A Virtual Wall Following Robot With PID Controller

Jon-Paul Boyd
School of Computer Science and Informatics
De Montfort University
United Kingdom

***Abstract*—** This report presents a Proportional, Integral and Derivative (PID) controller tasked with setting motor control velocities to solve a virtual robot wall-following problem. Implemented as a plugin to an existing control layer written in Python **[1]**, it controls the Pioneer P3DX robot **[2]** modelled in the V-REP **[3]** test environment via API. A final evaluation is made, comparing PID variants. Using a fixed baseline speed, the PI controller performed best, reducing navigation error by 12% over best efforts from previous research **[1]**. The PD controller delivers optimum performance when dynamically adjusting baseline speed with proportional error, reducing navigation error by 55.8% over previous best. Recommendations include refined sensor noise filtering to improve integral and derivative control.

Index – Mobile Robots, PID, Wall Follower, Control Systems

I. Introduction

This paper investigates a PID controller setting motor velocities of a differential-wheeled robot, building on earlier work [1] by testing an alternative control mechanism. Benchmarking performance with the same testing scenario, where robot motion at random bearing locates a wall which is then traversed, ensures valid, empirical comparison.

This paper is organized as follows. Section II provides a general PID overview before details of a specific baseline controller design in Section III. Section IV covers controller variants extending the baseline. Section V closes with final evaluation and thoughts.

II. PID Controller Design

The Hellenic Greeks were known to control liquid levels and flow with feedback devices, which in the seventeenth century were applied to the control of furnace temperature, and in the eighteenth to positioning windmills and their grinding stones [4]. By the 1940's full field adjustable field controllers were available [4]. Today a PID controller is "*the control strategy most frequently used in the industry*" [5]. Here the PID (Fig. 1) is a reactive system maintaining robot (plant) constant distance to the wall, referred to as the setpoint (SP). For input the system takes an ultrasonic sensor distance value, known as the process variable (PV). Within each control loop cycle the controller sets the output of both left and right motor velocities, working to reduce the difference between PV and SP, known as error $e(t)$, to 0.

The PID has 3 control elements, each assigned a gain constant value (K_P, K_I, K_D) that multiplies the error and combined generates the velocity values. The p*roportional* element measures how far PV is from SP by taking SP – PV $e(t)$ and multiplying by K_P to scale proportional output directly to error magnitude. *Integral* accumulates errors over time to show a history of PV from SP and can be summed then multiplied by K_I to offset K_P. *Derivative* as the difference between current and previous error multiplied by K_D indicates the rate of error change. All 3 elements are combined to generate a final signal output.

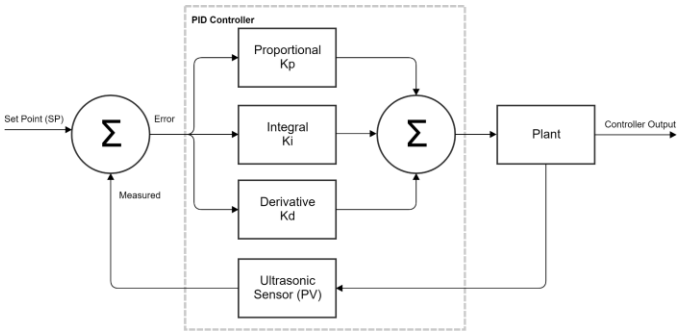

Fig.1 PID Controller Design

III. Baseline P Controller

A PID controller may be proportional-only or include integral and derivative elements (e.g. P, PI, PD, PID). A P-only controller, with integral and derivative gains set to 0, will establish a baseline.

Recalling P is SP – PV $e(t)$ * KP and working to reduce to 0, a robot with little motion is a symptom of controller output set by P alone. Therefore, a baseline optimal motor velocity (*baseline_v*) is set. Keeping the same scenario constraints as [1] with SP set at 0.23m, first tests setting K_P, with *baseline_v* +/- P, showed the robot unable to avoid wall collisions, especially where navigation required ~90° turns. Simply put, there was insufficient differential motor input and the robot was unable to turn fast enough.

Establishing optimal *baseline_v* and K_P followed a heuristic approach using PV, $e(t)$ and controller output as tuning input. The goal was setting lowest possible K_P to prevent oscillations and high-frequency error-correction while providing sufficient differential steering ability to safely negotiate turns at highest possible *baseline_v*. A range of tested K_P values are presented in Table I. K_P 6.29 and 7.29 perform similarly, whereas correlated measures *distance* and *navigation error* (accumulated SP – PV error) for other gains are detrimentally impacted. This suggests 7.29 is the sweet spot.

TABLE I
P-CONTROLLER GAIN TEST RESULTS

K_P	Time Taken (s)	Average Speed (m/s)	Distance (m)	Navigation Error (cm)
5.29	156.3	0.57	89.19	434.17
6.29	153.43	0.58	88.93	367.77
7.29	153.22	0.58	88.71	364.4
8.29	154.53	0.58	89.09	450.14
9.29	157.15	0.57	89.7	539.57

Supporting these results are graphed gain errors during navigation of an inside 90° corner (Fig. 2), selected as an experimental test given it's the most common hazard. Negative errors represent PV > SP (robot above min. distance). All gains settle to SP eventually but 7.29 is quickest. The lowest K_P, 5.29, is too damped hence away from SP for longer and thus cause of higher navigation error. The robot was observed to wall collide during the turn in which a small oscillation occurs, seen on the green line as a z-curve during descent. 6.29 shows a similar pattern but a gain increase gives a faster response. Increasing gain further, 7.29 responds faster still, however with a small SP overshoot before moving back to target faster than all others.

With higher gains comes instability. 8.29 exhibits a first strong SP overshoot before settling. 9.29 is highly unstable, with high frequency oscillations, the robot seen to swing around SP. Such behaviour accounts for the highest navigation error and slowest time.

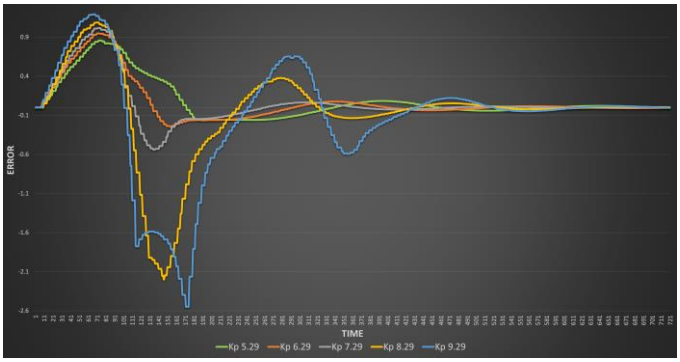

Fig.2 P-Controller Gain Error 90° corner

The final distance between SP and PV as error settles, known as steady-state error, is comparable for K_P 5.29. 6.29 and 9.29 at ~-0.003, while 7.29 and 8.29 show ~0.002. A baseline configuration with *baseline_v* of 0.39m/s and K_P*=7.29* is now established (Fig. 3). An error below 0 is the robot is too close to the wall and the left motor is biased to steer it away. Conversely, an error greater than 0 biases the right motor to steer it closer.

```
'pid': {'kp': 7.29, 'ki': 0.0, 'kd': 0.0}
..
distance = self.prox_dist_dir_travel()
error = min_dist - distance

p = abs(self.state['int']['pid']['kp'] * error)
i = 0
d = 0

output = p + i + d

baseline_v = 0.39
if error < 0:
   self.state['int']['motor_l_v'] = baseline_v + output
   self.state['int']['motor_r_v'] = baseline_v - output
else:
   self.state['int']['motor_l_v'] = baseline_v - output
   self.state['int']['motor_r_v'] = baseline_v + output
```

Fig.3 Baseline P-Controller Configuration

IV. Results Extending P Baseline With Variants

Further fine-tuning of baseline first looked to reduce the ~-0.5 error overshoot (Fig. 2) as the robot completes the 90° turn. This was first attempted by the addition of K_D to dampen K_P response to positive error (SP > PV). To ensure robust testing, the error sampling rates were verified to match the main V-REP server thread response rate.

The best result achieved is with K_D *-7* where overshoot is minimised by ~-0.1 compared with baseline, at cost of slowed response at oscillatory peak but settles as quickly (Fig. 4), resulting in comparable speed performance, however navigation error is reduced by 12cm, due to lower frequency negative error. Higher K_D *gains* (e.g. -10) exhibited poorer response and introduced jitter (Fig. 4, red). The derivative element proved hard to tune but does reward with minor improved navigation performance over baseline (Table II).

A. PI Controller

Improving baseline K_P by adding the integral element K_I as an accumulation of errors over time was next tested, and soon found to reach a condition known as "*windup*". This is where the aggregation of error becomes too large and ultimately unusable. Considering the problem domain, the presence of this state is unsurprising. The robot navigates corners which generate large positive error. Despite the majority of time spent following straight walls generating minimal (steady-state) negative error, integral has been wound up too far by turning and therefore its effect felt well beyond the corner, the integral value often maximising out at a clamping limit set to 0.09. To regulate the significant effect of cornering, integral was adapted to give it a short-term memory (STM), where each new error is added to the top of an array limited to the 100 latest samples.

Tuning again was challenging in establishing the best interacting balance of memory "*term*" and integral gain. However, as the graph shows (Fig. 4), additional performance is extracted over the PD controller, settling to steady-state faster with less aggregated error (Table II). This suggests an effective STM implementation that smooths out individual error disturbance.

B. PID Controller

A controller applying all 3 control elements using individually tested optimal gains was assessed, resulting in increased navigation error compared with PD and PI (Table II). Reviewing error output (Fig. 4) to compare with PI, PID exhibits similar behaviour on the initial negative error overshoot but peaks higher, causing slower recovery to SP. In comparison with PD, error correction of first positive overshoot is faster but undamped, causing greater overshoot into higher negative error frequency followed by lag on initial approach to SP before increased recovery to steady-state. A pattern similar to PI suggests integral dominates the derivative control element.

C. Baseline Speed Gain By P

The core technical approach of this study followed a typically "*vanilla*" PID implementation, with controller output added or subtracted to a fixed baseline speed to direct differential steering. Oscillations during turning are much easier to control at lower forward speeds, for example by reducing K_P. However, with the first study capturing time metrics, there was effectively a minimum top speed constraint in place. As has been observed in testing, the challenge is managing SP overshoot during turning. The author proposes a 4th control element which dynamically adapts baseline speed by subtracting the square root of the proportional error.

```
baseline_v = 0.47 - math.sqrt(abs(p))
```

Comparing with fixed baseline speed scenarios, this one simple enhancement shows settling to steady-state in similar times (Fig. 4) yet with a 49% decrease in navigation error (Table II). Interestingly, the PD variant benefits most from a dynamic baseline speed (PI best for fixed), followed by P only. This suggests that with reduced momentum, oscillation during correction is almost eliminated, and the proportional control element alone deals with error satisfactorily. Note a second benefit is improved lap time due to increased baseline speed.

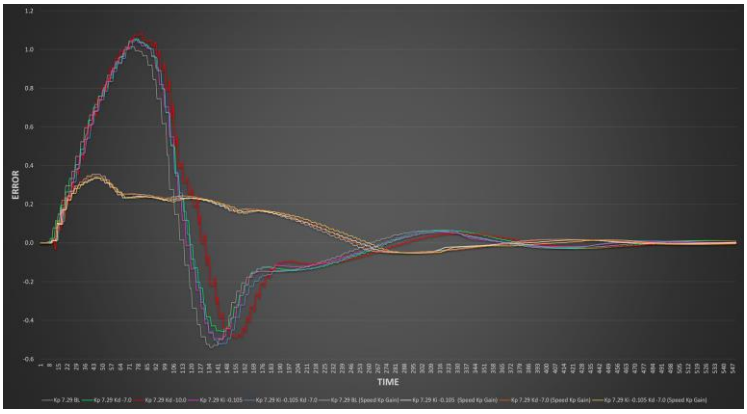

Fig.4 Controller Variant Gain Error 90° corner

TABLE II
CONTROLLER VARIANT GAIN TEST RESULTS

K_P	K_I	K_D	Time Taken (s)	Average Speed (m/s)	Distance (m)	Navigation Error (cm)
7.29	0	0	153.22	0.58	88.71	364.4
7.29	0	-7	153.59	0.58	88.79	352.28
7.29	0	-10.0	153.56	0.58	88.88	380.05
7.29	-0.105	0	153.08	0.58	88.74	349.64
7.29	-0.105	-7	153.41	0.58	88.81	357.8
7.29*	0	0	150.9	0.58	87.08	184.03
7.29*	-0.105	0	149.68	0.58	87.07	226.83
7.29*	0	-7	151.53	0.57	87.09	176.81
7.29*	-0.105	-7	150.21	0.58	87.08	225.4

* Indicates baseline speed adjusted by K_P

The PI controller with fixed baseline speed traverses the wall with a high degree of accuracy (Fig. 5). However, the controller is having to rectify a fixed momentum carrying the robot further into error as illustrated by the trace arcing away from the wall at each corner. In contrast, the PD controller with dynamic baseline speed can scale forward momentum to P control and slow velocity sufficiently so both 90° and 180° corners are navigated with greater precision (Fig. 6).

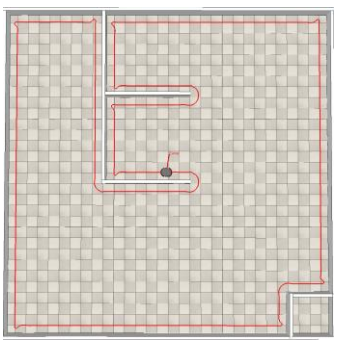

Fig. 5 PI (7.29, -0.105, fixed) trace

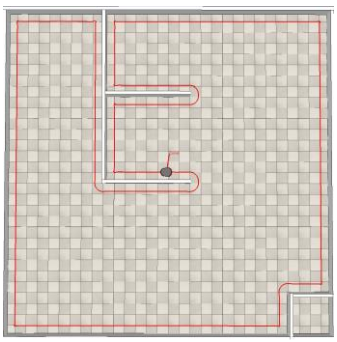

Fig. 6 PD (7.29, -7, dynamic) trace

V. Conclusion

Tuning a PID controller, especially integral and derivative elements, and assessing the configuration best suited to the host plant and problem domain, is a complex and time-consuming exercise that no doubt benefits from prior expertise. Extensive research has looked to automate the tuning process with algorithms that are nature-inspired (evolutionary, swarm) [6] [7] or fuzzy [8]. However, this study demonstrates that even without expertise a controller based on a proven control instrument delivers superior performance over the homebrew control implementation best result [1]. A fixed baseline speed PI control (7.29, -0.105) and dynamic baseline speed PD control (7.29, -7) reduced navigation error by 12% and 55.8% respectively. Typically, there is always a trade-off between speed and accuracy, yet by using a well-established approach both a fast and highly accurate wall traversal is possible as the results clearly showed.

With the addition of a 4th element leveraging P control to manipulate baseline speed, best results were obtained with a PD controller. If absolute minimum navigation error was not the primary objective a P-only controller using dynamic baseline speed control may be an acceptable compromise, given its simplicity and high performance (2nd best dynamic category).

Throughout a turn, the sensor returning minimum can originate from any 1 of 8 sensors. This would suggest the introduction and presence of feedback noise and PV frequency oscillations. To further maximise the potential of integral and derivative these filtering challenges would need to be addressed.

For information the experimentation setup is given in Appendix A. Appendix B presents the PID controller source code as a plugin to the solution developed in [1].

References

[1] J. Boyd, “A Virtual Wall Following Robot,” 2019.

[2] M. R. A. Robotics, “Pioneer Manual,” [Online]. Available: https://www.inf.ufrgs.br/~prestes/Courses/Robotics/manual_pioneer.pdf.

[3] VREP, “V-REP Virtual Robot Experimentation Platform Home Page,” [Online]. Available: http://www.coppeliarobotics.com/.

[4] S. Bennett, “Development of the PID controller,” *IEEE Control Systems Magazine,* vol. 13, no. 6, pp. 58-62, 1993.

[5] J. E. Normey-Rico, I. Alcala, J. Gomez-Ortega and E. F. Camacho, “Mobile robot path tracking using a robust PID controller,” *Control Engineering Practice,* vol. 9, pp. 1209-1214, 2001.

[6] D. Fister, I. Fister and R. Safaric, “Parameter tuning of PID controller with reactive nature-inspired algorithms,” *Robotics and Autonomous Systems,* vol. 84, pp. 64-75, 2016.

[7] K. Tae-Hyoung, M. Ichiro and S. Toshiharu, “Robust PID controller tuning based on the constrained particle swarm optimization,” *Automatica,* vol. 44, p. 1104–1110, 2008.

[8] J. Heikkinen, T. Minav and A. D. Stotckaia, “Self-tuning Parameter Fuzzy PID Controller for Autonomous Differential Drive Mobile Robot,” in *2017 XX IEEE International Conference on Soft Computing and Measurements (SCM)*, St. Petersburg, Russia, 2017.

APPENDIX A – EXPERIMENTATION SETUP

Software Windows 10 Professional (v18.3), V-REP Pro Edu (v3.6.2), PyCharm Professional 2018.3

Hardware Intel i7-5820K, Nvidia GeForce GTX 1080, 16GB RAM

Main Python Modules vrep (2.2.4)

Modified state to manage PID

```
# Pioneer specific internal and external state
self.state = {'int': {'motors': [],
                      'motor_l_v': 0.0,
                      'motor_r_v': 0.0,
                      'motor_all_v': [],
                      'pid': {'kp': 7.29, 'ki': 0.0, 'kd': -7.0},
                      'pid_integral_clamp': 0.09,
                      'pid_prev_error': 0.0,
                      'pid_sum_error_short_term': [],
                      'pid_sum_error': 0,
                      'jpos': {},
                      'prox_history': [],
                      'prox_s_arr': [],
                      'prox_s': None,
                      'prox_is_less_min_dist_f': False,
                      'prox_is_less_min_dist_r': False,
                      'prox_min_dist_f': 0,
                      'prox_min_dist_r': 0,
                      'compass': None,
                      'err_corr_count': 0,
                      'cycle_i': 0,
                      'error_history': []},
                      'ext': {'abs_pos_s': None,
                              'abs_pos_n': None,
                              'abs_pos_all': []}}
```

Wall Follower PID Controller Plugin

```
def wall_follow_pid(self, min_dist, max_dist, gain, motor_index):
    """
    This wall follow algorithm uses a PID controller approach
    """
    # Get PV
    distance = self.prox_dist_dir_travel()

    # e(t) is SP - PV
    error = min_dist - distance

    # Accumulate navigation error
    self.state['int']['err_corr_count'] += abs(error)

    # Integral short-term memory
    self.state['int']['pid_sum_error_short_term'].insert(0, error)  # Add error to top of list
    if len(self.state['int']['pid_sum_error_short_term']) > 100:
       self.state['int']['pid_sum_error_short_term'].pop()  # Remove last item

    # Traditional integral implementation
    #self.state['int']['pid_sum_error'] += error

    p = self.state['int']['pid']['kp'] * error

    #i = 0
    #i = self.state['int']['pid']['ki'] * self.pid_integral_clamp(self.state['int']['pid_sum_error'])
    i = self.state['int']['pid']['ki'] * self.pid_integral_clamp(sum(self.state['int']['pid_sum_error_short_term']))

    #d = 0
    d = self.state['int']['pid']['kd'] * (error - self.state['int']['pid_prev_error'])

    output = p + i + d
    self.state['int']['error_history'].append((self.state['int']['cycle_i'], output))  # Supports telemetry

    #baseline_v = 0.39  # Fixed baseline speed
    baseline_v = 0.47 - math.sqrt(abs(p))  # Dynamic baseline speed with P control gain

    if error < 0:
       self.state['int']['motor_l_v'] = baseline_v + abs(output)
       self.state['int']['motor_r_v'] = baseline_v - abs(output)
    else:
       self.state['int']['motor_l_v'] = baseline_v - abs(output)
       self.state['int']['motor_r_v'] = baseline_v + abs(output)

    self.state['int']['pid_prev_error'] = error  # Log error supporting derivative



def pid_integral_clamp(self, integral):
    if integral > self.state['int']['pid_integral_clamp']:
       return self.state['int']['pid_integral_clamp']
    elif integral < -self.state['int']['pid_integral_clamp']:
       return -self.state['int']['pid_integral_clamp']
    else:
       return integral
```