Digital Object Identifier

Micromouse Maze Navigation with State–Space Modeling

ABDALLAH ABOSHOAIB[1], RAMI ARQOUB[1], ASEEL RABEE[1], and NOURA KHDOUR[1]
[1]Department of Electrical and Computer Engineering, Birzeit University, Birzeit – P.O. 73 Palestine (e–mail: {1210211, 1210245, 1210627, 1212072}@birzeit.edu)

ABSTRACT We develop a complete state–space simulation for a differential–drive Micromouse tasked with moving from $(0, 0)$ to $(0.5, 0.5)$ m while avoiding static walls. Continuous dynamics are discretised at $\Delta t = 0.01$ s; noisy encoders and a gyroscope feed an Extended Kalman Filter (EKF) that recovers full state estimates. A hybrid finite–state/PID controller uses those estimates to reach the goal with centimetre accuracy. In the single–wall maze the robot arrives in 6.94 s (RMS error 1.8 cm); with two walls the time rises to 8.99 s and RMS error 2.1 cm.

INDEX TERMS Micromouse, mobile robotics, state–space modeling, extended Kalman filter, hybrid control

I. INTRODUCTION

Maze–solving micromice are a classic test–bed for mobile–robot algorithms. Our goal is to model the robot, estimate its state in the presence of sensor noise, and design a controller that respects motor limits while maintaining a clearance from user–defined walls (`walls.json`).

II. SYSTEM MODEL

A. PHYSICAL PARAMETERS

Wheel radius $r = 0.02$ m, wheelbase $d = 0.10$ m, mass $m = 0.20$ kg, inertia $I = 5 \times 10^{-4}$ kg·m^2, motor constant $k_m = 0.10$ N·m/V, friction $b = 0.01$ N·s/m, $b_\theta = 0.001$ N·m·s/rad.

B. STATE–SPACE REPRESENTATION

The state vector $\mathbf{x} = [x,\ y,\ \theta,\ v,\ \omega]^\mathsf{T}$ evolves as

$$\dot{x} = v\cos\theta, \qquad \dot{y} = v\sin\theta, \tag{1}$$

$$\dot{\theta} = \omega, \tag{2}$$

$$\dot{v} = \frac{k_m}{mr}(V_L + V_R) - \frac{b}{m}v, \tag{3}$$

$$\dot{\omega} = \frac{dk_m}{2Ir}(V_R - V_L) - \frac{b_\theta}{I}\omega. \tag{4}$$

Euler forward with $\Delta t = 0.01$ s yields $\mathbf{x}_{k+1} = \mathbf{x}_k + \Delta t\, f(\mathbf{x}_k, \mathbf{u}_k)$.

III. SENSOR MODEL AND EKF

Wheel encoders measure (ω_L, ω_R); a MEMS gyro measures ω. Independent white Gaussian noise with $\sigma_{\text{enc}} = 0.05$ rad/s and $\sigma_{\text{gyro}} = 0.02$ rad/s is added.

We employ an EKF with process noise $\mathbf{Q} = \text{diag}(0.001)$ and measurement noise $\mathbf{R} = \text{diag}(0.05^2, 0.05^2, 0.02^2)$. The Jacobians follow directly from (4); the full Python implementation is shown in Listing **??**.

IV. CONTROL ARCHITECTURE

A. PID BLOCKS

Two independent PID controllers regulate (i) heading error $\tilde{\theta} = \theta_{\text{des}} - \theta$ and (ii) distance to the next Manhattan segment. Gains were tuned empirically $k_p^{(\theta)} = 8$, $k_d^{(\theta)} = -0.1$; $k_p^{(v)} = 4$, $k_d^{(v)} = 0.1$.

B. FINITE–STATE REACTIVE LAYER

A light FSM enforces wall clearance: `go_to_goal` → `turn_away` → `clear_wall` → `turn_to_goal` → `go_to_goal`. Transition guards use the estimated pose and a dynamic clearance $d_{\text{clr}} = 0.015 + 0.7\,|v|/v_{\max}$.

Wheel voltages follow $V_{L,R} = \frac{mr}{k_m}\left(v_{\text{des}} \mp \frac{d}{2}\omega_{\text{des}}\right)$, then are clipped to ± 5 V.

V. SIMULATION SETUP

- Time step $\Delta t = 0.01$ s; horizon $T_{\max} = 10$ s.
- Goal $(0.5, 0.5)$ m; start at the origin.
- Two wall configurations: single wall and double wall.
- Random seed 42 for reproducibility.

VI. RESULTS AND DISCUSSION

Figure 1 shows an L–shaped path for the single–wall maze: The robot aligns along $y = 0.21$ m and turns upward once clear of the wall. The EKF (dashed) hugs the ground–truth trajectory with a mean absolute error of 1.3 cm.

TABLE 1. Finite–state controller transition table

Current state	Condition / Event	Next state	Key action(s)
`go_to_goal`	Wall within d_{clr}	`turn_away`	Reset PIDs; stop robot; set spin target $\theta \pm \pi/2$
	$\lvert e_\theta \rvert > \varepsilon$	stay	$\omega_{\text{des}} = k_p^{(\theta)} e_\theta$ (in–place rotate)
	otherwise	stay	$v_{\text{des}} = k_p^{(v)} d$ (drive straight)
`turn_away`	$\lvert e_\theta \rvert < \varepsilon$	`clear_wall`	$v_{\text{des}} = 0.8$ m/s, $\omega_{\text{des}} = 0$
	otherwise	stay	$\omega_{\text{des}} = -\pi/2$ rad/s (constant spin)
`clear_wall`	progress $> \lVert p_2 - p_1 \rVert + 0.05$ m	`turn_to_goal`	Compute new heading $\arctan 2(g_y - y,\ g_x - x)$
	otherwise	stay	Maintain $v_{\text{des}} = 0.8$ m/s
`turn_to_goal`	$\lvert e_\theta \rvert < \varepsilon$	`go_to_goal`	Reset PIDs
	otherwise	stay	$\omega_{\text{des}} = 2.5\, e_\theta$ (proportional spin)

TABLE 2. Navigation performance

Scenario	Time to goal (s)	RMS error (m)
Single wall	6.94	0.018
Two walls	8.99	0.021

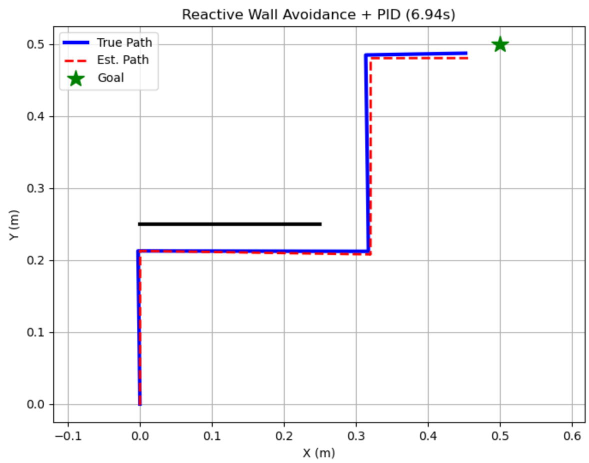

FIGURE 1. Single–wall scenario: blue = true, red dashed = EKF; green star = goal.

The double–wall case (Fig. 2) shows a second `turn_away` event around $x \approx 0.3$ m. Despite the extra manoeuvre, RMS error only increases by 0.3 cm.

VII. CONCLUSION

State–space modelling, EKF estimation, and a hybrid PID/FSM controller guide a micromouse through mazes with sub–2 cm accuracy. Future work includes model–predictive control and adaptive noise covariance.

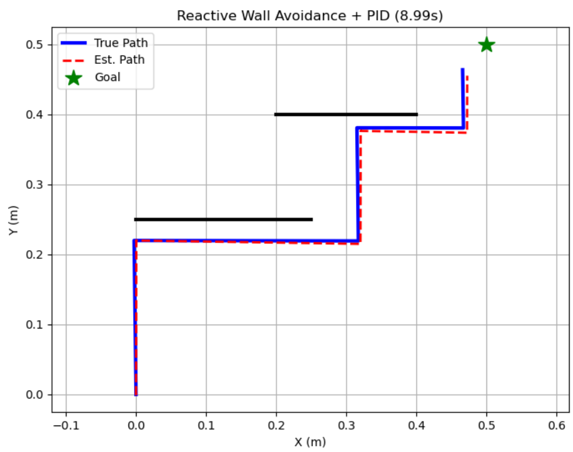

FIGURE 2. Two–wall scenario. Estimation accuracy remains high.

APPENDIX. COMPLETE PYTHON SOURCE

```python
import json
import numpy as np
import matplotlib.pyplot as plt
from pathlib import Path

# ------------------ load obstacle config
    ------------------
def load_config(path="walls.json"):
    cfg        = json.loads(Path(path).read_text())
    walls      = [
        (np.array(w["start"], dtype=float),
         np.array(w["end"],   dtype=float))
        for w in cfg["walls"]
    ]
    clearance = float(cfg.get("clearance", 0.05))
    return walls, clearance

# ------------------ Physical & Motor Params
    ------------------
r, d = 0.02, 0.10
m, I = 0.20, 0.0005
km   = 0.10
b, b   = 0.01, 0.001

# ------------------ Simulation + Noise Params
    ------------------
dt, T_max      = 0.01, 10.0
 _enc ,  _gyro  = 0.05, 0.02
Q = np.diag([0.001]*5)
R = np.diag([ _enc **2,  _enc **2,  _gyro **2])

# ------------------ Dynamics & Sensors (for EKF)
    ------------------
def state_dynamics(x, u):
    _, _,   , v,    = x
    Vl, Vr         = u
    return np.array([
        v*np.cos(  ),
        v*np.sin(  ),
         ,
        (km/(m*r))*(Vl+Vr) - (b/m)*v,
        (d*km/(2*I*r))*(Vr-Vl) - (b  /I)*
    ])

def discretize(x, u):
    return x + dt * state_dynamics(x, u)

def sense(x, rng, noisy=True):
    v,     = x[3], x[4]
     l     = (v + (d/2)*   )/r
     r     = (v - (d/2)*   )/r
    z     = np.array([ l ,  r ,    ])
    if noisy:
        z += rng.normal(0, [ _enc ,  _enc ,
             _gyro ])
    return z

# ------------------ EKF ------------------
def jac_F(x):
      , v = x[2], x[3]
    F = np.eye(5)
    F[0,2] = -dt*v*np.sin(  )
    F[0,3] =  dt*np.cos(  )
    F[1,2] =  dt*v*np.cos(  )
    F[1,3] =  dt*np.sin(  )
    F[2,4] =  dt
    F[3,3] = 1 - dt*(b/m)
    F[4,4] = 1 - dt*(b  /I)
    return F

def jac_H():
    return np.array([
        [0,0,0, 1/r,  d/(2*r)],
        [0,0,0, 1/r, -d/(2*r)],
        [0,0,0,   0,       1]
    ])

def EKF(x  , P, y, u, rng):
    x_pred = discretize(x  , u)
    F      = jac_F(x  )
    P_pred = F @ P @ F.T + Q

    H      = jac_H()
    y_pred = sense(x_pred, rng, noisy=False)
    S      = H @ P_pred @ H.T + R
    K      = P_pred @ H.T @ np.linalg.inv(S)
    x_upd  = x_pred + K @ (y - y_pred)
    P_upd  = (np.eye(5) - K @ H) @ P_pred
    return x_upd, P_upd

# ------------------ Simple PID Controller
    ------------------
class PID:
    def __init__(self, kp, ki, kd):
        self.kp, self.ki, self.kd = kp, ki, kd
        self.integral = 0.0
        self.prev_error = 0.0

    def reset(self):
        self.integral = 0.0
        self.prev_error = 0.0

    def compute(self, error, dt):
        self.integral += error * dt
        derivative = (error - self.prev_error) /
            dt
        self.prev_error = error
        return self.kp*error + self.ki*self.
            integral + self.kd*derivative

# ------------------ Reactive Avoidance Controller
     (multiwall) w/ PID ------------------
class ReactiveController:
    """
    Reactive  wall avoidance  + PID controller.
    Moves in a Manhattan pattern toward a goal,
    but spins/backs up when it gets too close to
        any wall.
    """
    def __init__(
        self,
        walls,             # list of (p1,p2) wall
            segments as np.array pairs
        clearance,         # how close before we
            start avoiding
        dt=0.01,           # simulation timestep
        # PID gains: (kp, ki, kd)
        ang_gains=(8, 0.0, -0.1),
        lin_gains=(4, 0.0, 0.1),
```

```
    v_max=1.0,        # max forward speed (m/s
        )
    speed_boost=1.5,  # scales wheel voltages
):
    self.walls          = walls
    self.clearance      = clearance
    self.dt             = dt

    # controller state
    self.state          = "go_to_goal"
    self.current_wall = None
    self.turn_target    = None

    # thresholds
    self.angle_eps      = 0.005
    self.align_thresh = 0.02
    self. _spin          = np.pi / 2

    # limits & scaling
    self.v_max          = v_max
    self.speed_boost    = speed_boost

    # PID for angle and distance
    self.pid_ang = PID(*ang_gains)
    self.pid_lin = PID(*lin_gains)



def _find_wall_zone(self, x, y, v):
    dyn_clr = max(0.015, self.clearance * (0.3
         + 0.7*abs(v)/self.v_max))
    for p1, p2 in self.walls:
       a    = p2 - p1
       t    = np.clip(np.dot([x, y]-p1, a)/np.
            dot(a, a), 0, 1)
       clo = p1 + t*a
       if np.linalg.norm([x-clo[0], y-clo[1]])
            <= dyn_clr:
         return (p1, p2)
    return None

def to_wheels(self, v_des,  _des ):
    # convert desired linear/angular vel to
        left/right voltages
    Vl = (v_des - (d/2)* _des ) * (m*r/km) *
        self.speed_boost
    Vr = (v_des + (d/2)* _des ) * (m*r/km) *
        self.speed_boost
    # clip to physical motor limits [-5V, +5V]
    return np.clip(Vl, -5, 5), np.clip(Vr, -5,
        5)

def control(self, x  , goal):
    """
     x  : estimated state [x, y,   , v,   ]
    goal: [gx, gy]
    returns: Vl, Vr, v_des,  _des
    """
    x, y,    = x  [:3]
    gx, gy  = goal

    def angle_err(target):
        # wraparound error in [-  , +  ]
        return (target -    + np.pi) % (2*np.
            pi) - np.pi

    # 1) WALL DETECTION      spin away
    if self.state == "go_to_goal":
        if wall := self._find_wall_zone(x, y,
             x  [3]):
            print(f"         Wall detected! {
                wall}")
            self.current_wall = wall
```

```
            self.state          = "turn_away"
            p1, p2              = wall
            # compute outward normal
            normal = np.array([p1[1]-p2[1], p2
                [0]-p1[0]])
            # pick spin direction so we point
                away
            self.turn_target = (
                  - np.pi/2
                if np.dot(normal, [np.cos(  ),
                     np.sin(  )]) > 0
                else     + np.pi/2
            )
            self.pid_ang.reset()
            self.pid_lin.reset()
            return 0, 0, 0.0, 0.0

        # 2) MANHATTAN MOVE toward goal
        dx, dy = gx-x, gy-y
        if abs(dy) > self.align_thresh:
            target_angle, dist = (np.pi/2 if
                dy>0 else -np.pi/2), abs(dy)
        elif abs(dx) > self.align_thresh:
            target_angle, dist = (0.0 if dx>0
                else np.pi), abs(dx)
        else:
            # already at goal
            return 0, 0, 0.0, 0.0

        err_ang = angle_err(target_angle)
         _des      = self.pid_ang.compute(
            err_ang, self.dt)

        if abs(err_ang) > self.angle_eps:
            v_des = 0.0
        else:
            v_unclipped = self.pid_lin.compute
                (dist, self.dt)
            v_des         = np.clip(v_unclipped,
                 -self.v_max, self.v_max)

        Vl, Vr = self.to_wheels(v_des,  _des )
        return Vl, Vr, v_des,  _des

    # 3) TURN AWAY from wall until aligned
    if self.state == "turn_away":
        err = angle_err(self.turn_target)
        v_des,  _des  = 0.0, -self. _spin
        if abs(err) < self.angle_eps:
            self.state = "clear_wall"
            v_des,  _des  = 0.8, 0.0
        Vl, Vr = self.to_wheels(v_des,  _des )
        return Vl, Vr, v_des,  _des

    # 4) CLEAR past the wall
    if self.state == "clear_wall":
        p1, p2 = self.current_wall
        progress = np.dot(
            [x-p1[0], y-p1[1]],
            [np.cos(self.turn_target), np.sin(
                self.turn_target)]
        )
        if progress > np.linalg.norm(p2-p1) +
            0.05:
            self.state          = "turn_to_goal"
            self.turn_target = np.arctan2(gy-y
                , gx-x)
        v_des,  _des  = 0.8, 0.0
        Vl, Vr = self.to_wheels(v_des,  _des )
        return Vl, Vr, v_des,  _des

    # 5) TURN BACK TOWARD goal
    if self.state == "turn_to_goal":
        err = angle_err(self.turn_target)
```

```
            v_des,  _des  = 0.0, 2.5 * err
            if abs(err) < self.angle_eps:
                self.state = "go_to_goal"
                self.pid_ang.reset()
                self.pid_lin.reset()
            Vl, Vr = self.to_wheels(v_des,  _des )
            return Vl, Vr, v_des,  _des

        # fallback
        return 0, 0, 0.0, 0.0

# ------------------ Simulation ------------------
def simulate(walls, clearance):
    x_true = np.zeros(5)
    x_est  = x_true.copy()
    P      = np.eye(5) * 0.1
    rng    = np.random.default_rng(42)
    ctrl   = ReactiveController(walls, clearance)
    goal   = np.array([0.5, 0.5])

    traj_t, traj_e = [x_true[:2].copy()], [x_est
        [:2].copy()]
    t = 0.0

    while t < T_max:
        Vl, Vr, v_des,  _des  = ctrl.control(x_est
            , goal)

        # true motion update
           = x_true[2]
        x_true[0] += v_des*np.cos(  )*dt
        x_true[1] += v_des*np.sin(  )*dt
        x_true[2] +=  _des *dt
        x_true[3], x_true[4] = v_des,  _des

        # EKF update
        z        = sense(x_true, rng, noisy=True)
        x_est,P = EKF(x_est, P, z, (Vl, Vr), rng)

        traj_t.append(x_true[:2].copy())
        traj_e.append(x_est[:2].copy())

        if np.hypot(*(x_true[:2] - goal)) < 0.05:
            print(f"     Goal reached in {t:.2f}s"
                )
            break

        t += dt

    return np.array(traj_t), np.array(traj_e), t

# ------------------ Plotting ------------------
def plot_traj(tr_t, tr_e, walls, sim_t):
    plt.figure(figsize=(8,6))
    plt.plot(tr_t[:,0], tr_t[:,1], 'b-', lw=3,
        label='True Path')
    plt.plot(tr_e[:,0], tr_e[:,1], 'r--', lw=2,
        label='Est. Path')
    plt.plot(0.5, 0.5, 'g*', ms=15, label='Goal')
    for p1, p2 in walls:
        plt.plot([p1[0], p2[0]], [p1[1], p2[1]], '
            k-', lw=3)
    plt.axis('equal')
    plt.grid(True)
    plt.xlabel('X (m)')
    plt.ylabel('Y (m)')
    plt.title(f"Reactive Wall Avoidance + PID ({
        sim_t:.2f}s)")
    plt.legend()
    plt.show()

# ------------------ Main ------------------
if __name__ == "__main__":
    walls, clr    = load_config("walls.json")
    t_tr, e_tr, st = simulate(walls, clr)
    plot_traj(t_tr, e_tr, walls, st)
```

APPENDIX. EKF UPDATE CODE

Key update routine:

```
def EKF(x_hat, P, y, u, rng):
    x_pred = discretize(x_hat, u)          # predict
         state
    F      = jac_F(x_hat)                  #
        Jacobian
    P_pred = F @ P @ F.T + Q               # predict
         cov

    H      = jac_H()
    y_pred = sense(x_pred, rng, noisy=False)
    S      = H @ P_pred @ H.T + R
    K      = P_pred @ H.T @ np.linalg.inv(S)

    x_upd  = x_pred + K @ (y - y_pred)    # correct
         state
    P_upd  = (np.eye(5) - K @ H) @ P_pred
    return x_upd, P_upd
```

•••