ANSWERS

U9.9 Planning for a car

Planning · University · about 40 min

What this lesson is about

Paths a car can drive: Dubins curves, A* over the car's own moves in position and heading, and replanning after every move so the errors never add up.

- Why a grid path will not do
- The shortest car path: Dubins
- Searching over the car's moves
- Driving the plan: feedback by planning again

The task: a U-turn round the wall, driven like a car

The green bay is on the other side of the wall, a 24 cm square round the goal, and the car must finish in it facing back down the mat (180 degrees, within 25). Driving one plan to the end misses it, as above. Write `plan(sx, sy, sh)`, an A* search over the car's six moves: straight, full lock left (steer -1) and full lock right (steer 1), each held for `STEP_S` and for `STEP_S / 2` (write the half length as `STEP_S / 2` in your code, not as a number, so the checker can find it). It takes the robot's pose on the mat, `x` and `y` in cm and the heading in degrees, and returns the moves to the goal as a list of `(steer, seconds)` pairs, `[]` if the robot is already there, or `None` if there is no way. A move costs the distance it drives, `V_SEEN * seconds`, with turning a little more. A state has arrived when it is within 8 cm of the goal and 15 degrees of 180. The first move from the robot's own pose is checked against half the margin, every later one against the whole of it. Then write the loop. Each time round, plan from `here()`, print `plan <n> has <k> moves` (n counts the plans, k is how many moves this one has), drive only the first move with `car(SPEED, steer, seconds)`, and plan again. Stop when the plan comes back empty, and say so if it comes back with no path at all: those are different endings. No sliding, no turning on the spot, no touching the wall, and no list of moves typed in by hand. `free()`, `arc()`, `here()` and `car()` are written for you.

```
from bugbot import *
import heapq, math
connect()

V_MAX, W_MAX, R_MIN = 20.0, 120.0, 25.0
SPEED, STEP_S = 60, 1.6                    # every move: 60 percent for 1.6 s, about 19 cm on paper
INFLATE = 9.0
START = (50.0, 40.0)
GOAL, GOAL_H = (155.0, 40.0), 180.0        # the far side of the wall, facing back down the mat
WALLS = [(95.0, 0.0, 10.0, 120.0)]
V_SEEN = SPEED / 100 * V_MAX               # the speed the model uses; car() replaces it with the one it measures
TURN = 100 / W_MAX                         # rot per degree a second of turn; car() learns the real figure as it drives

def free(x, y, margin=INFLATE):
    if x < margin or y < margin or x > 200 - margin or y > 200 - margin:
        return False
    return not any(ox - margin <= x <= ox + ow + margin and oy - margin <= y <= oy + oh + margin
                   for ox, oy, ow, oh in WALLS)

def arc(x, y, h, steer, seconds, n=8):
    """Where car(SPEED, steer, seconds) goes at V_SEEN on a circle of R_MIN, as n points along the way."""
    v = V_SEEN
    w = math.degrees(v / R_MIN) * steer
    pts = []
    for k in range(n):
        dt = seconds / n
        a, wr = math.radians(h), math.radians(w)
        if abs(wr) < 1e-9:
            C, S = math.cos(a) * dt, math.sin(a) * dt
        else:
            C = (math.sin(a + wr * dt) - math.sin(a)) / wr
            S = (math.cos(a) - math.cos(a + wr * dt)) / wr
        x, y, h = x + v * S, y + v * C, (h + w * dt) % 360
```

```
        pts.append((x, y, h))
    return pts

def here():
    px, py = position()
    return START[0] + px, START[1] + py, heading()

def car(speed, steer, seconds, dt=0.1):
    """Drive at speed, steer -1..1, and trim the turn every dt so the radius stays R_MIN at
the speed measured."""
    global V_SEEN, TURN
    speed, steer = max(-100, min(100, speed)), max(-1, min(1, steer))       # top speed and
full lock, as in U2.8
    v = V_SEEN * speed / SPEED                                              # the speed the
last move measured
    rot = TURN * math.degrees(v / R_MIN) * steer                            # the turn that
gives R_MIN at it
    for k in range(round(seconds / dt)):
        (x0, y0), h0 = position(), heading()
        drive(speed, 0, rot)
        wait(dt)
        (x1, y1), h1 = position(), heading()
        v = math.copysign(math.hypot(x1 - x0, y1 - y0) / dt, speed)
        w = ((h1 - h0 + 180) % 360 - 180) / dt
        if k >= 3:                                # the robot takes about 0.3 s to answer
a new command
            rot += 0.2 * (math.degrees(v / R_MIN) * steer - w) * TURN
            V_SEEN += 0.2 * (abs(v) - V_SEEN)
    if steer and seconds >= 4 * dt:               # keep what the trim learned for the
next move's first guess
        TURN += 0.5 * (rot / (math.degrees(V_SEEN / R_MIN) * steer) - TURN)

# write plan(): A* over the car's six moves

# then the loop: plan from here(), drive only the first move, print a line, and plan again
```

The hint students can ask for: Two jobs. First the search: work out what a state is, what its six neighbours are (the three steers, each for STEP_S and for half of it; use arc() and keep only moves whose points are all free()), what a move costs when moves differ in length, a heuristic that never overestimates, and what counts as arriving (within 8 cm of the goal and 15 degrees of 180). The plan() in the lesson has three moves and returns bare steers; yours needs to return each move's seconds too. Think about what happens when drift has put the robot inside the wall's margin, where no move from it is free. Then the loop: call plan() in the loop body, from here(), drive only the first move, print the plan line, and think about the two ways plan() can come back with no move to drive: already there, or no way from here. They need different endings.

A solution

```
from bugbot import *
import heapq, math
connect()

V_MAX, W_MAX, R_MIN = 20.0, 120.0, 25.0
SPEED, STEP_S = 60, 1.6                   # every move: 60 percent for 1.6 s, about 19 cm
on paper
INFLATE = 9.0
START = (50.0, 40.0)
GOAL, GOAL_H = (155.0, 40.0), 180.0       # the far side of the wall, facing back down the
```

```
mat
WALLS = [(95.0, 0.0, 10.0, 120.0)]
V_SEEN = SPEED / 100 * V_MAX               # the speed the model uses; car() replaces it
with the one it measures
TURN = 100 / W_MAX                         # rot per degree a second of turn; car() learns
the real figure as it drives

def free(x, y, margin=INFLATE):
    if x < margin or y < margin or x > 200 - margin or y > 200 - margin:
        return False
    return not any(ox - margin <= x <= ox + ow + margin and oy - margin <= y <= oy + oh +
margin
                   for ox, oy, ow, oh in WALLS)

def arc(x, y, h, steer, seconds, n=8):
    """Where car(SPEED, steer, seconds) goes at V_SEEN on a circle of R_MIN, as n points
along the way."""
    v = V_SEEN
    w = math.degrees(v / R_MIN) * steer
    pts = []
    for k in range(n):
        dt = seconds / n
        a, wr = math.radians(h), math.radians(w)
        if abs(wr) < 1e-9:
            C, S = math.cos(a) * dt, math.sin(a) * dt
        else:
            C = (math.sin(a + wr * dt) - math.sin(a)) / wr
            S = (math.cos(a) - math.cos(a + wr * dt)) / wr
        x, y, h = x + v * S, y + v * C, (h + w * dt) % 360
        pts.append((x, y, h))
    return pts

def here():
    px, py = position()
    return START[0] + px, START[1] + py, heading()

def car(speed, steer, seconds, dt=0.1):
    """Drive at speed, steer -1..1, and trim the turn every dt so the radius stays R_MIN at
the speed measured."""
    global V_SEEN, TURN
    speed, steer = max(-100, min(100, speed)), max(-1, min(1, steer))       # top speed and
full lock, as in U2.8
    v = V_SEEN * speed / SPEED                                              # the speed the
last move measured
    rot = TURN * math.degrees(v / R_MIN) * steer                            # the turn that
gives R_MIN at it
    for k in range(round(seconds / dt)):
        (x0, y0), h0 = position(), heading()
        drive(speed, 0, rot)
        wait(dt)
        (x1, y1), h1 = position(), heading()
        v = math.copysign(math.hypot(x1 - x0, y1 - y0) / dt, speed)
        w = ((h1 - h0 + 180) % 360 - 180) / dt
        if k >= 3:                                      # the robot takes about 0.3 s to answer
a new command
            rot += 0.2 * (math.degrees(v / R_MIN) * steer - w) * TURN
            V_SEEN += 0.2 * (abs(v) - V_SEEN)
    if steer and seconds >= 4 * dt:                     # keep what the trim learned for the
next move's first guess
```

```
        TURN += 0.5 * (rot / (math.degrees(V_SEEN / R_MIN) * steer) - TURN)

MOVES = [(steer, seconds) for steer in (0, -1, 1) for seconds in (STEP_S, STEP_S / 2)]
# straight and full lock each way, each for STEP_S and for half as long

def plan(sx, sy, sh):
    """A* over the car's six moves: the (steer, seconds) moves from (sx, sy, sh) to the
goal, [] if already there, None if no way."""
    def cell(x, y, h):
        return (round(x / 6), round(y / 6), round(h / 45) % 8)
    def cost_to_go(x, y):
        return max(0.0, math.hypot(GOAL[0] - x, GOAL[1] - y) - 8)      # the goal counts
from 8 cm away
    q = [(cost_to_go(sx, sy), 0.0, (sx, sy, sh), [])]
    seen = set()
    while q:
        f, g, (x, y, h), moves = heapq.heappop(q)
        if math.hypot(GOAL[0] - x, GOAL[1] - y) < 8 and abs((h - GOAL_H + 180) % 360 - 180)
< 15:
            return moves
        c = cell(x, y, h)
        if c in seen:
            continue
        seen.add(c)
        for steer, seconds in MOVES:
            pts = arc(x, y, h, steer, seconds)
            if all(free(px, py, INFLATE if moves else INFLATE / 2) for px, py, _ in pts):
                nx, ny, nh = pts[-1]
                step = V_SEEN * seconds * (1.0 if steer == 0 else 1.15)         # turning
costs a little more
                heapq.heappush(q, (g + step + cost_to_go(nx, ny), g + step, (nx, ny, nh),
moves + [(steer, seconds)]))
    return None

# receding horizon: plan from where the robot really is, drive only the first move, and
plan again
for i in range(60):
    moves = plan(*here())
    if moves is None:
        print("no path from here")
        break
    if not moves:
        break
    print("plan", i + 1, "has", len(moves), "moves")
    steer, seconds = moves[0]
    car(SPEED, steer, seconds)
stop()
x, y, h = here()
print("ended at", round(x), round(y), "facing", round(h), "after", i, "moves")
```

Any program that meets the task's checks is marked correct in the simulator; this is one way, not the only way.