Pathfinding Robot

James Bao
*Dept. of Electrical, Computer,
and Software Engineering*
The University of Auckland
Auckland, New Zealand
jbao577@aucklanduni.ac.nz

Benson Cho
*Dept. of Electrical, Computer,
and Software Engineering*
The University of Auckland
Auckland, New Zealand
bcho892@aucklanduni.ac.nz

Nicholas Wolf
*Dept. of Electrical, Computer,
and Software Engineering*
The University of Auckland
Auckland, New Zealand
nwol626@aucklanduni.ac.nz

Viktor Neshikj
*Dept. of Electrical, Computer,
and Software Engineering*
The University of Auckland
Auckland, New Zealand
vnes637@aucklanduni.ac.nz

Abstract—Our Pathfinding Robot is an intelligent, two-wheeled robot capable of navigating the shortest-path between points-of-interest within a maze, projected as a line from a ceiling mounted projector. This robot has been developed in COMPSYS 301 during Semester Two of 2023 by Team 1, and has involved the exploration of both hardware and firmware/software aspects. This development has included time- and frequency-domain analysis of phototransistor and photodiode sensors, analogue circuit design, sensor constellation & printed circuit board design, pathfinding algorithm implementation, and firmware development on a PSoC 5LP microcontroller. This engineering effort has produced a high-quality, robust robot that comfortably exceeds the goals that we set out to achieve. The engineering design considerations and decisions made towards this outcome are outlined in this following report.

Index Terms—psoc 5, pathfinding, photodiode, firmware, pcb

I. Hardware Design

A. Frequency Domain Analysis

B. Analogue Circuitry Design

The goal whilst designing our sensor circuitry was to produce a robust output signal that was immune to ambient lighting conditions. We concluded in our frequency-domain analyses that this ambient noise primarily consisted of a DC ($0\,\mathrm{Hz}$) component, which could be blocked by a high-pass/coupling capacitor. Since we decided to use a photodiode, we opted for a transimpedance amplifier instead of a simple resistor due to its lower input/output impedance even when using a feedback resistor in the $\mathrm{M\Omega}$ range. The maximum photocurrent for the provided `SD 019-101-411` photodiode is $2.3\,\mathrm{\mu A}$, which is very small—hence requiring a large gain. To get a signal with an acceptable range, where $V_{\mathrm{out}} = I \times R_{\mathrm{feedback}}$, we chose a $10\,\mathrm{M\Omega}$ resistor to produce an output in the order of $2.3\,\mathrm{V}$. We also included a feedback capacitor to the transimpedance amplifier that added a low-pass effect, where the cut-off frequency is given by $f_{\mathrm{c}} = 1/(2\pi R_{\mathrm{feedback}}C)$. Knowing that the relevant projector frequency was $120\,\mathrm{Hz}$, we decided that any $f_{\mathrm{c}} \geq 300\,\mathrm{Hz} \sim 400\,\mathrm{Hz}$ was acceptable, but a higher cut-off would likely be more desirable to minimise any phase-shift effects despite the effect being small. Consequently, we selected a $40\,\mathrm{pF}$ capacitor to produce an $f_{\mathrm{c}} = 400\,\mathrm{Hz}$ as it was the smallest capacitor available, and also produced an acceptable f_{c}. For the cascaded high-pass stage we decided to choose an f_{c} of $110\,\mathrm{Hz}$, assuming the effects of phase shift to be negligible—there was no visible change from experimentation—as this blocks any DC component whilst retaining frequencies of interest. This also allowed us to select a $42\,\mathrm{k\Omega}$ resistor and a $47\,\mathrm{\mu F}$ capacitor which were readily available in the ECSE Component Store.

The next stage comprised a comparator created with an OpAmp. No hysteresis was included as we observed our time-domain signals to be very clean with sharp edges. One disadvantage of this OpAmp topology is that an OpAmp contains an internal capacitor whilst a dedicated comparator IC does not, resulting in a lower slew rate due to the RC time constant. This was acceptable however as we did not require such precision, and it was more practical to simply use the second OpAmp in the transimpedance amplifier IC. We decided to use a DAC on the PSoC to provide the comparator reference voltage, as this offered flexibility in case we needed to vary this quantity once our PCB was assembled.

We selected the rail-to-rail `TLC082` OpAmp as it drove closer to the rails than an `LM324`, which was essential towards reliably satisfying the high-voltage threshold of the PSoC. We decided against the BJT-based `LT6221`, as the FET-based `TLC082` permitted a higher input impedance for each stage due to its lower input bias current ($150\,\mathrm{nA}$ for the `LT6221` versus $50\,\mathrm{pA}$ for the `TLC082`). In the end, this was prioritised over the slightly higher slew rate of the `LT6221` ($20\,\mathrm{V\,\mu s^{-1}}$) versus $16\,\mathrm{V\,\mu s^{-1}}$) due to the negligible effects of phase shift in both cases.

C. Printed Circuit Board Design

II. Pathfinding Algorithms

A. Finding the Shortest Path

The project requirements state that we must visit five points in *sequential* order, such that the selected algorithm must be capable of computing the shortest path between any two arbitrary points. Consequently, we decided to explore two different algorithms:

1) Dijkstra's Shortest Path (Dijkstra's), and
2) Breadth First Search (BFS).

Dijkstra's is a greedy algorithm that adds only the least-cost node to its path, whilst BFS is a naïve algorithm—ie the traversal does not have any 'intelligence'. We knew that in this project each point carried no weight, which guaranteed that the first time that BFS saw each node would be the shortest—or

shortest equal—path. It is for this reason that Depth First Search (DFS) was not considered, as it could be proven that DFS is not guaranteed to take the shortest path to an end node, even in an unweighted graph.

We were additionally required to create a tool to demonstrate our implemented algorithms, which we created in `C++` due to its built-in data structures, whilst retaining a syntax that closely resembled `C` for easier porting into our firmware. Additionally, we used `MatPlotLib` for visualisation. We verified the functional correctness of both Dijkstra's and BFS with our visualisation tool, but decided to proceed with BFS as it required only a simple queue, whilst Dijkstra's required a priority queue. As these data structures are not provided in `C`'s standard library, we elected to reduce the risk of introducing a point of failure by selecting the one with an easier implementation—BFS.

B. Conversion to Instructions

Finding the shortest path alone would have necessitated a considerable amount of processing and state management in firmware for the robot to remember its position in the maze. In the interest of simplifying this business logic, we exploit the fact that the pathfinding algorithm only needs to be called at initialisation by implementing a supporting algorithm that generates our required actions, an example of which is shown in Figure 1.

This provides the instructions by which our robot abides at intersections, and specifies the distance for which to travel if the required food location is not found at an intersection. We verified the correctness of this supplementary algorithm by tracing out the correct path the robot should take and comparing against the generated output.

III. FIRMWARE IMPLEMENTATION

In order to facilitate collaboration on different aspects of our firmware, we decided to produce a series of modules, where each module exposed its own API. This modular architectural approach enabled ease-of-understanding during integration, and prevented name collisions with our use of namespacing.

A. Movement Module

The mission of the movement module was to streamline interactions with the most vital component of our robot—the motors. This was achieved by abstracting-away the motor synchronisation, precise timing, and fine-tuning.

Just as no two motors are mechanically identical, we visually observed a substantial difference in the output of each motor during the early stages of the project. This inhibited the ability of our robot to perform the basic action of moving in a straight line, leading to the first iteration of our control loop. This was a looping routine that measured the error in observed angular position with respect to the expected angular position, determined using the pulses from quadrature decoders. The expectation was that we could negate this error by quantifying the necessary power increase to each motor in order to run the motors at a constant speed, but we encountered problems with instability.

The second motor control concept was to benchmark the performance of the motors at varying speeds and battery voltages to formulate a mathematical model of each motor. After fitting a linear equation to the rotational speed response outside of the dead-band, we achieved a fairly accurate target-to-output speed when the robot was raised off the ground, but we could not easily model the effect of variance in environment. Consequently, we revisited the closed loop control system a third time.

In further detail, the control system reads a representation of the change in angular position through quadrature encoders, the count of which we read in firmware using a timed interrupt—where we use the count to determine whether it is more or less than we expected in that time period. This required an engineering decision to be made regarding the rate at which we sampled the counts—if we sampled too quickly we may only read one or nil decoder counts, reducing the possible motor speeds to a binary fully-on or fully-off. In contrast, sampling too infrequently would make the control loop too slow to respond, and likely with large corrections due to a larger accumulated error. Having learned from our first iteration, we chose a sampling frequency of $40\,\mathrm{Hz}$, producing a control loop with an acceptable range for precise comparisons whilst also running at high speed—this worked. We verified the functional correctness of this control system by restraining the wheel on one motor, which produced an increase in torque in an attempt to reach the desired reference speed. This was done by changing the PWM duty cycle provided to the motor driver, effectuating an increase/decrease in the voltage provided to the motor.

Following the implementation of our control system, we focussed on the ability for the robot to travel a specified real-world distance. This required a method to correlate the quadrature decoder count to millimetres, which we determined by making measurements of dimensions including the robot's:
1) pivotal circumference,
2) wheel diameter, and
3) wheel circumference.

These measurements are shown in Figure 2.

Knowing the number of decoder pulses in one full rotation of the wheel, we then determined the arc-length between each pulse. We knew that the length of the arc on the outermost point of the wheel (ie the circumference) would be directly translational to the distance travelled, which gave us a relationship between decoder pulses and ground millimetres.

The next point of action was to use the robot's pivotal circumference to determine the number of pulses required to turn a given number of degrees. This was simple to implement, but we debated the use of a polling- versus interrupt-driven implementation. Previously, we elected to use interrupts for the `Movement_move_mm()` function as we wanted the control loop operate alongside simultaneously. In this case however, we decided that we only cared about completing the turn as precisely as possible whilst executing a turn, and did

not want any other component of the system to disrupt the turn. Consequently, we employed a blocking, polling-driven implementation.

Finally, whilst the majority of the movement module operates entirely on the control loop and the turning functions, there is another key component of fluid movement—skew correction. This is necessary to enable the robot to follow the projected line, and consequently requires the integration of our photodiode light sensors. When at least one of the skew detection sensors departs the line, a movement module function is called to correct this deviation. We employ two methods to achieve this skew correction:

1) **Fast Response.** This is the simplest method, where we set a variable that directly increases the PWM duty cycle to one motor. This produces a fast response, and is best used when the robot is already travelling straight.
2) **Smooth Response.** This method is achieved by utilising the control loop to gradually increase the speed to a motor, and allows the robot to straighten out smoothly after completing a turn.

We found that a combination of these two methods produces the best results—by using the smooth control loop method after a turn, until the majority of the skew sensors acquire the line—when we then switch to the fast duty cycle method.

A secondary reason for this fast response method stems from the resolution of the control loop, where the 40 Hz sampling frequency translates to an effectual increase of 40 pulses per second for each unit (ie 1 pulse) increase of the input reference value. This is too great of a change for fine-tuned skew correction, and often caused oscillations with a unacceptably long settling time.

B. Sensor Processing Module

We made use of the digital pin PSoC component to interface with the output of our sensor circuitry, and combined it with an ISR component as shown in Figure 3 to notify us that one of the sensors was under 'white' light, or off the line. We knew that the timing of these rising edges would be very similar for all the sensors as they are all under the same projected light, but component variances and tolerances in our high-pass filters and RC circuitry of the OpAmps may lead to slight variations in these signals.

To account for this, we decided to take more than just a single sample after a rising edge is detected, such that any phase-shifted sensor signals are correctly detected. A sketch detailing this is shown in Figure 4. We concluded that one extra sample after the rising edge was sufficient to account for this, and that excessive sampling introduced unneeded complexity that negatively affected system performance.

Debouncing was also implemented, which required a signal to hold its level either `HIGH` or `LOW` for at least four projector periods to ensure that any false triggers weren't inadvertently counted, corresponding to a a debouncing period of $(1/120\,\text{Hz}) \times 4 = 33.3\,\text{ms}$—not long enough to be visibly slower, and which served to be very reliable. A sketch detailing this debouncing process is shown in Figure 5.

IV. INTEGRATION

REFERENCES

[1]

APPENDIX

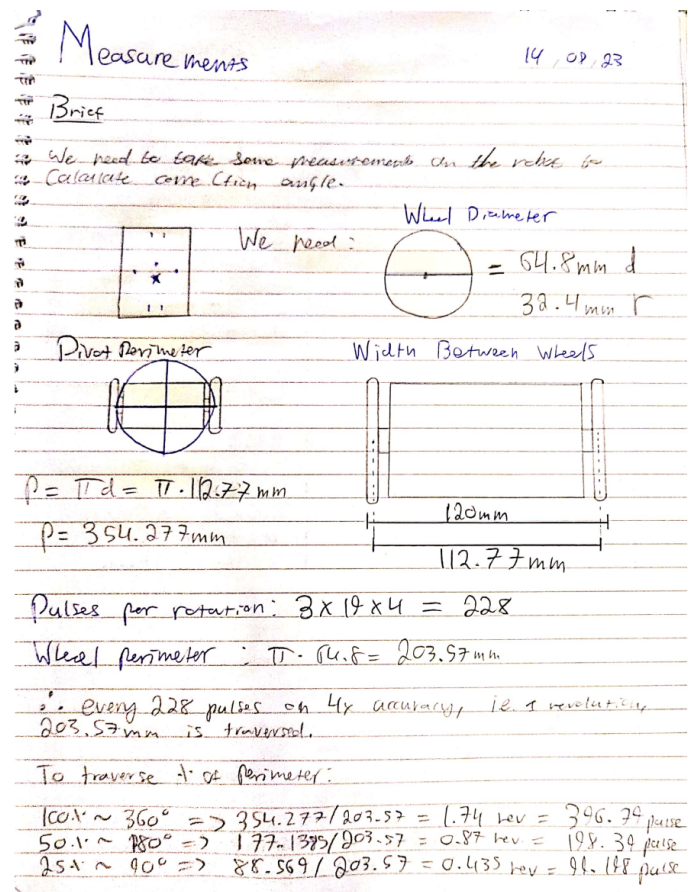Fig. 1. Debugging output of the generated instructions

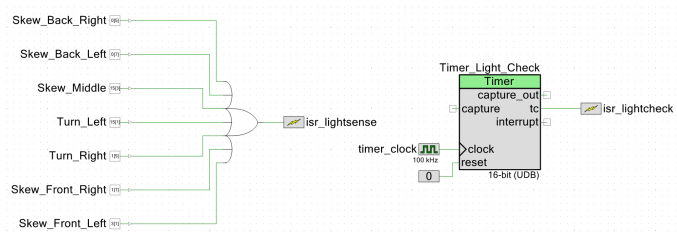Fig. 2. Measurements of the robot's physical dimensions

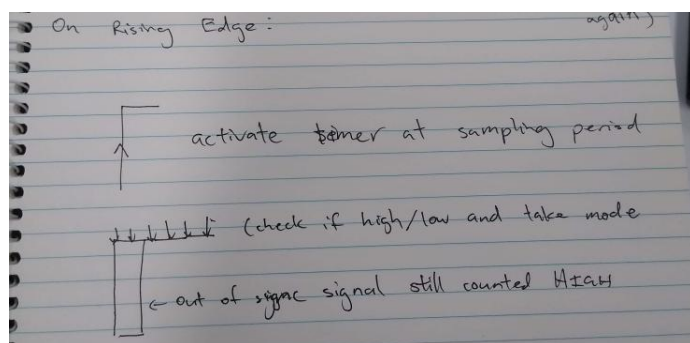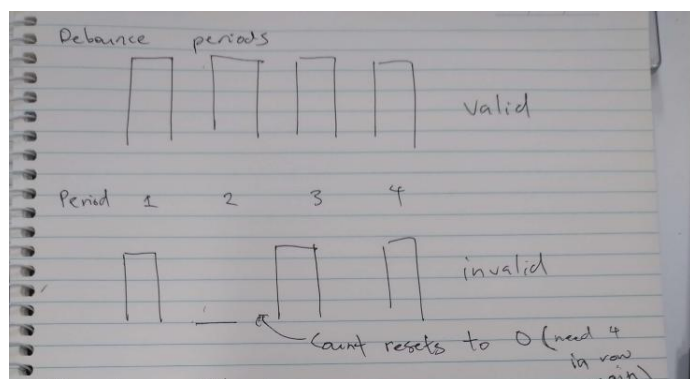Fig. 3. Screenshot of PSoC components used in sensor processing

Fig. 4. Explanation of sensor sampling process

Fig. 5. Explanation of sensor debouncing process