Duckietown 2019: Project City Rescue

Carl Biagosch, Shengjie Hu, Martin Ziran Xu

I. Mission and Scope

A. Motivation

As stated on the homepage Duckietown is "a playful way to learn robotics". However, as with every learning initiative, solutions aren't perfect yet and there are cases where duckiebots crash or generally behave in an unexpected manner and human intervention is required to bring back order into the roads of duckietown. For every demo and every edition of AIDO, it is necessary that humans monitor the city the whole time and manually bring duckiebots back onto the road, when something unexpected happens. This is a tedious and cumbersome process, which could be avoided if there was a system that would automatically track every vehicle and could remotely control them in case of an emergency. This is where our project City-Rescue has its place in the duckietown universe, and our mission is simply to detect duckiebots in distress and get them back on track.

B. Existing solution

Currently there is no such solution, and the result of our project is the first effort of such kind. However, there already existed a watchtower localization system which runs on several computers in the robotariums and gives an estimation of the positions and headings of all the bots on the map. We build our rescue system on top of the output of this localization system.

C. Opportunity

Whilst the localization system works very accurately in offline mode, it lacks a way of giving an online output in real time, since the optimization of all the inputs is a computationally heavy process, which results in a non-negligible lag. To overcome this problem we build our own light-weight localization system on top of the infrastructure of the existing system, which allows us to track vehicles in real time (although not as accurately as the other system). This is described in further detail later, when documenting the technical specifications of our whole system.

II. Definition of the problem

The goal of the project is to localize duckiebots in real-time with a watchtower localization system and detect whether the bot is in a distress situation. Furthermore, if a duckiebot is detected to be in distress, the final objective is to get it back into a state where lane following works, such that normal operation can be continued.

A. Definition of distress

A situation in which a Duckiebot currently cannot resume normal operation autonomously. There are different situations where this might be the case:

- Duckiebots that are out of lane: Outside of the road, opposite lane
- Duckiebots that have crashed: Collisions with other duckiebots, collisions with the environment
- Duckiebots that just stopped for no apparent reason
- Duckiebots that are out of sight
- Duckiebots that are physically unable to move

For those cases involving zero velocity (i.e. bots have stopped) to be classified as "distress" we define that the duckiebot has been in the state for at least 10 seconds.

B. Scope of the project

For the localization of duckiebots, only duckiebots in sight and on a tile (boundary of map) are taken into account. For getting the duckiebost back on track, the duckiebots have to be fully functional, we furthermore only consider situations where there exists a valid path back onto the lane. Also out of scope are scenarios where the duckiebot detects itself that it is in a distress situation and maneuvers involving other bots rescuing a distressed bot.

C. Assumptions

We assume that the following conditions are fulfilled to build our system upon:

- Existing localization hardware: Watchtowers (camera + RPi) and traffic lights (additional LEDs).
- Duckiebot locations can be calculated using April Tags, which are placed on top of the vehicles
- There is an existing set of city rules that we can build upon (definition of the "right" side to drive, forward/backward operation, tile placement)

D. Performance metrics

We define the metrics to measure the performance of our solution with as follows:

- Number of duckiebots that can be monitored and rescued at the same time
- Different types of distress situations that can be handled (i.e. out of lane, crashed, etc.)
- Percentage of distress situations which can be handled completely successfully from detection until successful continuation of lane following
- Time of the whole rescue operation from detection until successful continuation of lane following

III. Contribution and Added Functionality

In this section, a detailed description of our rescue system is provided.

A. Software Architecture

For a successful rescue system, the following subproblems need to be solved:

1) **Localization**: Where are my duckiebots?
2) **Map representation**: Where should I be? Where do I want to be?
3) **Distress Classification**: How do I know, if I should rescue?
4) **Rescue Operation**: What commands should I send to the duckiebot?

While the next sections will deeply explain of how each of the subproblems has been solved, this section will mainly focus on the software architecture, we have used to implement our rescue system.
In our approach, we decided to shift most our calculations to a server, which is constantly monitoring, classifying and, if a distressed vehicle is detected, issuing rescue commands to the duckiebots in the city. In order to communicate between server and duckiebot an acquisition bridge has been set up[1] (see Figure 1).

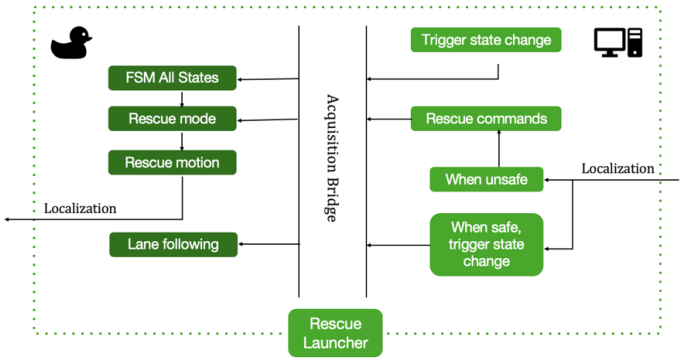

Fig. 1. Software Architecture: this figure shows the interaction between server and duckiebot side, when running our rescue algorithm

The heart of our rescue operation is the *RESCUE_CENTER*[2] node, which runs on the server side. It monitors all duckiebots visible by the localization system and classifies their distress based on the heading, position and FSM state of the duckiebot. Whenever a new duckiebot appears, it also launches a new *RESCUE_AGENT*[3] node on the server side. Every rescue agent is only responsible for one duckiebot and remains in idle state during normal operation.

[1]https://github.com/carlbi/rescue-acquisition-bridge/tree/6fac8e4eb4d0aa68ede1ea05c1e18f9c529d113a

[2]https://github.com/jasonhu5/duckie-rescue-center/blob/v1/packages/rescue_center/src/rescue_center_node.py

[3]https://github.com/jasonhu5/duckie-rescue-center/blob/v1/packages/rescue_center/src/rescue_agent_node.py

If a distressed duckiebot is detected by the the *RESCUE_CENTER* (see subsection III-D), the corresponding *RESCUE_AGENT* is activated and takes care of the closed loop rescue by sending specific car commands to the distressed duckiebot (see subsection III-E). In addition, a trigger is sent to the *FSM* node of the distressed duckiebot, which then transitions to *LANE_RECOVERY*. In that state the *CAR_CMD_SWITCH* node listens to car commands coming from the *RESCUE_AGENT*. After finishing the rescue operation, the *RESCUE_AGENT* notifies the *RESCUE_CENTER*, which then terminates the rescue operation by triggering a state change to *LANE_FOLLOWING*.

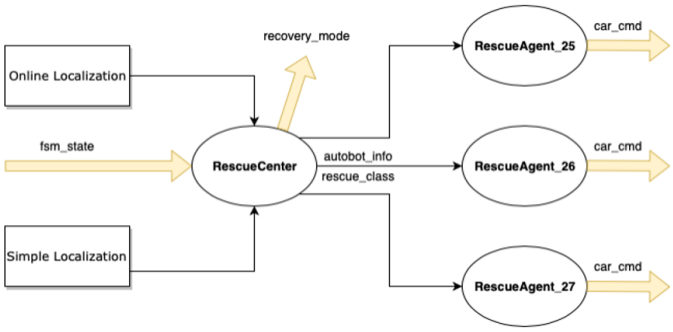

Fig. 2. Software Architecture (server side): this figure shows the nodes running on the server side

In order for the rescue system to work, *INDEFINITE LANE FOLLOWING* (including an added state *LANE_RECOVERY* in the YAML file of the FSM node)[4] and our modified acquisition bridge need to run on the duckiebots. Other than that, no additional software packages need to be build on the duckiebot. Furthermore, our decentralized approach using several rescue agents allows for multiple duckiebots being rescued at the same time.

B. Localization

While very accurate for offline localization, the cslam system did not suit our requirements in online localization. Albeit our efforts in tuning parameters, we still ended up with a lag of approximately 2 seconds, which would have made closed loop rescue infeasible. We therefore decided to create our own localization (called: simple localization[5]), which directly subscribes to the apriltag processors of the watchtower localization system and skips the computationally and time demanding graph optimization. From that, a lag-free position estimate can be used for closed-loop control. As a downside, simple localization does not combine images from several watchtowers, which could lead to position offsets. To mitigate that, we included an offset calibration procedure, which compares the simple localization output with

[4]https://github.com/mistermzx/dt-core-cityrescue/tree/09c798178ae7d07345a67fc75d0a6d633c7977cb

[5]https://github.com/jasonhu5/simple-localization/tree/4aea9cf03f81a8fd334f16a9783ca2c687c169ba

the clsam localization output and adjusts for the offset. If the watchtowers are not moved during operations, the position measurements are pretty well aligned.
For testing purposes, we added the output of simple localization to the visualization pipeline (see Figure 3).

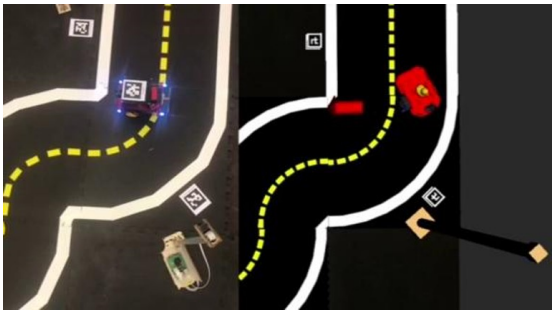

Fig. 3. Simple localization: the left screenshot shows the ground truth, while the right one shows the output of the simple localization (arrow) vs. the output of the cslam localization(car)

C. Map Representation

For distress classification and rescue operation, we need to know desired positions and headings of duckiebots based on their current positions in the city. For that purpose, we have created a SimpleMap Class [6], which reads city specification through a YAML-file. Figure 4 shows the map used in the AMOD-lab K31.

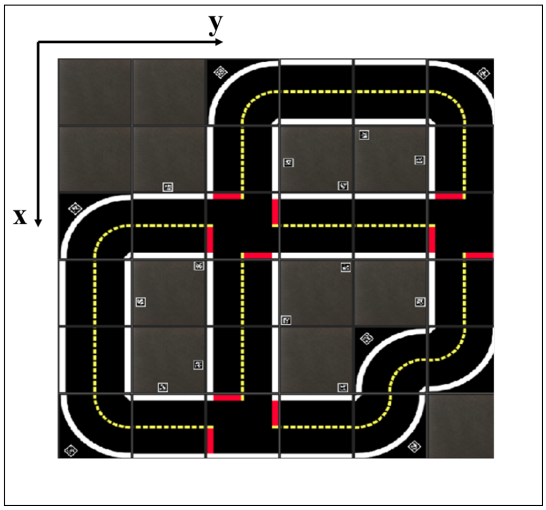

Fig. 4. Map Visualization: AMOD-K31

Based on the current position of a duckiebot, it provides information, whether the duckiebot is out of map and which desired position and heading it should have (*pos_to_ideal_position(self, position, heading=None), pos_to_ideal_heading(self, position)*). It distinguishes between different tile-types:

- **Straights**: The desired position is in the middle of the lane and only specified in one direction. For vertical straights e.g. (see Figure 4) the desired position is only a y-component. The desired heading is a multiple of $90°$ depending on which direction the duckiebot is facing
- **Curves**: The desired position is defined through a quarter arch, which is specified by the curve center and the radius pointing to the middle of the lane.
- **Asphalt**: The desired position is found through a grid search in reverse direction of the duckiebot up to a quarter of a tile length. The best coordinates are defined through a cost function penalizing distance and rotation. If no point in map is found within the search region, None is returned.
- **3-way/4-way intersections**: Intersections are modeled through curves, which attract the duckiebot back to a straight road. The desired position and heading is defined in the same way as on curves. (see Appendix VI-A for more information)

D. Distress Classification

The distress classifier is implemented in the AutobotInfo[7] class, which is called by the *RESCUE_CENTER* node, whenever a new position measurement arrives. It distinguishes between the following distress cases:

- **NORMAL_OPERATION**: everything as planned. RESCUE_CENTER node just keeps monitoring.
- **OUT_OF_LANE**: duckiebot is out of lane (based on SimpleMap)
- **WRONG_HEADING**: duckiebot deviates more than 90 degrees from the desired heading. This is only active, if the duckiebot is on a straight tile.
- **STUCK_AT_INTERSECTION**: duckiebot does not move for more than 30 seconds in intersection navigation.
- **STUCK_IN_INTERSECTION**: duckiebot does not move for more than 10 seconds standing in an intersection.
- **STUCK_GENERAL**: duckiebot does not move for more than 10 seconds outside of intersections.
- **DEBUG**: One can manually trigger a rescue operation by changing the rosparam (*/autobot[VEH_ID]/rescue/rescue_agent/trigger_rescue*) to True.

If a duckiebot is classified as distressed (not NORMAL_OPERATION), the RESCUE_CENTER triggers the rescue operation, which changes the FSM state of the duckiebot to LANE_RECOVERY and activates the corresponding RESCUE_AGENT. To guarantee safety, the duckiebot is stopped first, before applying any rescue commands.

E. Rescue Operation

During rescue operation, the RESCUE_CENTER keeps monitoring the distressed duckiebot and sends information

[6]https://github.com/jasonhu5/duckie-rescue-center/blob/v1/packages/rescue_center/src/simple_map.py

[7]https://github.com/jasonhu5/duckie-rescue-center/blob/v1/packages/rescue_center/src/autobot_info.py

to the RESCUE_AGENT. The RESCUE_AGENT uses the position measurement as an input to the closed-loop controller. At every time step it calculates a desired position and heading based on its current position on the map. It then uses it to as a reference to a PI controller, which calculates an input command to the distressed duckiebot.
We define the distance between desired and current position d and the heading ϕ as state variables $x = (d, \phi)$ and the forward velocity v and angular velociy ω as input variables $u = (v, \omega)$. In order to design a SISO PI controller, we artificially define our output to be:

$$y_k = c_1 \cdot d_k + c_2 \cdot \phi_k \tag{1}$$

c_1 and c_2 are weighting coefficients for penalizing the distance and heading deviations respectively. Similar the reference is defined as:

$$r_k = c_1 \cdot d_{ref,k} + c_2 \cdot \phi_{ref,k} \tag{2}$$

A PI controller with the coefficients k_P and k_I is then used to calculate the angular velocity:

$$\omega_k = k_p \cdot (r_k - y_k) + k_I \cdot \sum_{i=0}^{k} (r_i - y_i) \tag{3}$$

The forward velocity v_k is set to a constant negative value v_{ref}. This way, the duckiebot is guaranteed to go backwards during rescue and turn accordingly to the desired position and heading.
The implementation can be found in the StuckController class[8], which uses the following parameters:

TABLE I
CONTROLLER PARAMETERS

Parameter	Value
v_{ref}	-0.3
c_1	-5
c_2	1
k_P	6
k_I	0.1

Within the RESCUE_AGENT node the control commands u_k are sent to the duckiebot for execution. To adjust for possible delays in the localization system, we decided to hold a command for 0.3 seconds and stop the duckiebot for 1 second, before applying the next command. After applying 3 commands, the RESCUE_AGENT checks, if the duckiebot is ready for lane following. Here the position needs to lie within a quarter tile tolerance and the heading within a 30 degree tolerance to the desired values.
If this is given, the RESCUE_Agent sends a message to the RESCUE_CENTER, which then triggers a FSM state change to the duckiebot. Otherwise, the RESCUE_Agent will keep applying control commands and check again after 3 executions.

[8]https://github.com/jasonhu5/duckie-rescue-center/blob/v1/packages/rescue_center/src/stuck_controller.py

To make the rescue operation robust for different distressed situations, we needed to account for some special cases:

- If a duckiebot went into an intersection while rescuing, it will immediately stop and check, if it is ready for lane following. If not, it will drive forward a bit and reapply the control commands.
- If a duckiebot is stuck in an intersection, it will not go back to lane following, before it has reached a straight.
- If the SimpleMap class could not return a desired position (i.e. there is no proper position within the search radius of the duckiebot), the duckiebot will just go backwards and check again in the next time step.

IV. FORMAL PERFORMANCE EVALUATION AND RESULTS

Overall the distress classification and rescue operation highly depends on a working, real-time, offset-free localization system. Due to the fact, that there were not enough watchtowers to cover all areas of the AMOD-K31 lab, we were not able to formally test our rescue algorithm during normal operation, i.e. indefinite lane following.
However, for performance evaluation we manually created distressed cases using joystick-operation and observed the following results:

- Due to the system architecture, there is no theoretical upper limit as to how many duckiebots can be monitored and rescued simultaneously. Practical issues arise due to lacking communication between vehicles. We have tested the system successfully with three bots on the map.
- Out-of-lane and Stuck cases can be rescued very reliably on straights (9 out of 10). It even works reliably with heading deviations of approximately 180 degrees. The same distress case is more challenging on curves. Here 6 out of 10 duckiebots could be rescued in the desired position. For the 4 failed rescue operations, the duckiebot mostly turned too much and could not recover. This is due to the inaccuracy of the positioning system, since the reference headings in curves change quite significantly depending on which part of the curve the duckiebot is on. The same is true for the STUCK-AT-INTERSECTION case. Unfortunately we could not repeatably test our algorithm in intersections, because the watchtowers could not reliably monitor the whole intersection.
- Almost all successful rescue operations terminate within 2 rescue cycles, i.e. 6 issued commands. The rescue time is therefore less than 10 seconds. In some cases, the controller overshot and pushed the duckiebot into the other lane. The new desired heading required a 180 degree turn by the duckiebot, which took more than 2 rescue cycles. However, the duckiebot was able to continue lane following in the other direction.
- With the simple localization system out-of-lane cases can be detected within one duckiebot-length. This has been tested by manually driving out a duckiebot using joystick control. After one duckiebot-length the RESCUE_AGENT took over and stopped the duckiebot.

V. Future Avenues of Development

As described in the previous section, our rescue algorithm works quite well for rescuing duckiebots individually. Although we are able to rescue several duckiebot simultaneously by having separate RESCUE _AGENTS for each duckiebot, we run into problems, if duckiebots crash into each other and the rescue operation require coordination between them. This could be an interesting road to explore in future projects.
Furthermore, we propose to increase the number of watchtowers and properly fix them with respect to the map. This way, we ensure, that simple localization works reliably and distressed duckiebots can be reliably detected and saved at each position of the map.

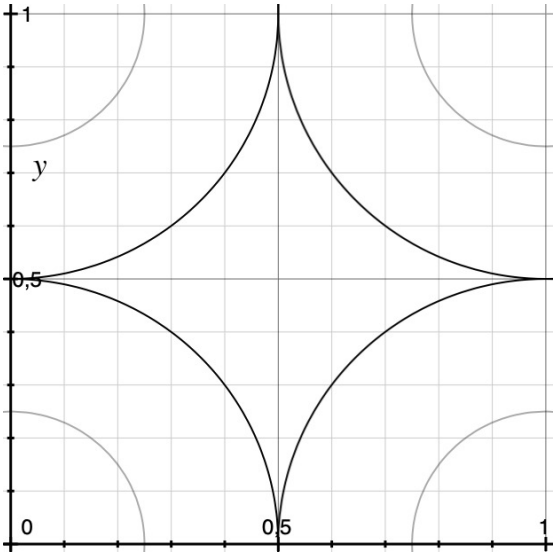

Fig. 6. 4-way intersection: the dark lines represent the curves and the light line the desired position/heading on the curves

VI. Appendix

A. Desired Position

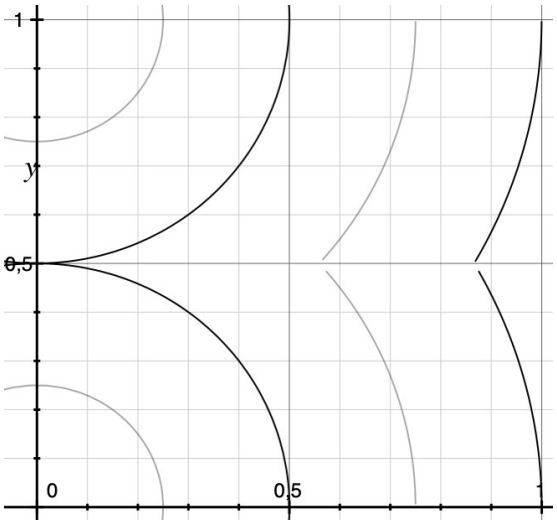

Fig. 5. 3-way intersection: the dark lines represent the curves and the light line the desired position/heading on the curves