FLIP TASK ALLOCATION FOR ROBOT PATH COVERAGE

A Thesis

by

STEVEN TYLER LONGA

Submitted to the Graduate and Professional School of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

August 2023

Major Subject: Engineering Technology

ABSTRACT

The usage of multi-robot systems to complete monotonous yet complex tasks has become increasingly popular. One such category is tasks that require the complete coverage of an area, such as the task of vacuuming. The undertaking of a complete coverage task by a singular mobile floor cleaning robot requires a minimum of path planning capabilities to prevent the recleaning of previously cleaned areas. When more than one robot is utilized to complete the same coverage task, there must be some form of global strategy implemented that can aid the multi-robot system in reducing the amount of coverage overlap, idle time, and overall time required to complete the vacuuming task. Such global strategies often utilize a method of decomposing the larger task into smaller subtasks which are then allocated among the number of robots within the system. However, many of these strategies are either static in their task allocation or are based on a singular robot system to accomplish the complete coverage task. The algorithm for global strategy proposed in this thesis presents a methodology for utilizing the techniques of triangular mesh decomposition, Traveling Salesman Problem optimization, and dynamic flip task allocation for multiple floor cleaning robots.

DEDICATION

This thesis is dedicated to my dear mother Karen who bravely battled cancer until September 25, 2021. It was her unwavering support and encouragement that inspired me to pursue a master’s degree. In doing so, I hoped to provide her with a source of joy in envisioning the day she would witness her son proudly crossing the stage after enduring two challenging years of various cancer treatments. Although the journey through graduate school has been arduous, I did it Mom.

ACKNOWLEDGMENTS

I would like to thank my committee chair, Dr. Ufodike, and my committee members, Dr. Shell and Dr. Kuttolamadom, for their guidance and support throughout the course of this research.

I would also like express my gratitude to my friends, colleagues, and the department faculty and staff for making my time at Texas A&M University a great experience.

Lastly, a heartfelt thanks to my father, brother, and family for their constant encouragement and love.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supervised by a thesis committee consisting of Dr. Ufodike and Dr. Kuttolamadom of the Department of Engineering Technology and Industrial Distribution and Dr. Shell of the Department of Computer Science and Engineering.

All other work conducted for the thesis was completed by the student with the help and guidance of Danyal Ansarid, Brey Caraway, and Chukwubuikem Ewelike.

Funding Sources

Research was supported by funding from Dr. Ufodike.

NOMENCLATURE

LiDAR	Light Detection and Ranging
PGM	Portable Gray Map
TSP	Traveling Salesman Problem

TABLE OF CONTENTS

LIST OF FIGURES

Page

LIST OF TABLES

Page

LIST OF CODE EXAMPLES

Page

1. INTRODUCTION

1.1. Background

The task of vacuuming is one of necessary importance to the cleanliness of a building and its floors yet the act of completing such a task can be perceived as mundane given the repetitiveness and boredom which accompany it. To solve this problem, companies such as iRobot and Roborock offer commercially available solutions that can complete the vacuuming task. Released in July of 2019 [1] and April of 2022 [2] respectively, iRobot's Roomba s9+ and Roborock's S7 MaxV Ultra offer state-of-the-art mobile vacuum cleaning systems that can autonomously keep a household's floor clean with little user intervention.

To perform their cleaning operations, such systems employ a variation or combination of exploratory and complete coverage path planning algorithms. These exploratory algorithms allow the robot to understand and make sense of its workspace by creating a detailed map of its surroundings utilizing various onboard sensors. While initially inefficient, the data collected by the exploratory algorithm is saved such that it can later be used by the complete coverage path planning algorithm for a more time and coverage efficient next cleaning. Systems such as those mentioned above typically employ a singular robot that goes about performing the task of cleaning the assigned work area. These do not exhibit multi-robot system functionality as these systems were designed from the ground up to be utilized as standalone devices.

While the need for multi-robot functionality may not be a priority in a household or small office setting as many commercially available systems are intended for, the ability to deploy multiple mobile floor cleaning robots in larger settings such as but not limited to hotels, warehouses, and schools gives rise to the need for multi-robot functionality which can help free custodians and housekeepers from the mundane task of vacuuming long hallways and multiple rooms within a shift. Despite the usefulness of exploratory algorithms when employed by a single robot to clean an assigned work area as used by the commercially available mobile floor cleaning robots, the problem of strategy arises when multiple robots are tasked with cleaning the same work area.

The scope of strategy in the application of vacuuming comes down to *how* should the work area be cleaned. The primary factor for consideration in implementing a strategy is the number and size of the robots to be utilized. The size and number of robots utilized influence other aspects of strategy implementation such as the size, shape, and number of decomposed regions created within the work area. It is these generated decomposed regions that lead to the second factor for consideration in strategy implementation which is the determining of an optimal path to traverse the work area such that all decomposed regions are visited and cleaned by a robot. In a multi-robot system, a third factor of how the decomposed regions, now tasks, will be allocated among the number of available robots must also be considered when implementing a strategy as the method of task allocation can have an effect on the overall time and power efficiency in completing the cleaning task.

Strategy planning in multi-robot systems is approached in one of two ways: centralized and decentralized. In a centralized system, a single central computer commands each robot in accordance with the overall task assigned. Furthermore, robots within a centralized system do not exhibit individual intelligence but rather feature sensors and systems which record information about that particular robot including speed, heading, and orientation with respect to the working environment that is then relayed to the central computer for processing and further commands if necessary. By utilizing a centralized approach, the robots within the system operate predictably and can be used in various settings including organizing warehouses such as Amazon’s fleet of robotic pickers [3], commanding a fleet of 1,800 drones to perform mesmerizing light displays like the one performed for the Tokyo 2020 Olympics opening ceremony [4], and even perform search and rescue operations without human intervention [5].

With regards to decentralized systems, this approach is often found in robot swarms and is described by [6] as a system consisting of autonomous robots cooperatively working together where each robot features local sensing and communication capabilities but lacks centralized control or access to global information. In short, the individual robots by themselves can perform a simple function but the overall goal of the swarm is only achieved when there are many robots involved in completing the assigned task. With respect to this research, the employing of a decentralized approach would be akin to having multiple robots wandering around aimlessly vacuuming and occasionally bumping into objects to then make a preprogrammed turn similar to the more dated commercially sold mobile floor cleaning robots. The robots would eventually clean the work area, but the time required and energy expended would be highly inefficient. While such an approach may not be best suited for a vacuuming application, researchers have been exploring potential uses as most decentralized robot swarms, especially large ones, are typically only found in research laboratories. One such example is the Kilobot swarm created in the lab of Radhika Nagpal which consists of over 1,000 individual robots that can self-organize into various shapes [7].

Given the saturation of the commercial market for non-multi-robot mobile floor cleaning systems, this research aims to provide a foundation for the development of a centralized method of strategy planning to be used by a fleet of N number of mobile floor cleaning robots in macro-scale cleaning applications such as large indoor spaces. It is within these large spaces where one or multiple commercially available floor cleaning robots would be inefficient in the amount of time required and the high potential for the overlapping of previously cleaned areas due to there being no employed multi-robot strategy for the cleaning process. Hence the need for implementing a strategy planning algorithm into a fleet of mobile vacuum cleaners which can lead to more efficient cleaning times, can help save battery power, and reduce the overlapping of already previously cleaned areas.

1.2. Problem Statement

The problem this research aims to solve is the implementation of a strategy for the dynamic task allocation of multiple robots in a complete coverage application such that the amount of time a robot is sitting idle after completing its assigned task is minimized when there are still tasks assigned to other robots yet to be completed.

1.3. Literature Review

Approaches to strategy planning vary from application to application. Such variations can be based on factors such as the complexity of the work environment, if the map of the work environment is already known, the number of robots planned to operate within the work environment, and the circumstances unique to the particular situation. Furthermore, the strategy employed to complete the assigned complete coverage tasks often operates behind the scenes of the path planning algorithm but plays a critical role in guiding the path planning algorithm through the work environment.

1.3.1. Environment Representation

One element of strategy planning is how will the robot's environment be represented. A typically utilized approach involves using a grid-based method to discretize the robot's environment into a two-dimensional grid of square cells. The size of the square cells dictates the resolution of the grid which can be modified as needed to suit the particular application. Each cell within the grid can then be assigned a value based on the state of that specific cell. Two common techniques employed in assigning cell values within the grid are discussed below.

Occupancy maps are one method a cell can be assigned a value. Within an occupancy map, each cell is given a value that is representative of the likelihood that the cell is occupied by an object. In most implementations, the lower the cell value the lower the likelihood that the cell is occupied while the higher the cell value the higher the likelihood that the cell is occupied. Đakulovic in both [8] and [9] implement an occupancy map where free cells are assigned a value of one while occupied cells are assigned a value of infinity. As the path planning algorithm attempts to navigate a robot that can be represented by a $\mathbf{7 \times 7}$ grid of cells, the strategy portion of the algorithm further assigns cells surrounding obstacles out to seven cells a higher value than the free cells. This assignment of higher values to obstacles and cells near obstacles is performed such that the path planning portion of the algorithm, which is based on a least-cost function that sums up all the cell values within the next possible location of the robot, is dissuaded from choosing those cells as the next robot location.

A second technique utilized to assign cell values is through the use of a neural network where each cell within the grid contains a value representative of the amount of neural activity present in that particular cell. A popular neural network approach takes inspiration from biology and is defined by the following computational model shown below in Equation 1 first proposed by Hodgkin and Huxley in 1952 [10].

$$C_m \frac{dV_m}{dt} = -\left(E_p + V_m\right) g_p + \left(E_{Na} - V_m\right) g_{Na} - \left(E_K + V_m\right) g_K \qquad \textbf{(1)}$$

By making a few adjustments to the proposed computational model such as setting $C_m = 1$ and substituting $A = E_p + V_m$, the resulting shunting equation first proposed by Grossberg [11] is shown below in Equation 2.

$$\frac{dx_i}{dt} = -Ax_i + (B - x_i)S_i^e(t) - (D + x_i)S_i^i(t) \qquad \textbf{(2)}$$

Through utilizing Grossberg’s shunting equation, each cell within the grid can then be assigned a neural activity value to represent if that cell is a viable next candidate for the path planning algorithm to move to or if that particular cell should be avoided. A typical application of a bio-inspired neural network (BINN) approach is assigning a positive value to unclean cells, a neutral value of zero to clean cells, and a negative value to occupied cells which includes obstacles and other robot locations. This is such that the path planning algorithm will be attracted to unclean cells exhibiting the highest neural activity values and will be repelled by occupied cells containing the lowest neural activity values. Implementations of a BINN approach can be found in Luo and Yang's works on an algorithm for implementing both single robot [12] and multi-robot [13] complete coverage strategies in an unknown environment. Sun et al. [14] utilizes an improved version of Luo and Yang's multi-robot approach in the complete coverage path planning for autonomous underwater vehicles and Luo further applies a BINN approach to the trajectory planning of an autonomous vehicle [15].

1.3.2. Environment Decomposition

A second concern regarding strategy planning is how will the work environment be decomposed into smaller areas that can then be tasked to an individual robot. Decomposition of the work environment can be performed in a multitude of ways however the main goal for each strategy is to divide the environment into subregions such that the union of the subregions is equal to that of the original environment.

One common method employed in the decomposition of an environment is that of Voronoi decomposition. As defined by [16], this method of decomposition utilizes a set of n generating points within the environment to then partition the environment into convex polygons such that every polygon contains exactly one generating point and that any point contained within a particular polygon is closest in distance to its generating point than to any other. The technique used to calculate the distance between any point within the polygon and its generating point will dictate the resulting shape of the polygon. Nair and Guruprasad [17] utilize a combination of two generalizations of Voronoi decomposition which are geodesic-distance-based and Manhattan-distance-based Voronoi decomposition for the multi-robot complete coverage of a known environment. Fu et al. [18] implemented an adaptive Voronoi decomposition method for the exploratory coverage of an unknown environment by multiple robots using a Divide and Conquer method. Hu et al. [19] performs a similar approach to the exploration of an unknown environment utilizing a deep reinforcement learning technique in conjunction with dynamic Voronoi decomposition which was later applied to a multi-robot complete coverage application [20]. While implementations of Voronoi decomposition are successful in achieving the complete coverage of an environment, the resulting polygons, also referred to as cells, oftentimes are irregular in shape and tend to vary widely in shape and area. A method for governing the resulting cell shapes is Centroidal Voronoi Tessellations (CVT) as presented in [21] where an iterative process utilizing a hybrid of Lloyd's and MacQueen's algorithms produces cells more uniform in shape and area.

Another commonly employed method is that of Morse decomposition. This method of decomposition involves sweeping a line defined by a Morse function through a work environment which is then either divided upon encountering an obstacle or recombined after the obstacle has been passed as described by [22]. Points at which the connectivity of the sweeping line changes as it moves through the work environment are referred to as "critical points" which are then used to determine the cell boundaries. Acar and Choset utilize Morse decomposition for the coverage of an unknown environment by a single robot with a robot-sized end effector [23] and later along with Lee presented a coverage method for a single robot with an end effector larger than the robot yet finite in size [24]. While Morse decomposition is a more general form of decomposition that can theoretically be applied to any n-dimensional space [22], Boustrophedon decomposition, introduced by Choset [25], occurs within the two-dimensional plane of the work environment and has been shown to be a specific case of Morse decomposition with a Morse function of $h(x, y) = x$ [26]. Boustrophedon decomposition in a complete coverage application was implemented in both [27] and [28] for a single robot. While Morse decomposition excels in environments with obstacles, it is dependent upon the presence of obstacles to produce partitioned areas given the nature of the sweeping line.

A method seldom employed is that of triangular mesh decomposition. Upon conducting this literary review, it was difficult to locate works that implemented this method of decomposition in a complete coverage application that partitions the environment into triangles of similar size. One such implementation by Oh et al. [29] proposed a method of decomposing the work environment into triangular cells with a width equal to the robot. The work environment would then be covered by a robot visiting the center of each triangle along its planned path using seven templates to perform navigation. While not utilized in the complete coverage sense, [30] and [31] utilize triangular mesh for the path planning of the shortest route from a start point to an endpoint. Due to the lack of research on utilizing triangular mesh decomposition for complete coverage applications, this research aims to fill this knowledge gap presented.

1.3.3. Environment Negotiation

A final matter with regard to strategy planning is how will the environment be negotiated by the number of available robots and in the most time-efficient manner possible. One method of traversing an environment is to simply allow the robots to go about path planning throughout the environment randomly. Known as the Random Walk method, this type of method is used in exploratory applications such as search and rescue where a large area needs to be covered promptly but not completely. Given the random nature of this strategy, the Random Walk method suffers from inefficiencies such as the repeated searching of already searched areas. To increase the search efficiency, [32] presents an improvement to the Random Walk method by having each robot adjust its step size dynamically. While the proposed improvements to the Random Walk method may enhance the efficiency of the amount of environment covered by the robots, this method falls short in a complete coverage application. The amount of time such a method would require to eventually achieve complete coverage of the environment renders it inefficient.

Another strategy that can be employed in environment negotiation, assuming the environment was previously decomposed into smaller regions, is the method of planning the shortest path which visits each region created by the decomposition only once. Known as the Traveling Salesman Problem (TSP), this problem relates to a salesman needing to visit a set of locations in the most efficient and thereby shortest path possible. The solution to this problem can be applied to a complete coverage application by replacing the list of locations the salesman needs to visit for points within the regions of the decomposed environment that the robot(s) must pass through. This technique is applied in the environmental negotiation strategy implemented in [33] by which TSP-based reinforcement learning optimization is utilized to generate an optimal path for how a singular robot should go about completely covering the work environment. The path generated by such an approach provides direction for how the robot should visit each decomposed region, or in this case tiles, which would suggest the existence of an ordered list detailing which tile to move to next as the robot negotiates the environment.

For the case of a single robot, each region can simply be visited by the robot along its path however when the number of robots is increased, this path must be divided among the number of robots to prevent the overlap of cleaned areas along the path. A method referred to as "Coverage with Route Clustering" is presented in [34] which was later modified for Dubins vehicles in [35] for which an optimal path, such as the path generated by a TSP algorithm, is split between multiple robots. In doing so, segments of the global path can be assigned to individual robots without overlap. By translating the regions to be visited within each segment to tasks, this research aims to implement a similar method as a means of allocating tasks between robots with the additional feature of flipping every other path segment. This is such that globally the robots negotiate the environment by working toward one another which is novel in the application of complete coverage.

1.4. Research Goals and Objectives

The goal of this research is to:

- Highlight how multiple robots with a global strategy can be used to clean a work area more quickly and efficiently when compared to a single robot or multiple robots without a global strategy cleaning the same work area.
- Demonstrate the decrease in idle time and overall cleaning time by dynamically reassigning a robot that has completed its assigned tasks to help another robot complete its unfinished tasks in simulation.
- Attempt to apply the algorithm to a real-world cleaning application using at least two robots.

2. METHODOLOGY

2.1. Overview

The strategy planning algorithm can be divided into three main blocks: Image Processing, Map Decomposition, and Task Allocation. Each block must be run sequentially in the order as mentioned as each subsequent block is reliant upon the output of the previous block. In this section, an overview of the algorithm will be presented while a detailed description of each block will follow in the succeeding sections. Assume that all functions mentioned were provided by MATLAB either through the base program or any additional toolboxes utilized unless otherwise stated.

2.1.1. Inputs

The inputs to the algorithm are:

1. The edge length of the longest side of the robot measured in grid coordinates.
2. Length of the longest diagonal from the center point to a corner of the robot measured in grid coordinates.
3. Target minimum triangle mesh edge length.
4. Desired starting location specified as an integer of either 1) the farthest corner from the shape centroid, 2) the closest corner to the shape centroid, or 3) the centroid of the shape.
5. The file path of the completed post-edited LiDAR scan of the work area in PGM file format.
6. The number of robots to be utilized.

2.1.2. Outputs

The strategy planning algorithm outputs have been tailored to reflect the inputs required by the subsequently used path planning algorithm. The first output is a probabilistic occupancy map of the work area to be completely covered where a value of 0 represents a cell to be cleaned denoted by the color white, a value of 1 represents an occupied cell denoted by the color black, and 0.5 represents an already cleaned cell when the path planning algorithm is performing complete coverage. The second output is a coordinate pair containing the starting location for the path planning algorithm to begin completely covering. The method by which these outputs are generated will be described in sections 2.4.3 and 2.4.4 respectively.

2.1.3. Software Utilized

In addition to the MATLAB base program, the addon toolbox software utilized by the strategy planning algorithm is as follows:

- Image Processing Toolbox
- Navigation Toolbox
- Optimization Toolbox
- Parallel Computing Toolbox
- Partial Differential Equations Toolbox
- ROS Toolbox
- Statistics and Machine Learning Toolbox

2.1.4. Required Functions

Listed below are the functions required in addition to the main *roomStrategy* script for the successful execution of the strategy planning algorithm. The function files are obtainable from the repository mentioned in [36].

- *allocateTasks*
- *directedTSP*
- *identifyBoundaries*
- *identifyBox2merge*
- *identifyTri2merge*
- *poly2occgrid*
- *polyConn*
- *polyNeighbor*
- *triDecomposition*

2.1.5. Assumptions

The strategy planning algorithm is based on the following assumptions:

1. All required software needed to run the algorithm has been preinstalled on the machine executing the algorithm.
2. The map of the completed LiDAR scan is in the PGM file format.
3. The work area within the completed LiDAR scan forms an enclosed polygon and is bordered by at least one layer of occupied cells.
4. The number of robots to be utilized does not exceed two.
5. The shape of the robot is a square.

2.2. Image Processing

The Image Processing block involves converting an input PGM file of the completed LiDAR scan of the work area to both an occupancy map and a polyshape object.

2.2.1. LiDAR Map Pre-Processing

Before the PGM file of the LiDAR scan can be input to the algorithm for conversion, there is some pre-processing that must be performed to the map to ensure correct boundary identification. To begin pre-processing, an image editing software such as GIMP is recommended which provides tools for easy image manipulation. In the image editing software, the raw LiDAR scan image can be cleaned up by removing any extraneous data readings and ensuring that the work area forms an enclosed polygon bordered by at least one layer of occupied cells. An example of a raw image compared to an edited image is shown in Figure 1 below.

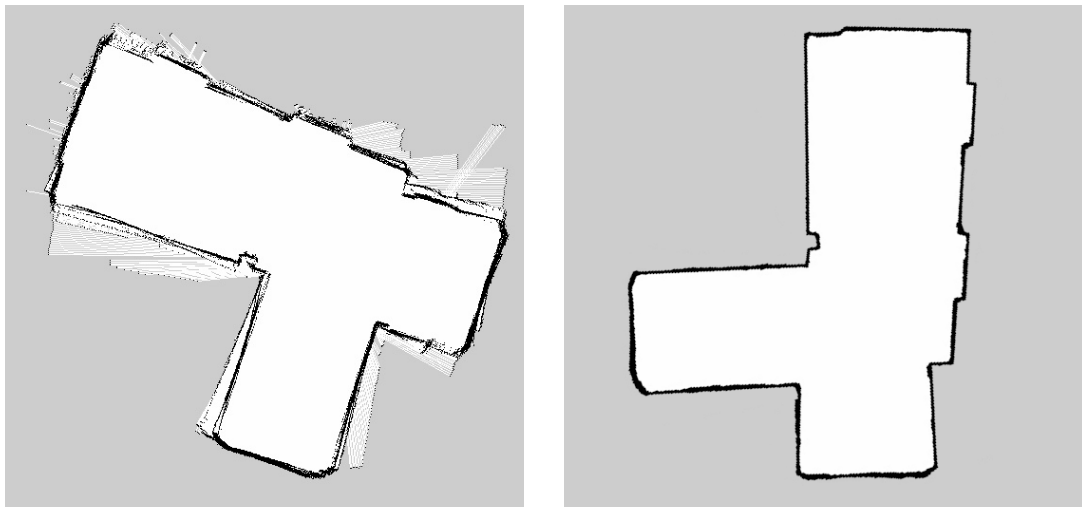


Figure 1. Raw LiDAR scan image compared to edited LiDAR scan image.

As can be seen, the raw image on the left features multiple locations where the LiDAR scan produced data points that lie outside the work area. This results in streaked areas, which are especially evident in the middle left and top right portions of the raw image. To remove these streaked areas, the default shade of light gray which occupies the region outside the work area is painted over the streaked areas thereby removing the extraneous readings. If these streaked areas are not removed during pre-processing, an incorrect representation of the work area may result following the processing of the image in the boundary identification step.

Another issue that may require closer visual inspection is if the work area does not form a closed polygon. In the above raw image, various locations about the exterior boundary of the work area have gaps in between occupied cells. These gaps must be filled in by editing in segments of occupied cells represented by the color black. The result should be similar to the edited image on the right in Figure 1 above where all the streaked areas have been removed and the exterior border is clearly defined without gaps. In performing these edits on the raw image, the work area should now be devoid of most measurement errors caused during mapping.

A final issue is the raw map orientation following the mapping process. This issue pertains more to how the algorithm perceives the work environment which in turn affects the resulting coverage efficiency. To lessen the impact of this potential deficiency, the raw map is visually rotated such that the majority of the exterior boundary borders are vertical. Following these edits the map is ready to be input to the algorithm. This is performed by utilizing the *imread* function where the function input is the file path of the edited image. The function output can then be assigned to a variable for access in the following steps.

2.2.2. Identifying Boundaries

The function utilized to identify the work area boundaries is that of *identifyBoundaries*, a required function that accepts the edited PGM image as input and returns the identified exterior and interior boundary coordinates. The Code Example 1 below depicts integral parts of the *identifyBoundaries* function.

```
function [exterior, interior] = identifyBoundaries(editedPGM)
binaryPGM = imbinarize(editedPGM);    % binarize the edited PGM image

[B,~,N,A] = bwboundaries(binaryPGM);
obs_idx = 1;    % variable to count the number of identified obstacles

%%% MATLAB bwboundaries example code %%%
% loop through object boundaries
for k = 1:N
    % boundary k is the parent of a hole if the k-th column
    % of the adjacency matrix A contains a non-zero element
    if (nnz(A(:,k)) > 0)
        exterior = B{k};
        % loop through the children of boundary k
        for l = find(A(:,k))'
            intBoundary = B{l};

            % additional code to save the points associated with each
            % identified interior obstacle
            tempVar = strcat('obs',num2str(obs_idx));   % update name of obstacle
            interior.(tempVar)= intBoundary;    % save current obstacle to struct variable
            obs_idx = obs_idx+1;    % update obstacle count
        end
    end
end
%%% End MATLAB example code %%%
```

```
% reduce the number of points describing the exterior boundary
exterior = reducepoly(exterior,.015);
end
```

Code Example 1. Boundary Identification

The *identifyBoundaries* function is primarily centered around *bwboundaries* which requires the input image to be in binary format. Given that the PGM file type stores image data in grayscale format, each pixel within the image can be represented by a value ranging from 0 to 255 where 0 represents black and 255 represents white. Hence the edited image must first be converted to binary format using the *imbinarize* function. The output of this function is a matrix containing the binarized values of each pixel within the input edited image. Given that an image is just a matrix of values containing the color values for each pixel within the image, the resulting matrix of binarized values can be passed directly into the *bwboundaries* function.

The *bwboundaries* function attempts to locate boundaries and any holes associated with that particular boundary. In the current use case of the function, the input image should ideally contain one boundary which is representative of the work area and any identified holes within this boundary are akin to static obstacles within the work area. By slightly modifying an example code by MATLAB showing how to use the *bwboundaries* function outputs to loop through each identified boundary and any identified holes associated with a particular boundary, any identified holes – more specifically the coordinates of the identified hole – are named and stored within the struct variable *interior* for output.

Similarly, the coordinates which make up the exterior boundary of the work area are also stored in the variable *exterior* for output. This array of coordinates however can be quite long containing hundreds of points identifying every occupied cell on the work area exterior boundary. This large quantity of points is not necessary as ideally only the coordinates which describe the corners of the work area are required to generate a polyshape object. To reduce the number of coordinates describing the work area, the function *reducepoly* can be utilized. Inputting the identified work area coordinates stored in the *exterior* variable and a tolerance value, in this case 0.015, the number of coordinates describing the work area can be drastically lessened from hundreds to just tens or less. Figure 2 below depicts a sample identification of the work area exterior boundary outlined in red, which was originally described by 1895 points now described only by 17 points, and any identified interior obstacles within the work area outlined in green.

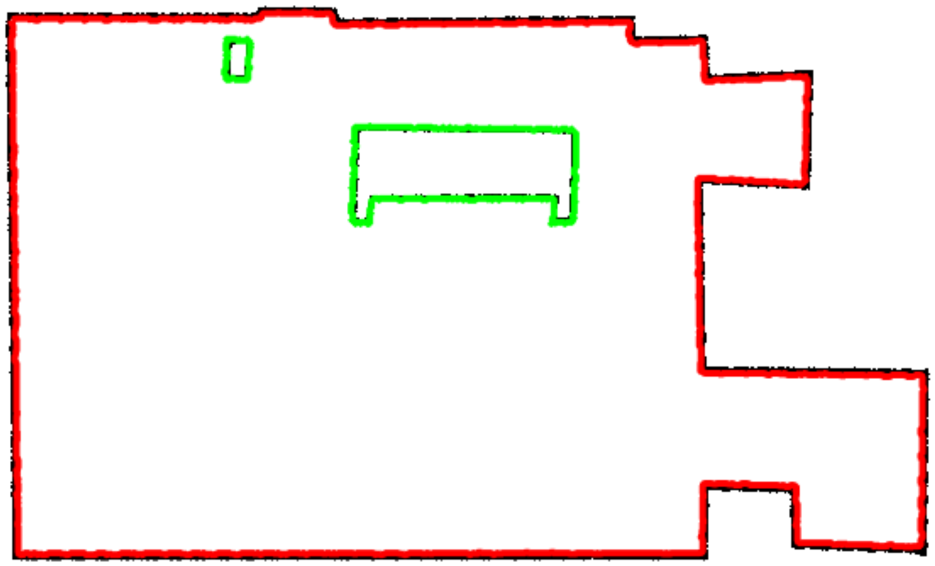


Figure 2. Boundary identification with two obstacles outlined in green.

Both *exterior* and *interior* variables are returned to the main workspace however in some instances if the work area does not contain any interior obstacles, *bwboundaries* can misidentify the actual work area as an interior obstacle and the image border as the exterior boundary resulting in an output similar to that of Figure 3 below.

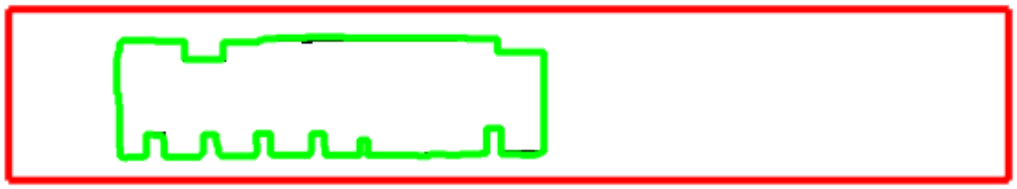


Figure 3. Boundary identification with no obstacles.

To prevent potential errors during interior obstacle removal as a result of the misidentification of the work area, a check is conducted on the returned outputs to determine if misidentification took place. As shown in Code Example 2 below, the check is performed by determining if the number of identified obstacles is equal to one and the number of identified exterior boundary coordinates is less than ten. These check values were determined based on the fact that the work area would be the lone identified hole within the image boundary and the number of coordinates describing the image boundary are typically equal to four following the use of *reducepoly*, although not in all cases, so a value of ten was chosen to capture the majority of misidentification incidents.

```
% determine if the exterior boundary has been misidentified as an interior obstacle,
% if so set the interior obstacle as the exterior boundary
obsNum = fieldnames(interiorIJ);    % determine number of identified interior obstacles
if (length(exteriorIJ) < 10) && (numel(obsNum) == 1)
    % reduce the number of points describing the exterior boundary
    exteriorIJ = reducepoly(interiorIJ.obs1,.01);
    removeObs = 0;  % do not remove interior obstacles
else
```

```
    removeObs = 1;  % do remove interior obstacles
end

% convert the exterior boundary points from IJ to XY coordinates
exteriorXY = grid2world(editedMap,exteriorIJ);
```

Code Example 2. Work Area Misidentification Check

If it is found that the work area was misidentified, the identified interior obstacle boundary is resaved as the exterior boundary and the remove obstacle flag is set to 0 to skip the obstacle removal process. However before the obstacle removal process can take place, it is important to note that the returned coordinates identified by *bwboundaries* are in grid coordinates of the form (i, j) where i represents the row and j the column. In order to convert these grid coordinates to world coordinates of the form (x, y) where x represents the distance along the horizontal axis and y the distance along the vertical axis, the *grid2world* function must be utilized. This function requires two inputs, an occupancy map of the work area and the grid coordinates to be converted. The latter of the two inputs is known however the former input will need to be generated.

To create an occupancy map of the work area, the input edited PGM image pixel values will need to be normalized to a value between 0 and 1. In terms of occupancy maps, each pixel is referred to as a cell where 0 denotes the likelihood of an unoccupied cell and 1 an occupied cell. Code Example 3 below outlines the occupancy map generation process.

```
% normalize the image to values between 0 and 1 then convert to occupancy
% values by subtracting from 1
editedMap_occ = 1 - double(editedPGM)/255;
% generate an occupancy map from the occupancy values
editedMap = occupancyMap(editedMap_occ,resolution);
```

Code Example 3. PGM to Occupancy Map

As previously mentioned, the edited image is simply a matrix of pixel values ranging from 0 to 255, and therefore each pixel value will need to be converted to a double precision data type using the *double* function and then divided by 255 to normalize the resulting values. These values are almost occupancy values however they are inverted due to the inverse meaning of 0 and 1 with respect to occupancy grids and the normalized grayscale values of the edited image. For occupancy maps, a value of 0 represents an unoccupied cell and is denoted by the color white while in grayscale a value of 0 represents the color black. To correct this inverse relationship, the normalized grayscale values of the edited image are then subtracted from 1 to convert them to probabilistic occupancy values. It is with this matrix of occupancy values that an occupancy map can be generated using the *occupancyMap* function. This function generates an occupancy map based on an input array of probabilistic occupancy values and an optional resolution value which determines the size of each cell within the occupancy map. By default each cell within the occupancy map is 2 cm by 2 cm, however due to the default resolution of the LiDAR map generating software of 1 cell per meter, a resolution value of 50 must be applied. Figure 4 below depicts a generated occupancy map of a work area.

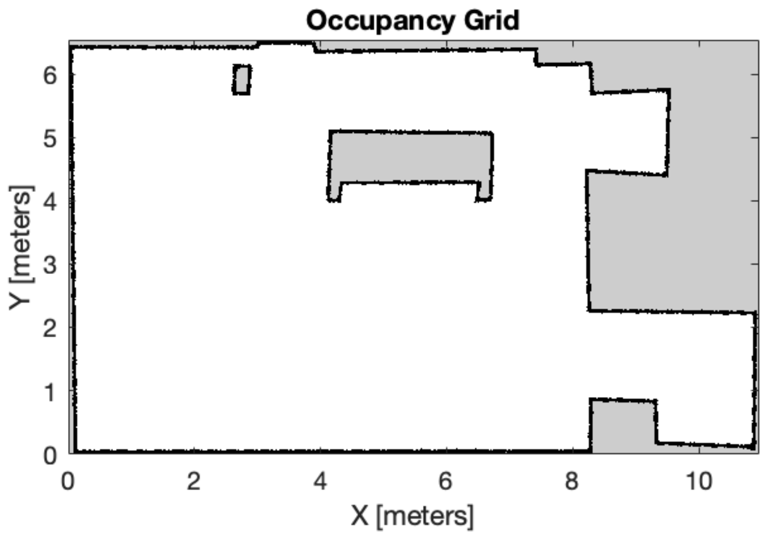


Figure 4. Work area occupancy map.

Now inputting the recently created occupancy map of the work area and the grid coordinates to be converted into the *grid2world* function, the grid coordinates can be successfully converted to world coordinates. It is at this point that a polyshape object of the work area can be generated.

2.2.3. Polyshape Representation

With the coordinates of the exterior boundary of the work area and any identified obstacles now known, the work area which was once represented as pixels within an image can now be represented as a polyshape object or polyshape for short. The purpose for creating a polyshape of a particular shape is that MATLAB can perform various geometric queries and calculations on the polyshape itself which will be a useful tool in future steps. For now, the coordinates of the exterior boundary of the work area will be utilized to generate a polyshape of the work area by calling the *polyshape* function and inputting the (x, y)

coordinate pairs saved in the *exterior* variable. An example resulting polyshape is shown in Figure 5 below.

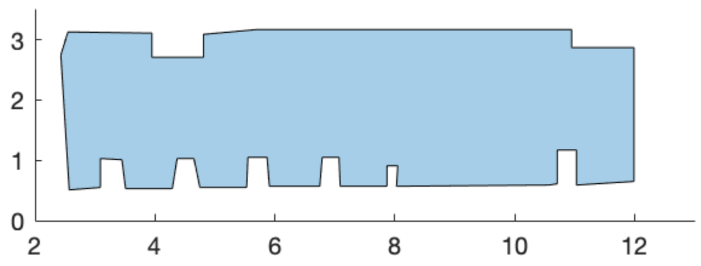


Figure 5. Polyshape representation of work area.

In the case of the work area represented in Figure 5 above, the work area did not have any identified interior obstacles and was therefore misidentified as an interior obstacle as shown in Figure 3. After being checked for misidentification the remove obstacle flag for this work area would be set to 0 to skip the obstacle removal process. However, if the conditions outlined in the check for misidentification are not met, the remove obstacle flag is set to 1. This signals to the algorithm that there are interior obstacles to be removed from the work area polyshape. Figure 6 below depicts a work area polyshape before obstacle removal.

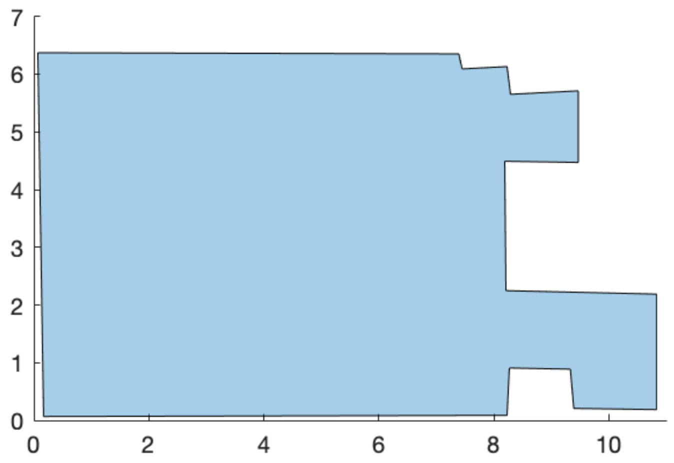


Figure 6. Polyshape representation of work area prior to obstacle removal.

The obstacle removal process iterates through the number of identified interior obstacles. For each obstacle, its points are converted from grid to world coordinates, the number of points describing the obstacle is reduced using *reducepoly*, and a polyshape of the obstacle is generated. The final step of the process involves subtracting the obstacle polyshape from the work area polyshape. Figure 7 below illustrates a work area polyshape following the obstacle removal process.

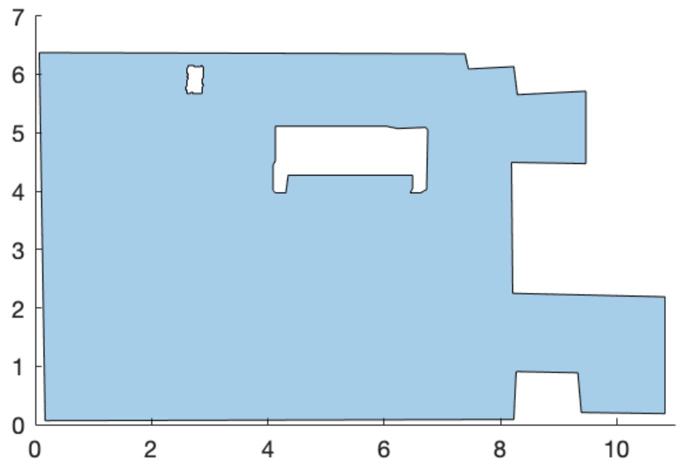


Figure 7. Polyshape representation of work area post obstacle removal.

With the work area now represented as a polyshape, the Image Processing block of the algorithm is concluded. This polyshape representation of the work area will then be passed to the next block for decomposition.

2.3. Map Decomposition

The Map Decomposition block involves both decomposing the polyshape object representing the work area into sections and then determining an optimal global path for navigating the decomposed work area. The outputs of this block are a list of tasks to be completed and the corresponding polyshape objects.

2.3.1. Triangulation Mesh

The method of decomposition utilized to decompose the work area polyshape is that of a triangulation mesh which is applied within the *triDecomposition* function, a required function that accepts the work area polyshape and the target minimum triangle mesh edge length as inputs. The function then outputs a struct variable containing the triangulation mesh connectivity list, triangle points, a list of polyshape objects, and triangle centroid coordinates. The triangulation mesh is formed by the function *generateMesh* which requires first creating a *model* variable containing the geometry that the triangular mesh will be generated from. Code Example 4 below depicts the triangular mesh generation steps.

```
% perform initial triangulation on room
TR = triangulation(room);

% generate a geometric model based on initial triangulation
model = createpde;
tnodes = TR.Points';
telements = TR.ConnectivityList';
geometryFromMesh(model,tnodes,telements);

% generate triangular mesh
room_mesh = generateMesh(model,'GeometricOrder','linear','Hmin',min_triEdge);
```

Code Example 4. Triangular Mesh Generation

The first step is to perform an initial triangulation on the polyshape object. Utilizing the *triangulation* function, which accepts polyshape objects as an input, a triangulation object containing the triangulation points and connectivity list is outputted. While the triangulation object produced would be a valid form of decomposition as shown in Figure 8 below, the reason it is not used is due to the inability to have control over the size of the triangles produced.

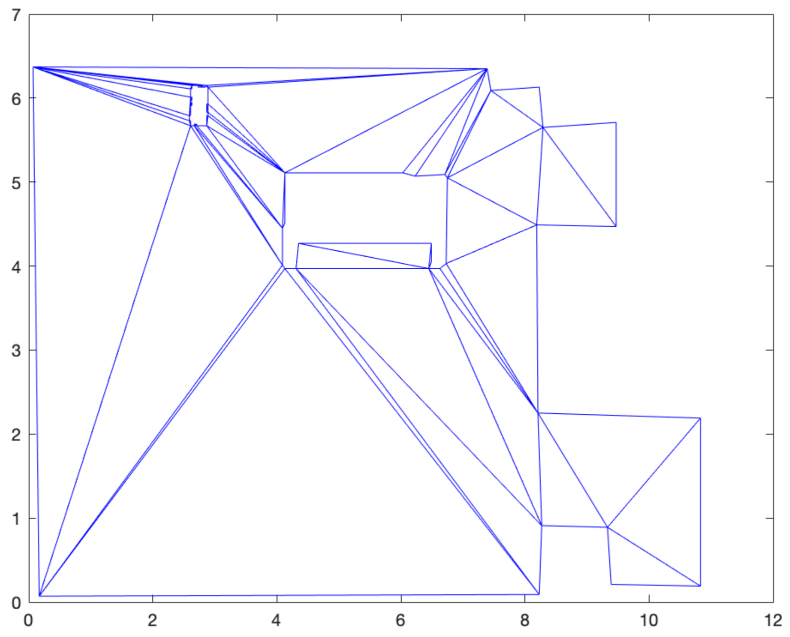


Figure 8. Sample triangulation decomposition output.

The triangulation object outputted by the *triangulation* function is a struct variable containing the triangulation points and connectivity list which can then be directly correlated to the *nodes* and *elements* variable inputs of the *geometryFromMesh* function respectively. With the inclusion of the desired *model* variable which was previously assigned with “createpde” to denote it as a geometric model as input, *geometryFromMesh* creates geometry within the *model* variable according to the geometry stored within the *nodes* and *elements* variables. It is at this point that the *model* variable required to generate the triangular mesh has been successfully created.

Now calling the *generateMesh* function with the *model* variable input and the "GeometricOrder" and "Hmin" options set to "linear" and the target minimum triangle mesh edge length respectively, a triangulation mesh of the work area polyshape object is generated and saved to a variable. Figure 9 below illustrates a resulting triangulation mesh.

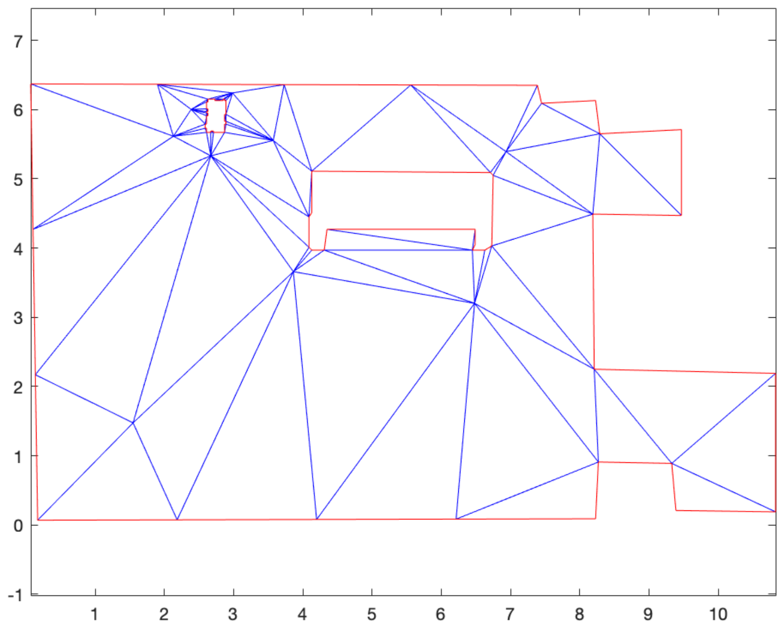


Figure 9. Sample triangulation mesh decomposition output.

To address the potential issue of generating a high number of triangles using this method, caused by a low input target minimum edge length or not adequately reducing the number of points describing each boundary which may lead to the generation of many triangles when performing the initial triangulation and subsequently the triangular mesh, a check for the number of generated triangles is implemented as shown in the Code Example 5 below.

```
% regulate the number of generated triangles in the mesh to 250 or less
if length(room_mesh.Elements') >= 250
    num_triangles = length(room_mesh.Elements');
    min_triEdge = min_triEdge+.1; % increment minimum edge length to create larger triangles

    loop_count = 1; % initialize loop counter
    while (num_triangles > 250) && (loop_count <= 500)
        % regenerate triangular mesh
        room_mesh = generateMesh(model,'GeometricOrder','linear','Hmin',min_triEdge);

        num_triangles = length(room_mesh.Elements'); % update number of triangles within mesh
        min_triEdge = min_triEdge+.1;   % increment target minimum edge length
        loop_count = loop_count+1;  % increment loop counter
    end
end
```

Code Example 5. Triangle Mesh Regulation

The threshold for this check is 250 triangles which was settled upon after reviewing the performance of the TSP algorithm, which will be described in a later section, whose problem size to determine an optimal path through N number of points scales by N^2 according to [37]. Hence increasing the amount of time required by the algorithm to determine an optimal path and the time required by the strategy planning algorithm overall. If it is found that the number of triangles exceeds 250, the process of generating the triangular mesh is repeated by incrementing the target triangle mesh minimum edge length by a value of 0.1 and rechecking the number of triangles produced. This continues until either the number of triangles falls below the 250 threshold or the number of iterations performed exceeds 500 to prevent the algorithm from becoming stuck in this process.

Following the generation of the triangulation mesh and checking the number of triangles generated, each triangle within the mesh is transformed into a polyshape object by iterating through the number of triangles generated. This value is obtained from the number of rows within the *elements* array of the triangulation mesh where each row describes a triangle and each column contains an index value corresponding to a row within the *nodes* array of the triangulation mesh denoting where each coordinate pair is stored. By assigning each coordinate to a variable A, B, and C, a polyshape object of the current triangle can be generated based on these points. Furthermore, once the triangle is represented as a polyshape object the triangle centroid can also be calculated using the *centroid* function. Saving both the *elements* array and *nodes* array to a struct variable for later reference, the polyshape object of each triangle and corresponding centroid coordinates are also saved to the same struct variable which is then returned to the main workspace alongside the variable storing the triangulation mesh.

2.3.2. Global Optimal Path Identification

To identify a global optimal path, a solver-based TSP algorithm is utilized. Contained within the required *directedTSP* function, this algorithm accepts coordinates of N points and attempts to determine an optimal path that passes through all points exactly once. In this use case of the algorithm, the input points are the centroids of the triangles, although in later uses of the TSP algorithm centroids of the current polygons. The overall flow of the algorithm as paraphrased from [37] is to initially determine all possible connections or "trips" between each of the input points, add constraints to these trips to ensure each point has only two associated trips, perform an initial optimization which often results in more than one cycle referred to as "subtours", and then continuing to optimize the path until only one subtour remains by iteratively adding additional constraints to prevent that particular subtour from occurring again. The output path of this algorithm is a circular graph where each node represents a polygon centroid, and each edge is the path direction from node to node as shown in Figure 10 below.

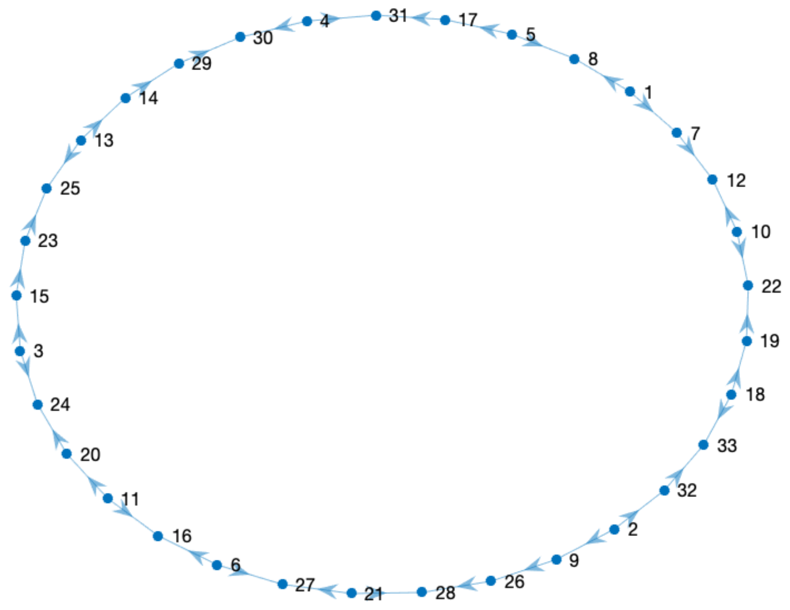


Figure 10. TSP algorithm output graph.

In examining Figure 10, it becomes evident there is no clear global path for navigating from node to node. To achieve a global path, all edge directions from node to node must point in the same direction. This is remedied by including additional code beneath the TSP algorithm within the *directedTSP* function to iterate through each node beginning at node 1, determine if the current node is the source or target of either edge, then flip one of the edges per one of two rules:

1) If the current node is a first edge target and is not yet a source node, flip the second edge such that the current node is now the second edge source.

2) If the current node is a first edge target and is a source node, flip the first edge such that the current node is now the first edge source.

The resulting graph in following the above rules is shown in Figure 11 below.

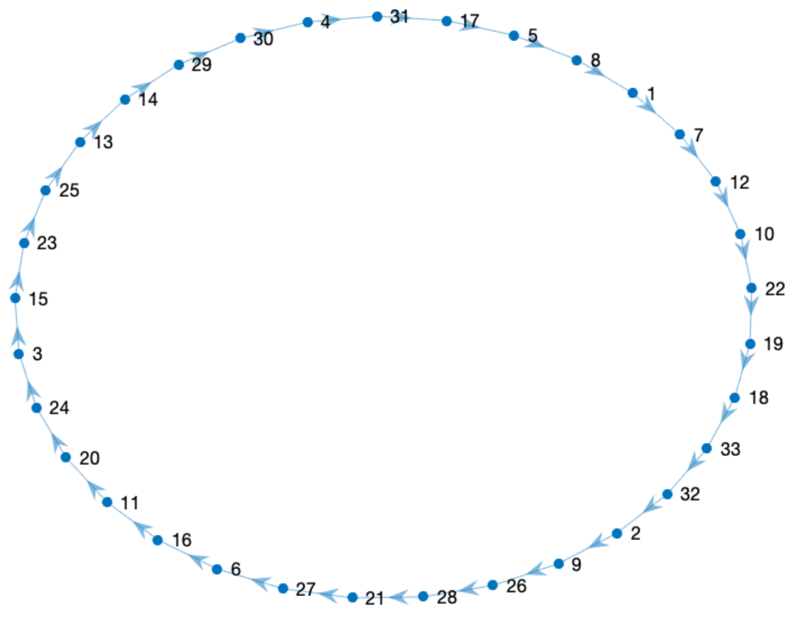


Figure 11. TSP algorithm directed output graph.

As can be observed, there is now a clear global path for navigating from node to node. Figure 12 below depicts the directed output graph overlayed on the global map highlighting the correspondence between the circle graph nodes and the centroids of the decomposed regions within the work environment.

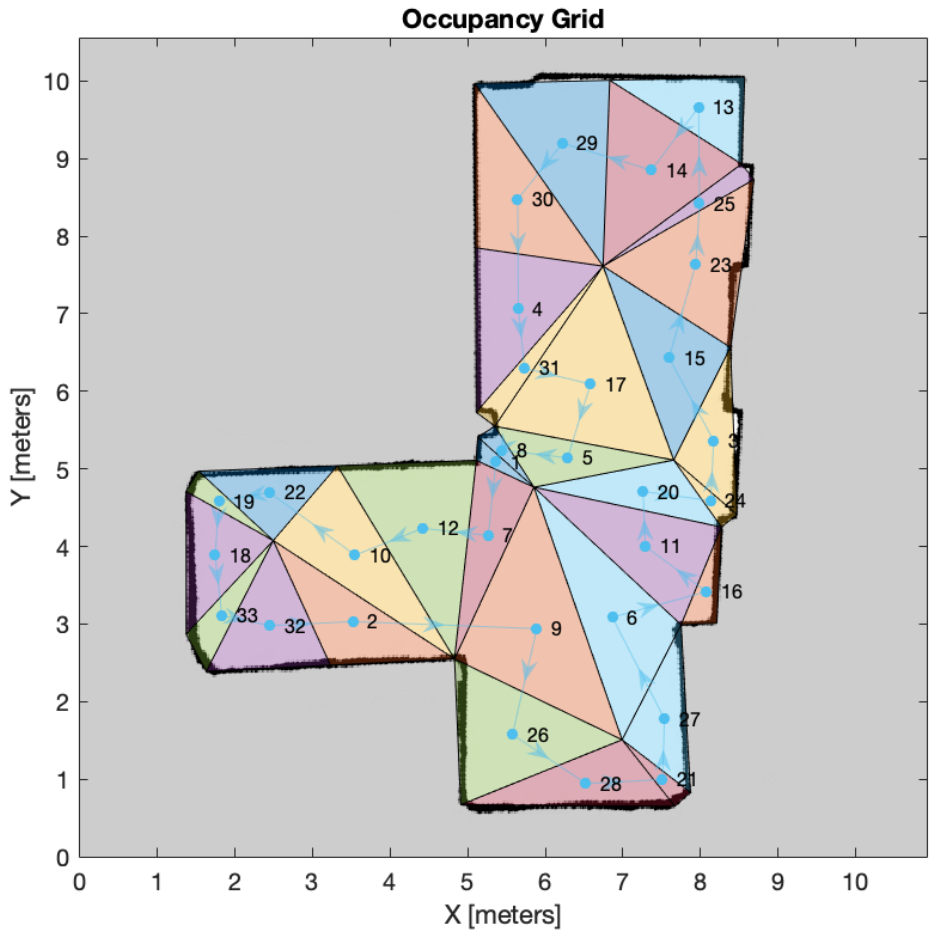


Figure 12. Directed output graph overlayed on global map.

A list of nodes can now be generated which is outputted back to the main workspace for later reference. An added benefit to utilizing the TSP algorithm is the automatic labeling of each node to a numeric value that directly corresponds to an index of a polygon within the polyshape list. This benefit helps in identifying which shapes to merge, are neighbors, or the specific tasks to be completed as the *directedTSP* function is called after any change in the number of polyshape objects within the work area for a total of four times throughout the course of the strategy planning algorithm.

2.3.3. Triangle Merging

As seen in Figure 9, some resulting triangulation mesh triangles are too small for a robot to fit into. To remedy this issue, triangles can be merged with their neighbors to form polygons of larger areas that a robot could then fit into. Performed within the required function *identifyTri2merge*, identifying which triangles to be merged simply involves iterating through each triangle, calculating its base and height, then determining if either of these values falls below a specific threshold; in this case, the threshold value is 0.5 m for both base and height. If a triangle is identified to require merging, its node label is saved to a variable. After this process has been completed, the identified node labels are then placed in ascending order which will then be referenced in the merging process depicted in the Code Example 6 below.

```
offset = 0; % counter to track the number of nodes removed from the original node list
for i = 1:length(tri2merge)
    % identify node to be merged by subtracting the offset from the current node value
    node2merge = tri2merge(i,1)-offset;

    Gtri = polyConn(tri.shape); % get current polygon connectivity

    % determine which neighboring node to merge with
    neighbor2merge = polyNeighbor(Gtri, node2merge, tri.shape);

    % merge neighbor node with current triangle node
    % save resulting merged polygon to neighbor node index in shape list
    neighborTri = tri.shape(neighbor2merge);
    currentTri = tri.shape(node2merge);
    tri.shape(neighbor2merge) = union(neighborTri,currentTri);

    offset = offset+1;  % increment offset value
    tri.shape(node2merge) = [];   % remove merged triangle node from shape list
end
```

Code Example 6. Triangle Merging Process

The process of merging the identified triangles involves first determining the neighbors of the triangle-to-be-merged and then implementing a criterion for which of the neighbors the identified triangle should be merged with. To identify the neighbors of the triangle-to-be-merged, the connectivity of the triangles must first be found which is done by the required function *polyConn* shown in the Code Example 7 below.

```matlab
function Gpoly = polyConn(pshape)
pbuff = polybuffer(pshape,.0001);   % add a buffer to each polyshape

% allocate an adjacency matrix of zeros of size NxN where N is equal to the
% number of polyshapes
pborder = zeros(length(pshape));

% loop through each polyshape object and determine which other polyshapes
% border it
for i = 1:length(pshape)
    for j = (i+1):length(pshape)
        % if the area of intersection between the current polyshape and
        % another exceeds the set threshold, set the corresponding
        % adjacency matrix location to true
        pborder(j,i) = area(intersect(pbuff(j),pbuff(i))) > 3e-6;
    end
end

% generate a connectivity graph of the filled in adjacency matrix
Gpoly = graph(pborder,'lower');
end
```

Code Example 7. Polygon Connectivity

Modified from a function by Loren Shure to identify neighboring states [38], *polyConn* identifies neighboring triangles from a list of polyshape objects by adding a small buffer to the borders of all triangles using the *polybuffer* function and then examining the area of intersection between them. The result is a graph where each neighboring triangle node is connected by an edge as depicted in the Figure 13 below and is overlayed on top of the work area for better visual reference in the following Figure 14.

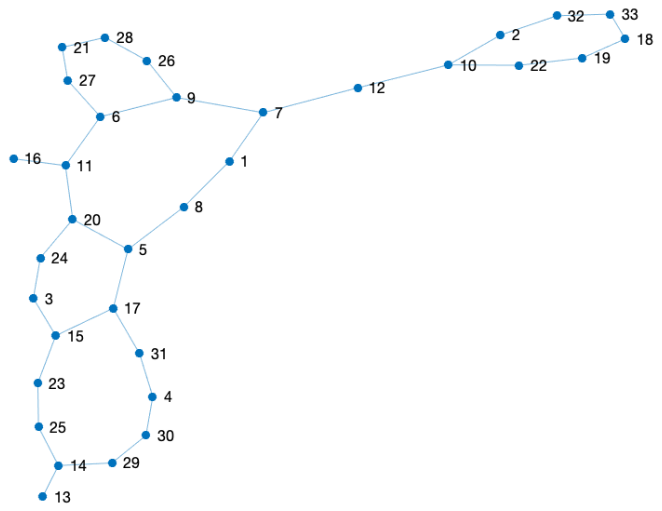


Figure 13. Triangle connectivity graph.

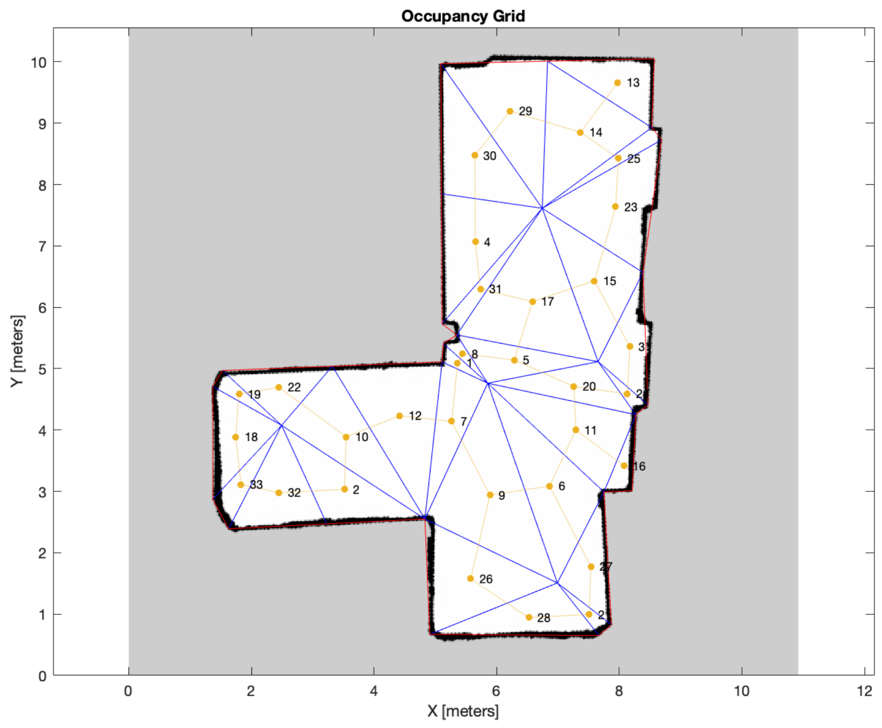


Figure 14. Triangle connectivity graph overlayed on work area.

With the connectivity of the triangles now known, the neighbors of the identified triangle-to-be-merged can then be determined by examining the end nodes of the connectivity graph. By calling the required function *polyNeighbor*, which takes the connectivity graph, the triangle-to-be-merged node label, and the list of polyshape objects as input, neighboring triangles are identified by locating the indices of the triangle-to-be-merged node label within the end node array of the connectivity graph. In finding these indices, the neighboring triangle is the other end node value within that row index and are saved to a variable. Identifying which of these neighbors to merge with is similar to *polyConn* by inflating each of the identified neighbors' borders and examining the area of intersection between them and the triangle-to-be-merged. The neighbor to merge with is then selected by choosing the neighboring triangle with the greatest amount of overlap with the triangle-to-be-merged. The node value of this neighbor is then returned to the main workspace.

The identified neighbor and triangle-to-be-merged are then finally merged into a new single polyshape object by using the *union* function. This new polyshape object is then stored within the identified neighbor's node index of the polyshape object list while the polyshape stored within the now merged triangle-to-be-merged node index is removed from the polyshape object list. This merging process continues iteratively until all triangles to be merged within the list have been merged. The result of the triangle merging process in comparison to before is shown below in Figure 15.

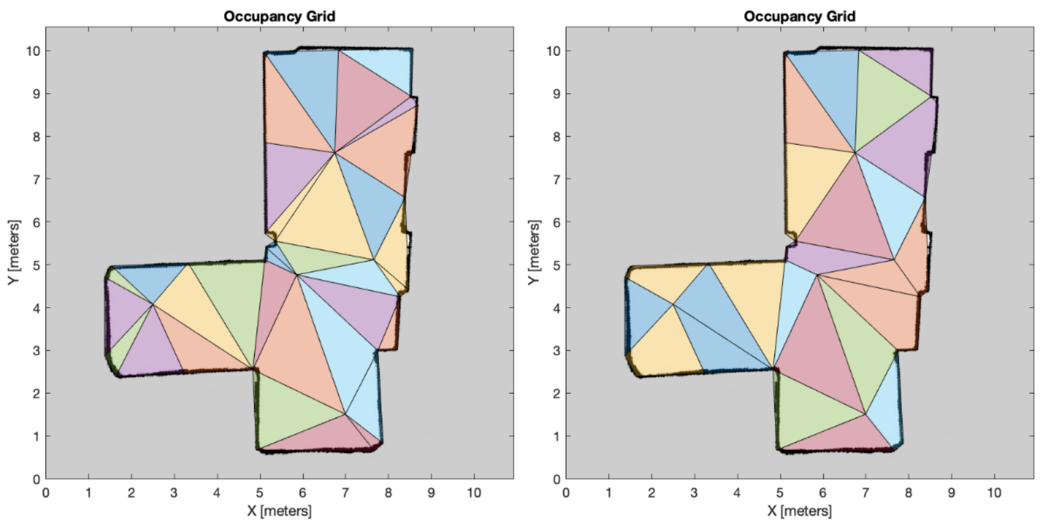


Figure 15. Triangle merging process before and after comparison.

Do note that since array elements are deleted throughout the process, the order in which the merging process is conducted is critical. Every time an element is deleted from the polyshape object list, an offset variable needs to be incremented. This offset value is then subtracted from the node value of the next identified triangle to be merged such that the correct node index values are utilized.

2.3.4. Bounding Boxes

Despite ridding of many smaller triangles, the resulting polygons still exhibit many narrow corners and awkward edges that would inhibit the ability of a robot to sufficiently cover. The solution to this issue is to generate bounding boxes for each of the resulting polygon shapes. In doing so, many of the narrow corners and awkward edges are removed with sharp 90-degree corners and straight edges.

To generate bounding boxes for each polygon, the *boundingbox* function is used which outputs the x and y limits for each input polygon polyshape object. Utilizing these limits, the minimum and maximum coordinate pairs are determined and are utilized to create a new rectangular polyshape object for each polygon as shown in Figure 16 below.

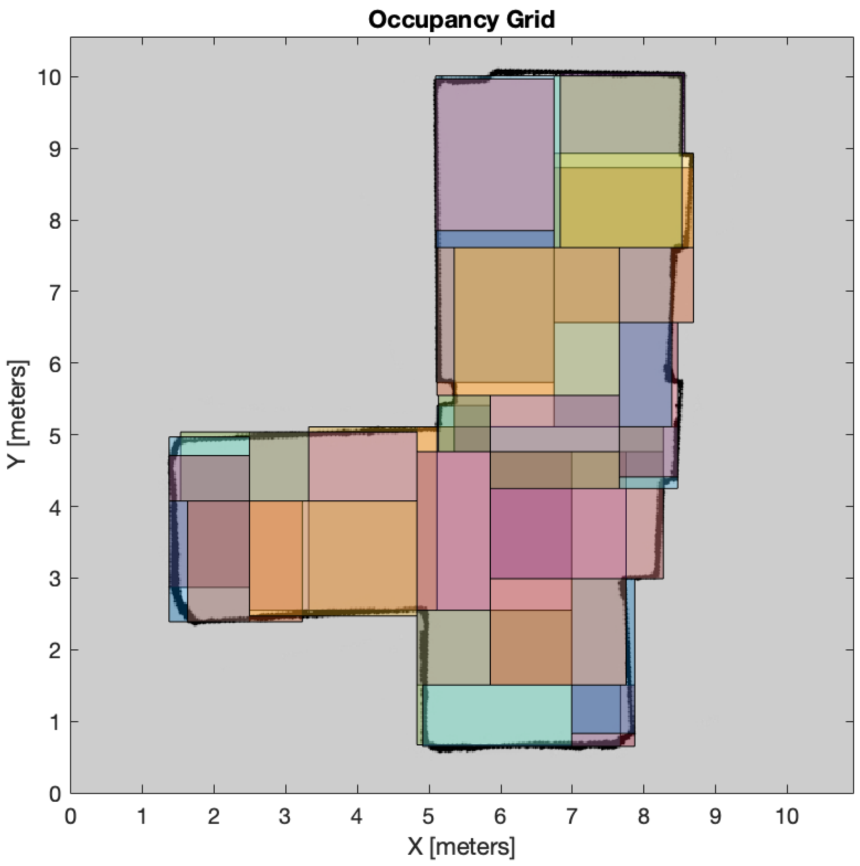


Figure 16. Merged triangle polygons converted into bounding boxes.

In performing the conversion to bounding boxes, many areas of overlap between other bounding boxes occur as well as some boxes extending outside the work area and hence will need to be cleaned up. The cleanup process begins by creating a polyshape object that is the negative of the work area then iterating through each bounding box to remove any overlaps between it and neighboring bounding boxes or the negative of the work area. Overlap between the current bounding box and the negative of the work area is removed by simply subtracting the latter from the former. Overlaps between bounding boxes require first utilizing *polyConn* to get the current bounding box connectivity, implementing a similar process to *polyNeighbor* where the identified neighbors are narrowed down to one except in this case all identified neighbors are needed, then removing any overlapping regions between the current bounding box and identified neighbors from the bounding box with the higher area value. In some instances, this process will result in a current bounding box that is completely removed. If such a situation occurs, the node value corresponding to the bounding box is removed from the bounding box polyshape object list and an offset value is incremented similar to the triangle merging process. Figure 17 depicts the resulting bounding boxes following the cleanup process.

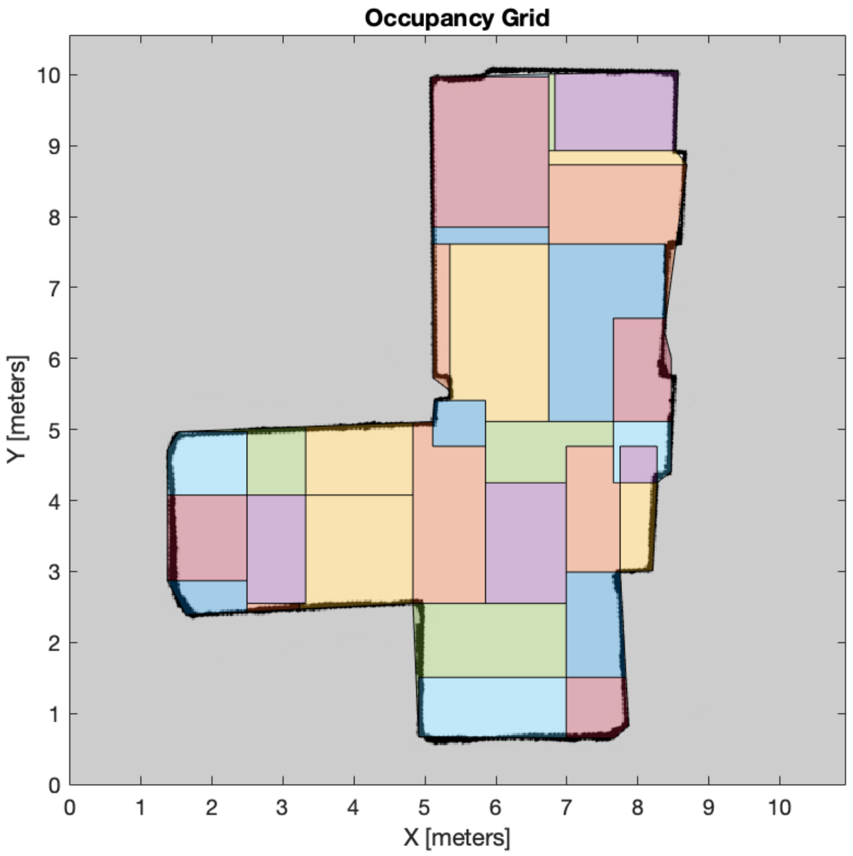


Figure 17. Cleaned up bounding boxes.

Each bounding box is then examined for slivers which are small areas that jut out from the main polyshape object and checked such that each polyshape object consists of only one region. If any slivers are identified that area is turned into its own polyshape object, subtracted from the parent polyshape object, and appended to the list of polyshape objects along with any supplementary regions found. It is at this point the merging process can take place.

Similar to the triangle merging process, the bounding box merging process begins with identifying which bounding boxes to merge by calling the required *identifyBox2merge* function. This is done by iterating through each polyshape object within the list and determining if the robot, whose size was an input to the strategy planning algorithm, can fit within that polyshape object. If the robot cannot fit, the node value index for that particular polyshape object is added to a merging list. This list is then put into ascending order before the merging process which features the same steps utilized in the triangle merging process. The final step is to then ensure that each resulting polyshape object contains only one region and has no unnecessary holes. If more than one region or a hole is identified, a bounding box covering the entire polyshape object is created and replaces the original polyshape object. This can result in some overlap with neighboring bounding boxes or even regions outside the work area. Hence, these new polyshape objects are recleaned before being saved to the final bounding box polyshape object list. Figure 18 below depicts the results of the bounding box merging process.

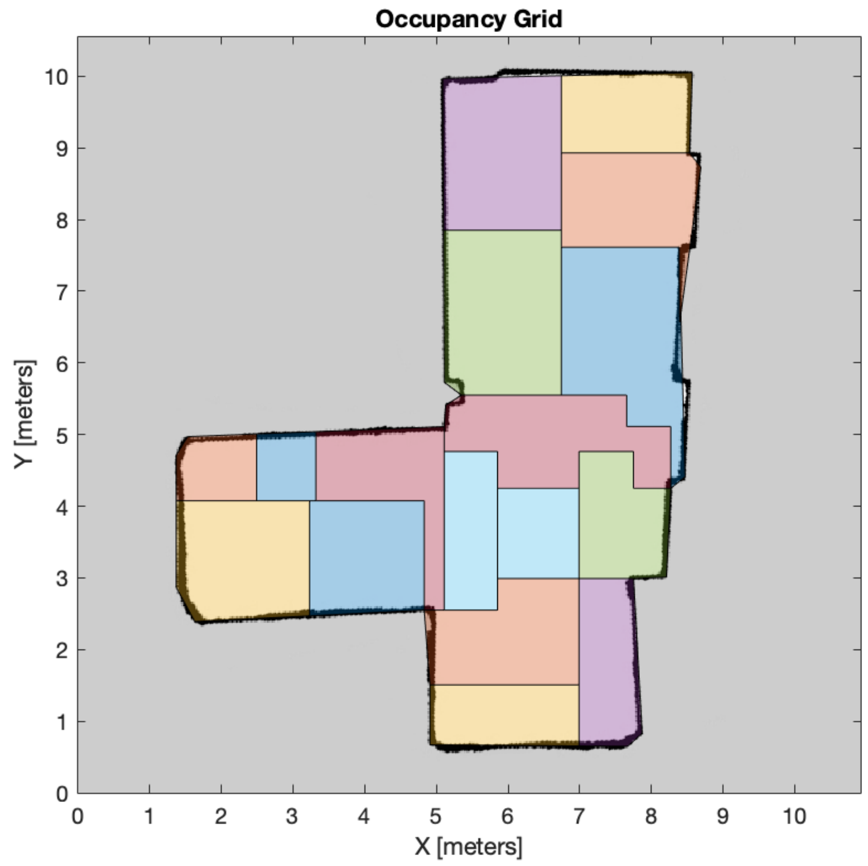


Figure 18. Bounding boxes post-merging process.

Following the bounding box merging process, the centroid coordinates of each remaining polygon are calculated. These coordinates are then passed into the *directedTSP* function one final time. It is this generated global path, output in the form of a node list, which will dictate the order by which each of the resulting bounding boxes are tasked to the robots for complete coverage. Figure 19 below illustrates the outcome of the map decomposition block of the strategy planning algorithm by overlaying the identified global path by the TSP algorithm over the resulting bounding boxes shown in the previous Figure 18.

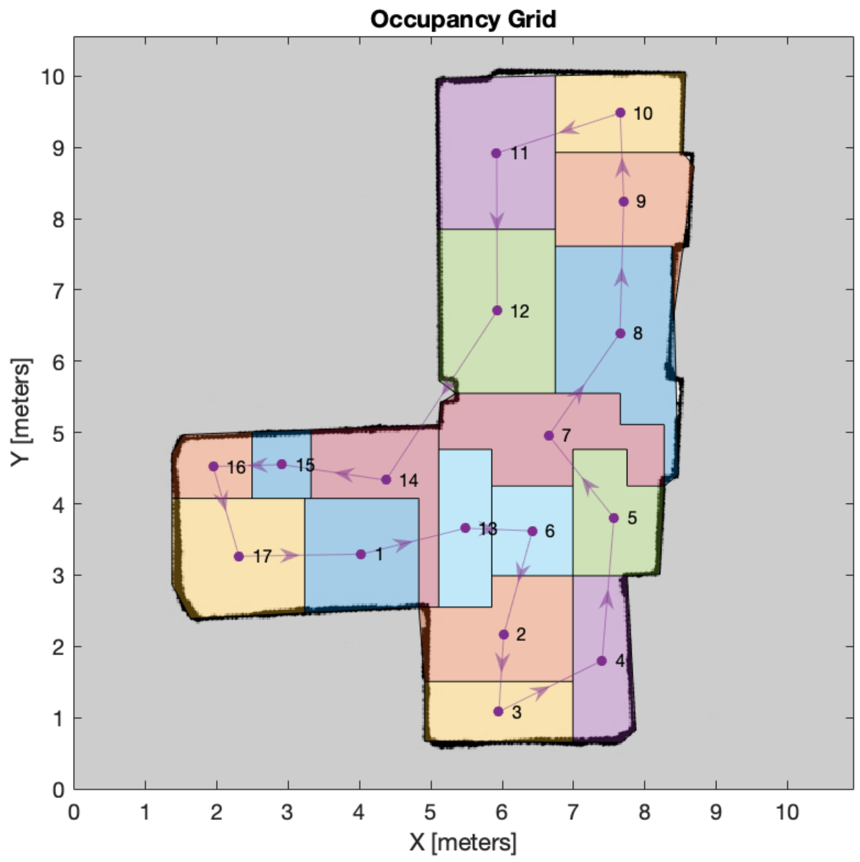


Figure 19. Resulting bounding boxes with global path overlayed.

This concludes the Map Decomposition block of the algorithm. The final bounding box list of polyshape objects and node list will then be passed to the next block for task allocation.

2.4. Task Allocation

The Task Allocation block involves allocating the tasks as mentioned within the node list, then generating an occupancy map and starting location for each task as required by the path planning algorithm.

2.4.1. Allocation of Tasks

The process of allocating tasks involves first dividing the node list into adjacent segments based on the number of available robots. A sample node list is depicted in Figure 20.

```
    1.0000    4.0232    3.2928
   13.0000    5.4817    3.6566
    6.0000    6.4218    3.6200
    2.0000    6.0208    2.1683
    3.0000    5.9532    1.0850
    4.0000    7.3996    1.8009
    5.0000    7.5652    3.8020
    7.0000    6.6526    4.9576
    8.0000    7.6499    6.3962
    9.0000    7.7028    8.2464
   10.0000    7.6528    9.4816
   11.0000    5.9220    8.9206
   12.0000    5.9290    6.7075
   14.0000    4.3690    4.3322
   15.0000    2.9090    4.5499
   16.0000    1.9465    4.5238
   17.0000    2.3091    3.2645
    1.0000    4.0232    3.2928
```

Figure 20. Sample node list.

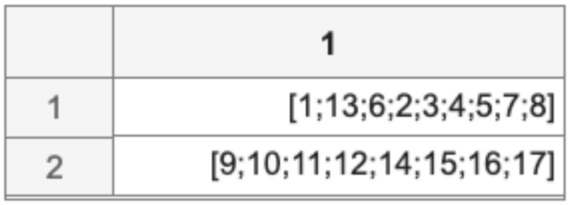
The node list is divided into three columns. The first column is the node index value while the second and third columns contain the x and y coordinates for each polygon centroid. For the purposes of task allocation, only the first column will need to be examined. To begin the allocation process, the last row of the node list is removed due to the circular nature of the graph produced by the TSP algorithm where node 1 appears twice within the list. With the last row now removed, the required function *allocateTasks* shown in Code Example 8 below takes the task list and the number of robots as inputs and outputs a cell array with the allocated tasks in segments to be assigned.

```
function task_segments = allocateTasks(task_list, numRobots)
% due to limitations of the below process with 3 tasks divided between 2
% robots, check if the number of tasks is equal to 3
if height(task_list) == 3
    task_segments = {task_list(1:2,:)}; % assign tasks 1 and 2 to segment 1
    task_segments(2,:) = {task_list(3,:)};  % assign task 3 to segment 2
else
    % calculate the number of tasks per segment and round down to the nearest integer
    num_task_per_segment = fix(height(task_list)/numRobots);

    % calculate the number of segments with an equal number of tasks and
    % round down to the nearest integer
    num_equal_segments = fix(height(task_list)/num_task_per_segment);

    % generate an array of ones with length equivalent to the number of equal
    % segments and multiply each array element by the number of task per segment
    segment_array = num_task_per_segment*ones(1,num_equal_segments);

    % determine the number of remaining tasks
    rem_tasks = rem(height(task_list),num_task_per_segment);

    % for each remaining task, increase the number of tasks within the
    % current segment index by 1
    for i = 1:rem_tasks
        segment_array(i) = segment_array(i)+1;
    end

    % convert the task list matrix to a cell array
    task_segments = mat2cell(task_list,segment_array,1);
end

% for every even task segment, flip the task order
for i = 1:length(task_segments)
    if mod(i,2) == 1; continue;
    else task_segments{i} = flip(task_segments{i});
    end
end
end
```

Code Example 8. Task Allocation

This is achieved by first determining the number of tasks to be assigned per segment by dividing the number of tasks by the number of robots and rounding down the result to the nearest whole number. Referencing the node list in the Figure 20 above which contains 17 tasks to be allocated between two robots, the number of tasks per segment would be equal to 8. Following this, the number of segments with an equal number of tasks is calculated and the result is rounded down to the nearest integer value which in this example would be two. These generated values are then used to form a segment array with a length equivalent to the number of segments with an equal number of tasks. Each element value within the newly made segment array is representative of the number of tasks to be assigned to the corresponding segment. In this instance, the segment array would be of length two with values $[8,8]$.

Following this, the number of remaining tasks is determined. As the number of remaining tasks will always be less than the number of elements within the segment array, this value can be utilized as an index to iterate through the segment array values in an ascending fashion and increase each encountered array index value by one. In the case of the ongoing example, the number of remaining tasks would be one for a final segment array result of $[9,8]$ meaning that the first segment will consist of nine tasks and segment two will consist of eight tasks. Utilizing the *mat2cell* function, the task list array is then converted into an $N \times 1$ cell array where N is equal to the number of elements within the segment array generated previously. Each cell is then assigned a length according to the value of the corresponding segment array element. The resulting example allocated task output is shown in Figure 21 below where task segment one contains nine tasks while task segment two contains eight tasks.



	1
1	[1;13;6;2;3;4;5;7;8]
2	[9;10;11;12;14;15;16;17]

Figure 21. Task allocation output.

In performing this task allocation step, the global path is sliced into adjacent segments with global directions as shown in Figure 22 and overlayed on the global map in Figure 23. It is worth noting that the described process does not work for task lists containing one or three tasks. The reasoning for one task is trivial in that a single task cannot be divided up in the above manner however the reason behind three tasks is not so trivial. It was found that the above process does not work properly with three tasks so a check was implemented before the main task allocation process such that if there are three tasks to be allocated, the first segment will contain two tasks and the second segment one task.

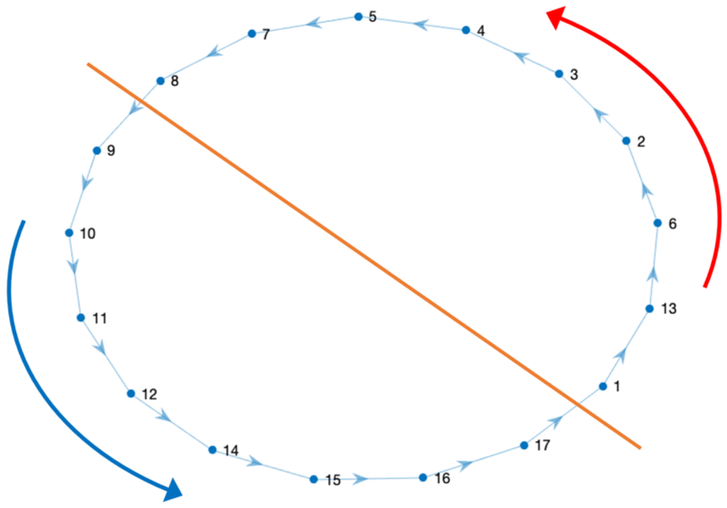


Figure 22. Segmented global path.

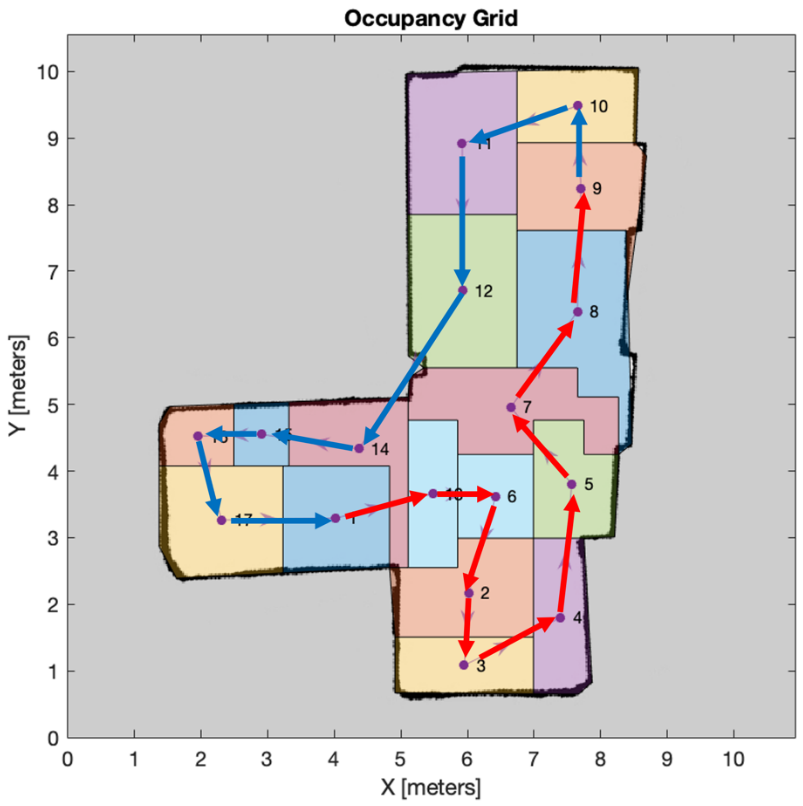
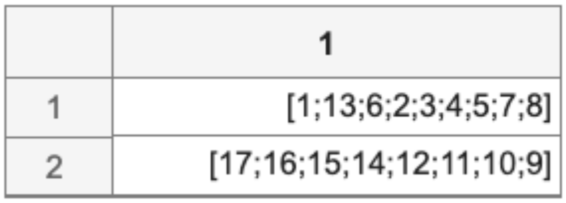


Figure 23. Segmented global path overlayed on global map.

If these were the global directions utilized, the robots would work moving away from each other meaning that if task reallocation were to be performed then a robot may be required to travel a longer distance to reach the new reallocated tasks. To remedy this issue, the global direction of each even task segment is flipped as shown in the task allocation output of Figure 24.



	1
1	[1;13;6;2;3;4;5;7;8]
2	[17;16;15;14;12;11;10;9]

Figure 24. Flipped task allocation output.

In performing the flipping of each even task segment, the robots will now work toward each other as shown by the global direction depicted in Figure 25 and the overlayed path in Figure 26.

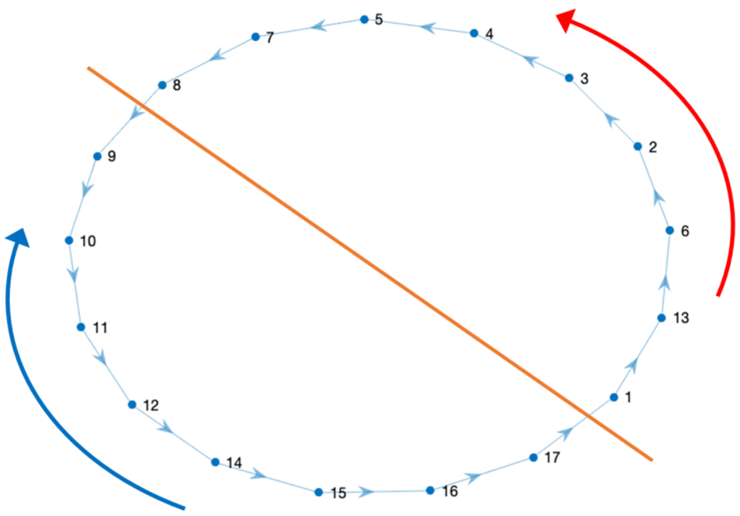


Figure 25. Flipped segmented global path.

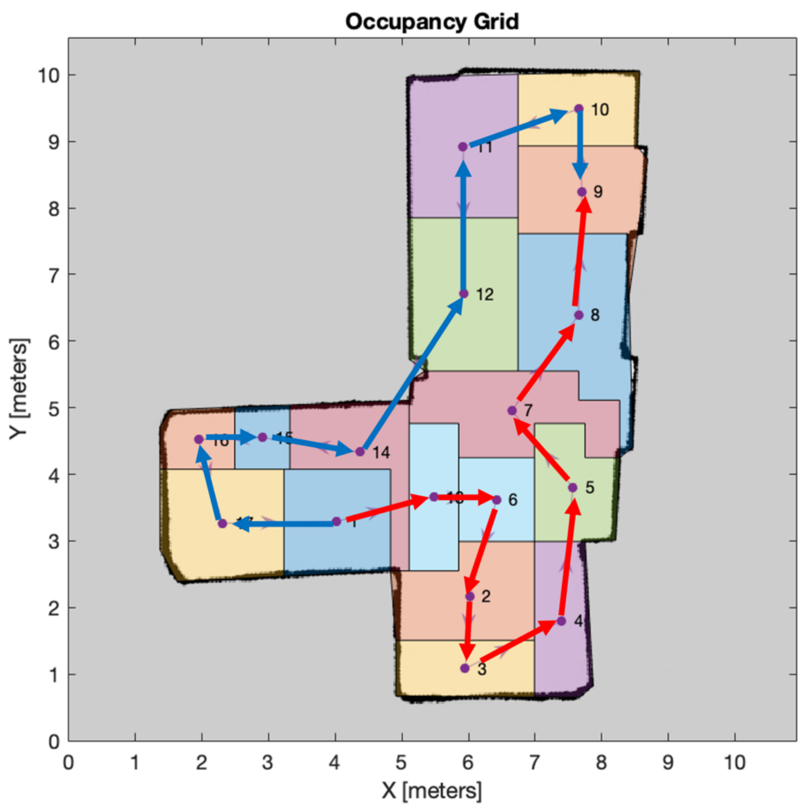


Figure 26. Flipped segmented global path overlayed on global map.

In working towards one another, the time required for navigating to any reallocated task should theoretically be reduced for a physical robot system and thereby decrease the overall cleaning time. With the tasks now allocated and the global direction for each task segment known, the process of performing complete coverage on each task can begin.

2.4.2. Dynamic Task Allocation

Performing complete coverage with multiple robots and the incorporating of task reallocation requires the use of parallel computing. MATLAB offers three main solutions for executing programs in parallel which are parfor, parfeval, and SPMD described in [39]. Both parfor and parfeval do not permit the exchanging of data between parallel processes which leaves SPMD as the remaining solution. An acronym for Single Program Multiple Data, SPMD does allow for the sharing of data between parallel processes which are referred to as “workers”. This is useful in the case of task reallocation as the robots are able to signal to each other when one robot has completed its assigned tasks so that the idle robot can then be sent reallocated tasks. A program overview of each SPMD worker is shown in Figure 27 below.

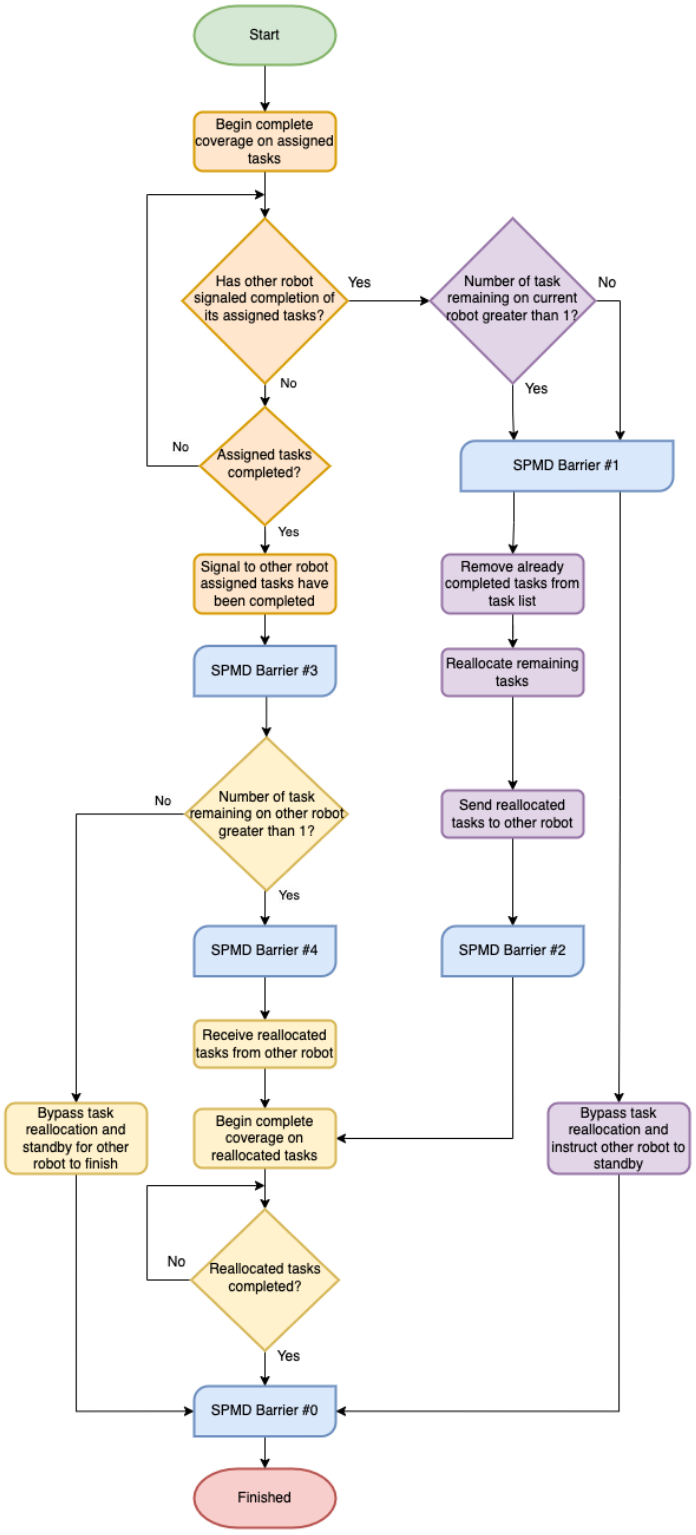

Start
Begin complete coverage on assigned tasks
Has other robot signaled completion of its assigned tasks?
Yes
No
Number of task remaining on current robot greater than 1?
No
Yes
Assigned tasks completed?
No
Yes
SPMD Barrier #1
Signal to other robot assigned tasks have been completed
Remove already completed tasks from task list
SPMD Barrier #3
Reallocate remaining tasks
Number of task remaining on other robot greater than 1?
No
Yes
Send reallocated tasks to other robot
SPMD Barrier #4
SPMD Barrier #2
Receive reallocated tasks from other robot
Bypass task reallocation and standby for other robot to finish
Begin complete coverage on reallocated tasks
Bypass task reallocation and instruct other robot to standby
Reallocated tasks completed?
No
Yes
SPMD Barrier #0
Finished

Figure 27. SPMD worker flowchart.

The SPMD program can be simplified into three main parts which have been highlighted for visual reference. Highlighted in orange, this portion of the program involves completing the originally assigned tasks while also checking after completing each task if the other robot has signaled it has finished its originally assigned tasks. The portion highlighted in purple accounts for the situation in which the other robot has signaled it has completed its originally assigned tasks. If the other robot does communicate it has finished, the loop outlining the orange portion is broken and a decision based on the number of remaining tasks on the current robot is encountered. If the number of remaining tasks is greater than one, the task reallocation part of the program is executed and the reallocated tasks are sent to the other robot. If there is only one task left, the task reallocation portion is bypassed and the other robot is instructed to stand by until the current robot has completed the remaining task. However if the current robot is able to complete all of its originally assigned tasks, the program segment highlighted in yellow is executed. This program portion handles the situation in which the current robot signals to the other robot it has completed its originally assigned tasks. Based on the information received from the other robot, the current robot will either bypass task reallocation and stand by for the other robot to finish its remaining task or receive the reallocated tasks from the other robot and then proceed to complete these tasks before finishing the program.

In order to keep both workers in sync based on the current situation and to prevent one worker from finishing before the other which results in an error due to the synchronization required by SPMD in that all workers must exit the SPMD block at the same time, SPMD barriers must be utilized. When a barrier is encountered by a worker, program execution stops and waits until the other worker has also encountered a barrier before proceeding. This pause in execution is useful when exchanging information between workers to ensure the receiving worker is always waiting on the sender to send the information and not in the reverse. In total five barriers are utilized for each worker and while not implemented within the program with an identifier, each SPMD barrier has been assigned an identifier to differentiate it from the others. In some cases when both robots finish their assigned tasks simultaneously, the program outlined in the flowchart above results in an error. To catch this potential error, a try-catch block is implemented around the entirety of the SPMD code block such that if the error occurs when trying to execute the program, the catch block will catch the error.

2.4.3. Occupancy Map Generation and Starting Location Identification of Task

When a task is identified by a worker to be completed, the work area represented by the task must be converted from a polyshape object to an occupancy map that can then be passed to the path planning algorithm for complete coverage as well as a starting location for where to begin coverage. This is done by calling the required *poly2occgrid* function which is split into two main parts. The first part of this function involves generating an occupancy map of the work area shown in the Code Example 8 below.

```
% obtain the current work area polyshape vertices
localPoints = polybox.shape(polybox.nodeList(idx,1)).Vertices;
localx = localPoints(:,1);
localy = localPoints(:,2);
```

```
% identify the minimum and maximum XY coordinates of the work area polyshape
xmin = min(localx); xmax = max(localx);
ymin = min(localy); ymax = max(localy);

% generate an occupancy map of the work area polyshape with a padding of .5
localMap = occupancyMap(xmax-xmin+.5,ymax-ymin+.5,resolution);
localMap.GridLocationInWorld = [xmin,ymin]; % update origin of occupancy map
updateOccupancy(localMap,1);    % set all cells within occupancy map to 1 (occupied)

% create check points to compare against the current work area polyshape
step = 1/(2*resolution);    % calculate the step value between each check point
localx_check = xmin:step:xmax;
localy_check = ymin:step:ymax;

% generate grid of check points
[localx_grid,localy_grid] = meshgrid(localx_check,localy_check);

% determine which grid points are in or outside the current work area
% polyshape
[in,on] = inpolygon(localx_grid,localy_grid,localx,localy);

% set the cell value to 0 if inside the work area polyshape
setOccupancy(localMap,[localx_grid(in & ~on),localy_grid(in & ~on)],0);
```

Code Example 9. Polyshape to Occupancy Map

To begin, the coordinate pairs describing the work area polyshape are obtained and then split into two arrays for x and y coordinates. Using the split arrays, the minimum and maximum values of both x and y coordinate values are determined. An occupancy map of the work area section, referred to as a local map, is generated using these values to create a bounding box by subtracting the maximum from the minimum x and y values with an 0.5 padding value added on. Since generated occupancy map origins always default to [0,0], the origin of the newly created local map must be changed to the minimum x and y values to reflect its location on the global map. Every cell value within the local map is then defaulted to a value of 1 representing occupied using the *updateOccupancy* function.

To have the local map represent the work area polyshape, arrays of x and y check points are created with each point separated by a "step" value calculated based on the resolution of the local map. Using the *meshgrid* function and both x and y check point arrays, a grid is created which can then be overlayed on top of the work area polyshape. This grid is then checked for which points lie within the polyshape using the function *inpolygon*. The output of this function is logical meaning the point is either in or not within the polyshape. Using this as the criteria for representing the work area, all the cell occupancy values which lie within the work area polyshape are set to 0 representing unoccupied using the *setOccupancy* function. A sample result of the conversion from a polyshape object to an occupancy map is shown in Figure 28 below.

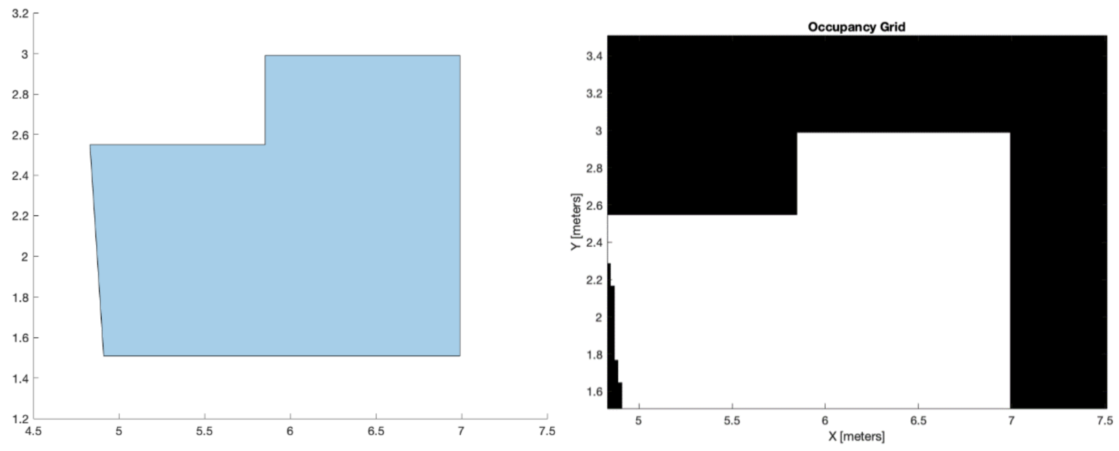


Figure 28. Sample polyshape to occupancy map conversion.

With the occupancy map of the work area now generated, the second part of the *poly2occgrid* function involves identifying viable starting corners for the robot and then selecting one of these corners based on the desired starting location criteria. This is done by generating four test points representing the top-left, bottom-left, top-right, and bottom-right corners utilizing the previously determined x and y minimum and maximum values of the work area polyshape. These points are then used to identify which polyshape vertex is closest to that corresponding test corner. For each identified vertex, a slightly enlarged polyshape of the robot is placed at each vertex and then translated in each of the four cardinal and ordinal directions to determine if the robot polyshape is fully within the boundaries of the work area polyshape. If so, this location is saved as a viable starting location. All identified starting locations for the above work area are shown in Figure 29 below.

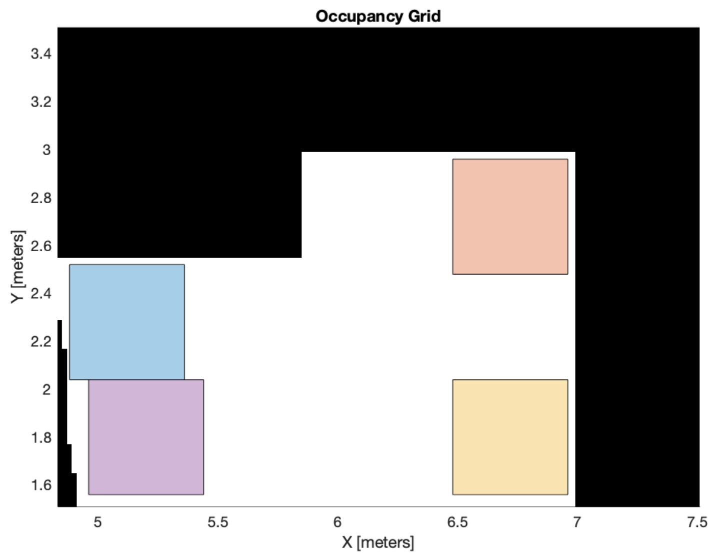


Figure 29. Sample viable starting locations.

These viable starting locations are then sorted based on the desired starting location input. The highest-ranking starting location following the sort is then selected as the starting location and returned to the main workspace along with the work area occupancy map. These outputs are then passed to the path planning algorithm to begin complete coverage and thereby concludes the strategy planning algorithm.

3. EXPERIMENTAL RESULTS AND DISCUSSION

3.1. Equipment Used

The equipment used to conduct the following experiments is a 2019 MacBook Pro with 8 GB of LPDDR3 RAM and a 2.4 GHz Quad-Core Intel Core i5 processor, MATLAB R2022b, a local version of the path planning algorithm named *cleanRoomSimTest* based on the *cleanRoom* complete coverage path planning algorithm obtainable from [40], and a modified version of the strategy planning algorithm for collecting simulation data named *roomStrategySim*.

3.2. Map Environments Used

The figures below depict the map environments utilized when conducting the following experiments. The workable area in each of the following maps is $22.95\ m^2$ for the map shown in Figure 30, $54.19\ m^2$ for the map shown in Figure 33, and $39.83\ m^2$ for the map shown in Figure 36.

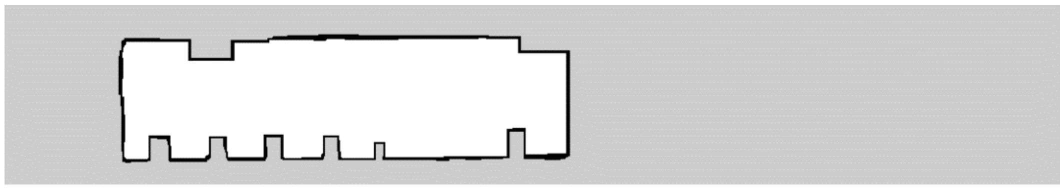


Figure 30. Map environment 1 PGM image.

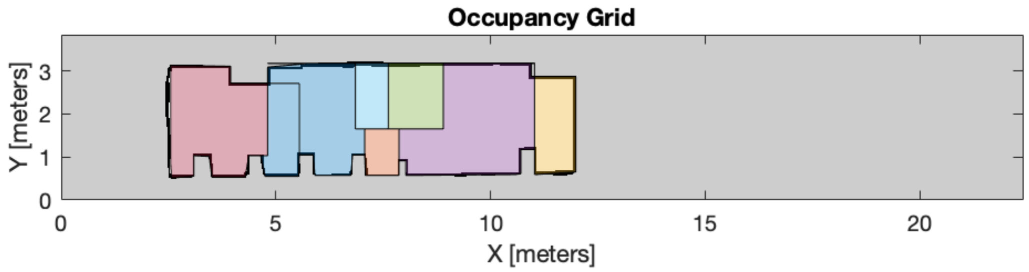


Figure 31. Map environment 1 decomposed.

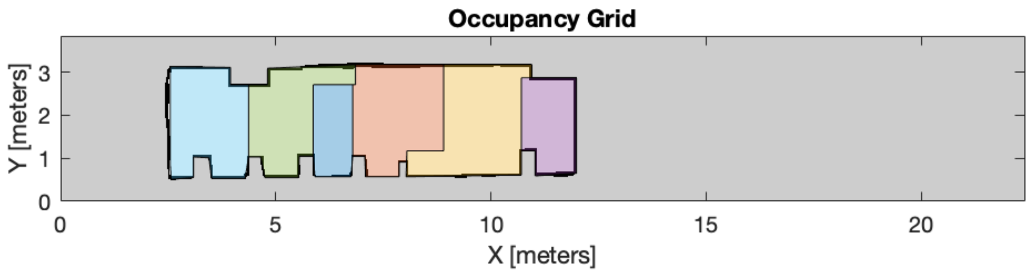


Figure 32. Map environment 1 decomposed without triangle merging.

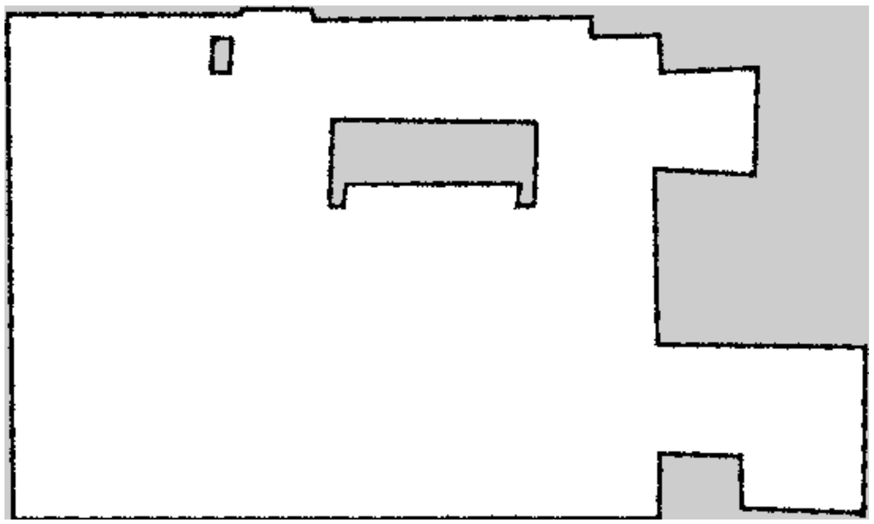


Figure 33. Map environment 2 PGM image.

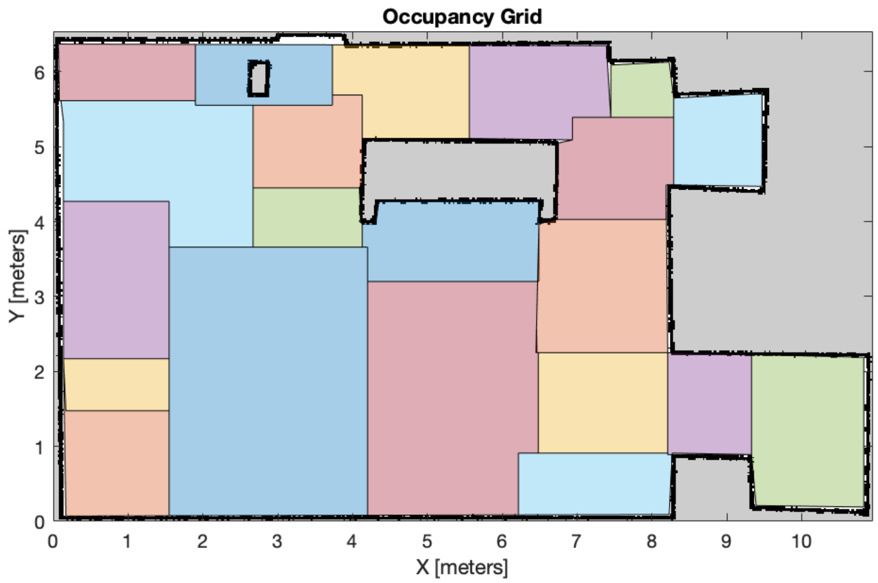


Figure 34. Map environment 2 decomposed.

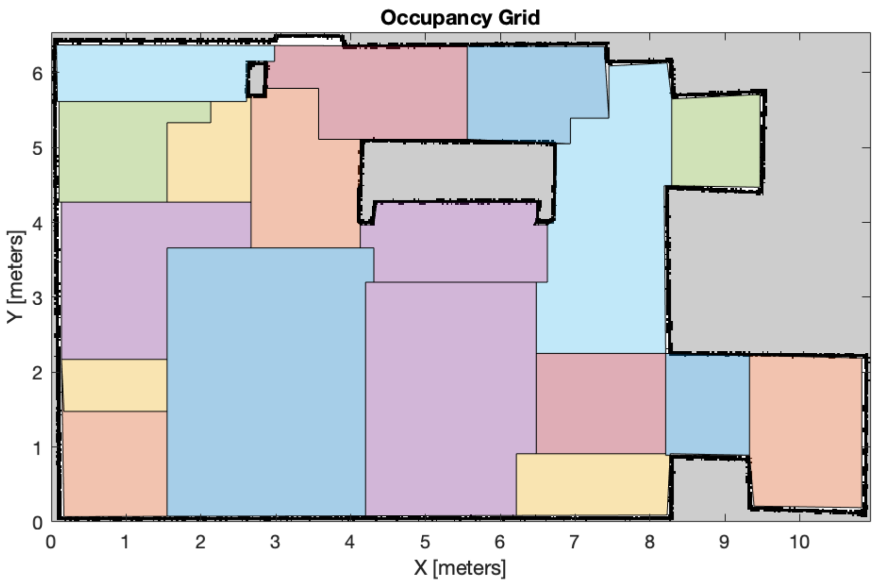


Figure 35. Map environment 2 decomposed without triangle merging.

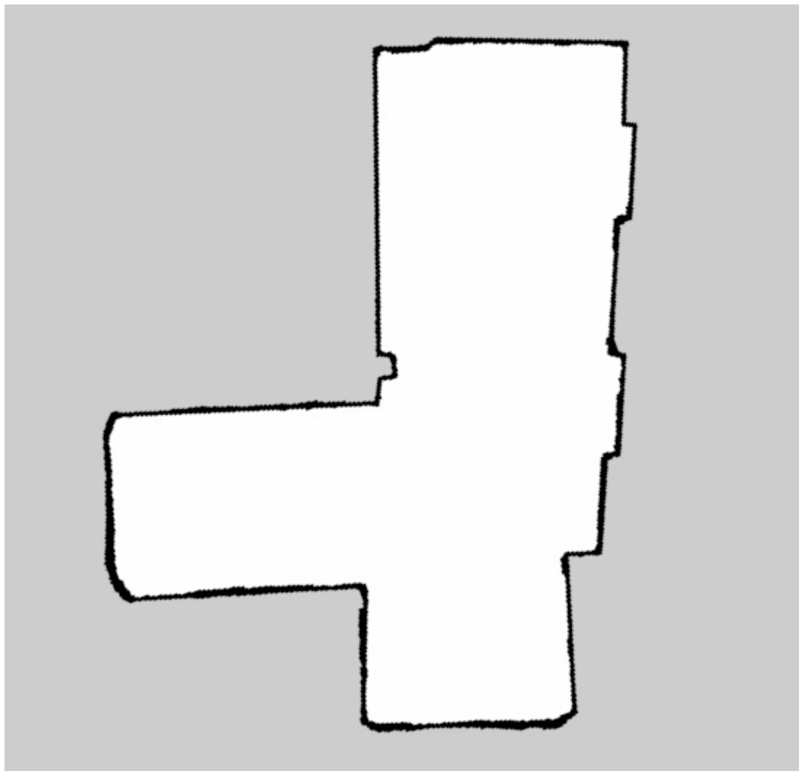


Figure 36. Map environment 3 PGM image.

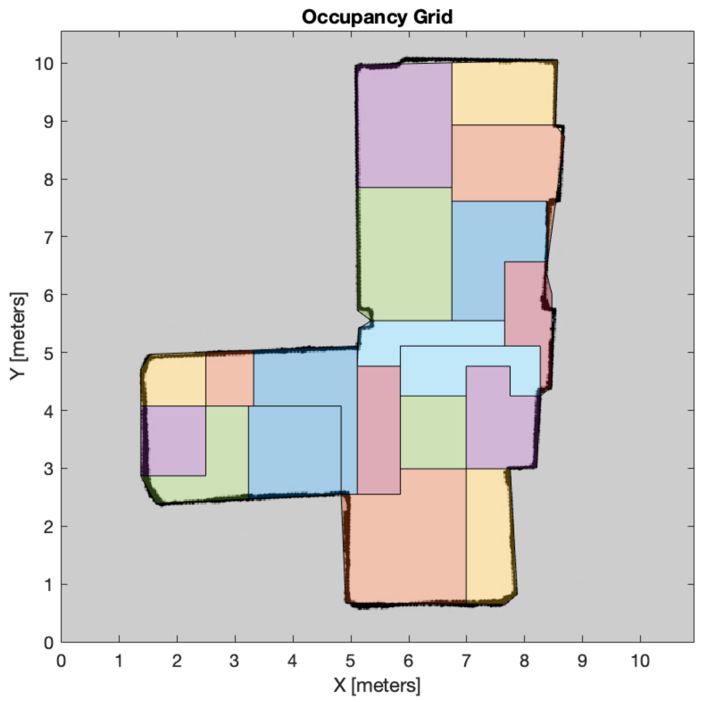


Figure 37. Map environment 3 decomposed.

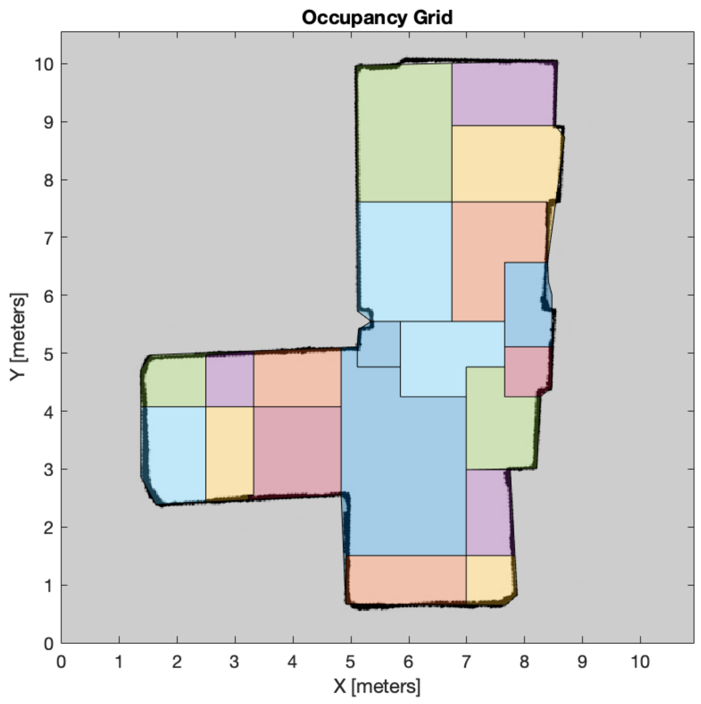


Figure 38. Map environment 3 decomposed without triangle merging.

3.3. Experiment 1: Comparing Cleaning Time and Efficiency between a Decomposed and Non-Decomposed Work Area

3.3.1. Hypothesis

It is hypothesized that constraining a path-planning algorithm to smaller, sectioned-off regions within the work area will help to decrease the amount of time spent by the path-planning algorithm performing complete coverage while also increasing the cleaning efficiency as compared to allowing the path-planning algorithm to perform complete coverage within a non-decomposed work area.

3.3.2. Procedure

The procedure for this experiment consists of simulating three cleaning methods by a single robot within the three different map environment work areas previously described. Each method is broken down into three tests respectively. For each test, the desired starting location of the path planning algorithm will be changed from the farthest viable start corner from the work area centroid, the closest viable start corner to the work area centroid, and the work area centroid itself. Method 1 will be the control experiment by assigning the path planning algorithm to the entire work area from each of the three starting locations. Method 2 involves decomposing the work area into sections and then having the path planning algorithm clean each section by starting at the starting location specific to the current test. Method 3 is similar to the previous except when decomposing the work area, the triangle merging section of the strategy planning algorithm is skipped. For each method, the amount of time required by the path planning algorithm to completely cover the work area and the overall coverage efficiency will be analyzed.

3.3.3. Results

The results below are the tabulated data collected when performing each simulation run. Nine experiments were run in total with three tests performed within each experiment resulting in 27 data points. Collected within each test was the coverage efficiency percentage, the simulated cleaning time in seconds, the average amount of overlapped or re-coverage of already cleaned spaces per section, and the runtime of the strategy planning algorithm in seconds. Following the completion of each experiment, the data values from each of the three tests are averaged in the final right-hand column. The last two rows within each table show the number of output sections generated by the strategy planning algorithm and the number of triangles generated from the triangular mesh. These values remained constant within each experiment.

In some instances, the path planning algorithm can get trapped searching for an area to cover for long periods of time referred to as a “deadlock”. To mitigate this, a deadlock counter was implemented to count the number of overlapped spaces visited before locating a new uncovered space. If the path planning algorithm is able to successfully locate a new uncovered area, the deadlock counter is reset. However if the path planning algorithm is unable to locate a new uncovered area after 30 iterations, the simulation is terminated. Terminated simulation runtime data is italicized with a following

"*" symbol indicating the number of simulations terminated as a result of a deadlock event during that specific test.

It is worth noting that the simulation runtime data is reflective of ideal conditions within a simulated environment and does not consider the time required by a physical robot to move from an ending location to a start location nor the speed of the robot performing the complete coverage.

3.3.3.1. Map Environment 1 Results

Table 1. Results for Map 1 work area with no decomposition.

	Method 1: No Decomposition			
Starting Location	Farthest	Closest	Centroid	Average
Coverage Efficiency (%)	75.17%	76%	69.39%	73.52%
Simulated Cleaning Time (s)	44.11	51.68	38.68	44.82
Overlap per Section Average	20	37	13	23.33
Strategy Planning Algorithm Runtime (s)	0.71	0.79	0.75	0.75
Number of Sections	-	-	-	1
Number of Generated Mesh Triangles	-	-	-	N/A

Table 2. Results for Map 1 work area with decomposition.

	Method 2: With Decomposition			
Starting Location	Farthest	Closest	Centroid	Average
Coverage Efficiency (%)	65.98%	64.83%	57.26%	62.69%
Simulated Cleaning Time (s)	35.43	33.45	43.59	37.49
Overlap per Section Average	2.63	2.25	5.25	3.38
Strategy Planning Algorithm Runtime (s)	22.85	23.5	25.73	24.03
Number of Sections	-	-	-	8
Number of Generated Mesh Triangles	-	-	-	36

Table 3. Results for Map 1 work area with triangle merging process skipped during decomposition.

	Method 3: Decomposition without Triangle Merging			
Starting Location	Farthest	Closest	Centroid	Average
Coverage Efficiency (%)	62.26%	77.53%	57.31%	65.7%
Simulated Cleaning Time (s)	31.38	42.57	34.03	35.99
Overlap per Section Average	2.33	3.83	4	3.39
Strategy Planning Algorithm Runtime (s)	33.6	33.3	32.6	33.17
Number of Sections	-	-	-	6
Number of Generated Mesh Triangles	-	-	-	36

3.3.3.2. Map Environment 2 Results

Table 4. Results for Map 2 work area with no decomposition.

	Method 1: No Decomposition			
Starting Location	Farthest	Closest	Centroid	Average
Coverage Efficiency (%)	85.75%	80.16%	82.87%	82.93%
Simulated Cleaning Time (s)	121.73	*110.98**	134.31	122.34
Overlap per Section Average	35	32	67	44.67
Strategy Planning Algorithm Runtime (s)	1.19	1.21	1.1	1.17
Number of Sections	-	-	-	1
Number of Generated Mesh Triangles	-	-	-	N/A

Table 5. Results for Map 2 work area with decomposition.

	Method 2: With Decomposition			
Starting Location	Farthest	Closest	Centroid	Average
Coverage Efficiency (%)	74.27%	72.15%	65.99%	70.8%
Simulated Cleaning Time (s)	97.27	92.07	*102.58**	97.31
Overlap per Section Average	2.14	2.1	4	2.75
Strategy Planning Algorithm Runtime (s)	112.61	111.2	111.23	111.68
Number of Sections	-	-	-	21
Number of Generated Mesh Triangles	-	-	-	92

Table 6. Results for Map 2 work area with triangle merging process skipped during decomposition.

	Method 3: Decomposition without Triangle Merging			
Starting Location	Farthest	Closest	Centroid	Average
Coverage Efficiency (%)	71.89%	72.22%	68.98%	71.03%
Simulated Cleaning Time (s)	92.28	86.29	99.17	92.58
Overlap per Section Average	2.22	2.11	3.33	2.55
Strategy Planning Algorithm Runtime (s)	254.56	256.19	262.09	257.61
Number of Sections	-	-	-	18
Number of Generated Mesh Triangles	-	-	-	92

3.3.3.3. Map Environment 3 Results

Table 7. Results for Map 3 work area with no decomposition.

	Method 1: No Decomposition			
Starting Location	Farthest	Closest	Centroid	Average
Coverage Efficiency (%)	82.8%	82.68%	78.56%	81.35%
Simulated Cleaning Time (s)	83.49	98.56	*87.58**	89.88
Overlap per Section Average	26	52	40	39.33
Strategy Planning Algorithm Runtime (s)	1.16	0.94	0.91	1
Number of Sections	-	-	-	1
Number of Generated Mesh Triangles	-	-	-	N/A

Table 8. Results for Map 3 work area with decomposition.

	Method 2: With Decomposition			
Starting Location	Farthest	Closest	Centroid	Average
Coverage Efficiency (%)	65.56%	64.47%	57.23%	62.42%
Simulated Cleaning Time (s)	60.58	60.89	61.09	60.85
Overlap per Section Average	1.95	2.26	2.74	2.32
Strategy Planning Algorithm Runtime (s)	25.84	25.76	26.74	26.11
Number of Sections	-	-	-	19
Number of Generated Mesh Triangles	-	-	-	33

Table 9. Results for Map 3 work area with triangle merging process skipped during decomposition.

	Method 3: Decomposition without Triangle Merging			
Starting Location	Farthest	Closest	Centroid	Average
Coverage Efficiency (%)	72.87%	63.57%	62.12%	66.19%
Simulated Cleaning Time (s)	72.63	61.77	72.64	69.01
Overlap per Section Average	2.25	2.2	3.4	2.62
Strategy Planning Algorithm Runtime (s)	28.06	36.03	26.81	30.3
Number of Sections	-	-	-	20
Number of Generated Mesh Triangles	-	-	-	33

3.3.3.4. Method Comparison

Table 10. Comparison of averaged experimental results from each method.

Method	No Decomposition	With Decomposition		Decomposition w/o Triangle Merging	
Coverage Efficiency (%)	79.26%	65.3%	-17.61%	67.64%	-14.67%
Simulated Cleaning Time (s)	85.68	65.22	-23.88%	65.86	-23.13%
Overlap per Section Average	35.78	2.81	-92.14%	2.85	-92.03%
Strategy Planning Algorithm Runtime (s)	0.97	53.94	-	107.03	+98.42%

As a note on the much lower average strategy planning algorithm runtime exhibited by the experiment with no decomposition performed, this is a result of the strategy planning algorithm only needing to identify the work area boundaries, generate an occupancy map of the work area, and then determine a viable starting location before handing off to the strategy planning algorithm for complete coverage. It was also found that deadlocks did not have a considerable impact on the collected data with only two occurring throughout the entirety of the experiment.

3.3.4. Conclusions

From reviewing the experimental results, there are two findings to be made concerning the initial hypothesis. The first finding is that by decomposing the room for complete coverage, the coverage efficiency decreased contrary to what was hypothesized. Reviewing Table 10 in the above section, it can be seen that the coverage efficiency with no decomposition is 79.26% while the coverage efficiency with both methods of decomposition is 65.3% and 67.64% respectively. Furthermore, the decomposition method in skipping the triangle merging process resulted in a 14.67% reduction in coverage efficiency while the full decomposition method led to a 17.61% reduction in coverage efficiency when compared to the control experiment with no decomposition.

The second finding is the decrease in the average simulated cleaning time experienced by both decomposition methods when compared to the control experiment with no decomposition. With an average cleaning time of 85.68 seconds for the control experiment, both the full decomposition and decomposition without triangle merging methods were able to reduce their respective cleaning times by 20.46 and 19.82 seconds which calculates to an over 23% reduction in time required for cleaning. Although agreeing with the initial hypothesis, this decrease in cleaning time could be attributed to the reduction in coverage efficiency mentioned previously as a result of the path planning algorithm completing its assigned sections in a "faster" manner. This is due to the path planning algorithm covering all spaces where it believes the robot can fit which is facilitated by generating a grid reliant on the provided starting location that determines the possible locations the path planning algorithm can propagate to. If the generated grid is offset from the given work area or does not match up accordingly to a location that the robot can visually be perceived to fit, those locations will not be covered and hence "finish" in a shorter amount of time. This is the reasoning behind varying the desired starting location for each experiment as each starting location impacts the coverage efficiency as shown within the collected data.

Some additional findings which were not anticipated are the greater than 92% reductions in average overlap per section exhibited by both decomposition methods and the 98.42% increase in strategy planning algorithm runtime of the decomposition method with the triangle merging process skipped as compared to the full decomposition process. The former finding is the most surprising however the dramatic decreases in average overlap per section could be attributed to the fact that any high overlap values incurred by any one section are averaged out by the little to no overlap values of the other sections. The latter finding was expected but in the reverse sense with the full decomposition method resulting in a higher average strategy planning algorithm runtime when compared to the method with the triangle merging process skipped. However, the experimental data proves the opposite to be true in that the decomposition method without triangle merging results in almost double the runtime of the full decomposition method. This nearly 100% increase in algorithm runtime required by the decomposition method with the triangle merging skipped could be attributed to the process by which the bounding boxes are generated – more specifically the process of removing overlaps between neighboring bounding boxes. During the full decomposition routine, the preceding triangle merging process removes many of the triangle mesh centroids leading to fewer bounding box overlaps to remove. This is most evident in Table 6 where the Map 2 environment was decomposed into a mesh of 92 triangles which required an average time of 257.61 seconds to fully run.

These experimental findings would suggest the strategy planning algorithm with triangle merging outperforms the method without triangle merging with regard to the time required by the strategy planning algorithm to fully run however both methods perform similarly with respect to coverage efficiency, simulated cleaning time, and average overlap per section. Of these three aforementioned data, the largest difference is seen in the coverage efficiency with a marginal 2.94% edge for the decomposition method without triangle merging while the differences between simulated cleaning time and average overlap per section are within one percentage point. Therefore it can be concluded that by incorporating the process of triangle merging into the decomposition method, algorithm run time can be reduced by nearly half with the current algorithm setup. Furthermore, with future modification to enhance the ability of the algorithm to select a starting location which aims to optimize coverage efficiency as opposed to being based on what the user desires may lead to increased performance to potentially match or exceed the measured coverage efficiency without decomposition.

3.4. Experiment 2: Comparing Idle Time between Dynamic and Static Task Allocation

3.4.1. Hypothesis

The hypothesis for this experiment is that by dynamically reallocating tasks amongst the available robots, the length of idle time spent by either robot after completing its assigned tasks should be considerably reduced as compared to statically allocating tasks.

3.4.2. Procedure

In this experiment, two multi-robot scenarios are simulated in the three different work areas as described previously above. For each work area, both the decomposition method with triangle merging and the decomposition method without triangle merging are applied.

Three simulations for each scenario are then conducted with each decomposition method where the desired starting location of the path planning algorithm is changed from the farthest viable start corner from the work area centroid, the closest viable start corner to the work area centroid, and the work area centroid itself. The first scenario will be a cleaning session by two robots performed without task reallocation. The second scenario will be a cleaning session by two robots performed with task reallocation. For each scenario, the time difference between the completion of tasks by one robot and the other will be examined.

3.4.3. Results

The tables below contain the resulting data collected when performing each simulation run. Twelve experiments were conducted in total, six experiments with no task reallocation and six experiments with task reallocation. Three simulations were conducted within each experiment resulting in 36 data points. Collected after each simulation run was the simulation runtime in seconds required by each robot to complete its assigned tasks. After the completion of each experiment, the idle time in seconds was calculated by subtracting the longest time for a robot to complete its assigned tasks from the shortest time in that particular simulation run. These calculated idle times are then averaged in the final column of each table.

Despite the ability of the algorithm to reallocate tasks between robots during those tests in which it is enabled, task reallocation does not always occur. In most conducted simulations task reallocation is bypassed as a result of the robot with the remaining tasks only having one task left to complete or both robots finishing their assigned tasks simultaneously. In simulations in which task reallocation did occur, the calculated idle time has been bolded.

As implemented in Experiment 1, a deadlock counter was utilized to terminate a simulation if the threshold of 30 failed attempts by the path planning algorithm to locate a new uncovered area is met. Terminated simulation runtime data is italicized with a following "*" symbol indicating the number of simulations terminated as a result of a deadlock event during that specific test. Note that the simulation runtime data is reflective of ideal conditions within a simulated environment and does not consider the time required by a physical robot to reach that particular section nor the speed of the robot performing the complete coverage.

3.4.3.1. No Task Reallocation vs Task Reallocation Results for Decomposition Method with Triangle Merging

Table 11. No task reallocation results for Map 1 decomposed with triangle merging.

Map 1 No Task Reallocation				
Starting Location	Robot 1 Completion Time (s)	Robot 2 Completion Time (s)	Idle Time (s)	Idle Time Average
Farthest	20.49	17.23	3.26	3.62
Closest	19.1	16.04	3.06	
Centroid	24.53	19.99	4.54	

Table 12. Task reallocation results for Map 1 decomposed with triangle merging.

Map 1 With Task Reallocation				
Starting Location	Robot 1 Completion Time (s)	Robot 2 Completion Time (s)	Idle Time (s)	Idle Time Average
Farthest	19.73	16.55	3.18	3.6
Closest	18.48	15.45	3.03	
Centroid	24.32	19.74	4.58	

Table 13. No task reallocation results for Map 2 decomposed with triangle merging.

Map 2 No Task Reallocation				
Starting Location	Robot 1 Completion Time (s)	Robot 2 Completion Time (s)	Idle Time (s)	Idle Time Average
Farthest	52.76	43.26	9.5	8.71
Closest	45.22	41.71	3.51	
Centroid	*57.46**	44.35	13.11	

Table 14. Task reallocation results for Map 2 decomposed with triangle merging.

Map 2 With Task Reallocation				
Starting Location	Robot 1 Completion Time (s)	Robot 2 Completion Time (s)	Idle Time (s)	Idle Time Average
Farthest	51.15	43.11	8.04	4.17
Closest	46.46	43.08	3.38	
Centroid	*52.05**	50.95	**1.1**	

Table 15. No task reallocation results for Map 3 decomposed with triangle merging.

Map 3 No Task Reallocation				
Starting Location	Robot 1 Completion Time (s)	Robot 2 Completion Time (s)	Idle Time (s)	Idle Time Average
Farthest	38.43	19.19	19.24	19.47
Closest	40.51	18.98	21.53	
Centroid	40.77	23.14	17.63	

Table 16. Task reallocation results for Map 3 decomposed with triangle merging.

Map 3 With Task Reallocation				
Starting Location	Robot 1 Completion Time (s)	Robot 2 Completion Time (s)	Idle Time (s)	Idle Time Average
Farthest	36.83	21.47	**15.36**	14.95
Closest	38.39	22.2	**16.19**	
Centroid	38.73	25.43	**13.3**	

3.4.3.2. No Task Reallocation vs Task Reallocation Results for Decomposition Method without Triangle Merging

Table 17. No task reallocation results for Map 1 decomposed without triangle merging.

Map 1 No Task Reallocation				
Starting Location	Robot 1 Completion Time (s)	Robot 2 Completion Time (s)	Idle Time (s)	Idle Time Average
Farthest	14.49	17.82	3.33	3.85
Closest	18.17	25.74	7.57	
Centroid	17.35	18.01	0.66	

Table 18. Task reallocation results for Map 1 decomposed without triangle merging.

Map 1 With Task Reallocation				
Starting Location	Robot 1 Completion Time (s)	Robot 2 Completion Time (s)	Idle Time (s)	Idle Time Average
Farthest	14.6	18.16	3.56	3.91
Closest	18.28	25.56	7.28	
Centroid	17.08	17.96	0.88	

Table 19. No task reallocation results for Map 2 decomposed without triangle merging.

Map 2 No Task Reallocation				
Starting Location	Robot 1 Completion Time (s)	Robot 2 Completion Time (s)	Idle Time (s)	Idle Time Average
Farthest	41.18	44.96	3.78	5.74
Closest	40.95	45.89	4.94	
Centroid	46.32	54.81	8.49	

Table 20. Task reallocation results for Map 2 decomposed without triangle merging.

Map 2 With Task Reallocation				
Starting Location	Robot 1 Completion Time (s)	Robot 2 Completion Time (s)	Idle Time (s)	Idle Time Average
Farthest	41.48	45.47	3.99	5.58
Closest	41.55	46.31	4.76	
Centroid	45.27	53.27	8	

Table 21. No task reallocation results for Map 3 decomposed without triangle merging.

Map 3 No Task Reallocation				
Starting Location	Robot 1 Completion Time (s)	Robot 2 Completion Time (s)	Idle Time (s)	Idle Time Average
Farthest	37.56	41.08	3.52	4.04
Closest	27.67	33.84	6.17	
Centroid	36.21	38.64	2.43	

Table 22. Task reallocation Results for Map 3 decomposed without triangle merging.

Map 3 With Task Reallocation				
Starting Location	Robot 1 Completion Time (s)	Robot 2 Completion Time (s)	Idle Time (s)	Idle Time Average
Farthest	33.72	37.56	3.84	2.96
Closest	29.89	32.95	**3.06**	
Centroid	36.43	38.4	1.97	

3.4.3.3. Average Idle Time Comparison

Table 23. Comparison of average idle times.

		Scenario		
		No Task Reallocation	With Task Reallocation	Percent Change
Method	Decomposition with Triangle Merging	10.6 sec	7.57 sec	-28.54%
	Decomposition without Triangle Merging	4.54 sec	4.15 sec	-8.68%

3.4.4. Conclusions

In comparing the average idle times for both decomposition methods within each scenario of no task allocation and with task allocation as aggregated in Table 23 above, it is evident that the full decomposition method with triangle merging exhibited the largest reduction in idle time with task allocation at 28.54% while the decomposition method without triangle merging had an 8.68% decrease in idle time with task allocation. As mentioned previously, task reallocation does not always occur despite being available. During the 18 simulations conducted with task reallocation available, task reallocation was performed only in five or 28% of the simulations ran. Furthermore, of the five task reallocations performed, four occurred during simulations utilizing the full decomposition method as opposed to the single occurrence of the decomposition method with the triangle merging process skipped.

The largest impact of task reallocation on idle time occurs in the comparison of the centroid starting location simulation results of Tables 13 and 14. In Table 13, the calculated idle time of the centroid starting location simulation with no task reallocation is 13.11 seconds while in Table 14 with task reallocation available the idle time calculated to 1.1 seconds. This suggests that rather than sitting idle for 12 seconds as Robot 2 did without task reallocation, both robots were busy completing reallocated tasks to then complete those reallocated tasks within 1.1 seconds of each other. The result is a 91.61% reduction in idle time for the specific simulation and overall leads to the average idle time between the two experiments to reduce from 8.71 seconds to 4.17 seconds, a 52.12% decrease.

The second most significant impact of task reallocation on idle time occurred during the closest starting location simulation of Tables 21 and 22. For this simulation, the resulting idle times are 6.17 and 3.06 seconds for no task reallocation and with task reallocation respectively. This change in idle time results in a 50.41% decrease in the amount of idle time spent by one robot waiting for the other robot to complete its assigned tasks. The halving of idle time for this one simulation had a 26.73% impact in reducing the average idle time from 4.04 seconds with no task reallocation to 2.96 seconds with task reallocation for this experiment.

The final three instances where task reallocation impacted idle time all occurred during the same experiment recorded in Table 16. When compared to the experimental data in Table 15 in which no task reallocation was performed, an idle time decrease of 20.17%, 24.8%, and 24.56% occurred for each of the farthest, closest, and centroid starting locations respectively. These decreases in idle times for each simulation result in a 23.22% decrease in the average idle time from 19.47 seconds with no task reallocation to 14.95 seconds with task reallocation.

These findings in the 28.54% and 8.68% reduction in idle time for both the full decomposition method and the decomposition method with the triangle merging process skipped respectively for the scenario with task reallocation corroborate the initial hypothesis made. It also further supports the use of task reallocation in situations where there are a greater number of complete coverage tasks to be completed. In the case of the experiments conducted, task reallocation occurred when the number of tasks to be completed was within the range of 19 to 21 tasks as in Maps 2 and 3 as compared to no task reallocation occurring in low task situations as in Map 1 which had a lower range of 6 to 8 tasks depending on the method of decomposition. Furthermore, real-world data gathered from a physical system would better help to support or disprove the findings made in this experiment and the comparison of the flip task allocation approach when compared to a split segment approach as this data would consider the global path traveled by each robot while the simulation does not. However due to a multitude of technical challenges, expanded on further in section 5.1 and time constraints, these data were unable to be collected at the time of writing.

4. RESEARCH CONTRIBUTIONS, TECHNICAL CHALLENGES, AND FUTURE WORK

4.1. Research Contributions

- **Novel implementation of task flipping:** The primary contribution made by this research is the novel use of task flipping. This refers to the flipping of the task segments assigned to the robots such that globally the robots work towards one another, theoretically minimizing the amount of time required by one robot to travel to a reallocated task. As only simulation experiments were conducted for this research which does not account for the amount of time required for a robot to move from one location to another, the impact of task flipping concerning idle time and overall cleaning time efficiency is not yet known. However, the framework for utilizing task flipping on a real robot system connected to a ROS network can be found within the repository mentioned in section 2.1.4.
- **Methodology for utilizing triangular mesh in a complete coverage application:** The secondary contribution made from this research is the implementation of a triangulation mesh in a complete coverage application. As mentioned previously during the literature review process, triangulation mesh is seldom utilized to decompose a work area for complete coverage. In performing this research, it is apparent why this method of decomposition is not widely used due to the angled edges, narrow corners, and in some cases too small of an area for a physical robot to fit into the triangles produced. However, what this research contributes is a methodology for

utilizing triangulation mesh as a viable method of decomposition for a work area despite the issues mentioned.

- **Idle time minimization:** A tertiary contribution made from performing this research is the utilization of task reallocation once a robot has completed its assigned tasks to minimize the amount of idle time experienced by any robot. Idle time is an important metric to reduce when utilizing multiple robots simultaneously to help reduce the amount of time required to complete the overall cleaning task. This research was able to demonstrate a methodology for reallocating unfinished tasks once and recording the impact on idle time however with some modifications could be expanded further to continuing to reallocate tasks until all tasks have been completed within a work area; especially those which contain a larger number of tasks.

4.2. Technical Challenges

4.2.1. Physical System Implementation

The largest technical challenge faced overall during this research was the implementation of the physical system. Two Sensing, Connected, Utility Transport Taxi for Level Environments (SCUTTLE) robotic platforms [41] were constructed as shown in Appendix A to perform physical system experiments whose data could then be compared to the simulation experimental results previously discussed. The issue experienced stems from the fact that the initial SCUTTLE driver controller, the program which controls the differential drive motion of the robot, was not equipped to handle encountered obstacles during movement from one waypoint to another. As this was a feature that was needed to prevent the possible collision of the two robots which would be navigating within the same work area, an alternative solution needed to be found.

One solution was to modify the existing SCUTTLE driver controller to be able to account for encountered obstacles mid-motion. However with little knowledge of the source code operation and many hours of attempting to understand and make minor modifications to the controller code, the desired result was unable to be achieved.

The second solution was the MATLAB Pure Pursuit Controller [42]. Implemented with checks for obstacles, this controller was able to successfully account for obstacles encountered midway between waypoints. However, this controller is not of the "point and go" variety in which the desired robot movement is to rotate in place toward its goal location and then move in a straight line until reaching the goal location. Due to the method in which the controller navigates between waypoints, the Pure Pursuit controller attempts to make smooth curving motions such as that of an "S" shape. Rather than move in a straight line as desired, the robot makes wide sweeping turns which can sometimes cause the robot to miss its goal location. This results in the robot making extra turns and movements that can cause it to get stuck along the wall or stuck in a perpetual circle motion forever circling the goal location. To account for these issues, various parameters of the controller were tweaked which reduced the desired "S" shape motion of the controller to some extent, and recovery features were implemented to mitigate situations in which the motions of the controller would cause the robot to get stuck. Despite best efforts, countless hours of testing, and modifications to the Pure Pursuit controller to behave in the desired "point and go" way, a sufficient solution was unable to be found and as a result a full test was never able to be completed on the physical system.

4.2.2. Boundary Identification

The most significant technical challenge faced with regard to implementing the strategy planning algorithm was the problem of how to go about identifying boundaries within the input PGM image. This process requires not only identifying the coordinates of each occupied cell within the image but also the boundary each occupied cell belongs to such that a polyshape for that boundary can be generated.

The simplest solution to this problem is to filter out all the occupied cell locations by utilizing the *find* function to locate the indices of cells with an occupancy value greater than 0.9. While this may provide all the occupied cell locations, they are not separated by boundary which leads to the issue of needing to identify the cells which make up the edge of each boundary. Furthermore, most boundary edges are thicker than one layer of occupied cells which arises yet another issue of identifying the cell locations which directly border the work environment contained within the image.

A second solution attempted for this problem was the drawing of rays in an inward direction from the outermost cell in each column and row. In this case the locations of each occupied cell bordering the work environment could be found. However, this process was very time intensive in looping through each cell within each row and column as well as the recording of duplicate points and the inability to easily distinguish which boundary each cell belonged to.

A third solution attempted was more focused on the desired outcome which is the identification of the occupied cell locations such that a polyshape of that boundary could be created. Generating a polyshape simply requires knowing the vertex coordinates of the shape in question and as such a method of identifying the work environment corners within the PGM image was implemented. This method included setting cases that describe a corner and checking each occupied cell if it matched one of the cases. For example, a top left corner can be described as a 2×2 grid of cells in which the bottom right cell is unoccupied while the remaining three cells are occupied. While somewhat successful in identifying corner locations, due to the many edge cases in how a corner can be described led to some corners not being identified. This resulted in the final polyshape becoming an inaccurate representation of the work environment and this solution being replaced. In hindsight, this attempted solution could have been achieved with the implementation of a corner detection algorithm such as the one described in [43].

The final solution settled upon is the use of the MATLAB function *bwboundaries*. This function performs the desired identification of occupied cell locations and the denoting of which cells belong to which boundary. Albeit without its drawbacks as it has been observed to be very particular in the images it identifies boundaries within and also requires interior obstacles to correctly identify the work environment or else it may be misidentified if free of obstacles. It is the use of this function that requires the need for the input PGM image to be edited however in an ideal implementation there should be little to no editing of the PGM image required by the algorithm user. A potential alternative to *bwboundaries* is the MATLAB function *regionprops* although further research and testing are required to validate its usefulness regarding the problem of boundary identification.

4.2.3. Parallel Computing

A second technical challenge encountered was the implementation of SPMD. Having no prior parallel computing experience, SPMD proved challenging to utilize in the sharing of information between workers which often resulted in an error. It was found that specific data, such as flag variables or reallocated tasks, could be shared between workers utilizing a “tag” in the *spmdSend* and *spmdReceive* functions. This in conjunction with the careful placement of *spmdBarrier* throughout the workers’ program, which was determined based on if data is either being sent or received by each worker, the sharing of information between workers was able to be made mostly error-free. However in some instances, the SPMD block would still result in error when a message sent by one worker fails to be received by the other worker. This error was found to occur when both workers would finish their assigned tasks simultaneously. This led to the inclusion of a try-catch block added around the SPMD code block to catch this potential error for cases in which occurs.

4.2.4. ROS Implementation

A technical challenge worth mentioning regarding this research was the implementation of the Robot Operating System; often referred to as “ROS”. This operating system is based on nodes and topics in which nodes can either publish or subscribe to a topic to respectively send data to or receive data posted to that topic via messages. ROS was utilized to construct a wireless network consisting of both physical robots and a master laptop as nodes to facilitate the control of both robots from a single computer.

One issue was the simultaneous display of both robots within the ROS visualization software Rviz which is used to visualize the real-time physical location of the robot within its environment utilizing sensors such as that of a LiDAR. The default SCUTTLE library already featured a code block containing the ability to display one robot within Rviz however creating a duplicate of this code block and modifying associated node and topic names failed to resolve the issue. What resolved the issue instead was the use of namespaces which when incorporated proved to be successful in differentiating the robots within Rviz allowing for the simultaneous visualization of both robots within the environment.

A second issue was the utilization of ROS to publish messages within the SPMD block. The problem stemmed from the location by which the ROS nodes and topics should be initialized for each robot. From previous experience in initializing ROS nodes and topics, this is typically done before the main code block which in this case would translate to initializing the nodes and topics before the SPMD block. This approach however did not work with SPMD. To rectify this problem, further research was conducted on this topic however there exists little to no online documentation on a solution to this problem. Attempts at finding a solution involved initializing the nodes and topics directly after *spmd* which is the keyword that heads the SPMD code block, within each worker directly following the *if* statement which specifies the worker code blocks, and utilizing "parallel.pool.Constant" which creates a constant object, in this instance a node, which can be shared between workers. It was later determined that most of the pieces to successfully be able to achieve a solution to the problem were found however the missing piece was the reference to the individual SPMD workers themselves when setting up the nodes and topics. A proper solution to the problem of publishing ROS messages within an SPMD block can be seen in the Code Example 10.

```
if spmdIndex == 1
    % set ROS node constant
    node{spmdIndex} = parallel.pool.Constant(ros.Node('/Robot1',"http://X.X.X.X:11311"));

    % set ROS publisher constants
    map_pub{spmdIndex} = parallel.pool.Constant(ros.Publisher(node{spmdIndex}.Value...
        ,'/localMap',"nav_msgs/OccupancyGrid","DataFormat","struct"));

    % set ROS message types
    map_msgType{spmdIndex} = rosmessage("nav_msgs/OccupancyGrid","DataFormat","struct");

    % set ROS messages
    map_msg{spmdIndex} = rosWriteOccupancyGrid(map_msgType{spmdIndex},localMap);

    % send ROS messages
    send(map_pub{spmdIndex}.Value,map_msg{spmdIndex})
end
```

Code Example 10. Publishing ROS Message within SPMD

The presented solution outlines the steps for the sending of an occupancy map message type over ROS however can be modified to send any message type. While the described ROS network was ultimately not utilized with regard to the research presented, a considerable amount of time and effort went into its implementation. As a result, the strategy planning algorithm to be used in tandem with a physical system features a framework for the sending of ROS messages to be utilized in future research.

4.3. Future Work

- **Physical system operation and data collection:** The first and primary focus area for future work should be in getting the physical system operational such that real-world data collection can be conducted. This is critical as such real-world data would consider robot speed and distance traveled between tasks, of which the former would also need to be factored into the simulations ran once this value has been set as it impacts the cleaning time required while the latter would help to better prove or disprove the usefulness of the flipping task allocation method presented. It is with great optimism that future work in implementing the physical system can forego the issues experienced and begin at a stage where the physical system has been constructed and a framework for its operation has been provided.
- **Scaling:** A second focus area for future work should be in scaling the algorithm to account for a larger number of robots with N number of robots as the eventual end goal. The algorithm was created with this end goal in mind however the SPMD portion currently only supports the reallocation of tasks between two robots. Given the current method implemented, the greater number of robots added, the more complex the task allocation section of the algorithm becomes as there are that many more connections between robots that would need to be made. Perhaps a future approach would be to rid of the connections between robots and instead have the robots report their status directly to the main computer being utilized.
- **Improved task reallocation:** The final focus area of future work should be in improving the task reallocation method. One way is by implementing an iterative

approach to the dynamic reallocation of tasks. As the current task reallocation program has been written, task reallocation will only occur once throughout the duration of the cleaning process. However with some modifications to the program, an iterative approach to the dynamic reallocation of tasks until all tasks have been completed should be able to be achieved. A second way is by implementing a method of saving the progress of the current task when task reallocation is performed. When tasks are reallocated between robots in the current program, the cleaning process for the task stopped on is started over. Rather than spend time overlapping already cleaned portions of the task stopped on, the already cleaned area should be saved such that following task reallocation the robot can continue its progress in completing the task as if it was never interrupted.

5. CONCLUSIONS

In conclusion, this thesis presents a strategy planning algorithm for a complete coverage application by either one or multiple robots. It features the utilization of triangulation mesh in a complete coverage application as well as the novel implementation of the flipping of task segments for both the original task assignment and during task reallocation. From the simulated experiment data, the algorithm was found to marginally decrease the amount of idle time experienced by a robot up to 28% and considerably reduce the amount of overlap per section by greater than 90%. The impact of flipping task segments in a real-world application remains to be observed yet these simulated findings, which may eventually be corroborated by real-world data, provide a basis for the potential real-world effectiveness of the strategy planning algorithm presented within this thesis.

REFERENCES

1. Wang, M. *iRobot Roomba s9+: price, specs, feature and release date*. 2019 [cited 2022 August 25]; Available from: https://www.gearbest.com/blog/new-gear/irobot-roomba-s9-price-specs-feature-and-release-date-5384.

2. Karcz, A. *The Roborock S7 MaxV Ultra Is The Greatest Robovac System Ever*. 2022 [cited 2022 August 25]; Available from: https://www.forbes.com/sites/anthonykarcz/2022/03/31/the-roborock-s7-maxv-ultra-is-the-greatest-robovac-system-ever/.

3. Romeo, J. *How Amazon Robotics Helps Fill Orders, Create Jobs*. 2021 [cited 2022 August 25]; Available from: https://www.robotics247.com/article/how_amazon_robotics_helps_fill_orders_create_jobs.

4. Bonifacic, I. *Tokyo Olympics opening ceremony included a light display with 1,800 drones*. 2021 [cited 2022 August 25]; Available from: https://www.engadget.com/tokyo-olympics-drone-display-183750593.html?guccounter=1.

5. Walter, J. *This Swarm of Search and Rescue Drones Can Explore Without Human Help*. 2019 [cited 2022 August 25]; Available from: https://www.discovermagazine.com/technology/this-swarm-of-search-and-rescue-drones-can-explore-without-human-help.

6. Brambilla, M., et al., *Swarm robotics: a review from the swarm engineering perspective.* Swarm Intelligence, 2013. **7**(1): p. 1-41.

7. Perry, C. *The 1,000-robot swarm*. 2014 [cited 2022 August 25]; Available from: https://news.harvard.edu/gazette/story/2014/08/the-1000-robot-swarm/.

8. Đakulovic, M. and I. Petrovic, *Complete coverage path planning of mobile robots for humanitarian demining.* Industrial Robot: An International Journal, 2012. **39**(5): p. 484-493.

9. Dakulović, M., S. Horvatić, and I. Petrović, *Complete coverage D* algorithm for path planning of a floor-cleaning mobile robot.* IFAC Proceedings Volumes, 2011. **44**(1): p. 5950-5955.

10. Hodgkin, A.L. and A.F. Huxley, *A quantitative description of membrane current and its application to conduction and excitation in nerve.* The Journal of physiology, 1952. **117**(4): p. 500.

11. Grossberg, S., *Nonlinear neural networks: Principles, mechanisms, and architectures.* Neural networks, 1988. **1**(1): p. 17-61.

12. Luo, C. and S.X. Yang, *A bioinspired neural network for real-time concurrent map building and complete coverage robot navigation in unknown environments.* IEEE Transactions on Neural Networks, 2008. **19**(7): p. 1279-1298.

13. Luo, C. and S.X. Yang. *A real-time cooperative sweeping strategy for multiple cleaning robots*. in *Proceedings of the IEEE Internatinal Symposium on Intelligent Control*. 2002. IEEE.

14. Sun, B., et al., *Complete coverage autonomous underwater vehicles path planning based on glasius bio-inspired neural network algorithm for discrete and centralized programming.* IEEE Transactions on Cognitive and Developmental Systems, 2018. **11**(1): p. 73-84.

15. Luo, C., et al. *A computationally efficient neural dynamics approach to trajectory planning of an intelligent vehicle*. in *2014 International Joint Conference on Neural Networks (IJCNN)*. 2014. IEEE.

16. Weisstein, E.W. *Voronoi diagram, From MathWorld–A Wolfram Web Resource*. 2009 [cited 2022 August 31]; Available from: https://mathworld.wolfram.com/VoronoiDiagram.html.

17. Nair, V.G. and K. Guruprasad, *GM-VPC: An algorithm for multi-robot coverage of known spaces using generalized Voronoi partition.* Robotica, 2020. **38**(5): p. 845-860.

18. Fu, J.G.M., T. Bandyopadhyay, and M.H. Ang. *Local Voronoi decomposition for multi-agent task allocation*. in *2009 IEEE International Conference on Robotics and Automation*. 2009. IEEE.

19. Hu, J., et al., *Voronoi-based multi-robot autonomous exploration in unknown environments via deep reinforcement learning.* IEEE Transactions on Vehicular Technology, 2020. **69**(12): p. 14413-14423.

20. Hu, J., B. Lennox, and F. Arvin. *Collaborative coverage for a network of vacuum cleaner robots*. in *Towards Autonomous Robotic Systems: 22nd Annual Conference, TAROS 2021, Lincoln, UK, September 8–10, 2021, Proceedings*. 2021. Springer.

21. Burns, J., *Centroidal voronoi tessellations.* Centroidal Voronoi Tessellations, 2009.

22. Galceran, E. and M. Carreras, *A survey on coverage path planning for robotics.* Robotics and Autonomous systems, 2013. **61**(12): p. 1258-1276.

23. Acar, E.U. and H. Choset, *Sensor-based coverage of unknown environments: Incremental construction of morse decompositions.* The International Journal of Robotics Research, 2002. **21**(4): p. 345-366.

24. Acar, E.U., H. Choset, and J.Y. Lee, *Sensor-based coverage with extended range detectors.* IEEE Transactions on Robotics, 2006. **22**(1): p. 189-198.

25. Choset, H., *Coverage of known spaces: The boustrophedon cellular decomposition.* Autonomous Robots, 2000. **9**: p. 247-253.

26. Acar, E.U., et al., *Morse decompositions for coverage tasks.* The international journal of robotics research, 2002. **21**(4): p. 331-344.

27. Zhou, P., et al. *Complete coverage path planning of mobile robot based on dynamic programming algorithm*. in *2nd international conference on electronic and mechanical engineering and information technology*. 2012.

28. Viet, H.H., et al., *BA*: an online complete coverage algorithm for cleaning robots.* Applied intelligence, 2013. **39**: p. 217-235.

29. Oh, J.S., et al., *Complete coverage navigation of cleaning robots using triangular-cell-based map.* IEEE Transactions on Industrial Electronics, 2004. **51**(3): p. 718-726.

30. Meysami, A., et al., *Investigating the impact of triangle and quadrangle mesh representations on AGV path planning for various indoor environments: With or without inflation.* Robotics, 2022. **11**(2): p. 50.

31. Liu, Y. and Y. Jiang, *Robotic path planning based on a triangular mesh map.* International Journal of Control, Automation and Systems, 2020. **18**(10): p. 2658-2666.

32. Pang, B., et al., *A swarm robotic exploration strategy based on an improved random walk method.* Journal of Robotics, 2019. **2019**.

33. Le, A.V., et al., *Coverage path planning using reinforcement learning-based TSP for hTetran—a polyabolo-inspired self-reconfigurable tiling robot.* Sensors, 2021. **21**(8): p. 2577.

34. Karapetyan, N., et al. *Efficient multi-robot coverage of a known environment*. in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2017. IEEE.

35. Karapetyan, N., et al. *Multi-robot dubins coverage with autonomous surface vehicles*. in *2018 IEEE International Conference on Robotics and Automation (ICRA)*. 2018. IEEE.

36. Longa, S. *Flip Task Strategy Planning Algorithm Repository*. 2023; Available from: https://github.com/steveo11780/FlipTaskStrategyPlanning.

37. *Traveling Salesman Problem: Solver-Based*. [cited 2023 March 19]; Available from: https://www.mathworks.com/help/optim/ug/travelling-salesman-problem.html.

38. Shure, L. *Loren's Excellent Adventure: Maps, Graphs, and Polygons*. 2017 [cited 2023 March 19]; Available from: https://blogs.mathworks.com/loren/2017/10/12/lorens-excellent-adventure-maps-graphs-and-polygons/.

39. *Choose Between spmd, parfor, and parfeval*. [cited 2023 March 19]; Available from: https://www.mathworks.com/help/parallel-computing/choose-spmd-parfor-parfeval.html.

40. Ewelike, C. *Clean Room Complete Coverage Path Planning Algorithm Repository*. 2023; Available from: https://github.com/victor056/CompleteCoveragePathPlanning.

41. *Welcome to the SCUTTLE Community*. [cited 2023 March 19]; Available from: https://www.scuttlerobot.org.

42. *Pure Pursuit Controller*. [cited 2023 March 19]; Available from: https://www.mathworks.com/help/nav/ug/pure-pursuit-controller.html.

43. Hernández, C.A.V. and F.A.P. Ortiz, *A Corner Detector Algorithm for Feature Extraction in Simultaneous Localization and Mapping*. Journal of Engineering Science & Technology Review, 2019. **12**(3).

APPENDIX A: SCUTTLE ROBOT PLATFORM

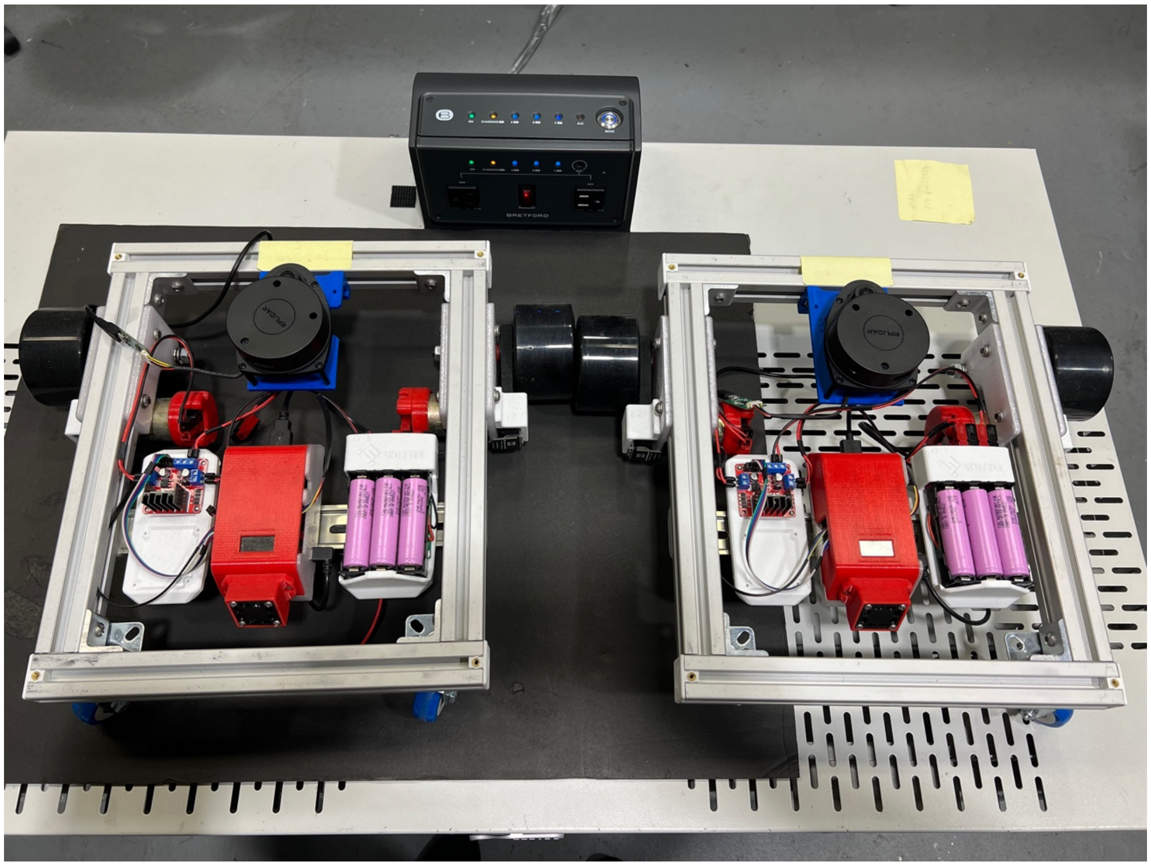


Figure 39. Top view of both constructed SCUTTLE robots.

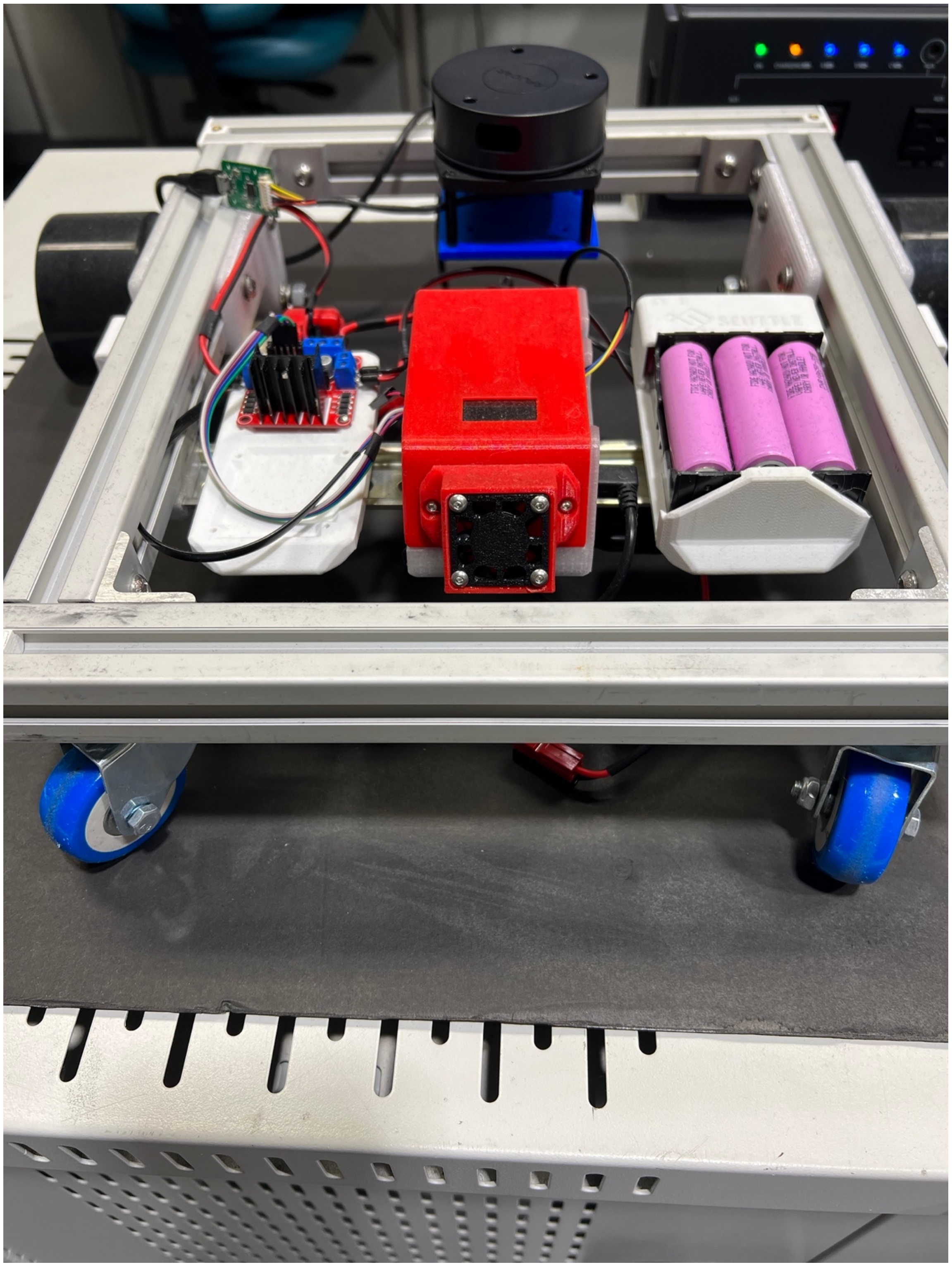


Figure 40. SCUTTLE front view.

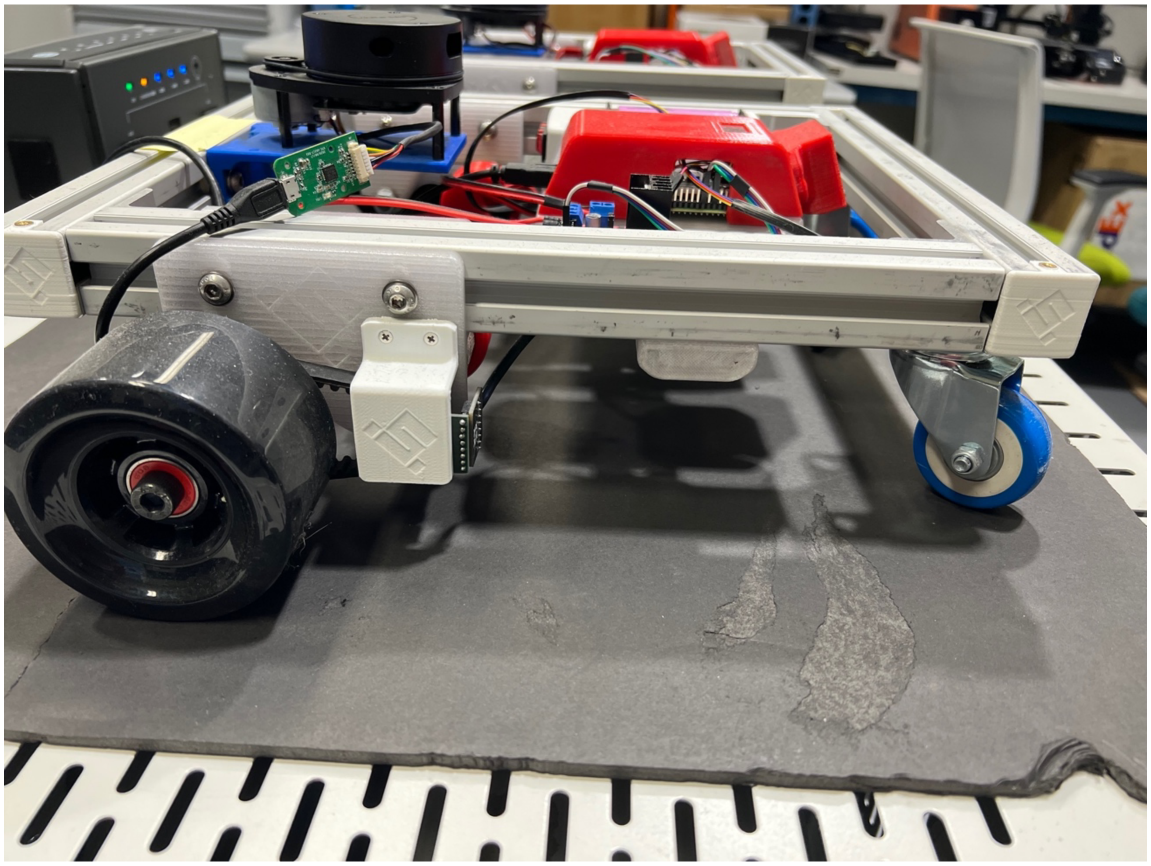


Figure 41. SCUTTLE side view.

APPENDIX B: MAIN SCRIPT

The *roomStrategy* algorithm has been formatted for both simulation and real-world applications. For simulations, the *roomStrategySim* main script is to be used along with the supplementary *cleanRoomSimTest* function in Appendix D. For use with a real-world robot, the *roomStrategyReal* main script is to be utilized. Note that the ROS network implementation of the real-world script has not been extensively tested and should be approached as a starting point for future research involving the use of a ROS network.

I. roomStrategySim

Setup

```
clearvars; close all; % clear all workspace variables and close all figures
```

% INPUT PARAMETERS

```
rob.size = 12;  % distance from robot center to edge midpoint in IJ coordinates
rob.diag = 14;  % distance from robot center to corner in IJ coordinates

min_triEdge = 2; % target minimum edge length for triangulation mesh

cornerLoc = 1;  % desired starting location of path planning algorithm.
Valid values for this parameter are:
% 1 (farthest valid starting corner from polyshape centroid)
% 2 (closest valid starting corner from polyshape centroid)
% 3 (polyshape centroid)

editedPGM_path = "/Users/~";  % file path of post-edited lidar map in PGM format

numRobots = 1; % number of robots to be utilized in decomposed work area
```

% ADDITIONAL PARAMETERS

```
mergeTri = true; % false skip triangle merging, true perform triangle merging (default)
if mergeTri; minTri.base = .5; minTri.height = .5; end % minumum values for base and height
when determining triangles to merge
```

```
 rosSetup = false; % false skip connecting to ROS network (default), true connect to ROS network

 % due to potential differences between the raw occupancy map origin and
 % the default occupancy map origin in MATLAB (0,0), this difference can be
 % corrected by importing the raw occupancy map
 importRawMap = false; % false skip importing raw occupancy map (default), true import raw
occupancy map

 wholeRoom = false; % false perform map decomposition (default), true do not perform map
decomposition

 enableRetasking = true; % false do not reallocate tasks, true reallocate tasks (default)

 resolution = 50;   % occupancy map resolution in cells per meter

 mergeRob.size = rob.size+3; % padded robot size to consider when identifying polygons to merge
 mergeRob.diag = rob.diag+3; % padded robot diagonal length to consider when identifying
polygons to merge
```

% CONNECT TO ROS NETWORK

```
 if rosSetup
     rosshutdown()   % make sure any ROS sessions are closed
     rosinit("http://X.X.X.X:11311") % initialize ROS network and connect to host
 end
```

% IMPORT RAW MAP

```
 if importRawMap
     try
         % try to subscribe to /map topic and receive the sent message
         sub = rossubscriber("/map",DataFormat='struct');
         msg = receive(sub);

         % generate an occupancy map with the received map message
         rawMap = rosReadOccupancyGrid(msg);
         show(rawMap)    % visually inspect the imported raw occupancy map
     catch
         % if unable to subscribe to /map topic, load raw map from local save
         savedMap = load('ros_occMap.mat');
         rawMap = savedMap.occupancyMapObj;
         show(rawMap); clear savedMap;
     end
```

```
else
    rawMap = [];
end
```

Image Processing

```
tic
editedPGM = imread(editedPGM_path); % load in pgm image of post edited lidar scan

% normalize the image to values between 0 and 1 then convert to occupancy
% values by subtracting from 1
editedMap_occ = 1 - double(editedPGM)/255;

% generate an occupancy map from the occupancy values
editedMap = occupancyMap(editedMap_occ,resolution);
show(editedMap) % visually inspect the generated occupancy map
```

% IDENTIFY EXTERIOR BOUNDARY AND INTERIOR OBSTACLE BOUNDARIES

```
[exteriorIJ, interiorIJ] = identifyBoundaries(editedPGM);

% determine if the exterior boundary has been misidentified as an interior obstacle,
% if so set the interior obstacle as the exterior boundary
obsNum = fieldnames(interiorIJ);    % determine number of identified interior obstacles
if (length(exteriorIJ) < 10) && (numel(obsNum) == 1)
    % reduce the number of points describing the exterior boundary
    exteriorIJ = reducepoly(interiorIJ.obs1,.01);
    removeObs = 0;  % do not remove interior obstacles
else
    removeObs = 1;  % do remove interior obstacles
end

% convert the exterior boundary points from IJ to XY coordinates
exteriorXY = grid2world(editedMap,exteriorIJ);
```

% POLYSHAPE REPRESENTATION

```
% generate a polyshape of the room from the exterior boundary points
room = polyshape(exteriorXY(:,1),exteriorXY(:,2));
plot(room)

% if applicable, remove the identified interior obstacles from the room polyshape
if removeObs == 1
    for i = 1:numel(obsNum)
```

```
            % convert the identifed interior obstacles points from IJ to XY coordinates
            interiorXY = grid2world(editedMap,interiorIJ.(obsNum{i}));

            % reduce the number of points describing the obstacle
            reducedObs = reducepoly(interiorXY,.01);

            % generate a polyshape representing the obstacle
            obs = polyshape(reducedObs(:,1),reducedObs(:,2));

            % remove the obstacle polyshape from the room polyshape
            room = subtract(room,obs);
        end
        plot(room); % visually inspect the resulting polyshape
    end
```

Map Decomposition

```
    if wholeRoom
        programTime = toc;

        % get occupancy map and starting location of room
        [localMap, startXY] = poly2occgrid(room, 0, rawMap, rob, cornerLoc, resolution);

        % convert starting location to IJ coordinates
        startIJ = world2grid(localMap,startXY);

        % call path planning algorithm to perform complete coverage
        figure; deadlock_cntr = 0;
        [localMap,time,turnCount,overlap,pathLength,deadlock_cntr] =
cleanRoomSimTest(localMap,startIJ,deadlock_cntr);

        % determine the number of cleaned and uncleaned cells
        localOcc = occupancyMatrix(localMap);
        clean = length(find((localOcc>.4)&(localOcc<.6)));
        unclean = length(find(localOcc<.1));

        % display simualtion data
        fprintf("Simulation runtime: %.2fs",time)
        fprintf("Turn count: %d",turnCount)
        fprintf("Overlap: %d",overlap)
        fprintf("Path length: %.2f",pathLength)

        covEff = clean / (unclean+clean);
        fprintf("Coverage efficiency: %.2f%%",covEff*100)
        fprintf("Number of deadlock events: %d",deadlock_cntr)
        fprintf("Strategy planning algorithm runtime: %.2fs",programTime)
    else
        % decompose polyshape using triangular mesh
        tri = triDecomposition(room, min_triEdge);

        % determine an initial optimal path passing through all triangle centroids using
        % Traveling Salesman Problem (TSP) algorithm
        [Gdir, tri.nodeList] = directedTSP(tri, room.Vertices);

        % overlay triangular mesh and identified optimal path on the edited map
        show(editedMap); hold on;
        plot(tri.shape)
        hGraph =
plot(Gdir,'XData',tri.centroid(:,1),'YData',tri.centroid(:,2),'LineStyle','none','NodeLabel',{});
        highlight(hGraph,Gdir,'LineStyle','-')
        hold off
        if mergeTri
            % determine if a triangle needs to be merged
            tri2merge = identifyTri2merge(tri, minTri)
```

% MERGE IDENTIFIED TRIANGLES

```
            % configure waitbar to monitor merging process
            triWait = waitbar(0,'Initializing Merging Process','Name','Merging Triangles...',...
                'CreateCancelBtn','setappdata(gcbf,''canceling'',1)');

            % configure a cancel button on waitbar
            setappdata(triWait,'canceling',0);

            offset = 0; % counter to track the number of nodes removed from the original node list
            for i = 1:length(tri2merge)
                % check if cancel button has been pressed
                if getappdata(triWait,'canceling'); break; end

                % identify node to be merged by subtracting the offset from the current node value
                node2merge = tri2merge(i,1)-offset;

                % update waitbar and message
                waitbar(i/length(tri2merge),triWait,sprintf('Merging triangle %d of
%d',i,length(tri2merge)))
```

```
              % fprintf('%d\n',node2merge+offset) % display current node being merged

              % get current polygon connectivity
              Gtri = polyConn(tri.shape);

              % determine which neighboring node to merge with
              neighbor2merge = polyNeighbor(Gtri, node2merge, tri.shape);

              % merge neighbor node with current triangle node
              neighborTri = tri.shape(neighbor2merge);
              currentTri = tri.shape(node2merge);
              tri.shape(neighbor2merge) = union(neighborTri,currentTri);    % save resulting
merged polygon to neighbor node index in shape list

              offset = offset+1;  % increment offset value
              tri.shape(node2merge) = [];   % remove merged triangle node from shape list
          end

          delete(triWait) % delete waitbar

          % get current polygon connectivity post merge
          Gtri = polyConn(tri.shape);

          % update centroid values of new polyshapes
          [x,y] = centroid(tri.shape);
          tri.centroid = [x;y]';

          % visually inspect polygon connectivity
          show(editedMap); hold on;
          plot(tri.shape)
          plot(Gtri,'XData',tri.centroid(:,1),'YData',tri.centroid(:,2))
          hold off;

          % determine a final optimal path passing through all triangles using TSP algorithm
          [Gdir, tri.nodeList] = directedTSP(tri, room.Vertices);

          % visually inspect resulting TSP path
          show(editedMap); hold on;
          plot(tri.shape)
          hGraph =
plot(Gdir,'XData',tri.centroid(:,1),'YData',tri.centroid(:,2),'LineStyle','none');
          highlight(hGraph,Gdir,'LineStyle','-')
          hold off
```

```
    end
```

% GENERATE BOUNDING BOXES

```
    tri.nodeList(end,:) = []; % remove repeated node at end of node list

    % convert merged triangle polygons into bounding boxes
    for i = 1:length(tri.nodeList)

        % obtain bounding box XY limits
        [xlim,ylim] = boundingbox(tri.shape(tri.nodeList(i,1)));

        % identify min and max of XY limits
        xmin = min(xlim); xmax = max(xlim);
        ymin = min(ylim); ymax = max(ylim);

        % generate a polyshape of the bounding box and add to shape list
        polyMin = [xmin,ymin;xmax,ymin;xmax,ymax;xmin,ymax];
        polybox.shape(i) = polyshape(polyMin);
    end

    % determine centroid values of bounding boxes
    [x,y] = centroid(polybox.shape);
    polybox.centroid = [x;y]';

    % create a polyshape representing the area outside the room
    [xlim,ylim] = boundingbox(room);
    xmin = min(xlim); xmax = max(xlim);
    ymin = min(ylim); ymax = max(ylim);
    polyMin = polyshape([xmin,ymin;xmax,ymin;xmax,ymax;xmin,ymax]);
    roomOutside = subtract(polyMin,room);
    % plot(roomOutside)
```

% CLEAN UP BOUNDING BOXES

```
    % configure waitbar to monitor clean up process
    cleanWait = waitbar(0,'Initializing Clean Up Process','Name','Cleaning Up Bounding
Boxes...',...
        'CreateCancelBtn','setappdata(gcbf,''canceling'',1)');

    % configure a cancel button on waitbar
    setappdata(cleanWait,'canceling',0);

    % clean up bounding boxes by removing overlaps and areas outside the room
    figure; show(editedMap); hold on;
    offset = 0;     % counter to track the number of bounding boxes removed from the original
shape list
    for i = 1:length(polybox.shape)-offset
        % check if cancel button has been pressed
        if getappdata(cleanWait,'canceling'); break; end

        % identify bounding box to be cleaned up by subtracting the offset from the current
index value
        polyIdx = i - offset;

        % update waitbar and message
        waitbar(i/length(polybox.shape)-offset,cleanWait,sprintf('Cleaning up bounding box %d
of %d',i,length(polybox.shape)-offset))

        % fprintf('%d\n',polyIdx+offset)  % display current bounding box index being cleaned

        currentpoly = polybox.shape(polyIdx); % identify the current bounding box

        % determine the region of intersection (ROI) between the current bounding box
        % and the area outside the room then remove it from the current bounding box
        roomintsec = intersect(currentpoly,roomOutside);
        currentpoly = subtract(currentpoly,roomintsec);

        Gbox = polyConn(polybox.shape); % get current polygon connectivity

        % determine current bounding box neighbors
        [row,col] = find(Gbox.Edges.EndNodes==polyIdx);
        index = [row col]; neighbors = [];

        for j = 1:height(index)
            if index(j,2) == 1
                neighbors(j,1) = Gbox.Edges.EndNodes(index(j,1),2);
            elseif index(j,2) == 2
                neighbors(j,1) = Gbox.Edges.EndNodes(index(j,1),1);
            end
        end

        % remove current bounding box neighbor overlaps
        for k = 1:height(neighbors)
            % determine the ROI between current bounding box and neighbor
            polyintsec = intersect(currentpoly,polybox.shape(neighbors(k)));
            % check if the ROI is valid
            if height(polyintsec.Vertices) == 0
                continue;
            else
                % remove the ROI from the current bounding box if its area is
                % greater than the neighbor, else remove the ROI from the neighbor
                if area(currentpoly) > area(polybox.shape(neighbors(k)))
                    currentpoly = subtract(currentpoly,polyintsec);
                else
                    polybox.shape(neighbors(k)) =
subtract(polybox.shape(neighbors(k)),polyintsec);
                end
            end
        end

        % check if the current bounding box is valid following removal of neighbor overlaps
        if height(currentpoly.Vertices) == 0
            polybox.shape(polyIdx) = [];   % remove current bounding box from list
            offset = offset+1;          % increment offset counter by 1
            continue;
        else
            polybox.shape(polyIdx) = currentpoly;  % save the cleaned up bounding box to the
original shape list index
            plot(currentpoly)
        end
    end
    hold off;

    delete(cleanWait) % delete waitbar

    % update centroid values of bounding boxes
    [x,y] = centroid(polybox.shape);
    polybox.centroid = [x;y]';

    % get current bounding box connectivity post clean up
    Gbox = polyConn(polybox.shape);

    % visually inspect bounding box connectivity
    figure; show(editedMap); hold on;
    plot(polybox.shape)
    plot(Gbox,'XData',polybox.centroid(:,1),'YData',polybox.centroid(:,2))
    hold off;
```

```
% PREPARE AND IDENTIFY BOUNDING BOXES TO MERGE
```

```
      % identify any slivers formed during the clean up process
      for i = 1:length(polybox.shape)
          currentpoly = polybox.shape(i); % identify current bounding box
          polynosliver = rmslivers(currentpoly,.05);  % remove any slivers from current bounding
box
          polysliver = subtract(currentpoly,polynosliver);    % save the polyshape consisting of
the removed slivers

          % check if the removed slivers polyshape is valid, if so append to end
          % of shape list
          if height(polysliver.Vertices) == 0
              continue;
          else
              polybox.shape(1,end+1) = polysliver;
          end

          % check if the polygon with slivers removed is valid, if so save
          % to the original polygon index
          if height(polynosliver.Vertices) == 0
              continue;
          else
              polybox.shape(i) = polynosliver;
          end
      end

      % ensure that each polygon has one region
      for i = 1:length(polybox.shape)
          % if multiple regions are identifed, append these regions to polygon list
          if polybox.shape(i).NumRegions > 1
              polyRegions = regions(polybox.shape(i));

              for j = 1:length(polyRegions)
                  % check if the polygon region is valid
                  if height(polyRegions(j).Vertices) == 0
                      continue;

                      % save the first region to the original polygon index
                  elseif j == 1
                      polybox.shape(i) = polyRegions(1);

                      % append following regions to end of shape list
```

```
                else
                    polybox.shape(end+1) = polyRegions(j);
                end
            end
        end
    end

    % update centroid values
    [x,y] = centroid(polybox.shape);
    polybox.centroid = [x;y]';

    % get current polygon connectivity
    Gbox = polyConn(polybox.shape);

    % visually inspect polygon connectivity
    figure;
    show(editedMap); hold on;
    plot(polybox.shape)
    plot(Gbox,'XData',polybox.centroid(:,1),'YData',polybox.centroid(:,2))
    hold off;

    % determine an optimal path passing through all polygons using TSP algorithm
    [~, polybox.nodeList] = directedTSP(polybox, room.Vertices);

    % identify the polygons that need to be merged
    box2merge = identifyBox2merge(mergeRob, polybox, editedMap);
```

% MERGE BOUNDING BOXES

```
    % configure waitbar to monitor merging process
    boxWait = waitbar(0,'Initializing Merging Process','Name','Merging Bounding Boxes...',...
        'CreateCancelBtn','setappdata(gcbf,''canceling'',1)');

    % configure a cancel button on waitbar
    setappdata(boxWait,'canceling',0);

    offset = 0; % counter to track the number of nodes removed from the original node list
    for j = 1:length(box2merge)
        % check if cancel button has been pressed
        if getappdata(boxWait,'canceling'); break; end

        % identify node to be merged by subtracting the offset from the current node value
        node = box2merge(j,1)-offset;
        % update waitbar and message
        waitbar(j/length(box2merge),boxWait,sprintf('Merging polygon %d of
%d',j,length(box2merge)))

        % fprintf('%d\n',node+offset) % display current node being merged

        Gbox = polyConn(polybox.shape);    % get current polygon connectivity

        try
            % determine which neighbor to merge with
            neighbor2merge = polyNeighbor(Gbox, node, polybox.shape);

            % merge neighbor with current node
            polybox.shape(neighbor2merge) =
union(polybox.shape(neighbor2merge),polybox.shape(node));

            offset = offset+1;  % increment offset value
            polybox.shape(node) = [];   % remove merged node from shape list
        catch
            fprintf('Node %d has no identified neighbors',node+offset)
            offset = offset+1;  % increment offset value
            polybox.shape(node) = [];   % remove neighborless node from shape list
        end
    end

    delete(boxWait) % delete waitbar
```

% FINALIZE RESULTING POLYGONS

```
    % ensure that each polygon has one region and no unnecessary holes
    for i = 1:length(polybox.shape)
        % if multiple regions are identifed, create a bounding box to merge them together
        if polybox.shape(i).NumRegions > 1 || polybox.shape(i).NumHoles >= 1
            currentpoly = polybox.shape(i); % identify the current polygon

            % obtain bounding box XY limits
            [xlim,ylim] = boundingbox(currentpoly);

            % identify min and max of XY limits
            xmin = min(xlim); xmax = max(xlim);
            ymin = min(ylim); ymax = max(ylim);
            % generate a polyshape of the bounding box and replace the current
            % polygon with the bounding box
            polyMin = [xmin,ymin;xmax,ymin;xmax,ymax;xmin,ymax];
            currentpoly = polyshape(polyMin);

            % identify and remove any areas of the current polygon outside the room
            roomintsec = intersect(currentpoly,roomOutside);
            currentpoly = subtract(currentpoly,roomintsec);

            % remove any overlaps between other polyshapes
            for before = 1:i-1
                currentpoly = subtract(currentpoly,polybox.shape(before));
            end

            for after = i+1:length(polybox.shape)
                currentpoly = subtract(currentpoly,polybox.shape(after));
            end

            % remove any resulting slivers
            currentpoly = rmslivers(currentpoly,.05);

            % save the edited polygon to the original polygon index
            polybox.shape(i) = currentpoly;
        end
    end

    % update centroid values
    [x,y] = centroid(polybox.shape);
    polybox.centroid = [x;y]';

    % get current polygon connectivity
    Gbox = polyConn(polybox.shape);

    % visually inspect polygon connectivity
    figure; show(editedMap); hold on;
    plot(polybox.shape)
    plot(Gbox,'XData',polybox.centroid(:,1),'YData',polybox.centroid(:,2))
    hold off;

    % determine a final optimal path passing through all polygons using TSP algorithm
    [Gdir, polybox.nodeList] = directedTSP(polybox, room.Vertices);

    % visually inspect the generated optimal path
```

```
     figure; show(editedMap); hold on;
     plot(polybox.shape)
     hGraph =
plot(Gdir,'XData',polybox.centroid(:,1),'YData',polybox.centroid(:,2),'LineStyle','none');
     highlight(hGraph,Gdir,'LineStyle','-')
     hold off

     programTime = toc; % record strategy planning algorithm runtime
     % if the number of robots is 1 assign all tasks to one robot
     if numRobots == 1
         deadlock_cntr = 0;  % initialize a variable to count the number of encountered
deadlocks
         % loop through each task for complete coverage
         for i = 1:length(polybox.shape)

             % get local occupancy map and starting location of current task
             [localMap, startXY] = poly2occgrid(polybox, i, rawMap, rob, cornerLoc, resolution);

             % convert starting location to IJ coordinates
             startIJ = world2grid(localMap,startXY);

             % call path planning algorithm to perform complete coverage
             figure;
             [localMap,time(i,1),turnCount(i,1),overlap(i,1),pathLength(i,1),deadlock_cntr] =
cleanRoomSimTest(localMap,startIJ,deadlock_cntr);

             % determine the number of cleaned and uncleaned cells of current task
             localOcc = occupancyMatrix(localMap);
             clean(i,1) = length(find((localOcc>.4)&(localOcc<.6)));
             unclean(i,1) = length(find(localOcc<.1));
         end

         % display simualtion data
         fprintf("Total simulation runtime: %.2f",sum(time))
         fprintf("Total turn count: %d",sum(turnCount))
         fprintf("Total path length: %.2f",sum(pathLength))

         fprintf("Total overlap: %d",sum(overlap))
         fprintf("Minimum task overlap: %d",min(overlap))
         fprintf("Maximum task overlap: %d",max(overlap))
         fprintf("Average overlap per task: %.2f",mean(overlap))

         covEff = sum(clean) / (sum(unclean)+sum(clean));
```

```
        fprintf("Coverage efficiency: %.2f%%",covEff*100)

        fprintf("Number of deadlock events: %d",deadlock_cntr)
        fprintf("Strategy planning algorithm runtime: %.2f",programTime)
        fprintf("Number of triangles within mesh: %d",length(tri.list))
        fprintf("Number of tasks: %d",length(polybox.shape))
```

Task Allocation

```
    elseif numRobots == 2
        task_list = polybox.nodeList(:,1); % assign the node list first column to the task list
        task_list(end,:) = [];  % remove the repeated node at bottom of list

        % divide the task list into segments according to the number of robots
        task_segments = allocateTasks(task_list, numRobots)

        delete(gcp('nocreate'));    % end any background parallel pool processes
        parpool(2); % start a parallel pool process

        % tag naming reference
        % tag 1 -> robot1retask
        % tag 2 -> robot2retask
        % tag 3 -> robot1done
        % tag 4 -> robot2done
        % tag 5 -> robot1onetaskleft
        % tag 6 -> robot2onetaskleft

        % set flag variables
        finished = 0;
        otherFinished = 0;
        reAllocateTasks = 0;
        robot1onetaskleft = 0;
        robot2onetaskleft = 0;

        % initialize simulation data variables
        tot_time = 0;
        tot_turnCount = 0;
        tot_overlap = 0;
        tot_pathLength = 0;
        clean = 0;
        unclean = 0;

        if enableRetasking
            spmd
```

```
                  if spmdIndex == 1
                      try
                          disp('Robot 1: Cleaning in progress')

                          deadlock_cntr{spmdIndex} = 0;   % initialize variable on worker to
count the number of encountered deadlocks
                          % loop through the assigned tasks for complete coverage
                          for currentTask = 1:height(task_segments{spmdIndex})

                              % check if any message received from other robot
                              if spmdProbe
                                  otherFinished = spmdReceive('any',4)   % receive robot2done
with tag 4

                                  % check if the received message indicates the other robot has
finished
                                  if otherFinished == 1
                                      disp('Robot 1: Robot 2 has finished')

                                      % check if the number of remaining tasks is greater than 1
                                      if (height(task_segments{spmdIndex}) - currentTask) > 1
                                          reAllocateTasks = 1;    % set reAllocateTasks flag to 1
                                          robot1onetaskleft = 0;  % set robot1onetaskleft flag to
0

                                          spmdSend(robot1onetaskleft,2,5) % send
robot1onetaskleft to robot 2 with tag 5
                                          spmdBarrier      % barrier #1; update robot 2 there is
more than 1 task remaining for robot 1
                                          break;  % break out of current loop
                                      else
                                          disp("Robot 1: 1 task remaining")
                                          robot1onetaskleft = 1;  % set robot1onetaskleft to 1
                                          spmdSend(robot1onetaskleft,2,5) % send
robot1onetaskleft to robot 2 with tag 5
                                          spmdBarrier     % barrier #1; update robot 2 there is 1
task remaining for robot 1
                                      end
                                  end
                              end

                              % get local occupancy map and starting location of current task
                              [localMap, startXY] = poly2occgrid(polybox,
task_segments{spmdIndex}(currentTask), rawMap, rob, cornerLoc, resolution);
```

```
                              % convert starting location to IJ coordinates
                              startIJ = world2grid(localMap,startXY);

                              % call path planning algorithm to perform complete coverage
                              figure;

[localMap,time,turnCount,overlap,pathLength,deadlock_cntr{spmdIndex}] =
cleanRoomSimTest(localMap,startIJ,deadlock_cntr{spmdIndex});

                              % determine the number of cleaned and uncleaned cells of current
task
                              localOcc = occupancyMatrix(localMap);
                              clean = clean+length(find((localOcc>.4)&(localOcc<.6)));
                              unclean = unclean+length(find(localOcc<.1));

                              % update simulation data
                              tot_time = tot_time +time;
                              tot_turnCount = tot_turnCount+turnCount;
                              tot_overlap = tot_overlap+overlap;
                              tot_pathLength = tot_pathLength+pathLength;

                              % report cleaning progress
                              fprintf('Robot %d: Task %d complete. Progress: %d of %d\n', ...

spmdIndex,task_segments{spmdIndex}(currentTask),currentTask,length(task_segments{spmdIndex}))
                          end

                          % enter this code block if the reAllocateTasks flag is set to 1
                          if reAllocateTasks
                              % report task reallocation is occurring and task stopped on
                              fprintf('Robot %d: Reallocating tasks. Stopped on Task %d\n', ...
                                  spmdIndex,task_segments{spmdIndex}(currentTask))

                              % remove already completed tasks from current task list
                              for i = 1:currentTask-1
                                  task_segments{spmdIndex}(1,:) = [];
                              end

                              % reallocate remaining tasks
                              retasked_segments =
allocateTasks(task_segments{spmdIndex},num_robots)
```

```
                              spmdSend(retasked_segments,2,1)   % send robot1retask to robot 2
with tag 1

                              spmdBarrier     % barrier #2; send robot 1 retask to robot 2 and
wait

                              % loop through the reassigned tasks for complete coverage
                              for currentTask = 1:length(retasked_segments{1})

                                  % get local occupancy map and starting location of current task
                                  [localMap, startXY] = poly2occgrid(polybox,
retasked_segments{1}(currentTask), rawMap, rob, cornerLoc, resolution);

                                  % convert starting location to IJ coordinates
                                  startIJ = world2grid(localMap,startXY);

                                  % call path planning algorithm to perform complete coverage
                                  figure;

[localMap,time,turnCount,overlap,pathLength,deadlock_cntr{spmdIndex}] =
cleanRoomSimTest(localMap,startIJ,deadlock_cntr{spmdIndex});

                                  % determine the number of cleaned and uncleaned cells of
current task
                                  localOcc = occupancyMatrix(localMap);
                                  clean = clean+length(find((localOcc>.4)&(localOcc<.6)));
                                  unclean = unclean+length(find(localOcc<.1));

                                  % update simulation data
                                  tot_time = tot_time +time;
                                  tot_turnCount = tot_turnCount+turnCount;
                                  tot_overlap = tot_overlap+overlap;
                                  tot_pathLength = tot_pathLength+pathLength;

                                  % report cleaning progress
                                  fprintf('Robot %d: Reallocated Task %d complete\nProgress: %d
of %d\n',
...spmdIndex,retasked_segments{1}(currentTask),currentTask,length(retasked_segments{1}))
                              end
                          end

                          finished = 1;   % set finished flag to 1
```

```
                         disp('Robot 1: Done')   % report robot 1 has completed its assigned
tasks

                         % enter this code block if the finished flag is set to 1 and the
otherFinished flag is set to 0
                         if (finished == 1) && (otherFinished == 0)
                             spmdSend(finished,2,3)    % send robot1done to robot 2 with tag 3

                             spmdBarrier     % barrier #3; communicate robot 1 has finished
before robot 2; wait for task remaining update from robot 1

                             robot2onetaskleft = spmdReceive("any",6)    % receive2onetaskleft
with tag 6

                             % check status of one task remaining flag
                             if robot2onetaskleft == 0
                                 disp("Robot 1: Receiving reallocated tasks from Robot 2")

                                 spmdBarrier     % barrier #4; wait for robot 2 retask update

                                 retasked_segments = spmdReceive("any",2)    % receive
robot2retask with tag 2

                                 % loop through the received reassigned tasks for complete
coverage
                                 for currentTask = 1:length(retasked_segments{2})

                                     % get local occupancy map and starting location of current
task
                                     [localMap, startXY] = poly2occgrid(polybox,
retasked_segments{2}(currentTask), rawMap, rob, cornerLoc, resolution);

                                     % convert starting location to IJ coordinates
                                     startIJ = world2grid(localMap,startXY);

                                     % call path planning algorithm to perform complete coverage
                                     figure;

[localMap,time,turnCount,overlap,pathLength,deadlock_cntr{spmdIndex}] =
cleanRoomSimTest(localMap,startIJ,deadlock_cntr{spmdIndex});

                                     % determine the number of cleaned and uncleaned cells of
current task
```

```
                                      localOcc = occupancyMatrix(localMap);
                                      clean = clean+length(find((localOcc>.4)&(localOcc<.6)));
                                      unclean = unclean+length(find(localOcc<.1));

                                      % update simulation data
                                      tot_time = tot_time+time;
                                      tot_turnCount = tot_turnCount+turnCount;
                                      tot_overlap = tot_overlap+overlap;
                                      tot_pathLength = tot_pathLength+pathLength;

                                      % report cleaning progress
                                      fprintf('Robot %d: Reallocated Task %d complete\nProgress:
%d of %d\n',
...spmdIndex,retasked_segments{2}(currentTask),currentTask,length(retasked_segments{2}))
                                  end
                              else
                                  disp("Robot 1: Bypassing retask since 1 task remaining on Robot
2")
                              end
                          end
                          spmdBarrier % barrier #0; wait for both robots to complete assigned
tasks
                      catch
                          disp("Both robots have finished at the same time")
                      end
                  end

                  if spmdIndex == 2
                      try
                          disp('Robot 2: Cleaning in progress')

                          deadlock_cntr{spmdIndex} = 0;   % initialize variable on worker to
count the number of encountered deadlocks
                          % loop through the assigned tasks for complete coverage
                          for currentTask = 1:height(task_segments{spmdIndex})

                              % check if any message received from other robot
                              if spmdProbe
                                  otherFinished = spmdReceive("any",3)   % receive robot1done
with tag 3

                                      % check if the received message indicates the other robot has
finished
```

```
                                  if otherFinished == 1
                                      disp('Robot 2: Robot 1 has finished')

                                      % check if the number of remaining tasks is greater than 1
                                      if (height(task_segments{spmdIndex}) - currentTask) > 1
                                          reAllocateTasks = 1;    % set reAllocateTasks flag to 1
                                          robot2onetaskleft = 0;  % set robot2onetaskleft flag to
0
                                          spmdSend(robot2onetaskleft,1,6) % send
robot2onetaskleft to robot 1 with tag 6
                                          spmdBarrier     % barrier #3; update robot 1 there is
more than 1 task remaining for robot 2
                                          break;
                                      else
                                          disp("Robot 2: 1 task remaining")
                                          robot2onetaskleft = 1;  % set robot1onetaskleft to 1
                                          spmdSend(robot2onetaskleft,1,6) % send
robot2onetaskleft to robot 1 with tag 6
                                          spmdBarrier     % barrier #3; update robot 1 there is 1
task remianing for robot 2
                                      end
                                  end
                              end

                              % get local occupancy map and starting location of current task
                              [localMap, startXY] = poly2occgrid(polybox,
task_segments{spmdIndex}(currentTask), rawMap, rob, cornerLoc, resolution);

                              % convert starting location to IJ coordinates
                              startIJ = world2grid(localMap,startXY);

                              % call path planning algorithm to perform complete coverage
                              figure;

[localMap,time,turnCount,overlap,pathLength,deadlock_cntr{spmdIndex}] =
cleanRoomSimTest(localMap,startIJ,deadlock_cntr{spmdIndex});

                              % determine the number of cleaned and uncleaned cells of current
task
                              localOcc = occupancyMatrix(localMap);
                              clean = clean+length(find((localOcc>.4)&(localOcc<.6)));
                              unclean = unclean+length(find(localOcc<.1));
                              % update simulation data
                              tot_time = tot_time +time;
                              tot_turnCount = tot_turnCount+turnCount;
                              tot_overlap = tot_overlap+overlap;
                              tot_pathLength = tot_pathLength+pathLength;

                              % report cleaning progress
                              fprintf('Robot %d: Task %d complete\nProgress %d of %d\n', ...
 spmdIndex,task_segments{spmdIndex}(currentTask),currentTask,length(task_segments{spmdIndex}))
                          end

                          % enter this code block if the reAllocateTasks flag is set to 1
                          if reAllocateTasks
                              % report task reallocation is occurring and task stopped on
                              fprintf('Robot %d: Reallocating tasks. Stopped on Task %d\n', ...
                                  spmdIndex,task_segments{spmdIndex}(currentTask))

                              % remove already completed tasks from current task list
                              for i = 1:currentTask-1
                                  task_segments{spmdIndex}(1,:) = [];   % remove already
completed tasks
                              end

                              % reallocate remaining tasks
                              retasked_segments =
allocateTasks(task_segments{spmdIndex},num_robots)

                              spmdSend(retasked_segments,1,2)   % send robot2retask to robot 1
with tag 2

                              spmdBarrier     % barrier #4; send robot 2 retask to robot 1 and
wait

                              % loop through the reassigned tasks for complete coverage
                              for currentTask = 1:length(retasked_segments{1})

                                  % get local occupancy map and starting location of current task
                                  [localMap, startXY] = poly2occgrid(polybox,
retasked_segments{1}(currentTask), rawMap, rob, cornerLoc, resolution);

                                  % convert starting location to IJ coordinates
                                  startIJ = world2grid(localMap,startXY);
```

```
                                 % call path planning algorithm to perform complete coverage
                                 figure;

[localMap,time,turnCount,overlap,pathLength,deadlock_cntr{spmdIndex}] =
cleanRoomSimTest(localMap,startIJ,deadlock_cntr{spmdIndex});

                                 % determine the number of cleaned and uncleaned cells of
current task
                                 localOcc = occupancyMatrix(localMap);
                                 clean = clean+length(find((localOcc>.4)&(localOcc<.6)));
                                 unclean = unclean+length(find(localOcc<.1));

                                 % update simulation data
                                 tot_time = tot_time +time;
                                 tot_turnCount = tot_turnCount+turnCount;
                                 tot_overlap = tot_overlap+overlap;
                                 tot_pathLength = tot_pathLength+pathLength;

                                 % report cleaning progress
                                 fprintf('Robot %d: Reallocated Task %d complete\nProgress %d of
%d\n', ...spmdIndex,retasked_segments{1}(currentTask),currentTask,length(retasked_segments{1}))
                             end
                         end

                         finished = 1;   % set finished flag to 1
                         disp('Robot 2 done')    % report robot 2 has completed its assigned
tasks

                         % enter this code block if the finished flag is set to 1 and the
otherFinished flag is set to 0
                         if (finished == 1) && (otherFinished == 0)
                             spmdSend(finished,1,4)    % send robot2done to robot 1 with tag 4

                             spmdBarrier     % barrier #1; communicate robot 2 has finished
before robot 1; wait for task remaining update from robot 1

                             robot1onetaskleft = spmdReceive("any",5)    % receive
robot1onetaskleft with tag 5

                             % check status of one task remaining flag
                             if robot1onetaskleft == 0
                                 disp("Robot 2: Receiving reallocated tasks from Robot 1")
```

```
                                  spmdBarrier     % barrier #2; wait for robot 1 retask update

                                  retasked_segments = spmdReceive("any",1)    % receive
robot1retask with tag 1

                                  % loop through the received reassigned tasks for complete
coverage
                                  for currentTask = 1:length(retasked_segments{2})

                                      % get local occupancy map and starting location of current
task
                                      [localMap, startXY] = poly2occgrid(polybox,
retasked_segments{2}(currentTask), rawMap, rob, cornerLoc, resolution);

                                      % convert starting location to IJ coordinates
                                      startIJ = world2grid(localMap,startXY);

                                      % call path planning algorithm to perform complete coverage
                                      figure;

[localMap,time,turnCount,overlap,pathLength,deadlock_cntr{spmdIndex}] =
cleanRoomSimTest(localMap,startIJ,deadlock_cntr{spmdIndex});

                                      % determine the number of cleaned and uncleaned cells of
current task
                                      localOcc = occupancyMatrix(localMap);
                                      clean = clean+length(find((localOcc>.4)&(localOcc<.6)));
                                      unclean = unclean+length(find(localOcc<.1));

                                      % update simulation data
                                      tot_time = tot_time+time;
                                      tot_turnCount = tot_turnCount+turnCount;
                                      tot_overlap = tot_overlap+overlap;
                                      tot_pathLength = tot_pathLength+pathLength;

                                      % report cleaning progress
                                      fprintf('Robot %d: Reallocated Task %d complete\nProgress
%d of %d\n',
...spmdIndex,retasked_segments{2}(currentTask),currentTask,length(retasked_segments{2}))
                                  end
                              else
                                  disp("Robot 2: Bypassing retask since 1 task remaining on Robot
1")
```

```
                            end
                        end
                        spmdBarrier % barrier #0; wait for both robots to complete assigned 
tasks
                    catch
                        disp("Both robots have finished at the same time")
                    end
                end
            end
        else
            spmd
                if spmdIndex == 1
                    disp('Robot 1: Cleaning in progress')

                    deadlock_cntr{spmdIndex} = 0;   % initialize variable on worker to count 
the number of encountered deadlocks
                    % loop through the assigned tasks for complete coverage
                    for currentTask = 1:height(task_segments{spmdIndex})

                        % get local occupancy map and starting location of current task
                        [localMap, startXY] = poly2occgrid(polybox, 
task_segments{spmdIndex}(currentTask), rawMap, rob, cornerLoc, resolution);

                        % convert starting location to IJ coordinates
                        startIJ = world2grid(localMap,startXY);

                        % call path planning algorithm to perform complete coverage
                        figure;
                        [localMap,time,turnCount,overlap,pathLength,deadlock_cntr{spmdIndex}] = 
cleanRoomSimTest(localMap,startIJ,deadlock_cntr{spmdIndex});

                        % determine the number of cleaned and uncleaned cells of current task
                        localOcc = occupancyMatrix(localMap);
                        clean = clean+length(find((localOcc>.4)&(localOcc<.6)));
                        unclean = unclean+length(find(localOcc<.1));

                        % update simulation data
                        tot_time = tot_time +time;
                        tot_turnCount = tot_turnCount+turnCount;
                        tot_overlap = tot_overlap+overlap;
                        tot_pathLength = tot_pathLength+pathLength;

                        % report cleaning progress
```

```
                         fprintf('Robot %d: Task %d complete. Progress: %d of %d\n', ...

spmdIndex,task_segments{spmdIndex}(currentTask),currentTask,length(task_segments{spmdIndex}))
                     end

                     disp('Robot 1: Done')   % report robot 1 has completed its assigned tasks
                     spmdBarrier % barrier #0; wait for both robots to complete assigned tasks
                 end

                 if spmdIndex == 2
                     disp('Robot 2: Cleaning in progress')

                     deadlock_cntr{spmdIndex} = 0;   % initialize variable on worker to count
the number of encountered deadlocks
                     % loop through the assigned tasks for complete coverage
                     for currentTask = 1:height(task_segments{spmdIndex})

                         % get local occupancy map and starting location of current task
                         [localMap, startXY] = poly2occgrid(polybox,
task_segments{spmdIndex}(currentTask), rawMap, rob, cornerLoc, resolution);

                         % convert starting location to IJ coordinates
                         startIJ = world2grid(localMap,startXY);

                         % call path planning algorithm to perform complete coverage
                         figure;
                         [localMap,time,turnCount,overlap,pathLength,deadlock_cntr{spmdIndex}] =
cleanRoomSimTest(localMap,startIJ,deadlock_cntr{spmdIndex});

                         % determine the number of cleaned and uncleaned cells of current task
                         localOcc = occupancyMatrix(localMap);
                         clean = clean+length(find((localOcc>.4)&(localOcc<.6)));
                         unclean = unclean+length(find(localOcc<.1));

                         % update simulation data
                         tot_time = tot_time +time;
                         tot_turnCount = tot_turnCount+turnCount;
                         tot_overlap = tot_overlap+overlap;
                         tot_pathLength = tot_pathLength+pathLength;

                         % report cleaning progress
                         fprintf('Robot %d: Task %d complete\nProgress %d of %d\n', ...
   spmdIndex,task_segments{spmdIndex}(currentTask),currentTask,length(task_segments{spmdIndex}))
```

```
                    end

                    disp('Robot 2 done')    % report robot 2 has completed its assigned tasks
                    spmdBarrier % barrier #0; wait for both robots to complete assigned tasks
                end
            end
        end

        % display simualtion data
        runtime = [tot_time{:}];
        fprintf("Robot 1 simulation runtime: %.2fs",runtime(1))
        fprintf("Robot 2 simulation runtime: %.2fs",runtime(2))
        fprintf("Idle time: %.2fs",max(runtime)-min(runtime))

        retask = [reAllocateTasks{:}];
        if any(retask); fprintf("Tasks reallocated"); else; fprintf("Tasks not reallocated");
end
        fprintf("Robot 1 turn count: %d",tot_turnCount{1})
        fprintf("Robot 2 turn count: %d",tot_turnCount{2})

        fprintf("Robot 1 path length: %.2f",tot_pathLength{1})
        fprintf("Robot 2 path length: %.2f",tot_pathLength{2})

        fprintf("Robot 1 overlap: %d",tot_overlap{1})
        fprintf("Robot 2 overlap: %d",tot_overlap{2})

        covEff1 = clean{1} / (clean{1}+unclean{1});
        fprintf("Robot 1 coverage efficiency: %.2f%%",covEff1*100)
        covEff2 = clean{2} / (clean{2}+unclean{2});
        fprintf("Robot 2 coverage efficiency: %.2f%%",covEff2*100)
        covEff = (clean{1}+clean{2}) / (clean{1}+clean{2}+unclean{1}+unclean{2});
        fprintf("Total coverage efficiency: %.2f%%",covEff*100)

        deadlock = [deadlock_cntr{:}]; deadlock = cell2mat(deadlock);
        fprintf("Robot 1 deadlock events: %d",deadlock(1))
        fprintf("Robot 2 deadlock events: %d",deadlock(2))

        fprintf("Strategy planning algorithm runtime: %.2fs",programTime)
        fprintf("Number of triangles within mesh: %d",length(tri.list))
        fprintf("Number of tasks: %d",length(task_list))
    else
        disp("Current program can only account for 1 or 2 robots")
    end
```

```
end
```

II. roomStrategyReal

Setup

```
clearvars; close all; % clear all workspace variables and close all figures
```

% INPUT PARAMETERS

```
rob.size = 12;  % distance from robot center to edge midpoint in IJ coordinates
rob.diag = 14;  % distance from robot center to corner in IJ coordinates

min_triEdge = 2; % target minimum edge length for triangulation mesh

cornerLoc = 1;  % desired starting location of path planning algorithm.
Valid values for this parameter are:
% 1 (farthest valid starting corner from polyshape centroid)
% 2 (closest valid starting corner from polyshape centroid)
% 3 (polyshape centroid)

editedPGM_path = "X";  % file path of post-edited lidar map in PGM format

numRobots = 1; % number of robots to be utilized in decomposed work area
```

% ADDITIONAL PARAMETERS

```
mergeTri = true; % false skip triangle merging, true perform triangle merging (default)
if mergeTri; minTri.base = .5; minTri.height = .5; end % minumum values for base and height
when determining triangles to merge

rosSetup = true; % false skip connecting to ROS network, true connect to ROS network (default)

% due to potential differences between the raw occupancy map origin and
% the default occupancy map origin in MATLAB (0,0), this difference can be
% corrected by importing the raw occupancy map
importRawMap = false; % false skip importing raw occupancy map (default), true import raw
occupancy map

wholeRoom = false; % false perform map decomposition (default), true do not perform map
decomposition

enableRetasking = true; % false do not reallocate tasks, true reallocate tasks (default)
```

```
resolution = 50;   % occupancy map resolution in cells per meter

mergeRob.size = rob.size+3; % padded robot size to consider when identifying polygons to merge
mergeRob.diag = rob.diag+3; % padded robot diagonal length to consider when identifying
polygons to merge
```

% CONNECT TO ROS NETWORK

```
if rosSetup
    rosshutdown()   % make sure any ROS sessions are closed
    rosinit("http://X.X.X.X:11311") % initialize ROS network and connect to host
end
```

% IMPORT RAW MAP

```
if importRawMap
    try
        % try to subscribe to /map topic and receive the sent message
        sub = rossubscriber("/map",DataFormat='struct');
        msg = receive(sub);

        % generate an occupancy map with the received map message
        rawMap = rosReadOccupancyGrid(msg);
        show(rawMap)    % visually inspect the imported raw occupancy map
    catch
        % if unable to subscribe to /map topic, load raw map from local save
        savedMap = load('ros_occMap.mat');
        rawMap = savedMap.occupancyMapObj;
        show(rawMap); clear savedMap;
    end
else
    rawMap = [];
end
```

Image Processing

```
tic
editedPGM = imread(editedPGM_path); % load in pgm image of post edited lidar scan

% normalize the image to values between 0 and 1 then convert to occupancy
% values by subtracting from 1
editedMap_occ = 1 - double(editedPGM)/255;

% generate an occupancy map from the occupancy values
```

```
editedMap = occupancyMap(editedMap_occ,resolution);
show(editedMap) % visually inspect the generated occupancy map
```

% IDENTIFY EXTERIOR BOUNDARY AND INTERIOR OBSTACLE BOUNDARIES

```
[exteriorIJ, interiorIJ] = identifyBoundaries(editedPGM);

% determine if the exterior boundary has been misidentified as an interior obstacle,
% if so set the interior obstacle as the exterior boundary
obsNum = fieldnames(interiorIJ);    % determine number of identified interior obstacles
if (length(exteriorIJ) < 10) && (numel(obsNum) == 1)
    % reduce the number of points describing the exterior boundary
    exteriorIJ = reducepoly(interiorIJ.obs1,.01);
    removeObs = 0;  % do not remove interior obstacles
else
    removeObs = 1;  % do remove interior obstacles
end

% convert the exterior boundary points from IJ to XY coordinates
exteriorXY = grid2world(editedMap,exteriorIJ);
```

% POLYSHAPE REPRESENTATION

```
% generate a polyshape of the room from the exterior boundary points
room = polyshape(exteriorXY(:,1),exteriorXY(:,2));
plot(room)

% if applicable, remove the identified interior obstacles from the room polyshape
if removeObs == 1
    for i = 1:numel(obsNum)
        % convert the identifed interior obstacles points from IJ to XY coordinates
        interiorXY = grid2world(editedMap,interiorIJ.(obsNum{i}));

        % reduce the number of points describing the obstacle
        reducedObs = reducepoly(interiorXY,.01);

        % generate a polyshape representing the obstacle
        obs = polyshape(reducedObs(:,1),reducedObs(:,2));

        % remove the obstacle polyshape from the room polyshape
        room = subtract(room,obs);
    end
    plot(room); % visually inspect the resulting polyshape
```

```
end
```

Map Decomposition

```
if wholeRoom
    programTime = toc;

    % set up ROS publishers
    localMap_pub = rospublisher("/localMap","nav_msgs/OccupancyGrid",DataFormat='struct');
    goal_pub = rospublisher("/goal","geometry_msgs/PoseStamped","DataFormat","struct");

    % set up ROS publisher messages
    localMap_msgType = rosmessage("nav_msgs/OccupancyGrid","DataFormat","struct");
    goal = rosmessage("geometry_msgs/PoseStamped","DataFormat","struct");

    % set up ROS subscribers
    localMap_sub = rossubscriber("/map/local_coverage_map",DataFormat='struct');
    taskFlag_sub = rossubscriber("/taskFlag",'std_msgs/Int32');
    taskFlag = 0;   % initialize task flag to 0

    % get occupancy map and starting location of room
    [localMap, startXY] = poly2occgrid(room, 0, rawMap, rob, cornerLoc, resolution);

    % format ROS messages
    localMap_msg = rosWriteOccupancyGrid(localMap_msgType,localMap);
    localMap_msg.Header.FrameId = 'map';

    goal.Header.FrameId = 'map';
    goal.Pose.Position.X = startXY(1,1);
    goal.Pose.Position.Y = startXY(1,2);

    heading  = eul2quat([0,0,pi],'XYZ');
    goal.Pose.Orientation.W = heading(1);
    goal.Pose.Orientation.X = heading(2);
    goal.Pose.Orientation.Y = heading(3);
    goal.Pose.Orientation.Z = heading(4);

    % send ROS messages
    send(localMap_pub,localMap_msg)
    send(goal_pub,goal)

    % wait for flag to signal task complete
    while taskFlag ~= 1
        taskFlag_msg = receive(taskFlag_sub);
        taskFlag = taskFlag_msg.Data;
    end

    % receive cleaned map message
    localMap_msg = receive(localMap_sub);
    cleanedMap = rosReadOccupancyGrid(localMap_msg);
    show(cleanedMap)

    % determine the number of cleaned and uncleaned cells
    localOcc = occupancyMatrix(cleanedMap);
    clean = length(find((localOcc>.4)&(localOcc<.6)));
    unclean = length(find(localOcc<.1));

    % display data
    covEff = clean / (unclean+clean);
    fprintf("Coverage efficiency: %.2f%%",covEff*100)

    fprintf("Strategy planning algorithm runtime: %.2fs",programTime)
else
    % decompose polyshape using triangular mesh
    tri = triDecomposition(room, min_triEdge);

    % determine an initial optimal path passing through all triangle centroids using
    % Traveling Salesman Problem (TSP) algorithm
    [Gdir, tri.nodeList] = directedTSP(tri, room.Vertices);

    % overlay triangular mesh and identified optimal path on the edited map
    show(editedMap); hold on;
    plot(tri.shape)
    hGraph =
plot(Gdir,'XData',tri.centroid(:,1),'YData',tri.centroid(:,2),'LineStyle','none','NodeLabel',{})
;
    highlight(hGraph,Gdir,'LineStyle','-')
    hold off
    if mergeTri
        % determine if a triangle needs to be merged
        tri2merge = identifyTri2merge(tri, minTri)
```

```
% MERGE IDENTIFIED TRIANGLES
```

```
        % configure waitbar to monitor merging process
        triWait = waitbar(0,'Initializing Merging Process','Name','Merging Triangles...',...
            'CreateCancelBtn','setappdata(gcbf,''canceling'',1)');
        % configure a cancel button on waitbar
        setappdata(triWait,'canceling',0);

        offset = 0; % counter to track the number of nodes removed from the original node list
        for i = 1:length(tri2merge)
            % check if cancel button has been pressed
            if getappdata(triWait,'canceling'); break; end

            % identify node to be merged by subtracting the offset from the current node value
            node2merge = tri2merge(i,1)-offset;

            % update waitbar and message
            waitbar(i/length(tri2merge),triWait,sprintf('Merging triangle %d of
%d',i,length(tri2merge)))

            % fprintf('%d\n',node2merge+offset) % display current node being merged

            % get current polygon connectivity
            Gtri = polyConn(tri.shape);

            % determine which neighboring node to merge with
            neighbor2merge = polyNeighbor(Gtri, node2merge, tri.shape);

            % merge neighbor node with current triangle node
            neighborTri = tri.shape(neighbor2merge);
            currentTri = tri.shape(node2merge);
            tri.shape(neighbor2merge) = union(neighborTri,currentTri);    % save resulting
merged polygon to neighbor node index in shape list

            offset = offset+1;  % increment offset value
            tri.shape(node2merge) = [];   % remove merged triangle node from shape list
        end

        delete(triWait) % delete waitbar

        % get current polygon connectivity post merge
        Gtri = polyConn(tri.shape);

        % update centroid values of new polyshapes
        [x,y] = centroid(tri.shape);
        tri.centroid = [x;y]';
        % visually inspect polygon connectivity
        show(editedMap); hold on;
        plot(tri.shape)
        plot(Gtri,'XData',tri.centroid(:,1),'YData',tri.centroid(:,2))
        hold off;

        % determine a final optimal path passing through all triangles using TSP algorithm
        [Gdir, tri.nodeList] = directedTSP(tri, room.Vertices);

        % visually inspect resulting TSP path
        show(editedMap); hold on;
        plot(tri.shape)
        hGraph =
plot(Gdir,'XData',tri.centroid(:,1),'YData',tri.centroid(:,2),'LineStyle','none');
        highlight(hGraph,Gdir,'LineStyle','-')
        hold off
    end
```

```
% GENERATE BOUNDING BOXES
```

```
    tri.nodeList(end,:) = []; % remove repeated node at end of node list

    % convert merged triangle polygons into bounding boxes
    for i = 1:length(tri.nodeList)

        % obtain bounding box XY limits
        [xlim,ylim] = boundingbox(tri.shape(tri.nodeList(i,1)));

        % identify min and max of XY limits
        xmin = min(xlim); xmax = max(xlim);
        ymin = min(ylim); ymax = max(ylim);

        % generate a polyshape of the bounding box and add to shape list
        polyMin = [xmin,ymin;xmax,ymin;xmax,ymax;xmin,ymax];
        polybox.shape(i) = polyshape(polyMin);
    end

    % determine centroid values of bounding boxes
    [x,y] = centroid(polybox.shape);
    polybox.centroid = [x;y]';

    % create a polyshape representing the area outside the room
    [xlim,ylim] = boundingbox(room);
    xmin = min(xlim); xmax = max(xlim);
    ymin = min(ylim); ymax = max(ylim);
    polyMin = polyshape([xmin,ymin;xmax,ymin;xmax,ymax;xmin,ymax]);
    roomOutside = subtract(polyMin,room);
    % plot(roomOutside)
```

```
% CLEAN UP BOUNDING BOXES
```

```
    % configure waitbar to monitor clean up process
    cleanWait = waitbar(0,'Initializing Clean Up Process','Name','Cleaning Up Bounding
Boxes...',...
        'CreateCancelBtn','setappdata(gcbf,''canceling'',1)');

    % configure a cancel button on waitbar
    setappdata(cleanWait,'canceling',0);

    % clean up bounding boxes by removing overlaps and areas outside the room
    figure; show(editedMap); hold on;
    offset = 0;     % counter to track the number of bounding boxes removed from the original
shape list
    for i = 1:length(polybox.shape)-offset
        % check if cancel button has been pressed
        if getappdata(cleanWait,'canceling'); break; end

        % identify bounding box to be cleaned up by subtracting the offset from the current
index value
        polyIdx = i - offset;

        % update waitbar and message
        waitbar(i/length(polybox.shape)-offset,cleanWait,sprintf('Cleaning up bounding box %d
of %d',i,length(polybox.shape)-offset))

        % fprintf('%d\n',polyIdx+offset)  % display current bounding box index being cleaned

        currentpoly = polybox.shape(polyIdx); % identify the current bounding box

        % determine the region of intersection (ROI) between the current bounding box
        % and the area outside the room then remove it from the current bounding box
        roomintsec = intersect(currentpoly,roomOutside);
        currentpoly = subtract(currentpoly,roomintsec);

        Gbox = polyConn(polybox.shape); % get current polygon connectivity
        % determine current bounding box neighbors
        [row,col] = find(Gbox.Edges.EndNodes==polyIdx);
        index = [row col]; neighbors = [];

        for j = 1:height(index)
            if index(j,2) == 1
                neighbors(j,1) = Gbox.Edges.EndNodes(index(j,1),2);
            elseif index(j,2) == 2
                neighbors(j,1) = Gbox.Edges.EndNodes(index(j,1),1);
            end
        end

        % remove current bounding box neighbor overlaps
        for k = 1:height(neighbors)
            % determine the ROI between current bounding box and neighbor
            polyintsec = intersect(currentpoly,polybox.shape(neighbors(k)));

            % check if the ROI is valid
            if height(polyintsec.Vertices) == 0
                continue;
            else
                % remove the ROI from the current bounding box if its area is
                % greater than the neighbor, else remove the ROI from the neighbor
                if area(currentpoly) > area(polybox.shape(neighbors(k)))
                    currentpoly = subtract(currentpoly,polyintsec);
                else
                    polybox.shape(neighbors(k)) =
subtract(polybox.shape(neighbors(k)),polyintsec);
                end
            end
        end

        % check if the current bounding box is valid following removal of neighbor overlaps
        if height(currentpoly.Vertices) == 0
            polybox.shape(polyIdx) = [];   % remove current bounding box from list
            offset = offset+1;          % increment offset counter by 1
            continue;
        else
            polybox.shape(polyIdx) = currentpoly;  % save the cleaned up bounding box to the
original shape list index
            plot(currentpoly)
        end
    end
    hold off;

    delete(cleanWait) % delete waitbar

    % update centroid values of bounding boxes
    [x,y] = centroid(polybox.shape);
    polybox.centroid = [x;y]';

    % get current bounding box connectivity post clean up
    Gbox = polyConn(polybox.shape);

    % visually inspect bounding box connectivity
    figure; show(editedMap); hold on;
    plot(polybox.shape)
    plot(Gbox,'XData',polybox.centroid(:,1),'YData',polybox.centroid(:,2))
    hold off;
```

% PREPARE AND IDENTIFY BOUNDING BOXES TO MERGE

```
    % identify any slivers formed during the clean up process
    for i = 1:length(polybox.shape)
        currentpoly = polybox.shape(i); % identify current bounding box
        polynosliver = rmslivers(currentpoly,.05);  % remove any slivers from current bounding
box
        polysliver = subtract(currentpoly,polynosliver);    % save the polyshape consisting of
the removed slivers

        % check if the removed slivers polyshape is valid, if so append to end
        % of shape list
        if height(polysliver.Vertices) == 0
            continue;
        else
            polybox.shape(1,end+1) = polysliver;
        end

        % check if the polygon with slivers removed is valid, if so save
        % to the original polygon index
        if height(polynosliver.Vertices) == 0
            continue;
        else
            polybox.shape(i) = polynosliver;
        end
    end
```

```
% ensure that each polygon has one region
for i = 1:length(polybox.shape)
    % if multiple regions are identifed, append these regions to polygon list
    if polybox.shape(i).NumRegions > 1
        polyRegions = regions(polybox.shape(i));

        for j = 1:length(polyRegions)
            % check if the polygon region is valid
            if height(polyRegions(j).Vertices) == 0
                continue;

                % save the first region to the original polygon index
            elseif j == 1
                polybox.shape(i) = polyRegions(1);

                % append following regions to end of shape list
            else
                polybox.shape(end+1) = polyRegions(j);
            end
        end
    end
end

% update centroid values
[x,y] = centroid(polybox.shape);
polybox.centroid = [x;y]';

% get current polygon connectivity
Gbox = polyConn(polybox.shape);

% visually inspect polygon connectivity
figure;
show(editedMap); hold on;
plot(polybox.shape)
plot(Gbox,'XData',polybox.centroid(:,1),'YData',polybox.centroid(:,2))
hold off;

% determine an optimal path passing through all polygons using TSP algorithm
[~, polybox.nodeList] = directedTSP(polybox, room.Vertices);

% identify the polygons that need to be merged
box2merge = identifyBox2merge(mergeRob, polybox, editedMap);
```

% MERGE BOUNDING BOXES

```
    % configure waitbar to monitor merging process
    boxWait = waitbar(0,'Initializing Merging Process','Name','Merging Bounding Boxes...',...
        'CreateCancelBtn','setappdata(gcbf,''canceling'',1)');

    % configure a cancel button on waitbar
    setappdata(boxWait,'canceling',0);

    offset = 0; % counter to track the number of nodes removed from the original node list
    for j = 1:length(box2merge)
        % check if cancel button has been pressed
        if getappdata(boxWait,'canceling'); break; end

        % identify node to be merged by subtracting the offset from the current node value
        node = box2merge(j,1)-offset;

        % update waitbar and message
        waitbar(j/length(box2merge),boxWait,sprintf('Merging polygon %d of
%d',j,length(box2merge)))

        % fprintf('%d\n',node+offset) % display current node being merged

        Gbox = polyConn(polybox.shape);    % get current polygon connectivity

        try
            % determine which neighbor to merge with
            neighbor2merge = polyNeighbor(Gbox, node, polybox.shape);

            % merge neighbor with current node
            polybox.shape(neighbor2merge) =
union(polybox.shape(neighbor2merge),polybox.shape(node));

            offset = offset+1;  % increment offset value
            polybox.shape(node) = [];   % remove merged node from shape list
        catch
            fprintf('Node %d has no identified neighbors',node+offset)
            offset = offset+1;  % increment offset value
            polybox.shape(node) = [];   % remove neighborless node from shape list
        end
    end

    delete(boxWait) % delete waitbar
```

```
% FINALIZE RESULTING POLYGONS
```

```
    % ensure that each polygon has one region and no unnecessary holes
    for i = 1:length(polybox.shape)
        % if multiple regions are identifed, create a bounding box to merge them together
        if polybox.shape(i).NumRegions > 1 || polybox.shape(i).NumHoles >= 1
            currentpoly = polybox.shape(i); % identify the current polygon

            % obtain bounding box XY limits
            [xlim,ylim] = boundingbox(currentpoly);

            % identify min and max of XY limits
            xmin = min(xlim); xmax = max(xlim);
            ymin = min(ylim); ymax = max(ylim);

            % generate a polyshape of the bounding box and replace the current
            % polygon with the bounding box
            polyMin = [xmin,ymin;xmax,ymin;xmax,ymax;xmin,ymax];
            currentpoly = polyshape(polyMin);

            % identify and remove any areas of the current polygon outside the room
            roomintsec = intersect(currentpoly,roomOutside);
            currentpoly = subtract(currentpoly,roomintsec);

            % remove any overlaps between other polyshapes
            for before = 1:i-1
                currentpoly = subtract(currentpoly,polybox.shape(before));
            end

            for after = i+1:length(polybox.shape)
                currentpoly = subtract(currentpoly,polybox.shape(after));
            end

            % remove any resulting slivers
            currentpoly = rmslivers(currentpoly,.05);

            % save the edited polygon to the original polygon index
            polybox.shape(i) = currentpoly;
        end
    end

    % update centroid values
    [x,y] = centroid(polybox.shape);
```

```
     polybox.centroid = [x;y]';

     % get current polygon connectivity
     Gbox = polyConn(polybox.shape);

     % visually inspect polygon connectivity
     figure; show(editedMap); hold on;
     plot(polybox.shape)
     plot(Gbox,'XData',polybox.centroid(:,1),'YData',polybox.centroid(:,2))
     hold off;

     % determine a final optimal path passing through all polygons using TSP algorithm
     [Gdir, polybox.nodeList] = directedTSP(polybox, room.Vertices);

     % visually inspect the generated optimal path
     figure; show(editedMap); hold on;
     plot(polybox.shape)
     hGraph =
plot(Gdir,'XData',polybox.centroid(:,1),'YData',polybox.centroid(:,2),'LineStyle','none');
     highlight(hGraph,Gdir,'LineStyle','-')
     hold off

     programTime = toc; % record strategy planning algorithm runtime
     % if the number of robots is 1 assign all tasks to one robot
     if numRobots == 1

         % set up ROS publishers
         localMap_pub = rospublisher("/localMap","nav_msgs/OccupancyGrid",DataFormat='struct');
         goal_pub = rospublisher("/goal","geometry_msgs/PoseStamped","DataFormat","struct");
         stopFlag_pub = rospublisher("/stop",'std_msgs/Int32');

         % set up ROS publisher messages
         localMap_msgType = rosmessage("nav_msgs/OccupancyGrid","DataFormat","struct");
         goal = rosmessage("geometry_msgs/PoseStamped","DataFormat","struct");
         stopFlag = rosmessage(stopFlag_pub);
         stopFlag.Data = 0;      % initialize stop flag to 0
         send(stopFlag_pub,stopFlag) % send stop flag status

         % set up ROS subscribers
         localMap_sub = rossubscriber("/map/local_coverage_map",DataFormat='struct');
         taskFlag_sub = rossubscriber("/taskFlag",'std_msgs/Int32');
         taskFlag = 0;   % initialize task flag to 0
```

```
    % loop through each task for complete coverage
    for i = 1:length(polybox.shape)

        % get local occupancy map and starting location of current task
        [localMap, startXY] = poly2occgrid(polybox, i, rawMap, rob, cornerLoc, resolution);

        % format ROS messages
        localMap_msg = rosWriteOccupancyGrid(localMap_msgType,localMap);
        localMap_msg.Header.FrameId = 'map';

        goal.Header.FrameId = 'map';
        goal.Pose.Position.X = startXY(1,1);
        goal.Pose.Position.Y = startXY(1,2);

        heading  = eul2quat([0,0,pi],'XYZ');
        goal.Pose.Orientation.W = heading(1);
        goal.Pose.Orientation.X = heading(2);
        goal.Pose.Orientation.Y = heading(3);
        goal.Pose.Orientation.Z = heading(4);

        % send ROS messages
        send(localMap_pub,localMap_msg)
        send(goal_pub,goal)

        % wait for flag to signal task complete
        while taskFlag ~= 1
            taskFlag_msg = receive(taskFlag_sub);
            taskFlag = taskFlag_msg.Data;
        end

        % receive cleaned map message
        localMap_msg = receive(localMap_sub);
        cleanedMap = rosReadOccupancyGrid(localMap_msg);
        show(cleanedMap)

        % determine the number of cleaned and uncleaned cells of current task
        localOcc = occupancyMatrix(cleanedMap);
        clean(i,1) = length(find((localOcc>.4)&(localOcc<.6)));
        unclean(i,1) = length(find(localOcc<.1));

        taskFlag = 0; % reset task flag to 0
    end
```

```
        stopFlag.Data = 1;  % set stop flag to 1 to signal all tasks complete
        send(stopFlag_pub,stopFlag) % send stop flag status

        % display data
        covEff = sum(clean) / (sum(unclean)+sum(clean));
        fprintf("Coverage efficiency: %.2f%%",covEff*100)

        fprintf("Strategy planning algorithm runtime: %.2fs",programTime)
```

Task Allocation

```
    elseif numRobots == 2
        task_list = polybox.nodeList(:,1); % assign the node list first column to the task list
        task_list(end,:) = [];  % remove the repeated node at bottom of list

        % divide the task list into segments according to the number of robots
        task_segments = allocateTasks(task_list, numRobots)

        delete(gcp('nocreate'));    % end any background parallel pool processes
        parpool(2); % start a parallel pool process

        % tag naming reference
        % tag 1 -> robot1retask
        % tag 2 -> robot2retask
        % tag 3 -> robot1done
        % tag 4 -> robot2done
        % tag 5 -> robot1onetaskleft
        % tag 6 -> robot2onetaskleft

        % set flag variables
        finished = 0;
        otherFinished = 0;
        reAllocateTasks = 0;
        robot1onetaskleft = 0;
        robot2onetaskleft = 0;

        % initialize simulation data variables
        tot_time = 0;
        tot_turnCount = 0;
        tot_overlap = 0;
        tot_pathLength = 0;
        clean = 0;
        unclean = 0;
        if enableRetasking
            spmd
                if spmdIndex == 1
                    try
                        disp('Robot 1: Cleaning in progress')

                        % set ROS node constant
                        nodeConstant{spmdIndex} = 
parallel.pool.Constant(ros.Node('/Robot1',"http://X.X.X.X:11311"));

                        % set ROS publisher constants
                        map_pubConstant{spmdIndex} = 
parallel.pool.Constant(ros.Publisher(nodeConstant{spmdIndex}.Value,'/localMapRobot1',"nav_msgs/O
ccupancyGrid","DataFormat","struct"));
                        goal_pubConstant{spmdIndex} = 
parallel.pool.Constant(ros.Publisher(nodeConstant{spmdIndex}.Value,'/goalRobot1',"geometry_msgs/
PoseStamped","DataFormat","struct"));

                        % set ROS message types
                        map_msgType{spmdIndex} = 
rosmessage("nav_msgs/OccupancyGrid","DataFormat","struct");
                        goal{spmdIndex} = 
rosmessage("geometry_msgs/PoseStamped","DataFormat","struct");

                        % loop through the assigned tasks for complete coverage
                        for currentTask = 1:height(task_segments{spmdIndex})

                            % check if any message received from other robot
                            if spmdProbe
                                otherFinished = spmdReceive('any',4)   % receive robot2done 
with tag 4

                                % check if the received message indicates the other robot has 
finished
                                if otherFinished == 1
                                    disp('Robot 1: Robot 2 has finished')

                                    % check if the number of remaining tasks is greater than 1
                                    if (height(task_segments{spmdIndex}) - currentTask) > 1
                                        reAllocateTasks = 1;    % set reAllocateTasks flag to 1
                                        robot1onetaskleft = 0;  % set robot1onetaskleft flag to 
0
```

```
                                         spmdSend(robot1onetaskleft,2,5) % send
robot1onetaskleft to robot 2 with tag 5
                                         spmdBarrier      % barrier #1; update robot 2 there is
more than 1 task remaining for robot 1
                                         break;  % break out of current loop
                                     else
                                         disp("Robot 1: 1 task remaining")
                                         robot1onetaskleft = 1;  % set robot1onetaskleft to 1
                                         spmdSend(robot1onetaskleft,2,5) % send
robot1onetaskleft to robot 2 with tag 5
                                         spmdBarrier     % barrier #1; update robot 2 there is 1
task remaining for robot 1
                                     end
                                 end
                             end

                             % get local occupancy map and starting location of current task
                             [localMap, startXY] = poly2occgrid(polybox,
task_segments{spmdIndex}(currentTask), rawMap, rob, cornerLoc, resolution);

                             % set ROS messages
                             map_msg{spmdIndex} =
rosWriteOccupancyGrid(map_msgType{spmdIndex},localMap);

                             goal{spmdIndex}.Header.FrameId = 'map';
                             goal{spmdIndex}.Pose.Position.X = startXY(1,1);
                             goal{spmdIndex}.Pose.Position.Y = startXY(1,2);

                             heading  = eul2quat([0,0,pi],'XYZ');
                             goal{spmdIndex}.Pose.Orientation.W = heading(1);
                             goal{spmdIndex}.Pose.Orientation.X = heading(2);
                             goal{spmdIndex}.Pose.Orientation.Y = heading(3);
                             goal{spmdIndex}.Pose.Orientation.Z = heading(4);

                             % send ROS messages
                             send(map_pubConstant{spmdIndex}.Value,map_msg{spmdIndex})
                             send(goal_pubConstant{spmdIndex}.Value,goal{spmdIndex})

                             % report cleaning progress
                             fprintf('Robot %d: Task %d complete. Progress: %d of %d\n', ...
   spmdIndex,task_segments{spmdIndex}(currentTask),currentTask,length(task_segments{spmdIndex}))
                         end
```

```
                          % enter this code block if the reAllocateTasks flag is set to 1
                          if reAllocateTasks
                              % report task reallocation is occurring and task stopped on
                              fprintf('Robot %d: Reallocating tasks. Stopped on Task %d\n', ...
                                  spmdIndex,task_segments{spmdIndex}(currentTask))

                              % remove already completed tasks from current task list
                              for i = 1:currentTask-1
                                  task_segments{spmdIndex}(1,:) = [];
                              end

                              % reallocate remaining tasks
                              retasked_segments =
allocateTasks(task_segments{spmdIndex},num_robots)

                              spmdSend(retasked_segments,2,1)   % send robot1retask to robot 2
with tag 1

                              spmdBarrier % barrier #2; send robot 1 retask to robot 2 and wait

                              % loop through the reassigned tasks for complete coverage
                              for currentTask = 1:length(retasked_segments{1})

                                  % get local occupancy map and starting location of current task
                                  [localMap, startXY] = poly2occgrid(polybox,
retasked_segments{1}(currentTask), rawMap, rob, cornerLoc, resolution);

                                  % set ROS messages
                                  map_msg{spmdIndex} =
rosWriteOccupancyGrid(map_msgType{spmdIndex},localMap);

                                  goal{spmdIndex}.Header.FrameId = 'map';
                                  goal{spmdIndex}.Pose.Position.X = startXY(1,1);
                                  goal{spmdIndex}.Pose.Position.Y = startXY(1,2);

                                  heading  = eul2quat([0,0,pi],'XYZ');
                                  goal{spmdIndex}.Pose.Orientation.W = heading(1);
                                  goal{spmdIndex}.Pose.Orientation.X = heading(2);
                                  goal{spmdIndex}.Pose.Orientation.Y = heading(3);
                                  goal{spmdIndex}.Pose.Orientation.Z = heading(4);

                                  % send ROS messages
                                  send(map_pubConstant{spmdIndex}.Value,map_msg{spmdIndex})
                                  send(goal_pubConstant{spmdIndex}.Value,goal{spmdIndex})

                                  % report cleaning progress
                                  fprintf('Robot %d: Reallocated Task %d complete\nProgress: %d
of %d\n', ...
spmdIndex,retasked_segments{1}(currentTask),currentTask,length(retasked_segments{1}))
                              end
                          end

                          finished = 1;   % set finished flag to 1
                          disp('Robot 1: Done')   % report robot 1 has completed its assigned
tasks

                          % enter this code block if the finished flag is set to 1 and the
otherFinished flag is set to 0
                          if (finished == 1) && (otherFinished == 0)
                              spmdSend(finished,2,3)    % send robot1done to robot 2 with tag 3

                              spmdBarrier     % barrier #3; communicate robot 1 has finished
before robot 2; wait for task remaining update from robot 1

                              robot2onetaskleft = spmdReceive("any",6)    % receive2onetaskleft
with tag 6

                              % check status of one task remaining flag
                              if robot2onetaskleft == 0
                                  disp("Robot 1: Receiving reallocated tasks from Robot 2")

                                  spmdBarrier     % barrier #4; wait for robot 2 retask update

                                  retasked_segments = spmdReceive("any",2)    % receive
robot2retask with tag 2

                                  % loop through the received reassigned tasks for complete
coverage
                                  for currentTask = 1:length(retasked_segments{2})

                                      % get local occupancy map and starting location of current
task
                                      [localMap, startXY] = poly2occgrid(polybox,
retasked_segments{2}(currentTask), rawMap, rob, cornerLoc, resolution);

                                      % set ROS messages
```

```
                                     map_msg{spmdIndex} =
rosWriteOccupancyGrid(map_msgType{spmdIndex},localMap);

                                     goal{spmdIndex}.Header.FrameId = 'map';
                                     goal{spmdIndex}.Pose.Position.X = startXY(1,1);
                                     goal{spmdIndex}.Pose.Position.Y = startXY(1,2);

                                     heading  = eul2quat([0,0,pi],'XYZ');
                                     goal{spmdIndex}.Pose.Orientation.W = heading(1);
                                     goal{spmdIndex}.Pose.Orientation.X = heading(2);
                                     goal{spmdIndex}.Pose.Orientation.Y = heading(3);
                                     goal{spmdIndex}.Pose.Orientation.Z = heading(4);

                                     % send ROS messages
                                     send(map_pubConstant{spmdIndex}.Value,map_msg{spmdIndex})
                                     send(goal_pubConstant{spmdIndex}.Value,goal{spmdIndex})

                                     % report cleaning progress
                                     fprintf('Robot %d: Reallocated Task %d complete\nProgress:
%d of %d\n', ...
spmdIndex,retasked_segments{2}(currentTask),currentTask,length(retasked_segments{2}))
                                 end
                             else
                                 disp("Robot 1: Bypassing retask since 1 task remaining on Robot
2")
                             end
                         end
                         spmdBarrier % barrier #0; wait for both robots to complete assigned
tasks
                     catch
                         disp("Both robots have finished at the same time")
                     end
                 end

                 if spmdIndex == 2
                     try
                         disp('Robot 2: Cleaning in progress')

                         % set ROS node constant
                         nodeConstant{spmdIndex} =
parallel.pool.Constant(ros.Node('/Robot2',"http://X.X.X.X:11311"));

                         % set ROS publisher constants
                         map_pubConstant{spmdIndex} = 
parallel.pool.Constant(ros.Publisher(nodeConstant{spmdIndex}.Value,'/localMapRobot2',"nav_msgs/O
ccupancyGrid","DataFormat","struct"));
                         goal_pubConstant{spmdIndex} = 
parallel.pool.Constant(ros.Publisher(nodeConstant{spmdIndex}.Value,'/goalRobot2',"geometry_msgs/
PoseStamped","DataFormat","struct"));

                         % set ROS message types
                         map_msgType{spmdIndex} = 
rosmessage("nav_msgs/OccupancyGrid","DataFormat","struct");
                         goal{spmdIndex} = 
rosmessage("geometry_msgs/PoseStamped","DataFormat","struct");

                         % loop through the assigned tasks for complete coverage
                         for currentTask = 1:height(task_segments{spmdIndex})

                             % check if any message received from other robot
                             if spmdProbe
                                 otherFinished = spmdReceive("any",3)   % receive robot1done 
with tag 3

                                 % check if the received message indicates the other robot has 
finished
                                 if otherFinished == 1
                                     disp('Robot 2: Robot 1 has finished')

                                     % check if the number of remaining tasks is greater than 1
                                     if (height(task_segments{spmdIndex}) - currentTask) > 1
                                         reAllocateTasks = 1;    % set reAllocateTasks flag to 1
                                         robot2onetaskleft = 0;  % set robot2onetaskleft flag to 
0
                                         spmdSend(robot2onetaskleft,1,6) % send 
robot2onetaskleft to robot 1 with tag 6
                                         spmdBarrier     % barrier #3; update robot 1 there is 
more than 1 task remaining for robot 2
                                         break;
                                     else
                                         disp("Robot 2: 1 task remaining")
                                         robot2onetaskleft = 1;  % set robot1onetaskleft to 1
                                         spmdSend(robot2onetaskleft,1,6) % send 
robot2onetaskleft to robot 1 with tag 6
                                         spmdBarrier     % barrier #3; update robot 1 there is 1 
task remianing for robot 2
```

```
                                    end
                                end
                            end

                            % get local occupancy map and starting location of current task
                            [localMap, startXY] = poly2occgrid(polybox,
task_segments{spmdIndex}(currentTask), rawMap, rob, cornerLoc, resolution);

                            % set ROS messages
                            map_msg{spmdIndex} =
rosWriteOccupancyGrid(map_msgType{spmdIndex},localMap);

                            goal{spmdIndex}.Header.FrameId = 'map';
                            goal{spmdIndex}.Pose.Position.X = startXY(1,1);
                            goal{spmdIndex}.Pose.Position.Y = startXY(1,2);

                            heading  = eul2quat([0,0,pi],'XYZ');
                            goal{spmdIndex}.Pose.Orientation.W = heading(1);
                            goal{spmdIndex}.Pose.Orientation.X = heading(2);
                            goal{spmdIndex}.Pose.Orientation.Y = heading(3);
                            goal{spmdIndex}.Pose.Orientation.Z = heading(4);

                            % send ROS messages
                            send(map_pubConstant{spmdIndex}.Value,map_msg{spmdIndex})
                            send(goal_pubConstant{spmdIndex}.Value,goal{spmdIndex})

                            % report cleaning progress
                            fprintf('Robot %d: Task %d complete\nProgress %d of %d\n', ...
   spmdIndex,task_segments{spmdIndex}(currentTask),currentTask,length(task_segments{spmdIndex}))
                        end

                        % enter this code block if the reAllocateTasks flag is set to 1
                        if reAllocateTasks
                            % report task reallocation is occurring and task stopped on
                            fprintf('Robot %d: Reallocating tasks. Stopped on Task %d\n', ...
                                spmdIndex,task_segments{spmdIndex}(currentTask))

                            % remove already completed tasks from current task list
                            for i = 1:currentTask-1
                                task_segments{spmdIndex}(1,:) = [];   % remove already
completed tasks
                            end
```

```
                             % reallocate remaining tasks
                             retasked_segments =
allocateTasks(task_segments{spmdIndex},num_robots)

                             spmdSend(retasked_segments,1,2)   % send robot2retask to robot 1
with tag 2

                             spmdBarrier     % barrier #4; send robot 2 retask to robot 1 and
wait

                             % loop through the reassigned tasks for complete coverage
                             for currentTask = 1:length(retasked_segments{1})

                                 % get local occupancy map and starting location of current task
                                 [localMap, startXY] = poly2occgrid(polybox,
retasked_segments{1}(currentTask), rawMap, rob, cornerLoc, resolution);

                                 % set ROS messages
                                 map_msg{spmdIndex} =
rosWriteOccupancyGrid(map_msgType{spmdIndex},localMap);

                                 goal{spmdIndex}.Header.FrameId = 'map';
                                 goal{spmdIndex}.Pose.Position.X = startXY(1,1);
                                 goal{spmdIndex}.Pose.Position.Y = startXY(1,2);

                                 heading  = eul2quat([0,0,pi],'XYZ');
                                 goal{spmdIndex}.Pose.Orientation.W = heading(1);
                                 goal{spmdIndex}.Pose.Orientation.X = heading(2);
                                 goal{spmdIndex}.Pose.Orientation.Y = heading(3);
                                 goal{spmdIndex}.Pose.Orientation.Z = heading(4);

                                 % send ROS messages
                                 send(map_pubConstant{spmdIndex}.Value,map_msg{spmdIndex})
                                 send(goal_pubConstant{spmdIndex}.Value,goal{spmdIndex})

                                 % report cleaning progress
                                 fprintf('Robot %d: Reallocated Task %d complete\nProgress %d of
%d\n', ...
spmdIndex,retasked_segments{1}(currentTask),currentTask,length(retasked_segments{1}))
                             end
                         end

                         finished = 1;   % set finished flag to 1
                         disp('Robot 2 done')    % report robot 2 has completed its assigned
tasks

                         % enter this code block if the finished flag is set to 1 and the
otherFinished flag is set to 0
                         if (finished == 1) && (otherFinished == 0)
                             spmdSend(finished,1,4)    % send robot2done to robot 1 with tag 4

                             spmdBarrier     % barrier #1; communicate robot 2 has finished
before robot 1; wait for task remaining update from robot 1

                             robot1onetaskleft = spmdReceive("any",5)    % receive
robot1onetaskleft with tag 5

                             % check status of one task remaining flag
                             if robot1onetaskleft == 0
                                 disp("Robot 2: Receiving reallocated tasks from Robot 1")

                                 spmdBarrier     % barrier #2; wait for robot 1 retask update

                                 retasked_segments = spmdReceive("any",1)    % receive
robot1retask with tag 1

                                 % loop through the received reassigned tasks for complete
coverage
                                 for currentTask = 1:length(retasked_segments{2})

                                     % get local occupancy map and starting location of current
task
                                     [localMap, startXY] = poly2occgrid(polybox,
retasked_segments{2}(currentTask), rawMap, rob, cornerLoc, resolution);

                                     % set ROS messages
                                     map_msg{spmdIndex} =
rosWriteOccupancyGrid(map_msgType{spmdIndex},localMap);

                                     goal{spmdIndex}.Header.FrameId = 'map';
                                     goal{spmdIndex}.Pose.Position.X = startXY(1,1);
                                     goal{spmdIndex}.Pose.Position.Y = startXY(1,2);

                                     heading  = eul2quat([0,0,pi],'XYZ');
                                     goal{spmdIndex}.Pose.Orientation.W = heading(1);
                                     goal{spmdIndex}.Pose.Orientation.X = heading(2);
```

```
                                      goal{spmdIndex}.Pose.Orientation.Y = heading(3);
                                      goal{spmdIndex}.Pose.Orientation.Z = heading(4);

                                      % send ROS messages
                                      send(map_pubConstant{spmdIndex}.Value,map_msg{spmdIndex})
                                      send(goal_pubConstant{spmdIndex}.Value,goal{spmdIndex})

                                      % report cleaning progress
                                      fprintf('Robot %d: Reallocated Task %d complete\nProgress
%d of %d\n', ...
spmdIndex,retasked_segments{2}(currentTask),currentTask,length(retasked_segments{2}))
                                  end
                              else
                                  disp("Robot 2: Bypassing retask since 1 task remaining on Robot
1")
                              end
                          end
                          spmdBarrier % barrier #0; wait for both robots to complete assigned
tasks
                      catch
                          disp("Both robots have finished at the same time")
                      end
                  end
              end
          else
              spmd
                  if spmdIndex == 1
                      disp('Robot 1: Cleaning in progress')

                      % set ROS node constant
                      nodeConstant{spmdIndex} =
parallel.pool.Constant(ros.Node('/Robot1',"http://X.X.X.X:11311"));

                      % set ROS publisher constants
                      map_pubConstant{spmdIndex} =
parallel.pool.Constant(ros.Publisher(nodeConstant{spmdIndex}.Value,'/localMapRobot1',"nav_msgs/O
ccupancyGrid","DataFormat","struct"));
                      goal_pubConstant{spmdIndex} =
parallel.pool.Constant(ros.Publisher(nodeConstant{spmdIndex}.Value,'/goalRobot1',"geometry_msgs/
PoseStamped","DataFormat","struct"));

                      % set ROS message types
                      map_msgType{spmdIndex} =
rosmessage("nav_msgs/OccupancyGrid","DataFormat","struct");
                      goal{spmdIndex} =
rosmessage("geometry_msgs/PoseStamped","DataFormat","struct");

                      % loop through the assigned tasks for complete coverage
                      for currentTask = 1:height(task_segments{spmdIndex})

                          % get local occupancy map and starting location of current task
                          [localMap, startXY] = poly2occgrid(polybox,
task_segments{spmdIndex}(currentTask), rawMap, rob, cornerLoc, resolution);

                          % set ROS messages
                          map_msg{spmdIndex} =
rosWriteOccupancyGrid(map_msgType{spmdIndex},localMap);

                          goal{spmdIndex}.Header.FrameId = 'map';
                          goal{spmdIndex}.Pose.Position.X = startXY(1,1);
                          goal{spmdIndex}.Pose.Position.Y = startXY(1,2);

                          heading  = eul2quat([0,0,pi],'XYZ');
                          goal{spmdIndex}.Pose.Orientation.W = heading(1);
                          goal{spmdIndex}.Pose.Orientation.X = heading(2);
                          goal{spmdIndex}.Pose.Orientation.Y = heading(3);
                          goal{spmdIndex}.Pose.Orientation.Z = heading(4);

                          % send ROS messages
                          send(map_pubConstant{spmdIndex}.Value,map_msg{spmdIndex})
                          send(goal_pubConstant{spmdIndex}.Value,goal{spmdIndex})

                          % report cleaning progress
                          fprintf('Robot %d: Task %d complete. Progress: %d of %d\n', ...
spmdIndex,task_segments{spmdIndex}(currentTask),currentTask,length(task_segments{spmdIndex}))
                      end

                      disp('Robot 1: Done')   % report robot 1 has completed its assigned tasks
                      spmdBarrier % barrier #0; wait for both robots to complete assigned tasks
                  end

                  if spmdIndex == 2
                      disp('Robot 2: Cleaning in progress')

                      % set ROS node constant
```

```
                     nodeConstant{spmdIndex} =
parallel.pool.Constant(ros.Node('/BK',"http://192.168.8.178:11311"));

                     % set ROS publisher constants
                     map_pubConstant{spmdIndex} =
parallel.pool.Constant(ros.Publisher(nodeConstant{spmdIndex}.Value,'/localMapBK',"nav_msgs/Occup
ancyGrid","DataFormat","struct"));
                     goal_pubConstant{spmdIndex} =
parallel.pool.Constant(ros.Publisher(nodeConstant{spmdIndex}.Value,'/goalBK',"geometry_msgs/Pose
Stamped","DataFormat","struct"));

                     % set ROS message types
                     map_msgType{spmdIndex} =
rosmessage("nav_msgs/OccupancyGrid","DataFormat","struct");
                     goal{spmdIndex} =
rosmessage("geometry_msgs/PoseStamped","DataFormat","struct");

                     % loop through the assigned tasks for complete coverage
                     for currentTask = 1:height(task_segments{spmdIndex})

                         % get local occupancy map and starting location of current task
                         [localMap, startXY] = poly2occgrid(polybox,
task_segments{spmdIndex}(currentTask), rawMap, rob, cornerLoc, resolution);

                         % set ROS messages
                         map_msg{spmdIndex} =
rosWriteOccupancyGrid(map_msgType{spmdIndex},localMap);

                         goal{spmdIndex}.Header.FrameId = 'map';
                         goal{spmdIndex}.Pose.Position.X = startXY(1,1);
                         goal{spmdIndex}.Pose.Position.Y = startXY(1,2);

                         heading  = eul2quat([0,0,pi],'XYZ');
                         goal{spmdIndex}.Pose.Orientation.W = heading(1);
                         goal{spmdIndex}.Pose.Orientation.X = heading(2);
                         goal{spmdIndex}.Pose.Orientation.Y = heading(3);
                         goal{spmdIndex}.Pose.Orientation.Z = heading(4);

                         % send ROS messages
                         send(map_pubConstant{spmdIndex}.Value,map_msg{spmdIndex})
                         send(goal_pubConstant{spmdIndex}.Value,goal{spmdIndex})

                         % report cleaning progress
```

```
                        fprintf('Robot %d: Task %d complete\nProgress %d of %d\n', ...
  spmdIndex,task_segments{spmdIndex}(currentTask),currentTask,length(task_segments{spmdIndex}))
                    end

                    disp('Robot 2 done')    % report robot 2 has completed its assigned tasks
                    spmdBarrier % barrier #0; wait for both robots to complete assigned tasks
                end
            end
        end
    else
        disp("Current program can only account for 1 or 2 robots")
    end
end
```

APPENDIX C: REQUIRED FUNCTIONS

I. allocateTasks

```
function task_segments = allocateTasks(task_list, numRobots)
% due to limitations of the below process with 3 tasks divided between 2
% robots, check if the number of tasks is equal to 3
if height(task_list) == 3
    task_segments = {task_list(1:2,:)}; % assign tasks 1 and 2 to segment 1
    task_segments(2,:) = {task_list(3,:)};  % assign task 3 to segment 2
else
    % calculate the number of tasks per segment and round down to the nearest integer
    num_task_per_segment = fix(height(task_list)/numRobots);

    % calculate the number of segments with an equal number of tasks and
    % round down to the nearest integer
    num_equal_segments = fix(height(task_list)/num_task_per_segment);

    % generate an array of ones with length equivalent to the number of equal
    % segments and multiply each array element by the number of task per segment
    segment_array = num_task_per_segment*ones(1,num_equal_segments);

    % determine the number of remaining tasks
    rem_tasks = rem(height(task_list),num_task_per_segment);

    % for each remaining task, increase the number of tasks within the
    % current segment index by 1
    for i = 1:rem_tasks
        segment_array(i) = segment_array(i)+1;
    end

    % convert the task list matrix to a cell array
    task_segments = mat2cell(task_list,segment_array,1);
end

% for every other task segment, flip the task order
for i = 1:length(task_segments)
    if mod(i,2) == 1, continue; else, task_segments{i} = flip(task_segments{i}); end
end
end
```

II. directedTSP

```matlab
function [Gdir,nodeList] = directedTSP(poly, roomOutline)
%%% MATLAB TSP Solver Based Example Code with some edits %%%
centroidx = poly.centroid(:,1);
centroidy = poly.centroid(:,2);

x = roomOutline(:,1);
y = roomOutline(:,2);

% calculate distances between points
num_points = length(poly.centroid);
idxs = nchoosek(1:num_points,2);

dist = hypot(centroidy(idxs(:,1)) - centroidy(idxs(:,2)), ...
    centroidx(idxs(:,1)) - centroidx(idxs(:,2)));
lendist = length(dist);

% create and draw graph
G = graph(idxs(:,1),idxs(:,2));
figure
hGraph = plot(G,'XData',centroidx,'YData',centroidy,'LineStyle','none','NodeLabel',{});
hold on
% Draw the outside border
plot(x,y,'r-')
hold off

% Constraints
Aeq = spalloc(num_points,length(idxs),num_points*(num_points-1)); % Allocate a sparse matrix
for ii = 1:num_points
    whichIdxs = (idxs == ii); % Find the trips that include stop ii
    whichIdxs = sparse(sum(whichIdxs,2)); % Include trips where ii is at either end
    Aeq(ii,:) = whichIdxs'; % Include in the constraint matrix
end
beq = 2*ones(num_points,1);

% Binary bounds
intcon = 1:lendist;
lb = zeros(lendist,1);
ub = ones(lendist,1);

% Optimize using intlinprog
opts = optimoptions('intlinprog','Display','off');
```

```
[x_tsp,costopt,exitflag,output] = intlinprog(dist,intcon,[],[],Aeq,beq,lb,ub,opts);
x_tsp = logical(round(x_tsp));
Gsol = graph(idxs(x_tsp,1),idxs(x_tsp,2),[],numnodes(G));

% Visualize solution
hold on
highlight(hGraph,Gsol,'LineStyle','-')
title('Solution with Subtours')

% subtour constraints
tourIdxs = conncomp(Gsol);
numtours = max(tourIdxs); % Number of subtours
fprintf('# of subtours: %d\n',numtours);

A = spalloc(0,lendist,0); % Allocate a sparse linear inequality constraint matrix
b = [];
while numtours > 1 % Repeat until there is just one subtour
    % Add the subtour constraints
    b = [b;zeros(numtours,1)]; % allocate b
    A = [A;spalloc(numtours,lendist,num_points)]; % A guess at how many nonzeros to allocate
    for ii = 1:numtours
        rowIdx = size(A,1) + 1; % Counter for indexing
        subTourIdx = find(tourIdxs == ii); % Extract the current subtour
        %         The next lines find all of the variables associated with the
        %         particular subtour, then add an inequality constraint to prohibit
        %         that subtour and all subtours that use those stops.
        variations = nchoosek(1:length(subTourIdx),2);
        for jj = 1:length(variations)
            whichVar = (sum(idxs==subTourIdx(variations(jj,1)),2)) & ...
                (sum(idxs==subTourIdx(variations(jj,2)),2));
            A(rowIdx,whichVar) = 1;
        end
        b(rowIdx) = length(subTourIdx) - 1; % One less trip than subtour stops
    end

    % Try to optimize again
    [x_tsp,costopt,exitflag,output] = intlinprog(dist,intcon,A,b,Aeq,beq,lb,ub,opts);
    x_tsp = logical(round(x_tsp));
    Gsol = graph(idxs(x_tsp,1),idxs(x_tsp,2),[],numnodes(G));

    % Visualize result
    hGraph.LineStyle = 'none'; % Remove the previous highlighted path
    highlight(hGraph,Gsol,'LineStyle','-')
    drawnow

    % How many subtours this time?
    tourIdxs = conncomp(Gsol);
    numtours = max(tourIdxs); % number of subtours
    fprintf('# of subtours: %d\n',numtours)
end

title('Solution with Subtours Eliminated');
hold off

disp(output.absolutegap)
%%% End MATLAB code %%%

% convert outputted undirected graph Gsol edge table to array
Esol = Gsol.Edges;
Esol_array = table2array(Esol);

% separate undirected edge array into source and target nodes
s_undir = Esol_array(:,1);
t_undir = Esol_array(:,2);

% create a directed graph to be manipulated later from undirected source
% and target nodes
Gdir = digraph(s_undir,t_undir);
% figure; plot(Gdir,'Layout','force');    % visualize arrows pointing in different directions

% extract directed graph Gdir edge table and convert to array
Edir = Gdir.Edges;               % observe that 1 source node can point to multiple targe nodes
Edir_array = table2array(Edir);

% define initial conditions for following loop
current_node = s_undir(1,1);    % starting node
nodeList = [current_node...    % array to track node path with starting node set
    poly.centroid(current_node,1)...
    poly.centroid(current_node,2)];
nodeList_idx = 2;              % index value for node path
% set to position 2 given starting node is in positon 1

% iterate through directed edge array Edir_array such that each source node
% points to only 1 target node
while length(nodeList) <= length(Edir_array)
```

```
     [row,col] = find(Edir_array==current_node); % locate the indices of the current node wihin
the edge array
     index = [row col]; % horizontally concatenate row and col vars into one 2x2 index variable
     % read by row, col 1 value denotes the corresponding edge within the array
     % the current node belongs to
     % col 2 value denotes whether the current node is the source (1) or
     % target (2) node of the edge (source -> target)

     % if the current node is a first edge target AND is not in the source
     % node column, flip the second edge such that the current node is now
     % the second edge source
     if (index(1,2)==2) && (~ismember(Edir_array(index(2,1)),nodeList(:,1)))
         Gdir = flipedge(Gdir,index(2,1));       % flip second edge direction
         Edir_updated = table2array(Gdir.Edges); % update array with new flipped edge
         Edir_array = Edir_updated;

         % if the current node is a first edge target AND is in the source node
         % column, flip the first edge such that the current node is now the
         % first edge source
     elseif (index(1,2)==2) && (ismember(Edir_array(index(2,1)),nodeList(:,1)))
         Gdir = flipedge(Gdir,index(1,1));       % flip first edge direction
         Edir_updated = table2array(Gdir.Edges); % update array with new flipped edge
         Edir_array = Edir_updated;
     end

     % update current node indices following flips
     [row,col] = find(Edir_array==current_node);
     index = [row col];

     % determine next node using first edge index value and target node col of array
     next_node = Edir_array(index(1,1),2);

     % check if the value of the next node is not the same as the current node
     if next_node ~= current_node
         nodeList(nodeList_idx,1:3) = [next_node...  % add next node to node list
             poly.centroid(next_node,1)...
             poly.centroid(next_node,2)];
         current_node = next_node;               % update current node to next node
         nodeList_idx = nodeList_idx+1;        % increment node list index value
     end
 end
 figure; plot(Gdir,'Layout','force');    % visually inspect directed graph output
 end
```

III. identifyBoundaries

```
function [exterior, interior] = identifyBoundaries(editedPGM)
binaryPGM = imbinarize(editedPGM);    % binarize the edited PGM image

[B,~,N,A] = bwboundaries(binaryPGM);
obs_idx = 1;    % variable to count the number of identified obstacles

%%% MATLAB bwboundaries example code with some edits %%%
imshow(binaryPGM); hold on;
% loop through object boundaries
for k = 1:N
    % boundary k is the parent of a hole if the k-th column
    % of the adjacency matrix A contains a non-zero element
    if (nnz(A(:,k)) > 0)
        exterior = B{k};
        plot(exterior(:,2),...
            exterior(:,1),'r','LineWidth',2);

        % loop through the children of boundary k
        for l = find(A(:,k))'
            intBoundary = B{l};

            % additional code to save the points associated with each
            % identified interior obstacle
            tempVar = strcat('obs',num2str(obs_idx));   % update name of obstacle
            interior.(tempVar)= intBoundary;    % save current obstacle to struct variable
            obs_idx = obs_idx+1;    % update obstacle count

            plot(interior.(tempVar)(:,2),...
                interior.(tempVar)(:,1),'g','LineWidth',2);
        end
    end
end
hold off;
%%% End MATLAB example code %%%

% reduce the number of points describing the exterior boundary
exterior = reducepoly(exterior,.015);
end
```

IV. identifyBox2merge

```
function box2merge = identifyBox2merge(rob, polybox, map)
toMerge = []; % initialize an empty array to store node values to be merged
robSize = rob.size; diag = rob.diag; % set robot size and diagonal parameters

for i = 1:length(polybox.nodeList)-1
    currentpoly = polybox.shape(polybox.nodeList(i,1)); % identify the current polygon

    % create a test point to locate the top left vertex of the current polygon
    minx = min(currentpoly.Vertices(:,1));
    maxy = max(currentpoly.Vertices(:,2));
    testpoint = [minx maxy];

    % identify the top left vertex of the current polygon by locating the
    % nearest vertex to the test point
    [~,~,ind] = nearestvertex(currentpoly,testpoint);
    TL = currentpoly.Vertices(ind,:);
    V = world2grid(map,TL); % convert vertex XY coordinates to IJ

    % using the identified vertex as the robot center since this point
    % would result in portions of the robot outside the boundaries of the
    % current polygon, translate the robot center in each of the four
    % cardinal and ordinal directions
    pV.N = [V(1,1)+robSize V(1,2)];
    pV.NE = [V(1,1)+diag V(1,2)+diag];
    pV.E = [V(1,1) V(1,2)+robSize];
    pV.SE = [V(1,1)-diag V(1,2)+diag];
    pV.S = [V(1,1)-robSize V(1,2)];
    pV.SW = [V(1,1)-diag V(1,2)-diag];
    pV.W = [V(1,1) V(1,2)-robSize];
    pV.NW = [V(1,1)+diag V(1,2)-diag];

    % for each robot center point, determine the robot corner points and generate a polyshape
    dir = fieldnames(pV);
    for j = 1:numel(dir)
        vcheck = pV.(dir{j});
        vNE = [vcheck(1,1)+robSize vcheck(1,2)+robSize];
        vSE = [vcheck(1,1)-robSize vcheck(1,2)+robSize];
        vSW = [vcheck(1,1)-robSize vcheck(1,2)-robSize];
        vNW = [vcheck(1,1)+robSize vcheck(1,2)-robSize];
        vpoly(j) = polyshape([vNE;vSE;vSW;vNW]);
    end
```

```
      % convert the vertices of each polyshape from IJ to XY coordinates
     for k = 1:numel(dir)
         verts = vpoly(k).Vertices;
         vpoly(k).Vertices = grid2world(map,verts);
     end

     % for each polyshape determine if each corner is in or outside the
     % boundaries of the current polygon
     in = [];
     for l = 1:length(vpoly)
         in(l,:) =
inpolygon(vpoly(l).Vertices(:,1),vpoly(l).Vertices(:,2),currentpoly.Vertices(:,1),currentpoly.Ve
rtices(:,2));
     end

     % determine if all corners of any of the polyshapes are inside the current polygon
     for m = 1:length(in)
         % if so the current polygon is large enough for
         % the robot to fit and does not need to be merged
         if all(in(m,:)==1)
             toMerge(i,1) = 0;   % set merge flag to 0 and break the loop
             break;
         else
             toMerge(i,1) = 1;   % if no polyshapes are completely enclosed by the current
polygon set merge flag to 1
         end
     end
 end

 % check the merge status of each polygon
 idx = 1;    % initialize an index variable
 for i = 1:length(toMerge)
     if toMerge(i) == 1
         box2merge(idx,:) = polybox.nodeList(i,1);   % identify the node value of the polygon to
be merged and store to list
         idx = idx+1;    % increment index variable
     end
 end

 box2merge = sortrows(box2merge,'ascend');   % sort the list of polygon node values into
ascending order
 end
```

V. identifyTri2merge

```
 function tri2merge = identifyTri2merge(tri, minTri)
 idx = 1; tri2merge = [];
 % loop through triangle node list except for last node as it is the same
 % as the first node
 for i = 1:length(tri.nodeList)-1
     % obtain points A,B,C of each triangle by referenceing the column
     % values of the connectivity list row corresponding to the node value
     % of the ith row of the node list to the correlating row
     % within the list of points to retrieve the corresponding coordinate
     A = [tri.points(tri.list(tri.nodeList(i,1),1),1)
tri.points(tri.list(tri.nodeList(i,1),1),2)];
     B = [tri.points(tri.list(tri.nodeList(i,1),2),1)
tri.points(tri.list(tri.nodeList(i,1),2),2)];
     C = [tri.points(tri.list(tri.nodeList(i,1),3),1)
tri.points(tri.list(tri.nodeList(i,1),3),2)];

     % calculate the triangle edge lengths
     AB = pdist([A;B]); BC = pdist([B;C]); CA = pdist([C;A]);

     % calculate the minimum triangle height using the equation h = 2A/b
     % where b is the maximum base length
     shapeArea = area(polyshape([A;B;C]));
     base = max([AB BC CA]);
     h = (2*shapeArea)/base;

     % determine the minimum triangle base value
     base = min([AB BC CA]);

     % determine if the current triangle minimum base and height values are below the
     % criteria values
     if (base < minTri.base) || (h < minTri.height)
         tri2merge(idx,1) = tri.nodeList(i,1);   % if so, add node value of current triangle to
list to be merged
         idx = idx+1;
     else
         continue;
     end
 end
 tri2merge = sortrows(tri2merge,'ascend');   % sort node values into ascending order
 end
```

VI.   poly2occgrid

```
function [localMap, startXY] = poly2occgrid(poly, idx, rawMap, rob, cornerLoc, resolution)
% obtain the work area polyshape vertices
if idx == 0     % an idx value of 0 denotes a single polyshape object
    localPoints = poly.Vertices;
else
    localPoints = poly.shape(poly.nodeList(idx,1)).Vertices;
end

% check if the raw occupancy map is supplied or is empty
if ~isempty(rawMap), rawMapShift = 1; else, rawMapShift = 0; end

% if the raw occupancy map is available, shift the polyshape vertex points
% to align with the raw map origin
if rawMapShift == 1
    localx = localPoints(:,1) + rawMap.LocalOriginInWorld(1,1);
    localy = localPoints(:,2) + rawMap.LocalOriginInWorld(1,2);
else
    localx = localPoints(:,1); localy = localPoints(:,2);
end

% concatenate and update work area polyshape vertex points
localPoints = [localx,localy];

% identify the minimum and maximum XY coordinates of the work area polyshape
xmin = min(localx); xmax = max(localx);
ymin = min(localy); ymax = max(localy);

% LOCAL OCCUPANCY MAP GENERATION
% generate an occupancy map of the work area polyshape with a padding of .5
localMap = occupancyMap(xmax-xmin+.5,ymax-ymin+.5,resolution);
localMap.GridLocationInWorld = [xmin,ymin]; % update origin of occupancy map
updateOccupancy(localMap,1);    % set all cells within occupancy map to 1 (occupied)

% create a grid of x by y check points to compare against the work area polyshape
step = 1/(2*resolution);    % calculate the step value between each check point
% if the raw occupancy map is available, include additional check point values
% beyond the maximum value
if rawMapShift == 1
    localx_check = xmin:step:xmax+abs(rawMap.LocalOriginInWorld(1,1)/4);
    localy_check = ymin:step:ymax+abs(rawMap.LocalOriginInWorld(1,2)/4);
else
```

```
    localx_check = xmin:step:xmax;
    localy_check = ymin:step:ymax;
end

% generate grid of check points
[localx_grid,localy_grid] = meshgrid(localx_check,localy_check);

% determine which grid points are in or outside the work area polyshape
[in,on] = inpolygon(localx_grid,localy_grid,localx,localy);

% set the cell value to 0 if inside the work area polyshape
setOccupancy(localMap,[localx_grid(in & ~on),localy_grid(in & ~on)],0);

% IDENTIFY STARTING LOCATION
% generate a new work area polyshape based on the updated polyshape vertex points
localShape = polyshape(localPoints);
[x,y] = centroid(localShape);   % determine the polyshape centroid coordinate

% set robot size and diagonal parameters
robSize = rob.size; diag = rob.diag;

% if the desired starting location is the polygon centroid, set the
% starting location to the centroid value
if cornerLoc == 3
    startXY = [x,y];
else
    idx = 1;    % initialize index variable
    % generate a test point for each occupancy map corner
    for corner = 1:4
        % for each loop change the corner location
        switch corner
            case 1  % Top Left
                testpoint = [xmin ymax];
            case 2  % Top Right
                testpoint = [xmax ymax];
            case 3  % Bottom Right
                testpoint = [xmax ymin];
            case 4  % Bottom Left
                testpoint = [xmin ymin];
        end

        % identify the closest work area polyshape vertex to the test point
        [~,~,ind] = nearestvertex(localShape,testpoint);
```

```
testCorner = localShape.Vertices(ind,:);

V = world2grid(localMap,testCorner); % convert vertex XY coordinates to IJ

% using the identified vertex as the robot center since this point
% would result in portions of the robot outside the boundaries of the
% current work area, translate the robot center in each of the four
% cardinal and ordinal directions
pV.N = [V(1,1)+robSize V(1,2)];
pV.NE = [V(1,1)+diag V(1,2)+diag];
pV.E = [V(1,1) V(1,2)+robSize];
pV.SE = [V(1,1)-diag V(1,2)+diag];
pV.S = [V(1,1)-robSize V(1,2)];
pV.SW = [V(1,1)-diag V(1,2)-diag];
pV.W = [V(1,1) V(1,2)-robSize];
pV.NW = [V(1,1)+diag V(1,2)-diag];

% for each robot center point, determine the robot corner points and
% generate a polyshape representative of the possible starting location
dir = fieldnames(pV);
for j = 1:numel(dir)
    vcheck = pV.(dir{j});
    vNE = [vcheck(1,1)+robSize vcheck(1,2)+robSize];
    vSE = [vcheck(1,1)-robSize vcheck(1,2)+robSize];
    vSW = [vcheck(1,1)-robSize vcheck(1,2)-robSize];
    vNW = [vcheck(1,1)+robSize vcheck(1,2)-robSize];
    vpoly(j) = polyshape([vNE;vSE;vSW;vNW]);

    % convert polyshape vertices from IJ to XY coordinates
    verts = vpoly(j).Vertices;
    vpoly(j).Vertices = grid2world(localMap,verts);
end

% check if each of the possible starting location polyshapes are completely
% within the work area polyshape
for k = 1:length(vpoly)
    % check by subtracting the work area polyshape from the
    % possible starting location polyshape
    testpoly = subtract(vpoly(k),localShape);

    % the possible starting location is completely within the work
    % area if the resulting test polyshape has no vertices (no longer exists)
    if height(testpoly.Vertices) == 0
```

```
                  viableStart(idx) = vpoly(k);    % append the current possible starting location
to a viable starting location list
                  idx = idx+1;    % increment the index value
              end
          end
      end

      % visualize the identified viable starting locations within the work area
      clf; figure; hold on; show(localMap); plot(viableStart); hold off

      % calculate centroids of viable starting locations
      [vsX,vsY] = centroid(viableStart);

      % transpose and concatenate centroid coordinates
      startPoints = [vsX',vsY'];

      % calculate the distance between the work area centroid and viable
      % starting location centroids
      dist = pdist2([x,y],startPoints);

      % sort the distance list according to the desired starting location
      if cornerLoc == 1
          [~,I] = sort(dist,'descend'); % farthest viable starting location from work area
centroid
      elseif cornerLoc == 2
          [~,I] = sort(dist,'ascend');  % closest viable starting location to work area centroid
      end

      % make sure starting location is within the work area polyshape
      for j = 1:length(startPoints)
          checkStart = startPoints(I(j),:);   % obtain the current check point based on the
sorted distance index

          % check if current check point is within the work area polyshape
          in =
inpolygon(checkStart(1,1),checkStart(1,2),localShape.Vertices(:,1),localShape.Vertices(:,2));

          % if so, set this coordinate as the starting location
          if in; startXY = checkStart; break; end
      end
 end
 end
```

VII. polyConn

```
 function Gpoly = polyConn(pshape)

 pbuff = polybuffer(pshape,.0001);   % add a buffer to each polyshape
 pborder = zeros(length(pshape));    % allocate an adjacency matrix of zeros of size NxN where N
is equal to the number of polyshapes

 % loop through each polyshape object and determine which other polyshapes border it
 for i = 1:length(pshape)
     for j = (i+1):length(pshape)
         % if the area of intersection between the current polyshape and another exceeds the
         % set threshold, set the corresponding adjacency matrix location to true
         pborder(j,i) = area(intersect(pbuff(j),pbuff(i))) > 3e-6;
     end
 end

 Gpoly = graph(pborder,'lower'); % generate a connectivity graph of the filled in adjacency
matrix
 end
```

VIII. polyNeighbor

```
function neighbor2merge = polyNeighbor(Gpoly, node2merge, pshape)
% determine the row locations of the input polygon within the connectivity graph
[row,col] = find(Gpoly.Edges.EndNodes==node2merge);
index = [row col];

% for each identified row, determine the neighboring polygon by examining
% the other node within the row
for i = 1:height(index)
    if index(i,2) == 1
        neighbors(i) = Gpoly.Edges.EndNodes(index(i,1),2);
    elseif index(i,2) == 2
        neighbors(i) = Gpoly.Edges.EndNodes(index(i,1),1);
    end
end

% add a buffer to the current polygon and identified neighbors
neighborBuff = polybuffer(pshape(neighbors),.01);
nodeBuff = polybuffer(pshape(node2merge),.01);

% for each identified neighbor, determine the area of intersection between
% it and the current polygon
for k = 1:length(neighbors)
    border(k) = area(intersect(nodeBuff,neighborBuff(k)));
end

% determine which neighbor borders the current polygon the most and merge
% with this neighbor
[~,I] = max(border);
neighbor2merge = neighbors(I);
end
```

IX. triDecomposition

```
function tri = triDecomposition(room, min_triEdge)
% perform initial triangulation on room
TR = triangulation(room);
figure; triplot(TR); axis padded;   % visually inspect initial triangulation

% generate a geometric model based on initial triangulation
model = createpde;
tnodes = TR.Points';
telements = TR.ConnectivityList';
geometryFromMesh(model,tnodes,telements);

% generate triangular mesh
room_mesh = generateMesh(model,'GeometricOrder','linear','Hmin',min_triEdge);

% regulate the number of generated triangles in the mesh to 250 or less
if length(room_mesh.Elements') >= 250
    num_triangles = length(room_mesh.Elements');
    min_triEdge = min_triEdge+.1;   % increment target minimum edge length to create larger 
triangles

    loop_count = 1; % initialize loop counter
    while (num_triangles > 250) && (loop_count <= 500)
        % regenerate triangular mesh
        room_mesh = generateMesh(model,'GeometricOrder','linear','Hmin',min_triEdge);

        num_triangles = length(room_mesh.Elements');    % update number of triangles within 
mesh
        min_triEdge = min_triEdge+.1;   % increment target minimum edge length
        loop_count = loop_count+1;  % increment loop counter
    end
end

figure; pdeplot(room_mesh); % visually inspect triangular mesh

tri.list = room_mesh.Elements'; % save connectivity list of mesh to struct
tri.points = room_mesh.Nodes';  % save list of triangle points to struct

% loop through each triangle within the mesh and generate a polyshape of
% each triangle
for i = 1:length(tri.list)
    % obtain points A,B,C of each triangle by referenceing the column
    % values within the ith row of the connectivity list to the correlating row
    % within the list of points to retrieve the corresponding coordinate
    A = [tri.points(tri.list(i,1),1) tri.points(tri.list(i,1),2)];
    B = [tri.points(tri.list(i,2),1) tri.points(tri.list(i,2),2)];
    C = [tri.points(tri.list(i,3),1) tri.points(tri.list(i,3),2)];

    pgon = polyshape([A;B;C]);  % generate polyshape using points A,B,C
    tri.shape(i) = pgon;    % save generated polyshape to struct

    [tri.centroid(i,1),tri.centroid(i,2)] = centroid(pgon); % save polyshape centroid to struct
end
end
```

APPENDIX D: SUPPLEMENTARY FUNCTION

I. cleanRoomSimTest

```
 function [map,timeElapsed,turn_count,overlap,pathLength,deadlock_cntr] = cleanRoomSimTest(map,
start,deadlock_cntr)

 tic
 show(map)

 n = zeros(map.GridSize(1), map.GridSize(2)); %generate array to store neural activity values

 % For plotting
 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
 % indicator for robot heading direction on map
 up_dir = '^r';
 down_dir = 'vr';
 left_dir = '<r';
 right_dir = '>r';
 hold on
 pc = start; %starting grid location
 xy = grid2world(map,pc); %convert starting grid location to xy coordinates
 plot(xy(1), xy(2), down_dir) %plot starting location xy coordinates on map
 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

 go = 1;
 deadlockSearch = 0;

 %Building neural network of map
 %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
 prevHeading = 0; turn_count = 0; overlap = 0;
 prevPC = []; pathLength = 0;

 while go == 1
     %find top left corner points in all directions
     TL = [pc(1)-6, pc(2)-12];
     NW_TL = [pc(1)-25, pc(2)-37];
     N_TL = [pc(1)-25, pc(2)-12];
     NE_TL = [pc(1)-25, pc(2)+13];
     E_TL = [pc(1)-6, pc(2)+13];
     SE_TL = [pc(1)+13, pc(2)+13];
```

```
    S_TL = [pc(1)+13, pc(2)-12];
    SW_TL = [pc(1)+13, pc(2)-37];
    W_TL = [pc(1)-6, pc(2)-37];

    %update status of current grid
    for tl_j = TL(2):1:TL(2)+24
        for tl_i = TL(1):1:TL(1)+18
            setOccupancy(map,[tl_i, tl_j],0.5,'grid')
        end
    end
    pause(0.005)
    show(map)

    %--------------------PATH SELECTION ALGORITHM------------------------------%
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    %Heuristic sequence
    %left -> bottom -> top -> right -> right bottom -> right top

    %Find status of all possible next positions
    kN = directionStatus(N_TL);
    kNE = directionStatus(NE_TL);
    kE = directionStatus(E_TL);
    kSE = directionStatus(SE_TL);
    kS = directionStatus(S_TL);
    kW = directionStatus(W_TL);

    % modify check the status of all possible next locations
    pn =[kW, kS, kN, kE, kSE, kNE]; %don't change order of this array

    %check the status of all possible next locations
    for value = 1:1:length(pn)
        if pn(value) < 0
            pn(value) = -1; %indicate cell is obstacle
        elseif pn(value) > 0.5
            pn(value) = 1; %indicate cell is uncovered
        elseif (pn(value) > 0) && (pn(value) < 0.5)
            pn(value) = 0; %indicate cell is covered
        end
    end

    %modify deadlock event
    if sum(pn==1) == 0 %check to for deadlock event
        kSW = directionStatus(SW_TL);
        kNW = directionStatus(NW_TL);
        deadlockSearch = deadlockSearch+1;
        search = [kN,kNE,kE,kSE,kS,kSW,kW,kNW]; %search for uncovered location in closest
neighbors using neural propagation
        switch max(search)
            case search(1)
                pc = moveNextDirection(N_TL,up_dir);
                heading = 1;
                %disp('north is next')
            case search(2)
                pc = moveNextDirection(NE_TL,up_dir);
                heading = 2;
                %disp('ne is next')
            case search(3)
                pc = moveNextDirection(E_TL,right_dir);
                heading = 3;
                %disp('east is next')
            case search(4)
                pc = moveNextDirection(SE_TL,down_dir);
                heading = 4;
                %disp('se is next')
            case search(5)
                pc = moveNextDirection(S_TL,down_dir);
                heading = 5;
                %disp('south is next')
            case search(6)
                pc = moveNextDirection(SW_TL,down_dir);
                heading = 6;
                %disp('sw is next')
            case search(7)
                pc = moveNextDirection(W_TL,left_dir);
                heading = 7;
                %disp('west is next')
            case search(8)
                pc = moveNextDirection(NW_TL,up_dir);
                heading = 8;
                %disp('nw is next')
        end

        %once uncovered location is found revert back to heuristic path selection
    else %continue on with heuristic path selection
        deadlockSearch = 0;
        switch max(pn)
```

```
            case pn(1)
                pc = moveNextDirection(W_TL,left_dir);
                heading = 7;
                %disp('west is next')
            case pn(2)
                pc = moveNextDirection(S_TL,down_dir);
                heading = 5;
                %disp('south is next')
            case pn(3)
                pc = moveNextDirection(N_TL,up_dir);
                heading = 1;
                %disp('north is next')
            case pn(4)
                pc = moveNextDirection(E_TL,right_dir);
                heading = 3;
                %disp('east is next')
            case pn(5)
                pc = moveNextDirection(SE_TL,down_dir);
                heading = 4;
                %disp('se is next')
            case pn(6)
                pc = moveNextDirection(NE_TL,up_dir);
                heading = 2;
                %disp('ne is next')
        end
    end

    % Stop condition
    %check if map/environment is completely covered and end operation
    if sum(n>=0.5, 'all') == 0
        disp("Coverage complete, stopped at...")
        disp(grid2world(map, pc))
        go = 0;
    end

    % collect simulation data
    if deadlockSearch > 30
        disp("Deadlock max reached. Terminating Sim");
        deadlock_cntr = deadlock_cntr+1;
        break;
    end

    if prevHeading ~= heading; turn_count = turn_count+1; end
```

```
     prevHeading = heading;

     checkPC = checkOccupancy(map,pc,'grid');
     if checkPC == -1; overlap = overlap+1; end

     pcXY = grid2world(map,pc);
     if ~isempty(prevPC)
         dist = pdist2(pcXY,prevPC);
         pathLength = pathLength+dist;
     end
     prevPC = pcXY;

 end %end of while loop

 timeElapsed = toc;
 return

     function y = directionStatus(x)
         %Use a try block to iterate through and calcuate n-values for all cells eqauting to the
size of bot in specified direction
         try
             for j = x(2):1:x(2)+24 %{Reference; j is cols (goes left to right)}
                 for i = x(1)-1:1:x(1)+18  %{Reference; i is rows (goes up and down)}
                     status = checkOccupancy(map, [i,j], "grid"); %check status of current grid
position
                     %convert grid values to 'I' values
                     if status == 1
                         i = x(1) + 6;
                         j = x(2) + 12;
                         I = -100;
                         calculateNeuralActivity
                         return
                     end
                 end
             end

             %if no cell in the group is an obstacle, check the status of cell where COG is
located and use it's n-value for the specified direction
             i = x(1)+6;
             j = x(2)+12;
             %ab = grid2world(map, [i,j]);
             status = checkOccupancy(map, [i,j], "grid");
             switch status
```

```
                case -1 %uncovered
                    I = 0;
                    calculateNeuralActivity
                    return
                case 0 %covered
                    I = 100;
                    calculateNeuralActivity
                    return
            end

        catch %Catch "index out of bounds" errors and assign n-value of specified direction as
obstacle (-0.6)
            y = -0.6;
            return
        end %for the try/catch

        function calculateNeuralActivity
            I_plus = max([I 0]);
            I_neg = max([-1*I 0]);

            %check neural activity level of neighboring neurons, evaluate using ReLu, and
calcuate euclidean distance
            %input the correct displacement values for the COG of the neighboring direction
            [North, dNorth] = ReLu(-19,0); %n(i, j-1)
            [South, dSouth] = ReLu(19,0); %n(i+1, j)
            [West, dWest] = ReLu(0,-25); %n(i, j-1)
            [East, dEast] = ReLu(0,25); %n(i, j+1)
            [NW, dNW] = ReLu(-19,-25); %n(i-1, j-1)
            [NE, dNE] = ReLu(-19,25); %n(i-1, j+1)
            [SW, dSW] = ReLu(19,-25); %n(i+1, j-1)
            [SE, dSE] = ReLu(19,25); %n(i+1, j+1)

            %store evaluated neural activity values for neighboring neurons (8x1 matrix/array)
            xplus = [North, South, West, East, NW, NE, SW, SE];

            %store euclidean distances (1x8 matrix/array)
            wij = [dNorth; dSouth; dWest; dEast; dNW; dNE; dSW; dSE];

            %calculate the weight
            weight = xplus * wij; % Matrix operation calculating for the weight (order of
operation should not be changed)

            %set variables
```

```
              A = 80;
              B = 1;
              D = 1;
              %define & evaluate equation using ODE solver
              eqn = @(t,Xi) ((-A*Xi) + (B-Xi)*(I_plus + weight) - (D+Xi)*I_neg);
              [t,Xi] = ode45(eqn, [0:1], n(i,j));
              n(i,j) = Xi(end);
              y = n(i,j);

          end

          % function to check neural activity level of neighboring neurons, evaluate using ReLu,
and calcuate euclidean distance
          function [dir, dir_dist] = ReLu(RowDisp,ColDisp)
              %input the correct displacement values for the COG of the neighboring direction
              ab = grid2world(map, [i,j]); %current grid position for neural activity calculation
              try
                  dir = max([n(i + RowDisp, j + ColDisp) 0]);
                  dir_dist = 0.7/((norm(ab - grid2world(map, [i + RowDisp, j + ColDisp]))));
              catch
                  dir = 0;
                  dir_dist = 0;
              end
          end

      end %for the function directionStatus()

      function pc = moveNextDirection(pTL,heading)
          pc = [pTL(1)+6, pTL(2)+12];
          gpc = grid2world(map, pc);
          plot(gpc(1), gpc(2),heading)
      end
  end
```