DYNAMIC PATH PLANNING IN PARTIALLY KNOWN ENVIRONMENTS

A PREPRINT

Justin Kleiber
Department of Computer Engineering
University of Oklahoma
jkleiber@ou.edu

Preston Gray
Department of Computer Engineering
University of Oklahoma
preston.r.gray-1@ou.edu

Trey Sullivent
Department of Computer Science
University of Oklahoma
trey.sullivent@ou.edu

May 3, 2019

ABSTRACT

Efficient and effective path planning in a dynamic environment is an important endeavor for the future of robotics. As autonomous robots become globally used and increasingly versatile, the need for faster and more complex planning algorithms becomes apparent. This paper will focus on the application of the D*Lite algorithm in a static and partially known environment. D*Lite was implemented in tandem with a PID controller inside a layered architecture, and was able to quickly and consistently plan an efficient path between two nodes in a uniform grid map. The results show that D*Lite is a fast and effective planning algorithm that can be adapted to a multitude of industry applications.

1 Introduction

With autonomous robots becoming a staple application in many aspects of life, adaptive and higher level thought is essential for robotic systems to handle more complex tasks. However, the more complicated a deliberative algorithm becomes, the longer a mobile robot will sit idle while performing calculations. This has led to a high demand for robots with fast and effective path planning algorithms, even when they must operate within more complex scenarios.

Originally, such technology was developed to aid a tour guide robot. The tour guide robot in question is required to give a semi-dynamic tour of two engineering facilities on the University of Oklahoma campus. These facilities have people moving around and modifying the environment by moving furniture or adding temporary obstacles. The creation of a layered robot architecture by Simmons et. al on Xavier [1] was used as the foundation of this project. This work was then applied using state of the art technology in the form of Robot Operating System (ROS). The robot's software is capable of mapping and navigating through a partially known environment using a uniform grid and D*Lite [2], and can be adapted to different situations and configurations. A kinect sensor and TurtleBot2 were used to test the software; a similar test method using a different robot was taken by Cunha et. al[3] in their design for the RoboCup Home competition in 2011. The software design and adaptability to future work will apply well to the navigation and mapping portion of a warehouse robot tasked with moving and organizing packages.

2 Approach

2.1 Robot Architecture

In order to balance the deliberative nature of D*Lite with the reactive requirements caused by a dynamic environment, a hybrid architecture is proposed. In Figure 1, the specific architecture is described. This architecture is based on Xavier's hybrid paradigm [1], creating a layered system in which high level path planning can be overridden by collision detection and human control. Tasks are realized through keyboard input, which are then processed by the task planning node. This node creates a queue of commands and assigns them accordingly when a robot designates that it has completed all previous tasks. Once a task is received, the D*Lite algorithm is implemented in order to plan a path through the uniform grid map, passing the next cell that the robot should approach to the navigation node.

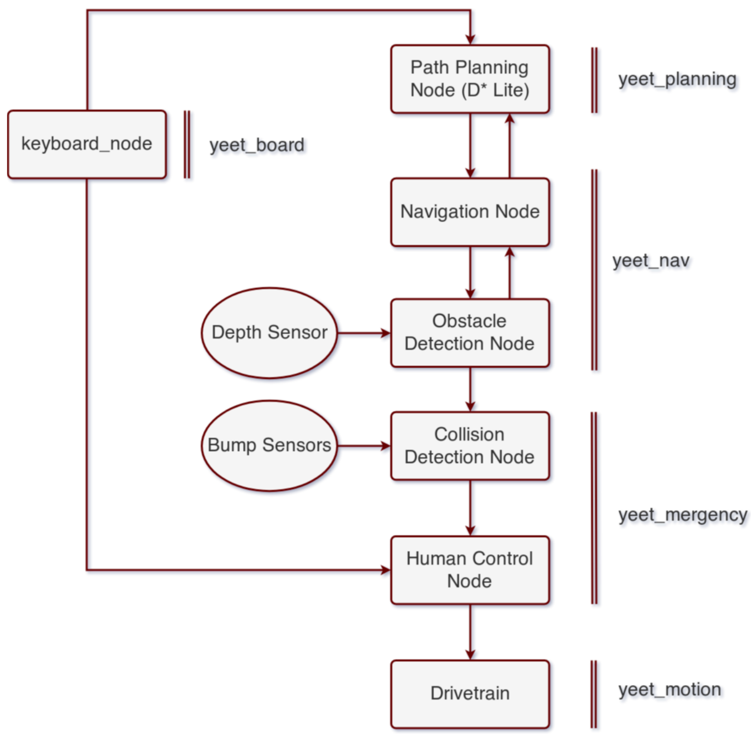

Figure 1: A hybrid robot software architecture is proposed to balance path planning and obstacle avoidance

Here, a PID Controller[4] is utilized to ensure precise movement to each cell of the uniform grid. The obstacle node shifts the architecture from a purely deliberative to a hybrid paradigm by halting movement in the navigation node if an obstacle in the target cell is detected as well as sending a signal to D*Lite to create a new path plan. At the lowest level, collisions detected by the bump sensors will stop any movement and request a new path from D*Lite or wait for human input to initiate an escape sequence.

2.2 Experiments

The software architecture in Figure 1 was implemented in ROS and was tested on the TurtleBot2 in a consistent environment with a variety of obstacle configurations.

Experiment 1 The first test took place in an obstacle free environment to test the robot's localization and motor control capabilities, as well as to test the ability of the D*Lite algorithm to plan the shortest path in a simple environment. Upon running the experiment, effective navigation was observed but a substantial amount of drift in occurred in the position of the robot due to accumulating error caused by dead reckoning. After rerunning the experiment, tweaking motor control constants, and getting consistent results, more complex obstacle configurations were considered.

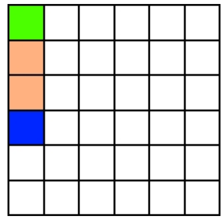

Figure 2: The map and planned path (orange) in an environment with no obstacles with the robot (green) and goal (blue) highlighted.

Experiment 2 The second test required the robot to plan a straight path with an unknown obstacle directly between the starting cell and the goal. This tested obstacle detection and the D*Lite algorithm's re-planning capability.

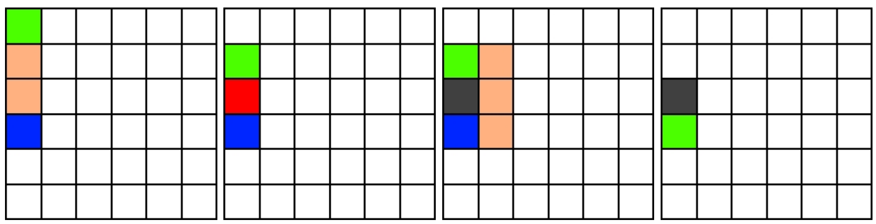

Figure 3: The robot (green) plans a path (orange) but discovers (red) an obstacle (black, afterwards). It then re-plans and travels to the goal (blue).

Experiment 3 The third test repurposed Experiment 2, maintaining the same environment while making the single obstacle's location known. This tested the initial planning capacity of the algorithm with obstacles.

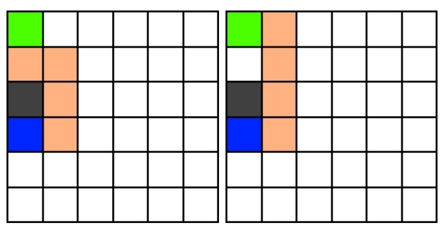

Figure 4: The robot consistently planned one of these two paths. Given multiple paths of equal distance, the robot will randomly choose one of them.

Experiments 4 and 5 These experiments had the robot traveling to multiple nodes in sequence to test the range limits created due to localization drift. The robot was tasked with navigating environments with 3-4 obstacles, known and unknown, and asked to travel through over 50 nodes along the planned paths.

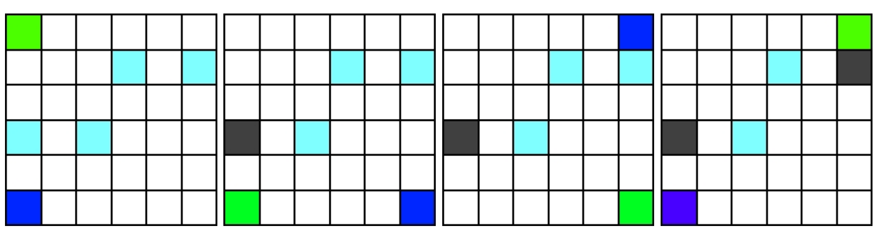

Figure 5: The robot traveled to 3 goals and discovered two obstacles, and is tasked with visiting a third.

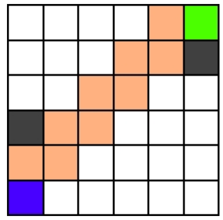

Figure 6: Due to using Manhattan distance, diagonal paths always have multiple ideal options. In this case, the robot has chosen this path (orange).

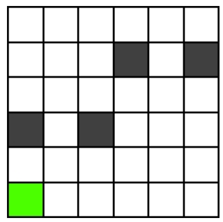

Figure 7: The robot has reached it's 4th goal and has traversed 29 nodes in this run of the experiment. All 4 obstacles were discovered which improves the speed of future path planning.

Experiments 6-12 The remaining experiments consisted of changing the number and arrangement of obstacles, along with changing which obstacles the code was aware of at start time. These tests had between 4 and 8 chairs with up to 3 known to the robot.

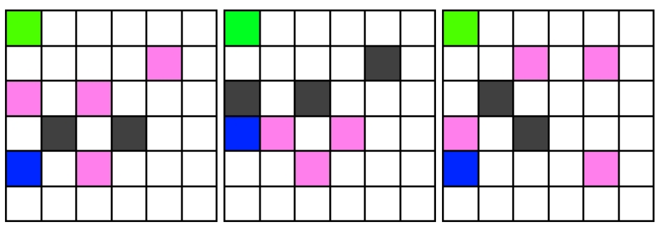

Figure 8: Examples of courses traversed by the robot populated with known (black) and unknown (pink) obstacles.

3 Results

The experiment results consistently show that the robot is capable of traversing a uniform grid environment using D*Lite regardless of obstacle orientation and the ratio of known to unknown obstacles. A sub-optimal path was occasionally chosen due to the D*Lite algorithm's bias towards navigating known cells over uncharted territory. This longer path resulted in significant time loss over the true optimal path, but in most applications this is offset by the faster re-planning calculation time over standard path planning algorithms as minimal cells of the gradient map are updated at a time.

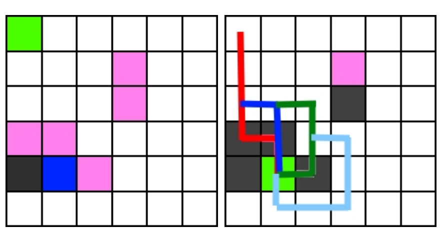

Figure 9: Here we can see the sequence of paths planned by the algorithm (red > blue > green > light blue). The robot will detect any obstacle directly in front of it and will replan when there is an obstacle in it's planned path.

The optimality of the path planning is largely dependent on the amount of the environment that is known. The robot may plan paths that are ultimately blocked by unknown obstacles, but it will always plan a near-optimal path around its known version of the uniform grid. Once the entire environment is known, either by sending the robot through the entire environment or by manually entering all obstacles, the robot was consistently able to plan and traverse the most efficient path. This result indicates that the planned paths will become closer to the optimal path as more obstacles are discovered.

The robot's ability to detect obstacles and add them to the map was successful overall. At times the Kinect sensor used for obstacle detection presented false negatives due to a larger minimum range than was suitable for the size of grid spaces, but by minimizing the localization drift through PID tuning, a consistent and appropriate distance from the edge of each obstacle was achieved.

4 Discussion

After conducting all of the experiments it was determined that this implementation of D*Lite was effective at planning optimal or nearly optimal paths in known, unknown, and semi-known environments. The main goal of experimentation was to ensure that the D*Lite algorithm was robust and consistent. The implementation was checked against solutions produced by hand, and proved that it was consistently producing the expected path. The tests also had a secondary goal, which was to improve the driving by fine tuning the other components of the software through trial and error, such as the PID, speed ramping, and obstacle detection. Errors were isolated and corrected in the code, and constants were tuned to make motion consistent. After a code change, a short sequence of tests were run in rapid succession to confirm that issues were truly resolved. By improving movement and localization, the algorithm was effectively evaluated.

The complexity dealt with in the tests presented was never able to break the algorithm, nor did any specific configuration take noticeably more time than another. This was likely due to a small number of nodes and a relatively slow robot, both of which cause the code execution to be one of the fastest components of the robot. However, given the design of the algorithm and arguments put forth in Sven Koenig's implementation of D*Lite [2], it is shown that this algorithm has performance advantages over other path planning algorithms.

5 Conclusions and Future Work

The D*Lite algorithm implemented in this robot has shown that it is highly effective at dynamically planning paths between nodes in a grid world. The ability to plan an adaptive path as new information is discovered is applicable to many types of systems, including tour robots, warehouse robots, or nearly any other type of automated agent in a mapped environment. For future work, expanding this project to a multi-robot system as originally intended would be relatively simple while vastly increasing the effectiveness of the algorithm in each robot. Specifically, the ROS architecture allows for grid spaces to be populated from any source by publishing to an appropriate topic. Robots within the system can publish occupied nodes in order to update the map information for each robot. This would result in more information being known about the obstacles within the map, increasing D*Lite's ability to plan the most optimal path. In order to make the current code more applicable to real life scenarios, the obstacle detection would have to be significantly upgraded in order to detect when previously known obstacles have moved and subsequently free up nodes in the uniform grid.

References

[1] Reid G. Simmons, Richard Goodwin, Karen Zita Haigh, Sven Koenig, Joseph O'Sullivan, and Manuela M. Veloso. Xavier: Experience with a layered robot architecture. *SIGART Bull.*, 8(1-4):22–33, December 1997.

[2] Sven Koenig and Maxim Likhachev. D*lite. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence and Fourteenth Conference on Innovative Applications of Artificial Intelligence, July 28 - August 1, 2002, Edmonton, Alberta, Canada.*, pages 476–483, 2002.

[3] João Cunha, Eurico Pedrosa, Cristóvão Cruz, António J. R. Neves, and Nuno Lau. Using a depth camera for indoor robot localization and navigation. In *In Robotics Science and Systems (RSS) RGB-D Workshop*, 2011.

[4] Justin Kleiber, Noah Zemlin, Ethan Vong, and Dorothy He. Sooner-competitive-robotics/robotlib.

Appendix

A Contributions

A.1 Justin Kleiber

A.1.1 Design

Justin developed and presented the variation on the layered architecture presented by Simmons. [1] This initial design served as the basis for the final project with minimal modifications. Justin also designed several of the messages used between the nodes.

A.1.2 Implementation

Justin implemented the whole of the D*Lite algorithm into a C++ ROS node. He also provided previous versions of the PID code written by himself for other team members to implement. Justin wrote a large portion of the navigation node, specifically the pieces that interface with the D*Lite node.

A.1.3 Testing

Justin was present for every test of the final robot, and tested all code he wrote as the project progressed.

A.1.4 Reporting

Justin assisted in editing the final report and made the final edits on every section. Justin provided direction for structure and content of figures, and heavily assisted in formatting the final document. Justin added all appendices after but not including "Contributions" and properly cited all material. Justin assumed much of the responsibility of presenting the poster due to his refined understanding of the D*Lite algorithm.

A.2 Trey Sullivent

A.2.1 Design

Trey assisted in modifying the paradigm to provide keyboard input at various levels of the architecture, as well as preventing deadlock from bump sensors. Trey also designed the "move" message.

A.2.2 Implementation

Trey adapted the Collision Detection and Drivetrain nodes from the previous two projects, and wrote the Human Control and Keyboard nodes.

A.2.3 Testing

Trey was present for nearly every test of the final robot, and tested all code he wrote as the project progressed.

A.2.4 Reporting

Trey designed and produced the draft poster and assisted in editing in changes for the final version. Trey assisted in writing initial drafts of the paper and was responsible for writing the rough versions of the Experiments, Results, Discussion, and Conclusions sections and created the figures used in the Experiments and Results sections.

A.3 Preston Gray

A.3.1 Design

Preston assisted in clearly defining the final architecture and provided various pieces of useful feedback during initial design meetings. He also designed the structure behind the navigation node.

A.3.2 Implementation

Preston wrote the entirety of the obstacle detection system and and adapted the P.I.D. system from Sooner Competitive Robotics [4] for this project. He also wrote much of the navigation node.

A.3.3 Testing

Preston was present for nearly every test of the final robot, and tested all code he wrote as the project progressed.

A.3.4 Reporting

Preston wrote the majority of the official draft of the project report, and continued to assist with adding content as the final document was created. Throughout the various internal drafts produced, Preston wrote most of the Abstract, Introduction, and Approach sections. Preston assisted with editing the final poster and adding the paragraph content not in the initial draft. He was also responsible for picking up the final poster from the print shop.

B Planning Package

B.1 D*Lite

D*Lite ROS Node - This node handled the majority of the path planning from a high level perspective. It also provided a service to the navigation node for getting the next driving instructions.

```
//The algorithm for D* lite can be found at the following link:
//https://www.cs.cmu.edu/~motionplanning/lecture/AppH-astar-dstar_howie.pdf

#include <ros/ros.h>
#include <memory>

//Constants
#include "yeet_msgs/Constants.h"

//Messages
#include "nav_msgs/OccupancyGrid.h"
#include "std_msgs/Empty.h"
#include "yeet_msgs/node.h"
#include "yeet_msgs/nav_status.h"
#include "yeet_msgs/TargetNode.h"

//Libraries
#include <eigen3/Eigen/Dense>

//D* and Mapping classes
#include "yeet_planning/yeet_priority_queue.h"
#include "yeet_planning/map_cell.h"
#include "yeet_planning/world_map.h"

//Constants
#define MAP_ROWS     yeet_msgs::Constants::MAP_ROWS
#define MAP_COLS     yeet_msgs::Constants::MAP_COLS
#define SQUARE_SIZE yeet_msgs::Constants::SQUARE_SIZE     //This is the size of the carpet squares in meters
#define MAX_BUFFER   10
#define VIEW_UP      0
#define VIEW_LEFT    1
#define VIEW_DOWN    2
#define VIEW_RIGHT   3
#define IDLE         0
#define NAVIGATING   1
#define WAYPOINT     2
```

```
#define OBS_REPLAN   3
#define NEW_GOAL     4


//Global map data
WorldMap current_map(MAP_ROWS, MAP_COLS, SQUARE_SIZE);

//Node management
std::shared_ptr<MapNode> goal_node;
std::shared_ptr<MapNode> start_node;

//Current node check for replanning
int direction;

//Search state management
int search_state;

//Publishers
ros::Publisher node_pub;
ros::Publisher enable_drive_pub;


//TODO: testing
void updateView(int cur_col, int cur_row, int goal_col, int goal_row)
{
    int col_diff;
    int row_diff;

    //Get the difference in rows and columns
    col_diff = cur_col - goal_col;
    row_diff = cur_row - goal_row;

    direction = VIEW_UP; //current_angle;
    //Down a square
    direction = (row_diff == 0 && col_diff == 1) ? VIEW_RIGHT : direction;
    //Right a sqaure
    direction = (row_diff == 1 && col_diff == 0) ? VIEW_DOWN : direction;
    //Up a square
    direction = (row_diff == 0 && col_diff == -1) ? VIEW_LEFT : direction;
    //Left a square
    direction = (row_diff == -1 && col_diff == 0) ? VIEW_UP : direction;

    //printf("CR: %d, CC: %d, GR: %d GC: %d\n", cur_row, cur_col, goal_row,
        ↪ goal_col);
}



void goalCallback(const yeet_msgs::node::ConstPtr& goal)
{
    //Read the new node
    goal_node = current_map.getNode(goal->row, goal->col);

    printf("NEW GOAL! row: %d, col: %d\n", goal->row, goal->col);

    //Set the search state to make a new plan
    search_state = NEW_GOAL;
}

```

```

void populateCallback(const yeet_msgs::node::ConstPtr& obstacle)
{
    current_map.getNode(obstacle->row, obstacle->col)->setObstacle(true);
    printf("NEW OBSTACLE! row: %d, col: %d\n", obstacle->row, obstacle->col);

}

void mapCallback(const yeet_msgs::node::ConstPtr & map_node)
{
    //Update the map
    //The node in this message is an obstacle, so we need to replan.
    //Reset the start node to where we actually are, and then replan
    start_node = current_map.getNode(map_node->row, map_node->col);

    //Set the robot to replan its route based on new environment information
    search_state = OBS_REPLAN;
}


bool nextTargetCallback(yeet_msgs::TargetNode::Request &req, yeet_msgs::
    ↪ TargetNode::Response &resp)
{
    int cur_row;
    int cur_col;
    int goal_col;
    int goal_row;

    //In IDLE mode, no targets exist
    if(search_state == IDLE)
    {
        return false;
    }

    //Get the current coords
    cur_col = start_node->getCol();
    cur_row = start_node->getRow();

    //Load the next node
    start_node = current_map.getNextWaypoint();

    //Get the goal coords
    goal_col = start_node->getCol();
    goal_row = start_node->getRow();

    updateView(cur_col, cur_row, goal_col, goal_row);

    //Calculate the target message
    resp.target.row = start_node->getRow();
    resp.target.col = start_node->getCol();
    resp.target.is_obstacle = start_node->isObstacle();

    //Change to the navigation system
    search_state = NAVIGATING;
    printf("GOING TO: [x: %d, y: %d]\n", start_node->getRow(), start_node->
        ↪ getCol());

    //Service succeeded
    return true;
```

```
}



int main(int argc, char **argv)
{
    //Initialize ROS
    ros::init(argc, argv, "d_star_node");

    //Set up the D* node
    ros::NodeHandle d_star_node;

    //Tracks obstacles
    std::shared_ptr<MapNode> obstacle_node;
    int obstacle_row;
    int obstacle_col;

    //Calculate the next target
    std::shared_ptr<MapNode> next_node;

    //Loop rate
    ros::Rate loop_rate(20);

    //Get data from our map when needed
    ros::Subscriber map_sub = d_star_node.subscribe(d_star_node.resolveName("/yeet_planning/map_update"), MAX_BUFFER, &mapCallback);

    //Subscribe to the task manager
    ros::Subscriber goal_sub = d_star_node.subscribe(d_star_node.resolveName("/yeet_planning/next_goal"), MAX_BUFFER, &goalCallback);

    ros::Subscriber populate_sub = d_star_node.subscribe(d_star_node.resolveName("/yeet_planning/grid_update"), MAX_BUFFER, &populateCallback);


    //Manage state and publish nodes only when requested
    ros::ServiceServer target_srv = d_star_node.advertiseService(d_star_node.resolveName("/yeet_planning/target_node"), &nextTargetCallback);

    //Publish drive enable commands whenever replanning is done
    std_msgs::Empty empty_msg;
    enable_drive_pub = d_star_node.advertise<std_msgs::Empty>(d_star_node.resolveName("/yeet_nav/enable_drive"), 10);

    //Initialize the start node
    start_node = current_map.getNode(0, 0);
    direction = VIEW_UP;

    //Wait for the task manager to tell us a goal node
    search_state = IDLE;

    //TODO: this is test code, pls remove once keyboard sends goal data
    //goal_node = current_map.getNode(3, 0);   //TODO: test
    //search_state = NEW_GOAL;                 //TODO: test

    while(ros::ok())
    {
        //If there is no path to goal, give up!
```

```
        if(start_node->getG() >= INFINITY)
        {
            search_state = IDLE;
            printf("D* LITE WARNING: Goal node unreachable\n");
        }

        //If the robot is in the process of navigating
        if(search_state == NAVIGATING)
        {
            //Don't do anything, let the robot drive.
        }
        //If an obstacle was found, replan
        else if(search_state == OBS_REPLAN)
        {
            //Since the obstacle is always in front, the node in front of the robot is an obstacle
            //Given the direction of the robot, determine the node that is an obstacle
            if(direction == VIEW_UP)
            {
                obstacle_row = start_node->getRow() + 1;
                obstacle_col = start_node->getCol();
            }
            else if(direction == VIEW_LEFT)
            {
                obstacle_row = start_node->getRow();
                obstacle_col = start_node->getCol() + 1;
            }
            else if(direction == VIEW_DOWN)
            {
                obstacle_row = start_node->getRow() - 1;
                obstacle_col = start_node->getCol();
            }
            else
            {
                obstacle_row = start_node->getRow();
                obstacle_col = start_node->getCol() - 1;
            }

            //Check to make sure we haven't found the wall
            if(obstacle_row < 0 || obstacle_row > (MAP_ROWS - 1) || obstacle_col < 0 || obstacle_col > (MAP_COLS - 1))
            {
                //TODO: Ignore out of bounds
                printf("This shouldn't happen\n");
                search_state = WAYPOINT;
                //FIXME: This is an error if this state is ever reached. We only care about obstacles that are in bounds
            }
            //Update the node that is an obstacle, assuming it is not out of bounds of the map
            else
            {
                printf("Obstacle located at Row: %d, Col %d!\n", obstacle_row, obstacle_col);
                //Update the RHS to be infinity
                obstacle_node = current_map.getNode(obstacle_row, obstacle_col);
                obstacle_node->reset();
```

```
                obstacle_node->setRHSInf();
                obstacle_node->setObstacle(true);

                /** Update all of the directed edge costs **/
                //Go through all adjacent nodes and update the vertices
                for(int i = 0; i < 4; ++i)
                {
                    next_node = current_map.getAdjacentNode(obstacle_node, i);

                    //Update the vertex of any valid node
                    if(next_node->getCol() != -1)
                    {
                        current_map.updateVertex(next_node);
                    }
                }

                //Reset the start node
                current_map.resetStartNode(start_node->getRow(), start_node->getCol());

                //Recalculate the path
                current_map.calculateShortestPath();

                current_map.printMap();
            }

            //Resume navigation
            search_state = WAYPOINT;
            enable_drive_pub.publish(empty_msg);
        }
        //If a new goal is selected, plan a new course
        else if(search_state == NEW_GOAL)
        {
            //Plan a new course
            current_map.planCourse(goal_node);

            //Set search state to load the next waypoint
            search_state = WAYPOINT;

            current_map.printMap();
        }

        //Get the callbacks
        ros::spinOnce();
        loop_rate.sleep();
    }

    return 0;
}
```

World Map Class - The world map maintained the state of the world and carried out the D*Lite processes using smart pointers

```
#ifndef WORLD_MAP_H
#define WORLD_MAP_H

#include <memory>
#include <vector>
#include "yeet_planning/map_cell.h"
```

```cpp
#include "yeet_planning/yeet_priority_queue.h"
#include "yeet_msgs/Constants.h"
#include "yeet_msgs/node.h"

#define DEBUG false

class WorldMap
{
    public:
        //Constructors
        WorldMap();
        WorldMap(int rows, int cols, double square_size);

        //Map functions
        void printMap();

        //D* helper functions
        void clearParams();
        void setGoal(int row, int col);
        int transitionCost(std::shared_ptr<MapNode> node_A, std::shared_ptr<MapNode> node_B);
        int heuristic(std::shared_ptr<MapNode> node_A, std::shared_ptr<MapNode> node_B);
        int calculateKey(std::shared_ptr<MapNode> node_ptr);
        void updateVertex(std::shared_ptr<MapNode> node);
        void expandNode(std::shared_ptr<MapNode> node);
        void calculateShortestPath();
        void planCourse(std::shared_ptr<MapNode> goal);
        std::shared_ptr<MapNode> getNextWaypoint();

        //Getters for nodes
        std::shared_ptr<MapNode> getNode(int row, int col);
        std::shared_ptr<MapNode> getAdjacentNode(std::shared_ptr<MapNode> node, int idx);
        std::shared_ptr<MapNode> resetStartNode(int row, int col);

        //Navigation helper functions
        std::shared_ptr<MapNode> getBestAdjNode(std::shared_ptr<MapNode> cur_node);

    private:
        //Map of nodes
        std::vector<std::vector<std::shared_ptr<MapNode> > > current_map;

        int num_rows;
        int num_cols;

        //D* variables
        yeet_priority_queue<MapNode> open_list;
        std::shared_ptr<MapNode> last_node; //TODO: use unique ptr here?
        std::shared_ptr<MapNode> start_node;
        std::shared_ptr<MapNode> goal_node;
};

#endif
```

```cpp
#include "yeet_planning/world_map.h"

```

```
WorldMap::WorldMap() //: current_map(yeet_msgs::Constants::MAP_ROWS, std::
    ↪ vector<std::shared_ptr<MapNode> >(yeet_msgs::Constants::MAP_COLS))
{
    //Declare local variables
    int c;
    int r;
    std::vector<std::shared_ptr<MapNode> > row;

    //Initialize values in the current map
    for(r = 0; r < yeet_msgs::Constants::MAP_ROWS; ++r)
    {
        for(c = 0; c < yeet_msgs::Constants::MAP_COLS; ++c)
        {
            //Set rows and columns
            std::shared_ptr<MapNode> node_ptr(new MapNode(r, c));
            node_ptr->reset();
            row.push_back(node_ptr);
        }

        this->current_map.push_back(row);
        row.clear();
    }

    start_node = current_map[0][0];
    goal_node = current_map[0][0];
}


WorldMap::WorldMap(int rows, int cols, double square_size) //: current_map(
    ↪ rows, std::vector<MapNode>(cols))
{
    //Declare local variables
    int c;
    int r;
    std::vector<std::shared_ptr<MapNode> > row;

    //Initialize values in the current map
    for(r = 0; r < rows; ++r)
    {
        for(c = 0; c < cols; ++c)
        {
            //Set rows and columns
            std::shared_ptr<MapNode> node_ptr(new MapNode(r, c));
            node_ptr->reset();
            row.push_back(node_ptr);
        }

        this->current_map.push_back(row);
        row.clear();
    }

    start_node = current_map[0][0];
    goal_node = current_map[0][0];
}


void WorldMap::printMap()
{
    for(int r = current_map.size() - 1; r >= 0; --r)
```

```
    {
        for(int c = current_map[0].size() - 1; c >= 0; --c)
        {
            printf("%d ", current_map[r][c]->getG());
        }
        printf("\n");
    }
}


void WorldMap::clearParams()
{
    //Declare local variables
    int i;
    int j;

    //Reset all nodes to g and rhs values of infinity
    for(i = 0; i < this->current_map.size(); ++i)
    {
        for(j = 0; j < this->current_map[i].size(); ++j)
        {
            this->current_map[i][j]->reset();
        }
    }
}


void WorldMap::setGoal(int row, int col)
{
    this->current_map[row][col]->setGoal();

    calculateKey(this->current_map[row][col]);
}


std::shared_ptr<MapNode> WorldMap::getAdjacentNode(std::shared_ptr<MapNode>
    ↪ node, int idx)
{
    //Declare local variables
    int col;
    int row;

    //Initialize variables
    col = node->getCol();
    row = node->getRow();

    /**
     * Check the 4 possible move directions
     */
    switch(idx)
    {
        case 0:
            return getNode(row + 1,col);
        case 1:
            return getNode(row - 1, col);
        case 2:
            return getNode(row, col - 1);
        case 3:
            return getNode(row, col + 1);
```

```
        default:
            return node;
    }
}


std::shared_ptr<MapNode> WorldMap::getBestAdjNode(std::shared_ptr<MapNode> cur_node)
{
    //Declare local variables
    int min_value = YEET_FINITY;
    std::shared_ptr<MapNode> min_node;
    std::shared_ptr<MapNode> neighbor;

    //If we are already at the goal, return the current node
    if(cur_node->getG() == 0)
    {
        return cur_node;
    }

    //Find the lowest valued element
    for(int i = 0; i < 4; ++i)
    {
        //Get an adjacent node
        neighbor = this->getAdjacentNode(cur_node, i);

        if(neighbor->getG() < min_value
        && neighbor->getCol() != -1)
        {
            printf("New Low!: G = %d, Row = %d, Col = %d\n", neighbor->getG(), neighbor->getRow(), neighbor->getCol());
            min_value = neighbor->getG();
            min_node = neighbor;
        }
    }

    return min_node;
}

std::shared_ptr<MapNode> WorldMap::getNode(int row, int col)
{
    std::shared_ptr<MapNode> err_ptr(new MapNode(-1, -1));

    //If the node is out of bounds, return an error
    if(row < 0 || row >= current_map.size() || col < 0 || col >= current_map[0].size())
    {
        return err_ptr;
    }

    return this->current_map[row][col];
}



/** D* Lite functionality **/

bool operator>(const std::shared_ptr<MapNode>& lhs, const std::shared_ptr<MapNode>& rhs)
```

```
{
    return *lhs > *rhs;
}

/**
 * @brief Find the cost of moving between two adjacent nodes
 * @param node_A Node A
 * @param node_B Node B
 * @return int
 */
int WorldMap::transitionCost(std::shared_ptr<MapNode> node_A, std::shared_ptr<MapNode> node_B)
{
    //If either node is an obstacle, the transitionCost is infinity
    if(node_A->isObstacle() || node_B->isObstacle())
    {
        return YEET_FINITY;
    }

    //Otherwise the distance is 1 (nodes are adjacent and diagonal moves are not allowed)
    return 1;
}


/**
 * @brief Heuristic used to find shortest possible distance between nodes
 * This uses manhattan distance, as diagonal movement is risky in our maze environment
 *
 * @param node_A Node A
 * @param node_B Node B
 * @return int The absolute value of the manhattan distance between two nodes
 */
int WorldMap::heuristic(std::shared_ptr<MapNode> node_A, std::shared_ptr<MapNode> node_B)
{
    return (abs(node_A->getRow() - node_B->getRow()) + abs(node_A->getCol() - node_B->getCol()));
}


int WorldMap::calculateKey(std::shared_ptr<MapNode> node_ptr)
{
    int key1 = std::min(node_ptr->getG(), node_ptr->getRHS()) + heuristic(start_node, node_ptr);
    int key2 = std::min(node_ptr->getG(), node_ptr->getRHS());

    //Set the node's new keys
    node_ptr->setKeys(key1, key2);

    return key1;
}



void WorldMap::updateVertex(std::shared_ptr<MapNode> node)
{
    //Declare local variables
```

```
    int i;
    std::shared_ptr<MapNode> neighbor_node;
    int node_g;
    int node_rhs;
    int succ_rhs;

    //Initialize local variables
    node_g = node->getG();
    node_rhs = node->getRHS();

    if(DEBUG)
    {
        printf("\tUpdating node(%d, %d)\n", node->getRow(), node->getCol());
    }
    //If the node is not a goal, update its RHS value
    if(!node->isGoal())
    {
        //Set the RHS to infinity for comparison
        node_rhs = YEET_FINITY;

        //Find the minimum RHS for this node
        for(i = 0; i < 4; ++i)
        {
            //Get neighboring nodes
            neighbor_node = this->getAdjacentNode(node, i);

            //Make sure the adjacent node is valid
            if(neighbor_node->getCol() != -1)
            {
                //Get the RHS computed from the neighbor node
                succ_rhs = neighbor_node->getG() + transitionCost(node, neighbor_node);

                if(DEBUG)
                {
                    printf("\t\tNeighbor Node(%d, %d) - G: %d, OBS: %d\n", neighbor_node->getRow(), neighbor_node->getCol(), neighbor_node->getG(), neighbor_node->isObstacle());
                    printf("\t\tSuccesssor RHS: %d\n", succ_rhs);
                }

                //If the RHS we just computed is lower, update the node's rhs
                if(succ_rhs < node_rhs)
                {
                    if(DEBUG)
                    {
                        printf("\t\tNew RHS: %d, Old Low: %d\n", succ_rhs, node_rhs);
                    }

                    node_rhs = succ_rhs;
                }
            }
        }

        //Calculate the new RHS for this node
        node->setRHS(node_rhs);

        if(DEBUG)
```

```
        {
            printf("\tNew RHS: %d, Current G: %d\n", node->getRHS(), node->
                ↪ getG());
        }
    }



    //If the node is on the open list, remove it from the list
    if(open_list.contains(*node))
    {
        //Remove the node from open list
        open_list.removeAll(*node);
    }

    //If the node is inconsistent, add it to the open list
    if(node->getG() != node->getRHS())
    {
        //Calculate the node's key
        calculateKey(node);

        if(DEBUG)
        {
            printf("\tNew Key: %d\n", node->getPrimaryKey());
        }

        //Add the node to the open list
        open_list.push(*node);
    }

}


void WorldMap::expandNode(std::shared_ptr<MapNode> node)
{
    //Declare local variables
    int i;
    std::shared_ptr<MapNode> neighbor_node;

    //Update each neighbor node to have an rhs value of 1 more than this node
    //Note: if this were being written for 8 possible movements, 1 would be
        ↪ used for
    //the non-diagonal moves, while 1.4 would be used for the diagonal moves
    for(i = 0; i < 4; ++i)
    {
        //Get a neighboring node
        neighbor_node = this->getAdjacentNode(node, i);

        //If the neighbor node is a valid node, update it
        if(neighbor_node->getCol() != -1)
        {
            updateVertex(neighbor_node);
        }
    }
}


void WorldMap::calculateShortestPath()
{
```

```
    //Declare local variables
    MapNode top_node;
    std::shared_ptr<MapNode> node_ptr;

    //Find top value on the open list and its pointer
    top_node = open_list.top();
    node_ptr = this->current_map[top_node.getRow()][top_node.getCol()];
    calculateKey(start_node);

    //Make nodes consistent
    while((top_node < *start_node || start_node->getG() != start_node->getRHS())
        && !open_list.empty())
    {
        //Pop the top node off the queue
        open_list.pop();

        //Find the pointer for this node
        node_ptr = this->current_map[top_node.getRow()][top_node.getCol()];

        if(DEBUG)
        {
            printf("queue size: %d\n", open_list.size());
            printf("Exploring node(%d, %d) g: %d, rhs: %d, key: %d\n", node_ptr->getRow(), node_ptr->getCol(), node_ptr->getG(), node_ptr->getRHS(), node_ptr->getPrimaryKey());
        }

        //If the g value is greater than the rhs, make the value consistent
        if(node_ptr->getG() > node_ptr->getRHS())
        {
            //Update the g-value to be over-consistent
            node_ptr->setG(node_ptr->getRHS());

            //Propagate changes to predecessor nodes
            expandNode(node_ptr);
        }
        else
        {
            //Set the g value to infinity
            node_ptr->setGInf();

            //Propagate changes to predecessor nodes
            expandNode(node_ptr);

            //Update this node
            updateVertex(node_ptr);
        }

        //Store the next top node and calculate the start node's key
        top_node = open_list.top();
        calculateKey(start_node);
    }
}



void WorldMap::planCourse(std::shared_ptr<MapNode> goal)
{
```

```
    /** Initialize search **/
    //Clear all values in the map nodes
    this->clearParams();

    //Set the goal node as the goal and calculate its key
    goal_node = goal;
    goal_node->setGoal();
    calculateKey(goal_node);

    //Add the goal node to the open list
    open_list.push(*goal_node);

    printf("D* Lite Initialized!\n");

    /** Search for optimal route **/
    //Calculate the shortest path from goal to start
    calculateShortestPath();
}


std::shared_ptr<MapNode> WorldMap::getNextWaypoint()
{
    //Load the next node
    last_node = start_node;
    start_node = this->getBestAdjNode(start_node);

    return start_node;
}


std::shared_ptr<MapNode> WorldMap::resetStartNode(int row, int col)
{
    start_node = this->getNode(row, col);

    return start_node;
}
```

Grid Cell Class - Grid cells were used to define the environment and this class helped with the search parameters as well.

```
#ifndef WORLD_MAP_NODE_H
#define WORLD_MAP_NODE_H

#include <algorithm>

//Tag constants
#define NEW     0
#define OPEN    1
#define CLOSED  2

//Set infinity
#define YEET_FINITY 1000000

class MapNode
{
    public:
        //Constructors
        MapNode();
        MapNode(int row, int col);
```

```

        //Operator overloads
        void operator=(const MapNode& node);
        bool operator==(const MapNode& node);
        bool operator>(const MapNode& right_node);
        bool operator<(const MapNode& right_node);

        //Node characteristic functions
        int getRow();
        int getCol();
        void setRow(int row);
        void setCol(int col);

        //D* search helper functions
        void reset();
        void setGoal();
        void setG(int g);
        void setGInf();
        void setRHS(int rhs);
        void setRHSInf();
        void setOpen();
        void setClosed();
        void setKeys(int primary, int secondary);
        void setObstacle(bool obs);
        int getG();
        int getRHS();
        int getPrimaryKey();
        bool isNew();
        bool isOpen();
        bool isGoal();
        bool isObstacle();
        //TODO: add more of these...

    private:
        //General node data
        int row;
        int col;
        int occupancy;

        //D* node data
        bool goal;
        int g_value;
        int rhs;
        int tag;
        int primary_key;
        int tiebreaker_key;

    friend bool operator>(const MapNode& lhs, const MapNode& rhs);
};

#endif
```

```
#include "yeet_planning/map_cell.h"

MapNode::MapNode()
{
    this->occupancy = 0;
}

```

```

MapNode::MapNode(int row, int col)
{
    this->col = col;
    this->row = row;
    this->occupancy = 0;
}


void MapNode::operator=(const MapNode& node)
{
    this->col = node.col;
    this->g_value = node.g_value;
    this->occupancy = occupancy;
    this->primary_key = node.primary_key;
    this->rhs = node.rhs;
    this->row = node.row;
    this->tag = node.tag;
    this->tiebreaker_key = node.tiebreaker_key;
}


bool MapNode::operator==(const MapNode& node)
{
    return (this->primary_key == node.primary_key) && (this->tiebreaker_key == node.tiebreaker_key) && (this->row == node.row) && (this->col == node.col);
}


bool MapNode::operator>(const MapNode& node)
{
    //Sort by tiebreaker key if needed
    if(node.primary_key == this->primary_key)
    {
        return this->tiebreaker_key > node.tiebreaker_key;
    }

    //Otherwise, determine the correct ordering with the primary key
    return this->primary_key > node.primary_key;
}

bool MapNode::operator<(const MapNode& right_node)
{
    //Sort by tiebreaker key if needed
    if(right_node.primary_key == this->primary_key)
    {
        return this->tiebreaker_key < right_node.tiebreaker_key;
    }

    //Otherwise, determine the correct ordering with the primary key
    return this->primary_key < right_node.primary_key;
}

bool operator>(const MapNode& lhs, const MapNode& rhs)
{
    if(lhs.primary_key == rhs.primary_key)
    {
        return lhs.tiebreaker_key > rhs.tiebreaker_key;
```

```
    }

    return lhs.primary_key > rhs.primary_key;
}


int MapNode::getRow()
{
    return this->row;
}


int MapNode::getCol()
{
    return this->col;
}


void MapNode::setRow(int row)
{
    this->row = row;
}


void MapNode::setCol(int col)
{
    this->col = col;
}


void MapNode::reset()
{
    this->g_value = YEET_FINITY;
    this->rhs = YEET_FINITY;
    this->primary_key = YEET_FINITY;
    this->tiebreaker_key = YEET_FINITY;
    this->tag = NEW;
    this->goal = false;
}


void MapNode::setGoal()
{
    this->rhs = 0;
    this->goal = true;
}


void MapNode::setG(int g)
{
    //YEET_FINITY is the maximum value for g
    if(g > YEET_FINITY)
    {
        g = YEET_FINITY;
    }

    this->g_value = g;
}

```

```cpp

void MapNode::setGInf()
{
    this->g_value = YEET_FINITY;
}


void MapNode::setRHS(int rhs)
{
    //YEET_FINITY is the maximum allowed value for RHS
    if(rhs > YEET_FINITY)
    {
        rhs = YEET_FINITY;
    }

    this->rhs = rhs;
}


void MapNode::setRHSInf()
{
    this->rhs = YEET_FINITY;
}


void MapNode::setOpen()
{
    this->tag = OPEN;
}


void MapNode::setClosed()
{
    this->tag = CLOSED;
}


void MapNode::setKeys(int primary, int secondary)
{
    this->primary_key = primary;
    this->tiebreaker_key = secondary;
}



void MapNode::setObstacle(bool obs)
{
    this->occupancy = obs;
}



int MapNode::getG()
{
    return this->g_value;
}


int MapNode::getRHS()
```

```
{
    return this->rhs;
}

int MapNode::getPrimaryKey()
{
    return this->primary_key;
}


bool MapNode::isNew()
{
    return (this->tag == NEW);
}


bool MapNode::isOpen()
{
    return (this->tag == OPEN);
}


bool MapNode::isGoal()
{
    return this->goal;
}


bool MapNode::isObstacle()
{
    return (occupancy >= 1);
}
```

Custom Priority Queue - A custom implementation of the C++ priority queue was needed for D*Lite and was implemented below

```
#ifndef YEET_PRIORITY_QUEUE_H
#define YEET_PRIORITY_QUEUE_H

#include <vector>
#include <queue>

//I learned how to do this here: https://stackoverflow.com/questions/19467485/
    ↪ how-to-remove-element-not-at-top-from-priority-queue

template<typename T>
class yeet_priority_queue : public std::priority_queue<T, std::vector<T>, std
    ↪ ::greater<T> >
{
    public:
        /**
         * @brief Removes an element from the priority queue
         *
         * @param val Element to remove
         * @return true If the element was successfully removed
         * @return false If the element could not be removed
         */
        bool remove(const T& val)
        {
```

```cpp
            //Find the element in the queue
            auto it = std::find(this->c.begin(), this->c.end(), val);

            //If the element was found, remove it and re-sort the queue
            if (it != this->c.end())
            {
                this->c.erase(it);
                std::make_heap(this->c.begin(), this->c.end(), this->comp);
                return true;
            }

            //The element is not in the queue, so return failure
            return false;
        }


        /**
         * @brief Identifies if an element exists in the queue
         *
         * @param val The element to find in the queue
         * @return true If the element is in the queue
         * @return false If the element is not in the queue
         */
        bool contains(const T& val)
        {
            //Find the element in the queue
            auto it = std::find(this->c.begin(), this->c.end(), val);

            //Return the status based on the iterator's location
            return (it != this->c.end());
        }


        /**
         * @brief Removes an element from the priority queue
         *
         * @param val Element to remove
         * @return true If the element was successfully removed
         * @return false If the element could not be removed
         */
        void removeAll(const T& val)
        {
            while(this->contains(val))
            {
                this->remove(val);
            }
        }

};

#endif
```

C Navigation Package

C.1 Navigation

Navigation ROS Node - The navigation node managed the robot's motion commands based on the D*Lite commands. It used a PID controller to keep the robot on its path.

```
//ROS libs and msgs
#include <ros/ros.h>
#include <nav_msgs/Odometry.h>
#include <tf/transform_datatypes.h>
#include <tf/transform_listener.h>
#include <std_msgs/Empty.h>

//User msgs and libs
#include <yeet_nav/pid_controller.h>
#include "yeet_msgs/Constants.h"
#include "yeet_msgs/nav_status.h"
#include "yeet_msgs/move.h"
#include "yeet_msgs/node.h"
#include "yeet_msgs/obstacle.h"
#include "yeet_msgs/TargetNode.h"

//Constants
#define RAD_TO_DEG (double)(180.0 / 3.14159) //Conversion factor from radians to degrees
#define DISTANCE_TOL (double)(0.05)          //Tolerance for drive distance
#define ANGLE_TOL (double)(1.0)              //Tolerance for angle distance
#define UP 0                                 //Up map angle
#define LEFT 90                              //Left map angle
#define RIGHT -90                            //Right map angle
#define DOWN 180                             //Down map angle
#define NAV_BUFFER (double)(0.125)          //Tiny buffer added to get to a square center

//Drive X PID
#define X_KP (double)(0.6)
#define X_KI (double)(0.002)
#define X_KD (double)(0.001)
#define X_MAX_OUTPUT 0.20
#define X_MIN_OUTPUT -0.20

//Drive Y PID
#define Y_KP (double)(0.6)
#define Y_KI (double)(0.002)
#define Y_KD (double)(0.01)
#define Y_MAX_OUTPUT 0.20
#define Y_MIN_OUTPUT -0.20

//Turn PID
#define TURN_KP (double)(0.23)
#define TURN_KI (double)(0.000)
#define TURN_KD (double)(0.003)
#define TURN_MAX_OUTPUT 0.5
#define TURN_MIN_OUTPUT -0.5

//Ramping
double sec_last_out;
double last_output;

//PID
PID_Controller turn;
PID_Controller drive_x;
PID_Controller drive_y;

```

```cpp
//Global variables
yeet_msgs::Constants constants;
double current_angle;
int map_angle;
int goal_row;
int goal_col;
int last_goal_row;
int last_goal_col;
int cur_row;
int cur_col;
int row_diff;
int col_diff;
double x;
double y;
double goal_x;
double goal_y;
bool drive_enabled;

//ROS service
ros::ServiceClient target_srv;
yeet_msgs::TargetNode target_node;

//ROS Publishers
ros::Publisher obstacle_pub;

/**
 * @brief angleWrap - Keep angles within the expected range
 *
 * @param angle - Unwrapped angle
 * @return double - Angle between 0-360
 */
double angleWrap(double angle)
{
    return angle < 0 ? fmod(angle, 360) + 360 : fmod(angle, 360);
}

/**
 * @brief goalCallBack- Updates global variables for the PID Controller to use .
 *
 * @param goal - The information about the robot's goal
 */
void goalCallBack(const yeet_msgs::node goal)
{
    drive_x.reset(x);
    drive_y.reset(y);
    turn.reset(current_angle);
    goal_row = goal.row;
    goal_col = goal.col;
    goal_x = ((double)(goal_row)*yeet_msgs::Constants::SQUARE_SIZE) + NAV_BUFFER;
    goal_y = ((double)(goal_col)*yeet_msgs::Constants::SQUARE_SIZE) + NAV_BUFFER;

    //Get the difference in rows and columns
    col_diff = last_goal_col - goal_col;
    row_diff = last_goal_row - goal_row;

    printf("last row: %d last col: %d\n", last_goal_row, last_goal_col);
```

```
    map_angle = 0; //current_angle;
    //Down a square
    map_angle = (row_diff == 0 && col_diff == 1) ? RIGHT : map_angle;
    //Right a sqaure
    map_angle = (row_diff == 1 && col_diff == 0) ? DOWN : map_angle;
    //Up a square
    map_angle = (row_diff == 0 && col_diff == -1) ? LEFT : map_angle;
    //Left a square
    map_angle = (row_diff == -1 && col_diff == 0) ? UP : map_angle;

    printf("NAV_NODE: Received command to move to row: %d col: %d angle:%d\n",
        ↪ goal_row, goal_col, map_angle);

    //Set the last goal row and column
    last_goal_col = goal_col;
    last_goal_row = goal_row;
}

/*
void odomCallBack(const nav_msgs::Odometry::ConstPtr &odom)
{
    //TODO
}
*/

/**
 * @brief odomCallBack - Gets the current angle of the robot in degrees
 *
 * @param odom - The odometry message containing robot angle position
 */
void odomCallBack(const nav_msgs::Odometry::ConstPtr& odom)
{
    //Get the robot orientation
    tf::Pose pose;
    tf::poseMsgToTF(odom->pose.pose, pose);

    //Get the current angle in degrees
    current_angle = angleWrap(tf::getYaw(pose.getRotation()) * RAD_TO_DEG);

    //printf("current_angle: %f\n", current_angle);

    //Get the current row and column from x and y position
    x = odom->pose.pose.position.x;
    y = odom->pose.pose.position.y;
    cur_col = (int) round(y / constants.SQUARE_SIZE);
    cur_row = (int) round(x / constants.SQUARE_SIZE);
}


void replanCallback(const yeet_msgs::obstacle::ConstPtr &obstacle_msg)
{
    //Declare local variables
    yeet_msgs::node obstacle_node;

    if (obstacle_msg->obstacle && drive_enabled)
    {
        drive_enabled = false;
        last_goal_col = cur_col;
```

```
        last_goal_row = cur_row;

        //Publish to D* to alert need to replan, given the goal node is an
            ↪ obstacle
        obstacle_node.col = cur_col;
        obstacle_node.row = cur_row;
        obstacle_node.is_obstacle = true;

        printf("REPLAN: Current node is: %d, %d\n", cur_row, cur_col);

        obstacle_pub.publish(obstacle_node);
    }
}

void enableDriveCallback(const std_msgs::Empty::ConstPtr &empty)
{
    drive_enabled = true;

    if (target_srv.call(target_node))
    {
        goalCallBack(target_node.response.target);
    }
}

/**
 * @brief sweep -
 *
 * @param target_angle - The desired turn angle
 * @return double - Returns the error in angle. Negative if the turn
 * should be to the right, positive if left.
 */
double sweep(double target_angle)
{
    double sweep = target_angle - current_angle;
    sweep = (sweep > 180) ? sweep - 360 : sweep;
    sweep = (sweep < -180) ? sweep + 360 : sweep;
    return sweep;
}

/**
 * @brief - Main method
 *
 * @param argc - Number of args
 * @param argv - Args into the executable
 * @return int - Exit code
 */
int main(int argc, char **argv)
{
    //Start the node
    ros::init(argc, argv, "nav_node");

    //Set up the node to handle motion
    ros::NodeHandle nav_node;

    //tf::TransformListener listener;

    //Initialize default location values
    current_angle = 0.0;
    x = 0.0;
```

```
    y = 0.0;
    cur_row = 0;
    cur_col = 0;
    goal_row = 0;
    goal_col = 0;
    last_goal_row = 0;
    last_goal_col = 0;
    last_output = 0;
    sec_last_out = 0;
    drive_enabled = true;

    //Initialize PID
    drive_x.init(X_KP, X_KI, X_KD, X_MAX_OUTPUT, X_MIN_OUTPUT);
    drive_y.init(Y_KP, Y_KI, Y_KD, Y_MAX_OUTPUT, Y_MIN_OUTPUT);
    turn.init(TURN_KP, TURN_KI, TURN_KD, TURN_MAX_OUTPUT, TURN_MIN_OUTPUT);

    //Subscribe to topics
    ros::Subscriber odom_sub = nav_node.subscribe(nav_node.resolveName("/odom"), 10, &odomCallBack);

    ros::Subscriber replan_sub = nav_node.subscribe(nav_node.resolveName("/yeet_nav/replan"), 10, &replanCallback);
    ros::Subscriber enable_drive_sub = nav_node.subscribe(nav_node.resolveName("/yeet_nav/enable_drive"), 10, &enableDriveCallback);

    //Publishers
    ros::Publisher move_pub = nav_node.advertise<yeet_msgs::move>(
        nav_node.resolveName("/yeet_nav/navigation"), 10);

    obstacle_pub = nav_node.advertise<yeet_msgs::node>(nav_node.resolveName("/yeet_planning/map_update"), 10);

    //Service for requesting new target_node
    target_srv = nav_node.serviceClient<yeet_msgs::TargetNode>(nav_node.resolveName("/yeet_planning/target_node"));

    //Set the loop rate of the nav function to 100 Hz
    ros::Rate loop_rate(100);

    //Create local messages
    yeet_msgs::move move;

    //The callback and logic loop
    while (ros::ok())
    {
        //Initialize to zero
        move.drive = 0;
        move.turn = 0;

        /*
        tf::StampedTransform transform;

        try
        {
            listener.lookupTransform("/map", "/base_link", ros::Time(0), transform);
        }
        catch(tf::TransformException e)
        {
```

```
            //Do nothing, lol
            printf("LOL get rekt\n");
        }

        //Get the current angle in degrees
        //current_angle = angleWrap(tf::getYaw(transform.getRotation()) * RAD_TO_DEG);

        //printf("current_angle: %f\n", current_angle);

        //Get the current row and column from x and y position
        x = transform.getOrigin().x();
        y = transform.getOrigin().y();
        cur_col = (int)round(y / constants.SQUARE_SIZE);
        cur_row = (int)round(x / constants.SQUARE_SIZE);
        */

        //Only drive if driving is enabled
        if (drive_enabled)
        {
            //Within tolerance, stop turning and start driving
            if (abs(sweep((double)(map_angle))) <= ANGLE_TOL)
            {
                //printf("X ERR: %f, Y ERR: %f, ANG: %d\n", fabs(x - goal_x), fabs(y - goal_y), map_angle);

                if ((fabs(x - goal_x) < DISTANCE_TOL && (map_angle == DOWN || map_angle == UP)) || (fabs(y - goal_y) < DISTANCE_TOL && (map_angle == LEFT || map_angle == RIGHT)))
                {
                    //We have reached the goal, so get a new node from the D*
                    if (target_srv.call(target_node))
                    {
                        goalCallBack(target_node.response.target);
                    }
                    //Otherwise notify there was an error
                    else
                    {
                        //printf("NAV_NODE ERROR: Service call to D* Lite failed!\n");
                    }
                }
                else
                {
                    //printf("YEET 2 %f vs %f @ %d \n", x, goal_x, map_angle);
                    move.drive = (map_angle == DOWN || map_angle == UP) ? fabs(drive_x.getOutput(goal_x, x)) : move.drive;
                    move.drive = (map_angle == LEFT || map_angle == RIGHT) ? fabs(drive_y.getOutput(goal_y, y)) : move.drive;
                }
            }
            //Keep turning and do not drive
            else
            {
                printf("ERR: %f\n", sweep((double)(map_angle)));
                move.turn = -turn.getOutput(0, sweep((double)(map_angle)));
            }

            //Ramp up and down
```

```
            //move.drive = (move.drive + last_output + sec_last_out) / 3.0;
            //sec_last_out = last_output;
            //last_output = move.drive;

            //Publish
            move_pub.publish(move);
        }

        ros::spinOnce();
        loop_rate.sleep();
    }
}
```

PID Controller - A PID controller was used to make the robot's motion more consistent and error free. Using past work from Sooner Competitive Robotics[4] we implemented our own version of the PID controller.

```
#ifndef PID_CONTROLLER_H
#define PID_CONTROLLER_H

//ROS and systems libs
#include <ros/ros.h>
#include <ros/console.h>


class PID_Controller
{
    public:
    PID_Controller();
    PID_Controller(double cur_var);
    void init(double p, double i, double d, double max_out, double min_out);
    void reset(double cur_var);
    double getOutput(double setpoint, double process_var);

    private:
    double coerce(double pid_val);
    double integrator;
    double last_time;
    double last_var;
    double err;
    double prev_err;

    double KP;
    double KI;
    double KD;
    double MAX_OUTPUT;
    double MIN_OUTPUT;

};

#endif
```

```
//User libs
#include "yeet_nav/pid_controller.h"

/**
 * @brief Construct a new pid controller::pid controller object
 *
 */
PID_Controller::PID_Controller()
```

```
{
    this->last_var = 0;
    this->integrator = 0;
    this->err = 0;
    this->prev_err = 0;
}

/**
 * @brief Construct a new pid controller::pid controller object
 *
 * @param cur_var - Current value
 */
PID_Controller::PID_Controller(double cur_var)
{
    this->last_var = cur_var;
    this->integrator = 0;
    this->err = 0;
    this->prev_err = 0;
}

/**
 * @brief Resets the PID if needed.
 *
 * @param cur_var - Current value
 */
void PID_Controller::reset(double cur_var)
{
    this->last_time = ros::Time::now().toNSec() / 1000.0 / 1000.0 / 1000.0;
    this->last_var = cur_var;
    this->integrator = 0;
    this->err = 0;
    this->prev_err = 0;
}

void PID_Controller::init(double p, double i, double d, double max_out, double
    ↪ min_out)
{
    this->KP = p;
    this->KI = i;
    this->KD = d;
    this->MAX_OUTPUT = max_out;
    this->MIN_OUTPUT = min_out;
}

/**
 * @brief Gets the output from the PID
 *
 * @param setpoint - The target value
 * @param cur_var - The current value
 * @return double - The output (p + i + d)
 */
double PID_Controller::getOutput(double setpoint, double process_var)
{
    double cur_time = ros::Time::now().toNSec() / 1000.0 / 1000.0 / 1000.0;
    this->err = setpoint - process_var;
    double p = KP * this->err;
    double dt = this->last_time - cur_time;

    this->prev_err = setpoint - this->last_var;
```

```
    this->integrator += (0.5) * (this->err + this->prev_err) * dt;
    double i = KI * this->integrator;

    double delta = (process_var - this->last_var)/dt;
    double d = KD * delta;

    double output = coerce(p + i + d);

    this->last_var = process_var;
    this->last_time = cur_time;

    return output;
}

/**
 * @brief Clamps the output to max and min output if needed
 *
 * @param pid_val - P + I + D
 * @return double - The coerced output
 */
double PID_Controller::coerce(double pid_val)
{
    pid_val = pid_val > MAX_OUTPUT ? MAX_OUTPUT : pid_val;
    pid_val = pid_val < MIN_OUTPUT ? MIN_OUTPUT : pid_val;

    return pid_val;
}
```

C.2 Obstacle Avoidance

Obstacle Avoidance ROS Node - The obstacle avoidance node took input from a kinect sensor and determined if there was an obstacle in front of the robot. If there was an obstacle, forward driving would be inhibited. If forward driving was necessary, a replanning command would occur.

```
//ROS msgs and libs
#include <ros/ros.h>
#include <sensor_msgs/LaserScan.h>


//User libs and msgs
#include "yeet_msgs/obstacle.h"
#include "yeet_msgs/move.h"
#include "yeet_msgs/Constants.h"

//Other libs
#include <cmath>
#include <math.h>


//Constants
#define TOTAL_SAMPLES 640                                          //Laser scan samples
#define N_CENTER_SAMPLES 75                                        //Width of center view region
#define N_SIDE_SAMPLES ((TOTAL_SAMPLES - N_CENTER_SAMPLES) / 2)    //Allocate the number of samples for the side views
#define LEFT_SAMPLES_IDX TOTAL_SAMPLES - N_SIDE_SAMPLES            //Where the left samples start
```

```
#define RIGHT_SAMPLES_IDX N_SIDE_SAMPLES                                //Where the right samples end
#define DETECT_CONST (double)(0.0)

yeet_msgs::Constants constants;

//Publishers
ros::Publisher move_pub;
ros::Publisher obstacle_pub;
ros::Publisher replan_pub;

//Track state of obstacles
yeet_msgs::obstacle obstacle_msg;

/**
 * @brief - Detects obstacles in front of the robot
 *
 * @param obstacle_event: the message sent from the LaserScan sensor messages containing the information of the scan
 */
void scanCallback(const sensor_msgs::LaserScan::ConstPtr& obstacle_event)
{
    //Loop through all the samples and update bool if any index has an object
    for(int i = RIGHT_SAMPLES_IDX; i < LEFT_SAMPLES_IDX; ++i)
    {
         //If an obstacle is detected in the next cell
        if(!std::isnan(obstacle_event->ranges[i]) && obstacle_event->ranges[i] <= constants.SQUARE_SIZE + DETECT_CONST)
        {
            //Update obstacle boolean to true
            obstacle_msg.obstacle = true;
            break;
        }
        else
        {
            //Set the obstacle detection to false
            obstacle_msg.obstacle = false;
        }

    }

    //TODO: is this publish needed?
    //Publish
    obstacle_pub.publish(obstacle_msg);
}


void navCallback(const yeet_msgs::move::ConstPtr& move_msg)
{
    yeet_msgs::move cmd;

    //Initialize the command to zeros
    cmd.drive = 0;
    cmd.turn = 0;

    //If there is an obstacle, inhibit driving forward
    if(obstacle_msg.obstacle)
    {
        printf("Obstacle detected! Inhibiting drive output\n");
```

```
        cmd.drive = 0;
        cmd.turn = move_msg->turn;

        //If the robot was supposed to drive forward, but the path is blocked,
            ↪ send a replan command
        if(fabs(move_msg->turn - 0.0) < 0.01)
        {
            replan_pub.publish(obstacle_msg);
        }
    }
    //Otherwise, send the command through
    else
    {
        cmd.drive = move_msg->drive;
        cmd.turn = move_msg->turn;
    }

    //Publish the move down the chain
    move_pub.publish(cmd);
}


/**
 * @brief Main method
 *
 * @param argc Number of args
 * @param argv Args into the executable
 * @return int Exit code
 */
int main(int argc, char **argv)
{
    //Start the node
    ros::init(argc, argv, "obstacle_avoid_node");

    //Set up the node handle for auto driving
    ros::NodeHandle obstacle_avoid_node;

    //Subscribe to the scanner
    ros::Subscriber obstacle_sub = obstacle_avoid_node.subscribe(
        obstacle_avoid_node.resolveName("/scan"), 10, &scanCallback);

    //Subscribe to the navigation node for passthrough
    ros::Subscriber nav_sub = obstacle_avoid_node.subscribe(
        ↪ obstacle_avoid_node.resolveName("/yeet_nav/navigation"), 10, &
        ↪ navCallback);

    //Publish state to the obstacle topic
    obstacle_pub = obstacle_avoid_node.advertise<yeet_msgs::obstacle>(
        obstacle_avoid_node.resolveName("/yeet_msgs/obstacle"), 10);

    //Publish movement to the lower nodes
    move_pub = obstacle_avoid_node.advertise<yeet_msgs::move>(
        ↪ obstacle_avoid_node.resolveName("/yeet_nav/nav_cmd"), 10);

    //Replan when needed
    replan_pub = obstacle_avoid_node.advertise<yeet_msgs::obstacle>(
        ↪ obstacle_avoid_node.resolveName("/yeet_nav/replan"), 10);

    //Handle the callbacks
```

```
    ros::spin();
}
```

D External Control Package

D.1 Collision Detection

Collision Detection ROS Node - In case of a collision, this node would stop the robot until it is rescued.

```
//ROS libs and msgs
#include <ros/ros.h>
#include <kobuki_msgs/BumperEvent.h>

//User libs and msgs
#include <yeet_msgs/move.h>


/* Typedefs */
//Bumper state struct
typedef struct bumper_state_t
{
    bool left_bumper;
    bool center_bumper;
    bool right_bumper;
} bumper_state;


//Publisher
ros::Publisher collision_pub;
yeet_msgs::move move_msg;    //Published message


/* Global variables */
bumper_state bump_states;    //Keep track of the bumper states
bool collision;              //Whether or not collision is detected


/**
 * collision_callback - when collisions are detected by the bumpers, track the state of the bumpers
 * in order to halt the robot before further damage occurs.
 *
 * @param bump_event: the message sent from the kobuki base indicating the state of a bumper
 */
void collision_callback(const kobuki_msgs::BumperEvent::ConstPtr& bump_event)
{
    //If a bumper was pressed, make sure to note that in the bumper tracking struct
    if(bump_event->state == bump_event->PRESSED)
    {
        bump_states.left_bumper = bump_event->bumper == bump_event->LEFT ? 1 : bump_states.left_bumper;
        bump_states.center_bumper = bump_event->bumper == bump_event->CENTER ? 1 : bump_states.center_bumper;
        bump_states.right_bumper = bump_event->bumper == bump_event->RIGHT ? 1 : bump_states.right_bumper;
    }
```

```
    //On the other hand, if this bumper isn't pressed, reset the appropriate flag
    else
    {
        bump_states.left_bumper = bump_event->bumper == bump_event->LEFT ? 0 : bump_states.left_bumper;
        bump_states.center_bumper = bump_event->bumper == bump_event->CENTER ? 0 : bump_states.center_bumper;
        bump_states.right_bumper = bump_event->bumper == bump_event->RIGHT ? 0 : bump_states.right_bumper;
    }

    //Collision message construction
    //If any of the flags are set to 1, we need to set the collision message flag to true
    if(bump_states.left_bumper || bump_states.center_bumper || bump_states.right_bumper)
    {
        collision = true;
    }
    //Otherwise, since no bumpers are pressed, there are no collisions
    else
    {
        collision = false;
    }
}

/**
 * human_control_callback - when updated instructions are recieved from the human_control_node,
 * updat the message that will be sent to contain its contents.
 *
 * @param human_control_msg: message containing the final instructions from the nodes above
 * collision_node in the paradigm
 */
void navigation_callback(const yeet_msgs::move::ConstPtr& navigation_msg)
{
    //Simply copy the contents over
    move_msg = *navigation_msg;
}


/**
 * Runs the loop needed to handle collision detection
 */
int main(int argc, char **argv)
{

    //Start the node
    ros::init(argc, argv, "collision_node");

    //Set up the node handle for collision detection
    ros::NodeHandle collision_node;

    //Subscribe to the bump sensors
    ros::Subscriber bump_sub = collision_node.subscribe(
        collision_node.resolveName("/mobile_base/events/bumper"), 10, &collision_callback);
```

```

    //Subscribe to the node directly above in the paradigm
    ros::Subscriber human_control_sub = collision_node.subscribe(
        collision_node.resolveName("/yeet_nav/nav_cmd"), 10, &
            ↪ navigation_callback);


    //Publish state to the collision topic
    collision_pub = collision_node.advertise<yeet_msgs::move>("/yeet_mergency/
        ↪ collision", 10);


    //Set the loop rate of the decision function to 100 Hz
    ros::Rate loop_rate(100);

    //Make a decisision for what to do
    while(ros::ok())
    {
        //Perform all the callbacks
        ros::spinOnce();

        if(collision)
        {
            //TODO: Call backup-routine
            move_msg.drive = 0;
            move_msg.turn = 0;
        }

        //Publish the message
        collision_pub.publish(move_msg);

        //Finish the current loop
        loop_rate.sleep();
    }

}
```

D.2 Human Control and Rescue

Human Control ROS Node - This node takes input from the keyboard to rescue the robot

```
//ROS libs and msgs
#include <ros/ros.h>

//User libs and msgs
#include <yeet_msgs/move.h>



//Publisher
ros::Publisher human_control_pub;
yeet_msgs::move move_msg;    //Published message


/* Global variables */
bool keyboard_override = false;
double keyboard_drive;
double keyboard_turn;

```

```cpp
/**
 * @brief collision_callback - recieves instructions from collision_node and places them into a message. This data may
 * potentially be overwritten by the keyboard controls.
 *
 * @param collision_msg: the message sent from collision_node
 */
void collision_callback(const yeet_msgs::move::ConstPtr& collision_msg)
{
    move_msg = *collision_msg;
}

/**
 * @brief keyboard_callback - recieves key values from the keyboard_node and coverts them to movement, overriding
 * any commands from higher up the chain.
 *
 * @param keyboard_msg: message recieved from keyboard_node
 */
void keyboard_callback(const yeet_msgs::move::ConstPtr& keyboard_move_msg)
{
    keyboard_override = true;
    keyboard_drive = keyboard_move_msg->drive;
    keyboard_turn = keyboard_move_msg->turn;
}



/**
 * Runs the loop needed to handle human control of the robot
 */
int main(int argc, char **argv)
{

    //Start the node
    ros::init(argc, argv, "human_control_node");

    //Set up the node handle for collision detection
    ros::NodeHandle human_control_node;

    //Subscribe to the bump sensors
    ros::Subscriber navigation_sub = human_control_node.subscribe(
        human_control_node.resolveName("/yeet_mergency/collision"), 10, &collision_callback);

    //TODO: Create topic between human_control_node and keyboard_node
    //Subscribe to the node directly above in the paradigm
    ros::Subscriber human_control_sub = human_control_node.subscribe(
        human_control_node.resolveName("/yeet_board/keyboard"), 10, &keyboard_callback);


    //Publish state to the collision topic
    human_control_pub = human_control_node.advertise<yeet_msgs::move>("/yeet_mergency/human_control", 10);


    //Set the loop rate of the decision function to 100 Hz
    ros::Rate loop_rate(100);
```

```

    //Make a decisision for what to do
    while(ros::ok())
    {
        //Perform all the callbacks
        ros::spinOnce();

        //TODO: All of it
        if(keyboard_override)
        {
            move_msg.drive = keyboard_drive;
            move_msg.turn = keyboard_turn;
            keyboard_override = false;
        }

        //Publish the message
        human_control_pub.publish(move_msg);

        //Finish the current loop
        loop_rate.sleep();
    }

}
```

Custom Keyboard ROS Node - A keyboard was implemented to send obstacles, goal grid cells and rescue commands to the robot. The Linux terminal was modified using TermiOS.

```
#!/usr/bin/env python

from __future__ import print_function

import roslib; roslib.load_manifest('yeet_board')
import rospy

from geometry_msgs.msg import Twist
from yeet_msgs.msg import move
from yeet_msgs.msg import node
from yeet_msgs.msg import Constants

import sys, select, termios, tty

class color:
    PURPLE = '\033[95m'
    CYAN = '\033[96m'
    DARKCYAN = '\033[36m'
    BLUE = '\033[94m'
    GREEN = '\033[92m'
    YELLOW = '\033[93m'
    RED = '\033[91m'
    BOLD = '\033[1m'
    UNDERLINE = '\033[4m'
    END = '\033[0m'


instructions = """
\n\n\n\n\n\n\n\n
Moving around:
         ^
         |
```

```
           w
<-- a      s      d -->
           |
           v

q : speed up
e : slow down
anything else : stop

CTRL-C (^C) to enter token mode
CTRL-Z (^Z) to quit
"""

# 'key': (movement direction, turn direction, move speed modify)
keyBindings = {
        'w':( 1, 0, 0),
        's':(-1, 0, 0),
        'a':( 0,-1, 0),
        'd':( 0, 1, 0),
        'q':( 0, 0,-1),
        'e':( 0, 0, 1),
        ' ':( 0, 0, 0)
    }

exitCommands = [
    'exit',
    'quit',
    'q',
]

stdCommands = [
    'goto',
    'node',
    'raw',
    'help',
]

def getKey():
    tty.setraw(sys.stdin.fileno())
    select.select([sys.stdin], [], [], 0)
    key = sys.stdin.read(1)
    termios.tcsetattr(sys.stdin, termios.TCSADRAIN, settings)
    return key

def status_report(drive, turn, drive_speed, turn_speed, key):
    return "drive:%s\tturn:%s\ndrive speed:%s\tturn speed:%s\tchar:%s" % (float(drive), float(turn), float(drive_speed), float(turn_speed), key)

def help():
    print("Current commands are",)
    for command in stdCommands:
        print(command, '\t')
    print()

def updateGrid(row, col, is_obstacle):
    update_msg = node()
    update_msg.row = int(row)
    update_msg.col = int(col)
```

```python
    update_msg.is_obstacle = bool(is_obstacle)
    update_pub.publish(update_msg)


grid = [bool(i % 2) for i in range(Constants.MAP_COLS*Constants.MAP_ROWS)]
def printGrid():
    #TODO: request data struct
    for i, node in enumerate(grid, start = 1):
        print('X' if node else 'O', end=' ')
        if i % Constants.MAP_COLS == 0:
            print()


if __name__=="__main__":
    settings = termios.tcgetattr(sys.stdin)

    update_pub = rospy.Publisher('/yeet_planning/grid_update', node, queue_size = 10)
    goto_pub = rospy.Publisher('/yeet_planning/next_goal', node, queue_size = 10)
    keyboard_pub = rospy.Publisher('/yeet_board/keyboard', move, queue_size = 10)
    rospy.init_node('keyboard_node')

    drive = 0   # drive command to be published in the move (-/+:backwards/forwards)
    turn = 0    # turn speed (-/+:R/L)
    drive_speed = 0.5   # drive speed scaler
    turn_speed = 0.5    # turn speed scaler (const)

    mode = "token"

    key = ' '

    try:
        while(1):
            if mode == "token":
                print(color.BOLD + 'Enter command: ' + color.END, end='')
                s = raw_input()
                tokens = s.split()

                if tokens and tokens[0] is not '':
                    if tokens[0] in exitCommands:
                        break

                    elif tokens[0] == "goto":
                        try:
                            x_coord = int(tokens[1])
                            y_coord = int(tokens[2])
                            goto_msg = node()
                            goto_msg.row = x_coord
                            goto_msg.col = y_coord
                            goto_msg.is_obstacle = False
                            goto_pub.publish(goto_msg)
                            print("Going to coordinate x=", x_coord, ", y=", y_coord)
                        except IndexError:
                            print(color.RED + "goto requires ", color.BOLD + " two" + color.END, color.RED + " integer
```

```
            parameters:" + color.END, "Usage \' goto 4 5 \'")
    except ValueError:
        print(color.RED + "goto requires two ", color.BOLD + "integer" + color.END, color.RED + " parameters:" + color.END, "Usage \' goto 4 5 \'")

elif tokens[0] == "node":
    try:
        flag = tokens[3]
    except IndexError:
        print(color.RED + "node requires two integer parameters ", color.BOLD + "and" + color.END, color.RED + "a flag:" + color.END, "Usage \' goto -c 4 5\'")
        print("Current flags are: -c, -p, -g")
    try:
        x_coord = int(tokens[1])
        y_coord = int(tokens[2])
    except IndexError:
        print(color.RED + "node requires ", color.BOLD + "two" + color.END, color.RED + " integer parameters and a flag:" + color.END, "Usage \' goto -c 4 5\'")
    except ValueError:
        print(color.RED + "node requires two ", color.BOLD + "integer" + color.END, color.RED + " parameters and a flag:" + color.END, "Usage \' goto -c 4 5\'")

    if flag == "-g" or flag =="goto":
        goto_msg = node()
        goto_msg.row = x_coord
        goto_msg.col = y_coord
        goto_msg.is_obstacle = False
        goto_pub.publish(goto_msg)
        print("Going to coordinate x=", x_coord, ", y=", y_coord)

    elif flag == "-c" or flag == "clear":
        updateGrid(x_coord, y_coord, False)
        print("Clearing grid coordinate x=", x_coord, ", y=", y_coord)

    elif flag == "-p" or flag == "populate" or flag == "fill":
        updateGrid(x_coord, y_coord, True)
        print("Populating grid coordinate x=", x_coord, ", y=", y_coord)

    else:
        print("Unrecognized flag, please try again.")

elif tokens[0] == "grid":
    printGrid()


elif tokens[0] == "raw":
```

```
                        print("* Switched to raw mode")
                        mode = 'raw'
                        print(instructions)

                    elif tokens[0] == 'help':
                        help()
                    else:
                        print("* Oops!", tokens[0], "is not a command.")
                        help()
                print()



            else: # mode == "raw"
                print(instructions)
                print(status_report(drive, turn, drive_speed, turn_speed, key))

                key = getKey()

                if key in keyBindings.keys():
                    drive = keyBindings[key][0] * drive_speed
                    turn = keyBindings[key][1] * turn_speed
                    drive_speed += keyBindings[key][2] * 0.1

                else:
                    drive = 0
                    turn = 0
                    if (key == '\x03'):         # Ctrl-C
                        print("Switched to token mode")
                        mode = "token"
                        key = ' '   # reset key to avoid ugly print

                    elif (key == '\x1A'):       # Ctrl-Z
                        break

                move_msg = move()
                move_msg.drive = drive
                move_msg.turn = turn
                keyboard_pub.publish(move_msg)


    except Exception as e:
        print(e)

    finally:
        termios.tcsetattr(sys.stdin, termios.TCSADRAIN, settings)
```

E Low Level Movement Package

E.1 Drivetrain Output

Motion ROS Node - After cascading down all of the layers, the motion commands were output to the robot using this node

```
//ROS libs and msgs
#include <ros/ros.h>
#include <geometry_msgs/Twist.h>
```

```

// User libs and msgs
#include "yeet_msgs/move.h"

// Movement message
geometry_msgs::Twist twist;

//ROS Publishers
ros::Publisher teleop_pub;

void movementCallback(const yeet_msgs::move::ConstPtr& move_event)
{
    twist.linear.x = move_event->drive;
    twist.angular.z = move_event->turn;
    teleop_pub.publish(twist);
}

/**
 * @brief Main method
 *
 * @param argc Number of args
 * @param argv Args into the executable
 * @return int Exit code
 */
int main(int argc, char **argv)
{
    // Start the node
    ros::init(argc, argv, "motion_node");

    // Set up the node to handle motion
    ros::NodeHandle motion_node;

    // Subscribe to topics
    ros::Subscriber movement_sub = motion_node.subscribe(
        motion_node.resolveName("/yeet_mergency/human_control"), 10, &
            ↪ movementCallback);

    // Publish to the turtlebot's cmd_vel_mux topic
    teleop_pub = motion_node.advertise<geometry_msgs::Twist>(motion_node.
        ↪ resolveName("/cmd_vel_mux/input/teleop"), 10);

    ros::spin();

    return 0;
}
```

F ROS Implementation

F.1 Launch Files

Launch File - By running this launch file, alongside the required turtlebot bringup launch files (minimal and 3dsensor) the program will run. The keyboard needs to be run separately using rosrun

```
<launch>
    <!-- Planning -->
    <node pkg="yeet_planning" type="d_star_node" name="d_star_node" output="
        ↪ screen"/>

```

```
    <!-- Navigation -->
    <node pkg="yeet_nav" type="nav_node" name="nav_node" output="screen"/>
    <node pkg="yeet_nav" type="obstacle_node" name="obstacle_node" output="screen"/>

    <!-- Obstacle avoidance and Human Intervention -->
    <node pkg="yeet_mergency" type="collision_node" name="collision_node" output="screen"/>
    <node pkg="yeet_mergency" type="human_control_node" name="human_control_node" output="screen"/>

    <!-- Motion -->
    <node pkg="yeet_motion" type="motion_node" name="motion_node" output="screen"/>

    <!-- SLAM mapping -->
    <arg name="3d_sensor" value="$(optenv TURTLEBOT_3D_SENSOR astra)"/> <!-- kinect, asus_xtion_pro -->
    <include file="$(find turtlebot_navigation)/launch/includes/gmapping/$(arg 3d_sensor)_gmapping.launch.xml"/>

</launch>
```

F.2 Messages

Global Constants

```
int8 MAP_ROWS = 6
int8 MAP_COLS = 6
float64 SQUARE_SIZE = 0.62
```

Keyboard Message

```
char c
```

Motion Message

```
float64 drive
float64 turn
```

Navigation Status Message

```
bool goal
```

Node Message

```
int8 row
int8 col
bool is_obstacle
```

Obstacle Message

```
bool obstacle
```

F.3 Services

Target Node Service

```
---
yeet_msgs/node target
```