Programming a Duckiebot using Python

Report by: Abrahim Toutoungi

Research plan

I plan on researching about how accidents could be prevented using autonomous cars via the Duckietown project. The work is important because there are lives that are being lost that could be saved. I have already started conducting experiments as to how the safety varies from human drivers to autonomous drivers. I am going to do the research and produce results that I can be able to draw a conclusion from.

Abstract:

Duckietown is an open, inexpensive and flexible platform for autonomy education and research. The Duckietown project mixes AI with a mobile robot and ducks to create a wonderful learning experience for students as well as offering a wide range of functionalities at a low cost. Duckiebots contain only one sensor, a monocular camera and perform all processing onboard with a Raspberry Pi 2 and still is capable of so much. Duckietown is useful as a tool to educators as it is cheap and time efficient.

Introduction and Background Information

I was around a mere 10 years old when I was introduced to the world of coding by my father, whom I was watching program a software for his work. My newfound interest in coding remained dormant, like a seed left on pavement, until I joined the research program at MISE, where my dream and I were once again introduced. The ultimate goal of my group was to create an autonomous robot, known as the Duckiebot.

On my way to my third MISE day I saw a van, called trotros in Ghana, flipped all the way upside down and it was completely destroyed. During the same week I saw another trotro that was flipped all the way over and torn apart. It hurt to think that there were people inside the trotro. That is where I came up with the idea of using Duckiebots to help solve the real problem of accidents in Ghana.

The accident rate in Ghana is spiraling out of control with a total of 2,076 people dead; 3,300 pedestrian knockdowns; 12,166 travelers injured; 20,444 vehicles involved in road traffic accidents in 2017 alone. Human drivers are prone to many things that autonomous drivers are not prone to, such as drowsiness, intoxication and emotional conflict which allow them to behave fairly as long as the code is free of bugs.

First invented in 2016 at the Massachusetts Institute of Technology by Liam Paul and his team, the Duckiebot is a low-cost mobile robot whose sole sensor, a front facing camera, allows it to stay within a lane that wrapped around a track.[1] Its primary focus was to educate more people and allow them to enter the world of coding and autonomy to help solve the issues of autonomous cars that stand in the way of our future. This invention was just recently brought to Ghana via the MISE program, encouraging high school students to endorse computer programming. In this study, a Duckiebot was built and programmed for the first time in Africa.

[1] Liam Paull et al., “Duckietown: An Open, Inexpensive and Flexible Platform for Autonomy Education and Research” (IEEE, 2017), 1497–1504, https://doi.org/10.1109/ICRA.2017.7989179.

The Duckiebot uses a combination of Python and the Robot Operating System (ROS) to carry out its mission. From its implementation in 1989 to its world-changing applications today, Python rapidly grew as a high-level programming language, whose legibility appealed to the programming community as a more approachable way for people to enter the realm of programming.[2]

The ROS, on the other hand, acts as a middleware, and is a collection of tools, libraries, and conventions which facilitate the creation of complex and robust robot behavior across a wide variety of robotic platforms.[3] The ROS primarily uses nodes and topics. Essentially, topics are named buses over which nodes exchange messages. In general, nodes are not aware of who they are communicating with. Instead, nodes that are interested in data *subscribe* to the relevant topic; nodes that generate data *publish* to the relevant topic. There can be multiple publishers and subscribers to a topic.[4] Such a network is illustrated below:

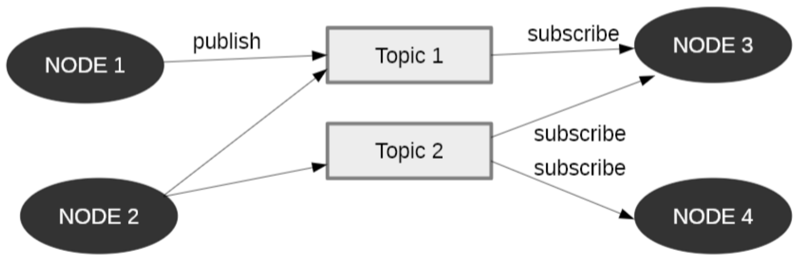


Figure 1 The solid black ovals are nodes and the grey rectangles are topics.

With the aforementioned tools, the Duckiebot should be able to maneuver the track in the diagram below such that it travels the track within the outer lane.

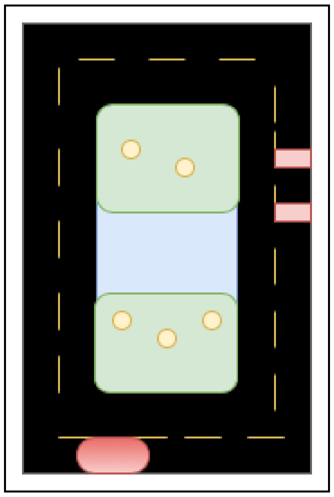


Figure 2 Duckietown: The Duckiebot, shown as a red ellipse, stays within the outer lane, between the dotted yellow line and the solid white outlines.

[2] Guido Van Rossum, "The History of Python: A Brief Timeline of Python," *The History of Python* (blog), January 20, 2009, http://python-history.blogspot.com/2009/01/brief-timeline-of-python.html.
[3] "ROS.Org | About ROS," accessed July 31, 2018, http://www.ros.org/about-ros/.
[4] "Topics - ROS Wiki," June 1, 2014, http://wiki.ros.org/Topics.

In order to stay within a lane, the Duckiebot must first look for the white and yellow lines on the road; that is, it must perceive its environment. For that, the Hough transform technique was used—initially being used for machine analysis of bubble chamber photographs—by converting pixels on the borders of the indications on the road into lines (two coordinates, the start point and the end point).[5] With said two points, a vector can be constructed to represent a solid line, enabling the Duckiebot to see it as a clear barrier and use it for measurements. Please see "Applying Vectors" section in appendix for more information.

After that, it runs a series of probabilities to place itself on an inbuilt map of the track called the "Global Map"; to localize itself. The program must then be able to determine what action to take next in order to move forward and turn corners while staying within the lane, which ultimately dictates how the motors behave. This motion control is possible by supplying the motors with a specific voltage that allows it to move at a certain speed.

This see-think-act mechanism is illustrated in the diagram below:[6]

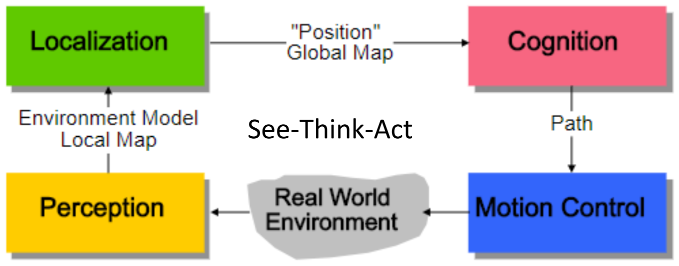


Figure 3 The program must go through a cycle of perception, localization, cognition and motion control in order to perform its mission.

In order to analyze data from the camera, an Open Source Computer Vision Library (OpenCV) is used. OpenCV, known in Python as *cv2*, uses BGR values, which means each pixel has three values which can be represented as a vector. The values of each of the blue, green and red can be between 0 and 256, with 256 not included as a result of Python's zero-based index. On this range, black is expressed as (0, 0, 0), and white as (255, 255, 255). This particular range is as a result of the BGR values being 8 bit each. Thus, the range for each individual color is 0-255, as 2^8 = 256 possibilities.

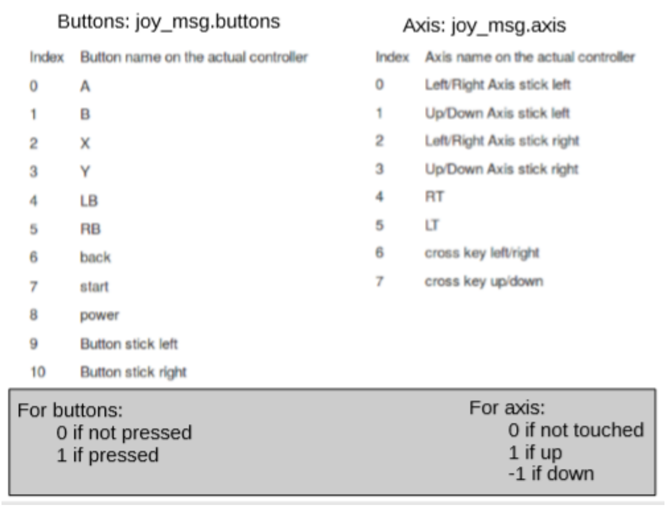
<u>Methodology</u>

In order to build a Duckiebot we must first understand how a robot works in general. Using three main devices to work: the sensors, the actuators, and the processors, autonomous mobile robots always ask themselves three fundamental questions: Where it is, where it should go, and how to get there. The

[5] "CS6640-F2012-HoughTransform-I.Pdf," accessed July 31, 2018, http://www.sci.utah.edu/~gerig/CS6640-F2012/Materials/CS6640-F2012-HoughTransform-I.pdf.
[6] "Perception 4 Sensors Uncertainty Features Localization Cognition - Ppt Download," accessed July 31, 2018, https://slideplayer.com/slide/5864185/.

Duckiebot uses two wheels as its actuators along with a Raspberry Pi for computation and a single monocular camera as its sensor.

To drive the Duckiebot with a joystick, we use a certain table, found below, that allows us to know which button translates to what command on the computer.



Buttons: joy_msg.buttons

Index	Button name on the actual controller
0	A
1	B
2	X
3	Y
4	LB
5	RB
6	back
7	start
8	power
9	Button stick left
10	Button stick right

Axis: joy_msg.axis

Index	Axis name on the actual controller
0	Left/Right Axis stick left
1	Up/Down Axis stick left
2	Left/Right Axis stick right
3	Up/Down Axis stick right
4	RT
5	LT
6	cross key left/right
7	cross key up/down

For buttons:
0 if not pressed
1 if pressed

For axis:
0 if not touched
1 if up
-1 if down

Figure 4 Every button and axis is represented by a particular index from which commands are referred.

We use the nodes and topics explained above to get the Duckiebot to respond to our commands.

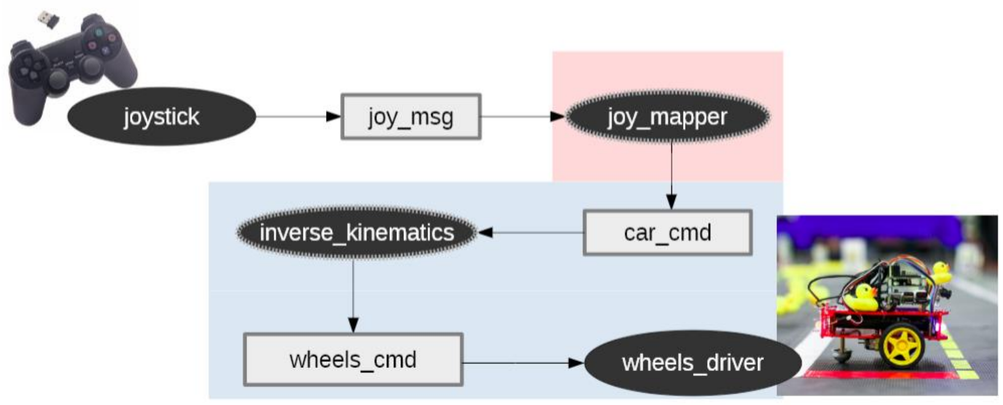


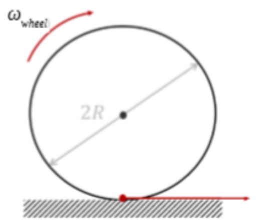
Figure 5 Map of the processing system of the Duckiebot.

In the code, our linear velocity is expressed as *car_cmd.v*, while our angular velocity is expressed as *car_cmd.omega*. To ensure smooth control of the robot, four essential directions of motion are required: Forwards, backwards, left, and right. If the axis, or control rod, is pushed up, there should be a positive *v_gain*, which is forward linear velocity. In turn, there should be a negative *v_gain* if it is pushed down, which is backwards linear velocity. It follows that if the axis is untouched, the Duckiebot should stay where it is. The same things go for the angular velocity, denoted as *omega_gain*. *k_d* and *k_ Θ* are both constants that essentially control the speed and responsiveness at which the Duckiebot corrects its path.

We use inverse kinematics, which is the calculation of linear velocity from angular velocity, to get how fast the wheels should turn for the Duckiebot to move at the desired speed. This is demonstrated in the diagram below:

Car linear velocity : v_{car}
Car angular velocity : ω_{car}
Wheel radius : R
Baseline : B

Wheel angular velocity : ω_{wheel}



$$\omega_{wheel} = \frac{v_{car} \pm (\omega_{car} \cdot B/2)}{R}$$

+ if right wheel
- if left wheel

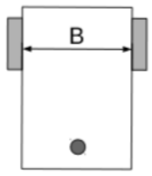


Figure 6 Formula for calculating angular velocity of wheel, ω_{wheel} ,necessary to result in linear velocity v_{car}.

The motors move with a voltage called the duty cycle, in order not to burn the motors we use limits so that the motors will not be supplied with more voltage than they can handle. Gain and trim are also used in order to control the voltage. Gain is the general voltage supplied to both wheels together, while the trim is an increment or reduction in voltage to a specific wheel. The latter allows for steering as well as wheel calibration since the two motors controlling the wheels were not perfectly identical, resulting in unequal wheel velocities and thus instability.

$$u_r = \omega_r * (g+t)/k$$
$$u_l = \omega_l * (g-t)/k$$

K : motor constant. Translates ω to u with $u = \omega/k$

g : gain. Allows to control the general speed.

t : trim. Allows to calibrate the two motors.

Figure 7 Formulas for right and left wheel velocities calculated with gain and trim values.

we coded the Duckiebot so that if you input a gain that would cause the motors to be supplied with a voltage greater than it can handle then it sets it to the upper limit, which is the maximum speed.

The Duckiebot has a single sensor (a monocular camera), two wheels, two motors, four chips (a raspberry pi 2, two motor chips and one LED chip), a frame, LED lights, a multidirectional wheel and a battery. We started by assembling the robot (link for instructions). Then we started programming the Duckiebot to follow a lane around a track.

We started the program by learning a prerequisite about some math and science, such as basic calculus, matrices, vectors, angular and linear dynamics, and probabilities. For more details on each of the aforementioned topics, please refer to the Appendix below.

A key concept for the Duckiebot's localization is its ability to analyze images from the onboard camera. Images can be represented as matrices of BGR values, known as pixels. To do that, we first coded the Duckiebot to be controlled by a game console controller. Then, in order to autonomize the Bot, we started coding the line detector which detects lines based on the difference in neighboring pixels in a greyscale version of the image. It then uses the Hough transform which goes point by point and determines whether the points are collinear or not. When it has the lines then it sets the color back to yellow and white. Then the ground projection is used to see where the lines are in respect to the Duckiebot.

There are two types of camera parameters, intrinsic and extrinsic. The intrinsic are the lens effect, the opening angle, the sensor orientation. The extrinsic are the position and orientation of the camera with respect to the ground and the Duckiebot. Now that the camera is in a set position and orientation we can assume that our intrinsic parameters are fine and then we calibrate the camera by using a checkerboard that has a known distance of space between squares, as demonstrated below:

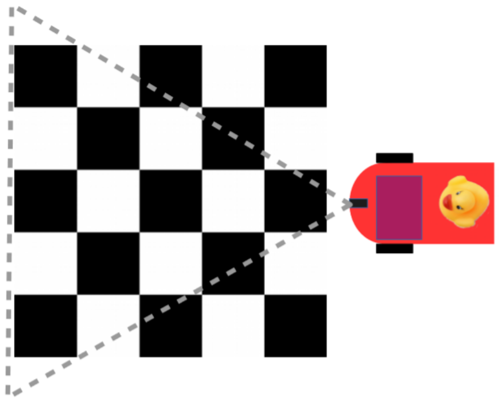


Figure 8 Checkerboard calibration pattern. Each square is of known dimensions. For example, 5cm by 5cm squares.

We calibrate by moving the checker board up and down, closer and further and then by tilting it about different axes. This allows the camera to correct the curve effect on the sides. Next we place the Duckiebot in a known place and face the camera towards the checkerboard so that it can know exactly where each pixel is on the ground. After that, the motion control comes in to make the Duckiebot move away from the yellow and white lines. The robot goes left from the white lines and right from the yellow lines towards the center of the lane. We used vectors to calculate how far the robot is from the middle of the lane and at what angle it is from the 0 degree normal. Then we used that to let the robot see how far it is from each line.

To ensure to have the best line detection possible the system uses a so called anti-Instagram algorithm. This algorithm determines a color transformation such that the input for the line detector has no color variation. This is important because based on color the Duckiebot knows whether it is a middle or a side line. The algorithm tries to minimize the influence of external illumination variation (scattered sunlight, different colors of light sources, etc…).[7]

<u>Experiments</u>

After the Duckiebot was assembled and coded, experiments were conducted to see how well the Duckiebot could be optimized and to see whether humans are safer drivers than robots.

Every experiment undoubtedly has to have a question, a hypothesis, results, and a conclusion. All the experiments below shared a common and simple rule: Dropping the duck or leaving the track disqualifies the trial. Also, any time, in seconds, that was spent on the other lane was multiplied by 1.5 and added to the Duckiebot's final time.

There were three main group of subjects: the automated Duckiebot, the humans who controlled the Duckiebot through direct vision, and the humans who controlled the Duckiebot by viewing from the camera's perspective through a monitor. The latter was intended to replicate a driving situation: where the driver is only able to fully view their front while relying on mirrors for side and rear views. The fully automated Duckiebot was intended to replicate a relatively reliable autonomous vehicle, while the humans controlling via direct vision were used to demonstrate the susceptibility to error that all humans inevitably possess.

<u>First experiment</u>

The first experiment investigated how fast the Duckiebot could go around the track five times with autonomous control, human direct control, or human control through a monitor. The question to be answered was "Which of the three drivers can score the lowest lap time?" We repeated the experiment five times because applying the law of large numbers puts us closer to the actual average. Each trial has one chance after disqualification to redeem itself. The amount of times the Duckiebot crosses over will be recorded.

The hypothesis of the experiment was that when the gain is at its highest, the Duckiebot would complete the 5 laps the fastest.

<u>Quantitative results</u>

[7] "Anti-Instagram: Intermediate Report," accessed August 2, 2018, https://docs.duckietown.org/class_fall2017_projects/out/anti_instagram_intermediate.html.

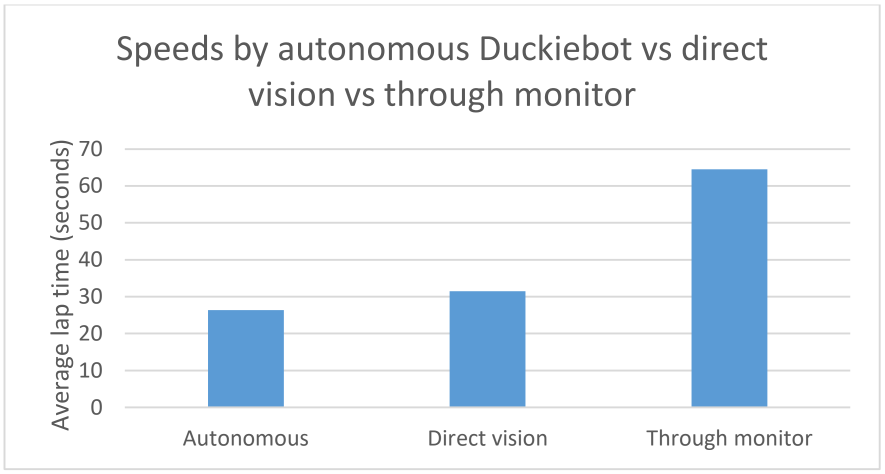


Figure 9 Bar chart showing that autonomous Duckiebot was faster than both direct and indirect control methods, with the indirect control method being the slowest.

<u>Qualitative observations</u>

It was observed that the robot cannot move too fast as the images will not be processed fast enough, causing the Duckiebot to have seemingly late reactions. Thus, there was a specific level of gain at which the Duckiebot performed most optimally and quickly. When it comes to speed, the Duckiebot was a few seconds faster than the human due to the fact that it never crossed any lines. Thus, it did not accumulate penalties.

<u>Second experiment</u>

In another experiment we tested the Duckiebot in different lighting situations. We used k_d and k_ θ values of -25 and -8 respectively, with a gain of 1, since these values yielded an almost perfect trial. Below are images illustrating how the lighting was varied:

<u>Quantitative results</u>

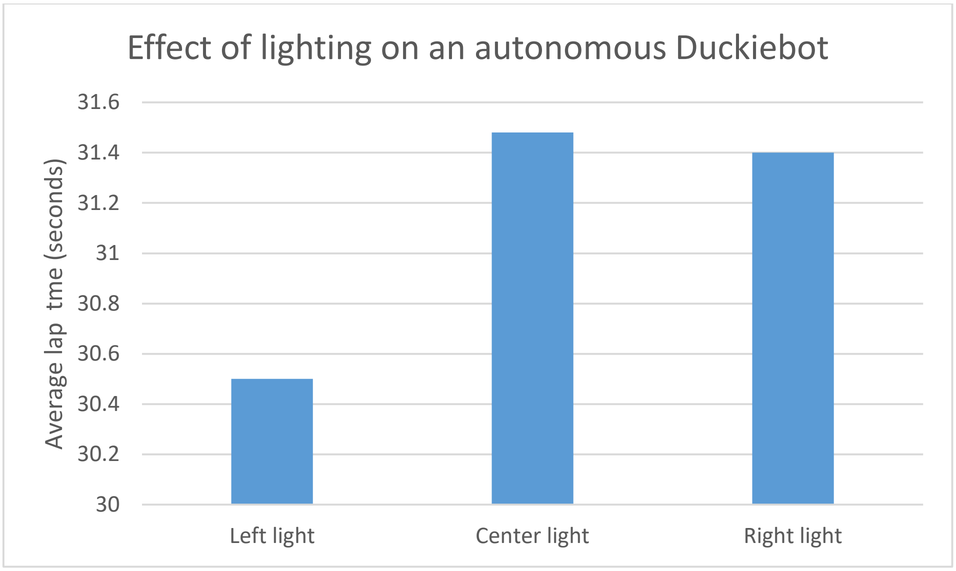


Figure 10 Bar chart showing the effect of lighting on an autonomous Duckiebot

Qualitative observations

The Duckiebot is not too robust as turning the lights off can hinder its performance. This is perhaps because it only has one sensor at the front. Using a flashlight doesn't help the Duckiebot either, as it will see the road as plain white, which renders the anti-instagram filter useless since there is no data that the algorithm can work with. On the other hand, a human could see using a flashlight, or even with relatively dim light.

Third experiment

This experiment merely investigated the number of times each group of subjects crossed any lines, along with how long the Duckiebot remained out of the correct lane. In this case, crossing the line is counted as an equivalent of a road traffic accident.

Quantitative results

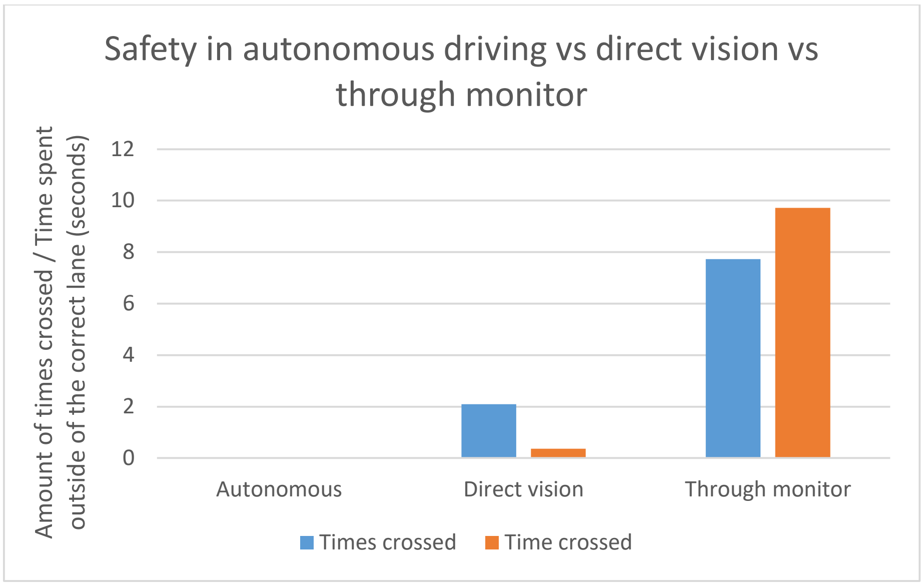


Figure 11 Bar chart showing how the autonomous Duckiebot is faster and safer than any of the human-controlled methods; whether through direct control or through viewing from a monitor.

<u>Qualitative observations</u>

After many attempts we found the ideal parameter levels for the safest ride around. The Duckiebot was able to use those parameters to complete 5 laps without crossing a single line, as shown in the diagram above. The humans, on the other hand, crossed lines and stayed outside of the correct lane significantly more [illegible]n controlling through a monitor than when using direct vision.

<u>Interpretations:</u>

The results gathered clearly prove that an autonomous car is much more efficient, while being safer as well. When the humans were driving, they crossed over many more times by making mistakes on the controller and not turning at the right time or speed.

With that being said, the second experiment demonstrated that this Duckiebot relied too much on perfect conditions. Its performance plummeted under slightly dimmer lighting situations; situations which a human could have easily seen through. This is a potential hazard as an autonomous vehicle passes through different lighting environments. Perhaps this dependency on optimal conditions could have been reduced with a higher-end monocular camera with wider aperture, or even a set of infrared blasters mounted onto the Duckiebot that would illuminate its path with light that is invisible to the naked human eye. That, however, would considerably increase the cost of production.

Whether or not the extra-cost-to-performance ratio of the automated Duckiebot is worth it depends on so many factors that make the answer to this question beyond the scope of this investigation. For example, economies like that of Ghana would struggle to afford such vehicles at their current state, resorting to other means to pay for them which may result in deep debt or instability of currency. Furthermore, since this Duckiebot is unequipped with the necessary sensors and code to make it stop at a close proximity with an obstacle or follow traffic light rules, the comparison between this experiment and the real world may not be entirely accurate nor fair.

Possible improvements

In hindsight, there are a few possible enhancements that could be done to the experiments above which would have made the results significantly more reliable.

Firstly, it seems rather strange that the Duckiebot did not make a single error. Indeed, this is not entirely the reality when it comes to the real world, where automated vehicles are prone to suffer from glitches or bugs in their complex programming. Thus, the automated Duckiebot should have been tested with four more sets of five laps, so that the total number of sets would be equal across all three subject types. That way, the consistency and reliability of the Duckiebot's programming could be truly tested.

Another way of testing consistency and reliability could be through adding more sensors and code to make the Duckiebot stop when it comes close to another object, or to follow traffic rules. With such a processing load, the safety of such automated systems could be more fairly and realistically compared to that of a human.

With regards to the subject group that controlled the Duckiebot through the monocular camera's perspective and a monitor, the existence of a slight delay due to the wireless connectivity and processing may have very well affected the overall performance. That is, the delay could have resulted in more line crossings, or accidents, due to late turning responses.

Conclusion

The objective of this research was ultimately successful: Coding a basic Duckiebot. However, there are several more features that could be unlocked by extra sensors and coding that make the concept of the Duckiebot more than just a small robot moving around a fictitious town called Duckietown. Indeed, there are many more questions to be answered as well, including, but not limited to, whether or not the Duckiebot could be an insight to a not-so-distant automated world.

Bibliography

"Anti-Instagram: Intermediate Report." Accessed August 2, 2018. https://docs.duckietown.org/class_fall2017_projects/out/anti_instagram_intermediate.html.

"CS6640-F2012-HoughTransform-I.Pdf." Accessed July 31, 2018. http://www.sci.utah.edu/~gerig/CS6640-F2012/Materials/CS6640-F2012-HoughTransform-I.pdf.

Paull, Liam, Jacopo Tani, Heejin Ahn, Javier Alonso-Mora, Luca Carlone, Michal Cap, Yu Fan Chen, et al. "Duckietown: An Open, Inexpensive and Flexible Platform for Autonomy Education and Research," 1497–1504. IEEE, 2017. https://doi.org/10.1109/ICRA.2017.7989179.

"Perception 4 Sensors Uncertainty Features Localization Cognition - Ppt Download." Accessed July 31, 2018. https://slideplayer.com/slide/5864185/.

"ROS.Org | About ROS." Accessed July 31, 2018. http://www.ros.org/about-ros/.

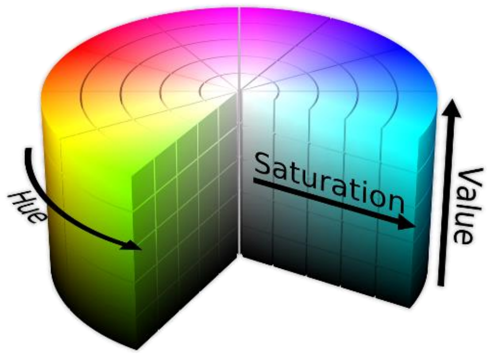
Rossum, Guido Van. "The History of Python: A Brief Timeline of Python." *The History of Python* (blog), January 20, 2009. http://python-history.blogspot.com/2009/01/brief-timeline-of-python.html.

"Topics - ROS Wiki," June 1, 2014. http://wiki.ros.org/Topics.

Appendix

1. Applying Vectors

Vectors were used in defining important functions such as the “detectline” function which, as its name suggests, detects lines. Line detection was programmed by taking an image with BGR (Blue, Green, Red) values and converting it to HSV (Hue, Saturation, Value) in order to be able to filter out the desired colors easily. The code basically takes an image and then converts it from BGR to HSV then uses ranges of colors to find the yellow and white areas.



After obtaining the resulting black and white image, we can detect areas where the difference between neighboring pixels is greater than 0, since every pixel is either black or not black, any non-black pixel that is in contact with a black pixel is a line on the border of an object. Since the lanes are either yellow or white, their respective colors are the only relevant ones to detect.

The dot product of two vectors can be used to calculate projection which is helpful in the localization part of the think-see-act process by calculating how far the bot is from the lines around it. A unit vector is a vector divided by its length.

2. Matrices

A matrix is an array of numbers, arranged in columns and vectors. Images can be represented as pixels, which can be represented as vectors. From that we can say that an image can be represented as an array of vectors; essentially a matrix, which ultimately means an image can be represented by a matrix.

3. Calculus

Two fundamental concepts were kept in mind:

1) An integral is the area under a curve. It uses infinitesimally small rectangles to calculate the area.
2) A derivative is the rate of change of a curve, the derivative of a straight line is its gradient.

4. Angular and Linear Dynamics

While angular dynamics calculates the rotational velocity of a wheel, linear dynamics determine how said rotational velocity is translated into linear velocity.

5. Probability and beliefs

Bayes' theorem was used to get an idea of where the robot is at all times. Our current idea of where we are is the prior ($p(d, \phi)$). The probability of observations m is the likelihood ($p(m|d, \phi)$). To get the posterior ($p(d, \phi|m)$) we use the formula:

$$p(d,\ \varphi|m) = (p(m|d,\ \varphi)\cdot p(d,\ \varphi))/p(m)$$

"The probability of d and phi (the posterior) while knowing m, is equal to the prior multiplied by the likelihood"

The Baye's theorem returns all the possible locations the Duckiebot could be at. The one most voted for, or most probable is used as its location. A belief map is a graph that is used alongside the Baye's theorem to predict the location of the bot. Every distance and angle votes as to where they are in relation to the robot and then the points are plotted on the belief map, the point that has the highest votes is the location the bot will use. After every prior the points must be blurred, what this means is that they should be spread out around the main point in order not to end up with only one point on the map.

In probability theory the law of large numbers is a theorem that describes the result of performing the same experiment a large number of times. This helped us understand that even if you have many random numbers and you keep adding to them and finding the average, the result will get closer to the actual average of the range. This is due to the elimination of any random errors. For example, if random numbers between 0 and 5 are being generated, the average will keep getting closer to 2.5 as the number of times the experiment increases.

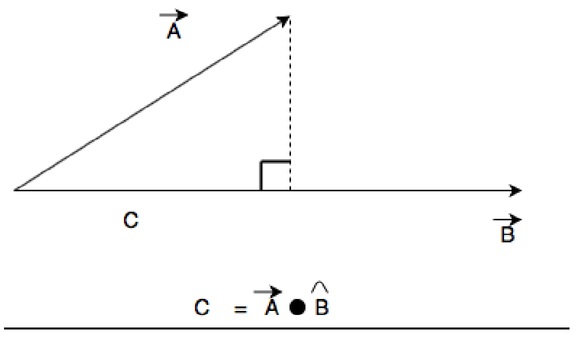


Figure 12 (Projection) Where B with hat is the unit vector of B and C is the dot product of Vector A and the unit vector of B