Cognitive Robots – Final Project Report

Technion
Faculty of Data Science and Decisions

Tank with Water Gun - Planning for Forbidden Zone Protection

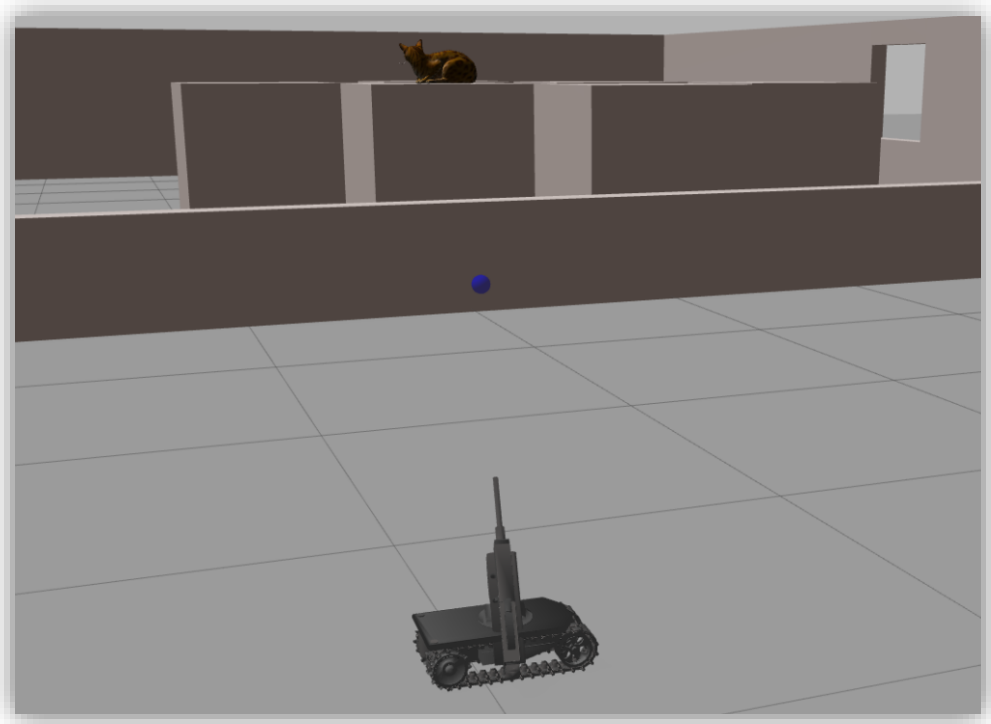
Authors
Daniel Moroz
Vladislav Klimenko

Lecturer and Supervisor
Prof. Erez Karpas

TA in Charge
Tal Kraicer

Project's GitHub page: https://github.com/VladKlimen/tank_robot

Simulation videos: fully reachable world, partially reachable world



Project idea [1]

One of us has several pet cats that enjoy playing in the backyard. However, street cats often sneak into the yard, posing a problem. They can attack our cats, mark their territory (with smelly marks…), and spread diseases. While we love all cats and try to help the street cats by providing food and medical care, their numbers are overwhelming. We want to make the backyard a safe place for our pets without hurting the street cats. Unfortunately, preventing their entries has proven difficult, as they always find another way, and sealing the entire area is too expensive. As a potential solution, we are considering using a robot guard equipped with a gentle yet annoying water gun to patrol the yard and deter the street cats, making the area less attractive to them.

In this project, we will propose a tank-like robot that operates in a 2D space and can turn in place (unlike an Ackerman steering vehicle). This robot is equipped with a water gun, that operates in a 3D space. The primary goal is to protect the "forbidden zone" - a backyard environment, from street cats. The robot must navigate around obstacles and aim accurately to ensure the safety of these zones, deterring street cats without harming them.

Note: the water gun can be replaced with any other non-lethal deterrent, like sound or light. This doesn't significantly affect the problem described.

Working environment

Tools and programs

- System: ROS noetic
- Simulator: Gazebo Classic 11
- Tank model and Gazebo worlds building: XML (URDF and SDF with Xacro)
- Ros working environment: VS Code with ROS extension
- Python scripts: Jupyter Lab and VS Code
- Visualization tools: pyvista and matplotlib (in Jupyter)

Tank robot modeling

We decided to make our own model from scratch, using URDF and Xacro for model description, and a mesh from SketchUp's 3D Warehouse. The reason: We didn't find a model that can drive and shoot water from its gun. Also, we thought it would be simpler to build it from scratch than to expand some ready model (like turtlebot3) to keep a firing gun.

Simulation modeling

We created a Xacro file from flexible SDF worlds creation. We used basic figures (boxes and cylinders) and their combinations, for simplicity. We used two simulation worlds – one with reachability of all free space on the map, and second with unreachable zones. These two were enough to check various scenarios.

---

[1] The report structure goes in the order of the project abstract structure and partially based on the abstract (see the project abstract in the appendix).

On top of these worlds, we spawned the robot and the goals. Goals and their place were explicitly set, but the spawning was in random groups of these goals.

For shooting simulation, we used a sphere that was spawned at the corner of the gun tip, with an initial velocity vector.

Robot control

We started with manual control of the tank and the gun. We used ROS diff_drive_controller plugin and a rospy script that basically is the thurtlebot3 controller script, with the gun control expansion.

Then we wrapped this wrapped with messages sending script to drive – drive, aim and shoot commands.

The problem

Definition

The problem involves motion planning and task planning. The naïve decoupled TAMP was sufficient for this problem, as there are no manipulations with objects in the environment. The main part of the problem is to find an (optimal) place to "shoot" the target from. The shooting involves collision checks, considering possible angles of the gun.

The brief definition of the problem is as follows (the key terms are underlined):

- We have a rectangle area with obstacles. There are several entrances into the area.
- Targets enter through these entrances at random time (each target is independent of others).
- The robot obtains positions of the targets once they enter the area.
- The goal is to "shoot" all the targets inside the area, by using the water gun to make their residence in the area uncomfortable.[2]

Assumptions

1. The robot knows the positions of cats and of itself.
2. The gun has two DOFs – the horizontal can rotate 360 degrees, while the vertical operates within a range of -30 to 45 degrees. The vehicle has also two DOFs – rotation and acceleration.
3. Known map, including 3D obstacles.
4. Cats randomly enter from several known "holes" from outside of the backyard area. The robot knowns cats positions after they enter the backyard. Cats continuously enter the backyard area, i.e. not necessarily all together.
5. Cats are static once they take some position in the backyard.
6. Objects heights are known, and consequently the cat's position on the object is known.
7. Battery and water for the gun are unlimited.
8. The water gun has ballistic behavior, with water starting velocity range.[3]
9. Flat floor (the robot drives in 2D).

[2] The minimization of the total time of the targets inside the area was replaced with just "reach all the goals".

[3] This was one of the extensions from the abstract, initially assumed the gun behaves like a laser (straight line).

10. Once a cat was “shot”, it escapes from the backyard.
11. The robot can’t shoot while moving.

Planning algorithm

We went with the idea of naïve decoupled TAMP:

1. First, find locations to shoot from on the targets.
2. Second, find an optimal (or suboptimal, for large number of goals) path to visit all those locations.
3. Third, find shortest paths on the grid-based map, for each edge on the above path. For our task we decided to use a variation of a grid-based planner (we also tried PRM and RRT, but they were slower and less precise than grid-based).

The plan consists of drive-aim-shoot sequences for each goal.

The algorithm[4]

1. Build a 3D grid matrix that represents the world.
   a. Build an empty grid matrix with some constant resolution $\boldsymbol{R}$. The resolution defines the cell size in the simulation world (for example, we used $R = 0.04m$ – i.e. grid cell dimensions are 0.04m each).
   b. Parse the world file and the goals models file and extract all the obstacles and the goals.
   c. Represent each element (obstacle, goal, floor etc.) with some unique code and fill the grid with these codes.
2. Calculate shooting trajectories for each goal.
   The idea is to find a legal shooting point (on the grid) from which the goal can be reached. The order of searching is from the farthest possible distance from the goal to the nearest, in circles. In more detail:
   For each goal, find points as follows:
   a. Calculate the maximum shooting distance $\boldsymbol{D}$ of the tank, given the maximum projectile starting velocity $\boldsymbol{v_{0_{max}}}$ and the gun pitch range.
   b. Consider circles around the goal, from the larger, with radius $\boldsymbol{r_{max} = D}$, to the smaller, predefined minimum radius $\boldsymbol{r_{min}}$. The number of the circles is such that the distance between them is equal to some constant distance $\boldsymbol{D_c}$.
   c. For each circle, consider points on it that are separated with predefined distance $\boldsymbol{D_p}$.[5]
   d. For each point on a circle, consider group of angles from the gun pitch range. The number of equally separated angles is some constant number $\boldsymbol{n_\theta}$.
   e. For each angle $\boldsymbol{\theta_{c_p}}$, find velocity $\boldsymbol{v_{0_\theta}}$ such that the resulting ballistic trajectory $\boldsymbol{t_{\theta v_0}}$ will be collision-free. Stop when first found such a trajectory.

---

[4] The final algorithm differs from that proposed in the project abstract.

[5] Thus, the number of points for each goal: $n = 2\pi \sum_{i=0}^{\left\lfloor \frac{r_{max}-r_{min}}{Dc} \right\rfloor} \left\lfloor \frac{D-D_c i}{D_p} \right\rfloor$.

- Collision and reachability check: move along the ballistic trajectory in the grid with constant step size t_b, until the goal reached or the collision with obstacle found (i.e. code of the obstacle in the grid cell). If there is no collision found, check if the point is reachable from the starting robot's position. If it is – the shooting point is found.

f. Once collision-free trajectories are found (or not found between the selected points and angles) for all goals, return the goal-velocity-angle tuples of them.

3. Find a driving plan.

   The main idea is to group goals into small groups of goals and solve the travelling salesman problem for each group, then build paths between the driving goals using the grid and the A* algorithm. The division to smaller groups is related to the exponential complexity of finding the optimal solution for the salesman problem. In detail:

   a. Divide goals into groups of size k (should be < 10).
      - In our approach, we tried to sort the goals before dividing them into groups, to heuristically achieve better results:
        - By distance from the robot's starting position.
        - By regions: divide the area to subareas and sort the areas by the distance of goals centroid of each area from the robot's starting position.

   b. For each group of goals, find the optimal plan (the goals order) to visit all of them once (i.e. solve the salesman problem).

   c. Combine all the subgroups to sequence of goals, preserving the inner optimal order and the original groups order.

   d. For each two adjacent goals in the sequence, find a shortest path between them using the grid and the A* algorithm (use the Euclidean distance heuristic).
      - We used a constant cost function, and it was fast enough for a 20X20m backyard and grid with resolution of 0.04m.

   e. Smooth paths. For each path:
      - Replace the stairs-like parts of the part with straight connection from the starting point of the part to the ending point.
      - Remove nodes of the part that lay on a straight line (vertical, horizontal or diagonal), keeping only the starting and the ending node (point) of the line.
        - The grid path consists of grid cell indexes connections, but we don't want all these middle points in the final plan, only the points that represent turn on the path in real world (aka simulation, not grid).

4. We have now a driving path and a shooting data for each goal. Combine the shooting goals and the driving goals and build the final plan, building the drive-aim-shoot sequence by the goals order in the driving plan.

Complexity vs precision trade-off

While some of the algorithm parameters are physically constrained (like projectile max starting velocity, gun pitch range), other parameters can be tuned for the trade-off between effectiveness and complexity.

These parameters are grid resolution (R), distance between circles for shooting point search (D_c), distance between potential shooting points on each circle (D_p), the number of equally-distanced

angles in the pitch range (n_θ), the ballistic trajectory collision check step size (t_b) and the goals groups size (k).

Tuning each of these parameters allowed us to reach sufficient precision of solution finding (i.e. get closer to completeness and optimality) while reducing the running time as much as possible.

Optimality and completeness

The algorithm is not optimal, even when all the above parameters $\rightarrow 0$. For each goal it takes the first legal shooting point it finds (it can be modified to check all possible legal points, but the optimality won't be feasible anyway in the meaning of running time).

Completeness is reachable when all parameters $\rightarrow 0$ (not within reasonable running time).

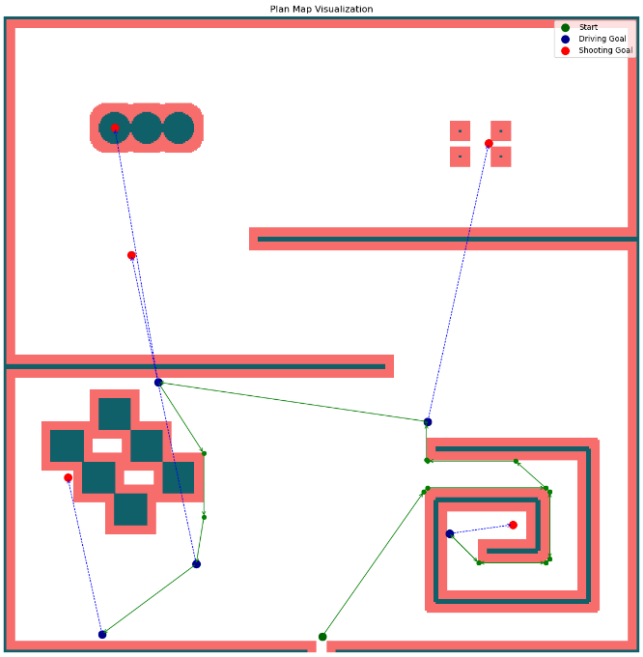
Algorithm parameters setting

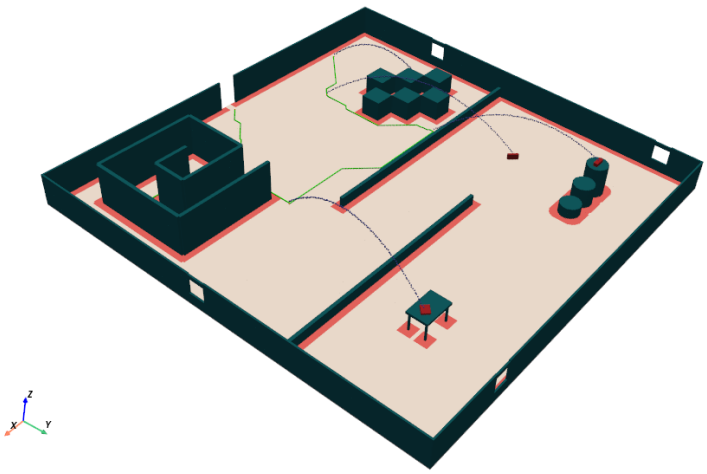
After continuous tuning and simulating, we have found (empirically) the next satisfying parameters setting: $R = 0.04m, D_c = 0.5m, D_p = 0.25m, n_\theta = 50, t_b = 0.02, k = 7$

Implementation process

Auxiliary visualization tools

Debugging process that involves 3D spaces is pretty much complicated, and visualization of the created grid and obstacles in it was necessary. We used both 2D and 3D visualizations with the help of matplotlib and pyvista python packages. This helped us quickly determine problems and bugs at the implementation stage. Below are examples of visualization.[6]





[6] Use examples for visualization are in the Jupyter notebook on GitHub

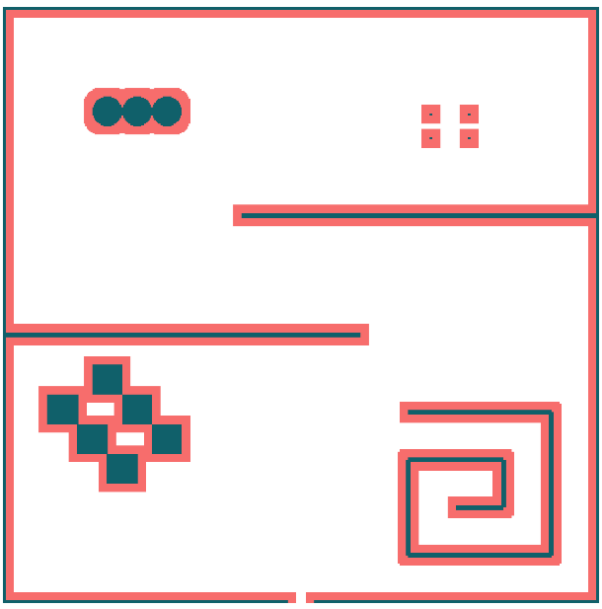
Challenges

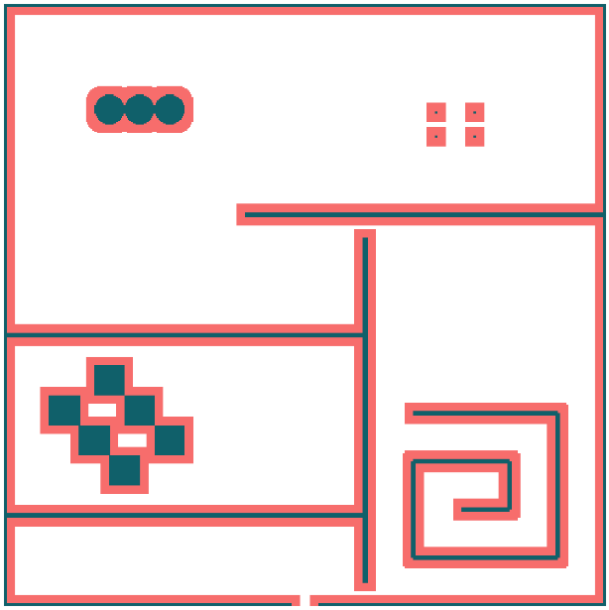
- A lot of challenges were met when working with gazebo simulator: Model building, model physics, collision detection problems, lack of information on the net. This took more time than the rest.
- Integrating lidar sensor: This was one of the extensions we wanted to implement, but we couldn't solve this problem with our model. The behavior of the sensor in Rviz was buggy and unpredictable. This problem is probably connected to poor odometry properties of the model, as the model was not created from the real robot and proper tuning of its physical properties was above our capabilities as data science students. Thus, we were forced to abandon the lidar integration and tuned the tank controller to behave accurately enough in the simulation without the lidar.
- Another kind of problem is dealing with 3D transformations we weren't familiar with. For example, the driving location had to be transformed from the shooting location, as it started from the gun tip corner, while the driving location is the center point of the robot body. This transformation involved finding the exact pose of the gun (including yaw and pitch) and transforming it into the tank body position.
- Tuning the algorithm parameters: this just consumed a significant amount of time. This involved setting the parameters combination, then running simulations to determine possible problems.

Results

Simulations

1. We used a 20X20m rectangle area for simulations, rounded by walls and filled with various obstacles (see the visualization above). There were two major maps – first without unreachable territories and the second (built on top of the first) included another wall that divided the territory to mutually unreachable parts (unreachable by driving), and left a tiny crossing from one region to another, for mutually reachable regions, as follows (green represents the ground obstacles, red – the buffer for collision avoidance):





2. On top of these maps, we randomly spawned predefined goals. I.e. the goals positions were predefined, but the spawning order was random. We tried to set hard-to-reach positions for the goals along with regular positions. The goals were spawned in groups (including groups of one goal).
3. The water shooting was represented as a sphere of radius 0.01m for the collision part (for visual part we made it bigger to be visible in the simulation videos).
4. The shooting tries were set to 3: if the target wasn't reached, the robot performs velocity recalculation (without changing the gun position) and tries again.
5. The collision-avoiding buffer was set to be the gun length from the tank center to the bun tip end. This ensured that the robot and the gun wouldn't collide with obstacles, without manipulations in the configuration space. This approximation was enough for our task and reduced unnecessary calculations complexity.

Simulation flow

Each goals group was spawned one after the other. The next group was spawned only when the robot achieved (or failed to achieve) the previous group. After all groups finished, robot tried once again (and only once) to reach the "failed" and "unreachable" goals.

The robot scores were recorded for each simulation. Each simulation contained 20 goals in total. The robot position for each simulation is where the previous simulation ended (except the first).

We ran several dozens of simulations to determine incorrect behavior and calculate scores. The scores involved numbers of achieved goals, goals that were marked as unreachable and goals that were marked as failed to achieve (the last one is more for the simulation assessment and doesn't relate to the algorithm itself). Below are the simulation scores after running 12-15 simulations for each of the 2 worlds (Table 1 – full reachable world, Table 2 – partially unreachable world):

simulation	reached	failed	unreachable	world_name	simulation_time	calculations_time
0	20	0	0	by_small	769,11	148,59
1	20	0	0	by_small	664,09	122,03
2	20	0	0	by_small	720,80	154,74
3	20	0	0	by_small	700,77	153,29
4	20	0	0	by_small	645,85	102,26
6	20	0	0	by_small	818,14	159,22
7	20	0	0	by_small	835,33	172,74
8	19	1	0	by_small	755,84	120,41
9	20	0	0	by_small	741,50	151,23
10	20	0	0	by_small	858,97	139,10
11	20	0	0	by_small	711,29	132,46
12	20	0	0	by_small	758,28	157,48
13	20	0	0	by_small	763,65	169,93
14	20	0	0	by_small	699,63	135,10
15	20	0	0	by_small	899,98	142,61
	299	**1**	**0**		**756,21**	**144,08**

Table 1

simulation	reached	failed	unreachable	world_name	simulation_time	calculations_time
0	20	0	0	by_small2	917,46	269,78
1	18	1	1	by_small2	914,87	212,55
2	19	1	0	by_small2	831,89	174,21
3	20	0	0	by_small2	899,12	227,42
4	20	0	0	by_small2	1155,56	264,62
5	19	0	1	by_small2	782,22	193,89
6	19	0	1	by_small2	911,16	339,80
7	19	0	1	by_small2	812,43	195,82
8	19	0	1	by_small2	880,47	196,13
9	20	0	0	by_small2	1029,57	254,13
10	20	0	0	by_small2	1317,40	468,49
11	20	0	0	by_small2	983,11	220,53
	233	**2**	**5**		**952,94**	**251,45**

Table 2

As we can see, the world with unreachable territories (Table 2) brings some problems when searching for shooting point for the goals within the unreachable region: more unreachable goals and higher mean calculation time, 251 vs 144 seconds for each 20 goals (12.5 vs 7.2 sec. per goal). That's because the search resolution doesn't always fit the problem (sometimes reducing the grid cell size, along with other related parameters needed, but then the calculations number may rise significantly). In addition, we set an upper bound for searching the point for each goal: if the robot found a collision-free shooting trajectory, but the point is not reachable for driving – then the robot retries, up to 10 times. Increasing the number of retries can also improve the search precision, but in cost of time.

Overall, above 99% of goals from Table 1 (299 of 300) and 97% from Table 2 (233 of 240) were reached.

Videos of some simulations: fully reachable world, partially reachable world.

Efficiency observations

Overall, all the goals were reached most of the time. The paths the robot chose were mostly not optimal: sometimes it drives far from the goal to shoot at it, even though it was close to the goal at first. This is because the trajectory searching order is not from the goal and farther, but from the tank initial position, and the distance metric doesn't consider the obstacles on the way at this stage. This is the price of fast plan calculation. This also is better than try to find optimal firing point, because theoretically, while the robot drives it can also "scare" the targets on the way, even without shooting. So, this is fine.

Conclusion

Most of the project goals were achieved. Some of the goals that were in the project abstract were changed on the way. The time minimization of goals being in the area was changed to "reaching all the goals", because the original goal didn't look feasible. But we tried to reduce the time.

We used A* for shortest paths and sub searching of optimal goals visiting order (up to 7 goals at the time), which reduced the goals reaching time. Overall, almost each simulation with 20 goals ended within up to 10 minutes and we found that acceptable, considering the robot's speed and the computer performance.

One extension was implemented (we weren't asked to implement extensions though): the ballistic motion of the water particles, instead of straight line.

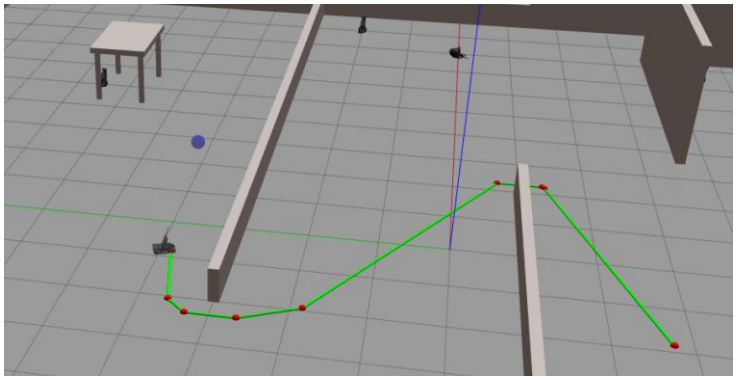
Future work

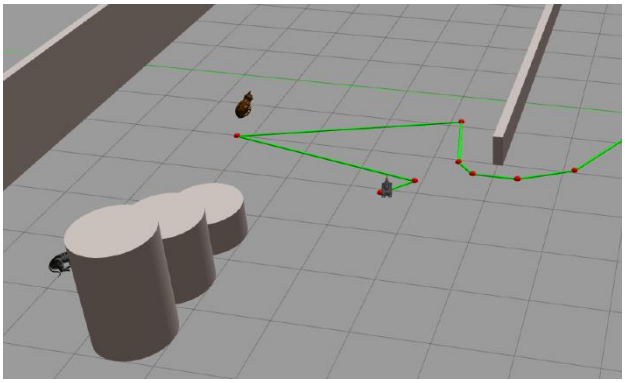
In addition to the proposed extensions, several ideas arose while working on the project:

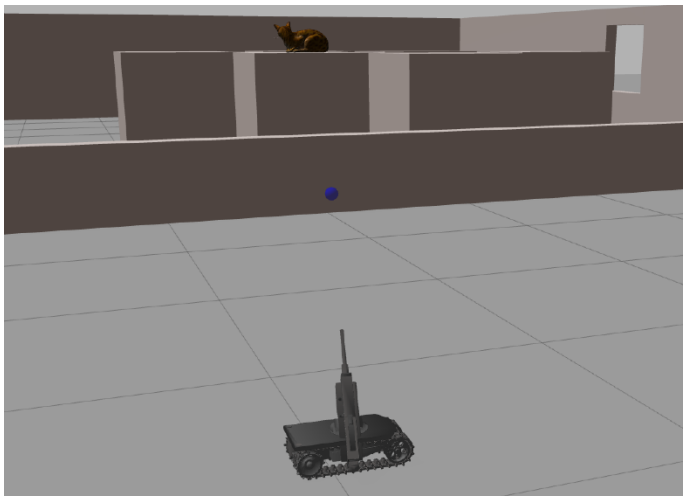
- Develop enhanced heuristics for shooting points search.
  - Since the first found shooting point is not always a good one, because a simple Euclidean metric doesn't consider obstacles on the way, more complex heuristic could improve the plan.
- Improve model physical properties or change the model to another, closer to the real-world model. Then, use simulated odometry instead of ground truth odometry obtained directly from gazebo model states information.

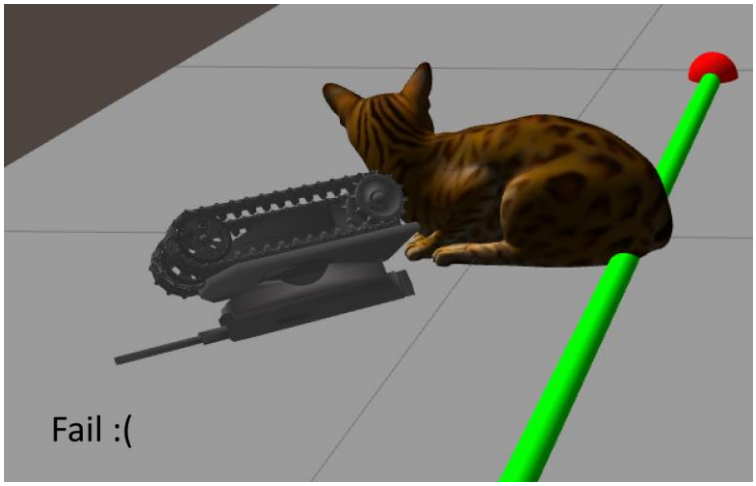
Appendix

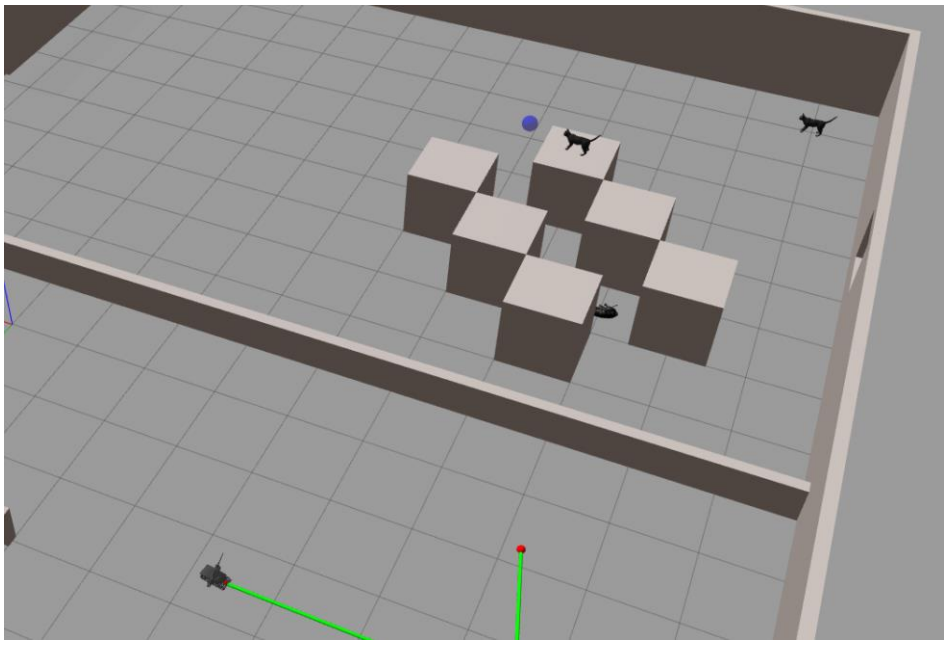
Simulations screenshots



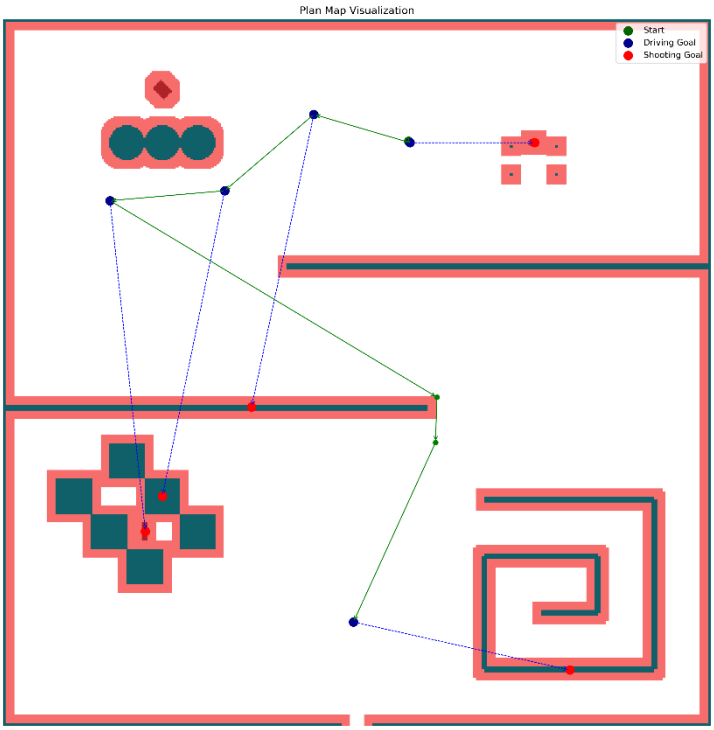


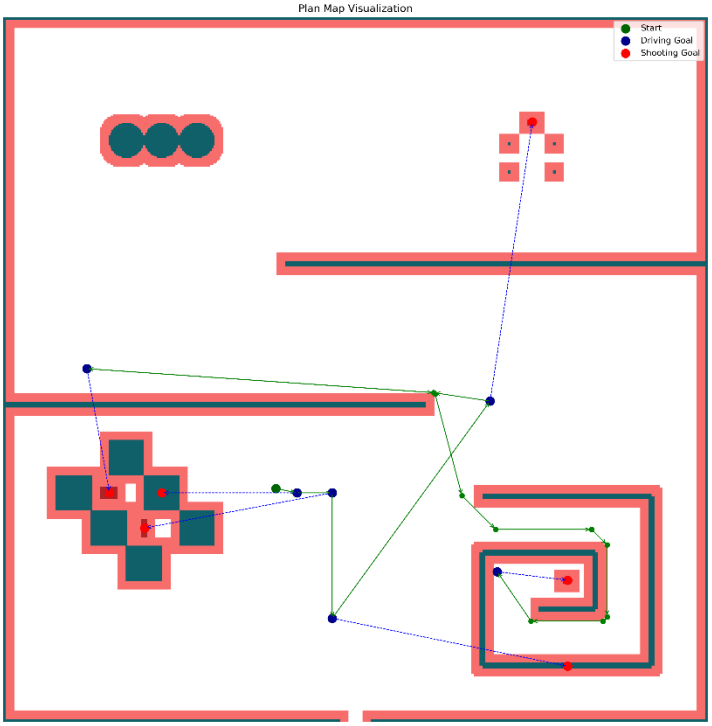


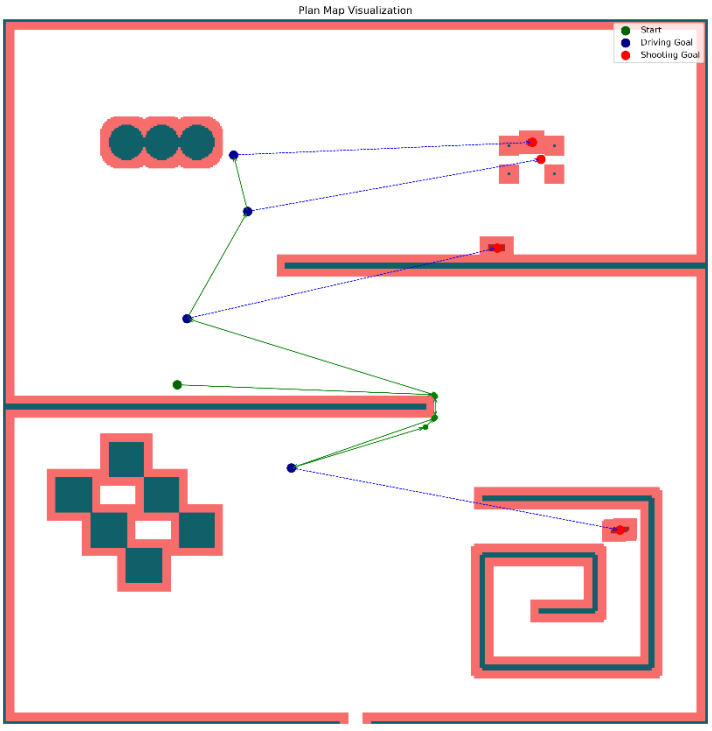


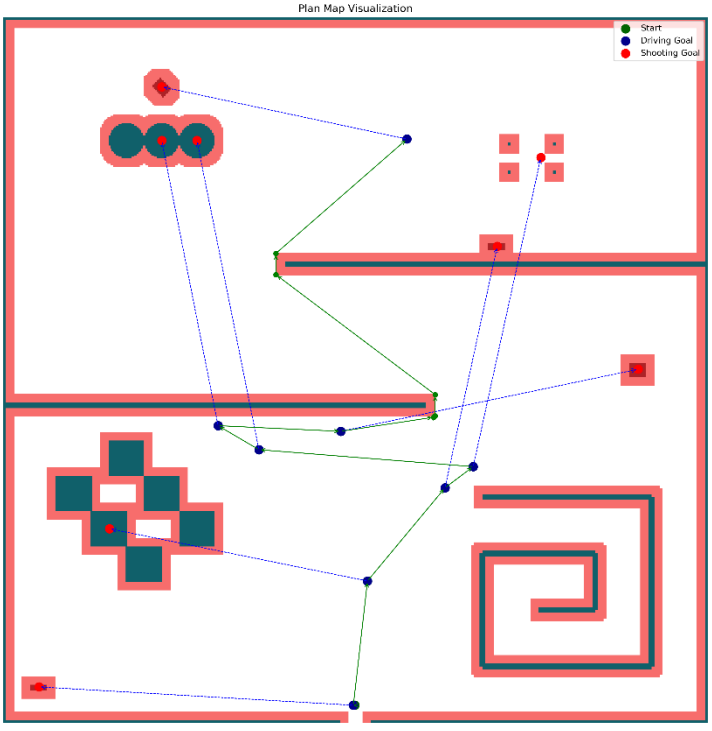


Visualized plan examples









Project abstract

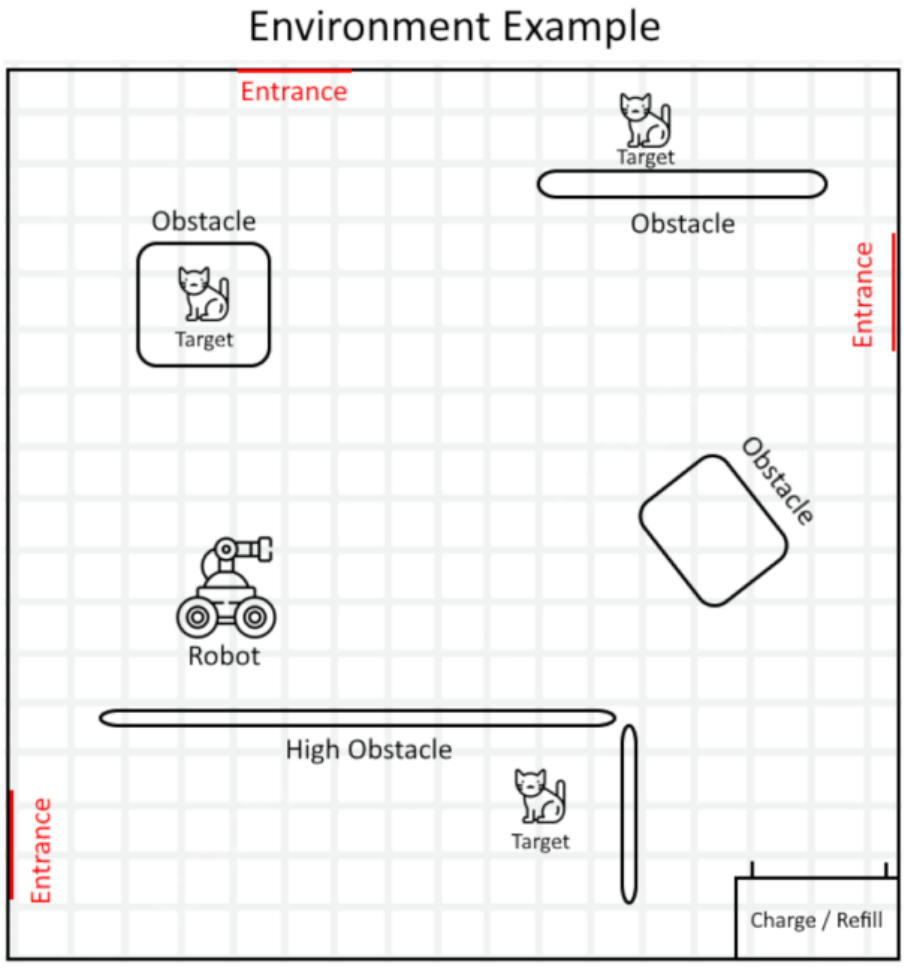
Project idea

One of us has several pet cats that enjoy playing in the backyard. However, street cats often sneak into the yard, posing a problem. They can attack our cats, mark their territory (with smelly marks…), and spread diseases. While we love all cats and try to help the street cats by providing food and medical care, their numbers are overwhelming. We want to make the backyard a safe place for our pets without hurting the street cats. Unfortunately, preventing their entry has proven difficult, as they always find another way in, and sealing the entire area is too expensive. As a potential solution, we are considering using a robot guard equipped with a gentle yet annoying water gun to patrol the yard and deter the street cats, making the area less attractive to them.

In this project, we will propose a car-like robot that operates in a 2D space and can turn in place (unlike an Ackerman steering vehicle). This robot will be equipped with a water gun on its roof, that operates in a 3D space. The primary goal is to protect the "forbidden zone" - a backyard environment, from randomly moving (yet theoretically predictable) street cats. The robot must navigate around obstacles, aim accurately, and manage its resources effectively to ensure the safety of these zones, deterring street cats without harming them.

Note: the water gun can be replaced to any other non-lethal deterrent, like sound or light. This doesn’t significantly affect the described problem.

Illustration of the sample backyard area



Working environment

We will use Gazebo inside Ros, because we are already familiar with the environment from tutorials. The robot we will use is TurtleBot3 with navigation node, or similar, with the water gun (or something similar) on top of it. We will create the water gun from scratch, using existing components and plugins in Gazebo.

The problem

1. Definition

   The problem involves motion planning and task planning. As we see it now, the naïve decoupled TAMP will be enough for this problem, as there are no manipulations with objects in the environment. The main part of the problem is to find an (optimal) place to "shoot" the target from. The shooting involves collision checks, considering possible angles of the gun.

   The brief definition of the problem is as follows (the key terms are underlined):

   - We have a rectangle area with obstacles. There are several entrances into the area.
   - Targets enter through these entrances at random time (each target is independent of others).
   - The robot obtains positions of the targets once they enter the area.
   - The goal is to minimize the total time of targets being inside the area, by using the water gun to make their residence in the area uncomfortable.
     - Define the goal via durative actions and minimize metric using PDDL.

2. Assumptions

   Below are assumptions for the base project. Possible extensions are mentioned inside the brackets, the color describes the expected complication of the extension (green – not complicated, yellow – complicated, red – highly complicated). These extensions bring the environment closer to the real world.

   12. The robot knows to detect cats.
   13. The gun has two DOFs – the horizontal can rotate 360 degrees, while the vertical operates within about 60 degrees. The vehicle has also two DOFs – rotation and acceleration (with constant speed).
   14. Known map, including 3D obstacles (extension 1: build a map using SLAM).
   15. Cats randomly enter from several known "holes" from outside of the backyard area. The robot knowns cats positions after they enter the backyard. Cats continuously enter the backyard area, i.e. not all together.
   16. Cats enter the backyard, find a place and then don't move (extension 2: cats are moving, most likely regions preferred by cats are known from past exploration).
   17. Objects heights are known, and consequently the cat's position on the object is known (extension 3: position of a cat on the object is not of constant height, for example a tree-like object).

18. Battery and water for the gun are unlimited (extension 4: need to refill water and recharge).
19. The water gun has a laser-like behavior with some maximum length (extension 5: the gun has ballistic behavior, with water starting velocity range).
20. Flat floor (extension 6: not flat floor, i.e. the vehicle also operates in 3D. Affects the gun position and adds uncrossable places on the map that basically are obstacles).
21. Once a cat was "shot", it escapes from the backyard (extension 7: the cat escapes with some probability, else changes its position).
22. The robot can't shoot while moving (extension 8: it can).

3. Algorithms

As the backyard is usually not a large area, for the vehicle we first will try the grid-based algorithm, with cell size of the larger dimension of the robot (will adjust the size at the implementation stage if needed).

For the gun – precompute for all ordered cell pairs:

- For each pair, the first cell is the source and the second is the target.
- Check collisions with obstacles on the way from source to target.
- If no collisions – calculate the vertical and horizontal angles of the gun to point to the target. Else - mark the cell pair as blocked.
- Store the calculated data in a matrix that represents the grid.

Problem with precomputing when implementing some of the extensions:

- Each time the environment map updates (new obstacle, obstacle moved, etc.) will need to recalculate the matrix. This can be optimized by updating only the relevant pairs.

Thus, the problem will be reduced to finding the closest cell the target is reachable from.

Another idea is to use RRT* in configuration space for the vehicle and the gun together. We will try it if the above algorithm rises unexpected problems.

To combine the task and the motion planning, Decoupled TAMP will be used as follows:

- Build a grid.
- Calculate the source-target matrix.
- Build a temporal task in PDDL based on current robot state and the targets state.
    - Each time new target appears – consider rebuilding the problem. Algorithm for such consideration will be constructed – it can be just rebuilding each time or only when the current goal is achieved (target was "shot") or heuristic approach or something smarter, like the Dynamic Task and Motion Planning (DTAMP) algorithms. We will consider several methods at the time of implementation.
      We also will consider the RRT* extension for dynamic planning, if necessary. The implementation roughly will be as follows:
        a. Tree Construction: Start the tree from the robot's current position.

b. Path Optimization: Continuously expand the tree towards the cats' positions, focusing on minimizing the time to reach them.
c. Path Execution: Once an optimal path is found, execute the first few steps and re-plan if necessary.
    - The task will use information from the source-target matrix to build the graph.
- When the vehicle arrives to the source cell, use the information from matrix to calibrate the gun and then "shoot". Attach this behavior to every source cell in the output plan.

Implementation ideas for extensions are listed below, in the extensions section.

Similar Works

We didn't find much of similar works that relate to our problem. Potentially related works we possibly will take ideas from:

- Integrating robotics into wildlife conservation: testing improvements to predator deterrents through movement
- Resource and Response Aware Path Planning for Long-term Autonomy of Ground Robots in Agriculture
- Other works with partial solutions on demand will be searched on implementation stage.

Measures

Robot dimensions, maximum robot speed, gun vertical degree of freedom angle range.

Simulations

Different numbers of cats (targets) and different targets arrival frequencies, different numbers of entrances to the area, different area size and number of obstacles, different obstacles forms. We will make 2-3 different areas and simulate different scenarios on them.

Project extensions

1. Build a map using SLAM (Simultaneous Localization and Mapping). (easy)
    - Use course tutorials to build a map by initial exploration of the area.
2. Cats are moving, with likely regions preferred by cats known from past exploration. (hard)
    - This will involve more complicated algorithms that may be far from the scope of the course.
3. The position of a cat on an object is not of constant height (e.g., a tree-like object). (easy)
    - This doesn't add much to the problem complexity, we just will track a cat position separately instead of assuming it is on the top of the obstacle (not one of the interesting extensions).

4. Need to refill water and recharge the robot. (medium)
    - We will use Mixed-Integer Linear Programming technique (MILP) to define the minimization problem with the refill and charge constraints. There are several optimizers for this kind of problem, we will start from Gurobi optimizer. We will consider other approaches if this fails.
5. The gun has ballistic behavior with water starting velocity and range. (medium)
    - The main problem is to check collisions when considering the ballistic behavior of the water: it will be possible to shoot over obstacles, expanding the shooting capabilities of the robot. This will involve reconsidering the grid-based approach and most probably change it to RTT*, as we will have to take into account the vertical angle of the gun and the shooting power at each place. The computation time for each source-target cell pairs will raise significantly and may not be the good idea, even when precomputing. Nevertheless, we will try both approaches, starting from grid-based, as the calculations for the gun will basically remain the same for both approaches.
6. The floor is not flat, meaning the vehicle also operates in 3D, affecting the gun position and adding uncrossable places on the map. (relatively easy)
    - For the real world it significantly adds complexity to the problem: need to stabilize the gun and consider acceleration of the robot, etc. For simulation-based environment, it may be relatively easy (as we think but not so sure), as the simulation can be simplified, and the problem can be reduced to just consider the vehicle floor position and adjust the gun angle appropriately.
7. Once a cat is "shot," it escapes from the backyard with some probability; otherwise, it changes its position. (hard)
    - This will involve additional modeling of cats behavior, but more significant is that the solution may go out of the course scope. Yet, this behavior is much closer to the real world, so we leave it as the possible extension to the project (maybe in the future or even now, if you really want us to suffer :().
8. The robot can shoot while moving. (medium)
    - The extension involves redefining the temporal task, so it considers concurrent moving and gun using. Also, the gun must be stabilized at each (discrete) point at the time of shooting (this is not an instant process, takes some delta time).

Note: some extensions may be (significantly) more complicated when combined with other extensions.