Humanoid Robotic Systems - ”NAO, the Matador”

Andrés González, Wenlan Shen, Sherif Shousha, and Jesús Varona

I. Project Description

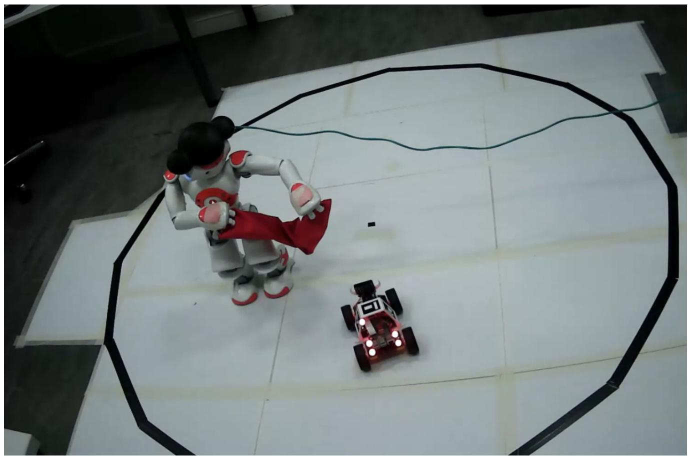


Fig. 1. NAO facing the bull

The goal of this project is to make a Soft Bank Robotic’s NAO into a *bull fighter* and a robot car [1] controlled by Raspberry Pi into a *bull*. In this setup, seen in Figure 1, both contenders are inside a ring drawn on the floor and the bull is replaced by a Raspberry Pi controlled robot car that charges for the humanoid’s cape. The NAO must track the movement of the bull, decipher its intention, juke when apropppiate and exclaim the outcome of every round.

This is the main routine of every round:

1) Both contenders (NAO and bull) should track each other while the bull is circling a ring drawn on the floor. The bull-car should move to its ready position and aim at the NAO’s red cape and both should communicate using LED signals.
2) NAO should inform an external operator when he detects that the bull-car is ready. On the go signal (“¡Adelante!”) from the external operator, the Matador should send a killer look (LED pattern) to the bull to provoke him.
3) Then the bull-car should charge along a linear collision trajectory with the cape.
4) The brave NAO Matador should in return move his body aside and let the robot-bull go past its cape at the very last moment, proudly exclaiming “¡Oléééé!”.
5) In the unfortunate case the Matador is not fast enough to avoid the bull, it should detect the collision with its bumpers, shouting “¡Ay Caramba!”. After each round, both the bull and the matador should re-position and prepare for the next run.

This project was implemented in C++ and Python with the use of the ROS (Robot Operating System) [2]. and the ArUco Library [3].

II. System Architectures

Since two different robots were used in the project, it was decided to deploy two different system architectures. Each one is catered to the respective robot’s hardware control and task.

A. NAO Architecture

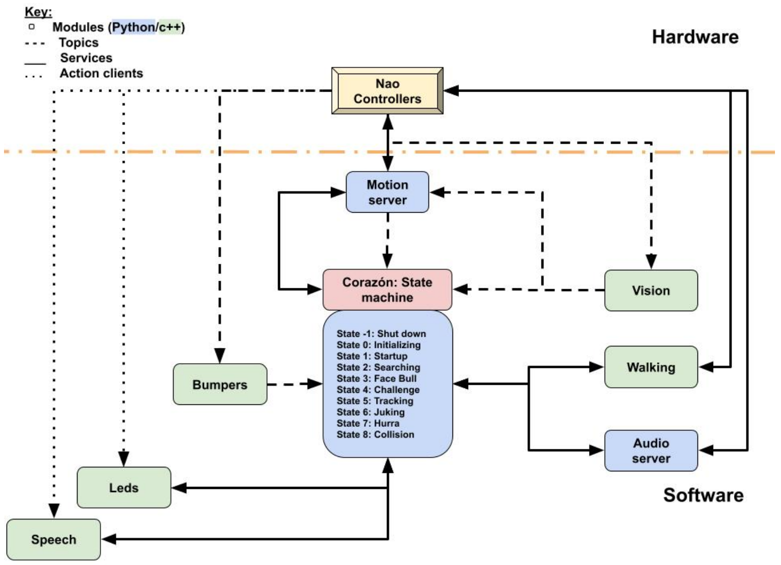


Fig. 2. NAO architecture: simplified node map

The NAO’s system architecture is *reactive* to guarantee a faster response to the stimuli provided by the bull. This was assured through three design decisions: First, callback functions for action performance tied to the arrival of a new message or service requests. Second, small message structures. Finally, the use of non-locking functions wherever possible.

The control architecture is defined as *Distributed and emerged*, which means that different subsystems (called modules from this point onward) run parallel, assuming responsibility for many of the robot’s high-level functions. The modules are coordinated by a central module, implemented in the form of a state machine. Communication with the robot’s hardware occurs distributively by multiple nodes through the use of server modules or dedicated messages. This architecture has a natural affinity to ROS’s nodal features and allows for low algorithmic complexity and a high level of abstraction for easier implementation.

Finally, at a local level, the modules are implemented in a task-based architecture. Each module is responsible for a small number of tasks, that are triggered once the state machine sends high-level commands together with the required parameters. Figure 2 illustrates the final node map in the aforementioned architecture.

B. Bull Architecture

The bull robot car has a *task-based* architecture, as shown in Figure 3.In each round of the bull fight, the bull needs to finish several tasks consecutively. For each task in the routine, the controller analyses necessary sensor data and commands the bull car to accomplish certain motions required by the task. The main controller and vision modules are implemented in a Raspberry Pi 4, using the ROS framework. In each iteration, the camera node publishes the results of the NAO detection, red cape tracking and LED detection algorithms. Consequently, the main controller module subscribes to these topics in order to control the bull's actuators. Details about the implementation of visual processing and control routine will be explained in section IV.

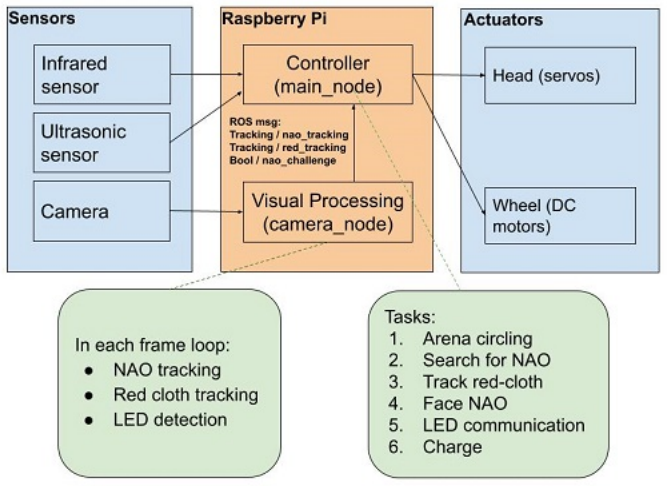


Fig. 3. Bull architecture: task-based architecture

III. NAO

A. Corazón: state machine

Figure 4 displays the simplified *Corazón: state machine* (Heart in Spanish) diagram. This module is responsible for all the high-level processing of the system. It is composed of a total of 10 states. States 0 and 1 initialize the machine and place it in state 2, searching for the Bull car. Following this, the main routine is triggered, which cycles endlessly until the operator stops the program. In state 2, the NAO moves its head and body looking for the bull's ArUco markers. When found, state 3 is triggered. Here, the Matador tracks the marker, using the angle between itself and the marker to rotate towards the Bull. If the opponent has also detected the humanoid robot, then it signals its intent by lighting on its blue LEDs. This action is detected by the Matador, triggering state 4. In this state the robot proceeds to execute the *Challenging* routine: It stops rotating, signals the operator for clearance, and if it recognises the command "Adelante", it swings the cape towards the bull and blinks his eye-LEDs at a particular colour and frequency, signalling the challenge.

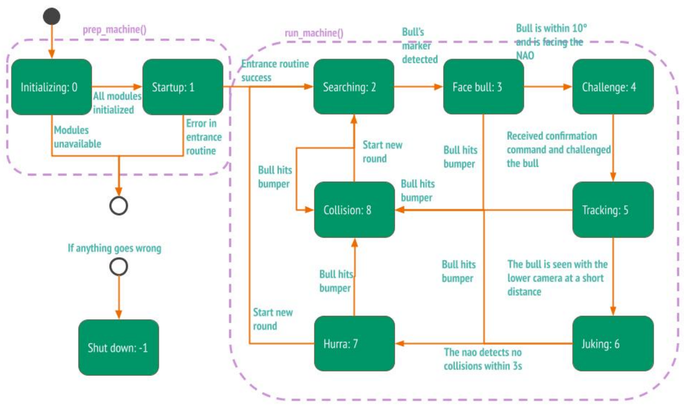


Fig. 4. Simplified Corazón state machine

If at this point the Bull detects the challenge, it signals approval by lighting on its red LEDs. The NAO recognizes the red light and enters into state 5: bull tracking. In this state, the robot moves to the right and places its cape to the left, allowing enough room for the bull to ram through while tracking the distance between itself and the bull's marker with its bottom camera. When the distance is below a threshold, the algorithm triggers state 6, where the NAO jukes its cape and waits for three seconds. Should nothing happen in this time, then a successful encounter is assumed and the machine enters into state 7 where the victory routine is executed. Here, the robot exclaims "¡Olé!", puts its arms in the air, blinks green and exclaims the current rounds won, before returning to state 2 again.

On the other hand, if at any given point the bull collides with one of the Matador's bumpers, it enters state 8 and assumes a failed round, starting the *Sad* routine. Here the robot drops his arms and head, swinging it sadly side to side while exclaiming "Ay caramba". After this, it also exclaims the current rounds won by each of the contenders, before returning to state 2. Finally, if there is an error or a manual stop command from the user at any point, the machine enters state -1, where it can safely shut down the system.

B. Motion proxy

This module server acts as the middleman between the control architecture and the NAO's hardware. It has two tasks. First to handle the joint movement requests from the main state machine, and second, to perform the frame transformations required to calculate the angle and distance from the ArUco markers to the humanoid's torso.

To achieve the former, when a high-level command is received, the command is matched to a subroutine, which is then executed. Subroutines are a series of motion commands followed by time delays using the angle interpolation control from Soft Bank Robotic's Joint Control API [4].

For the second task, before being transformed, the incoming vectors from the ArUco detection are extended to 4-dimensional vectors ${}_{opt}v$ by appending a 1 at the end. Then the same vector in torso coordinates ${}_{t}v$ is calculated through Equation 1, where ${}_{c}T_{opt}$ is the transformation matrix from the top or bottom optical frame to its respective camera frame and ${}_{t}T_{c}$ is the transformation matrix from the camera frame to the torso frame. The final translation vector is obtained by removing the 4th element.

$$ {}_{t}v = {}_{t}T_{c} * {}_{c}T_{opt} * {}_{opt}v \tag{1} $$

C. Walking

This module, as mentioned before, is implemented in a task-based control architecture and handles all of the system's calls to Soft Bank Robotic's Walk Control API [5]. The module is triggered when a command to either turn, move sideways or move forwards/backwards is received together with a motion parameter. Following this, the module publishes a walking goal, that the API fulfils. Additionally, the module supervises foot contact, cancelling motion if the feet hover for a long time.

D. Speech

The speech module handles the speech generation and recognition tasks. For speech generation, it can make the NAO say the words requested by the state machine by publishing the desired text to the NAO *texxt-to-speech* module. For speech recognition, a vocabulary, containing the words to be recognized by NAO, is set at the beginning. Recognition starts after the NAO says ”Preparado” by calling the service ”`recog_start_srv`” with an empty request. If the command ”Adelante” is heard, the recognition stops by calling the service ”`recog_stop_srv`”

Algorithm 1 Speech Recognition Routine

```
set vocabulary
NAO says “preparado”
starts recognition
**while** recognized word != ”Adelante” **do**
    pass
**end while**
ends recognition
proceed to challenge routine
```

E. Audio Player

The audio player server module handles loading and playing audio files stored in NAO based on the request by the state machine. There are two audio files stored in NAO´s memory unit, one for the word ”¡Olé!” and another for ”Ay caramba”. The two conditions that trigger the requests are: a successful bull avoidance or a hit to the bumpers (meaning bull collision).

F. Vision

The NAO vision module is responsible for recognizing the surrounding environment and collecting needed data, which is used in the state machine to make manipulation decisions. The vision module serves two main tasks - ArUco tracking and LEDs detection. The ArUco tracking method detects the arena center and the Bull car. Moreover, the module is responsible for collecting the location data used to calculate the detected object's angle and distance. The LED detection method is responsible for detecting the Bull's LEDs state in order to decipher the Bull's current intention.

1) ArUco tracking: The ArUco detection algorithm subscribes to the both top and bottom NAO camera. Both camera detects if there is ArUco marker inside the frame. If there is any marker detected, marker ID is used to decided whether the marker is on the bull body or on the ring center. This information is published along with the 3D position of the marker, which will be used to calculate the angle and distance from the NAO to the marker in the motion server. Figure 5 illustrates the algorithm's workflow.

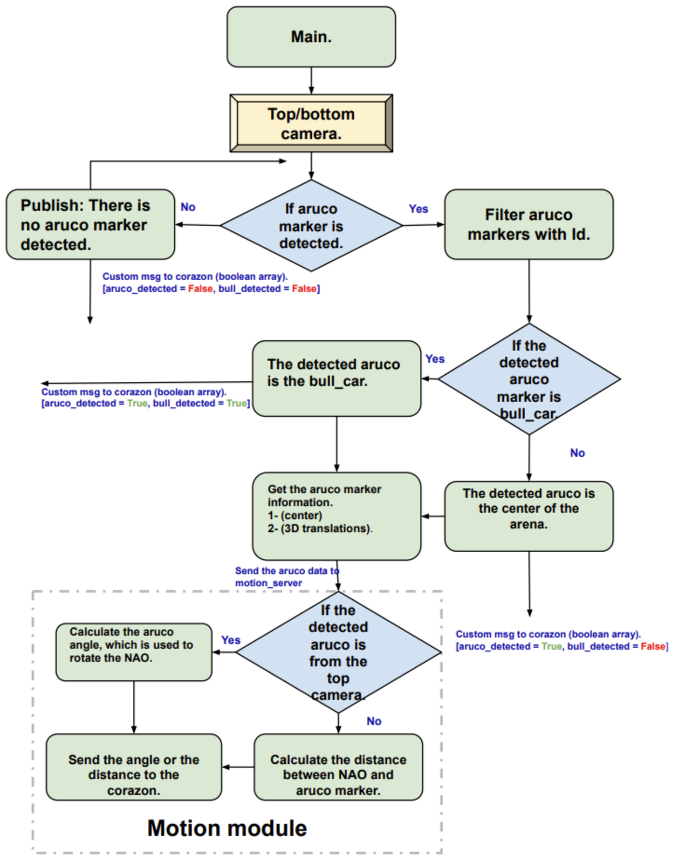

Main.
Top/bottom camera.
If aruco marker is detected.
No
Yes
Publish: There is no aruco marker detected.
Filter aruco markers with Id.
Custom msg to corazon (boolean array). [aruco_detected = False, bull_detected = False]
If the detected aruco marker is bull_car.
Yes
The detected aruco is the bull_car.
Custom msg to corazon (boolean array). [aruco_detected = True, bull_detected = True]
No
Get the aruco marker information. 1- (center) 2- (3D translations).
The detected aruco is the center of the arena.
Send the aruco data to motion_server
Custom msg to corazon (boolean array). [aruco_detected = True, bull_detected = False]
If the detected aruco is from the top camera.
Yes
Calculate the aruco angle, which is used to rotate the NAO.
No
Calculate the distance between NAO and aruco marker.
Send the angle or the distance to the corazon.
Motion module

Fig. 5. ArUco detection structure

2) LEDs detection: The robot LEDs' are used as a signaling method between the NAO and the Bull to communicate their intention. The LED desction part is included in the Vision module, which detects the Bull's LED color and uses the detected color to determine the Bull's state. Concretely, when the NAO detects the color red, the Matador starts the "challenge" routine.

IV. LEDs

This part subscribes to the NAO's LEDs topic, which is used to express the NAO's state. The states are represented by different LED colors and frequencies, which are generated depending on the interaction with the Bull. The eye-LEDs are blinking with different rates and colors in the challenge, start, hurrah, and sad routines depending on the action received from the state machine (Corazon).

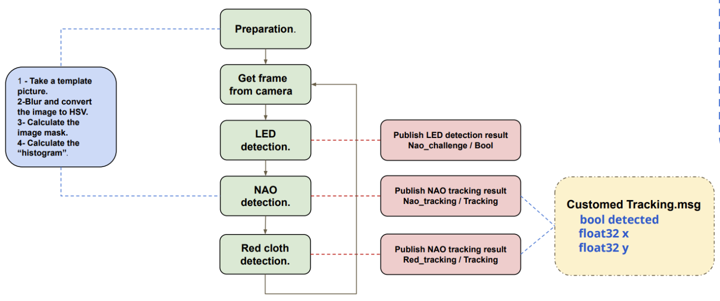
A. Bumpers

The Bumpers module consist of one callback function that subscribes to the NAO´s bumpers hardware controller and publishes whether they are pressed or not. This signals the state machine whether the NAO has been hit by the bull or not.

V. Bull

A. Vision

The vision node consists of three parts: NAO tracking, red cape tracking and LED detection. The tracking and detection results for three tasks are then published to three ROS topics. The vision overview is shown in Figure 6.

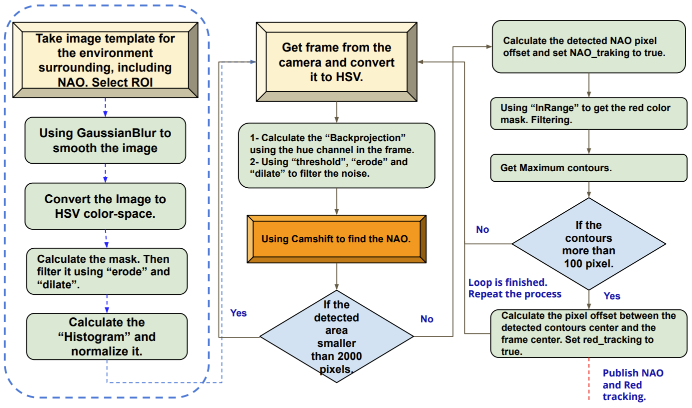


1 - Take a template picture.
2-Blur and convert the image to HSV.
3- Calculate the image mask.
4- Calculate the "histogram".
Preparation.
Get frame from camera
LED detection.
NAO detection.
Red cloth detection.
Publish LED detection result Nao_challenge / Bool
Publish NAO tracking result Nao_tracking / Tracking
Publish NAO tracking result Red_tracking / Tracking
Customed Tracking.msg
bool detected
float32 x
float32 y

Fig. 6. Vision node overview.

1) NAO Tacking: Camshift [6] method from OpenCV library is used to track the NAO. The algorithm starts with taking an template image for the environment, including the NAO. Then, the NAO is selected as the Region Of Interest (ROI). The ROI image is converted to HSV color-space, which is used to find the image's mask. The image's mask is used to calculate the *backproject* mask from the camera frame using the frame's hue channel. This mask is filtered using the erode and dilate functions [7] to remove noises.The filtered mask is then used in the *CamShift* function to track the NAO robot. The outcome of the *CamShift* function is not very robust to the environment changes despite a lot of improvement has been made. One more trick is added here to exclude false positive detection. Since the distance between NAO and the bull is 0.9m at most. The returned track window must be relatively large in the camera frame. The area of the track window is calculated in every iteration, if the area is smaller than certain threshold, then this detection will be excluded. Finally, a boolean variable is published to indicate if the NAO is detected. If the NAO is detected, then the pixel offset of object from the frame center is calculated and published through ROS. This will later been used for bull head tracking.

2) Red Cape Tacking: Blob detection [8] is used to detect the red color. This method is based on detecting all the target color's contours in the mask and search for the maximum contour's area to set it as the detection target. The function ignores all other smaller contours. The *InRange* function is used to calculate the mask. Since the red color has different degrees, the InRange function has to get the exact values of the cape's red color. To achieve that, the trackbars [9] are used to find the exact best values of the target red color and ignore all other red degrees. Similar to NAO tracking, a boolean variable, indicating whether the red cape is detected, as well as the pixel offset in both x and y direction are published through ROS.
The NAO and red cape detection algorithm is shown in Figure 7.



Take image template for the environment surrounding, including NAO. Select ROI
Using GaussianBlur to smooth the image
Convert the Image to HSV color-space.
Calculate the mask. Then filter it using "erode" and "dilate".
Calculate the "Histogram" and normalize it.
Get frame from the camera and convert it to HSV.
1- Calculate the "Backprojection" using the hue channel in the frame.
2- Using "threshold", "erode" and "dilate" to filter the noise.
Using Camshift to find the NAO.
If the detected area smaller than 2000 pixels.
Yes
No
Calculate the detected NAO pixel offset and set NAO_traking to true.
Using "InRange" to get the red color mask. Filtering.
Get Maximum contours.
If the contours more than 100 pixel.
No
Yes
Loop is finished. Repeat the process
Calculate the pixel offset between the detected contours center and the frame center. Set red_tracking to true.
Publish NAO and Red tracking.

Fig. 7. NAO and red cape detection algorithm.

3) LED Detection: LED detection is responsible for decyphering the NAO's intention using a frecuency and color measurement. The Bull isolates the pixels of blue-green LED color of NAO's killer look in the image by threshholding [10] in Hue and RGB color space. If the right color is detected, the blinking frequency is calculated by counting the time elapse when the LED is on. If the right color LED is blinking in the right frequency, that means the NAO is challenging the bull. A `true` is then published to the topic `/nao_challenge`.

B. Motion Control

The main controller node handles the control of actuators on the car using sensing data from the infrared sensor, the ultrasonic sensor and the camera node, so that the bull can accomplish certain tasks.To do this, the controller mainly needs to control the angle of two servo motors mounted on the bull head, as well as the speed of four DC motors connected to the car wheels. The motor control is done by setting different duty cycle of PWM (Pulse Width Modulation) for each motor.

The control routine for the bull robot car is shown in Figure 8 Each part of the routine is handled in dedicated algorithms, which will be explained in the following subsections.

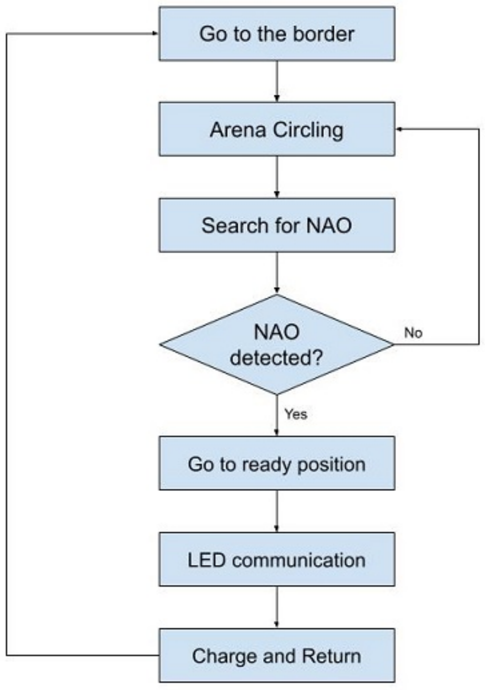


Fig. 8. Bull control flow

1) Find Border and Arena Circling: Three infrared sensors placed at front end of the robot car are used to detect the border of the arena. The arena is drawn as a black line on the floor. The output of the sensors can be read by the Raspberry Pi from GPIO. When the black line is detected, the sensor output is HIGH, otherwise it is LOW. Based on the the left, middle and right sensors' outputs, the direction can be determined using the algorithms I and II, so that the bull can find the border and consecutively track the black line.

The car direction is controlled by setting the rotation direction and speed on the wheel motors. If the two wheels on the left side rotate backwards while the ones on the right side rotate forwards, the car will turn left. Similarly, rotating left-side wheels forwards and right-side wheels backwards will turn the car to right.

TABLE I
FIND BORDER ALGORITHM

Infrared Sensor Output			**Car Direction**
Left	***Middle***	***Right***	
L	L	L	move forward
other states			start line tracking

2) Search Routine: After finding the border, the bull starts a search routine to look for its opponent NAO. Here the boolean variable `nao_detected`, which is part of the message subscribed from the topic `/nao_tracking`, is used to determine whether the search routine should end. In the search routine, it turns its head from left to right, then back to left to look for its target. If nothing is found after one search, it will then proceed along the border for 1 second then search again from a new position.

TABLE II
LINE TRACKING ALGORITHM

Infrared Sensor Output			**Car Direction**
Left	***Middle***	***Right***	
H	L	L	turn left slightly
H	H	L	turn left
L	H	L	move forward
L	L	H	turn right slightly
L	H	H	turn right
other states			stop

Algorithm 2 Search Routine

```
while NAO not detected do
    for $i = 0$; $i < 180$; $i++$ do
        $x\_angle \leftarrow i$
        if NAO detected then
            break
        end if
    end for
    for $i = 180$; $i > 0$; $i--$ do
        $x\_angle \leftarrow i$
        if NAO detected then
            break
        end if
    end for
    if NAO detected then
        break
    else
        Proceed along the border for 1 sec
    end if
end while
```

3) Head Tracking: After the bull detects its opponent, it should then always keep tracking its target - the red cape. Since the red cape should be inside the camera frame when the NAO is detected, the bull should use the pixel offsets of the red cape, which are obtained by subscribing the ROS topic `/red_tracking`, to control the position of its head, such that the red cape is always at the center of camera frame.

The pixel offsets of the object being tracked from the target position - frame center is shown in Figure 9. Both offsets in x and y direction have been normalized to the range $[-1, 1]$. In the head tracking function, two *proportional* controllers are used to control the head position in x and y direction respectively. The calculation of the new angle in each iteration can be described in the following equations.

$$\Delta\theta_x = K_{px} * x_{offset} \tag{2}$$

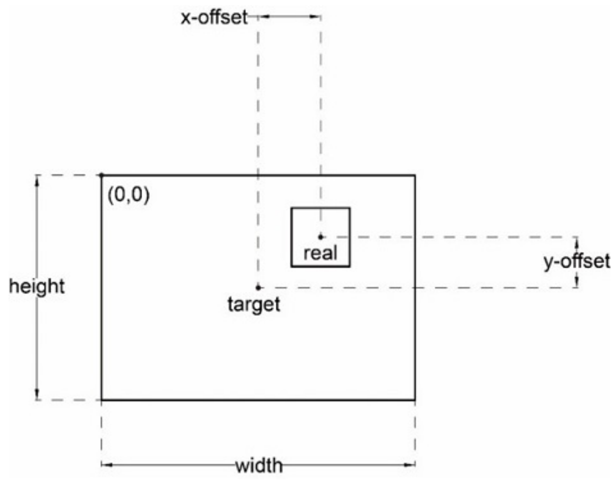


Fig. 9. Object tracking offsets

$$\Delta\theta_y = K_{py} * y_{offset} \tag{3}$$

$$\theta_{new} = \theta_{old} + \Delta\theta \tag{4}$$

4) Go To Ready Position: In this routine the bull enters the arena from the border and moves to a position right in front of the NAO. To achieve this, the head and the wheels are controlled at the same time. First, the controller needs to decided which direction to turn the wheels based on the head position. It should turn to the same side as the one the head is facing at. Then the wheels spin while the head keeps tracking the red cape until the head is facing front. This means that the NAO is now right in front of the bull, so it stops. To prevent the robot car from moving too fast and the camera loosing track of the red cape, the bull's speed of motion is set low and a pause is added every 10 iterations to assure that the head is able to track the red cape.
After reaching the ready position, the bull then lights on its blue LEDs and faces its camera up to gaze into the NAO's eye, waiting for a challenging killer look.

Algorithm 3 Ready Position

```
$counter \leftarrow 0$
**while** $x_{angle} < 85$ or $x_{angle} > 95$ **do**
    head_tracking()
    **if** $x_{angle} < 85$ **then**
        turn left
    **else if** $x_{angle} > 95$ **then**
        turn right
    **end if**
    **if** $counter = 10$ **then**
        wheels pause for 1 sec
        head_tracking()
        $counter \leftarrow 0$
    **end if**
**end while**
```

5) Charge and Return: In the charge routine all four motor speeds are set at the same high speed, such that the robot car follows a straight trajectory. However before the bull charges, it moves back and forth for several seconds, accumulating momentum, and additionally threatening its opponents.

The following motion then depends on the scenario. If the bull rams the NAO successfully, which it can tell by sensing the obstacle distance with its ultrasonic sensor, it then steps back and turns right to find the border. If the bull misses the NAO, then there's no obstacle in front of it. Therefore, the bull moves forward until it finds the border. Finally, the main routine starts over.

VI. System Integration

One of the most important part in this project is the interaction between both robots. There is no proper bull fight without a coordinated interaction between the Matador and the Bull. Figure 10 shows the integration of both systems. The order of events is shown in the Fight Routine block.

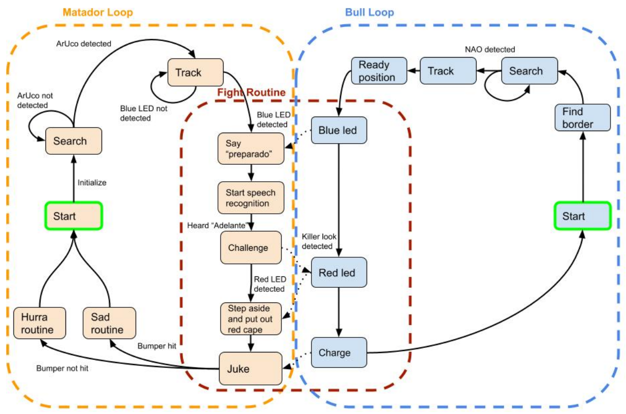


Fig. 10. Integration flow

VII. Obstacles and Conclusions

There were a number of obstacles that were faced during this project, some of them were:

1) Setting up the environment on Raspberry Pi 4. Raspberry Pi 4 is a relatively new hardware and a lot of libraries are still not compatible with Pi 4 on a Ubuntu Mate OS. After a lot of testing and switching between different operating systems, the Raspbian OS was installed on the Raspberry Pi.
2) Controlling the bull's wheels and head at the same time. The wheels can not spin too fast, since the camera would lose track of the red cape, nor too slow, because friction from the paper would stop the car. A lot of testing and some refinements on the algorithm, for example, adding a pause every 10 iteration, were done before it worked sufficiently.
3) From the NAO's point of view, its motion control presented a challenge - multiple movement commands

can be in a queue, and if one is triggered while the last one is still in execution, then both are run simultaneously, leading to unexpected behavior. As a solution, temporary delays were introduced, going against the system's reactive architecture.

4) Other major problems that lowered the performance of the robots were environmental conditions, such as variable light conditions, or a dark floor. This together with hardware limitations caused a number of issues. For example, during led communication, too much light meant that the robots could not detect the LEDs color pattern properly, or when the bull wanted to follow the outline of the ring, it struggled because of the color of the floor. To solve these problems, calibration was required before every experiment and again after a few hours of work. Additionally the arena was paved with white paper to increase the contrast with the ring.

In conclusion and as evidenced in this work, it is possible to automate bull fighting. The feasibility of this project provides an argument, however nuanced, that even a highly complex interaction between two agents such as the fight between a Matador and a Bull can be automated with relative ease. The main obstacles were all originated in the hardware, something that can be easily changed with better resources. Taking this result to its natural extension provides a possible alternative answer to the polemic that has accompanied the sport in recent years. In times of division and debate between traditionalism and animal protection, automation and the advent of robotics can also provide in bull fighting a new solution to an old problem as in many other fields.

References

[1] Freenove 4wd smart car kit for raspberry pi. https://github.com/Freenove/Freenove_4WD_Smart_Car_Kit_for _Raspberry_Pi.

[2] Documentation - ros wiki. http://wiki.ros.org/Documentation. (Accessed on 02/24/2021).

[3] Aruco: a minimal library for augmented reality applications based on opencv — aplicaciones de la visión artificial. http://www.uco.es/investiga/grupos/ava/node/26.

[4] Joint control — aldebaran 2.1.4.13 documentation. http://doc.aldebaran.com/2-1/naoqi/motion/control-joint.html.

[5] Walk control api — nao software 1.12 documentation. http://www.cs.cmu.edu/ cga/nao/doc/reference-documentation/naoqi/motion/control-walk-api.htmlcontrol-walk-api.

[6] Meanshift and camshift detection. https://docs.opencv.org/master/d7/d00 /tutorial_meanshift.html.

[7] Smoothing images using eroding and dilating. https://docs.opencv.org/3.4/db/df6/tutorial_erosion_dilatation.html.

[8] Opencv track object movement. https://www.pyimagesearch.com/2015/09/21 /opencv-track-object-movement/. (Accessed on 01/12/2021).

[9] python opencv hsv range finder creating trackbars. https://stackoverflow.com/questions/44480131/python-opencv-hsv-range-finder-creating-trackbars.

[10] Opencv image thresholding. https://docs.opencv.org/master/d7/d4d /tutorial_py_thresholding.html.