Two-Dimensional Environment Mapping: Mobile Robots Within Unknown Three-Dimensional Spaces

Adam Hubble
P17175774

Introduction

Described as the purpose and criteria of this interim assessment, in use of the Robot Operating System (ROS) platform and its featured TurtleBot Burger robot, it was anticipated that an application would be developed for enabling the robot to construct a two-dimensional map of its unknown, surrounding environment, and for displaying said map graphically, using online or offline approaches to visualisation.

Figure 1: TurtleBot 3 family model, burger [1].

Mapping could have been addressed by a set of coordinates, a set of line segments or via an occupancy grid, which was the targeted method for map generation, given its vaster and more rewarding complexity for populating a robot's environment. Moreover, in accordance with the visualisation technique nominated for displaying the graphical output of the map, the ROS Visualization (RVIZ) tool was applicated for its immediate availability and support for online map generation, within a three-dimensional space; this also aided with interpreting the robot's orientation and position in real-time for debugging purposes. In the proceeding passages of this document, the techniques used to both construct and display a map of the robot's environment are detailed, as well, are relevant evaluations and closer appeals to the applications implementation, programmatically.

Map Construction

As already acknowledged, the nominated approach to mapping the robot's environment is achieved via occupancy grid map, which is "used to represent a robot workspace as a discrete grid" [2] of probabilities, where "each cell in the occupancy grid has a value representing the probability of the occupancy of that cell". Being "highly accurate" [3] as well as "fast because each grid cell is updated for each single observation based on the independent cell assumption", occupancy grid maps are assumed "accurate and reliable robotic maps" that are capable within visualising and assisting with robot to environment interaction; thus, the nomination of an occupancy grid for this application was deemed suitable, especially given that they're "widely used with range sensors such as laser scanners", which the TurtleBot Burger robot equips [1].

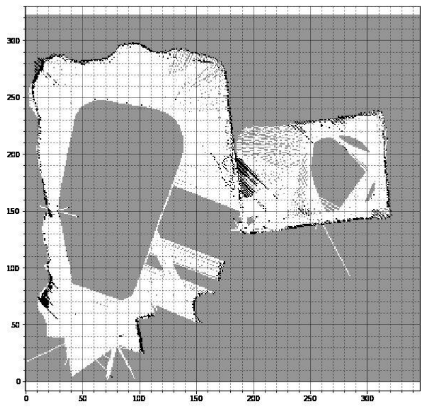


Figure 2: Occupancy grid, representing the map of multiple spaces combined into one common map [4].

Given their application within the field of probabilistic mapping, to orchestrate the probability of occupancy of each cell comprising the grid, recursive Bayesian inference [5] is applicated to address Markov's assumption, which states that the "future is independent of the past given the present". Whereby, from the implementation of recursive Bayesian inference, provides a "mechanism of computing the new estimate recursively from the old estimate and the new measurement", also renowned as the probability of interest or occupancy. This form of Bayesian Theorem is necessary for catering for "dynamical properties" [6] of a mobile robot's environment, that cause and effect the parameters determining the estimation of each cell's occupancy to "change with time"; particularly, this is acknowledged within the field of "autonomous robot navigation" where objects in an environment are subject to movement, which demands that their positions within a given map are updated accordingly. Otherwise, robot to environment interaction would prevail to be inaccurate and thereby unreliable in dynamic conditions.

$$\begin{aligned} p(x_t|z_{1:t}) &= \frac{p(z_t|x_t, z_{1:t-1})p(x_t|z_{1:t-1})}{p(z_t|z_{1:t-1})} \\ &= \frac{p(z_t|x_t)p(x_t|z_{1:t-1})}{p(z_t|z_{1:t-1})} \end{aligned}$$

Figure 3: Recursive Bayesian inference (estimation) mathematical notation [5].

For calculating all, prior, updated, and posterior probabilities of the Theorem and for resultingly populating the occupancy grid, recursive Bayesian inference relies upon the 'LaserScan' and 'Odometry' topics for formulating a “sensor model” [7], that enables “the interaction between the occupancy state of each grid cell and each sensor measurement” to be achieved. With reference to the ‘LaserScan’ topic, a subscriber is instantiated to listen to and return all laser sensor data that is broadcast to the topic; given that the TurtleBot Burger robot features a “360-degree planar lidar” [1], the distance measurement to detected objects at every one-degree angle within the range, can be known and utilised “to do SLAM and autonomous navigation out of the box”. Similarly, a subscriber is also instantiated to listen to and return all odometry data that is broadcast to the ‘Odometry’ topic, where the robot’s position and orientation can be realised within a local coordinate space.

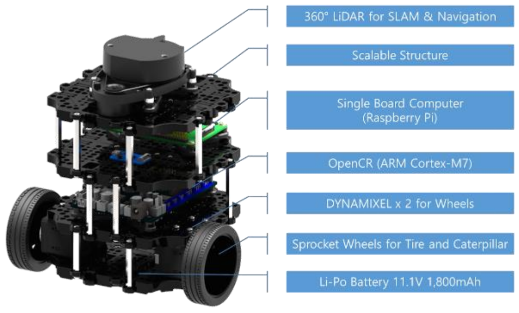

360° LiDAR for SLAM & Navigation
Scalable Structure
Single Board Computer (Raspberry Pi)
OpenCR (ARM Cortex-M7)
DYNAMIXEL x 2 for Wheels
Sprocket Wheels for Tire and Caterpillar
Li-Po Battery 11.1V 1,800mAh

Figure 4: TurtleBot 3 family model, Burger, mechanical composition overview [1].

Collectively, the data associated with either of the topics mentioned can then be utilised to initially calculate the detected obstacles position in local coordinate space, before being retargeted into global coordinate space; this enables the procedural generation of the robot’s environment, via occupancy grid map, to be conducted accurately with reference to scaling and with correspondence to the visualisation of the environment, live. Upon obtaining the global position of a detected obstacle, the cells of the grid that intersect the direction between the detecting laser sensor and the detected object can then be probabilistically interpolated by the sensor model, using the probability sum offered by recursive Bayesian inference. This in essence populates and defines the occupancy of obstacles in the grid map.

Fundamental to the sensor model designed for this application, Bresenham’s line algorithm, an “efficient method” [8] used for “scan converting a line”, is applicated for determining “all the intermediate points” [9] in the occupancy grid that intersect the position of the robot and detected obstacle; this enables a line to be drawn that linearly connects the positions. Operationally, Bresenham’s line algorithm functions within a bounding box, where each intersecting cell is iteratively discovered parallel to the intersecting line being generated by utilising a series of “integer addition, subtractions, and multiplication operations”, for perturbing along the ‘X’ and ‘Y’ axes of the grid, incrementally. The algorithm is invoked per laser sensor that detects an obstacle and is conditioned by the ‘LaserScan’ topic data, in which the distance reading value returned by the corresponding sensor will be smaller than the value of infinity (default value), if and only if an obstacle has been detected within or at the maximal range supported by the sensor.

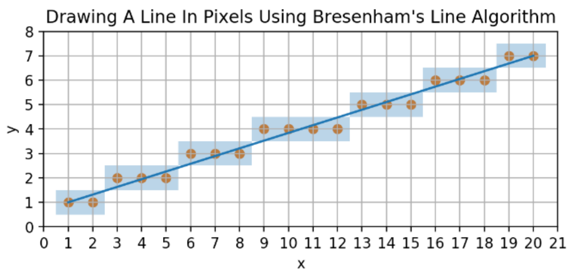


Figure 5: Bresenham's line algorithm visualisation, line generation from point (1, 1) to point (20, 7). Blue cells present to be the cells of intersection, as accompanied by orange-coloured circles, representing each cells origin [11].

Inspired by the basic sonar model [10], recursive Bayesian inference and Bresenham’s line algorithm are only invocated for cells that intersect the front-ten-most laser sensors of the robot, at which the mapping capability proves to provide higher degrees of fidelity and clarity, when compared to the utilisation of additional sensors. Whereby, upon applicating more sensors the apparency of erroneous data and noise increases in the occupancy grid calculations, given that the robots orientation continually adjusts with its angular traversal (wandering) and stationary turning (avoidance) behaviours, when being actuated. A ten-degree cone of vision for the robot caters for map construction relative to the front-most facing direction of the robot, which is comparatively more accurate and computationally inexpensive, however, map generation expectedly becomes significantly more time consuming.

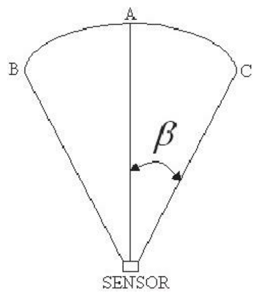


Figure 6: Cone of vision visualisation, where A represents the heading of the robot at zero-degrees, B represents the minimum sensor offset considered for the cone at five-degrees and C represents the maximum sensor offset considered for the cone at three-hundred-and-fifty-four-degrees [12]; beta represents the angular offset of each side of the cone from the robots heading.

Proceeding from gathering all the “integer coordinates” [9] of the cells intersecting the robot and obstacles position, the angular offset to each cell’s vertices can then be calculated, for determining the minimum and maximum error in direction, that represents the closest and furthest vertices from the line that bisects the centre of both the robot and obstacle cells. This measurement alongside the difference between the sensors offset and the direction or line generated between the robot and obstacles position, can be used within a divisive operation to obtain a probability of occupancy. Given this relation, the corresponding cells occupancy can be updated using recursive Bayesian inference by passing the resultant value of the operation mentioned; cells dispersed further away from the line of bisection resultingly have lower probabilities of being occupied, this is determined by being more spatially distant from the obstacle detected.

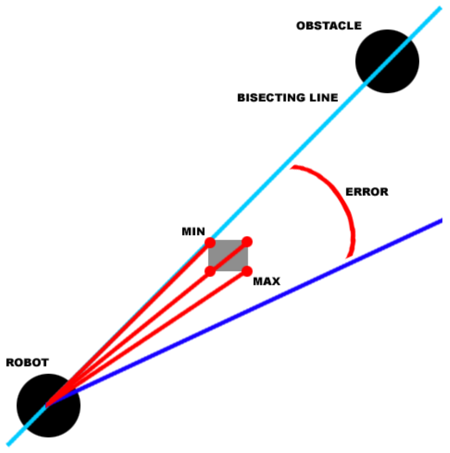
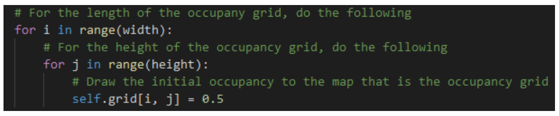


Figure 7: Probability calculation visualisation, demonstrating a cell and its vertices that constitute to the minimum and maximum error in direction, from the line of bisection.

Software Implementation

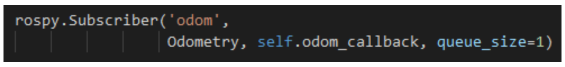
For its initialisation, the occupancy grid map is instantiated as a two-dimensional integer array variable, which is used to address the 'X' and 'Y' planes of the robot's environment that enables a two-dimensional map to be constructed and visualised from. Proceeding from the grid's instantiation, each of its cells are initialised to the value of '0.5', representing a neutral and uncertain probability of occupancy; this is achieved by implementing a nested 'for-loop' within the constructor method of the 'Map' class, which is used to iteratively set each cells value to '0.5' in the array, via indexing.



```
# For the length of the occupany grid, do the following
for i in range(width):
    # For the height of the occupancy grid, do the following
    for j in range(height):
        # Draw the initial occupancy to the map that is the occupancy grid
        self.grid[i, j] = 0.5
```

Figure 8: Code listing, visualising the nested for-loop for initialising the occupancy grid map, located within the constructor method of the Map class.

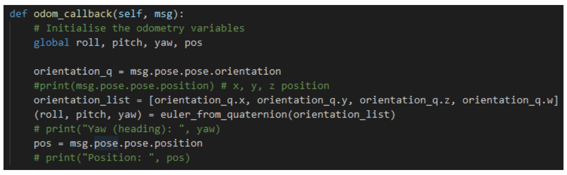
In support of the appliance of odometry data associated within the TurtleBot Burger robot, the constructor method within the 'Mapper' class was modified to facilitate a subscriber object, that subscribes to the 'Odometry' topic to access the position and orientation data of the robot.



```
rospy.Subscriber('odom',
                 Odometry, self.odom_callback, queue_size=1)
```

Figure 9: Code listing, visualising the instantiation of the 'Odometry' topic subscriber object, located within the constructor method of the Mapper class.

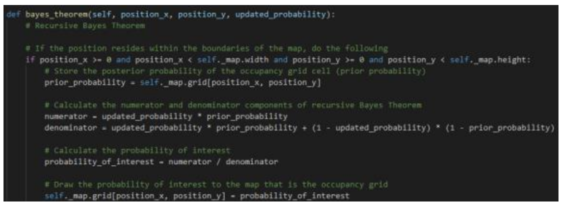
Accompanying this declaration, a method namely 'odom_callback()' was also implemented within the 'Mapper' class, as per name, the method functions to call back the odometry data of the 'Odometry' topic, for which is then stored as a series of globally declared variables that can be accessed throughout all other methods comprising the 'Mapper' class. Within said method, the orientation of the robot is stored globally before being casted from quaternion to Euler standard, thus enabling the robots heading measure to be understood relative to yaw. The robot's position is also stored globally but the values of the corresponding array variable are not required to be transformed.



```
def odom_callback(self, msg):
    # Initialise the odometry variables
    global roll, pitch, yaw, pos

    orientation_q = msg.pose.pose.orientation
    #print(msg.pose.pose.position) # x, y, z position
    orientation_list = [orientation_q.x, orientation_q.y, orientation_q.z, orientation_q.w]
    (roll, pitch, yaw) = euler_from_quaternion(orientation_list)
    # print("Yaw (heading): ", yaw)
    pos = msg.pose.pose.position
    # print("Position: ", pos)
```

Figure 10: Code listing, visualising the 'odom_callback()' method declaration, located within the Mapper class.

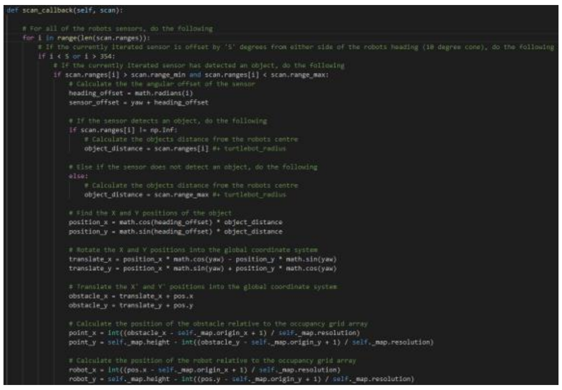
For the implementation of recursive Bayes Inference, a separate method namely 'bayes_theorem()' was implemented within the 'Mapper' class too, which enables the environment that the robot is subjected to, to be represented accurately and efficiently as a series of probabilities. As acknowledged prior, this is achieved by utilising the updated and prior probability values calculated for each cell comprising the grid. Showcased by the figure below, the method also incorporates a conditional statement to ensure that the probability of a cell is only updated if said cell resides within the boundaries of the map space; this is purposed for the prevention of error encounters. The probability of each cell is updated within this method, where it is consequently invocated in compliance with obstacle detections and found intersecting cells; the value of each cell is set to the probability of interest or occupancy calculated by the arithmetic of the theorem presented [5].



```
def bayes_theorem(self, position_x, position_y, updated_probability):
    # Recursive Bayes Theorem

    # If the position resides within the boundaries of the map, do the following
    if position_x >= 0 and position_x < self._map.width and position_y >= 0 and position_y < self._map.height:
        # Store the posterior probability of the occupancy grid cell (prior probability)
        prior_probability = self._map.grid[position_x, position_y]

        # Calculate the numerator and denominator components of recursive Bayes Theorem
        numerator = updated_probability * prior_probability
        denominator = updated_probability * prior_probability + (1 - updated_probability) * (1 - prior_probability)

        # Calculate the probability of interest
        probability_of_interest = numerator / denominator

        # Draw the probability of interest to the map that is the occupancy grid
        self._map.grid[position_x, position_y] = probability_of_interest
```

Figure 11: Code listing, visualising the 'bayes_theorem' method declaration, located within the Mapper class.

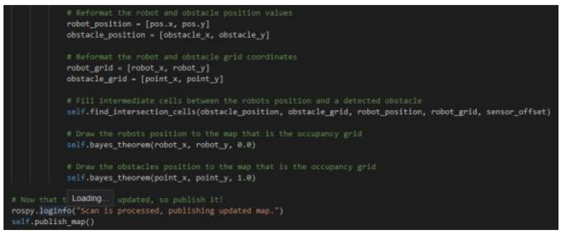
Advancing from the recursive Bayes theorem method, the ‘scan_callback()’ method located within the ‘Mapper’ class preliminarily contains the functionality required to translate the positions of detected obstacles from local to global coordinate spaces, via rotation matrix and translation operations for occupancy grid indexing purposes. For the premise of the function, each sensor comprising the “360-degree planar lidar” [1] is iterated through via ‘for-loop’, however, as previously acknowledged only the front-ten-most sensors are considered for operation, which creates the cone of vision for the robot discussed prior; this is conditionally upheld by using an ‘if-statement’ declaration. Noticeably, if the iterated sensor does not detect an object the sensors reading value is assigned the maximum range supported, for assigning probabilities to the cells that obstacles do not occupy when an obstacle is not detected; this enables the mapping procedure to be hastened within its generation, as cell values can be adjusted regardless of whether an object is detected or not.



```
def scan_callback(self, scan):

    # For all of the robots sensors, do the following
    for i in range(len(scan.ranges)):
        # If the currently iterated sensor is offset by '5' degrees from either side of the robots heading (10 degree cone), do the following
        if i < 5 or i > 354:
            # If the currently iterated sensor has detected an object, do the following
            if scan.ranges[i] > scan.range_min and scan.ranges[i] < scan.range_max:
                # Calculate the the angular offset of the sensor
                heading_offset = math.radians(i)
                sensor_offset = yaw + heading_offset

                # If the sensor detects an object, do the following
                if scan.ranges[i] != np.Inf:
                    # Calculate the objects distance from the robots centre
                    object_distance = scan.ranges[i] #+ turtlebot_radius

                # Else if the sensor does not detect an object, do the following
                else:
                    # Calculate the objects distance from the robots centre
                    object_distance = scan.range_max #+ turtlebot_radius

                # Find the X and Y positions of the object
                position_x = math.cos(heading_offset) * object_distance
                position_y = math.sin(heading_offset) * object_distance

                # Rotate the X and Y positions into the global coordinate system
                translate_x = position_x * math.cos(yaw) - position_y * math.sin(yaw)
                translate_y = position_x * math.sin(yaw) + position_y * math.cos(yaw)

                # Translate the X' and Y' positions into the global coordinate system
                obstacle_x = translate_x + pos.x
                obstacle_y = translate_y + pos.y

                # Calculate the position of the obstacle relative to the occupancy grid array
                point_x = int((obstacle_x - self._map.origin_x + 1) / self._map.resolution)
                point_y = self._map.height - int((obstacle_y - self._map.origin_y + 1) / self._map.resolution)

                # Calculate the position of the robot relative to the occupancy grid array
                robot_x = int((pos.x - self._map.origin_x + 1) / self._map.resolution)
                robot_y = self._map.height - int((pos.y - self._map.origin_y + 1) / self._map.resolution)
```

Figure 12: Code listing, visualising the preliminary section of the 'scan_callback()' method, located within the Mapper class.

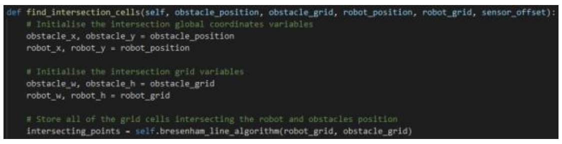
Proceeding from the functionality abovesaid, the method internally invocates the ‘bayes_theorem()’ method, for both obstacle detections and the position of the robot, where the values one (occupied) and zero (not occupied) are passed respectively. Moreover, for addressing the cells intersecting the position of the robot and detected obstacle, another method namely ‘find_intersection_cells()’ is invocated; this method is passed a series of variables local to the ‘scan_callback()’ method, concerning the cartesian coordinates and indices of the robot and obstacle positions relative to the occupancy grid, as well as the offset calculated for the iterated sensor, from the heading of the robot.



```
                # Reformat the robot and obstacle position values
                robot_position = [pos.x, pos.y]
                obstacle_position = [obstacle_x, obstacle_y]

                # Reformat the robot and obstacle grid coordinates
                robot_grid = [robot_x, robot_y]
                obstacle_grid = [point_x, point_y]

                # Fill intermediate cells between the robots position and a detected obstacle
                self.find_intersection_cells(obstacle_position, obstacle_grid, robot_position, robot_grid, sensor_offset)

                # Draw the robots position to the map that is the occupancy grid
                self.bayes_theorem(robot_x, robot_y, 0.0)

                # Draw the obstacles position to the map that is the occupancy grid
                self.bayes_theorem(point_x, point_y, 1.0)

    # Now that t Loading... updated, so publish it!
    rospy.loginfo("Scan is processed, publishing updated map.")
    self.publish_map()
```

Figure 13: Code listing, visualising the secondary section of the 'scan_callback()' method, located within the Mapper class.

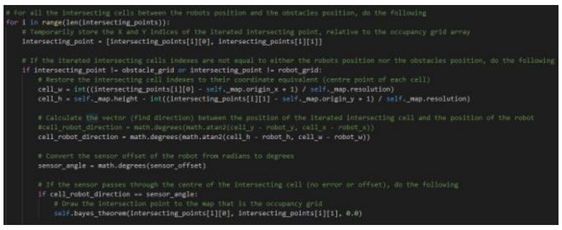
In continued mention of the ‘find_intersection_cells()’ method, the function initially invocates Bresenham’s line algorithm for determining the cells in the grid that intersect the positions of the robot and the detected obstacle, relative to the facing direction of the iterated laser sensor; this is achieved via functional invocation, where the ‘bresenham_line_algorithm()’ method is invocated.



```
def find_intersection_cells(self, obstacle_position, obstacle_grid, robot_position, robot_grid, sensor_offset):
    # Initialise the intersection global coordinates variables
    obstacle_x, obstacle_y = obstacle_position
    robot_x, robot_y = robot_position

    # Initialise the intersection grid variables
    obstacle_w, obstacle_h = obstacle_grid
    robot_w, robot_h = robot_grid

    # Store all of the grid cells intersecting the robot and obstacles position
    intersecting_points = self.bresenham_line_algorithm(robot_grid, obstacle_grid)
```

Figure 14: Code listing, visualising the preliminary section of the 'find_intersection_cells()' method, located within the Mapper class.

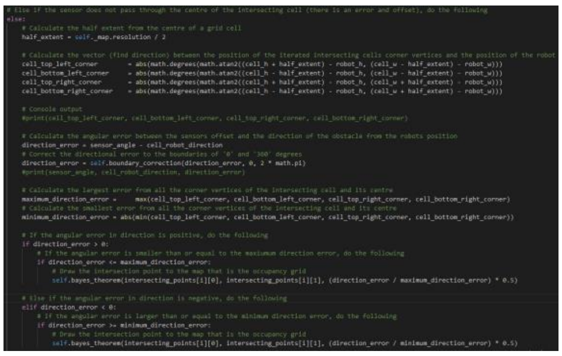
Upon the intersecting cells being computed and returned by the method as an integer array variable, each cell in said array is iterated through via indexing and a ‘for-loop’ for performing a series of operations relevant to the calculation of the updated probability, that is later passed to the ‘bayes_theorem()’ method. Initially, the direction (vector) from the robot to the iterated cell in the grid is calculated via arctangent operation and is used to condition and determine whether the detecting sensor is angularly offset from the direction calculated. Given the scenario that the sensor presents no offset from the calculated direction, the probability passed to the ‘bayes_theorem()’ method is decided as zero, inferring no occupancy; this is sensible to assume, provided that no obstacle can be detected between the corresponding sensor of the robot and the obstacle already detected.



```
# For all the intersecting cells between the robots position and the obstacles position, do the following
for i in range(len(intersecting_points)):
    # Temporarily store the X and Y indices of the iterated intersecting point, relative to the occupancy grid array
    intersecting_point = [intersecting_points[i][0], intersecting_points[i][1]]

    # If the iterated intersecting cells indexes are not equal to either the robots position nor the obstacles position, do the following
    if intersecting_point != obstacle_grid or intersecting_point != robot_grid:
        # Restore the intersecting cell indexes to their coordinate equivalent (centre point of each cell)
        cell_w = int((intersecting_points[i][0] - self._map.origin_x + 1) / self._map.resolution)
        cell_h = self._map.height - int((intersecting_points[i][1] - self._map.origin_y + 1) / self._map.resolution)

        # Calculate the vector (find direction) between the position of the iterated intersecting cell and the position of the robot
        #cell_robot_direction = math.degrees(math.atan2(cell_y - robot_y, cell_x - robot_x))
        cell_robot_direction = math.degrees(math.atan2(cell_h - robot_h, cell_w - robot_w))

        # Convert the sensor offset of the robot from radians to degrees
        sensor_angle = math.degrees(sensor_offset)

        # If the sensor passes through the centre of the intersecting cell (no error or offset), do the following
        if cell_robot_direction == sensor_angle:
            # Draw the intersection point to the map that is the occupancy grid
            self.bayes_theorem(intersecting_points[i][0], intersecting_points[i][1], 0.0)
```

Figure 15: Code listing, visualising the secondary section of the 'find_intersection_cells()' method, located within the Mapper class.

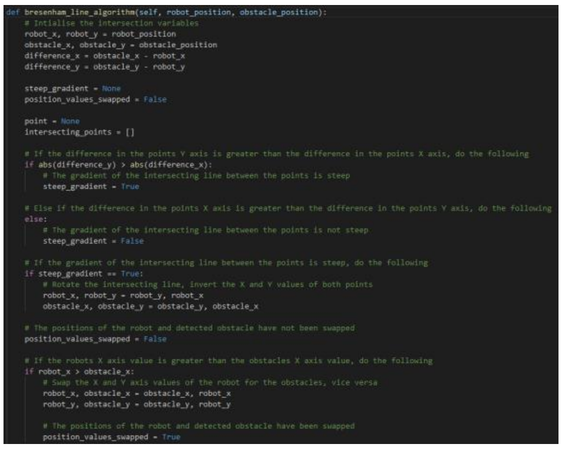
Alternatively, if the sensor presents an offset from the calculated direction, the direction from the robot to each of the iterated cell's vertices is then calculated, also via arctangent operation, for determining the maximum and minimum error in direction from the bisecting line, as discussed in previous sections. The error (difference) between the sensors angular offset and the direction from the robot to the iterated cell is then calculated also, simply by using a subtractive operation. Dependent on the sign of the error calculated, the direction error is divided by the maximum (if positive) or minimum (if negative) error calculated from either of the cell's vertices, in which provides a probability that depreciates with the furthest point of a cell becoming of increasing distance from the bisecting line.



```python
# Else if the sensor does not pass through the centre of the intersecting cell (there is an error and offset), do the following
else:
    # Calculate the half extent from the centre of a grid cell
    half_extent = self._map.resolution / 2

    # Calculate the vector (find direction) between the position of the iterated intersecting cells corner vertices and the position of the robot
    cell_top_left_corner       = abs(math.degrees(math.atan2((cell_h + half_extent) - robot_h, (cell_w - half_extent) - robot_w)))
    cell_bottom_left_corner    = abs(math.degrees(math.atan2((cell_h - half_extent) - robot_h, (cell_w - half_extent) - robot_w)))
    cell_top_right_corner      = abs(math.degrees(math.atan2((cell_h + half_extent) - robot_h, (cell_w + half_extent) - robot_w)))
    cell_bottom_right_corner   = abs(math.degrees(math.atan2((cell_h - half_extent) - robot_h, (cell_w + half_extent) - robot_w)))

    # Console output
    #print(cell_top_left_corner, cell_bottom_left_corner, cell_top_right_corner, cell_bottom_right_corner)

    # Calculate the angular error between the sensors offset and the direction of the obstacle from the robots position
    direction_error = sensor_angle - cell_robot_direction
    # Correct the directional error to the boundaries of '0' and '360' degrees
    direction_error = self.boundary_correction(direction_error, 0, 2 * math.pi)
    #print(sensor_angle, cell_robot_direction, direction_error)

    # Calculate the largest error from all the corner vertices of the intersecting cell and its centre
    maximum_direction_error =     max(cell_top_left_corner, cell_bottom_left_corner, cell_top_right_corner, cell_bottom_right_corner)
    # Calculate the smallest error from all the corner vertices of the intersecting cell and its centre
    minimum_direction_error = abs(min(cell_top_left_corner, cell_bottom_left_corner, cell_top_right_corner, cell_bottom_right_corner))

    # If the angular error in direction is positive, do the following
    if direction_error > 0:
        # If the angular error is smaller than or equal to the maximum direction error, do the following
        if direction_error <= maximum_direction_error:
            # Draw the intersection point to the map that is the occupancy grid
            self.bayes_theorem(intersecting_points[i][0], intersecting_points[i][1], (direction_error / maximum_direction_error) * 0.5)

    # Else if the angular error in direction is negative, do the following
    elif direction_error < 0:
        # If the angular error is larger than or equal to the minimum direction error, do the following
        if direction_error >= minimum_direction_error:
            # Draw the intersection point to the map that is the occupancy grid
            self.bayes_theorem(intersecting_points[i][0], intersecting_points[i][1], (direction_error / minimum_direction_error) * 0.5)
```

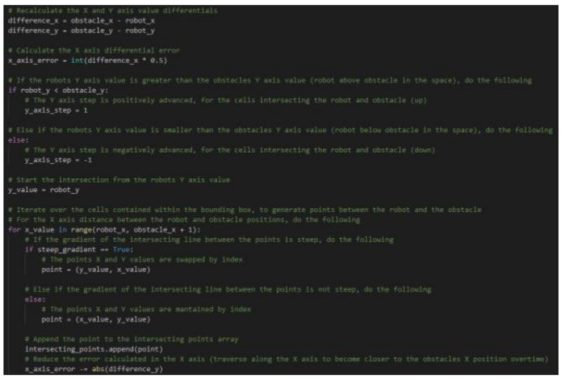
Figure 16: Code listing, visualising the tertiary section of the 'find_intersection_cells()' method, located within the Mapper class.

Lastly, as already mentioned and integrated as part of the 'find_intersection_cells()' method, the 'bresenham_line_algorithm()' method located within the 'Mapper' class purposes to incrementally establish all cells intersecting the positions of the robot and the detected obstacle. This is achieved by performing a series of perturbation cycles along the 'X' and 'Y' axes of the grid, to gradually displace nearer to the position of the obstacle, relative to the starting point of the robot whilst marginalising the error from the bisecting line.



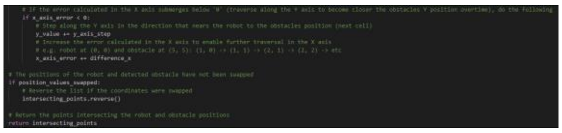
```python
def bresenham_line_algorithm(self, robot_position, obstacle_position):
    # Intialise the intersection variables
    robot_x, robot_y = robot_position
    obstacle_x, obstacle_y = obstacle_position
    difference_x = obstacle_x - robot_x
    difference_y = obstacle_y - robot_y

    steep_gradient = None
    position_values_swapped = False

    point = None
    intersecting_points = []

    # If the difference in the points Y axis is greater than the difference in the points X axis, do the following
    if abs(difference_y) > abs(difference_x):
        # The gradient of the intersecting line between the points is steep
        steep_gradient = True

    # Else if the difference in the points X axis is greater than the difference in the points Y axis, do the following
    else:
        # The gradient of the intersecting line between the points is not steep
        steep_gradient = False

    # If the gradient of the intersecting line between the points is steep, do the following
    if steep_gradient == True:
        # Rotate the intersecting line, invert the X and Y values of both points
        robot_x, robot_y = robot_y, robot_x
        obstacle_x, obstacle_y = obstacle_y, obstacle_x

    # The positions of the robot and detected obstacle have not been swapped
    position_values_swapped = False

    # If the robots X axis value is greater than the obstacles X axis value, do the following
    if robot_x > obstacle_x:
        # Swap the X and Y axis values of the robot for the obstacles, vice versa
        robot_x, obstacle_x = obstacle_x, robot_x
        robot_y, obstacle_y = obstacle_y, robot_y

        # The positions of the robot and detected obstacle have been swapped
        position_values_swapped = True
```

Figure 17: Code listing, visualising the preliminary section of the 'bresenham_line_algorithm()' method, located within the Mapper class.



```python
# Recalculate the X and Y axis value differentials
difference_x = obstacle_x - robot_x
difference_y = obstacle_y - robot_y

# Calculate the X axis differential error
x_axis_error = int(difference_x * 0.5)

# If the robots Y axis value is greater than the obstacles Y axis value (robot above obstacle in the space), do the following
if robot_y < obstacle_y:
    # The Y axis step is positively advanced, for the cells intersecting the robot and obstacle (up)
    y_axis_step = 1

# Else if the robots Y axis value is smaller than the obstacles Y axis value (robot below obstacle in the space), do the following
else:
    # The Y axis step is negatively advanced, for the cells intersecting the robot and obstacle (down)
    y_axis_step = -1

# Start the intersection from the robots Y axis value
y_value = robot_y

# Iterate over the cells contained within the bounding box, to generate points between the robot and the obstacle
# For the X axis distance between the robot and obstacle positions, do the following
for x_value in range(robot_x, obstacle_x + 1):
    # If the gradient of the intersecting line between the points is steep, do the following
    if steep_gradient == True:
        # The points X and Y values are swapped by index
        point = (y_value, x_value)

    # Else if the gradient of the intersecting line between the points is not steep, do the following
    else:
        # The points X and Y values are mantained by index
        point = (x_value, y_value)

    # Append the point to the intersecting points array
    intersecting_points.append(point)
    # Reduce the error calculated in the X axis (traverse along the X axis to become closer to the obstacles X position overtime)
    x_axis_error -= abs(difference_y)
```

Figure 18: Code listing, visualising the secondary section of the 'bresenham_line_algorithm()' method, located within the Mapper class.



```python
    # If the error calculated in the X axis submerges below '0' (traverse along the Y axis to become closer the obstacles Y position overtime), do the following
    if x_axis_error < 0:
        # Step along the Y axis in the direction that nears the robot to the obstacles position (next cell)
        y_value += y_axis_step
        # Increase the error calculated in the X axis to enable further traversal in the X axis
        # e.g. robot at (0, 0) and obstacle at (5, 5): (1, 0) -> (1, 1) -> (2, 1) -> (2, 2) -> etc
        x_axis_error += difference_x

# The positions of the robot and detected obstacle have not been swapped
if position_values_swapped:
    # Reverse the list if the coordinates were swapped
    intersecting_points.reverse()

# Return the points intersecting the robot and obstacle positions
return intersecting_points
```

Figure 19: Code listing, visualising the tertiary section of the 'bresenham_line_algorithm()' method, located within the Mapper class.

Evaluation

Throughout the development of the intended application, the occupancy grid map size, resolution, and origin was manipulated to suffice for more accurate depictions of the robot's environment; each parameters value was considered relative to the computational performance of the software platform itself, as extended resolutions and grid sizes were found to dramatically slow the simulation and the resultant pace of map generation. Moreover, through deferring from the original parameter values, the map in its submittable state is seen to be represented with a viable orientation, scale, and transformation.

Additionally, as speculated for the cone of vision or range of sensors considered for detecting obstacles, originally the cone of vision was vast with sixty-degrees of surveillance, which was well-suited to a time efficient approach to environment mapping but was also more prone to noisy occupancy calculations. Given this relation, the robots field of view (FOV) was marginalised over time, from the initial sixty-degrees of surveillance to the resultant ten-degrees. In effect of reducing the FOV of the robots sensing capability, the depiction of the robot's environment is noticeably sharper, where less noise is present within each of the representations, however, map generation is consequently more time consuming.

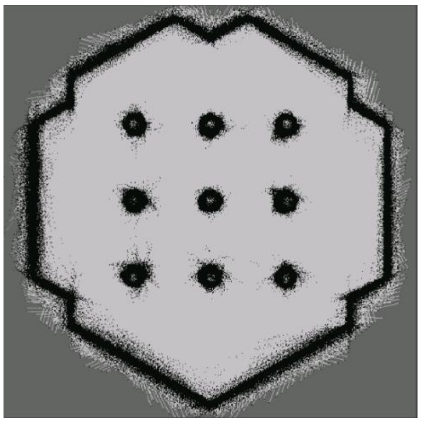


Figure 20: Occupancy grid map visualisation when the robot's cone of vision explores a sixty-degree angle of surveillance.

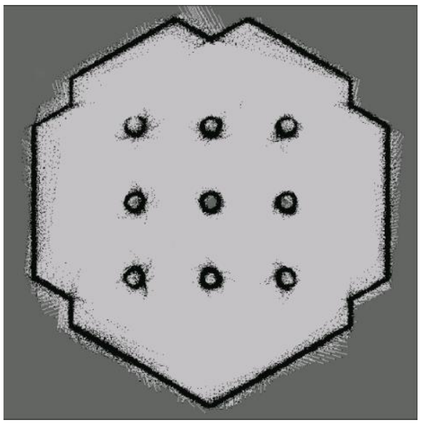


Figure 21: Occupancy grid map visualisation when the robot's cone of vision explores a thirty-degree angle of surveillance.

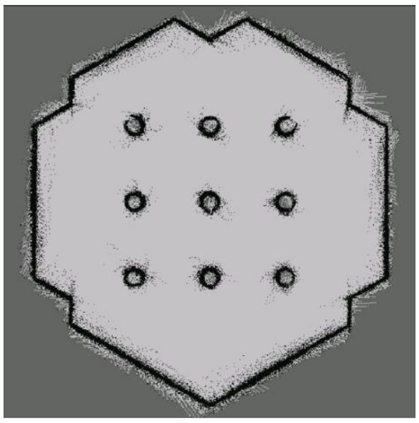


Figure 22: Occupancy grid map visualisation when the robot's cone of vision explores a twenty-degree angle of surveillance.

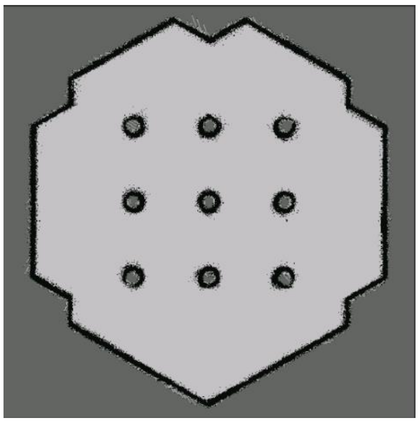


Figure 23: Occupancy grid map visualisation when the robot's cone of vision explores a ten-degree angle of surveillance.

Conclusions

In conclusion of the environment mapping task instructed for its undertaking, it is inevitable that a two-dimensional illustration of the robot's three-dimensional surroundings is feasible, through the presented implementation of an occupancy grid map which employs recursive Bayesian inference, to achieve an accurate and efficient approximation of an unknown environment. As is noticed by the graphical output of the maps generated, occupancy grids are well-suited to producing clear and structurally visible depictions of environments, for which the final configuration submitted presents minimalistic noise quantities.

Now acknowledging the projects completed state, throughout the development of the application, my awareness of the libraries and facilities available within the Python programming language has both been reinforced and advanced. Similarly, as previously inexperienced within the ROS platform and more specifically within the 'TheConstructSim' platform, I have inevitably become competent in navigating, operating and applicating the tools available in the software for the project's completion; this knowledge will undoubtedly prevail useful for upcoming assessments and future professional practises.

Given that more time were to be available for the projects working, I believe that abstracting the functionality out into separate classes and files would prove to be increasingly more efficient than the submittable state provided; this would provide support for higher fidelities of resolution also, whilst reducing performance degradations. Also, I would like to have alternated the way in which each cells probability is calculated within he grid, for reducing computational complexity and for adhering to generic workings and applications of the Bayesian Theorem.

References

[1] Ackerman, E. and Guizzo, E. (2017) Hands-on with TurtleBot 3, a Powerful Little Robot for Learning ROS. [Weblog] *IEEE Spectrum*. 2nd May. Available from: https://spectrum.ieee.org/automaton/robotics/robotics-hardware/review-robotis-turtlebot-3 [Accessed: 07/04/21].

[2] MathWorks (2021) *Occupancy Grids*. [Online] MathWorks. Available from: https://uk.mathworks.com/help/robotics/ug/occupancy-grids.html [Accessed: 07/04/21].

[3] Kim, S. and Kim, J. (2014) Recursive Bayesian Updates for Occupancy Mapping and Surface Reconstruction. In: *Proceedings of Australasian Conference on Robotics and Automation. The University of Melbourne, Melbourne, Australia, December 2014*. Australia: IEEE, pp. 1-8.

[4] Sundram, J. and Nguyen, H.D.V. and Soh, G.S. and Wood, K.L. (2018) Development of a Miniature Robot for Multi-robot Occupancy Grid Mapping. In: *IEEE International Conference on Advanced Robotics and Mechatronics (ICARM). Singapore, July 2018*. Singapore: IEEE, pp. 414-419.

[5] Li, H. (2014) A Brief Tutorial on Recursive Estimation with Examples from Intelligent Vehicle Applications (Part I): Basic Spirit and Utilities. [Online]. Available from: https://hal.archives-ouvertes.fr/hal-01015148/document [Accessed: 07/04/21].

[6] Bergman, N. (1999) Recursive Bayesian Estimation Navigation and Tracking Applications. [Online]. Available from: http://www.control.isy.liu.se/research/reports/Ph.D.Thesis/PhD579.pdf [Accessed: 07/04/21].

[7] Robbiano, C. and Chong, E.K.P. and Sadjadi-A.M.R. and Scharf, L.L. and Pezeshki, A. (2020) Bayesian Learning of Occupancy Grids. *IEEE Transactions on Intelligent Transportation Systems*. [Online]. Available from: https://ieeexplore.ieee.org/document/9190014/authors#authors [Accessed: 07/04/21].

[8] Java Point (2018) *Bresenham's Line Algorithm*. [Online] Java Point. Available from: https://www.javatpoint.com/computer-graphics-bresenhams-line-algorithm [Accessed: 07/04/21].

[9] GeeksforGeeks (2021) *Bresenham's Line Generation Algorithm*. [Online] GeeksforGeeks. Available from: https://www.geeksforgeeks.org/bresenhams-line-generation-algorithm/ [Accessed: 07/04/21].

[10] Ivanjko, E. and Petrovic, I. and Brezak, M. (2009) Experimental Comparison of Sonar Based Occupancy Grid Mapping Methods. [Online]. Available from: https://www.researchgate.net/publication/293076408_Experimental_Comparison_of_Sonar_Based_Occupancy_Grid_Mapping_Methods [Accessed: 07/04/21].

[11] Mbedded.ninja (2019) Bresenham's Line Algorithm. [Weblog] *mbedded.ninja*. 3rd January. Available from: https://blog.mbedded.ninja/programming/algorithms-and-data-structures/bresenhams-line-algorithm/ [Accessed: 07/04/21].

[12] Torres, C.R. and Abe, J.M. and Torres-L.G. Filho, J.I.D.S. (2011) Autonomous mobile robot Emmy III. [Online]. Available from: https://www.researchgate.net/publication/221918582_Autonomous_mobile_robot_Emmy_III [Accessed: 07/04/21].