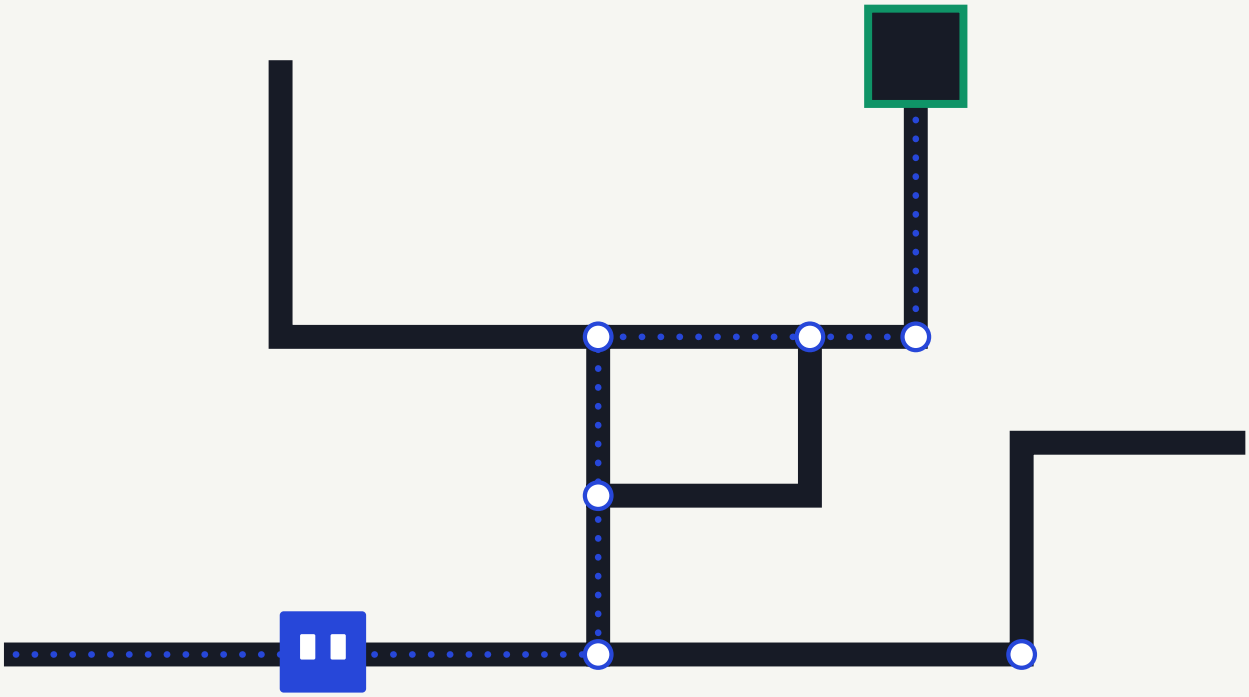


HOW THE MAZE ROBOT THINKS

# Zouzou

From five infrared sensors to a proven shortest path:
the algorithms, the mathematics, the intuition and the code.

| | |
|---|---|
| Robot | ESP32 maze solver, firmware v3 (`firmware/zouzou_v3`) |
| Covers | line sensing · PD steering · junctions · the turn list · odometry · the map · Dijkstra · exploration with a proof of optimality · wall mazes · testing |
| Author | Elmehdi Essamit · Club Electro-Power · EHTP Casablanca |
| Source | `https://github.com/akeanti/zouzou-maze-robot` |

# Contents

**1 The problem** 1

1.1 What the robot is asked to do . . . 1
1.2 The maze as a graph . . . 1
1.3 What the robot can sense . . . 2
1.4 The plan of this document . . . 2

**2 Seeing the line** 3

2.1 One sensor . . . 3
2.2 Calibration: making five sensors agree . . . 3
2.3 From five numbers to one position . . . 3
2.4 Why five . . . 4

**3 Steering: the PD controller** 5

3.1 The control law . . . 5
3.2 Why the robot wobbles more at speed . . . 5
3.3 Why a fixed 250 Hz . . . 6

**4 Reading a junction and turning** 7

4.1 The difficulty . . . 7
4.2 Three phases . . . 7
4.3 Turning . . . 7

**5 The classic way: the turn list** 8

5.1 The left-hand rule . . . 8
5.2 Writing the turns down . . . 8
5.3 Why it gives the shortest path, on a tree . . . 9
5.4 Why it fails with a loop . . . 9
5.5 The counter list is the same method . . . 9

**6 Knowing where you are: odometry** 11

6.1 Counting wheel steps . . . 11
6.2 Distance and angle from two wheels . . . 11
6.3 Why 90-degree mazes make this easy . . . 11
6.4 Where exactly does a road start and end? . . . 12

6.5 Grid snapping . . . . . 12

**7 The map** **13**

7.1 What is stored . . . . . 13

7.2 Arriving somewhere . . . . . 13

7.3 Closing a loop . . . . . 13

7.4 Noticing you are lost . . . . . 14

**8 The shortest path: Dijkstra on states** **16**

8.1 What “shortest” means . . . . . 16

8.2 Why the state must include the heading . . . . . 16

8.3 Dijkstra’s algorithm . . . . . 16

8.4 Cost on a microcontroller . . . . . 17

8.5 Reading the route out . . . . . 18

**9 Exploring, and knowing when to stop** **19**

9.1 The rule . . . . . 19

9.2 When is a frontier worth trying? . . . . . 19

9.3 Why stopping is safe . . . . . 19

9.4 Termination . . . . . 20

9.5 Driving home . . . . . 21

9.6 A complete run . . . . . 22

**10 The speed run** **23**

**11 Walls instead of tape** **24**

**12 How it was tested** **25**

12.1 The solver alone, on a PC . . . . . 25

12.2 What is *not* tested . . . . . 25

**13 Limits, and where to go next** **26**

**A Glossary** **27**

**B Where everything is in the code** **28**

**C Further reading** **29**

**How to read this.** Every chapter has three layers. The **intuition** boxes say the idea in plain words; read only those for a quick tour. The main text gives the precise version, with the mathematics. The **code** listings are copied from the firmware, with their real line numbers, so you can find them in the repository. Proofs can be skipped on a first reading.

# 1 The problem

## 1.1 What the robot is asked to do

A line maze is a drawing in black tape on a white floor. Roads meet only at right angles. The robot starts at the free end of one line; somewhere in the maze is a solid black square, the exit. The robot has never seen the maze. It must

1. find the exit,
2. work out the shortest route from the start to the exit,
3. and then drive that route, as fast as it can.

A wall maze is the same problem with walls instead of tape; chapter 11 shows that nothing above the driving layer changes.

**INTUITION**

Picture yourself in a hedge maze at night with a torch that only lights the ground at your feet. You can see whether the path goes left, ahead or right where you stand, nothing more. You can count your steps. The question this document answers is: what should you remember, and how should you choose, so that you get out and then know the best way for next time?

## 1.2 The maze as a graph

Strip away the tape and a maze is a *graph*: a set of *nodes* (places where the robot has to decide something) joined by *edges* (stretches of road with nothing to decide on).

**Definition 1** (Node)**.** A node is any point where the robot stops following and thinks: a junction where roads branch, a corner, a dead end, the start, or the exit.

**Definition 2** (Road)**.** A road is the stretch of tape between two nodes. It has a length $\ell > 0$ in millimetres and leaves each of its two nodes in one of the four directions north, east, south, west.

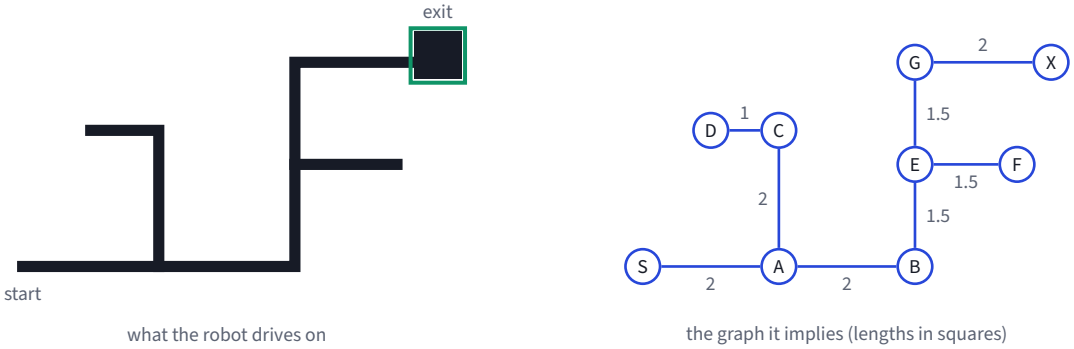



Figure 1: A maze and its graph. Plain stretches of tape vanish into edges; only places with a decision (or an end) become nodes.

In this language the three tasks become: *discover* the graph by driving it, find the *shortest path* in it

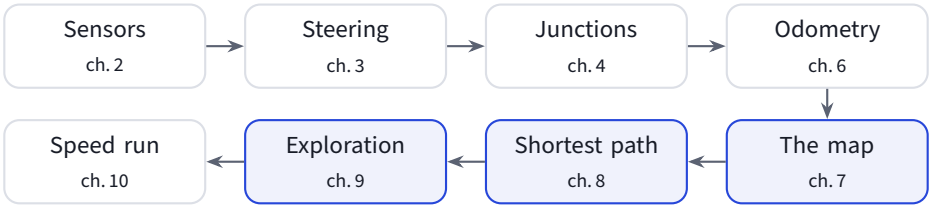
from $S$ to $X$, and *replay* that path. What makes it hard is the first part: the robot only learns about the graph by being somewhere in it, and every metre of exploration costs time.

### 1.3 What the robot can sense

| Sensor | What it says |
|---|---|
| 5 infrared reflectance sensors | how black the floor is under each of five points in a row, 15–17 mm apart |
| 2 wheel encoders | how many 0.15 mm steps each wheel has turned, forwards or backwards |
| 3 ultrasonic sensors | distance to the nearest object ahead, left and right (wall mazes, obstacles) |

There is no camera and no global position. Everything the robot ever knows about the maze it builds from these numbers.

### 1.4 The plan of this document




Chapter 5 is a detour through the classic method (the *turn list*) and why it is not enough; it is the motivation for everything in blue.

# 2 Seeing the line

## 2.1 One sensor

A TCRT5000 is an infrared LED next to a phototransistor. White floor reflects the light back and the transistor conducts; black tape absorbs it. The module turns that into a voltage on its A0 pin, which the ESP32 reads as a number from 0 to 4095. On these modules the voltage is *higher over black*.

The raw numbers are useless as they are: every sensor is a little different, the floor is not perfectly white, the room light changes. So each sensor is *calibrated*.

## 2.2 Calibration: making five sensors agree

During *Wake up* the robot pivots left and right over the line so that every sensor sees both the whitest floor and the blackest tape. For sensor $i$ it records the smallest raw value $w_i$ and the largest $b_i$. From then on a raw reading $r_i$ is mapped to

$$v_i \;=\; \mathrm{clamp}\Big(1000 \cdot \frac{r_i - w_i}{b_i - w_i},\; 0,\; 1000\Big), \tag{1}$$

so 0 means “as white as anything this sensor has seen” and 1000 means “as black as the tape”. A sensor counts as *on the line* when $v_i > 500$. If any sensor saw less than 400 counts between white and black, calibration is refused: that sensor is too high above the floor, or the floor is shiny.

## 2.3 From five numbers to one position

Number the sensors $i = 0, \ldots, 4$ from left to right and place sensor $i$ at coordinate $x_i = 1000\,i$. The line position is the *centre of mass* of the blackness:

$$p \;=\; \frac{\sum_i v_i\, x_i}{\sum_i v_i} \;-\; 2000 \qquad \in [-2000,\, 2000]. \tag{2}$$

$p = 0$ means the line is under the centre sensor, $p < 0$ that it is to the left.

> **INTUITION · why a weighted average**
>
> If the line is exactly between sensors 2 and 3, both see half of it and read about 500. The average of their positions, weighted by 500 each, is halfway between them: $p = 500$. So although the sensors are 16 mm apart, the position is known to about a millimetre. This only works because the tape (19 mm) is a little wider than the spacing: the line is almost always seen by two sensors at once.

Readings under 80 are treated as zero to ignore floor noise. If the total blackness is tiny (the line is under no sensor), equation (2) would divide by almost nothing; instead the function returns ±2000, the edge on the side where the line was last seen. That is exactly the right signal for the steering: “turn hard that way”.

sensors.cpp

```
54 int position() {
55   // Centre of mass of the "blackness" along the array. Sensor i sits at
56   // i * 1000, so two sensors that each see half the line give a position
```

```
  // halfway between them: the resolution is far finer than the spacing.
  long weighted = 0;
  long total = 0;
  for (uint8_t i = 0; i < SENSOR_COUNT; i++) {
    if (values[i] < 80) continue;  // ignore floor noise
    weighted += (long)values[i] * i * 1000L;
    total += values[i];
  }
  if (total < 200) return lastPos >= 0 ? 2000 : -2000;
  lastPos = (int)(weighted / total) - 2000;
  return lastPos;
}
```

## 2.4 Why five

The middle three sensors steer; the outer two are there to notice side roads. With only three, the same sensors would have to do both jobs, and a wobble would look like a junction. Seven would make the row wider than the chassis. Five is also exactly what the ESP32 allows: only the six ADC1 pins can be read while Wi-Fi is on, and the sixth measures the battery.

# 3 Steering: the PD controller

## 3.1 The control law

The robot turns by driving one wheel faster than the other. Every 4 ms it computes a *correction* $u$ from the line position $p$ and adds it to one wheel and subtracts it from the other:

$$u_k = K_p\, p_k + K_d\,(p_k - p_{k-1}), \qquad \text{left} = s + u_k, \quad \text{right} = s - u_k, \tag{3}$$

where $s$ is the cruising speed. $K_p$ is how strongly the robot steers towards the line; $K_d$ is how strongly it resists *changes* in the error.

**INTUITION · P and D**

$K_p$ alone is like a spring pulling the robot back to the line: it overshoots, swings to the other side, and oscillates. The derivative term looks at how fast the error is shrinking; if the robot is already rushing back towards the line, it eases off the steering before it overshoots. It is the shock absorber on the spring.

## 3.2 Why the robot wobbles more at speed

Let $y$ be the sideways offset of the wheel axle from the line, $\theta$ the angle between the robot and the line, $v$ the speed and $L$ the distance from the axle forward to the sensor row. For small angles the robot obeys

$$\dot{y} = v\,\theta, \qquad \dot{\theta} = \omega, \tag{4}$$

where $\omega$ is the turning rate the wheels produce. The sensors sit $L$ ahead of the axle, so what they measure is not $y$ but

$$e = y + L\,\theta. \tag{5}$$

With a proportional controller $\omega = -k\,e$ we get $\ddot{y} = v\dot{\theta} = -kv(y + L\theta) = -kv\,y - kL\,\dot{y}$, that is

$$\ddot{y} + kL\,\dot{y} + kv\,y = 0. \tag{6}$$

This is a damped oscillator with natural frequency and damping ratio

$$\omega_n = \sqrt{kv}, \qquad \zeta = \frac{kL}{2\sqrt{kv}} = \frac{L}{2}\sqrt{\frac{k}{v}}. \tag{7}$$

**KEY IDEA**

Damping grows with $L$ and falls as $1/\sqrt{v}$. Two practical rules follow: mounting the sensors further ahead of the wheels calms the robot; going faster needs more $K_d$. This is also why the firmware slows down slightly while correcting hard (`slowdown = |error|/40`): less speed is more damping exactly when it is needed.

Even with no explicit $K_d$, the geometry gives some damping through $L$: the sensors "look ahead". The D term adds more.

### 3.3 Why a fixed 250 Hz

The derivative term is a difference between two readings. If the loop ran “as fast as possible”, the time between readings would change every time the Wi-Fi server took a few milliseconds, and the effective $K_d$ would jump around. The loop therefore runs on a fixed 4 ms clock. If something held it up for more than two periods, it restarts the clock rather than firing a burst of catch-up ticks. The difference is also lightly filtered (`DERIVATIVE_FILTER = 0.5`), because a raw difference of noisy readings is mostly noise.

robot.cpp, inside followSegment()

```
278     // --- PD steering
279     const int error = sensors::position();
280     derivative += DERIVATIVE_FILTER * ((error - lastError) - derivative);
281     lastError = error;
282     const int correction = (int)(cfg.kp * error + cfg.kd * derivative);
283
284     int cruise = baseSpeed;
285     if (sprintMm > 0 && encoders::distanceMm() > sprintMm - BRAKE_MM) {
286       cruise = min(baseSpeed, (int)cfg.exploreSpeed);
287     }
288
289     // Ease off the throttle while correcting hard. Less speed means more
290     // damping (see the control section of the README) and a smoother ride.
291     int slowdown = abs(error) / 40;
292     if (slowdown > cruise / 3) slowdown = cruise / 3;
293     const int speed = cruise - slowdown;
294
295     motors::drive(speed + correction, speed - correction);
296   }
```

# 4 Reading a junction and turning

## 4.1 The difficulty

A junction is crossed in a fraction of a second, and the robot rarely arrives perfectly square. If it decided from one reading, a T-junction approached slightly crooked would look like a single corner: one outer sensor touches the side road a few millimetres before the other.

## 4.2 Three phases

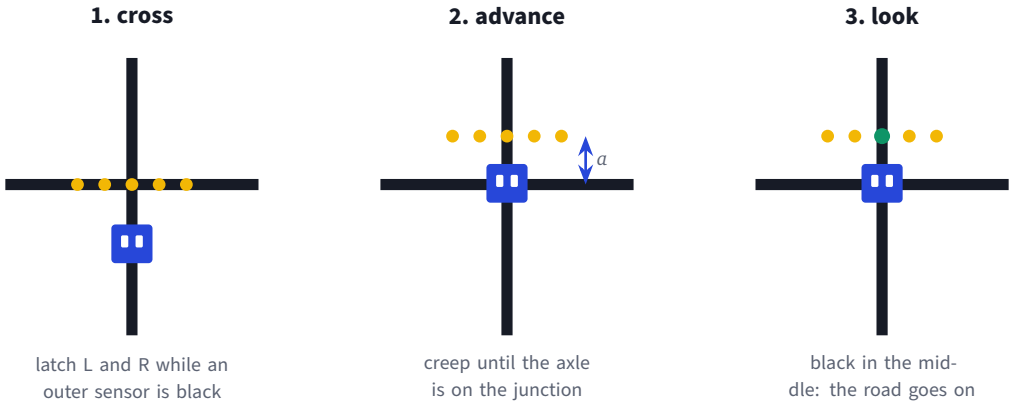



Figure 2: Reading a junction. $a$ is the distance from the sensor row back to the wheel axle.

**Phase 1. Cross.** Creep forward. For as long as an outer sensor is still on black, *latch* (logically OR) the left and right flags. Whichever arrives first, both sides get recorded. If all five sensors are still black after 45 mm, this is not a road but a big black square: the exit.

**Phase 2. Advance.** The sensors are $a$ millimetres ahead of the axle. Keep creeping until the encoders say the axle has advanced $a$ from where the sensors first saw the junction. The robot will now pivot exactly on the junction.

**Phase 3. Look.** The sensors are now past the junction. If any of the three middle ones sees black, the road continues straight.

## 4.3 Turning

To turn the robot spins on the spot until the centre sensor finds the new line. Two traps:

- If there is a road straight ahead, the centre sensor starts *on* it. It must leave that line before a new line counts.
- Turning right at a cross, the robot sweeps past the straight road first. So the first part of every turn is made *blind*: with encoders, the first 45° of a quarter turn (120° of a U-turn) ignore the sensors entirely.

The angle comes from the encoders (chapter 6). As a guard, a turn that has gone past 140° without finding a line stops rather than spinning forever.

# 5 The classic way: the turn list

Before the map, it is worth understanding the method almost every line-maze robot uses, because Zouzou still uses it in two situations (no encoders, and kids mode) and because *why it fails* is the best motivation for what follows.

## 5.1 The left-hand rule

```
at every junction:  if left exists take left
                    else if straight exists go straight
                    else if right exists take right
                    else turn back                          // dead end
```

Keep one hand on the wall and walk: in a maze without loops you visit every corridor at most twice and eventually reach the exit.

## 5.2 Writing the turns down

Each decision is a letter: `L`, `S`, `R`, or `B` for a U-turn. Assign each an angle,

$$S = 0^\circ, \qquad R = 90^\circ, \qquad B = 180^\circ, \qquad L = 270^\circ.$$

Whenever the list ends in $x\,B\,y$ (go in, turn around, come out) the robot is back at the junction it started from, facing a new direction. The single turn that has the same effect is the one with the total angle:

$$x\,B\,y \;\longrightarrow\; \text{turn}\left((\angle x + 180^\circ + \angle y) \bmod 360^\circ\right). \tag{8}$$

| Tried | Sum | mod 360 | Becomes |
|---|---|---|---|
| L B L | 270 + 180 + 270 = 720 | 0 | S |
| L B S | 270 + 180 + 0 = 450 | 90 | R |
| S B L | 0 + 180 + 270 = 450 | 90 | R |
| L B R | 270 + 180 + 90 = 540 | 180 | B |

The result can itself be `B`, which then cancels with its neighbours; so the reduction runs in a loop after every new letter.

**WORKED EXAMPLE · the test maze**

Driving the test maze with the left-hand rule records 17 decisions:

S R B L L L R R B L L L L B S L L

Reducing as it goes, the list collapses to five:

R S R L L.

path.cpp

```
void PathMemory::reduce() {
```

```
65    while (bestLen_ >= 3 && best_[bestLen_ - 2] == 'B') {
66      const int angle = angleOf(best_[bestLen_ - 3]) + 180 + angleOf(best_[bestLen_ - 1]);
67      bestLen_ -= 3;
68      best_[bestLen_++] = turnFor(angle);
69    }
70  }
```

### 5.3 Why it gives the shortest path, on a tree

**Theorem 1.** If the maze has no loops, the reduced list is the unique path from start to exit that never doubles back, and therefore the shortest.

*Proof.* A connected graph without cycles is a *tree*. In a tree there is exactly one simple path between any two nodes. The robot's drive is a walk from start to exit; every detour in it is a sub-walk that leaves a node and returns to it, and in a tree such a sub-walk must turn around at a dead end, so it contains a B. Each reduction removes one "in, back, out" and keeps the walk valid. When no B remains the walk never doubles back, so it is simple, so it is *the* path. Being the only simple path, it is also the shortest. ∎

### 5.4 Why it fails with a loop

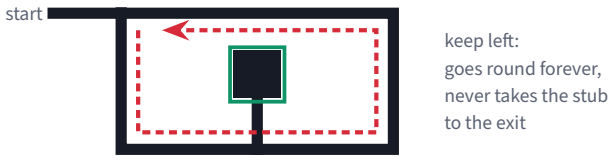



Figure 3: The "loop trap". Every junction on the ring is passed with the exit road on the right, so "left first" never takes it.

Two separate things break:

1. **Getting out.** Wall-following on a loop that is not connected to the outer wall can circle for ever.
2. **The shortest path.** Even if the robot gets out, the theorem's hypothesis is gone. With a loop there are several simple paths from start to exit; the reduction keeps the one the robot happened to drive, not the shortest.

### 5.5 The counter list is the same method

The algorithm first proposed for this project was: "prefer right; at a junction write how many roads are left to try; come back from a dead end, subtract one, take the next; at zero, cross the junction off and go back further". This is **depth-first search**, with the list as its stack. It shares both failures: the list assumes that driving forward always reaches a *new* junction. In a loop the robot reaches an old one from another side, cannot know it, starts a new entry, and the list stops describing the maze. And it keeps the first exit found, not the shortest.

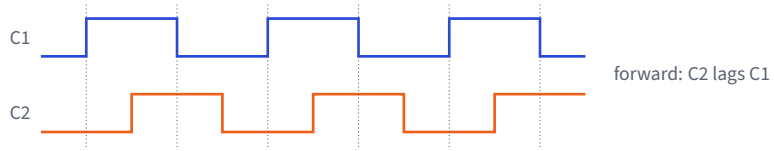
**KEY IDEA**

The bookkeeping is not the problem. The robot knows *which turns it took* but not *where it is*. Any method that only remembers turns has the same blind spot. The fix is to give the robot a position.

# 6 Knowing where you are: odometry

## 6.1 Counting wheel steps

Each N20 motor carries a magnet disc and two Hall sensors, C1 and C2, a quarter period apart. As the shaft turns they produce two square waves:




On every edge of C1 an interrupt looks at C2. If they differ the wheel turned one way, if they are equal the other way. Counting both edges of C1 gives

$$\text{ticks per wheel turn} = \underbrace{7}_{\text{pulses per motor turn}} \times \underbrace{2}_{\text{edges}} \times \underbrace{50}_{\text{gear ratio}} = 700, \tag{9}$$

$$\delta = \frac{\pi \cdot 34\ \text{mm}}{700} \approx 0.153\ \text{mm per tick.} \tag{10}$$

In practice $\delta$ is not computed but *measured*: push the robot 500 mm along a ruler and adjust $\delta$ until it reads 500.

## 6.2 Distance and angle from two wheels

Let $n_L$ and $n_R$ be the tick counts of the two wheels and $b$ the track width (distance between the wheel contact points). The arcs travelled are $s_L = \delta n_L$ and $s_R = \delta n_R$. Then

$$d = \frac{s_L + s_R}{2}, \qquad \Delta\theta = \frac{s_L - s_R}{b}\ \text{(radians, clockwise positive).} \tag{11}$$

The first is how far the middle of the axle went. The second is how much the robot turned: when it pivots on the spot $s_L = -s_R$, so $d = 0$ and all the motion is rotation.

## 6.3 Why 90-degree mazes make this easy

General odometry integrates $(x, y, \theta)$ continuously, and small angle errors grow into large position errors. Zouzou does not need that. Every turn in the maze is a multiple of 90°, and after each turn the line itself re-aligns the robot. So the heading is never estimated; it is *counted*:

$$h \in \{0,1,2,3\} \;\equiv\; \{\mathrm{N},\mathrm{E},\mathrm{S},\mathrm{W}\}, \qquad h \leftarrow (h+q) \bmod 4, \quad q = \begin{cases} 0 & \mathrm{S} \\ 1 & \mathrm{R} \\ 2 & \mathrm{B} \\ 3 & \mathrm{L} \end{cases} \tag{12}$$

and position only changes along straight roads:

$$(x, y) \leftarrow (x, y) + d \cdot (\Delta x_h, \Delta y_h), \qquad (\Delta x, \Delta y)_{h=0..3} = (0,1), (1,0), (0,-1), (-1,0). \tag{13}$$

“North” is simply the direction the robot faced at the start.

**INTUITION**

The robot is like someone walking a city of square blocks, counting steps and always knowing whether they face north, east, south or west, because they only ever turn at street corners. They do not need a compass. Their only error is miscounting steps, and that error does not grow when they turn.

### 6.4 Where exactly does a road start and end?

The distance that goes on the map must be *node to node*, measured at the axle. Three cases need care:

- **After a junction** the axle sits on the node (phase 2 put it there), so the next road is measured from zero.
- **At a dead end** the line vanishes from under the *sensors*, which are $a$ ahead of the axle. The node (the end of the tape) is $a$ further than the axle got. That amount is added to this road and *credited* to the next one: after the U-turn the axle is $a$ short of the node it is leaving.
- **At the exit** the same idea, with the 45 mm the robot crept onto the black square.

In the firmware this is the variable `roadCredit` in `robot.cpp`.

### 6.5 Grid snapping

If the maze is drawn on a grid of squares of side $g$, every measured road length is rounded to the nearest multiple:

$$\ell \leftarrow g \cdot \max\!\left(1,\ \left\lfloor \frac{\ell}{g} + \tfrac{1}{2} \right\rfloor\right). \tag{14}$$

An error of up to $g/2$ per road is then corrected to *zero*, so errors cannot accumulate at all. A wall maze is a grid by construction and always uses this.

# 7 The map

## 7.1 What is stored

maze.h

```
53  struct Node {
54    int16_t x, y;       // millimetres from the start, y grows to the north
55    uint8_t to[4];      // neighbour through each exit, NO_NODE while unknown
56    uint16_t length[4]; // millimetres to that neighbour
57    uint8_t open;       // bit d set: there is a road through exit d
58    bool finish;
59  };
```

Each node stores its position $(x, y)$ in millimetres, and for each of the four directions: whether a road leaves that way (`open`), where it leads (`to`, or `NO_NODE` while not driven) and how long it is. Plus the robot's current node and heading.

**Definition 3** (Frontier)**.** An exit $d$ of node $n$ is a *frontier* if a road leaves $n$ in direction $d$ but has not been driven: `open[d]` is set and `to[d]` is unknown. The map is complete when no frontier is left.

| Memory | bytes |
|---|---|
| one node: $x, y$ (4) + 4 neighbours (4) + 4 lengths (8) + flags (2) | 18 |
| 60 nodes | 1080 |
| search scratch: 240 costs (960) + 240 back-pointers (240) | 1200 |
| total | $\approx$ 2.3 kB of 320 kB |

## 7.2 Arriving somewhere

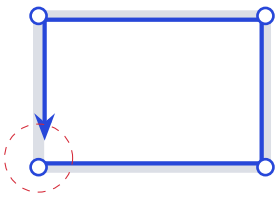
Every time the robot stops at a node it calls `arrive(distance, left, straight, right, finish)`. The work is:

```
function arrive(d, exits):
  if the road just driven is already on the map:          // re-driving a known road
      check that d agrees with the stored length; if not: LOST
      here ← where it leads; return KNOWN
  (x,y) ← position(here) + d · direction(heading)
  n ← a node within matchMm of (x,y), if there is one
  if none:  create node n at (x,y)                          // a new place
  link here ↔ n with length d
  mark n's exits (turned from robot-relative to N/E/S/W)
  here ← n
```

## 7.3 Closing a loop

The line that matters most is "a node within `matchMm` of $(x, y)$". Two points are the same node if

$$\max\left(|x - x_n|,\ |y - y_n|\right) \ \le\ m, \qquad m = 60 \text{ mm (or } g/3 \text{ on a grid).} \tag{15}$$



Back near $(0, 0)$ after going round. A node already sits within $m$: this is the *same* place. The road is linked to it; the loop is closed.

Figure 4: Loop closing. The measured end point never matches exactly; the matching radius absorbs the error.

**KEY IDEA**

This one comparison is the entire fix for loops. Without positions the robot would have created a fifth node and believed the maze went on.

**WATCH OUT · the radius is a trade-off**

Too small, and odometry error makes the robot miss a loop and invent a duplicate node. Too large, and two genuinely different junctions get merged. With $m = 60$ mm, two junctions must be at least about $3m \approx 20$ cm apart, and the accumulated error around a loop must stay under 60 mm. On a grid both problems disappear.

## 7.4 Noticing you are lost

When the robot drives a road the map already knows, the stored length predicts where it should stop. If the measured distance disagrees by more than $2m + \ell/8$, something went wrong (a junction missed, a wheel slipped) and `arrive` returns `Lost`. Exploration stops and says so, rather than carrying on with a map that no longer matches the floor. The same happens if a new road's end lands on a node whose matching side is already linked elsewhere.

maze.cpp

```
57 MazeMap::Arrival MazeMap::arrive(int distanceMm, bool left, bool straight, bool right, bool
       finish) {
58   if (distanceMm < 0) distanceMm = 0;
59   Node& from = nodes_[here_];
60   const uint8_t back = (heading_ + 2) & 3;
61
62   // A road driven before: the map already knows where it ends. Only check
63   // that the distance agrees, to notice a missed or imagined junction.
64   if (from.to[heading_] != NO_NODE) {
65     const int expected = from.length[heading_];
66     if (distance1(distanceMm, expected) > 2 * matchMm + expected / 8) return Arrival::Lost;
67     here_ = from.to[heading_];
68     return Arrival::KnownNode;
69   }
70
71   // A new road. Work out where it ended and look for a node already there.
72   if (gridMm > 0) {
73     const int squares = (distanceMm + gridMm / 2) / gridMm;
74     distanceMm = (squares < 1 ? 1 : squares) * gridMm;
75   }
76   const int x = from.x + DX[heading_] * distanceMm;
```

```
  const int y = from.y + DY[heading_] * distanceMm;
  uint8_t index = findNear(x, y);
  const bool isNew = index == NO_NODE;

  if (isNew) {
    if (count_ >= MAX_NODES) return Arrival::Full;
    index = count_++;
    nodes_[index] = Node{(int16_t)x, (int16_t)y, {NO_NODE, NO_NODE, NO_NODE, NO_NODE}, {0, 0, 0,
     0}, 0, false};
  } else if (nodes_[index].to[back] != NO_NODE) {
    return Arrival::Lost;  // that side of the old node already leads somewhere else
  }

  Node& at = nodes_[index];
  from.open |= 1 << heading_;
  from.to[heading_] = index;
  from.length[heading_] = (uint16_t)distanceMm;
  at.open |= 1 << back;
  at.to[back] = here_;
  at.length[back] = (uint16_t)distanceMm;

  if (finish) {
    at.finish = true;
    finish_ = index;
  } else {
    if (left) at.open |= 1 << ((heading_ + 3) & 3);
    if (straight) at.open |= 1 << heading_;
    if (right) at.open |= 1 << ((heading_ + 1) & 3);
  }

  here_ = index;
  return isNew ? Arrival::NewNode : Arrival::KnownNode;
}
```

# 8 The shortest path: Dijkstra on states

## 8.1 What “shortest” means

The robot wants the route that takes the least *time*, not just the least distance. A 90° turn costs time: stop, pivot, find the line. So a route is charged its length plus a fixed price $\tau$ per quarter turn:

$$\text{cost(route)} \;=\; \sum_{\text{roads}} \ell \;+\; \tau \cdot \#\{\text{quarter turns}\}, \qquad \tau = 80 \text{ mm}. \tag{16}$$

A U-turn counts as two quarter turns. Between two routes of the same length, the one with fewer turns wins.

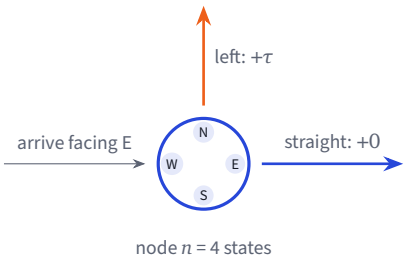
## 8.2 Why the state must include the heading

The price of a turn depends on which way the robot *arrived*. So a node alone is not enough to describe where the robot is; the search runs on *states*

$$(n, h) = \text{“at node } n\text{, facing } h\text{”},$$

four per node. From $(n, h)$ the robot can leave through any exit $d$ of $n$ that leads to a known neighbour $n'$, reaching $(n', d)$ at a cost of

$$c\big((n,h) \to (n',d)\big) = \ell(n,d) + \tau \cdot t\big((d-h) \bmod 4\big), \qquad t(0)=0,\ t(1)=t(3)=1,\ t(2)=2. \tag{17}$$




## 8.3 Dijkstra’s algorithm

```
function search(start state s0):
  for every state s: cost[s] ← ∞;  cost[s0] ← 0
  repeat:
    u ← the unfinished state with the smallest cost      // stop if none
    mark u finished
    for each move u → v with price c:
      if cost[u] + c < cost[v]:  cost[v] ← cost[u] + c;  from[v] ← u
```

The cheapest way to node $n$ is the cheapest of its four states; walking the `from` pointers back gives the route.

**Theorem 2.** When a state is marked finished, its cost is the cheapest possible cost from the start state.

*Proof.* Suppose $u$ is the first state finished with a cost larger than its true cheapest cost $c^*(u)$. Take a cheapest path to $u$ and let $v$ be the first state on it that is not yet finished. The state just before $v$ was finished with its correct cost, and the move to $v$ was then relaxed, so $\text{cost}[v] \le c^*(v)$. All prices are non-negative, so $c^*(v) \le c^*(u)$. Hence $\text{cost}[v] \le c^*(u) < \text{cost}[u]$, and the algorithm would have picked $v$ before $u$. Contradiction. ∎

**INTUITION**

Imagine pouring water in at the start. It spreads along the roads at constant speed and pays a delay at every corner. Dijkstra's algorithm is the order in which the water reaches each place: the first time it arrives anywhere, it came the fastest way.

## 8.4 Cost on a microcontroller

With $N \le 60$ nodes there are $S = 4N \le 240$ states. The firmware uses the simplest version of Dijkstra (scan for the cheapest unfinished state) which costs $O(S^2) \approx 58\,000$ comparisons: well under a millisecond on the ESP32. A priority queue would be faster asymptotically and slower in practice at this size.

maze.cpp

```
// Dijkstra over (node, heading) states. With at most 240 states the simple
// "scan for the cheapest unvisited state" version is fast enough: under a
// millisecond on the ESP32.
void MazeMap::search(uint8_t from, uint8_t heading) const {
  const uint16_t states = (uint16_t)count_ * 4;
  bool done[STATES];
  for (uint16_t s = 0; s < states; s++) {
    cost_[s] = FAR;
    from_[s] = 255;
    done[s] = false;
  }
  cost_[from * 4 + heading] = 0;

  for (;;) {
    uint16_t current = states;
    uint32_t cheapest = FAR;
    for (uint16_t s = 0; s < states; s++) {
      if (!done[s] && cost_[s] < cheapest) {
        cheapest = cost_[s];
        current = s;
      }
    }
    if (current == states) return;
    done[current] = true;

    const Node& n = nodes_[current / 4];
    const uint8_t facing = current % 4;
    for (uint8_t exit = 0; exit < 4; exit++) {
      if (n.to[exit] == NO_NODE) continue;
      const uint8_t quarters = (exit - facing) & 3;
      const uint32_t turning = (quarters == 0 ? 0 : quarters == 2 ? 2 : 1) * (uint32_t)
     turnCostMm;
      const uint32_t total = cheapest + turning + n.length[exit];
```

```
      const uint16_t next = n.to[exit] * 4 + exit;
      if (total < cost_[next]) {
        cost_[next] = total;
        from_[next] = (uint8_t)current;
      }
    }
  }
}
```

## 8.5 Reading the route out

`route()` runs the search from (start, north), walks the back-pointers from the exit, and converts each change of heading into a turn letter. It also stores the length of every road on the way, which the speed run uses (chapter 10).

# 9 Exploring, and knowing when to stop

## 9.1 The rule

At every node the robot calls `explore()`. In words:

1. If there is a frontier *right here* (and it is still worth trying, see below), take it, trying left, straight, right, back in that order.
2. Otherwise, find the *nearest* node, by Dijkstra cost from the robot's current state, that still has a worthwhile frontier, and take the first turn on the cheapest way there.
3. If there is no such node, exploration is over.

**INTUITION**

Step 1 is depth-first search: keep pushing into new ground. Step 2 is what the counter list could never do: when a branch is finished, the robot does not walk back junction by junction; it looks at its map and drives straight to the closest unfinished business, through any shortcut it has discovered, loops included.

## 9.2 When is a frontier worth trying?

Before the exit is found, every frontier is. After, most are not: some parts of the maze are so far away that no route through them could possibly beat the one already known. The question is how to tell, *without driving there*.

Write $B$ for the cost of the best known route from start to exit, and $\mathrm{reach}(n)$ for the cheapest known cost from the start to node $n$ (from a Dijkstra search over the map). For the exit at $(x_X, y_X)$ define the *Manhattan distance*

$$h(n) = |x_n - x_X| + |y_n - y_X|. \tag{18}$$

**Lemma 3** ($h$ never overestimates)**.** Any route from $n$ to the exit, through any roads, known or not, has cost at least $h(n)$.

*Proof.* Every road runs north–south or east–west. To get from $n$ to $X$ the route must cover the horizontal gap $|x_n - x_X|$ with east–west roads and the vertical gap with north–south roads; turns only add cost. ∎

**Definition 4** (Worth trying)**.** A node $n$ is *worth trying* if $\mathrm{reach}(n) + h(n) + m < B$, where $m$ is the matching tolerance from chapter 7.

## 9.3 Why stopping is safe

**Theorem 4** (optimality)**.** Assume distances are exact ($m = 0$) and every node's roads are recorded when the robot first stands on it. When `explore()` returns "nothing left", the best known route is a shortest route in the whole maze, including all the parts never driven.

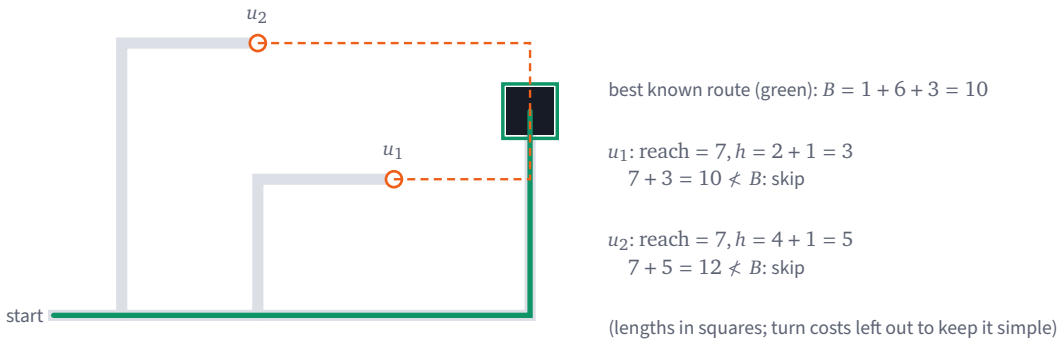



Figure 5: Pruning. The dashed lines are the most optimistic thing that could lie beyond each frontier. If even that is not shorter than $B$, the frontier is never driven.

*Proof.* Let $P^*$ be a truly shortest route from start to exit and suppose $\text{cost}(P^*) < B$. $P^*$ cannot use only known roads, otherwise Dijkstra on the map would have found it and $B \le \text{cost}(P^*)$. So let $u \to v$ be the *first* road on $P^*$ that is not on the map. Everything before it is on the map, so $u$ is a known node and

$$\text{reach}(u) \;\le\; \text{cost}(P^* \text{ up to } u).$$

Because $u$ was visited, its road towards $v$ was recorded as open; it is not driven; so $u$ has a frontier. The rest of $P^*$ goes from $u$ to the exit, so by the lemma it costs at least $h(u)$. Together,

$$\text{reach}(u) + h(u) \;\le\; \text{cost}(P^*) \;<\; B,$$

so $u$ is worth trying and has a frontier, and `explore()` would not have returned "nothing left". Contradiction. ∎

**KEY IDEA**

This is the same argument that makes the A* search algorithm correct: as long as the estimate of "how far is left" never overestimates, anything that cannot beat the best so far even under the most optimistic assumption can be thrown away. Here it decides which roads the robot does not need to drive at all.

With measurement noise the firmware adds the tolerance $m$: it will not drive a frontier to win less than $m$ millimetres, which is below what it can measure anyway. In exchange it never chases phantom improvements created by noise.

## 9.4 Termination

Each step either drives a frontier, which can happen at most once per road, or drives towards a node with a frontier, which takes finitely many steps on a finite map. The maze is finite, so exploration ends. In the worst case (an exit hidden at the far end of everything) the robot drives the whole maze; usually it stops much earlier.

maze.cpp

```
178 char MazeMap::explore(bool thorough) {
179   if (finishKnown() && !thorough) return 0;
180
181   // With the finish known, work out for every node the cheapest total a
```

```
  // route through it could possibly have. See "When to stop" in maze.h.
  bool worth[MAX_NODES];
  if (finishKnown()) {
    search(0, NORTH);
    const uint32_t best = costTo(finish_);
    const Node& goal = nodes_[finish_];
    for (uint8_t i = 0; i < count_; i++) {
      const uint32_t reach = costTo(i);
      const uint32_t remainder = distance1(nodes_[i].x, goal.x) + distance1(nodes_[i].y, goal.y
    );
      worth[i] = reach != FAR && reach + remainder + matchMm < best;
    }
  } else {
    for (uint8_t i = 0; i < count_; i++) worth[i] = true;
  }

  // A frontier right here: take it, in the preferred order.
  if (worth[here_] && hasFrontier(here_)) {
    const uint8_t order[4] = {(uint8_t)(rightFirst ? 1 : 3), 0, (uint8_t)(rightFirst ? 3 : 1),
    2};
    for (const uint8_t quarters : order) {
      const uint8_t exit = (heading_ + quarters) & 3;
      if (isFrontier(here_, exit)) return TURN_NAME[quarters];
    }
  }

  // Otherwise drive to the cheapest node that still has one.
  search(here_, heading_);
  uint8_t target = NO_NODE;
  uint32_t cheapest = FAR;
  for (uint8_t i = 0; i < count_; i++) {
    if (i == here_ || !worth[i] || !hasFrontier(i)) continue;
    const uint32_t reach = costTo(i);
    if (reach < cheapest) {
      cheapest = reach;
      target = i;
    }
  }
  return target == NO_NODE ? 0 : relative(firstExit(target));
}
```

## 9.5 Driving home

Exploration ends wherever the last frontier was. The robot already has the map, so it calls `toward(start)` at each node, which is step 2 of the rule with the start as target, drives back by the cheapest known way, and turns around. Each road on the way home is a known road, so each arrival is also a check: if a distance disagrees, the map is flagged as lost before the speed run relies on it.

### 9.6 A complete run

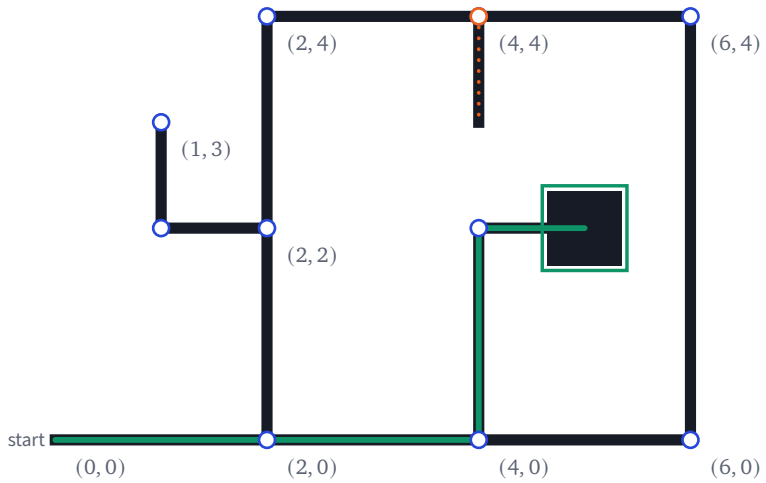



Figure 6: The "Two ways out" maze from the Maze Lab simulator. One unit per grid square, a turn costs 0.32. Green: the proven shortest way. Orange: the one road never driven.

This is the sequence the solver actually produces on that maze (it is the same code as the simulator):

1. **Explore, left first.** From the start to $(2, 0)$, left up to $(2, 2)$, left into the dead end at $(1, 3)$. Back, on up to $(2, 4)$, round the top through $(4, 4)$ and down the right side to $(6, 0)$, then west along the bottom through $(4, 0)$.
2. **The loop closes.** Driving west from $(4, 0)$, the end point lands exactly on $(2, 0)$, already on the map. The robot does not create a new node; it links the road to the old one and now knows the bottom edge is a loop.
3. **Shortcut to the nearest frontier.** Nothing is left to try at $(2, 0)$. The nearest untried road is the one north from $(4, 0)$, two squares back. The robot turns round, drives there, takes it, turns right and reaches the exit at $(5, 2)$.
4. **Prove.** The best route is now `S L R`: $2 + 2 + 2 + 1 = 7$ squares and two turns, 7.64. One frontier is left, the stub below $(4, 4)$: reach$(4, 4) = 8.64$ and $h = |4 - 5| + |4 - 2| = 3$, and $8.64 + 3 = 11.64 \not< 7.64$. It is never driven; exploration ends at the exit.
5. **Home and race.** `toward(start)` brings it back to the start; the speed run drives `S L R`.

# 10 The speed run

The stored route is not only a list of turns but also the length of every road on it. That changes how the robot can drive:

- On a road of known length $\ell$, cruise at the fast speed and drop to exploring speed for the last `BRAKE_MM` $= 120$ mm, so the junction is read as calmly as during exploration.
- At each node, take the stored turn. If what the sensors see does not offer that turn (a junction was missed earlier), fall back to the left-hand rule and stop trusting the stored distances, rather than turn into nothing.

**INTUITION**

The first run is a careful walk with a notebook. The second is a sprint with the notebook already read: the robot knows that the next crossing is 80 cm away and that it turns left there.

# 11 Walls instead of tape

The solver above never asks *how* the robot got from one node to the next. It needs exactly two things from the hardware:

1. `road()`: drive to the next place with a choice, and report which ways are open and how far it was;
2. `turn(L|S|R|B)`.

In `robot.cpp` these are a pair of function pointers, `Driver{road, turn}`. Tape mode plugs in the line-following versions. Wall mode plugs in:

| | **Tape** | **Walls** |
|---|---|---|
| Stay on course | PD on line position $p$ | P on $(d_L - d_R)/2$, distances to the walls |
| Notice a choice | outer sensor sees black | a side distance jumps past $\frac{3}{4}$ of a corridor, twice in a row |
| Stop on the node | creep $a$ past the junction | roll half a corridor past the wall end, or stop when the front wall is at $\frac{g}{2}-$(nose) |
| Turn | until the line appears | 90° or 180° by the encoders |
| Exit | black square | black floor (the line sensors are still on board) |

A wall maze is a grid by construction, so wall mode always snaps road lengths to whole corridors. The map, the proof, the drive home and the speed run are the same code.

# 12 How it was tested

## 12.1 The solver alone, on a PC

`maze.h` and `maze.cpp` contain no hardware calls, so they compile on a PC. `test/maze_test.cpp` drives a simulated robot through random mazes:

1. Generate a random spanning tree on a grid of 3–6 × 3–6 points (a maze without loops), pick two line ends as start and exit, then add 0–4 extra roads that close loops.
2. Simulate the robot: follow roads, stop where a real robot would stop, report the exits it would see, and measure each road with **noise**: ±1 % scale and ±6 mm offset (five times more in a "rough" variant, run only with grid snapping).
3. Explore until `explore()` says done, drive home, then replay the stored route in the true maze.
4. Compare its cost with the true optimum, found by brute force on the true maze.

| **Result over 3000 random mazes with loops** | |
|---|---|
| stored route is the true shortest | 3000 / 3000 |
| average driving to first reach the exit | 3.4 m |
| average driving to prove the route and return home | 6.5 m |

The same algorithm, ported to JavaScript in the Maze Lab simulator (`docs/simulator.html`), passes the same kind of check against 3000 random mazes.

## 12.2 What is *not* tested

The solver is proven and tested; the robot around it is not yet. Sensor noise on a real floor, wheel slip on a fast turn, and the wall-mode constants are all still to be tuned on the finished chassis.

# 13 Limits, and where to go next

**Right angles only.**
The heading is counted, not measured. Diagonal or curved tape would need continuous odometry, and probably a gyroscope.

**Odometry error around large loops.**
Each road adds a little error. Without a grid, a long loop can close more than $m$ away from where it started. Remedy: draw mazes on a grid, or add loop-closure correction (spread the error back over the loop, as SLAM systems do).

**60 nodes, 64 turns.**
Ample for club mazes; raise `MAX_NODES` for bigger ones. Memory grows linearly, search time quadratically.

**Time versus distance.**
$\tau = 80$ mm is a guess at what a turn costs in time. Timing real turns and straights on the finished robot would give the true value.

**Exploration is not optimal.**
The robot finds the optimal *route*, but the *exploration* is a good heuristic (nearest frontier), not the shortest possible tour. That problem is much harder (related to the travelling-salesman problem) and the gains would be small.

# A Glossary

**A*** A shortest-path search that uses an estimate of the remaining distance to look at promising nodes first. Correct as long as the estimate never overestimates.

**Calibration** Recording the whitest and blackest reading of each sensor so readings can be scaled to 0–1000.

**Centroid** Weighted average position; here, of the blackness under the sensor row.

**Depth-first search** Explore one branch to its end before trying the next.

**Dijkstra's algorithm** Finds cheapest paths from one start in a graph with non-negative costs.

**Encoder** Sensor on the motor that produces pulses as it turns; counting them gives distance.

**Frontier** A road seen from a node but not yet driven.

**Heading** Which of north, east, south, west the robot faces; north is wherever it faced at the start.

**Manhattan distance** $|dx| + |dy|$: distance along a grid of streets.

**Odometry** Working out position from wheel motion.

**PD controller** Steering from the error (P) and its rate of change (D).

**State** A node together with a heading; the unit the shortest-path search works on.

**Tree** A connected graph without cycles; a maze without loops.

# B Where everything is in the code

| File | Contents |
|---|---|
| `sensors.cpp` | calibration, normalisation, line position (ch. 2) |
| `robot.cpp` | `followSegment` (PD), `scanJunction`, `turn`, `driveRoad`, `wallRoad`, exploration and speed-run drivers (ch. 3, 4, 10, 11) |
| `path.cpp` | the turn list and dead-end reduction (ch. 5) |
| `encoders.cpp` | quadrature counting, distance, angle (ch. 6) |
| `maze.h`, `maze.cpp` | the map, `arrive`, Dijkstra, `explore`, `route` (ch. 7–9) |
| `test/maze_test.cpp` | the random-maze test (ch. 12) |
| `docs/simulator.html` | the same solver in JavaScript, animated |

# C Further reading


- E. W. Dijkstra, “A note on two problems in connexion with graphs”, *Numerische Mathematik* 1 (1959) 269–271.
- P. E. Hart, N. J. Nilsson, B. Raphael, “A formal basis for the heuristic determination of minimum cost paths”, *IEEE Trans. Systems Science and Cybernetics* 4(2) (1968) 100–107. The A* paper.
- T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, chapters on graph search and shortest paths.
- Pololu, *3pi Robot User’s Guide*, section “Maze solving”: the classic turn-list method. `https://www.pololu.com/docs/0J21/8`
- S. Thrun, W. Burgard, D. Fox, *Probabilistic Robotics*, MIT Press, 2005: odometry models and mapping.
- Wikipedia, “Maze-solving algorithm”: wall follower, Trémaux, flood fill.