Engineering Design Final Report: PiCar System Performance and Control

Kanan Ahmadov
McKelvey School of Engineering
Washington University in Saint Louis
Saint Louis, MO
a.kanan@wustl.edu

Lawrence Lokonobei
McKelvey School of Engineering
Washington University in Saint Louis
Saint Louis, MO
l.l.lokonobei@wustl.edu

***Abstract*—We utilized the PiCar class in python to complete three objectives. The PiCar comes equipped with a RasPi along with various sensors and devices for data collection that are useful for tracking distance, speed, and orientation. For objective 1 we implemented PID speed control in a no-load system. For objective 2, we utilized an ultrasonic-sensor-based system where the distance measured determined the motor speed, and with a blue color tracking algorithm to drive the PiCar and stop within 15 cm of a blue object. Objective 3 required us to utilize PID control to recreate the same result from objective 2, however, it was required that the PiCar maintained a constant speed before coming to a stop. We found that the PID-controlled PiCar from Objective 3 performed much smoother with a more direct path from it's starting point to the blue object when compared to the distance-controlled system from objective 2 which displayed poorer performance with large path inaccuracies.**

INTRODUCTION

PiCar is a robotic platform that uses Raspberry Pi, a small single-board computer, as its primary control unit. The PiCar we used for the Final Module included the sensors: Camera, DC Motor, Servos(s), Analog-to-Digital Converter, and Ultrasonic sensors.

Sensors used on the PiCar

The PiCar has the Ultrasonic sensor for the eyes and the Camera as its mouth. The Ultrasonic Sensor also has a built-in voltage divider to convert the 5 V for the Echo down to 3.3 V input to the Raspberry Pi. The DC motor is installed on the back of the PiCar; the motor has a rear-wheel drive connected to some gears underneath that drive the wheels. The Analog-to-Digital Converter (ADC) is also connected to the back end of the car next to the motor. The ADC was also connected to a photoresistor at channel 0 and an LED light that pointed at the inside of the left back wheel where the black and white disk is installed. Regarding the Servo(s), our PiCar has three servomotors on it:

- Nod servomotor (for tilting the head up and down)
- Swivel servomotor (for twisting the head left and right)
- Steer servomotor (for steering the front wheels left and right)

Power

Other than the sensors, the PiCar's Power component is also essential. The Power LEDs and switch is located on the PWM HAT, which sits on top of the Raspberry Pi. The up/down switch controlls the power from the batteries located under the PiCar, and the four LEDs indicates how much power was left in the batteries.

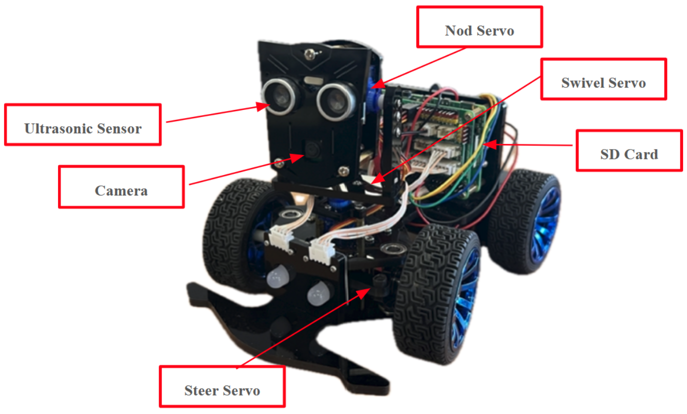


Fig. 1. Important components on the front part of PiCar

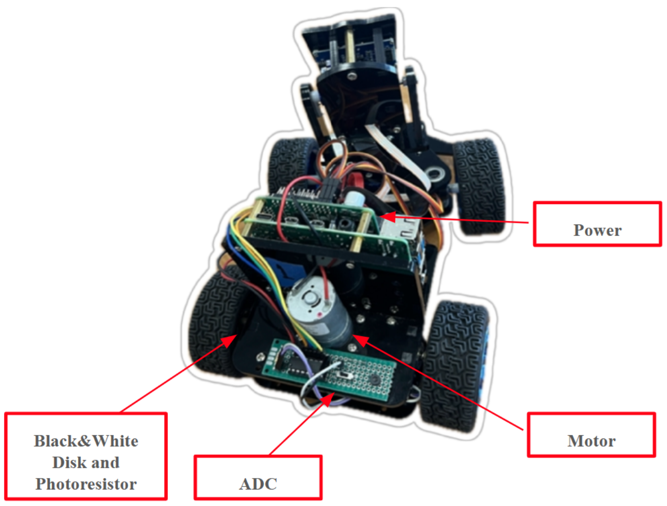


Fig. 2. Important components on the back part of PiCar

METHODS

The overall goal of the final module was to successfully implement the various strategies studied throughout the semester, such as control, movement, and movement with control, on the PiCar. We started this process by implementing the Module 9 tasks, but this time, using a PiCar instead of a mock car.

Part 1

Firstly, we configured our Pi-Car's servos using the **configure_servos.py** program. As we followed along with the prompts, we were able to set the -10, 0, and 10 position for the three servos: the nod, swivel, and steer servo-motors. The -10, 0, and 10 position correlates to the left/down, center, and right/up servo positions. Due to the variation and error in servo performance, this configuration step was necessary at any time that swapped Pi-Cars during our lab process.

Part 2

During this part we tested all of the programs that were given to us: **script1.py**, **script2.py**, **script3.py**, and **script4.py**. Looking at the first program, **script1.py**, it says that we were only going to execute a test script to see if the Pi-Car class is able to set the car is motion. **script2.py** set the Pi-Car in motion to see how it would perform a series of tasks and movements such as setting turning on the DC motors along with changing each of the servo-motor positions. This program also utilizes the ultrasonic sensor and the photo-resistor to produce distance and AD readings. **script3.py** is a program that uses the PiCar class to set the swivel servo position from the left, to the middle, and to the right. **script4.py** is a program takes a picture, writes it to a png file, and then uses the ultrasonic sensor to produce a distance reading.

Part 3

Then, we used our **module9a.py** to control the ultrasonic sensor and the A/D converter at the same time. We ran the program for 5 seconds while printing out the distance and photo-resistor readings every second.

Part 4

Next, we used **module9b.py** to adjust the DC motor pwm based on the distance readings from the ultrasonic sensor. The pwm of the motor was adjusted from a lower percentage to a higher percentage as the distance reading went from low to high. For any distance readings less than 10 cm, the pwm was set to 0%. The pwm increases by 10% for every 5 cm added to the distance reading until the distance reached 50 cm. If the distance read is greater than 50 cm, the pwm is set to 0% to stop the motor.

Part 5

Using **module9c.py** we implemented real-time PID control of the pwm to set the motor speed at 4 rps using the following system timing parameters: 200 Hz sample rate, 4 Hz speed calculation rate, 1 second motor delay, and 0 second A/D sample collection delay. We wrote the time, A/D readings, and rps values to the **data_9c.txt** file. Using the text file, we plotted both the raw rps values and an fft of the rps values. Kp and Ki were set to 1.0 and 0.025, respectively and Kd is kept at zero.

Furthermore, back in Module 8, once our real-time RPSs were verified, we determined a best fit relationship between PWM duty cycle to RPS using Excel. In plotting the data, we made sure to include a (0, 0) point in the data. This linearized best fit line was super useful to simplify the relationship and control strategies, relating the desired RPS to the resulting PWM.

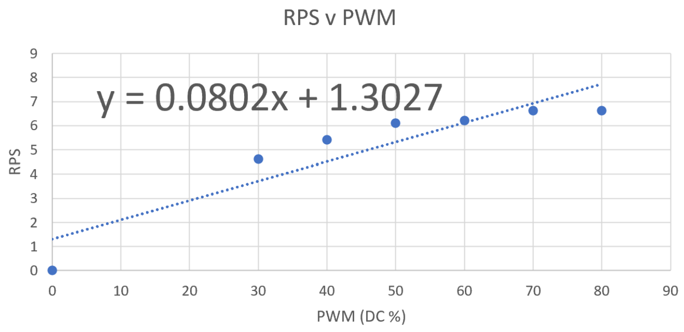


Fig. 3. Relationship between PWM and RPS along with best fit line for data

Since Open Loop Gain = 1 / best fit line slope, then our open loop gain was $\frac{1}{0.0802} = 12.4688$.

Part 6

We implemented the tracking program, **module9d.py**, using the PiCar class to track blue objects by adjusting the swivel servo based on the angle values in real time. The adjustments were weighted using the delta constant. When the blue object centered in the camera, the angle is zero. Objects on the left side of the camera give a negative angle, while objects on the right side of the camera give a positive angle. The coordinate system utilized to calculate camera angles is shown in figure 4.

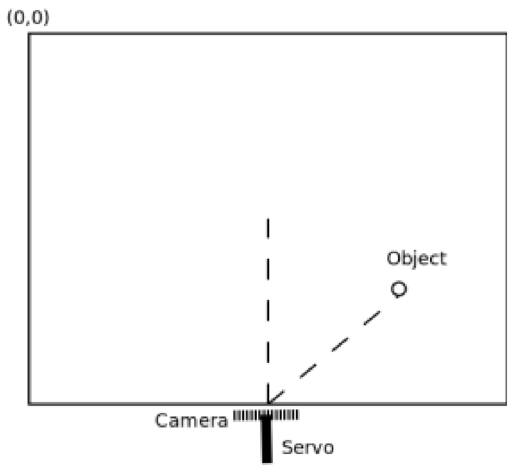


Fig. 4. A diagram that displays the coordinate system that calculates camera angles.

Analysis & Results

Objective 1: Control

The main goal of objective 1 was designing a control system for the car similar to what was used for the motor.

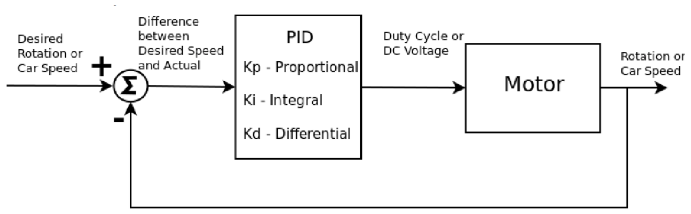
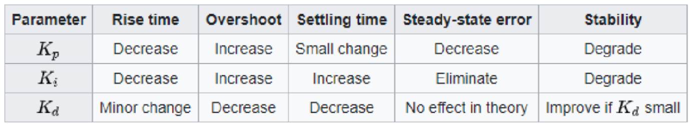


Fig. 5. Schematic of a system with PID Controller

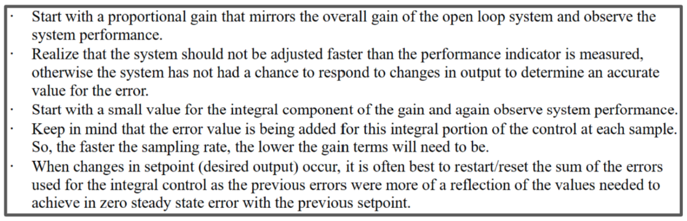
K_p, K_i, and K_d are the three important parameters of the PID controller to tune the controller's behaviour. K_p is the proportional value that provides a quick adjustment when the desired and current output differ. As soon as there is a difference present between the output and the desired output, the system can instantly respond. K_i is the integral control uses the integral (or sum of all errors in the case of a digital system). So even after no error is present in the system, the integral will likely be non-zero which allows the controller to achieve zero-steady state error. K_d is the derivative control uses the derivative of the error (in a digital system, the difference between the most recent measurements). It helps to reduce the impact of quick changes in the other control components, which help reduce the overshoot of the system.



Parameter	Rise time	Overshoot	Settling time	Steady-state error	Stability
K_p	Decrease	Increase	Small change	Decrease	Degrade
K_i	Decrease	Increase	Increase	Eliminate	Degrade
K_d	Minor change	Decrease	Decrease	No effect in theory	Improve if K_d small

Fig. 6. General effects of K_p,K_i, and K_d on a system

Picking values for Kp, Ki, and Kd was tricky. Large values resulted in quick response but also large overshoot and often instability. Therefore, we followed the general observations procedure in Module 8 Notes to choose the appropriate values for K_p and K_i:



- Start with a proportional gain that mirrors the overall gain of the open loop system and observe the system performance.
- Realize that the system should not be adjusted faster than the performance indicator is measured, otherwise the system has not had a chance to respond to changes in output to determine an accurate value for the error.
- Start with a small value for the integral component of the gain and again observe system performance.
- Keep in mind that the error value is being added for this integral portion of the control at each sample. So, the faster the sampling rate, the lower the gain terms will need to be.
- When changes in setpoint (desired output) occur, it is often best to restart/reset the sum of the errors used for the integral control as the previous errors were more of a reflection of the values needed to achieve in zero steady state error with the previous setpoint.

Fig. 7. General Observations procedure to choose the appropriate coefficient values

After the following the procedure in Fig. 7, we choose the $K_p = 1.0$, $K_i = 4.0$, and $K_d = 0.0$. While the derivative coefficient can be valuable in controlling systems with high dynamics and oscillations, it's not always necessary. Since we were working with a simple and stable system with computational constraints, and proportional and integral control provided the desired system response, we decided that adding derivative control might not offer substantial benefits. Therefore, we choose $K_d = 0$. A K_p of 1.0 provided enough overshoot to ensure that our PiCar was able to overcome inertia and get movement right away. The large K_i value of 4.0 surprised us, but multiple trials proved that lower K_i values resulted in small RPS oscillations – keeping the RPS values "stuck" at undesired outputs. Here is the usage we had for Objective 1 that contains all the command line arguments and their values we had:

```
$ python objective1.py --rps 5.0 --timRun 10 --timSamples 0.005 --timMotor 0.25 --delayLoop
0 --delayMotor 1 --Kp 1.0 --Ki 4.0 --Kd 0.0 --debug
```

With the car suspended, we had to determine the response time, overshoot, and steady state error for the system at a target of 5 RPS. The Response Time is the time for the output to reach 90% of its final output value after a change. Overshoot is the amount that the output goes past the desired output value. Finally, Steady State Error is the difference in output and desired output once transient behavior has faded. These three parameters provide a means of determining how well a system behaves.

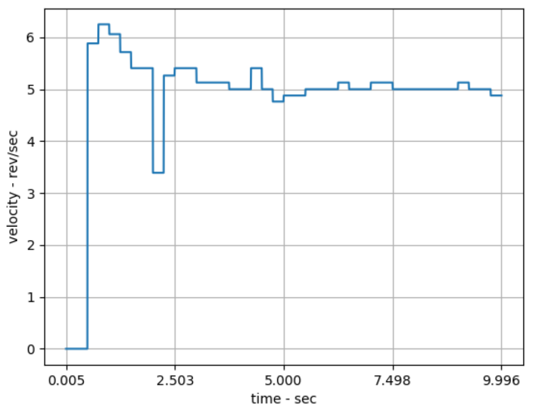


Fig. 8. Velocity-Time plot for PiCar with only Control at RPS = 5

As we can see from Fig. 8, we got a steady velocity-time plot for the PiCar with only Control at the RPS of 5. The strange bottom spike at the t = 2.4 sec could be due to some bad photo-resistor reading.

Regarding the system performance results, we calculated the RPS of 4.929 for our plot and found the Peak RPS to be 6.240. Since calculated the RPS, then the Steady State Error is 5 - 4.929 = 0.071. The 90% of our calculated RPS is 4.436, therefore we found the Response Time to be t = 0.60 sec where the RPS value has a sharp increase to an RPS of 5.940 – past an RPS of 4.436. Finally, the OverShoot was ((6.240 – 5.000)/5.000)*100 = - 24.8%.

To justify the reasoning why our real time calculations are accurate, we modified our plotting program and just examined a steady state portion of that data (power of 2 amount of data) to determine the FFT.

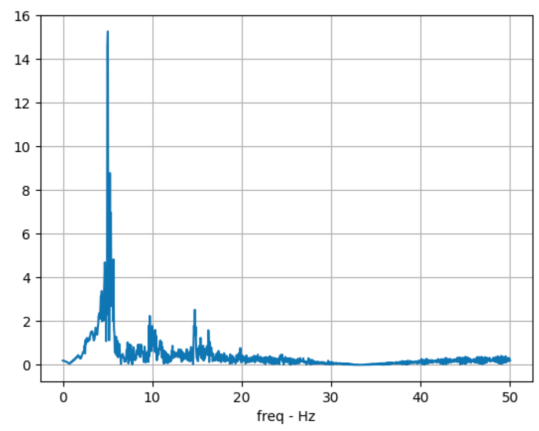


Fig. 9. FFT plot for PiCar with only Control at RPS = 5

Objective 2: Movement

The goal of the second objective was to create a program that would drive the PiCar to a blue object that is positioned at least 10 feet away with the PiCar turned up to 30° degrees away from the blue object in either direction. The blue object we used for objective 2 and for objective 3 is shown in figure 10. There were three targets that we were challenged to meet for this objective: A speed target, a distance target, and a direction target. The speed target was to reach the blue object in less than 10 seconds. The distance target was to start the PiCar at least 10 feet away from the blue object and stopping the PiCar within 15 cm of the blue object. The direction target is to direct the PiCar to the blue object with various starting angles. Using a mixture of the logic and functions from **module9b.py** and **module9d.py**, we created **objective2.py**, a program that tracks the blue object to output the steer servo position and simultaneously measure the distance from the blue object to determine the appropriate pwm for the DC motors.

Fig. 10. The blue object used for objectives 2 and 3.

The image capture delay and delta value were adjusted to create an accurately responsive PiCar that makes proportional changes in steer servo direction towards the blue object. To reduce the occurrence of over-steering/over-correction, we set the steer direction straight ahead for angle measurements below 7.5 degrees in either direction. The program initialization step centers the swivel servo, sets the nod servo slightly above center, and sets the pwm of the motor to 90% for 0.5 seconds to make sure that the PiCar overcomes inertia. Past the initialization steps, the motor pwm decreases to 62% until the PiCar gets within 200 cm. For a distance range of 199-100, 99-50, 49-10, and 9-0 cm the pwm is set to 50%, 45%, 40%, and 0% respectfully to allow for a smooth decrease in pwm as the PiCar gets closer to the blue object.

Objective 3: Movement with Control

The goal of Objective 3 was to combine the processes of Objective 1 and Objective 2 and make the car start from a specific point and travel in a straight line to a large colored object at a specific speed determined by the instructor. The purpose was that the car should not run into the object and should ”stop” as close to the object as possible without hitting it. Our PiCar got within 3/8”.

Regarding the PID coefficients, we again followed the procedure in Fig. 7. We chose the $K_p = 1.5$, $K_i = 4.0$, and $K_d = 0.0$. We choose the $K_d = 0$ again because we were working with a system that had computational constraints, and proportional and integral control provided the desired system response. Therefore, we decided that adding derivative control might not offer substantial benefits. A K_p value of 1.5 was needed because of the load that is experienced by the PiCar increases the inertia that must be overcome. K_p remained constant from objective 1 as it provided a smoother acceleration to our desired 5 RPS value compared to other values for K_p. Higher K_p caused the PiCar to accelerate too fast and lower K_p caused the PiCar to not accelerate at all. Here is the usage we had for Objective 3 that contains all the command line arguments and their values we had:

```
$ python objective3.py --rps 5.0 --timRun 10 --delay 0.5 --delta 0.2 --timSamples 0.005
--timMotor 0.25 --delayLoop 0 --delayMotor 1 --Kp 1.5 --Ki 4.0 --Kd 0.0 --debug
```

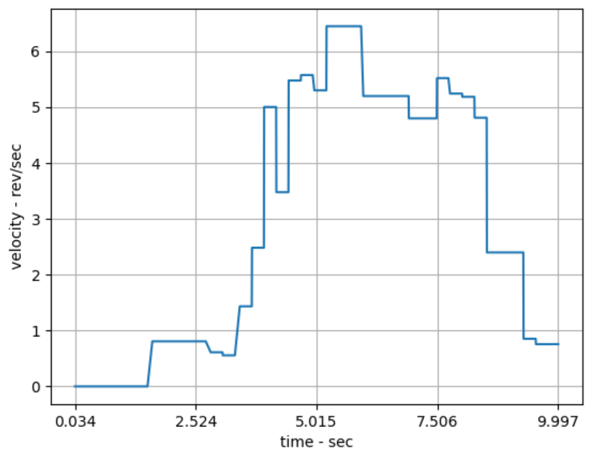


Fig. 11. Velocity-Time plot for PiCar with Movement and Control at RPS = 5

As shown in Fig. 5, the velocity-time plot with Movement and Control is not as smooth as the velocity-time plot with only Control; this is due to the increased system dynamics, non-linearities, and the friction.

Regarding the system performance results, we calculated the RPS of 4.812 for our plot and found the Peak RPS to be 6.450. Since calculated the RPS, then the Steady State Error is 5 - 4.812 = 0.188. The 90% of out calculated RPS is 4.331, therefore we founded Response Time to be t = 3.935 sec at RPS of 5.004. Finally, the OverShoot was ((6.450 – 5.000)/5.000)*100 = 29.0%.

Conclusion

Through the implementation of knowledge and methods from previous modules and the idea of PID control we were able to successfully complete three objectives using the PiCar and PiCar class in python. We first conducted a thorough investigation of the PiCar class to understand how to properly implement our previous programs. Our initialization and configuration steps further ensured that each servo motor was accurately oriented before conducting each trial.

For objective 1, we successfully implemented a PID-control system to control motor speed in a no-load environment. For objective 2, we were tasked to use an ultra-sonic sensor to control motor speed and a camera-angle-tracking algorithm to control servos in an effort to meet speed, distance, and direction targets as the PiCar traversed to a blue object. For objective 3, we combined methods from objective 1 and 2 to use PID control and camera tracking to achieve smooth and accurate pathing to the blue object while at a constant driving speed of 5 RPS. Overall, this final module challenged our abilities to implement previous methods to meet targets and criteria all while utilizing the new PiCar class.

Ultimately through these objectives, we came into a conclusion that K_p balances the stability. If it's too low, the system may be slow and exhibit steady-state error. If it's too high, it can cause instability. K_i ensures the system remains stable while achieving desired accuracy. It should be set to correct steady-state errors without causing oscillation or overshooting. Finally, K_d should dampen oscillations and stabilize the system without introducing excess noise. Future investigations into PID control would serve to further display the complex relationships between the three coefficients: K_p, K_i, and K_d.

References

[1] Ken Vaughan. “MODULE 1: Learn how to use the Pi”.

[2] Ken Vaughan. “MODULE 2: General Purpose Input/Output (I/O) Pins”.

[3] Ken Vaughan. “MODULE 3: Using an Analog to Digital Converter”.

[4] Ken Vaughan. “MODULE 4: More Sensors”.

[5] Ken Vaughan. “MODULE 5: Using the Camera with the PI & Basic Image Processing”.

[6] Ken Vaughan. “MODULE 6: More Imaging and Servos”.

[7] Ken Vaughan. “MODULE 7: Motors, Signal Analysis & Plotting”.

[8] Ken Vaughan. “MODULE 8: Controlling the Motor”.

[9] Ken Vaughan. “MODULE 9: Learn how to use the PiCar Class”.

[10] Ken Vaughan. “MODULE 10: Demo and Final Writeup”.