Pacerbot

Autonomous Unmanned Systems Stream Summer Report

By Aaron Barlev and Dr. Derrick Yeo

August 11th, 2016 - College Park, MD

I. Introduction

Human beings are social creatures. This desire for interaction with others extends to exercise as well. From casual joggers to Olympic athletes, no one likes to run alone. When a training partner is unavailable - the Pacerbot may be able to provide a substitute.

The desire to train with a partner extends beyond the social aspect for professional athletes. Self-pacing during workouts tends to be mentally taxing, especially towards the end when physical exhaustion begins to set in as well. When these athletes are pushing their bodies to the edge, it can be difficult to monitor and adjust the pace at the same time. In addition, human beings are naturally imperfect, and can make mistakes. A robotic pacer would provide a solution to these issues and take the pressure to hit the right times off of the athlete.

II. Technical Challenges Faced

Information Delay:

The original design for the Pacerbot used a small webcam attached via USB to the Raspberry Pi. However, within this setup, there was a large delay between any movement and the reaction to the movement made by the controller (> 500ms) which made line following very difficult. To solve this problem, we purchased a 5 megapixel Raspberry Pi Camera Module; made specifically for the Raspberry Pi with its own designated 15 pin connector. Additionally, I resized the video feed to 320 by 240 pixels. This solved the delay issue and brought down the processing time per frame to less than 10ms.

Track Inconsistencies:

Running tracks are not without imperfections. White specks and blobs can easily throw off the line detection algorithms, and often did in the early stages of development. The solution to this problem was a pixel averaging function, which is set to take a 15 by 15 pixel area and take the average value. By doing this, the impact of imperfections in the track is minimized.

Variable Lighting:

The fact that the lighting of a track tends to vary greatly caused some difficulty when writing line-detecting algorithms. Our solution was to calculate the average brightness of the image and use that value to calculate a threshold, which would be used to find the line (a large majority of running track surfaces sport a red track with white lines).

Markings on the Track:

Other markings on the track surface itself have the tendency to confuse the line-detection program. Currently, our solution scans the image for any potential lines, and picks the one that best fits what it expects. This is a work-in-progress, which is visualized in the Vision Algorithms section.

III. Sub-system description

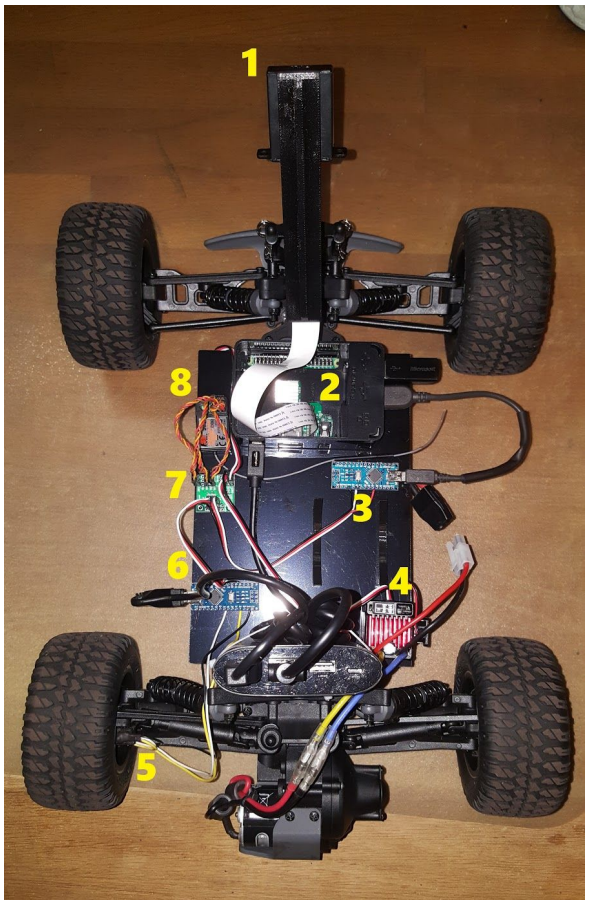
Embedded Computing Architecture:

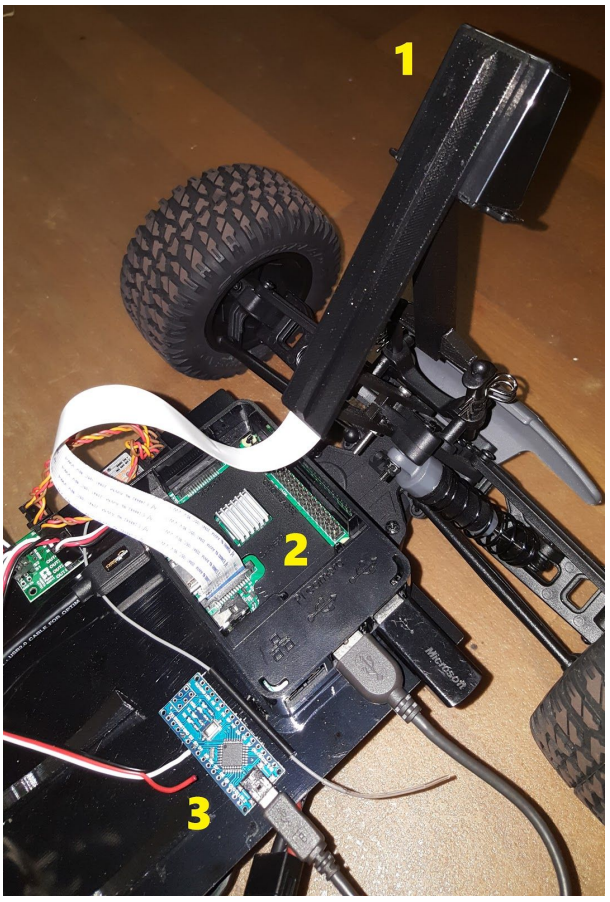
The front of the vehicle sports a Raspberry Pi computer (RPi) with a hardwired RPi Camera Module. We are able to access the RPi by mirroring its display to a laptop using an ethernet cable. This setup manages all image processing and command declaration. Basically, the camera sends the RPi the current image of the track, which decides what action to take based on the current state of things.

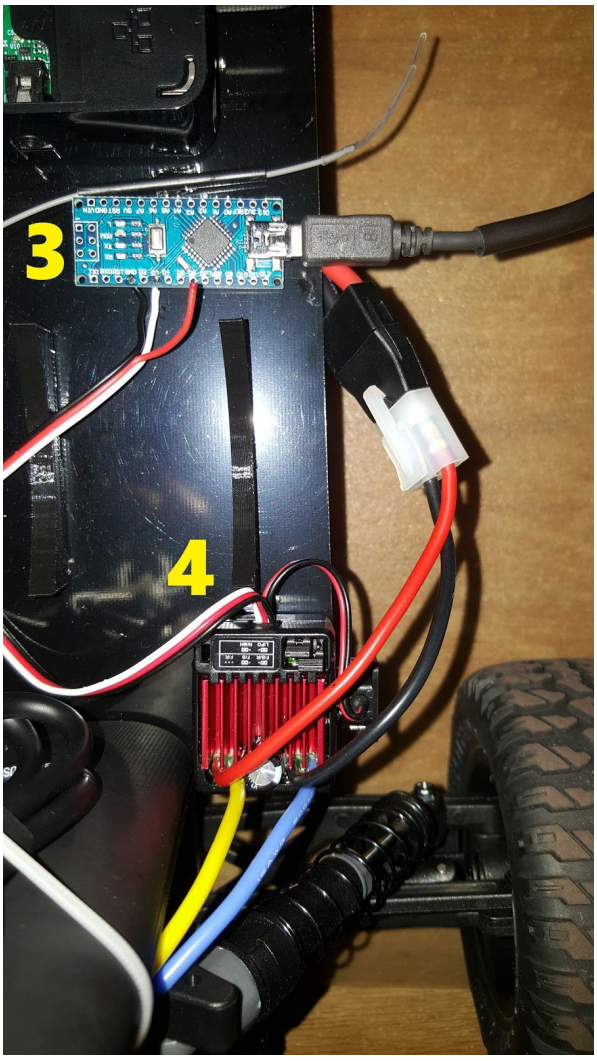
From there, the command is sent to an Arduino Nano microcontroller (MCU). The MCU has control over the steering mechanism of the car. Once it has interpreted its instructions, it will adjust the servo accordingly.

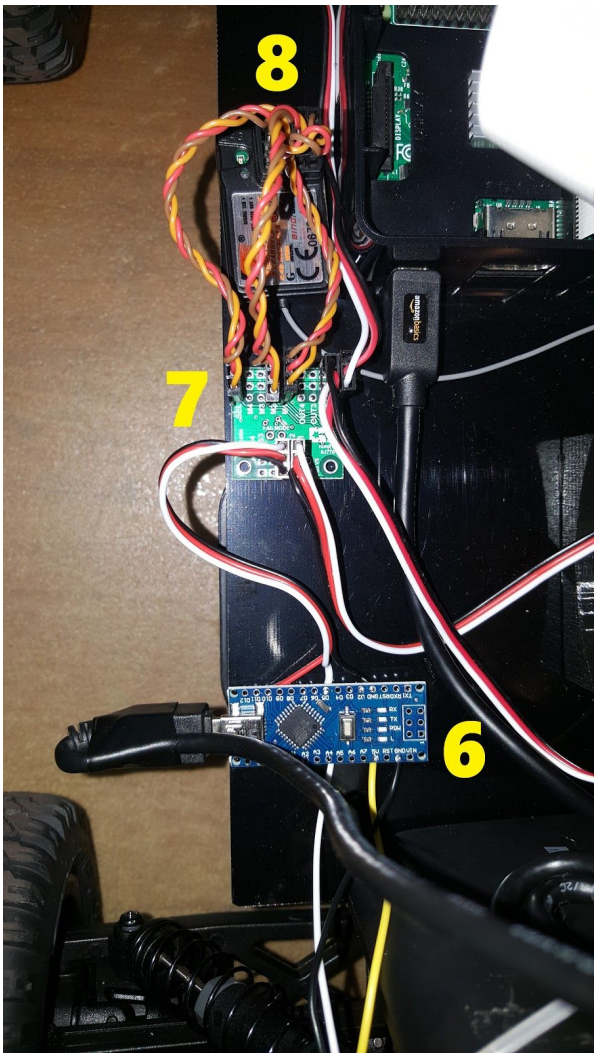
The back end of the platform houses the velocity control system. Inside the hub of the wheel rests a Hall effect sensor and eight mounted magnets, and on the plate sits another MCU. The sensor is able to detect the passing of each magnet - and therefore the RPM of the wheel and velocity of the car. Independent of the RPi, the MCU calculates the vehicle's speed, and supplies more or less power to the motor as needed.

1. Top view of the Pacerbot, depicting all electronic components. The vehicle itself is an ECX AMP MT RC truck, which is comprised of a plastic body, brushed motor, servo steering mechanism, and NiMH battery to power it all.
2. Shown is the Raspberry Pi Camera Module **(1)** sitting in its 3D-printed mount, connected to the Raspberry Pi computer **(2)** below (in which the steering Microcontroller Unit (MCU) **(3)** is plugged).
3. The Steering MCU is at the top, and HobbyWing QuicRun Electronic Speed Controller (ESC) **(4)** is below.
4. The top component is an FS-GR3E RC Receiver **(8)**, with a Pololu RC Servo Multiplexer **(7)** below it. The Servo Multiplexer allows the user to switch between manual control and autonomous mode. At the bottom is the velocity MCU **(6)**.
5. This is the Hall effect sensor **(5)**, which is soldered directly to the velocity MCU. One of the eight Neodymium magnets **(9)** can be seen attached to the inside of the wheel.


1
8
2
7
3
6
4
5


1
2
3


3
4


8
7
6

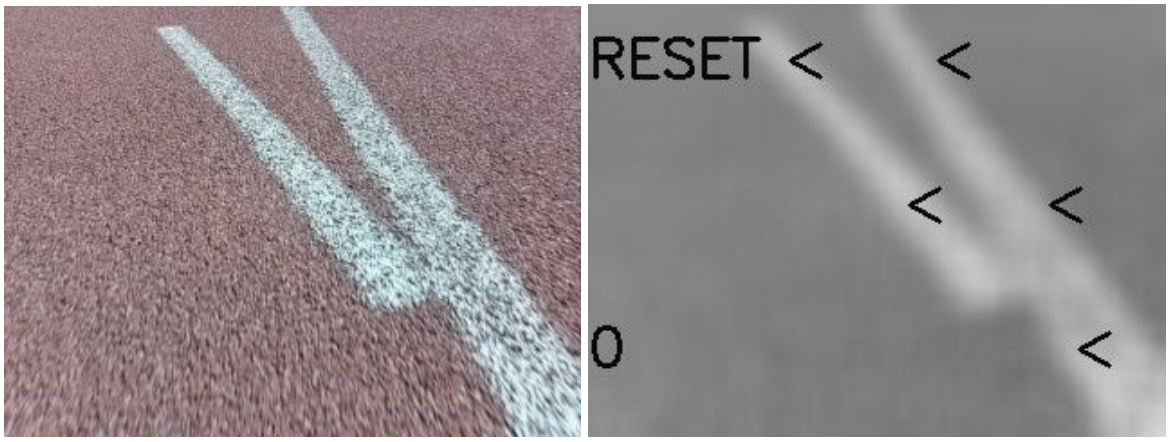

5
9

Vision Algorithms:

The visual system on the Raspberry Pi first determines what the line should look like, and then forms an approximate picture of its position. To do this, we used OpenCV - an open source computer vision library for Python. OpenCV provides numerous functions that allow for simple image processing.

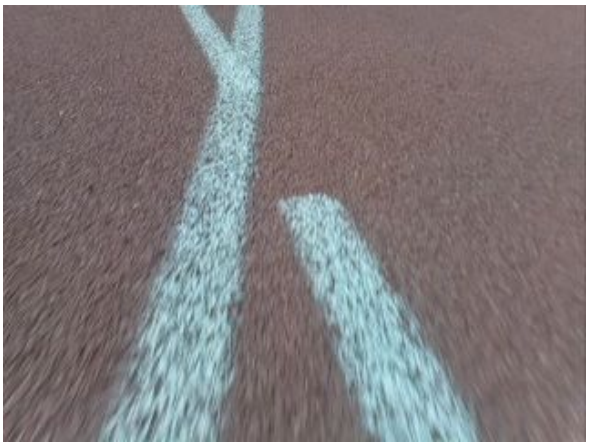
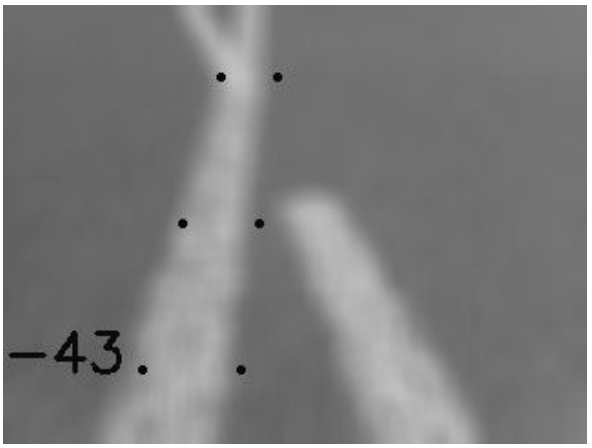
It begins by converting the image to grayscale, and using an averaging function to rid the frame of any noise. Next, the average brightness of the frame is calculated, which is used to compute a threshold for the white line. Since the painted lines on a running track are brighter than the rest of the material, the average brightness can be used to locate the line. The image is scanned at three intersections for the first point above that threshold, and then the next point where it lowers under the threshold again. If no warning signs are detected, those six points are then sent to the linear controller for processing.

Some sections of the track proved to be quite difficult for the visual system to make sense of. For example, just before each straightaway, dashed white lines would intersect the line the vehicle would be following. This easily confused the visual system, which tended to begin to follow the newly introduced set of lines. Here is a picture of that portion of the track, and a visualization of what the visual system is seeing. The arrows point to the places where the controller believes the sides of the line starts and ends, and the “RESET” text indicates that the controller has detected a problem with its line readings.



A problem.

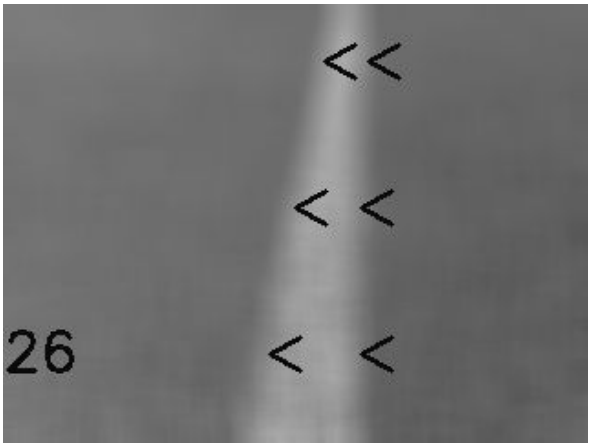
After this issue was made apparent, we sought to find a simple solution. The new controller scans the entire frame for multiple lines, if available, and determines which is more suitable by comparing both readings to the line it was previously following. Here is the result.

A perfect solution.

Control Strategy:

The vehicle required a control algorithm that would keep it driving on the line without rapidly swaying back and forth adjusting its course. This was accomplished using a linear controller with three terms of varying intensity: a differential controller with two variables and a proportional controller. The controllers issue commands depending on the data retrieved from each frame. A positive command initiates a turn to the right, and a negative command for a left turn. The larger the command, the more extreme of a turn. The differential controller consists of a line-straightening term and a position-change term. The strength of line-straightening term is dependent on the angle of the line. A positive slope indicates a positive command, and vice-versa. The position-change term is dependent on the distance the line has moved since the last adjustment. A large distance moved to the left would indicate a positive command. The proportional controller consists only of a line-centering term. The magnitude of the line-centering term is directly proportional to the distance the line is off of center. A line left of center would indicate a positive command. Through trial and error, we found that a strength ratio of roughly 3:2:1 for differential, straightening, and centering worked perfectly; keeping the vehicle following the line with minimal movement.

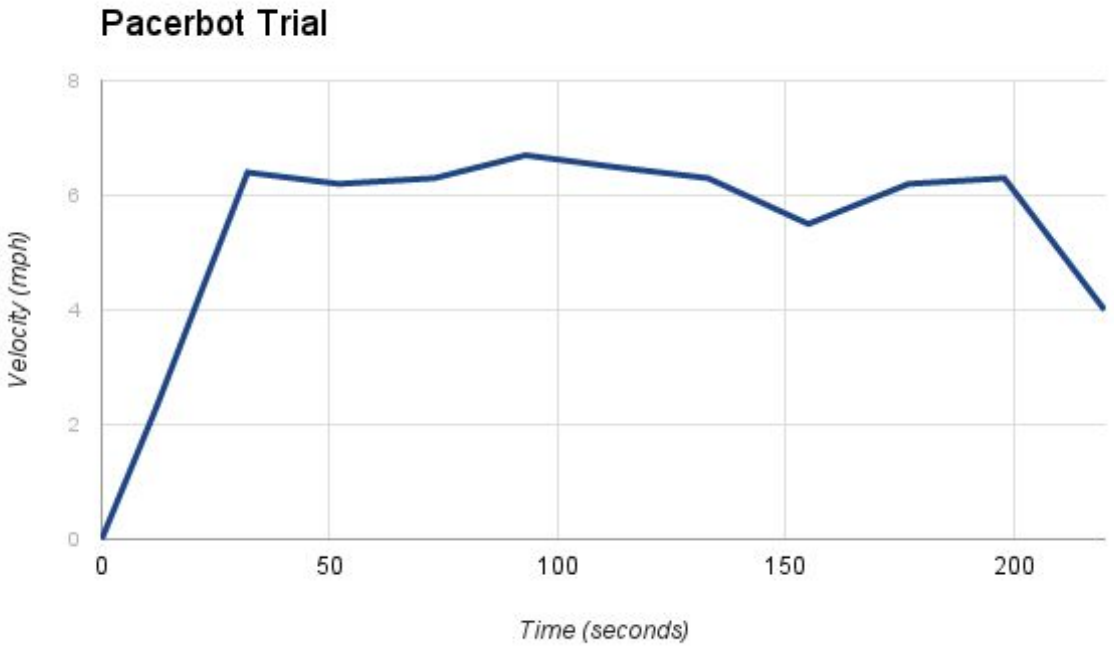



Here is an example of an ideal line detection and command computation. The arrows point to where the controller believes the line to be located. The number “26” indicates the recommended change to the current heading. A positive number indicates a turn to the right and anything below 75 is a relatively small adjustment. The line is slightly right of center and has a positive slope, which shows that the controller has generated a reasonable input.

IV. Current results

The Pacerbot has come a long way since the beginning of the summer. In its current state, it is able to drive around the track autonomously at slow to moderate speeds. The linear controller minimizes oscillation, and the visual algorithms can pick up the line in almost any frame. It sports a 3D printed camera stand, and a fiberglass plate to mount all electronics. Its components include a Raspberry Pi computer, a Camera Module, two MCUs, an ESC, and a Servo Multiplexer.

Here is a plot of the vehicle’s self-regulated speed over 1.5 laps of a track:



The vehicle accelerated up to its goal velocity of 9:30 min/mile (or 6.3 mph) within 30 seconds. Once at speed, fluctuation from goal velocity was minimal. The data collected after the platform had reached goal speed and before it had decelerated shows a maximum velocity of 6.7 mph, a minimum of 5.5 mph, and an average of 6.27 mph.

V. Ongoing results

As the velocity increases, the platform becomes much more sensitive to small control inputs. The highly marked sections of the track are the main culprits in this regard - as even a miniscule interruption to the intended path will have profound consequences. However, as we continue to perfect the visual algorithms and linear controller, these small hiccups will turn into non-issues. Additionally, the velocity controller requires further testing and calibration to improve its accuracy. Even a second over 200m is a significant amount of time in a workout or race of a professional athlete.