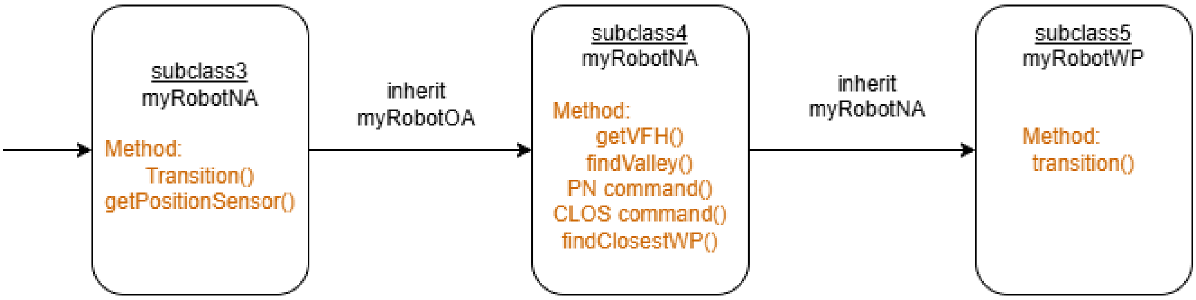
1132Advanced Programming in Robotic Navigation- final project

111323046 徐綺筑

Two new *.cpp* files are added to implement final project, those are:

1. myRobotNA inherits from myRobotOA
2. myRobotWP inherits from myRobotNA



I. Navigation-OA FSM

To combine obstacle avoidance and navigation, *free to go* state in OA needs to be replaced by navigation state transition. Five NAstates are used:

(1) free to go (use PN command).

(2) when close enough (use CLOS command).

(3) when target is reached (check remaining time/WP).

(4) if running out of time/all WPs have been visited (go home now).

(5) stop upon reaching home.

which is implemented in myRobotWP

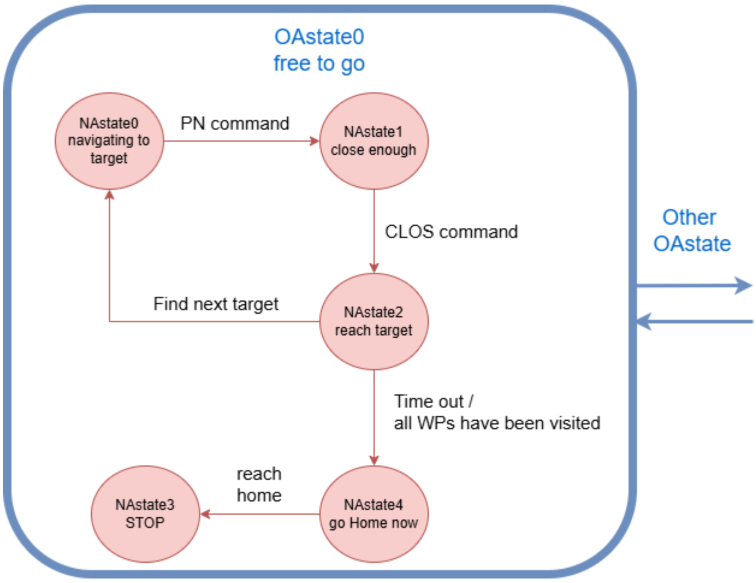


figure 1. Navigation state transition in OAstate 0

When the value from *getPositionSensor ()* exceeds the threshold, OAstate0 transitions to obstacle avoidance, and switches back to navigation when it escapes the trap.

II. VFH

VFH algorithm is implemented by

- *getVFH ()*: compute the histogram based on lidar reading.
- *getBearing ()*: this maps each sector to their angular direction.

VFH in navigation:

- *findValley ()*: identifies navigable valleys (continuous under-threshold sectors) in the VFH.
- *getBearHatfromValley ()*: uses the identified valleys to determine which direction ($\hat{\sigma}$) e-puck can move.
  - *(1)* Calls *getVFH ()* and *getBearing ()*.
  - (2) Reset *numValley ()* and using *findValley ()*.
  - (3) For each valley:
    - Loops over all sectors.
    - Check whether sector bearing is within the valley boundaries.
    - If yes, adds that sectors to *validsectors* and sums up

In navigation, which valley to choose:

The method used to calculate $\hat{\sigma}$ is non-deterministic and slightly differs from the original method. $\hat{\sigma}$ only samples from a Roulette-wheel that contains the VFH density of valleys.

$$H_{valley}{}^{*}(\sigma_k) = \frac{H_{valley}(\sigma_k)}{\sum_{i=0}^{N-1} H_{valley}(\sigma_k)} \qquad P_{valley}(\sigma_k) = \frac{1}{N-1}\left[1 - H_{valley}{}^{*}(\sigma_k)\right]$$

III. PN, CLOS navigation

This module controls a robot's navigation using GPS, compass and VFH data

$\nabla\sigma = \hat{\sigma} - \sigma_T$

- PN + CLOS bias: combines heading deviation and rate of target direction change.

- CLOS + PN bias: align the heading directly with the path to the target, biased by a selected $\hat{\sigma}$ (calculated from *getBearHatFromValley ()*) and PN.

Which is implemented using:

- *getHeading ()*: reads the compass and calculates the robot's heading
- *dLamT ()*: calculate the rate of change of the target bearing angle (used in PN)
- *PNcommand()*: primary navigation controller with PN as the base, incorporation CLOS direction bias ($\nabla\sigma$) via VFH:
  (1) If VFHDensity shows no obstacles, use pure PN
  (2) Otherwise, add CLOS correction
- *CLOS command ()*: primary navigation controller with CLOS as the base, incorporation PN bias via $\dot{\lambda_T}$:
  (1) Read compass, GPS data
  (2) Calculate heading from *getHeading ()*
  (3) Use *getBearHatfromValley ()* to find $\hat{\sigma}$
  (4) Updating control $\dot{\psi}_c = C\nabla\sigma + N'\dot{\lambda_T}$
  (5) If returning home, set target to (0, 0)

IV. Receiver and WP choosing policy

This component handles receiving waypoint from supervisor and implements a nearest waypoint selection strategy based on the distance between e-puck and waypoints

- *arraydata ()*: parses the packet from supervisor and stores the waypoint in the *waypoints* vector
  (1) check if the receiver has a packet
  (2) retrieve the data and extract *numWP*
  (3) store parsed waypoints in *vector<vector<double>>waypoints*
  (4) print each waypoint to the console foe debugging
- *findClosestWaypoint ()*: track the minimum distance target and its corresponding index.