# AMR LiDAR Simulator & PyQt5 Dashboard

**Quick-Start Guide and Full Source Code**

May 17, 2025

## Contents

# 1 Overview

This repository contains:

`barrier_ws_server.py` A back-end process that

1. generates synthetic 270°/10m LiDAR scans,
2. builds a live occupancy grid,
3. inflates obstacles by the robot radius,
4. plans an A* path that *fits* the footprint,
5. follows the path with a pure-pursuit controller,
6. streams telemetry & the remaining path via WebSockets.

`dashboard.py` A PyQt5 GUI that shows

- the static warehouse floor-plan (top pane),
- the live LiDAR map (bottom pane): grey free space, goal `X`, heading arrow, white A* path, 2 m grid + 4 m numeric ticks,
- telemetry cards (velocity, $\omega$, battery, wheel RPM),
- a *Manual arrow-key drive* toggle.

`main.py` One-liner launcher for the GUI.

`splash_screen.py` Example splash-screen loader (optional).

`resources/` Images (e.g. `splash.png`) referenced by the splash screen.

`amr_interface_guide.tex` This document.

No ROS is required; everything is pure Python.

# 2 Prerequisites

- **Python 3.9**+ (any OS). Verify with `python --version`.
- **Git** (only for cloning / updates).
- **(Recommended) Virtual env**

### Create and activate the virtual environment

```
# Windows (Git Bash / PowerShell):
> python -m venv venv
> venv\Scripts\activate

# macOS / Linux:
$ python3 -m venv venv
$ source venv/bin/activate
```

### Install the three required libraries

```
(venv) $ pip install numpy PyQt5 websockets
```

Nothing else is needed.

## 3 Folder structure after unzip

```
Interface/
 backend/
    barrier_ws_server.py
 frontend/
    dashboard.py
 doc/
    amr_interface_guide.tex
 resources/
    splash.png          (optional)
 main.py
 splash_screen.py
 venv/                   (local virtual-env, ignored by Git)
```

The back-end and front-end folders are optional; flat layout works too (the import paths are relative).

## 4 Running the system

Open two terminals (both inside the virtual-env):

1. **Start the back-end**

```
(venv) $ python backend/barrier_ws_server.py
# ... prints "Serving on ws://localhost:8765"
```

2. **Launch the GUI**

```
(venv) $ python main.py
```

**Workflow**

- Enter a goal (e.g. X = 18, Y = 8) → press *Set goal*. The robot plans a white route and starts moving.
- Tick *Manual arrow-key drive*. Autonomy pauses; use arrow keys or the on-screen buttons.
- Untick the box. Planner re-runs from the current pose and continues to the goal.
- If any LiDAR beam is $<25$cm a red alert appears.

# 5 Full Source Code

## 5.1 backend/barrier_ws_server.py

```
"""
barrier_ws_server.py  v9-fixed
 LiDAR  occupancy (FREE carving, OCC stamping)
 obstacle inflation before A*  path always fits
 pure-pursuit follower, auto re-plan, manual override
 transmits remaining path for UI overlay
"""

import asyncio, json, math, time, base64, random, heapq
import numpy as np, websockets

#  geometry & static map
RES, W, H = 0.10, 200, 100 # 20 m  10 m grid
UNKNOWN, FREE, OCC = 0, 1, 255
GRID = np.zeros((H, W), np.uint8)
GRID[0,:]=GRID[-1,:]=GRID[:,0]=GRID[:,-1]=OCC
for x in range(30, W-30, 40): GRID[10:H-10, x:x+3] = OCC # shelves
rng=np.random.default_rng(42)
```

```
for _ in range(12): # pallets
    rx,ry=rng.integers(35,W-35), rng.integers(15,H-15)
    GRID[ry-5:ry+5, rx-5:rx+5] = OCC
STATIC_B64 = base64.b64encode(GRID.tobytes()).decode()

# LiDAR
MAX_R, ANG0, INC, N_BEAMS = 10.0, -135, 1.0, 271

# robot
RADIUS, BASE = 0.05, 0.30
BODY_RAD = 0.18
R_CELLS = int(BODY_RAD/RES)

# state
x,y,yaw = 2.0, 2.0, 0.0
battery, seq = 100.0, 0
mode = "auto"
goal = None
manual_v = manual_w = 0.0
path, pi = [], 0

#  helpers
def wheel_rpm(v,w):
    vr,vl = v+w*BASE/2, v-w*BASE/2
    return [(vr/(2*math.pi*RADIUS))*60, (vl/(2*math.pi*RADIUS))*60]

def ray(px,py,ang):
    for r in np.arange(0, MAX_R, RES/2):
        gx,gy=int((px+r*math.cos(ang))/RES), int((py+r*math.sin(ang))/RES)
        if gx<0 or gx>=W or gy<0 or gy>=H or GRID[gy,gx]==OCC:
            return r
    return MAX_R

def lidar():
    a0=math.radians(ANG0)
    return [ray(x,y,a0+i*math.radians(INC)+yaw) for i in range(N_BEAMS)]

def update_grid(scan):
    a0=math.radians(ANG0)
    for i,r in enumerate(scan):
        ang=a0+i*math.radians(INC)+yaw
```

```
        for d in np.arange(0, min(r,MAX_R), RES/2):
            gx,gy=int((x+d*math.cos(ang))/RES), int((y+d*math.sin(ang))/RES)
            if 0<=gx<W and 0<=gy<H and GRID[gy,gx]!=OCC:
                GRID[gy,gx]=FREE
        if r<MAX_R:
            hx,hy=int((x+r*math.cos(ang))/RES), int((y+r*math.sin(ang))/RES)
            if 0<=hx<W and 0<=hy<H: GRID[hy,hx]=OCC

def blocked(cx,cy):
    for dy in range(-R_CELLS,R_CELLS+1):
        for dx in range(-R_CELLS,R_CELLS+1):
            if dx*dx+dy*dy>R_CELLS*R_CELLS: continue
            nx,ny=cx+dx, cy+dy
            if 0<=nx<W and 0<=ny<H and GRID[ny,nx]==OCC:
                return True
    return False

def occupied(px,py): # run-time footprint check
    return blocked(int(px/RES), int(py/RES))

def astar(start_xy,goal_xy):
    sx,sy=int(start_xy[0]/RES), int(start_xy[1]/RES)
    gx,gy=int(goal_xy[0]/RES) , int(goal_xy[1]/RES)
    h=lambda n: abs(n[0]-gx)+abs(n[1]-gy)
    OPEN=[(h((sx,sy)),0,(sx,sy),None)]
    came,cost={}, {(sx,sy):0}
    dirs=[(1,0),(-1,0),(0,1),(0,-1)]
    while OPEN:
        _,g,n,par=heapq.heappop(OPEN)
        if n in came: continue
        came[n]=par
        if n==(gx,gy): break
        for dx,dy in dirs:
            nb=(n[0]+dx,n[1]+dy)
            if not(0<=nb[0]<W and 0<=nb[1]<H): continue
            if blocked(*nb): continue
            ng=g+1
            if ng<cost.get(nb,1e9):
                cost[nb]=ng
                heapq.heappush(OPEN,(ng+h(nb),ng,nb,n))
    if (gx,gy) not in came: return []
```

```
    pts=[]
    n=(gx,gy)
    while n:
        pts.append((n[0]*RES+RES/2, n[1]*RES+RES/2)); n=came[n]
    return pts[::-1]

def pursue(tgt):
    dx,dy=tgt[0]-x, tgt[1]-y
    ang=math.atan2(dy,dx)
    err=((ang-yaw+math.pi)%(2*math.pi))-math.pi
    w=1.4*err; v=0.6*max(0,1-abs(err))
    return v,w

#  0.1 s update
def step(dt=0.1):
    global x,y,yaw,battery,seq,path,pi
    scan=lidar(); update_grid(scan)

    if mode=="manual":
        v,w=manual_v,manual_w
    else:
        if goal and (not path or pi>=len(path)-1):
            path,pi=astar((x,y),goal),0
        if path and any(blocked(int(px/RES),int(py/RES))
                        for px,py in path[pi:pi+3]):
            path,pi=astar((x,y),goal),0
        if not path: v=w=0.0
        else:
            tgt=path[pi]
            if math.hypot(tgt[0]-x,tgt[1]-y)<0.25 and pi<len(path)-1:
                pi+=1; tgt=path[pi]
            v,w=pursue(tgt)

    nx,ny=x+v*math.cos(yaw)*dt, y+v*math.sin(yaw)*dt
    if occupied(nx,ny): v,nx,ny=0.0,x,y
    yaw=(yaw+w*dt+math.pi)%(2*math.pi)-math.pi
    x,y=nx,ny
    battery=max(0,battery-0.002); seq+=1
    return v,w,scan,path[pi:] if path else []

#  websocket stuff
```

```
CLIENTS=set()
async def handler(ws):
    global mode,manual_v,manual_w,goal
    CLIENTS.add(ws)
    try:
        async for txt in ws:
            d=json.loads(txt)
            if d.get("type")=="mode": mode=d["mode"]
            elif d.get("type")=="cmd_vel": manual_v,manual_w=d["v"],d["w"]
            elif d.get("type")=="goal": goal=(d["x"],d["y"]); mode="auto"
    finally:
        CLIENTS.remove(ws)

async def telemetry_loop():
    while True:
        if CLIENTS:
            v,w,scan,rem=step()
            pkt={
                "type":"telemetry","seq":seq,"ts":int(time.time()*1000),
                "pose":{"x":round(x,2),"y":round(y,2),
                        "yaw":round(math.degrees(yaw),1)},
                "battery":round(battery,1),
                "enc_rpm":[round(r,1) for r in wheel_rpm(v,w)],
                "scan":{"angle_min":ANG0,"angle_inc":INC,"ranges":scan},
                "path":rem,
                "grid":{"w":W,"h":H,"data":STATIC_B64}
            }
            await asyncio.gather(*[c.send(json.dumps(pkt)) for c in CLIENTS])
        await asyncio.sleep(0.1)

async def main():
    async with websockets.serve(handler, "", 8765):
        await telemetry_loop()

if __name__ == "__main__":
    asyncio.run(main())
```

## 5.2 frontend/dashboard.py

```
# dashboard.py  two-pane UI
```

```
#  top : static warehouse map (never changes)
#  bottom : live Li-DAR map + heading arrow + A* path + goal X
#  grey grid every 2 m, numeric ticks every 4 m
#  manual arrow-key override / auto toggle
#  full int() casts  no Qt TypeError

import sys, json, math, base64, numpy as np
from PyQt5.QtCore import Qt, QUrl, QPointF
from PyQt5.QtGui import (QPixmap, QImage, QPainter, QBrush,
                         QPolygonF, QColor)
from PyQt5.QtWidgets import (QApplication, QWidget, QLabel, QPushButton,
                         QVBoxLayout, QHBoxLayout, QGridLayout, QFrame,
                         QTextEdit, QCheckBox, QLineEdit)
from PyQt5.QtWebSockets import QWebSocket
from PyQt5.QtNetwork import QAbstractSocket # ConnectedState


#
class Card(QFrame):
    def __init__(self, title: str):
        super().__init__()
        self.setStyleSheet(
            "QFrame{background:#2b2d3a;border-radius:10px}"
            "QLabel{color:white}")
        v = QVBoxLayout(self)
        t = QLabel(title); t.setStyleSheet("font-size:12px")
        self.val = QLabel("--"); self.val.setStyleSheet("font-size:19px")
        v.addWidget(t); v.addWidget(self.val); v.addStretch()


#
class Dashboard(QWidget):
    #  init
    def __init__(self):
        super().__init__()
        self.setWindowTitle("AMR UI  auto-plan  manual drive")
        self.resize(1280, 720)
        self.setFocusPolicy(Qt.StrongFocus)

        # robot / map state
        self.res = 0.10 # metres per cell
```

```
        self.base = None # static occupancy grid (numpy)
        self._static_done = False # drawn once flag
        self.prev_pose = self.prev_ts = None
        self.manual = False
        self.goal = None # (gx,gy)
        self.path = [] # list of way-points

        #  layout ------------------------------------------------------------
        root = QHBoxLayout(self)

        #  left column  two stacked map panes
        maps_col = QVBoxLayout()
        self.static_lbl = QLabel(); self.static_lbl.setMinimumSize(600, 300)
        self.static_lbl.setStyleSheet("background:#111")
        self.live_lbl = QLabel(); self.live_lbl.setMinimumSize(600, 300)
        self.live_lbl.setStyleSheet("background:#111")
        maps_col.addWidget(self.static_lbl); maps_col.addWidget(self.
            live_lbl)
        root.addLayout(maps_col, 3)

        #  right column
        right = QVBoxLayout(); root.addLayout(right, 2)

        # telemetry cards
        grid = QGridLayout(); right.addLayout(grid)
        self.cards = {}
        for n, (k, t) in enumerate([
            ("velocity", "Velocity (m/s)"), ("omega", "Omega (rad/s)"),
            ("battery", "Battery (%)"), ("rpm_l", "RPM-L"),
            ("rpm_r", "RPM-R")]):
            c = Card(t); self.cards[k] = c.val
            grid.addWidget(c, n // 2, n % 2)

        # goal entry row
        goal_row = QHBoxLayout(); right.addLayout(goal_row)
        goal_row.addWidget(QLabel("Goal X:"))
        self.goal_x = QLineEdit("18"); self.goal_x.setFixedWidth(60)
        goal_row.addWidget(self.goal_x)
        goal_row.addWidget(QLabel("Y:"))
        self.goal_y = QLineEdit("8"); self.goal_y.setFixedWidth(60)
        goal_row.addWidget(self.goal_y)
```

```
        set_btn = QPushButton("Set goal"); goal_row.addWidget(set_btn)
        set_btn.clicked.connect(self._set_goal)

        # manual toggle
        self.cb = QCheckBox("Manual arrow-key drive")
        self.cb.stateChanged.connect(self._toggle_mode)
        right.addWidget(self.cb)

        # log pane
        right.addWidget(QLabel("Logs:"))
        self.log = QTextEdit(readOnly=True)
        self.log.setStyleSheet("background:#101;color:#ccc")
        right.addWidget(self.log, 1)

        # arrow / stop buttons (only active in manual)
        row = QHBoxLayout(); right.addLayout(row)
        for k, txt in [("l",""), ("f",""), ("r",""),
                       ("b",""), ("s","")]:
            b = QPushButton(txt); b.setFixedSize(48,48)
            if k == "s": b.setStyleSheet("background:red;color:white")
            row.addWidget(b); b.clicked.connect(
                lambda _, kk=k: self._btn(kk))

        # websocket
        self.ws = QWebSocket()
        self.ws.textMessageReceived.connect(self._rx)
        self.ws.open(QUrl("ws://localhost:8765"))

    #  WebSocket helpers
    def _send(self, obj: dict):
        if self.ws.state() == QAbstractSocket.ConnectedState:
            self.ws.sendTextMessage(json.dumps(obj))

    def _toggle_mode(self, state: int):
        self.manual = bool(state)
        self._send({"type": "mode",
                    "mode": "manual" if self.manual else "auto"})

    def _set_goal(self):
        try:
            gx = float(self.goal_x.text())
```

```
            gy = float(self.goal_y.text())
        except ValueError:
            self.log.append("<b>Invalid goal coordinates</b>")
            return
        self.goal = (gx, gy)
        self._send({"type": "goal", "x": gx, "y": gy})
        if self.manual: # flip back to auto
            self.cb.setChecked(False) # triggers _toggle_mode

    # arrow / stop buttons
    def _btn(self, k: str):
        if not self.manual:
            return
        v, w = {"f": (0.5, 0), "b": (-0.5, 0),
                "l": (0, 1.2), "r": (0, -1.2),
                "s": (0, 0)}[k]
        self._send({"type": "cmd_vel", "v": v, "w": w})

    # arrow-key hold-to-drive
    def keyPressEvent(self, e):
        if not self.manual or e.isAutoRepeat(): return
        m = {Qt.Key_Up: (0.5, 0),
             Qt.Key_Down: (-0.5, 0),
             Qt.Key_Left: (0, 1.2),
             Qt.Key_Right: (0, -1.2)}
        if e.key() in m:
            v, w = m[e.key()]
            self._send({"type": "cmd_vel", "v": v, "w": w})

    def keyReleaseEvent(self, e):
        if not self.manual or e.isAutoRepeat(): return
        if e.key() in (Qt.Key_Up, Qt.Key_Down,
                       Qt.Key_Left, Qt.Key_Right):
            self._send({"type": "cmd_vel", "v": 0, "w": 0})

    #  WebSocket RX
    def _rx(self, txt: str):
        d = json.loads(txt)
        if d.get("type") != "telemetry" or "pose" not in d:
            return
```

```
        # path from server (remaining way-points)
        if "path" in d: self.path = d["path"]

        pose, ts = d["pose"], d["ts"]

        # cards
        if self.prev_pose:
            dt = (ts - self.prev_ts) / 1000
            dx = pose["x"] - self.prev_pose["x"]
            dy = pose["y"] - self.prev_pose["y"]
            v = math.hypot(dx, dy) / dt
            dyaw = (pose["yaw"] - self.prev_pose["yaw"] + 180) % 360 - 180
            w = math.radians(dyaw) / dt
            self.cards["velocity"].setText(f"{v:.2f}")
            self.cards["omega"] .setText(f"{w:.2f}")
        self.prev_pose, self.prev_ts = pose, ts
        self.cards["battery"].setText(f"{d['battery']:.0f}")
        self.cards["rpm_l"] .setText(f"{d['enc_rpm'][0]:.0f}")
        self.cards["rpm_r"] .setText(f"{d['enc_rpm'][1]:.0f}")

        # proximity alert
        closest = min(d["scan"]["ranges"])
        if closest < 0.25:
            self.log.append(
                f'<span style="color:#ff6b6b">ALERT {closest:.2f} m  '
                'obstacle very close!</span>')

        # static grid comes once
        if self.base is None:
            g = d["grid"]
            self.base = np.frombuffer(base64.b64decode(g["data"]), np.uint8
                      ).reshape((g["h"], g["w"]))

        self._draw(pose, d["scan"])

    #  drawing helper
    def _draw(self, pose: dict, scan: dict):
        img = self.base.copy()
        a0 = math.radians(scan["angle_min"])
        inc = math.radians(scan["angle_inc"])
        for i, r in enumerate(scan["ranges"]):
```

```
            if r >= 10.0: continue
            ang = a0 + i * inc + math.radians(pose["yaw"])
            gx = pose["x"] + r * math.cos(ang)
            gy = pose["y"] + r * math.sin(ang)
            cx, cy = int(gx / self.res), int(gy / self.res)
            if 0 <= cy < img.shape[0] and 0 <= cx < img.shape[1]:
                img[cy, cx] = 200 # grey free-space

        h, w = img.shape
        qimg = QImage(img.data, w, h, QImage.Format_Grayscale8)
        pix = QPixmap.fromImage(qimg).scaled(
                  self.live_lbl.size(),
                  Qt.KeepAspectRatio,
                  Qt.FastTransformation)

        painter = QPainter(pix)
        sx, sy = pix.width() / w, pix.height() / h

        # grid every 2 m; tick labels every 4 m
        painter.setPen(QColor(55, 55, 55))
        step = int(2 / self.res)
        for cx in range(0, w, step):
            x_pix = int(cx * sx)
            painter.drawLine(x_pix, 0, x_pix, pix.height())
        for cy in range(0, h, step):
            y_pix = int(cy * sy)
            painter.drawLine(0, y_pix, pix.width(), y_pix)
        painter.setPen(Qt.gray)
        tick = int(4 / self.res)
        for cx in range(0, w, tick):
            painter.drawText(int(cx * sx) + 2, 12, str(int(cx * self.res)))
        for cy in range(0, h, tick):
            painter.drawText(2, int(cy * sy) - 2, str(int(cy * self.res)))

        # A* path (white poly-line)
        if self.path and len(self.path) > 1:
            painter.setPen(Qt.white)
            for i in range(len(self.path) - 1):
                x1, y1 = self.path[i]
                x2, y2 = self.path[i + 1]
                painter.drawLine(int(x1 / self.res * sx),
```

```
                        int(y1 / self.res * sy),
                        int(x2 / self.res * sx),
                        int(y2 / self.res * sy))

    # robot square + heading arrow
    cx_pix = pose["x"] / self.res * sx
    cy_pix = pose["y"] / self.res * sy
    painter.setBrush(QBrush(Qt.white))
    painter.drawRect(int(cx_pix) - 4, int(cy_pix) - 4, 8, 8)

    yaw_rad = math.radians(pose["yaw"])
    tip = QPointF(cx_pix + 12 * math.cos(yaw_rad),
                  cy_pix + 12 * math.sin(yaw_rad))
    left = QPointF(cx_pix + 6 * math.cos(yaw_rad + 2.6),
                  cy_pix + 6 * math.sin(yaw_rad + 2.6))
    right= QPointF(cx_pix + 6 * math.cos(yaw_rad - 2.6),
                  cy_pix + 6 * math.sin(yaw_rad - 2.6))
    painter.drawPolygon(QPolygonF([tip, left, right]))

    # goal mark
    if self.goal:
        gx_pix = self.goal[0] / self.res * sx
        gy_pix = self.goal[1] / self.res * sy
        painter.drawLine(int(gx_pix - 6), int(gy_pix - 6),
                         int(gx_pix + 6), int(gy_pix + 6))
        painter.drawLine(int(gx_pix - 6), int(gy_pix + 6),
                         int(gx_pix + 6), int(gy_pix - 6))

    painter.end()
    self.live_lbl.setPixmap(pix)

    # draw static map once (scaled)
    if not self._static_done:
        s_img = QImage(self.base.data, w, h, QImage.Format_Grayscale8)
        s_pix = QPixmap.fromImage(s_img).scaled(
                    self.static_lbl.size(),
                    Qt.KeepAspectRatio,
                    Qt.FastTransformation)
        self.static_lbl.setPixmap(s_pix)
        self._static_done = True
```

```
#  run app
if __name__ == "__main__":
    app = QApplication(sys.argv)
    Dashboard().show()
    sys.exit(app.exec_())
```

## 5.3 main.py

Listing 1: GUI launcher

```
from PyQt5.QtWidgets import QApplication
from frontend.dashboard import Dashboard
import sys

if __name__ == "__main__":
    app = QApplication(sys.argv)
    Dashboard().show()
    sys.exit(app.exec_())
```

## 5.4 splash_screen.py (optional)

```
"""Simple splash screen loader (requires resources/splash.png)."""
from PyQt5.QtWidgets import QApplication, QSplashScreen
from PyQt5.QtGui import QPixmap
from frontend.dashboard import Dashboard
import sys, time

if __name__ == "__main__":
    app = QApplication(sys.argv)
    splash = QSplashScreen(QPixmap("resources/splash.png"))
    splash.show()
    app.processEvents()
    time.sleep(2) # hold the splash for 2s
    splash.close()
    Dashboard().show()
    sys.exit(app.exec_())
```

# 6 Verifying functionality

1. Both terminals must print *no errors*. The back-end shows increasing sequence numbers (`seq`) every 0.1s.
2. The GUI shows:
   - Black static map (top),
   - Live map (bottom) with grey dots expanding,
   - A white square with a little arrow,
   - A white poly-line toward the red `X`.
3. Toggle manual mode—square follows your keys, velocity/omega cards update live.
4. Untick manual—square returns to the poly-line.

# 7 Troubleshooting

`ModuleNotFoundError: PyQt5` Re-run `pip install PyQt5`.

`Permission denied (publickey)` **when** `git push` Add your SSH key in GitHub or push over HTTPS with a Personal Access Token.

**Black live map** Make sure the back-end script is running — the GUI waits for the first WebSocket packet before drawing.

# 8 Next steps

- Replace the synthetic LiDAR with a real `sensor_msgs/LaserScan` bridge.
- Swap pure-pursuit for DWA or TEB to obey dynamic constraints.
- Log telemetry to CSV / InfluxDB for offline analysis.