera to it as shown in code snippet 2. It also shows how to unsubscribe the camera. **Note:** Since the NAO can only subscribe five cameras at a time it is important to always unsubscribe the camera when done. If five cameras are already subscribed you simply cannot subscribe any more and you will receive an error when trying to do so.

Code snippet 2: Subscribing and unsubscribing the camera

```
import naoqi

videoProxy = naoqi.ALProxy('ALVideoDevice', IP, PORT)

cam_name = "camera" # creates an identifier for the camera subscription
cam_type = 0        # 0 for top camera, 1 for bottom camera
res      = 1        # 320x240
colspace = 13       # BGR colorspace
fps      = 10       # the requested frames per second

#To make sure there are no other camera subscriptions,
#we unsunscribe all previous video proxies
cams = videoProxy.getSubscribers()
for cam in cams:
    videoProxy.unsubscribe(cam)

#subscribing a camera returns a string identifier to be used later on.
cam = videoProxy.subscribeCamera(cam_name, cam_type, res, colspace, fps)

videoProxy.unsubscribe(cam)
```

Subscribing requires a couple of arguments in which you can vary. The most important one is the third argument, the resolution. There are several different resolution settings available, each identified by an integer which you pass as an argument. The integer 1 represents a resolution of 320 by 240 pixels. You can find a list of supported resolutions here: http://doc.aldebaran.com/2-1/family/robots/video_robot.html#cameraresolution-mt9m114. When deciding what resolution to use you have to make a trade-off between speed and quality: a larger image takes a lot more time to receive than a smaller image. More on this later. Another important variable

is the colorspace: how is the image stored? In this case we will be using a BGR colorspace: each pixel is saved with a blue, green and red value (in this order). Why we are using this over the RGB colorspace will become clear later on. Other colorspaces can be found here: http://doc.aldebaran.com/2-1/family/robots/video_robot.html#cameracolorspace-mt9m114

1.3 Capturing and visualizing a single frame

Now that we know how to subscribe to a camera, we will show how to use it to capture a frame and save it to your disk. For visualizing it we will be using OpenCV.

Code snippet 3: Capturing a frame

```
import cv2
import numpy as np

#Don't forget to subscribe to a camera!
#image_container contains info about the image
image_container = videoProxy.getImageRemote(cam)

#get image width and height
width = image_container[0]
height = image_container[1]

#the 6th element contains the pixel data
values = map(ord, list(image_container[6]))

#Write the pixel values to the image array for later use
image = np.array(values, np.uint8).reshape((height, width, 3))

#saves image in you current working directory
cv2.imwrite("firstimage.png", image)
```

If everything went well, you can now find an image captured with the NAO in your current working directory. We got the pixel information, along with the image resolution, from the image container. It also contains information about the colorspace, has a time stamp and some other useful information. The full list can be found here: http://doc.aldebaran.com/2-1/naoqi/vision/alvideodevice-api.html#image. In our next step we will load the image and show it on screen using OpenCV.

Code snippet 4: Showing the image

```
image = cv2.imread("firstimage.png")
cv2.imshow("First image", image)

cv2.waitKey()
```

Code snippet 4 is not that difficult but there is one important thing you should know: without *waitKey* the *imshow* function would only show an empty grey window. So whenever using *imshow*, do not forget to add *waitKey*. When it detects a keypress, it will stop and close the window.

1.4 Processing the image

Now we managed to capture, save and show an image it is time to do some cool stuff with it. In this case we will walk you through all steps needed to filter the image for a colored ball. First of all, start with capturing an image of a ball with the NAO. We will assume that the ball is blue for this tutorial.
A summary of all the steps we are going to perform:

1. Apply a threshold over the image to get everything blue
2. Denoise and smooth the thresholded image
3. Use a Hough-transform to find circular shapes in the image
4. Display the result

Let's start with creating a thresholded image:

Code snippet 5: Applying a threshold

```
image = cv2.imread("ballimage.png")

#determine threshold values
#(we will be using the HSV colorspace, more on this later)
lower_blue = np.array([70, 50, 50], dtype = np.uint8)
upper_blue = np.array([170, 255, 255], dtype = np.uint8)

#convert to a hsv colorspace
hsvImage = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)

#create a treshold mask
color_mask = cv2.inRange(hsvImage, lower_blue, upper_blue)

#apply the mask on the image
blue_image = cv2.bitwise_and(image, image, mask = color_mask)
```

We have now created a mask describing which pixel is within our threshold (applied on the image to filter out non-blue image parts). Indeed, if you visualize the resulting image, you should see that it only contains blue. Furthermore, as you may have have noted, the image was converted to an HSV colorspace. HSV stands for Hue Saturation Value. The first value describes the color, the second the saturation or "amount of gray", and the third describes the brightness. This is shown in figure 1 So in this case we've picked the values 70 to 170 as "blue" (which is quite a wide interval), 50 to 255 as the amount of grey (so from standard blue to dark blue), and 50 to 255 as the amount of brightness (from standard blue to light blue). Try tweaking the different values and observe the effect.

Next we will take the mask we created in code snippet 5 and remove some noise and smooth the edges.

Code snippet 6: Remove noise from the mask and smooth the result

```
kernel = np.ones((9,9), np.uint8)
#Remove small objects
opening = cv2.morphologyEx(color_mask, cv2.MORPH_OPEN, kernel)
#Close small openings
```

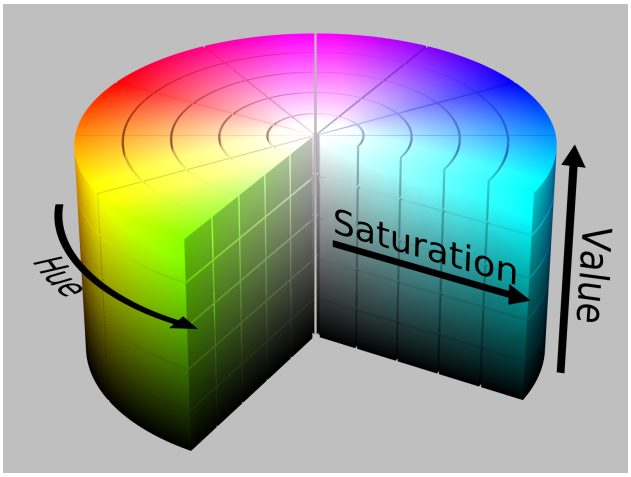

Figure 1: HSV values

```
closing = cv2.morphologyEx(opening, cv2.MORPH_CLOSE, kernel)

#Apply a blur to smooth the edges
smoothed_mask = cv2.GaussianBlur(closing, (9,9), 0)
```

Opening and closing are so called morphological operations which are used a lot in computer vision. The exact workings of these functions are outside the scope of this course, but the basic idea is that opening removes small groups of pixels, and closing fills up small holes in a bigger pixelated area. With the kernel size you determine the size of these objects and holes. You can try and visualize the result of each operation to see its effect. We've also applied a Gaussian blur to smooth the edges of our mask. A Gaussian blur, it is in the name, blurs an image. You can apply it on the original image to see its effect. Next up: find everything that is circular.

Code snippet 7: Finding circular shapes in the image

```
#Apply our (smoothend and denoised) mask
#to our original image to get everything that is blue.
blue_image = cv2.bitwise_and(image, image, mask=smoothed_mask)

#Get the grayscale image (last channel of the HSV image)
gray_image = blue_image[:, :, 2]

#Use a Hough transform to find circular objects in the image.

circles = cv2.HoughCircles(
    gray_image,           #Input image to perform the transformation on
    cv2.HOUGH_GRADIENT, #Method of detection
    1,                    #Ignore this one
    5,                    #Min pixel distbetween centers of detected circles
    param1 = 200,         #Ignore this one as well
    param2 = 20,          #Accumulator threshold: smaller = the more (false) circles
    minRadius=5,          #Minimum circle radius
   maxRadius=100)         #Maximum circle radius

#Get the first circle
```

```
circle = circles[0, :][0]

#Draw the detected circle on the original image
cv2.circle(image, (circle[0], circle[1]), circle[2], (0, 255, 0), 2)
cv2.imshow("Result", image)
cv2.waitKey()
```

If everything worked well, a green circle is drawn around the ball, indicating where OpenCV detected the ball. A lot of stuff happens in this last stap, so lets take a closer look. We first apply our created mask on the image, resulting in a image that only contains the ball (and other big blue specks if they are in the image). Next we convert the image to a grayscale, so we can apply a Hough circle-transform on it. The exact workings of the Hough circle-transform are again outside the scope of this course, which is why some of the arguments can be ignored, but there are two things you should know. First of all, it can only work with grayscale images (that is why we applied the conversion). Second, it outputs a list of detected circles, ordered by its certainty that the found circle actually is a circle, with the first item in the array being the most circular, which is why we pick the first item in the array as our final circle. Each circle in the list contains the x and y position of the center pixel and the radius of the circle. Try changing some of the arguments and observe the effect.

1.5 Streaming video

Now that we are able to process a single image, it is not that hard to move on processing a video stream. This can easily be done by simply keep asking for frames from the camera, process them, and visualize each resulting image. The global structure of such a program is laid out in code snippet 8. Important to note here is the difference in the use of *waitKey*: now it checks for the escape key every iteration, without holding up the whole process. You should be able to fill in the rest.

Code snippet 8: Global structure for processing streaming video

```
while True:
    #capture frame

    #process frame

    cv2.imshow("Detected image", image)

    if cv2.waitKey(33) == 27:
        videoProxy.unsubscribeCamea(camera)
        break #break the while loop
```

You might notice that the stream has a very low frame rate. What is the biggest bottle neck in this process? (You can measure the duration of functions by creating timestamps using time.time()) What can be done to improve on this?

2 Assignment

In this assignment, you will further develop the interactive NAO you built in the previous exercise. Task is to find blue balls in the environment. The NAO should:

- Walk around the environment, while looking for blue balls (move the head a bit l/r/u/d); you may direct the NAO using speech commands
- Display the image stream, processed as described above
- Avoid objects; upon detecting an object make sure it reports this by speech or gesture
- Upon detecting a blue ball: stand still; report that it found a blue ball; move the head such that the ball is in the center of it's camera
- Upon losing the detected ball: start looking for other balls