Abhi's NSF REU Report

Abhi Kamboj
May 2018 - August 2018

Robotic Networks Sensor Laboratory
University of Minnesota, Twin Cities

Table of Contents:

Summary

In this report, I explore various navigation and obstacle avoidance techniques for autonomous robots. The motivation is to determine an efficient and effective means for robot navigation in an agricultural setting. This report covers the various projects I worked on at the Robotic Sensor Networks Laboratory under Dr. Volkan Isler. This research work was supported by the National Science Foundation's Research Experience for Undergraduates (NSF REU) program. This work was also reported here: https://sites.google.com/umn.edu/rsn/projects/reu-project-abhi-kamboj, and the code can be found here: https://github.com/akamboj2/AutonomousNav-RL_Research2018.

Obstacle Avoidance Using Virtual Robotics Experimentation Platform (VREP)

In order to test the obstacle avoidance algorithms and get comfortable working with ROS and navigating a robot, VREP was used as a simulation. Virtual environments were created using the Pioneer robot as an agent and trees as obstacles. A 2D Laser scanner with a limited sensing range of 90 degrees was fixed onto the center of the Pioneer robot, as shown in Figure 1.

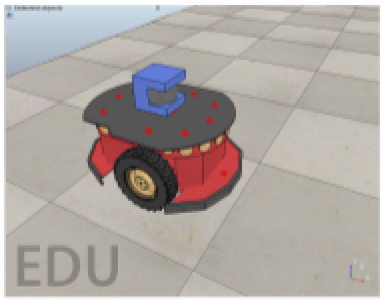
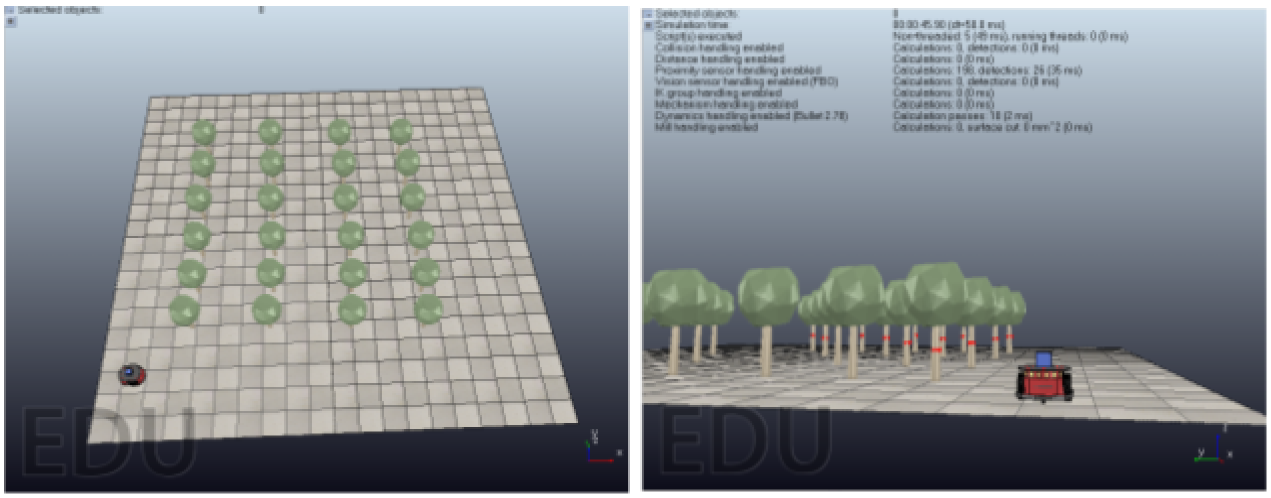


Figure 1: VREP Pioneer Robot with 2D Laser Scanner

Simple Waypoint Navigation

First simple waypoint navigation was achieved. Using the global position and orientation of the robot, the left and right wheel motors were controlled individually to keep the robot on track. Due to errors and uncertainties, often times after the robot moves the position it theoretically was supposed to be at is not where it is actually at, so each wheel has to constantly be updated according to the current pose of the robot and it's goal pose. By the end, the robot was successfully able to navigate through waypoints.



```
Selected objects:                          0
Simulation time                            00:00:45.90 (dt=50.0 ms)
Script(s) executed                         Non-threaded: 5 (45 ms), running threads: 0 (0 ms)
Collision handling enabled                 Calculations: 0, detections: 0 (0 ms)
Distance handling enabled                  Calculations: 0 (0 ms)
Proximity sensor handling enabled          Calculations: 196, detections: 26 (35 ms)
Vision sensor handling enabled (FBO)       Calculations: 0, detections: 0 (0 ms)
IK group handling enabled                  Calculations: 0 (0 ms)
Mechanism handling enabled                 Calculations: 0 (0 ms)
Dynamics handling enabled (Bullet 2.78)    Calculation passes: 10 (2 ms)
Mill handling enabled                      Calculations: 0, surface cut: 0 mm^2 (0 ms)
```

Figure 2: Orchard Environment
This environment was used to test simple waypoint navigation and obstacle avoidance.

Obstacle Avoidance and Navigation Random Velocity Vectors

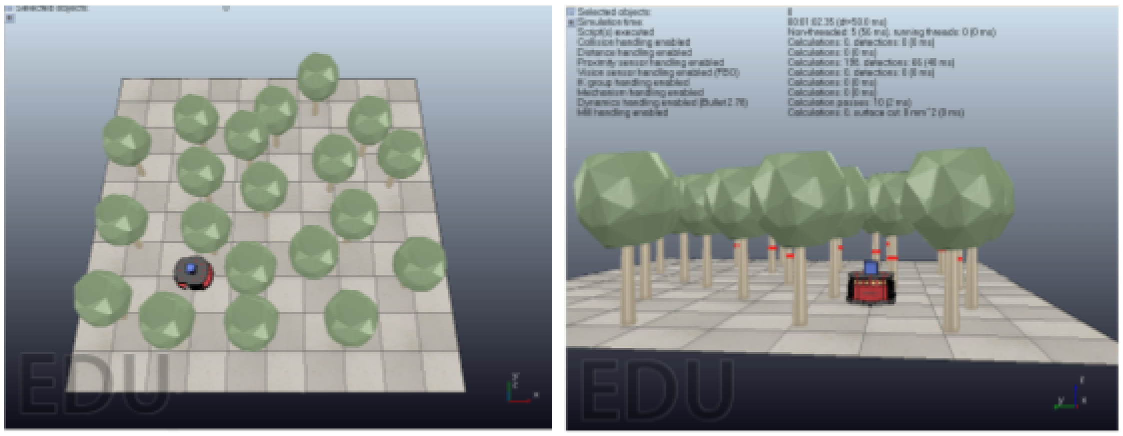
Next, obstacle avoidance was added onto the simple waypoint navigation program. This program generates 1 million random potential straight line trajectories for the robot to go to and chooses the best one. The criteria for choosing the best trajectory is the one that gets the robot closest to the goal and does not make it hit and objects. The robot does the following steps in this method:

1. It turns to face it's desired goal location

2. It scans the environment using it’s laser sensor
3. It generates 1 million potential trajectory
    a. Each trajectory includes a distance between 0 and the distance from the robot to it’s goal, and an angle between -pi/4 and pi/4 (the range of the sensor).
4. It chooses the trajectory that minimizes the Euclidean distance between the robot and it’s goal, and discards any trajectories that cause the robot to run into an object
5. The robot then moves to the end of its current trajectory*
6. Steps 1 through 4 repeat until the robot has reached its Waypoint.
7. Steps 1 through 5 repeat until the robot has navigated through all it’s waypoints

*note: whenever the robot is moving, the laser scanner is constantly checking the environment and if any object is less than 20 cm in front of the robot an emergency brake is applied and the robot backs up about 10 cm and jumps back to step 1 in order to recalculate a trajectory.

This method works very well when the robot is trying to navigate through various trees and obstacles that are smaller than it’s sensor’s ranges, however it fails if an object takes up its entire sensor’s range. For example, if a wall is placed in front of the robot, and it’s waypoint is set to the exact opposite side of the wall. The robot will repeatedly head straight for the wall as close as it can go and the emergency brakes will be applied. This occurs because the closest distance from the robot to its goal is to straight to the wall and any other trajectory that will make it go closer will cause it to run into the wall. This bug can be fixed to some extent by allowing the algorithm to generate potential trajectories outside of the laser scanners range, however this puts the robot at risk of generating a trajectory that makes it go straight into an obstacle that it can’t see, which occurs quite frequently in an environment such as the Random Trees environment depicted in Figure 2. A better solution is to have the robot remember what it previously saw and generate potential trajectories accordingly.



```
Selected objects:                          0
Simulation time:                           00:01:02.35 (dt=50.0 ms)
  Script(s) executed                       Non-threaded: 5 (56 ms), running threads: 0 (0 ms)
  Collision handling enabled               Calculations: 0, detections: 0 (0 ms)
  Distance handling enabled                Calculations: 0 (0 ms)
  Proximity sensor handling enabled        Calculations: 196, detections: 65 (48 ms)
  Vision sensor handling enabled (FBO)     Calculations: 0, detections: 0 (0 ms)
  IK group handling enabled                Calculations: 0 (0 ms)
  Mechanism handling enabled               Calculations: 0 (0 ms)
  Dynamics handling enabled (Bullet 2.78)  Calculation passes: 10 (2 ms)
  Mill handling enabled                    Calculations: 0, surface cut: 0 mm^2 (0 ms)
```

Figure 3: Random Trees Environment
This environment of randomly dispersed trees was used to test waypoint navigation with obstacle avoidance algorithm. In the second image notice that the tree directly to the left is not in the range of the sensor (sensing shown with red line), which could cause a potential crash if the best potential trajectory was generated in that direction. To minimize this error the majority of the velocity vectors were generated in the range of the sensor.

Obstacle Avoidance + Local Mapping

In order to allow the robot to better navigate around obstacles with it’s limited sensing range, the program was modified to remember the obstacles and generate potential next state vectors at any angle around the robot. This resulted in the robot creating a local map of the areas it has visited.

The main modification to the algorithm was allowing the agent to store it’s sensing information. Instead of using the sensed information just for one batch of a million potential trajectories, the algorithm uses the position and orientation of the robot to translate each sensed point from the reference frame of the robot to the reference frame of some fixed position on the vrep map and then stores that point in a global point cloud array. Then when the robot needs to generate it’s next set of potential trajectories it uses the global point cloud array which has accumulated all previously sensed information as well.

This algorithm allows the robot to navigate much more efficiently, without having to use the emergency brake as often. To visualize the robot’s ability to remember previous objects, a python script was written to plot the global point cloud array that was storing the sensed information. As the robot navigated through obstacles, the local map was constantly being updated and the end results are shown in Figure 4.

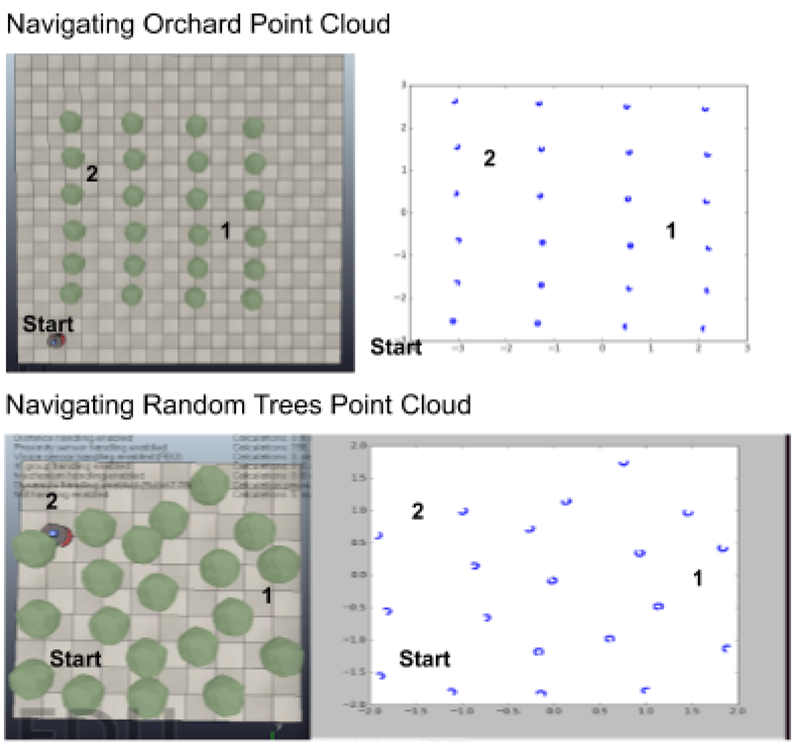


Figure 4: Orchard and Random Trees Local Maps
The robot navigates from start to waypoints 1 and 2 and the resulting map is depicted on the right. Notice how the tree trunks near the edge of the maps are incomplete circles. This would make sense because the robot never would have seen that part of the trunk.

Create Navigation Project

A Create 2 Robot and a Hokuyo URG Laser Scanner mounted atop the robot were used to create an application that allowed the robot to autonomously navigate through a mapped area. The many parts of this project were worked on individually and then combined into 2 launch files to run all at once. First, obstacle avoidance and navigation algorithms were created using the create_autonomy package as a controller. Next, the ros gmapping package was used with the Create navigation algorithms to create maps of indoor hallways. Then a graphical user interface was created using the python tkinter class. Lastly, a path planner that read a ROS Occupancy Grid message and outputted the shortest path was implemented using an A* algorithm in python. All of these pieces were put together and tested in a simple hallway environment.

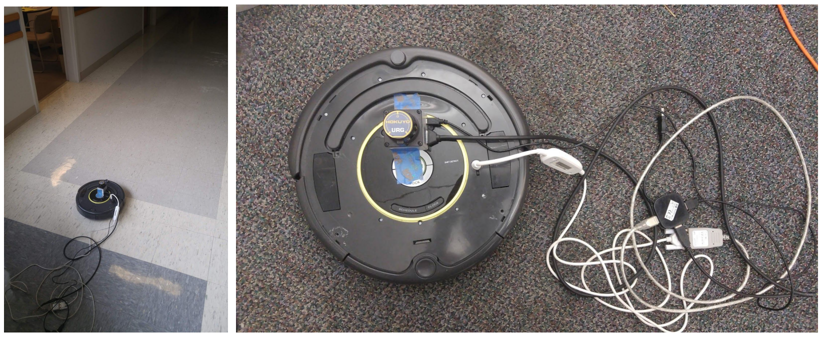


Figure 5: Create2 Robot and Hokuyo Urg Sensor Used in Project

Obstacle Avoidance

The obstacle avoidance and navigation algorithm that was tested in VREP with the pioneer robot (described above), was implemented with the iRobot Create 2 Robot. A Hokuyo URG 2D laser scanner with a range of 240 degrees was fixed on top of the robot. The original code was modified to fit the specifications of the sensor and create robot controls, and the odometry data was used to find the global position and orientation of the robot with respect to where it started.

Simple cases of avoiding obstacles while navigating through a hallway were tested. The robot never crashed however the emergency break did need to be applied a few times, and it does not always find the most optimal path. The robot obstacle avoidance algorithm wasn't designed for dynamic obstacles and performs poorly in such cases, often needing to apply the emergency break.

Mapping

Aside from obstacle avoidance the Create 2 Robot and Hokuyo sensor were used with the ROS gmapping package to create maps of different areas. While the ROS gmapping node was running the robot was manually navigated through a hallway. The resulting map image was then created with the Map Server ROS node. For small distances, the map was created relatively successfully, however there were loop closure problems for much longer routes.

After doing some research there are many problems that could be occurring with the robot, such as slipping, Odometry issues with create_autonomy or other noise. In fact, poor loop closure has been observed in some research publications such as Wang et al. Therefore, the rest of the project was tested with a map of a half hallway.

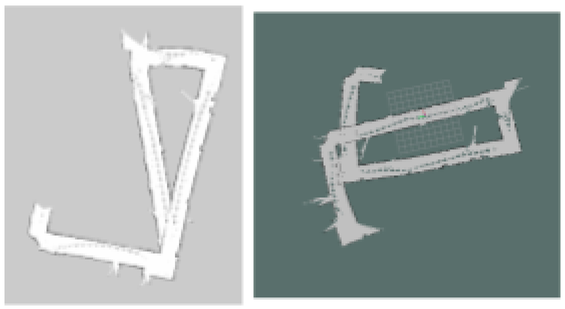


Figure 6: The following two images are maps of the hallways created by gmapping in Keller Hall, and show the loop closure problem.

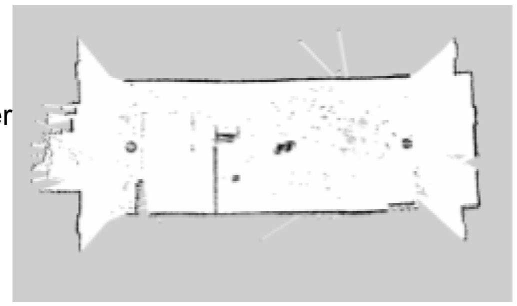
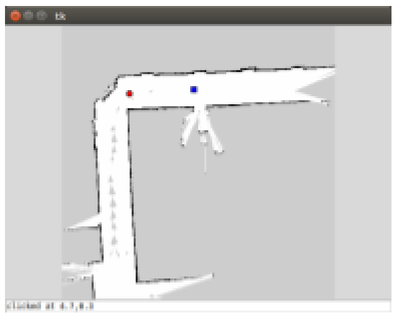


Figure 7: Gmapping was slightly more successful in the this open area

User Interface

A graphical user interface was created for the user to interact with the map. It loaded the pgm file and displayed the image on a Tkinter canvas. The user could then click on a point on the map, and the program responded by either creating that point as the destination of the robot or notifying the user that the chosen point is off the known mapped area.

This is done by running the map_server ROS node on the given map's yaml file. This node publishes a 4000x4000 occupancy grid, each value being 0 to 100 or -1, -1 indicating the area is unknown, 0 being there is definitely no obstacle and 100 means there definitely is an obstacle present. Each point on the grid translates to a pixel in the pgm file which translates to .005 m in the real setting (this resolution is specified in the yaml file). After truncating the occupancy grid array accordingly and reading it's values, an input pixel the user clicked on can be transformed to a destination in the real setting.



```
clicked at 4.7,8.3
```

Figure 8: Pre-Path Planner Pilot GUI

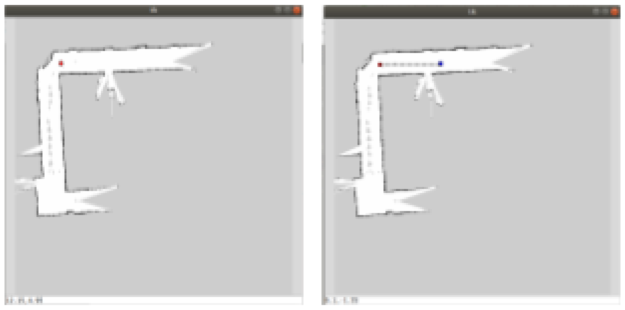


Figure 9: Example GUI
- Red circle is the robot location
- Blue Square is the clicked destination
- Black dots are the way points generated from the path planners resulting points

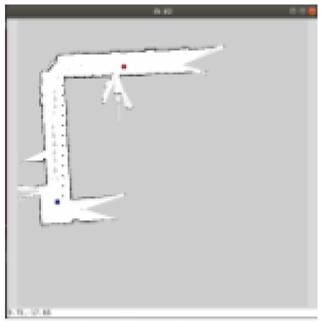


Figure 10 Necessity of Path Planner: For this situation, the path planner is needed to plan round the corner.

Path Planner

All the work done so far, allowed the Create navigate robot to accurately navigate in straight lines through the hallway environment, however if a point was entered around a corner, it resulted in the robot running straight into the wall. Clearly there was a need for some sort of path planner.

At initialization, the planner subscribes to the /map ROS topic and transforms the array in the OccupancyGrid ROS Navigation message to a graph. Each empty mapped cell in the grid (meaning it's value is 0) is a node and its neighbors are the nodes (if any) corresponding to the grid cells to the left, right, above, and below it. The node is a simple python class with x,y, and h (cost-to-goal heuristic) values.The nodes are stored in a 2D array in the order of the map to easily keep track of the nodes' neighbors.

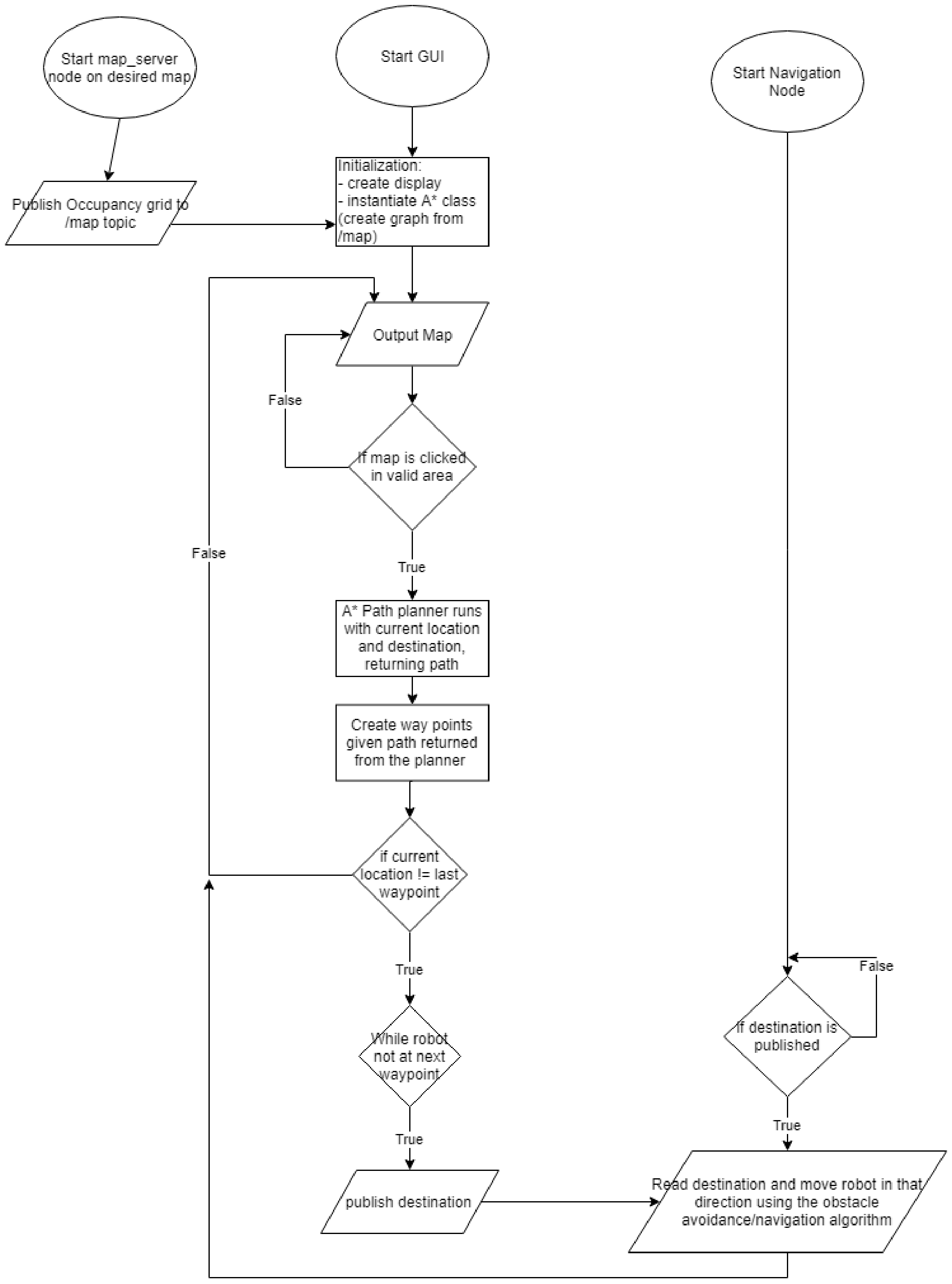
Once the Plan_Path function of the planner is called with an initial and final location given as parameters, an A* shortest path algorithm is run on the graph of nodes. The cost-to-goal heuristic h is the euclidean distance between each node and the destination. Once the path was planned, each point of the path was returned in an array.

The path planner worked well, however, it was very slow (about 6 minutes for a 10 meter hallway with a corner), presumably because it was running in python. In addition, a version of this function inputs the radius of the robot (assuming a circular robot) and remakes the graph with less nodes close to the unmapped/obstacle-present areas to prevent the robot from hitting the corner or wall when traveling through the shortest path. This theoretically worked, however it slowed the program down even more.

To combat the slow runtime, a resolution variable was created to scale down the graph. If the ROS map was an occupancy grid with a resolution of .005 meters per pixel and the path planner scale was 10, the graph for the A* algorithm would have a resolution of .05 meters per node, and there would be ten times less nodes than originally there would have been.

Finally, the User Interface was adjusted to read the path and output a certain amount of waypoints along the path for the robot to follow. The number of waypoints could be adjusted in the code. The interface showed the path as dots from the robot's current position to its destination.

Flowchart of Autonomous Navigation


Start map_server node on desired map
Publish Occupancy grid to /map topic
Start GUI
Initialization:
- create display
- instantiate A* class (create graph from /map)
Output Map
False
If map is clicked in valid area
True
A* Path planner runs with current location and destination, returning path
Create way points given path returned from the planner
False
if current location != last waypoint
True
While robot not at next waypoint
True
publish destination
Start Navigation Node
False
If destination is published
True
Read destination and move robot in that direction using the obstacle avoidance/navigation algorithm

VREP Reinforcement Learning

Navigation and control techniques are constantly evolving, and now more than ever artificial intelligence is being used to improve many basic navigation tasks. Much research has been done in the application of reinforcement learning to navigation and obstacle avoidance, and after implementing traditional methods of navigation, it only makes sense to work on and start learning some of the more advanced techniques using reinforcement learning.

GridWorld Q-Learning

In this project, a simple gridworld reinforcement problem was implemented in a VREP environment. The VREP Quadricopter was used as the agent with it's default controller. The agent began at x,y=0,0 and was tasked with traveling to the point 4,4, without colliding into any obstacles.. It could take steps of .25 meters in any of the positive or negative x or y directions. In addition, there were trees randomly scattered throughout the map blocking a direct path to the endpoint, however leaving enough way for the robot to discover a path.

In the reinforcement learning formulation of the problem, every x,y point in steps of .25 from (0,0) to (4,4) is a state, and possible actions for every state include moving to neighboring grid points if the neighbor x and y points are between 0 and 4. A small negative reward is given to the robot if it takes an action that will give it an x or y less than 0 or greater than 4, and the robot will stay in its current position instead of moving. This encourages the robot to explore within the grid. Larger negative rewards are given if the robot hits an object and the simulation is restarted. A large positive reward is given at the goal and smaller positive rewards are given at each step increasing exponentially as the distance to the final goal decreases.

To solve this reinforcement learning task with VREP, Lua scripts were written to remotely start, stop, reset, the simulation through ROS nodes, in addition to the usual quadcopter control and sensor output ROS nodes. A C++ node was written to implement the Q-learning algorithm with an epsilon-greedy policy on the environment. A 256x4 Qtable was used to learn the state-action values and was written to a text file every couple of episodes. This Q table was initialized with a bias towards actions in the positive x and y directions (such actions started with a value of .5, and the rest of the actions started with a value of zero.

The task was successfully completed after 2000 episodes of training with a max of 200 steps of training. The hyperparameters were epsilon=.2, learning rate= .2, and discount factor=.9. Below shows an gif of the agent performing a greedy policy on the same Qtable that was learned.

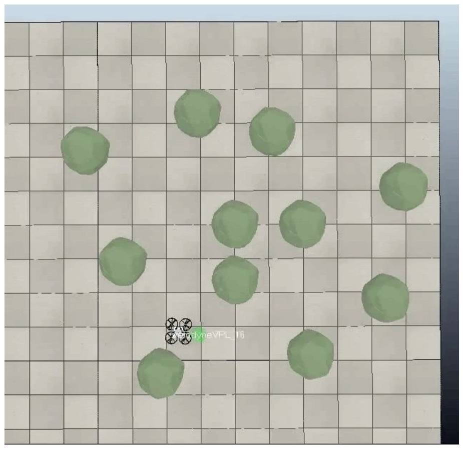


Figure 11 Gridwolrd Q-Learning: After 2000 episodes of training

Analysis and Alternative Implementation Idea

This project was a great way to get exposed to reinforcement learning in navigation, however it does not generalize well in practical application. The map has to be known ahead of time, and a model has to be given which can accurately represent the dynamics of the robot as well as the environment. Instead, implementing a Qlearning technique that maps sensor readings with goal-headings as states to motor control commands as actions, would be much more practical, yet more difficult to implement. This would generalize better to different environments because the states aren't encoded in the environment, but are instead encoded in online sensor readings. Hence, the robot is taking actions according to what it is seeing as opposed to what the environment is, which is something that most of the time cannot be known, especially in dynamic environments (environments with moving obstacles).

As a start to such an environment model-free implementation, a VREP environment was prepared with the fast_hokuyo (on vrep) sensor readings as states, and a pioneer robot's motor speeds being actions, however due to time constraints this reinforcement learning problem is left unfinished for now.

Acknowledgements

I would like to sincerely thank Dr. Volkan Isler for his support and mentorship. This summer was an invaluable research experience for me and my knowledge in robotics and artificial intelligence grew dramatically under Dr. Isler’s guidance. Furthermore, I would like to thank the National Science Foundation for this Undergraduate Research Experience grant. I was able to work with technology I would have never had access to otherwise, and I was able to work with experts in the field of robotics. Finally, I would like to thank everyone who was a part of the lab. Their support and guidance through numerous hurdles in my project allowed me to accomplish what I have this past summer.

Annotations and Literature Studies

In addition to the projects I was working on, I was also reading many research papers and studying much material. From textbooks, I referenced parts from Planning Algorithms by Lavelle, Probabilistic Robotics by Fox, Thrun and Burgard, and Principles of Robot Motion by Kantor, Choset, Lynch, et al. In addition, I thoroughly read Part 1 of Sutton and Barto's Introduction to Reinforcement Learning, taking notes and studying the examples. Informally, I also completed many tutorials and watched videos online. Finally, I wrote summaries of papers I found most interesting, and those are shown below.

Hindsight Experience Replay

Andrychowicz, M., Wolski, F., Ray, A., Schneider, J., Fong, R., Welinder, P., ... & Zaremba, W. (**2017**). Hindsight experience replay. In Advances in Neural Information Processing Systems(pp. 5048-5058).
https://arxiv.org/pdf/1707.01495.pdf

This research uses Hindsight Experience Replay (HER), to allow for efficient learning from sparse binary rewards. It shows that when HER is combined with Deep Q-Networks (DQN) or Deep Deterministic Policy Gradients (DDPG) (which is actor-critic, where actor's policy +noise is the behavioral policy, actor policy is updated with batch gradient descent on L=-expected value of (Q state, action behavior policy took), and critic is updated with the L=expected(difference between Q value and (rewards plus discounted Q(state, action behavior took))^2) so it's just like sarsa on policy TD). In order to do HER, they replayed a failure to reach a goal as a success in reaching another goal. This assumes there are multiple goals and that the initial state and goal are random. They used simulated (with MuJoCo physics engine) and tested it on a robot manipulator on 3 tasks, pushing, sliding and pick-and-place. In this case the states were angels, velocities, positions and rotations of all the robot joints. Goals were final positions of the object. Rewards were -1 (if not at goal) and 0 otherwise. Actions were 4 dimensional (3d space relative gripper position at next time step) and extra dimension for distance between fingers which control positions. Finally, the strategy of sampling goals for replay was generally the last step in episode was assumed to be the goal (called final) but could also be future (goal occurred after episode), episode (goal occurred sometime within episode), random (goal is random states throughout entire training procedure). HER with DDPG was demonstrated to perform best on all 3 tasks.

Personally, I think this is a cool discovery with relatively simple intuition, but difficult to apply/reproduce. If the manipulator is trying to move and object to a certain pose, instead of having it move to 50 million poses before finding one success, each time it moves it to a certain pose you could replay that experience saying, if that was the pose we wanted you would be correct, but we want something else. Then later on if you want the object at that pose (that it visited but was wrong at the time) it knows how to get there without having to retrain it.

Instead of using sparse binary rewards you could also just use a more complicated reward function that gives it a larger reward as the object gets closer to a desired position, but this is more complicated and probs requires much more computation and much more room for error. This paper mentioned this drawback. The paper didn't mention the storage necessary for their method, but I would think that is a drawback of their method. I would assume it takes a lot more storage space or computation power to augment the state space with a goal space as well.

Learning Deep Neural Network Policies with Continuous Memory Space

Zhang, M., McCarthy, Z., Finn, C., Levine, S., & Abbeel, P. (**2015**). Learning deep neural network policies with continuous memory states. arXiv preprint arXiv:1507.01273.
https://arxiv.org/pdf/1507.01273.pdf

In order to improve policy learning for control tasks using past observations, this research combines the state and action space with continuous valued memory states that can be read from or written to, and uses guided policy search to train a robotic manipulator. Guided policy search essentially has 2 phases. First it optimizes the trajectory by updating the value of the memory states to produce better actions in the future, after observing from the environment. Then it uses supervised learning to encourage the policy to use the memorized actions to produce those desirable states. Most recurrent neural networks hide past observations within the dynamics of the network. This can lead to vanishing and exploding gradients. The architecture proposed in this research called memory states includes the memory as part of the policy (augmented with the states and actions). This solution was tested against other networks that have tried to tackle the problem including Feedforward, LSTM, and Hybrid and proved to be the best one in 3 different tasks: sorting pegs, placing a bottle and plate in a cubby, and 2D navigation. It finished the tasks the most efficiently and coherently (less distance between end and goal as well as less sporadic behavior).

Agile Autonomous Driving using End-to-End Deep Imitation Learning

Pan, Y., Cheng, C. A., Saigol, K., Lee, K., Yan, X., Theodorou, E., & Boots, B. (**2017**). Agile Off-Road Autonomous Driving Using End-to-End Deep Imitation Learning. arXiv preprint arXiv:1709.07174.
https://arxiv.org/pdf/1709.07174.pdf

This research focuses on high speed, off-road, driving with low-cost on-board sensors. Due to a much more stochastic environment of being off road and potential damage of high speed crashes, the researchers use online imitation learning and batch imitation learning instead of typical online reinforcement learning or typical model-based approaches. A model predictive controller was learned through a DNN to model an expert controller that had many expensive sensors (accurate gps, IMUs etc) and lots of computational power. The learned controller only had a monocular camera and wheel speed sensors and was given the experts image and wheel speed data to train off of (supervised learning). The learned policy mapped the raw on-board sensor observations to steering and throttle commands. Two training methods were tested, online IL and batch IL. Batch IL performed worse due to accumulating errors and covariate shifts that occurred because the training images were collected executing different policies, so the neural network's inputs were of different distributions and because raw images were subject to different lighting conditions (testing occurred at different times of days). Despite these irregular policies online IL performed better.

*note: details of the dnn (like cnn, pooling, relu layers) are included in the paper but omitted from my summary

Learning to Plan for Visibility in Unknown Environments

Richter, C., & Roy, N. (**2016**, October). Learning to plan for visibility in navigation of unknown environments. In International Symposium on Experimental Robotics (pp. 387-398). Springer, Cham.
https://link.springer.com/chapter/10.1007/978-3-319-50115-4_34

This paper tries to increase visibility for robots when turning corners, the "naïve" way, ie the shortest path method. When this "baseline planner" follows the shortest path, it hugs the corner and turns tight but that actually limits visibility. Instead of distance, they try to minimize the cost of time by training an offset function to the shortest path heuristic that can correct the estimated cost to goal of the planner. They assume the planner has sensor information of the environment based on a limited range sensor and can take an action based on that info. They also assume a reasonable collision constraint on the action; the action must allow the robot to stop before running into an object or running into an unknown space (not seen by its sensor)àcauses the robot to slow down when turning because of decreased visibility, which causes time (cost to goal) to increase. To train function is the difference between optimal policy and shortest path heuristic (it's like a correction to the shortest path), they simulate the robot taking random actions around the corner and compute its shortest path + action cost minus the shortest path from where the robot is already before the corner. They encode some other predictive info as well about the frontier (line between what the robot can see and can't) and form a distribution, and then estimate f from that distribution. Path planner increases visibility but doesn't shorten time too much for laser sensors with a field of view greater than 80 degrees, because the benefit from having more visibility is offset by the fact that they had to travel a further distance, so it events out. Note: all the data and training were done in simulations, one run with an actual robot was done for demonstration.
My slides on it:
https://docs.google.com/presentation/d/1rDBHi6LO0jYG9sV_w7i9TTS7XBCnx0Frg9_5FMSo5ZA/edit?usp=sharing

From Perception to Decision: A Data-driven Approach to End-to-end Motion Planning for Autonomous Ground Robots

Pfeiffer, M., Schaeuble, M., Nieto, J., Siegwart, R., & Cadena, C. (**2017**, May). From perception to decision: A data-driven approach to end-to-end motion planning for autonomous ground robots. In 2017 IEEE International Conference on Robotics and Automation (ICRA) (pp. 1527-1533). IEEE.
https://arxiv.org/abs/1609.07910

In this paper a Turtlebot and Hokuyo 2D laser range finder is used to navigate an unknown environment of objects. The robot's steering commands are driven by CNNs that learn navigation strategies from an expert operator. In other words, someone or, in this research study's case, some machine that knows the entire environment gives optimal trajectories and the robot navigates itself and learns those trajectories. Then, when tested in a different environment without the expert navigator, the robot can navigate while avoiding obstacles. The robot does not know its global position, just the relative location of its destination. When an object is suddenly put in front of the robot it is able to swerve around it. However, the robot does get stuck at dead ends or convex objects. The main takeaway is that the experiments showed that a learned navigation model can transfer from training environments to unseen complex environments which is helpful because data generation for every possible scenario would be expensive.

However overall, it doesn't generalize well. It mostly just remembers things it has seen before, instead of learning. It wasn't effective in environments it hasn't seen.

Safe Trajectory Synthesis for Autonomous Driving in Unforeseen Environments

Kousik, S., Vaskov, S., Johnson-Roberson, M., & Vasudevan, R. (**2017**, October). Safe Trajectory Synthesis for Autonomous Driving in Unforeseen Environments. In ASME 2017 Dynamic Systems and Control Conference (pp. V001T44A005-V001T44A005). American Society of Mechanical Engineers.
https://arxiv.org/abs/1705.00091

This article focuses on the designing safe trajectories for autonomous vehicles. In most other methods such as RRTs MDPs, and Model Predictive controllers introduce uncertainty and overhead by discretization of the space. This paper introduces a method to compute a conservative Forward Reachable Set (FRS) in continuous space of all the places the car can go. Then sensors are used to find and eliminate which trajectories lead to a collision and the optimal plan from the remaining set is selected. The FRS is produced for a high fidelity (lots of noise interference) vehicle by modeling and tracking the trajectories of a low fidelity vehicle. This also helped model the uncertainty between mid-level (obstacle avoidance), and low level (brake throttle control) controllers. A computer simulation was testing this controller by navigating a car to a random goal around randomly generated obstacles between the car and goal for 1000 trials. There were 0 crashes, 82% success, 15% emergency breaks and 3% lost in the simulation iteration limit.

Personally, I thought it was cool that they try eliminating the error of discretized methods, however they assume everything is static (not moving) which makes it seem super inapplicable to detect cars and pedestrians (because those are not static in the real world).

Neural Network Memory Architectures for Autonomous Robot Navigation

Chen, S. W., Atanasov, N., Khan, A., Karydis, K., Lee, D. D., & Kumar, V. (**2017**). Neural Network Memory Architectures for Autonomous Robot Navigation. arXiv preprint arXiv:1705.08049.
https://arxiv.org/abs/1705.08049

This work focuses on the memory associated with navigating a robot through an environment directly using sensed information instead of map generation and path planning. They test several different deep neural network-based robot navigation systems and develop methods to train networks and quantify their abilities to generalize to unseen environments. More specifically, they examine navigation through cul-de-sac environments compared to parallel wall environments. They determined memory models in the neural network structure is necessary to achieve good performance by summarizing past information. The proposed method for estimating the VC dimension of the last network layer is a good indicator of generalization ability. Finally, they develop a new parallel training algorithm (asynchronous Dagger) for supervised learning of closed loop policies in sequential prediction problems.