a. **A concise description of how the system works. This should be a couple of paragraphs, describing your overall approach at a conceptual level. Some questions to consider: What choices did you make in designing the program, and why?**

At a high level our program utilizes one node, sim_node, in order to create a simulator for a differential drive robot equipped with a lidar sensor. Our goal for this node was to be able to subscribe to any messages published on the topics “/vr” and “/vl” and use them to move a robot within our simulation. The robot’s position and physical parameters as well as the conditions of the environment such as the origin of the world frame, obstacle locations and initial pose are loaded from YAML files upon launching the program. These YAML files are loaded and handled by the sim_node which handles the creation of the map by publishing an occupancy grid on the /map topic, and the initial settings such as pose of the robot, which will later publish laser scans on the /scan topic as well as point clouds on the /laser_pc (for visualization).

One of the challenging points for this project was creating the handle_scan() function to generate laser scan data using the parameters for the specific robot such as angle max/min, range max/min, rate of scans, and the number of scans. The robot will then create a laser scan with these parameters and scan the environment using the get_map_lines() function that returns lines representing the obstacles in the surrounding environment. Initially we had trouble getting accurate laser scan data. The original approach was to iterate over all the laser beams and calculate the distance to all the obstacle points in the map. The minimum distance value would be stored for each laser beam and used to build the LaserScan message that is then published. However, this approach gave inconsistent results and oftentimes the pointclouds generated from the lidar scan data would be offset from the position/orientation of the robot. Instead after receiving a few hints in class from professor O’Kane we tried an approach with a ray-casting algorithm. With this function, we iterate over all the laser beam rays to calculate their intersection points with the map lines which is done in the ray_line_intersection() function. If an intersection point is found then the closest distance value for each laser beam is stored in an array, which is then used to build the LaserScan message that is published. This method gave us a much more consistent generation of pointclouds and also better accounted for the initial pose and orientation of the robot itself. We also accounted for error in that if an intersection point is found, some Gaussian noise is added to the distance value itself. On top of this, we created a probability of failure to randomly replace the distance value with a NaN value. The error variance and failure probability are specified within the YAML file for the different robots, so we process the specific robot YAML file to get those values.

Another challenging aspect of this project was calculating the changing position and orientation of the robot, while also designing it such that the robot would stop if it had not received instructions or changes to its wheel velocities within a long enough time. The move_robot() function within sim_node computes the new position and orientation of our robot based on its current velocities, its wheelbase distance and the elapsed time since the previous movement. The function will also take into account any possible collisions using the does_robot_collide_x() and does_robot_collide_y() functions in order to make sure the new calculated location isn't incorrect or impossible to reach due to obstacles in the way of the path. Finally the init_robot() function keeps track of the last time in which the function was called which represents the last time the velocities/positional data of the robot was updated. Through this value we were able to stop the robot from moving if it hasn't received new data within a certain period of time that we set at 1 second. Furthermore, this function creates a TransformStamped object which contains transform information, such as the translation and rotation.We set the translation information of the transform by specifying the x, y, and z coordinates of the robot's position, and the rotation information of the transform by getting the quaternion representation of the robot's orientation with the function get_quaternion() and setting the x, y, z, and w components of the transform afterwards. This allows us to then use sendTransform() with the TransformStamped object to publish the transform from the world frame to the base_link frame allowing other nodes to see the position and orientation of the robot within the world frame.

The process when designing the robot collision system was to first obtain all the obstacle coordinates and then build a function which stores all the lines formed by the points. These functions would convert the map into a form in which we can detect collisions based on robot position and velocity. To further modularize our code, we have one collision function which detects x axis collisions and one function which detects y axis collision. Essentially these functions can detect vertical or horizontal line collisions respectively. By splitting the functions, we can then decide if we want to set the x velocity or y velocity to zero in our move robot function.

b. A short answer to these questions: Did the results meet your expectations? Why or why not?

In our case, the results did meet our expectations. Our code manages to take in the YAML file data to initialize the environment and robot specifications before moving and analyzing its environment accurately. While we did meet with quite a few challenges such as handling laser scans in handle_scan(), accounting for collisions which we eventually split into two functions, does_robot_collide_x() and does_robot_collide_y(), and updating positional data with init_robot() and move_robot() we did eventually get the robot to function as intended. In this way,

we would say our expectations were met although we didn’t meet them in the way we initially expected. This is especially the case for handle_scan() where we ended up creating the ray_line_intersection() function that iterates over all the laser beam rays to calculate their intersection points with the map lines instead of our initial approach to iterate over all the laser beams and calculate the distance to all the obstacle points in the map. Therefore, we would say that while we did meet our expectations, it took some adaptation in our approach to the project in order to meet the requirements in a satisfactory manner.