AUTONOMOUS NAVIGATION WITH NAO

David Lavy and Travis Marshall

Boston University
Department of Electrical and Computer Engineering
8 Saint Mary's Street
Boston, MA 02215
www.bu.edu/ece

December 9, 2015

Technical Report No. ECE-2015-06

Summary

We introduce a navigation system for the humanoid robot NAO that seeks to find a ball, navigate to it, and kick it. The proposed method uses data acquired using the 2 cameras mounted on the robot to estimate the position of the ball with respect to the robot and then to navigate towards it. Our method is also capable of searching for the ball if is not within the robots immediate range of view or if at some point, NAO loses track of it. Variations of this method are currently used by different universities around the world in the international robotic soccer competition called *Robocup*.
This project was completed within EC720 graduate course entitled "Digital Video Processing" at Boston University in the fall of 2015.

Contents

List of Figures

List of Tables

1 Introduction

Autonomous navigation and tracking requires appropriate modeling of environment. Remarkable progress has been done over the years and there is an ongoing research every year. For this project we utilize NAO, a complete humanoid robotics platform created by *Aldebaran Robotics*. NAO has 2 cameras mounted on his head which gives an enlarged field of vision, making it able to see in front of him or at his feet. His extensive capabilities due to the variety of sensors that he has plus its easy-to-use SDK makes him an excellent choice to fuse computer vision and motion dynamics into one single project. In this project, we will enable NAO to perform a low-level control of motion as well as use his cameras for streaming images and do a further processing of the video for tracking a target object.
This concept has been widely used in a popular competition called *Robocup* that happens every year, where robots play soccer among themselves.

Figure 1: Robocup in China - 2015

2 Literature Review

In the last few years, humanoid robots have become a popular research tool as they offer new perspectives compared to wheeled vehicles. For example, humanoids are able to access different types of terrain and to climb stairs. However, compared to wheeled robots, humanoids also have several drawbacks such as foot slippage, stability problems during walking, and limited payload capabilities. For example, there is often serious noise when executing motion commands depending on ground friction and backlash in the joints, i.e., odometry can only be estimated very inaccurately. Also, the observations provided by the lightweight sensors which typically have to be used with humanoids are rather noisy and unreliable. As a result, accurate navigation, which is considered to be mainly solved for wheeled robots, is still a challenging problem for humanoid robots. Numerous papers are written every year addressing this problem, especially using NAO [6], [2], [3]. A whole list of publications can be found in [5].

Based on these publications, we extracted some ideas to create our own method of tracking. This system is explained in the next section.

3 Problem Solution

Our approach to solving the problem (navigate to an object and interact with it) will be broken into 4 main parts: Initial scan of the room to find the ball, positioning the robot facing the ball, walking to the ball while updating the position every certain distance based on the visual information and finally kicking the ball.

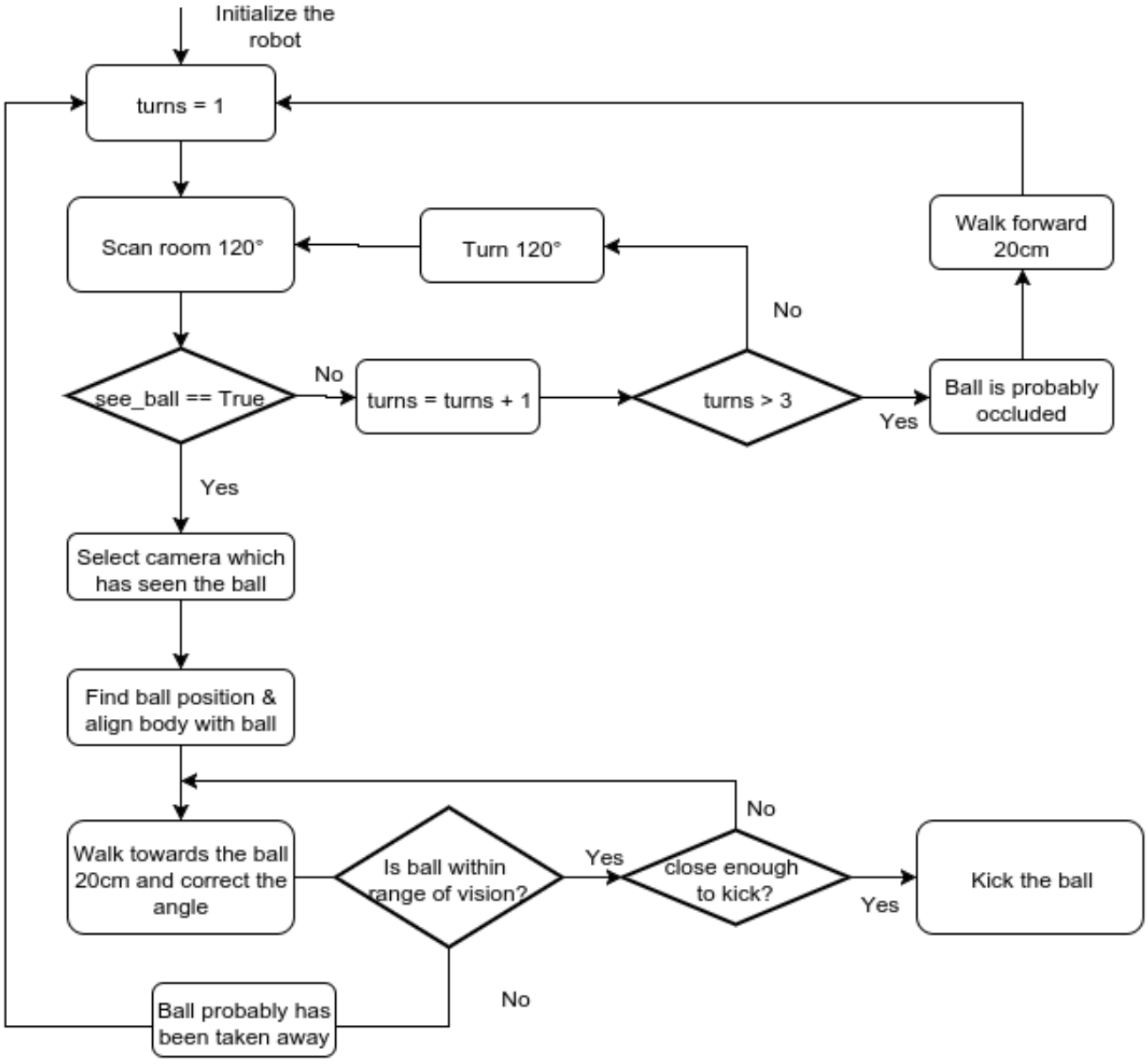


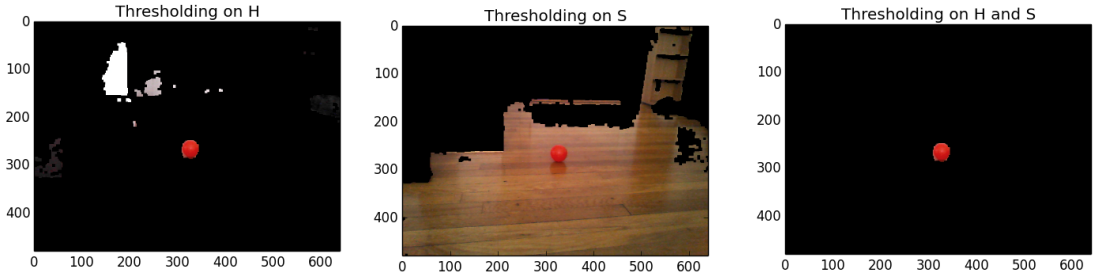
Figure 2: Flowchart of the algorithm for the system proposed

3.1 Find position of the ball

For every image scanned, these are what is used to determine whether the robot can see the ball and its location with respect to the robot. The methods used to identify the ball need to be as simple as possible to be used in real time during the robots search algorithm trying to locate and kick the ball.To find the ball position and approximate distance for scanning and feedback looping while the robot is in motion, three key features are taken into account to find and estimate where the ball is in front of the robot. The features used are color of individual pixels belonging to the ball, the expectation that pixels within the ball are connected and filled in, and the roundness of the ball. Several different methods were used to take these features into account. The following subsections, specifically 3.1.2 through 3.1.5 explain different solutions that were tried in chronological order as well as order of increasing complexity. Section 3.1.1 discusses the binary hypothesis test done as the first step to each method, and section 3.1.5 explains the final implemented solution.

3.1.1 Thresholding HSV color space

To determine which pixels belong to the ball, it was found that using the H and S channels of HSV space produce good results when used co-dependently in a binary hypothesis test to find pixels which most likely belong to the ball. This is more beneficial than using RGB space, for example, because color information is stored on all three dimensions, and luminance is tied into all three channels as well. As seen in Figure 3, using only one of the hue and saturation channels produces less than favorable results, but using both allows for a much more precise approximation of ball pixels at this stage. Also, as seen in Figure 4, the pdf for pixels belonging to the ball are assumed unimodal in both the circular H channel and the linear S channel.



(a) (b) (c)

Figure 3: (a) resulting mask of thresholding only on hue channel, (b) resulting mask of thresholding only on saturation channel, (c) resulting mask of threshold on hue and saturation channels

$$\frac{p(I[x]|H_1)}{p(I[x]|H_0)} \underset{ball}{\overset{not\ ball}{\lessgtr}} thresh \tag{1}$$

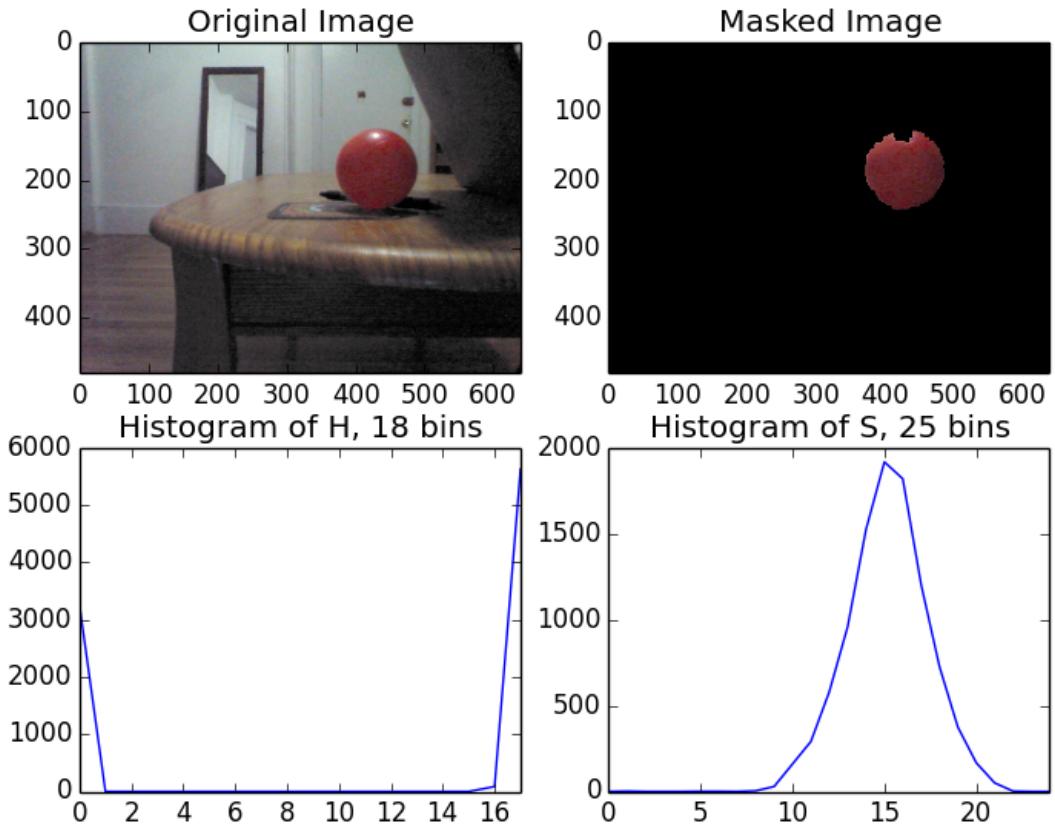


Figure 4: PDF approximation of ball pixels in hue and saturation channels

This means that the binary hypothesis test simplifies to a single set of upper and lower thresholds for each channel that can be implemented on pixel values instead of using a pdf lookup process (Eq 1). After the image is masked using the binary hypothesis test, morphological operations are then performed to reduce noise (opening) and fill in the ball (closing).

3.1.2 Center of mass approach

The first and simplest method used is a simple center of mass calculation (Eq. 2).

$$C_m = [x_{C_m} y_{C_m}]$$

where:

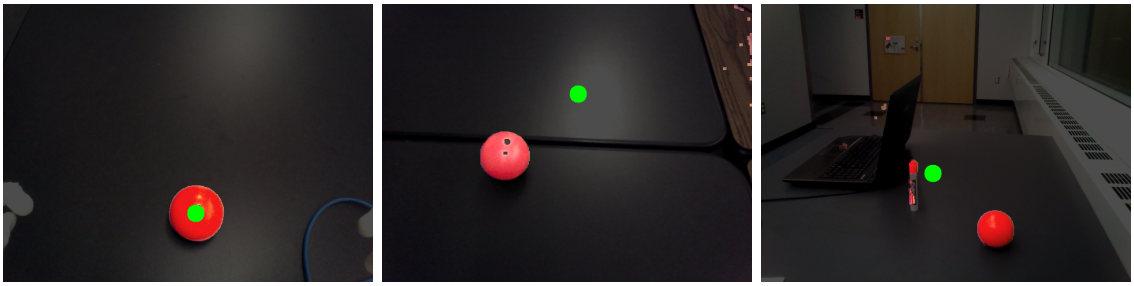
$$x_{C_m} = \frac{\bar{m}_x . \bar{x}^T}{M} \qquad y_{C_m} = \frac{\bar{m}_y . \bar{y}^T}{M} \tag{2}$$

This was used because from observing the masks generated by thresholding in HSV color space, ball pixels are clumped together, and for a perfect threshold, this will produce the exact center of the ball. Also, single noise pixels should not affect the center of mass calculation because the weight of any coordinate is proportional to the number of pixels in that coordinate. This means that because the ball is a very large number of pixels in one place compared to small numbers of noise pixels distributed randomly throughout the image. This is a very simple process and the steps can be seen in

Algorithm 1

1: Threshold image in HSV color space
2: Calculate center of mass on masked image

This method works very well when the threshold is finely tuned so that only the ball is captured, and works acceptably well if the threshold is too tight and some pixels from the ball are excluded. However, as seen in Figure 5, when small numbers of noise pixels are identified outside of the ball, the center of mass is skewed slightly, and when there are smaller red objects in the field of view, the center of mass is skewed greatly. Also, one very important missing functionality of this test is that there is no way to reject an image that has no ball in it unless there are zero identified noise pixels in the image. This means that any single reddish pixel will be a false positive.



(a) (b) (c)

Figure 5: Masked image as seen by NAO. All pixels not identified as the ball are darkened, and the green dot is where the center of the ball is as the output of the center of mass calculation. (a) Center of the ball is correctly identified as the threshold is tight enough to reduce all external noise to zero. (b) Center of the ball is not correctly identified due to external noise pixels. (c) small red objects greatly skew the center of mass calculation.

Moving forward from this method to the next, several important observations were made. The first is that all of the pixels belonging to the ball should be connected because of morphological operations and fairly well-tuned thresholds. Also, noise pixels are not completely erased through the morphological operations, but are

typically much smaller than the group of pixels belonging to the ball. From these observations, the next method needs to find groups of pixels and can make decisions based on the size of the groups of pixels identified in the image.

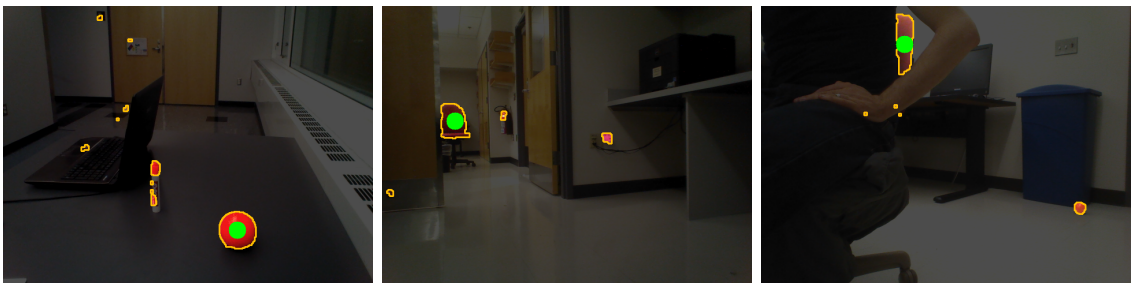
3.1.3 Largest contour approach

After the center of mass observations, it follows that each group of identified pixels can be identified by the contour that can be drawn around it. The ball is very likely to be the largest group of identified pixels since external pixels are noise-like or from smaller red objects. This is only slightly more complex than the center of mass method, and reduces the impact of external noise pixels to zero. The steps for this method can be seen in Algorithm 2.

Algorithm 2

1: Threshold image in HSV color space
2: Find contours around groups of identified pixels
3: Define largest contour as the ball
4: Find the center of the contour as the center of a minimum enclosing rectangle around the contour

This method produces much more favorable results than the center of mass method. Using contours works very well when the ball is the largest group of identified pixels, and can identify the ball with external groups of pixels from smaller objects. However, there is still no error rejection or way to decide whether a contour is actually the ball. As seen in Figure 6, when there is no ball or there are larger areas than the ball of identified pixels, this method fails.



(a) (b) (c)

Figure 6: Images as seen by NAO after contour algorithm. Non-identified pixels are darkened, orange lines are identified contours, and the green dot is the center of the identified ball. (a) Center of the ball is correctly identified with noise and small red objects in the image. (b) A red chair is marked as the ball even though no ball is in the image. (c) padding of a red chair is identified as the ball even though the ball is in the image as well.

The contour method performed better than the center of mass method because it now can ignore small noise pixels, but this method needs a way to determine if the contours belong to the ball or some object in the background. From these observations, the next method will need to implement a metric by which to test contours to decide whether or not a contour belongs to the ball.

3.1.4 Contour error metric approach

Using the contour method as a new baseline, there needs to be a way to determine whether or not the identified contour is actually a ball or not. To improve the contour method, an error metric was applied to the contour to test roundness of the identified contour (Eq. 3). The idea behind this error metric is that the distance from the center of the contour to each point on the contour should be almost the same as the maximum displacement in the X direction of the contour from its center (Figure 7). Instead of just subtracting these two values, the absolute value of this difference is divided by the maximum x displacement to make this metric a percent difference, thus is size-invariant.

$$E = \frac{\sum_{i=1}^{C} \alpha_i}{C}; \qquad \alpha_i = \frac{\left||\bar{d}| - |\bar{\rho}_i|\right|}{|\bar{d}|} \tag{3}$$

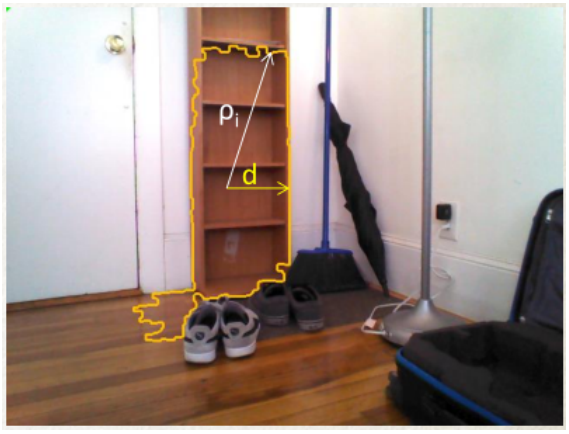
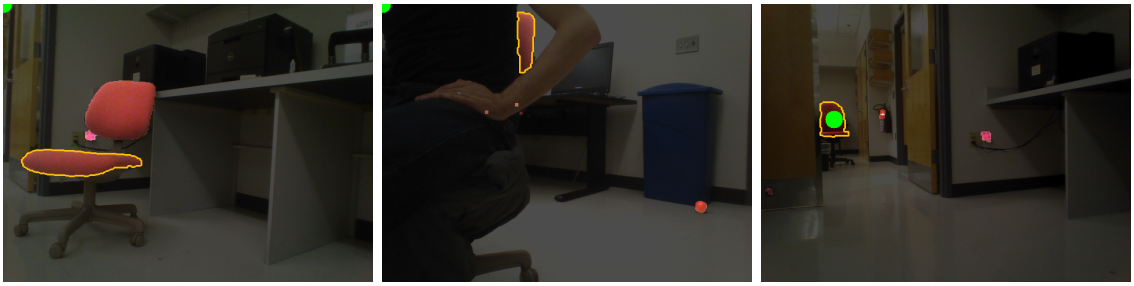


Figure 7: A visual representation of the error metric. Vector d is the maximum X displacement from the center of the contour, and vector ρ_i is the vector from the center of the contour to each point.

The average of the absolute value of the percent difference of these two vectors should be small for the ball, and should be large for non-round objects that are in the image. The implementation of this method can be seen in Algorithm 3.

Algorithm 3

1: Threshold image in HSV color space
2: Find contours around groups of identified pixels
3: Test the largest con tours error metric
4: Reject contour if error metric is too large



(a) (b) (c)

Figure 8: Images as seen by NAO after contour algorithm with the error metric. Non-identified pixels are darkened, the orange outline is the largest identified contour, and the green dot is the center of the identified ball. The green dot being in the top left corner indicates a negative read on the image, i.e. no ball is present. (a) The red chairs larger cushion is rejected as not a ball. (b) the small visible part of the chair back is marked as not a ball, but the ball is in the image as well and is not identified. (c) the back of the chair has a false roundness and is identified as a ball.

The contour with an error metric method works well when the ball is the largest contour or not in the image. Issues arise for this method, however, when the ball is in the image with a non-round larger red object, where the output is a false negative, or an object that isnt round has a false roundness, where the maximum X displacement is not an extreme displacement in terms of the distance to each point on the contour from its center, which reduces the error metric for that contour for non-ball contours (Figure 8).

Adding the error metric is an obvious improvement over just using the contours alone. With this method, there is a way to reject images with identified pixels as not having a ball. Important observations made here are that a better way to define the error metric is needed to reject false roundness, and a way to look at all contours is needed as well to look for a ball in an image where there is a larger red object also in the field of view.

3.1.5 Minimum enclosing circle (MEC) approach

The final method used is a slight improvement upon the contour method that allows two main features of the ball to be taken into account in addition to color- the size

and shape of the ball. The shape of the ball is determined by an error metric very similar to the error metric from the last method, but first a minimum enclosing circle is found around each contour. This minimum enclosing circle is the smallest circle possible that can fit every point in the contour inside of it. Each circle is defined by its center and its radius. This means that the error metric can be defined similarly to before in Eq. 3, but d is replaced by r, the radius of the minimum enclosing circle as seen in Figure 9 (Eq. 4). Similarly, for very round objects such as the ball, the radius of the minimum enclosing circle should be the same as or very close to the radius of the ball in the image. Because of this and the fact that the size of the ball is known and is assumed to be within a reasonable distance to the robot, radius of the minimum enclosing circle is also subjected to a minimum and maximum threshold.

Figure 9: A visual representation of the new error metric. Vector $\bar{r}$ is the radius of the minimum enclosing circle, and vector $\bar{\rho}_i$ is the vector from the center of the minimum enclosing circle to each point.

$$E = \frac{\sum_{i=1}^{C} \alpha_i}{C}; \qquad \alpha_i = \frac{||\bar{r}| - |\bar{\rho}_i||}{|\bar{r}|} \tag{4}$$

The algorithm for this method has not changed much, but now uses three thresholds to determine if there is a ball in an image and where it is. The first, thresholding in HSV color space, is common to all algorithms so far. The second and third are used in unison for each contour. The error metric and the radius of the minimum enclosing circle are tested on each contour so that a ball may be found in an image where a much larger red object is sitting as well (Algorithm 4).

Using the minimum enclosing circle to define the error metric clearly outperforms using the error metric on the largest contour. This approach now accurately rejects all non-round objects while also allowing another parameter to be tested with no noticeable impact on computation time. As seen in Figure 10, having color, size, and shape as parameters produces very favorable results, even on images misidentified in previous methods.

Algorithm 4

1: Threshold image in HSV color space
2: Find contours around groups of identified pixels
3: Find a minimum enclosing circle around each contour
4: Find the contour with the smallest error metric that is within the range of expected radius values
5: Reject contour if error metric is too large

(a) (b) (c)

Figure 10: Images as seen by NAO after minimum enclosing circle algorithm with the error metric. The green dot being in the top left corner indicates a negative read on the image, i.e. no ball is present. (a) The green circle correctly identifies the ball not identified in Figure 8. (b) The chair back previously misidentified as a ball is rejected. The best remaining contour is shown in orange. (c) The pink sticky note on the wall is identified as a ball inside the green circle because it fits in all parameters.

As seen in Figure 10, there are still objects that are not a ball that fit within the expected parameters of the ball. A future method could possibly account for the mean pixel of the ball as another possible metric to decide whether or not a contour is a ball. Disregarding this unlikely circumstance, however, this method works very well compared to previous methods.

3.2 Positioning the robot facing the ball

Once we have the location of the center of the ball, we will make use of pixel information of the image to rotate the robot and face it directly.
By using a 640x480 pixel image, we need to have the ball centered at 320. We need at least 2 images where the ball has been detected, and we also make use of the rotation angles where every picture has been taken. Fusing these two different informations we can calculate the closest rotation angle, which gives us a good centering of the ball.

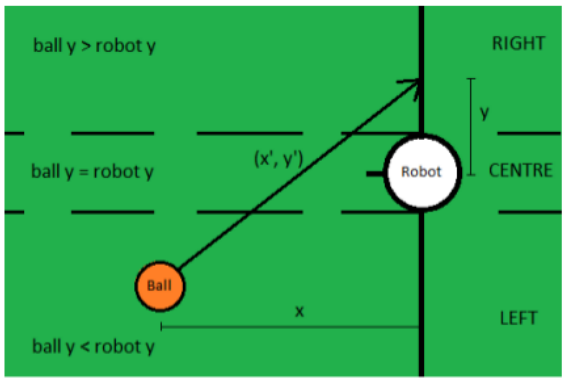


Figure 11: Visualization of ball position based on the robot facing' direction

3.3 Walking to the ball

Once it has aligned its body facing the ball, NAO will start walking towards it. Because NAO doesn't have a feedback for its odometry, information about real data can't be extracted. In Figure 12 we show an initial state where NAO is directly facing the ball in a straight line, and the final state after walking 30 cm.

(a) Initial position (b) Final position after walking 30cm

Figure 12: Straight walk test for NAO

As we can see there is an obvious drift due to sliding, friction between the robot's feet and floor, etc. This is why we can't just rely on the motion dynamics. The approach we took was to correct this drifting in two stages. The first stage is when NAO farther than 10cm from the ball; NAO will do a 20cm displacement, and using the visual information from his cameras (upper or lower, depending where it sees the

ball) it will correct the drifting. Once it gets close enough (∼10cm) it needs to walk in a more precise manner, as we are trying to have NAO stand right in front of the ball, so we rely again on the visual information from its lower camera but this time the displacement and rotations are done separately to compensate imprecisions.

3.4 Kicking the ball

In this final step, NAO should be positioned correctly and ready to kick the ball. This motion is pretty complicated as we need to take into consideration a lot of dynamics of the robot to have a good kick. Developers who have worked with NAO state that programming a whole kick requires good knowledge of humanoid balance, and the official documentation shows a very basic program to make NAO kick the ball. We played around with this code and found a decent kick, which has worked every time we have tested it. Although a much better kick will be desired, this will be part of our future works once we have better understanding of motion dynamics.

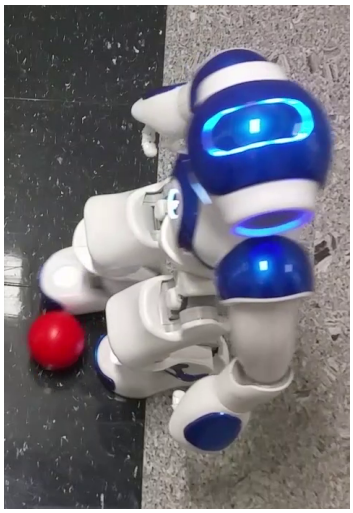


Figure 13: NAO side-kicking the ball with its right foot

4 Implementation

Our final implementation was written in Python (see Appendix). Although it would be nice to have it in C++ for a much faster compilation and running, it was a challenging task to learn most of the SDK for NAO (NAOqi) [4] using C++, whereas Python, being a friendly language, made our programming easier. We also used OpenCV [1] for image processing, we focused on using this library because NAOqi has OpenCV built-in, and has very useful functions for the processing we described before.

4.1 Image Acquisition

NAOqi is composed of modules that control different features of the robot. One of them is called ALVideoDevice which grabs the image at a certain rate and resolution. For this project we used 10fps with 640x480 pixel images. This was good enough to visualize the ball without having much distortion when far away (∼10ft). We also took advantage of both cameras. Although it was not possible to take two pictures at the same time, in our initial scan we did two swipes with each camera to localize the ball.

4.2 Finding Contours and Circles

OpenCV was used for the image processing part. To threshold the ball we used Python *numpy* and OpenCV funtions such as *morphologyEx*. Calculations of the center of mass and contours were addressed using *findContours* and *minEnclosingCircle*; these functions were fast enough to allow a real-time system.

4.3 Navigation and kicking for NAO

Motion of the robot was performed using the ALMotion and ALRobotPosture modules. The documentation explains how to achieve motion using the parameters x, y and θ of the robot as shown in Figure 14. However due to the imprecision of the robot, NAO doesn't achieve these translations and rotation. Figure 15 shows the expected position and the real position of the ball once NAO has walked 20cm in a straight line towards the ball.

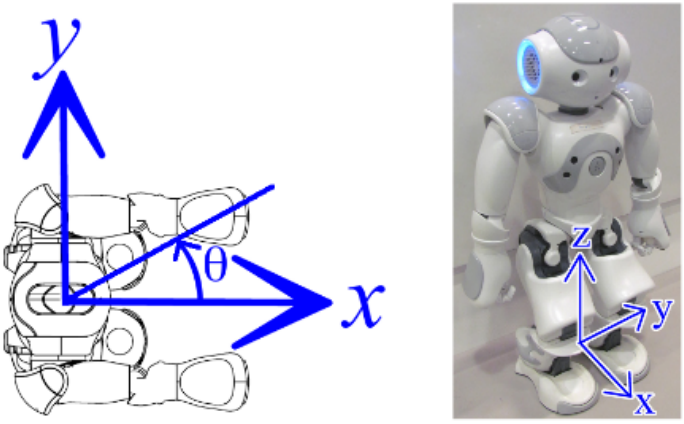


Figure 14: NAO coordinates for motion [4]

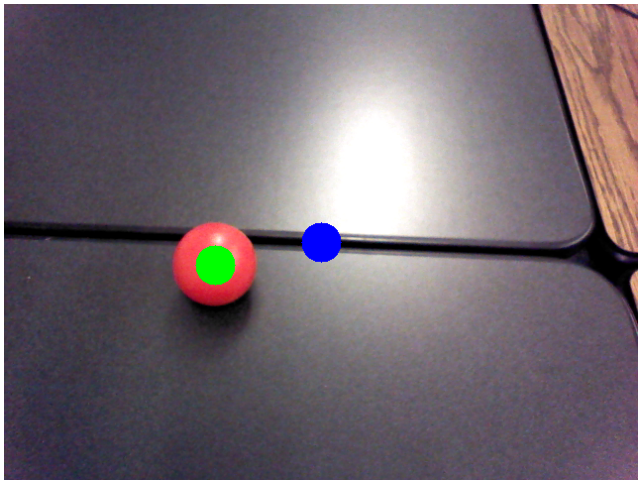


Figure 15: Ball position: Expected (*blue circle*) vs Real (*green circle*)

5 Experimental Results

5.1 Ball identification results by method

	Find ball	**~Find ball**	**Reach ball**	**~Reach ball**	**Kick ball**	**~Kick ball**
Center of Mass (CoM)	8	12	13	7	15	5
Contour	13	7	13	7	17	3
Contour + Error metric (EM)	16	4	15	5	17	3
Minimum Enclosing Circle (MEC)	15	5	14	6	17	3
MEC + EM	19	1	19	1	19	1

Table 1: Results for different ball identification methods. Find Ball is the ability to turn towards the ball's initial position. Reach Ball is the ability to walk towards a ball starting faced at the ball. Kick Ball is the ability to use fine movements to position and kick the ball from close.

5.2 Confusion matrices

To describe the performance of our ball identification model, we used the following confusion matrices showing the results for 4 different methods. 30 different cases were evaluated and the accuracy (Acc) was calculated as the sum of **true positives** and **true negatives** (sum of the diagonal values) divided by the total number of cases.

N = 30	Ball	No Ball
Ball	10	5
No Ball	7	8

Table 2: Contour: 60% Acc

N = 30	Ball	No Ball
Ball	12	3
No Ball	2	13

Table 3: Contour + EM: 83.3% Acc

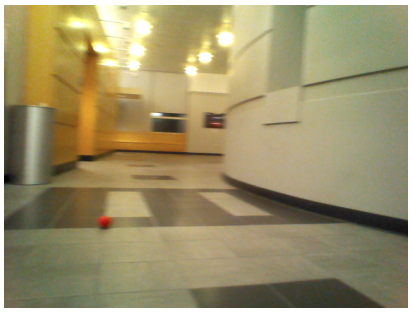
N = 30	Ball	No Ball
Ball	14	1
No Ball	4	11

Table 4: MEC: 93.3% Acc

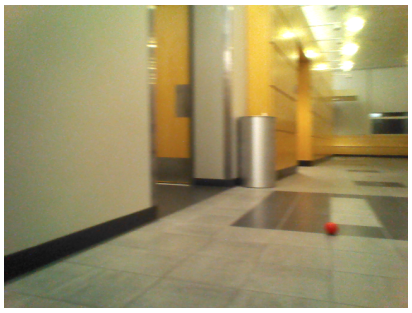
N = 30	Ball	No Ball
Ball	15	0
No Ball	2	13

Table 5: MEC + EM: 83% Acc

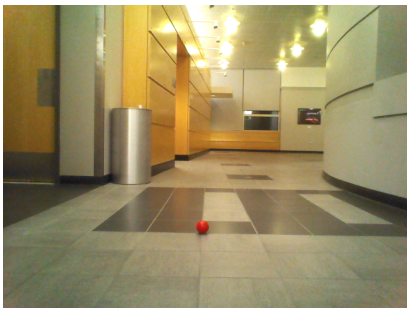
5.3 Centering NAO to face the ball



(a) Detected ball: Frame 1



(b) Detected ball: Frame 2



(c) Angle correction and rotation of NAO

Figure 16: Stages of angle correction to face the ball

As expected, the algorithm performed significantly better with more sophisticated ball detection methods aimed at improving the metrics by which a ball was defined. As can be seen in Table 1, although changing to an error metric based on the MEC caused an increase in the error metric for ground truth balls, it also was accompanied by a smarter search algorithm that allowed it to find the contour that minimized the error result. This performs better at separating non-ball contours from ball contours in the error metric while looking for contours that fit the metric the best. This means

that the MEC search can separate the ball from large contours of falsely identified pixels in the same image as the ball, whereas the contour and error metric had a difficult time with this. The confusion matrices in Section 5.2 reinforces this along with the trial results in Table 1. There are still scenarios that cause the MEC with error metric to fail, such as round-ish smaller objects with the right color, also seen in Figure 10.c. Since the contour and error metric without enclosing circle worked very well with constant backgrounds like having the ball at the robot's feet, we were also able to use the slightly less computationally expensive version of ball detection for close-range detection.
Finally using the detection method and the angle correction we are able to have a good bearing seen in Figure 16.c where the ball is almost centered. In Table 1, we can see how this method achieves a 19/20 (95%) success rate.

6 Conclusion and Future Work

The implementation and methods are successful in identifying and navigating to a ball in an acceptably consistent manner. As seen in the results, much has been done to improve the accuracy and rigour of ball finding results as well as movement methodology for more consistent results, but there is more that can be done to improve this project. For continued work on this project with NAO, there are several improvements that can be made to augment the performance of the ball search algorithm. The detection of the ball performs well, and has progressed from high misidentification with only center of mass calculation to very high correct identification using an error metric to compare contours to MECs. This piece of the algorithm would benefit from more testing and fine tuning of the error metric, and it was also suggested that an error metric that finds the percent difference in area of the contour versus the area of the corresponding MEC could be tested as a replacement for the average percent difference of contour pixels from the MEC radius being used currently. Another facet of the program that is currently performing well with room for improvement is walking to the ball. As of now, NAO's obstacle avoidance capabilities are not being used, so it is entirely possible that the robot could run into an obstacle or get stuck in an unsupervised search for a ball under normal circumstances.Also, the current search and movement pattern looks clunky because the robot has to completely stop between 0.2 meter walks to take pictures and decide where to move, which allows the ball to become lost easily. Adding obstacle avoidance navigation and allowing NAO to update trajectory while walking instead of having to stop would greatly improve the performance and external feel of the algorithm while also presenting new challenges such as pixel blur from slow image capture speeds because of indoor lighting conditions and sensor capabilities. Even with these improvements, the hardware and software capabilities of NAO have only begun being explored in this project, and many more modules and toolkits that provide useful information could be added as well.
A video demonstration of this project can be found online at `https://youtu.be/`

Lp79IdTRV4Q

7 Appendix

7.1 NAO features

The NAO humanoid robot is developed by Aldebaran Robotics, a French company founded in 2005 that is focused on humanoid robotics.
NAOs key components are a 57 cm tall body with 25 degrees of freedom with electric motors and actuators. He owns a whole sensor network of two cameras, four microphones, a sonar rangefinder, two IR emitters and receivers, one inertial board, nine tactile sensors and eight pressure sensors. NAO is also equipped with several communication devices, such as a voice synthesizer, LED lights and two high-fidelity speakers. Moreover, NAO owns two CPUs, one located in the head and the other one located in the torso, which are Intel ATOM 1,6ghz CPUs that run a Linux kernel and support NAOqi, which is Aldebaran's propriety middleware.

Figure 17: NAO Overview [4]

7.2 Python Code

The main code below and the other test functions can be found at `https://github.com/davidlavy88/autonomousNAO`

```
import qi
import argparse
```

```
import time
import math
from functools import partial
import numpy as np
import motion as mot

# Python Image Library
import Image
# OpenCV Libraries
import cv2
import cv2.cv as cv

''' ROBOT CONFIGURATION '''
robotIP = "169.254.194.108"
ses = qi.Session()
ses.connect(robotIP)
per = qi.PeriodicTask()
motion = ses.service('ALMotion')
posture = ses.service('ALRobotPosture')
tracker = ses.service('ALTracker')
video = ses.service('ALVideoDevice')
tts = ses.service('ALTextToSpeech')
landmark = ses.service('ALLandMarkDetection')
memory = ses.service('ALMemory')

resolution = 2    # VGA
colorSpace = 11   # RGB
trial_number = 12
path = 'trials/trial' + str(trial_number) + '/'

# During the initial scan, take a few pictures to analize where's the ball
def take_pics(angleScan, CameraIndex):
    names = "HeadYaw"
    useSensors = False
    motionAngles = []
    maxAngleScan = angleScan

    motion.angleInterpolationWithSpeed("Head", [-maxAngleScan, 0.035], 0.1)
    pic(path + 'bFound0.png', CameraIndex)
    commandAngles = motion.getAngles(names, useSensors)
    motionAngles.append(commandAngles)
    print str(commandAngles)
    motion.angleInterpolationWithSpeed("Head", [0, 0.035], 0.1)
```

```
    pic(path + 'bFound1.png', CameraIndex)
    commandAngles = motion.getAngles(names, useSensors)
    motionAngles.append(commandAngles)
    print str(commandAngles)
    motion.angleInterpolationWithSpeed("Head", [maxAngleScan, 0.035], 0.1)
    pic(path + 'bFound2.png', CameraIndex)
    commandAngles = motion.getAngles(names, useSensors)
    motionAngles.append(commandAngles)
    print str(commandAngles)
    centers = analyze_img2()
    return [centers, motionAngles]

# Find the ball and center its look to it, otherwise back to 0 and rotate again
def locate_ball(centers, rot_angles):
    index = numBalls(centers)
    if len(index) == 0:
        string = "I don't see the ball."
        ang = 100
        state = 0
        RF = 0
    elif len(index) == 1:
        a = index[0]
        string = "I need to get a better look at the ball."
        ang = rot_angles[a][0]
        # ang = ang.item()
        state = 1
        RF = 0
        motion.angleInterpolationWithSpeed("Head", [ang, 0.035], 0.1)
    else:
        string = "I see the ball."
        a = index[0]
        b = index[1]
        RF = (rot_angles[b][0] - rot_angles[a][0]) / (centers[a][1] - centers[b][1
        ang = rot_angles[a][0] - (320 - centers[a][1])*RF
        # ang = ang.item()
        state = 2
        motion.angleInterpolationWithSpeed("Head", [ang, 0.035], 0.1)
    print ang
    tts.say(string)
    return [ang, state, RF]

# Move HeadYaw from [-angleScan;angleScan]
def move_head(angleScan):
```

```
    print 'moving head'
    angleLists = [[0, angleScan]]
    timeLists = [[1.0, 2.0]]
    motion.angleInterpolation("HeadYaw", angleLists, timeLists, True)

# Calculate CoM of the thresholded ball (center of the circle)
def CenterOfMassUp(image):
    hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
    lowera = np.array([160, 95, 0])
    uppera = np.array([180, 250, 255])
    lowerb = np.array([0, 95, 0])
    upperb = np.array([10, 250, 255])

    mask1 = cv2.inRange(hsv, lowera, uppera)
    mask2 = cv2.inRange(hsv, lowerb, upperb)
    mask = cv2.add(mask1,mask2)
    kernel = np.ones((5,5),np.uint8)
    mask = cv2.morphologyEx(mask, cv2.MORPH_OPEN, kernel)
    mask = cv2.morphologyEx(mask, cv2.MORPH_CLOSE, kernel)

    cont, hier = cv2.findContours(mask,cv2.RETR_TREE,cv2.CHAIN_APPROX_NONE)

    if len(cont)>=1:
        minim=0
        minE=1
        for l in range(len(cont)):
            contour=cont[l]
            center, radius = cv2.minEnclosingCircle(contour)
            icenter = []
            icenter.append(int(center[0]))
            icenter.append(int(center[1]))
            radius = int(radius)
            d=radius
            err=[]
            for k in range(len(contour)):
                t=[contour[k,0,0]-center[0],contour[k,0,1]-center[1]]
                t=math.sqrt(math.pow(t[0],2)+math.pow(t[1],2))
                e=abs(d-t)/d
                err=err+[e]
            ERR=np.mean(err)
            if ERR < minE and radius >= 8 and radius < 65:
                minim = l
                minE = ERR
```

```
                Radius = radius
                Center = icenter
        if minE > .235:
            i = 0
            j = 0
            contour = []
            Radius = 0
        else:
            i = Center[1]
            j = Center[0]
            contour = cont[minim]
    else:
        i = 0
        j = 0
        minim = 1
        Center = [1, 1]
        contour = cont
        Radius = 0

    CM = [i, j]

    return CM

def CenterOfMassDown(image):
    hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
    kernel = np.ones((5,5),np.uint8)
    lowera = np.array([160,95,0])
    uppera = np.array([180,250,255])
    lowerb = np.array([0,95,0])
    upperb = np.array([10,250,255])

    mask1 = cv2.inRange(hsv, lowera, uppera)
    mask2 = cv2.inRange(hsv, lowerb, upperb)
    mask = cv2.add(mask1,mask2)
    mask = cv2.morphologyEx(mask, cv2.MORPH_OPEN, kernel)
    mask = cv2.morphologyEx(mask, cv2.MORPH_CLOSE, kernel)

    cont, hier = cv2.findContours(mask,cv2.RETR_TREE,cv2.CHAIN_APPROX_NONE)
    icenter = []
    if len(cont)>=1:
        maxim=0
        for i in range(len(cont)):
            if len(cont[i])>len(cont[maxim]):
```

```
                maxim=i
        contour=cont[maxim]

        center, radius = cv2.minEnclosingCircle(contour)
        icenter.append(int(center[0]))
        icenter.append(int(center[1]))
        radius = int(radius)
        print 'Center', icenter, '. Radius', radius
        if radius>8 and radius<60:
            i=icenter[1]
            j=icenter[0]
        else:
            i=0
            j=0
    else:
        i=0
        j=0
        contour=cont

    CM=[i,j]
    return CM

# Find the center of mass of the ball
def analyze_img():
    CM = []
    for i in range(0, 7):
        img = cv2.imread(path + "camImage" + str(i) + ".png")
        cm = CenterOfMassUp(img)
        CM.append(cm)
    return CM

def analyze_img2():
    CM = []
    for i in range(0, 3):
        img = cv2.imread(path + "bFound" + str(i) + ".png")
        cm = CenterOfMassUp(img)
        CM.append(cm)
    return CM

# Look if the ball is in front of the robot
def scan_area(_angleSearch, CameraIndex):
    # Search angle where angle of rotation is [-maxAngleScan;+maxAngleScan]
    names = "HeadYaw"
```

```
    useSensors = False
    motionAngles = []
    maxAngleScan = _angleSearch

    motion.angleInterpolationWithSpeed("Head", [-maxAngleScan, 0.035], 0.1)
    pic(path + 'camImage0.png', CameraIndex)
    commandAngles = motion.getAngles(names, useSensors)
    motionAngles.append(commandAngles)
    print str(commandAngles)
    motion.angleInterpolationWithSpeed("Head", [-2*maxAngleScan/3, 0.035], 0.1)
    pic(path + 'camImage1.png', CameraIndex)
    commandAngles = motion.getAngles(names, useSensors)
    motionAngles.append(commandAngles)
    print str(commandAngles)
    motion.angleInterpolationWithSpeed("Head", [-maxAngleScan/3, 0.035], 0.1)
    pic(path + 'camImage2.png', CameraIndex)
    commandAngles = motion.getAngles(names, useSensors)
    motionAngles.append(commandAngles)
    print str(commandAngles)
    motion.angleInterpolationWithSpeed("Head", [0, 0.035], 0.1)
    pic(path + 'camImage3.png', CameraIndex)
    commandAngles = motion.getAngles(names, useSensors)
    motionAngles.append(commandAngles)
    print str(commandAngles)
    motion.angleInterpolationWithSpeed("Head", [maxAngleScan/3, 0.035], 0.1)
    pic(path + 'camImage4.png', CameraIndex)
    commandAngles = motion.getAngles(names, useSensors)
    motionAngles.append(commandAngles)
    print str(commandAngles)
    motion.angleInterpolationWithSpeed("Head", [2*maxAngleScan/3, 0.035], 0.1)
    pic(path + 'camImage5.png', CameraIndex)
    commandAngles = motion.getAngles(names, useSensors)
    motionAngles.append(commandAngles)
    print str(commandAngles)
    motion.angleInterpolationWithSpeed("Head", [maxAngleScan, 0.035], 0.1)
    pic(path + 'camImage6.png', CameraIndex)
    commandAngles = motion.getAngles(names, useSensors)
    motionAngles.append(commandAngles)
    print str(commandAngles)
    centers = analyze_img()
    return [centers, motionAngles]

# Index of pictures that contains the ball
```

```
def numBalls(CM):
    """Takes in the CM list, outputs indices of frames containing balls"""
    index = []
    for i in range(len(CM)):
        if CM[i] != [0, 0]:
            index.append(i)
    return index

# Find the ball and center its look to it, otherwise back to 0 and rotate again
def rotate_center_head(centers, rot_angles):
    index = numBalls(centers)
    found = 1
    if len(index) == 0:
        string = "I don't see the ball."
        ang = 100
        found = 0
    elif len(index) == 1:
        a = index[0]
        string = "I think I see the ball."
        ang = rot_angles[a][0]
    else:
        string = "I see the ball."
        a = index[0]
        b = index[1]
        den = 3
        if len(index) < 3:
            ang = (rot_angles[b][0] + rot_angles[a][0])/2
        else:
            c = index[2]
            ang = (rot_angles[b][0] + rot_angles[a][0] + rot_angles[c][0])/3
    print ang
    motion.angleInterpolationWithSpeed("Head", [0, 0.035], 0.1)
    tts.say(string)
    return ang, found

# Will take 1 picture (and return it)
def pic(_name, CameraIndex):
    videoClient = video.subscribeCamera(
        "python_client", CameraIndex, resolution, colorSpace, 5)
    naoImage = video.getImageRemote(videoClient)
    video.unsubscribe(videoClient)
    # Get the image size and pixel array.
    imageWidth = naoImage[0]
```

```
    imageHeight = naoImage[1]
    array = naoImage[6]
    im = Image.fromstring("RGB", (imageWidth, imageHeight), str(array))
    im.save(_name, "PNG")

# Funtion that will look for position of the ball and return it
def initial_scan():
    state = 0
    angleSearch = 60*math.pi/180
    ang = 100

    motion.moveInit()
    camIndex = 0  # Starts with upper camera

    state = 0
    while ang == 100:
        [CC, AA] = scan_area(angleSearch, camIndex)
        print CC
        ang, found = rotate_center_head(CC, AA)
        if found == 0 and camIndex == 1:
            state = state + 1
            camIndex = 0
            motion.moveTo(0, 0, 2*math.pi/3)
        elif found == 0 and camIndex == 0:
            camIndex = 1
        if state == 3:
            tts.say('I need to move to find the ball')
            motion.moveTo(0.3, 0, 0)
            state = 0

    else:
        motion.moveTo(0, 0, ang*7/6)
    pic(path + "ball_likely.png",0)
    [CC1, AA1] = take_pics(math.pi /9, camIndex)
    print CC1
    [ang, X, delta] = locate_ball(CC1, AA1)
    if ang == 100:
        camIndex = 2
    print 'Delta', delta
    print 'Ang', ang
    motion.moveTo(0, 0, ang*7/6)
    img=cv2.imread(path + "ball_likely.png")
    CM=CenterOfMassUp(img)
```

```
    return CM, delta, camIndex

def walkUp(cm, delta):
    idx = 1
    lowerFlag = 0
    print "Entering uppercam loop"
    motion.moveTo(0.2, 0, 0)
    while cm[0] < 420 and cm[0] > 0:
        pp = "ball_upfront"
        ext = ".png"
        im_num = path + pp+str(idx)+ext
        pic(im_num, 0)
        img = cv2.imread(im_num)
        cm = CenterOfMassUp(img)
        print cm
        if cm[0] == 0 and cm[1] == 0:
            # Scan the area with lower camera
            pic(path + 'lower.png', 1)
            img = cv2.imread(path + "lower.png")
            cm2 = CenterOfMassUp(img)
            lowerFlag = 1
            break
        else:
            alpha = (cm[1] - 320) * delta
            motion.moveTo(0.2, 0, alpha*7/6)
            idx = idx + 1
            continue
    if lowerFlag == 1:
        if cm2[0] == 0 and cm2[1] == 0:
            lostFlag = 1
            print 'I lost the ball'
        else:
            lostFlag = 0
            print 'I need to switch cameras'
    else:
        pic(path + 'lower.png', 1)
        img = cv2.imread(path + 'lower.png')
        cm2 = CenterOfMassUp(img)
        lostFlag = 0
    print "Exiting up loop"
    return lostFlag, cm2
    # motion.moveTo(0.15, 0, 0)
```

```
def walkDown(cm, delta):
    idx = 1
    pp = "ball_downfront"
    ext = ".png"
    print 'Entering lowercam loop'
    # motion.moveTo(0.2, 0, alpha*7/6)
    motion.moveTo(0.2, 0, 0)
    while cm[0] > 0 and cm[0] < 230:
        # motion.moveTo(0.2, 0, 0)
        im_num = path + pp+str(idx)+ext
        pic(im_num, 1)
        img = cv2.imread(im_num)
        cm = CenterOfMassUp(img)
        print im_num, cm
        if cm == [0, 0]:
            return 0, cm
        alpha = (cm[1] - 320) * delta
        motion.moveTo(0.2, 0, alpha*7/6)
        idx = idx + 1
    # Tilt the head so it can have a better look of the ball
    anglePitch = math.pi * 20.6 / 180
    motion.angleInterpolationWithSpeed("HeadPitch", anglePitch, 0.1)
    print 'Pitching the head'
    # The threshold of 300 is equal to a distance of 15cm from the ball
    # The robot will do a small walk of 7cm and exit the loop
    print 'Entering sub-precise with cm[0]', cm[0]
    while cm[0] >= 0  and cm[0] < 300:
        im_num = path + pp+str(idx)+ext
        pic(im_num, 1)
        img = cv2.imread(im_num)
        cm = CenterOfMassDown(img)
        print im_num, cm
        if cm == [0, 0]:
            return 0, cm
        if cm[0] < 350:
            alpha = (cm[1] - 320) * delta
            motion.moveTo(0.07, 0, alpha*8/6)
        else:
            break
        idx = idx + 1
    taskComplete = 1
    return taskComplete, cm
```

```
def getReady(cm, delta):
    # The ball should be at about 10cm roughly from the ball
    # Do the correction before it starts the loop
    idx = 1
    pp = "ball_precise"
    ext = ".png"
    im_num = path + pp+str(idx-1)+ext
    pic(im_num, 1)
    img = cv2.imread(im_num)
    cm = CenterOfMassDown(img)
    alpha = (cm[1] - 320) * delta
    motion.moveTo(0, 0, alpha*7/6)
    print 'Precising the position'
    print 'This is my cm[0]', cm[0]
    while cm[0] < 370:
        print 'Entering the loop'
        im_num = path + pp+str(idx)+ext
        pic(im_num, 1)
        img = cv2.imread(im_num)
        cm = CenterOfMassDown(img)
        print im_num, cm
        if cm == [0, 0]:
            return 0, cm
        if cm[0] < 405:
            alpha = (cm[1] - 320) * delta
            motion.moveTo(0.05, 0, alpha)
        else:
            break
        idx = idx + 1
    return 1
    # This should exit at a good distance to kick the ball

def kickBall():
    # Activate Whole Body Balancer
    isEnabled  = True
    motion.wbEnable(isEnabled)

    # Legs are constrained fixed
    stateName  = "Fixed"
    supportLeg = "Legs"
    motion.wbFootState(stateName, supportLeg)
```

```
# Constraint Balance Motion
isEnable   = True
supportLeg = "Legs"
motion.wbEnableBalanceConstraint(isEnable, supportLeg)

# Com go to LLeg
supportLeg = "LLeg"
duration   = 1.0
motion.wbGoToBalance(supportLeg, duration)

# RLeg is free
stateName  = "Free"
supportLeg = "RLeg"
motion.wbFootState(stateName, supportLeg)

# RLeg is optimized
effectorName = "RLeg"
axisMask     = 63
space        = mot.FRAME_TORSO


# Motion of the RLeg
dx      = 0.025                 # translation axis X (meters)
dz      = 0.02                  # translation axis Z (meters)
dwy     = 5.0*math.pi/180.0     # rotation axis Y (radian)


times   = [1.0, 1.4, 2.1]
isAbsolute = False

targetList = [
  [-0.7*dx, 0.0, 1.1*dz, 0.0, +dwy, 0.0],
  [+2.2*dx, +dx, dz, 0.0, -dwy, 0.0],
  [0.0, 0.0, 0.0, 0.0, 0.0, 0.0]]

motion.positionInterpolation(effectorName, space, targetList,
                             axisMask, times, isAbsolute)


# Example showing how to Enable Effector Control as an Optimization
isActive     = False
motion.wbEnableEffectorOptimization(effectorName, isActive)
```

```
    time.sleep(1.0)

    # Deactivate Head tracking
    isEnabled    = False
    motion.wbEnable(isEnabled)

    # send robot to Pose Init
    posture.goToPosture("StandInit", 0.5)

def set_head_position(_angle):
    fracSpeed = 0.2
    names = ['HeadYaw']
    motion.setAngles(names, _angle, fracSpeed)

def zero_head():
    motion.angleInterpolationWithSpeed("HeadYaw", 0, 0.1)

def main(robotIP, PORT=9559):

    #Wake up the robot
    motion.wakeUp()
    taskCompleteFlag = 0
    while taskCompleteFlag == 0:
        ballPosition, delta, camIndex = initial_scan()
        if camIndex == 0:
            zero_head()
            lost, CoM = walkUp(ballPosition, delta)
            if lost == 0:
                # Switch cameras
                time.sleep(0.2)
                video.stopCamera(0)
                video.startCamera(1)
                video.setActiveCamera(1)
                zero_head()
                # Walk to the ball using lower camera
                taskCompleteFlag, CoM1 = walkDown(CoM, delta)
                taskCompleteFlag = getReady(CoM1, delta)
            else:
                tts.say('I lost the ball, I need to rescan.')
                motion.moveTo(-0.2, 0, 0)
        elif camIndex == 1:
            # Switch cameras
            time.sleep(0.2)
```

```
            video.stopCamera(0)
            video.startCamera(1)
            video.setActiveCamera(1)
            zero_head()
            # Walk to the ball using lower camera
            taskCompleteFlag, CoM1 = walkDown(ballPosition, delta)
            taskCompleteFlag = getReady(CoM1, delta)
    kickBall()
    motion.rest()


if __name__ == "__main__":

    main(robotIP)
```

References

[1] G. Bradski. Opencv official documentation. `http://opencv.org/documentation.html`, 2000. Accessed: 2015-11-05.

[2] David Budden, Shannon Fenn, Josiah Walker, and Alexandre Mendes. A novel approach to ball detection for humanoid robot soccer. In *AI 2012: Advances in Artificial Intelligence*, volume 7691 of *Lecture Notes in Computer Science*, pages 827–838. Springer Berlin Heidelberg, 2012.

[3] Widodo Budiharto, Bayu Kanigoro, and Viska Noviantri. Ball distance estimation and tracking system of humanoid soccer robot. In *Information and Communication Technology*, volume 8407 of *Lecture Notes in Computer Science*, pages 170–178. Springer Berlin Heidelberg, 2014.

[4] Aldebaran Robotics. Naoqi 2.1 documentation. `http://doc.aldebaran.com/2-1/index.html`, 2005. Accessed: 2015-11-20.

[5] Peter Stne. Publications related to the robocup soccer simulation league, 2013. Available online: `http://www.cs.utexas.edu/~pstone/tmp/sim-league-research.pdf`.

[6] Belinda Teh. Taste of research report robocup: Ball tracking and velocity, 2011. Available online: `http://cgi.cse.unsw.edu.au/~robocup/2011site/reports/Teh-Ball-Tracking.pdf`.