2004

Comparison of path-planning and search methods for cooperating unmanned aerial vehicles

Zachary Wilson Spritzer
West Virginia University

Comparison of Path-Planning and Search Methods For Cooperating Unmanned Aerial Vehicles

Zachary Wilson Spritzer

Thesis Submitted to the
College of Engineering and Mineral Resources
at West Virginia University
In Partial Fulfillment of the Requirements
For the Degree of

Master of Science
in
Aerospace Engineering

Marcello Napolitano, Ph. D., Chair
Gary Morris, Ph. D.
Jacky Prucz, Ph. D.

Department of Mechanical and Aerospace Engineering
Morgantown, West Virginia
2004

Keywords: Unmanned Aerial Vehicles, Path Planning, Task Allocation

Abstract

Comparison of Path-Planning and Search Methods for Unmanned Aerial Vehicles

Zachary W. Spritzer

The main goal of this research effort is develop a simulation environment for cooperating UAVs within MATLAB's SIMULINK. This is the first step in a process that will eventually lead to the implementation of model UAVs on a model battlefield. The interest in cooperation of UAVs over the past decade has grown significantly. This is due to several reasons including lower operational cost, lower risk for humans, and greater maneuverability.

This research explores two scenarios. The first is a scenario in which all of the characteristics of a battlefield are known prior to the UAVs being launched. Three prevalent path-planning methods are compared based on calculation speed and optimization. This thesis shows that a visibility graph method leads to the lowest cost solution, while the Voronoi diagram method provides a computationally inexpensive solution.

The second scenario is a search and destroy mission where nothing is known about the battlefield prior to UAVs launch. This will consist of the vehicles visiting a set of predetermined waypoints until a target is found. The result of this research produces a simulation of cooperating UAVs that shows the potential of fulfilling many realistic missions in a battlefield environment.

Acknowledgements

I would like to thank everyone that has made this document possible. Most importantly I like to thank my family, Nancy and Mark, which have helped me tremendously. Also, I would like to thank my girlfriend Cortney who helped me through the long nights. To my friends graduate school has been an interesting journey thanks for making it fun and to the Dave Matthews Band for giving me something good to listen to while working.

I would like to thank my committee chair Dr. Marcello Napolitano for all of his guidance and help over the last several years; it has proved to be invaluable. I also would like to also thank my committee members Dr. Jacky Prucz and Dr. Gary Morris for their assistance.

Finally, I would like to thank everyone that I've worked with over the course of the last year and half in the aerolab Matt Lechliter, Jennifer Hazelton, and Srikanth Gururajan, and to the people upstairs Elena Lucci, Dr. Giampero Campa, Dr. Mario George Perhinschi, and Dr. Brad Seanor. Thank you.

Table of Contents

List of Tables

List of Figures

Nomenclature

English	Units	Description
J_i	-	Total cost for line *i*
$J_{i,t}$	-	Threat cost for line *i*
$J_{i,f}$	-	Fuel cost of line *i*
L_i	-	Length of line *i*
k	-	Weighting factor
V	-	Number of vertices in Voronoi diagram
E	-	Number of edges in Voronoi diagram
v_1	-	Starting vertex in Dijkstra's Algorithm
v_2	-	Finishing vertex in Dijkstra's Algorithm
v_{max}	m/s	Maximum speed
f	N	Maximum g force for the aircraft
J	-	Total cost in MMKP algorithm
c_j	-	Cost of choice *j*
x_j	-	Binary decision variable
V_{ij}	-	Vehicle constraint
w_i	-	Vehicle constraint value
p_i	-	Probability of kill of threat
w	-	Weighting factor of threat
x	km	X position of object
y	km	Y position of object
d	km	Distance of a line

$d_{s,c}$	km	Distance start of line to center of threat
$d_{f,c}$	km	Distance finish of line to center of threat
$d_{s,f}$	km	Distance start to finish of line
$d_{s,n}$	km	Distance parallel to threat
d_p	km	Distance closest point on line
$Cost_M$	-	Mission Cost
$Cost_C$	-	Current Cost
N_V	-	Number of vehicles
F_{AX}	N	Force along the x axis
F_{AY}	N	Force along the y axis
F_{AZ}	N	Force along the z axis
L_A	Nm	Moment around the x axis
M_A	Nm	Moment around the y axis
N_A	Nm	Moment around the z axis
m	kg	Mass of aircraft
U	m/s	Velocity along x axis
V	m/s	Velocity along y axis
W	m/s	Velocity along z axis
P	m/s	Velocity around x axis
Q	m/s	Velocity around y axis
R	m/s	Velocity around z axis
g_x	N	Gravity along x axis
g_y	N	Gravity along y axis

g_z	N	Gravity along z axis
I_{XX}	N/m^2	Moment of inertia x axis
I_{YY}	N/m^2	Moment of inertia y axis
I_{ZZ}	N/m^2	Moment of inertia z axis
I_{XZ}	N/m^2	Product of inertia x and z
I_{XY}	N/m^2	Product of inertia x and y
I_{YZ}	N/m^2	Product of inertia y and z
$\bar{q}$	N/m^2	Local dynamic pressure
S	m^2	Area

Greek	Units	Description
ω	km	Maximum turn rate of vehicle
φ	degrees	Euler angle for heading
θ	degrees	Euler angle for pitch
ϕ	degrees	Euler angle for bank
α	degrees	Angle of attack
β	degrees	Sideslip angle
δ_A	degrees	Aileron deflection angle
δ_R	degrees	Rudder deflection angle
δ_E	degrees	Elevator deflection angle

Chapter 1

Introduction to Cooperating UAVs

1.1 - Introduction to Unmanned Aerial Vehicles

As technology grows, it is apparent that the use of Unmanned Aerial Vehicles (UAVs) will serve a larger purpose in military forces. The first UAV was developed in the 1960s as a supplement to the U-2 spy plane. The military program for this UAV was called Compass Arrow; the military designation for this aircraft was AQM-91A. Project Compass Arrow led to the development of an aircraft that had the capability to operate for two hours at 85,000 ft while maintaining subsonic speeds around Mach 0.8. This flight envelope gave the vehicle the ability to survive against threats such as anti-aircraft fire. Like many of the unmanned military aircraft of the 1960s, the AQM-91A was launched from a DC-130 aircraft and recovered by parachute[1]. In the 1970s, the military began to fund programs that would lead to vehicles with a larger flight envelope and operational time, which lead to the end of the Compass Arrow project in 1973. The trend of large high altitude UAVs continued into the 1980s including Boeing's Condor that boasted a gross weight of 16,000 lbs with the capability to operate for over 50 hours at an altitude of 65,000 ft.

The Department of Defense changed the trend of large UAVs in the late 1980s by establishing the UAV Joint Project Office (JPO). This shifted the focus to the development of small, low altitude, and low cost UAVs. It was clear that the UAV JPO's objective was to give UAVs global acceptance as a low cost disposable aircraft. These new smaller UAVs were designed to replace larger manned aircraft in a battlefield

environment. This led to projects such as the RQ-2 Pioneer, which was used in the 1990s in Operation Desert Storm[2]. The RQ-2 Pioneer was used primarily for target identification and battle damage assessment. It proved to be a great resource instead of using manned aircraft because the Pioneer benefited from lower operational cost and higher pilot safety.

As the advantages of using these vehicles for battlefield applications became more apparent, several other UAVs were developed including the RQ-1 Predator and the RQ-4 Global Hawk. The Predator played an important role in Bosnia as a reconnaissance and surveillance platform[2]. Both the Predator and the Global Hawk have been invaluable resources in recent conflicts such as Operation Enduring Freedom in Afghanistan and Operation Iraqi Freedom in Iraq. In recent years, the military has been developing several UAV programs that call for the aircraft to perform more tasks on the battlefield. Some of these programs include the modification of the Predator into a search and destroy aircraft, the Boeing X-45, and the Northrop Grumman X-47, which are all being designed as Unmanned Combat Air Vehicles (UCAVs).

Some of the many advantages UAVs posses over manned aircraft are excellent maneuverability, lower operational cost, large weight savings, dramatically lower human risk, and an opportunity to achieve superior coordination[3]. With the role of UCAVs becoming larger in the military, some of the missions they have the potential of achieving are the following:

- Reconnaissance
- Communication Jamming
- Suppression of Enemy Air Defenses

- Missile Defense
- Fixed/Moving Target Attack
- Air-to-air Combat
- Search and Destroy

It is evident that the implementation of multiple UAVs on a battlefield to complete these missions has tremendous potential. In addition to the vehicles becoming exceedingly complex, these tasks must be accomplished using superior coordination. Clearly, as UAVs and UCAVs take larger roles on the battlefield an enhanced level of control is required to operate these aircraft.

1.2 - Research Objectives

Along with the growing technology of UAVs comes the need to control and coordinate these vehicles. There are two main objectives of this research; the first is the development of a control scheme for a group of cooperating UAVs in a hostile environment and the second is the development of a control scheme for a group of cooperating UAVs in a search and destroy environment. The design and simulation of both objectives have been performed using Mathworks' SIMULINK environment in MATLAB. The first objective is using a hostile environment that implies a given number of conditions on the battlefield are known prior to launch. In the second objective a search and destroy environment is used in which the only knowledge about the battlefield is its area.

For the purposes of this research a hostile environment is defined as a battlefield that includes several no-fly zones, threats, targets, and UAVs. No-fly zones can be

political boundaries or physical boundaries such as mountains, which are modeled as a half-sphere with known location and radius. In this application the threats are considered to be a variety of surface-to-air missiles (SAM) and an anti-aircraft artillery weapon; the specifications for these are shown in Table 1.2.1[17]. The locations, ranges, and probability of kill for each threat are known.

Table 1.2.1 – List of different threats used

Threat Name	Threat Description	Threat Range	Probability of Kill
KS-19	100 mm Anti-Aircraft Artillery	4000 ft	40%
SA-7 Grail	Shoulder Fired SAM	5000 ft	50%
Crotale Rattlesnake	Vehicle Fired SAM	10000 ft	80%
V-75 SA-2 Guideline	Vehicle Fired SAM	30000 ft	80%

The targets are a point on the battlefield with known location and value. The value of a target can be in the range of 1-100, which is dependent on how valuable the target is to mission completion. A target can be various areas of interest such as buildings or enemy camps. In addition to the initial conditions of the battlefield described above, the initial location, speed, and heading angle of each UAV are also known. The objective of cooperating UAVs in a hostile environment is to minimize the mission completion time while maximizing the probability of mission completion. Many different algorithms for the simulation of cooperating UAVs have been developed with this main objective[3,6,7,10,13,15].

There are several steps involved in solving the cooperating UAVs problem. The first step is the generation of possible paths for the UAVs to follow in order to reach the targets. Several methods for the generation of these paths has been tried including the use of Delaunay triangulation or Voronoi diagrams[7,9,13], a grid[7], and a visibility graph[5,6,8].

A Voronoi diagram is constructed based solely on the locations of the threats and no-fly zones. The grid method involves the overlaying of a grid onto the battlefield. In contrast to both of these methods a visibility graph is based on the ranges of the threats and radii of the no-fly zones. Typically, the next step is assigning costs to all of these paths. In this case there are two costs assigned to each path. The first is the fuel cost, which is calculated as the Euclidian distance of each path[10]. The second is the cost associated with threat risk that is based on whether the path travels inside a threat's range or a no-fly zone's radius. In this research if a path travels through a threat's range a cost proportional to that threat's probability of kill is added to the path, also if a path travels through a no-fly zone's radius a cost of infinity is assigned to that particular path.

After costs for all of the paths are assigned, a lowest cost path must be selected for each permutation of UAV to target. This is accomplished through the use of a directed graph search algorithm such as Dijkstra's algorithm[7,13]. In a directed graph each segment of the graph has a starting point and an ending point. After all of the lowest cost paths have been selected for each UAV to travel to each target, they must be transformed into flyable paths. This is needed in order to give an accurate representation of the limitations that each UAV faces due to the dynamics of each aircraft. The final step in this process is to assign tasks for each UAV to perform or which target each UAV should visit. This problem was formulated as a Multi-Dimensional Multiple-Choice Knapsack Problem[5,6,8,11] (MMKP). The MMKP algorithm assigns each UAV a task leading to the global optimal solution for the cost of the mission.

The second objective of this research is the simulation of cooperating UAVs in a search and destroy environment. A search and destroy environment is defined such that

nothing is known except the area to be searched and the starting position of the UAVs. This type of mission has been researched by many people[18-24]. The process can be broken down into two main steps. The first step is assigning waypoints to each vehicle so that each UAV searches the given area in a serpentine pattern[23]. This pattern is used because nothing about the battlefield is known prior to launch. This is referred to as a random search in which no area in the battlefield is preferred over another[20]. After the waypoints for each UAV are defined, the area is then searched until a UAV detects a target.

The second step in this process is to assign UAVs to perform tasks on the targets as they are found. This is formulated as a market-based bidding procedure, which performs the task assignment[22]. In this procedure, after a target is detected every vehicle provides an estimate of the cost to visit the target. The vehicles with the lowest estimated costs are selected to visit the detected target.

As the case in both of the objectives, several tasks need to be performed on the targets. After a potential target is identified or detected, it needs to be classified as a target or not a target. If it is classified as a target, it must be destroyed by a vehicle. In order to determine if the target has been destroyed a battle damage assessment (BDA) must be performed. After all of the necessary tasks are preformed on the targets the UAVs are then free to visit other lower value targets or search the rest of the battlefield until another target is found or the entire area has been searched.

These two objectives must be implemented using the SIMULINK environment in MATLAB for the purpose of incorporating six degree of freedom aircraft dynamics. SIMULINK provides an extremely proficient environment to simulate dynamic systems,

which is especially important given these two objectives. In both circumstances the need to simulate dynamic changes in the environment is desired. Some of which are the changing of target states and the addition or subtraction of vehicles, threats, and targets. In addition, MATLAB provides an excellent coding interface similar to C++ and other computer languages, but it is designed in a math-oriented environment. This leads to an easier and more user-friendly way to simulate the desired system. Aside from MATLAB being a math oriented environment, the program is preloaded with many mathematical programming functions, which proves very beneficial to the research objectives of this project.

Chapter 2

Literary Review

2.1 - Review of Path-Planning and Task Allocation Methods

There have been several research efforts that take the approach in which everything about a battlefield is known prior to the launch of the UAVs. This has led to many different approaches by researchers to solve this problem. In general, the problem is the development of a path-planning algorithm with integrated task allocation. This algorithm must compute a trajectory from the UAV's present location to a desired future location[7]. In order for a path-planning algorithm to be optimal, it must yield the optimal path for each UAV to travel while accounting for two extremely important factors. These paths must be stealthy to avoid known enemy threat locations. Also, they must be of minimal length to minimize the cost of the mission and the time in enemy territory. This algorithm must be coded with software that can be executed on an airborne processor[7].

Much research has been done in this area especially with the use of Delaney triangulation or Voronoi diagrams[7,9,13]. In this research a Voronoi diagram is created based solely on the locations of static threats. This method yields paths that are optimal between previously known threats. For every three threats a Delaunay triangulation is calculated, which forms a circle that passes through these three points. The center of the circle that is created is called a Voronoi point[7]. After all of the Voronoi points in the battlefield are defined, lines are drawn connecting these points. These points are only connected if their Delaunay triangle shares a common edge. This process forms a graph of connected lines called a Voronoi diagram. In order to generate paths for the UAVs to

travel they must be connected into the diagram using the three closest nodes[13]. In addition, the targets are connected into the graph in the same fashion.

After all of the lines in the diagram have been defined, the cost of traveling along those lines must be assigned. The cost associated with each particular line consists of two components, which are the threat proximity cost and the fuel cost[9]. This leads to the total cost for the line *i*, J_i, shown in the following equation

$$J_i = J_{i,t} + J_{i,f} \tag{2.1.1}$$

where $J_{i,t}$ is the threat cost and $J_{i,f}$ is the fuel cost of the line *i*. The threat cost is calculated by finding the exposure of each line to enemy radar and is given by the expression[13]

$$J_{t,i} = L_i \sum_{j=1}^{N} \left(\frac{1}{d^4_{\frac{1}{6},i,j}} + \frac{1}{d^4_{\frac{1}{2},i,j}} + \frac{1}{d^4_{\frac{5}{6},i,j}} \right) \tag{2.1.2}$$

where N is the number of threats, $1/d^4$ is the strength of a UAV's radar signature, which is calculated at the 1/6, 1/2, and 5/6 point along each line, and L_i is the length of each line. The fuel cost is simply calculated as the length of each line, L_i. These two costs yield a final line cost[13]

$$J_i = k * J_{i,t} + (1-k) J_{i,f} \tag{2.1.3}$$

where *k* is between 0 and 1, which allows the total cost to be weighted toward a stealthy mission or a low fuel cost mission.

After the costs of each line in the diagram have been assigned, a graph search method such as Dijkstra's algorithm can be used to find the lowest cost path from one point to any other point in the diagram. If 'V' is the number of vertices in the diagram and 'E' is the number of edges or lines, then the complexity of solving the algorithm is

O(V log(V) + E)[7]. In order to use the algorithm the graph must be a weighted and directed graph, which requires all of the lines to be assigned a positive cost and a direction. After all of the lines are assigned costs and directions, the lowest cost path from one point to another point can be found. The main concept of Dijkstra's algorithm is to change temporary labels associated with vertices to permanent labels, which gives the lowest cost path from a source vertex to another vertex in the graph[25]. The application of Dijkstra's algorithm from source vertex v_1 to another vertex v_2 is outlined below:

Algorithm 2.1.1

1. Set $P = (v_1)$, $T = V - (v_1)$, $d(v_1) = 0$, $pred(v_2) = 0$, $d(j) = c_{nj}$
2. Do for all $(v_2, j) \in A$
   $d(j) = \infty$ for other vertices, $pred(j) = v_1$
3. Do while $P \neq V$
   choose the minimum $i \in T$, $d(i) = \min(d(j) : j \in T)$
4. Update P, and T
   $P = P \bigcup (i)$, $T = T - (i)$
5. Update temp labels, for all $j \in A(i)$
   $d(j) = \min(d(j), d(i) + c_{ij})$, set $pred(j) = i$
6. Go back to step 3
7. Go back to step 2

Dijkstra's algorithm is a time efficient and effective way to search a given directed graph for the lowest cost path from a starting point to any other point in the graph. In this application the algorithm is used to find the lowest cost path for each permutation of

UAV to target. Given that these paths are the lowest cost, they are neither the shortest possible path nor the safest possible path[13].

Once these paths have been selected, the dynamic constraints for the aircrafts must be implemented in order to give an accurate estimation of each path. This step is referred to as 'path refinement' or 'trajectory generation'. In order to simplify the model of the UAV dynamics the following assumptions are made by Bortoff[7].

- Each UAV flies at a constant altitude.
- Each UAV flies at a constant speed.

The constant altitude assumption is used to simplify the numerical complexity of the path-planning problem. The second assumption is made for the purpose of simplifying the calculations involved to find the length of each path. Using this assumption the path length can be estimated using Cartesian coordinates. Both of these assumptions are reasonable and simplify the complexity of the problem a great deal.

Richards states in a similar manner the aircraft is modeled as a point mass moving in a 2-D environment[6]. Although the aircraft can be modeled as a point mass, several other considerations must be taken into account. One such consideration is the maximum turning rate of the aircraft, which is given in equation 2.1.4.

$$\omega = \frac{f}{v_{\max}} \tag{2.1.4}$$

where v_{max} is the maximum velocity of the aircraft and f is the force applied to the aircraft. Considering these factors a flyable path is constructed for each UAV, in order to follow the dynamic constraints of the aircraft such as maintaining an acceptable turning rate and appropriate airspeed to avoid stall conditions. This flyable path is given by a set

of points along which the UAV is assigned to travel. This path is then assigned an updated cost based on the dynamic constraints of the aircraft.

The final step in the path-planning process takes place after all of the costs for each UAV to visit each target using a flyable path are defined. This step performs a task allocation of each UAV that leads to a globally optimal solution for the mission. The task allocation problem is formulated as a Multi-Dimensional Multiple-Choice Knapsack Problem (MMKP)[8]. The objective of the MMKP problem is to minimize the knapsack while satisfying all of the conditions placed on the problem. In this problem, the knapsack is the total mission cost. The multiple dimensions are the UAVs in which each vehicle has a multiple choice of the waypoint to visit.

Knapsack problems are an important class of problems that have many various applications in fields such as management, business, defense, or any other area in which tasks must be scheduled or budgeted[11]. The MMKP algorithm is a combination of two separate algorithms, the Multiple-Choice Knapsack Problem (MCKP) and the Multiple-Dimensional Knapsack Problem (MDKP). The MCKP is a problem in which there are multiple resource constraints for the knapsack. In the MDKP, there are several groups of items where one item is selected from each group. By combining the resource constraints from the MCKP with the selection of the different groups from the MDKP an algorithm for the MMKP is created.

There are two methods for solving an MMKP; one is a method that finds the exact solution and the other results in a heuristic solution[26]. Finding the exact solution to a MMKP is extremely computationally expensive, but can be accomplished using the branch and bound with linear programming (BBLP) technique. The algorithm for

solving MMKP using the BBLP technique is formulated as a zero-one knapsack problem. This leads to an exhaustive analysis, this technique can be seen in equations 2.1.5 through 2.1.7 from Bellingham[5].

$$J = \sum_{j=1}^{N_M} c_j x_j \tag{2.1.5}$$

$$\sum_{j=1}^{N_M} V_{ij} x_j \geq w_i \tag{2.1.6}$$

$$\sum_{j=N_p}^{N_{p+1}-1} x_j = 1 \tag{2.1.7}$$

where the cost function J is minimized with respect to the constraints in equations 2.1.6 and 2.1.7. The number of permutations of vehicle p are numbered N_p to $N_{p+1} - 1$, with N_1 and $N_{Nv+1} = N_M + 1$. The indices i, j, *and* p have ranges from 1 to N_W, N_M, and N_V respectively. In the cost equation, c_j is a vector of the costs for each permutation and x_j is a binary decision variable equal to one if the permutation j is selected or zero if the permutation is not selected.

The first constraint guarantees that each waypoint or target is visited the correct number of times, which for most cases is one. The second constraint prevents a vehicle from selecting more than one permutation. In this case each waypoint must be visited once and each vehicle may only be assigned one waypoint to visit. This particular algorithm, which leads to an exact solution, is extremely complex but is guaranteed to find the optimal solution for the knapsack. It should be noted that this solution it is not feasible to apply to all cases where a solution is desired. The second method for finding a solution to the MMKP is a heuristic method which has been researched by Moser[11] and

Akbar[26]. This particular method as shown by Moser is accomplished using Lagrange multipliers. This method leads to sub-optimal results, which is not a desirable result.

The path-planning and task allocation process described above leads to a globally optimal mission cost. The resulting mission cost is neither the lowest in fuel cost nor stealth cost, but is the best combination of the two. It can be seen that by using the methods described above a near real-time simulation can be created in the SIMULINK environment in MATLAB. From previous research it is apparent that the most computationally intense hurdles will be finding a way to limit the calculations required for the Dijkstra and the MMKP algorithms.

2.2 - Review of Search Methods

The second focus of this research is the development of a simulation in which nothing about a battlefield is known prior to the UAVs being deployed. There have been several different approaches to this problem. One approach is a random search in which every area of the battlefield is assigned the same value[18,19,21-24,]. Another approach is a greedy search in which there are more valuable areas of the battlefield than others, thus a way of weighting different areas on the battlefield is required[20]. For the purpose of this research only the first scenario will be considered, due to the fact that previous knowledge about the battlefield is accounted for in the previous section.

The most notable research effort in the random search approach has taken place at the Wright-Patterson Air Force Base in Dayton, OH. This research has led to a search simulation that is implemented in a hierarchical manner with inter-vehicle communication explicitly modeled[23]. This simulation was created using MATLAB's

SIMULINK environment and is named MultiUAV. The purpose of MultiUAV is to simulate a group of UAVs searching a battlefield and attacking any target that is detected. In this simulation the UAVs are modeled as disposable munitions. These types of vehicles are considered to be destroyed once they attack a target. This search mission is generally known as a wide area search munitions weapon system is which all of the vehicles operate independently of each other. Initially the vehicles are released in a target area and follow a set of waypoints that are present at the start of the simulation[22]. These waypoints are placed in a serpentine pattern to minimize the time it takes the group of UAVs to search the given area.

A target is first detected when it passes though the sensor footprint of a UAV. When a target is detected the vehicle communicates the location of the target to the rest of the group. A top level controller is then used to determine the task assignment for the UAVs. This controller is implemented using a hierarchical market-based bidding procedure, where each aircraft bids on each task that needs to be preformed. An optimal solution is reached with this method by having each UAV evaluate its cost to perform a certain task. This control system is developed as distributed to create a redundant system, which is fault tolerant because there is no central decision maker. All of the vehicles arrive at the same decisions; therefore conflict situations are avoided[22]. An example of this problem is multiple UAVs visiting the same target or a target not being visited at all. After the top layer of control has assigned the tasks, the lower layer control system performs the trajectory optimization and task management.

The initial state of a target in a search mission is not detected. After a target is detected it must be classified. When a target has been classified as a viable target it must

then be attacked. Due to the fact that these aircraft are disposable munitions after a vehicle attacks a target it must be eliminated from the group. If a target has been attacked a battle damage assessment (BDA) must be performed to ensure that the target has been destroyed[21]. The BDA of a target will result in two conclusions, the first being that the target has been destroyed and no further action is needed on that target. The second conclusion is that the target has not been destroyed and requires the processes to be repeated until the target is destroyed. In most cases a target needing to be attacked more than once is not likely. The mission is considered complete when the entire battlefield has been searched, all of the UAVs have been eliminated, or all of the targets have been destroyed. A market-based bidding procedure with a hierarchical control system is an excellent tool that can be used in the creation of a search simulation.

Chapter 3

The Path-Planning and Task Allocation Process

3.1 - Path Generation and Path Selection

This section will discuss the path generation, path cost assignment, and path selection steps in the path-planning and task allocation process used in this research effort. This process is based on a combination of different methods that have been discussed in the previous chapter. The objective is to select a path generation and cost assignment method that will lead to optimal results using algorithms that can be executed in a real-time manner. In order to accomplish this objective a Voronoi diagram will be used for possible path generation, because it will yield a low number of possible paths. This is desired to keep the calculations involved in Dijkstra's algorithm to a minimum. Dijkstra's algorithm is the most computationally expensive part of the path generation and selection process.

The first step is the generation of possible paths on which the UAV can travel. The use of Voronoi diagrams is an excellent method to perform this step. This graph yields the optimal paths to travel between a set of points. For this application the set of points that must be avoided are the locations of the threats and no-fly zones. Since the Voronoi diagram only uses points, the ranges and radii of the threats and no-fly zones are ignored at this time. A Voronoi diagram is constructed using a method called Delaunay triangulation. As described by Bortoff, this procedure begins with complete knowledge of each point to be avoided[7]. This can be seen in Figure 3.1.1.

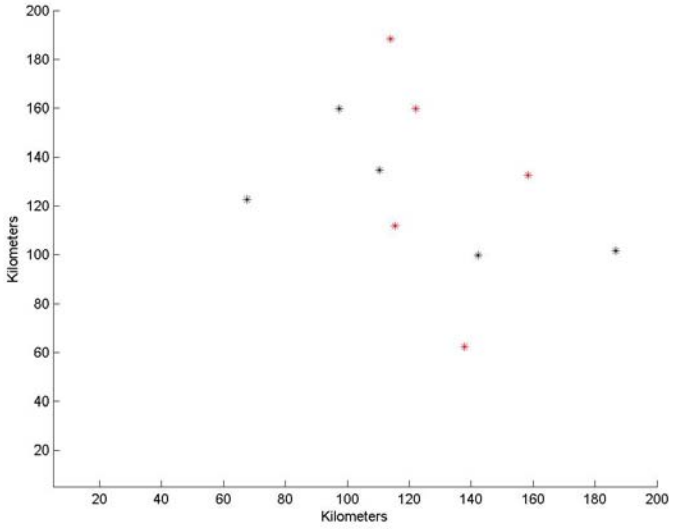


Figure 3.1.1 – Locations of the threats and no-fly zones

In this figure the red points are the threats and the black points are the no-fly zones. For every three points there exists a circle that passes through these points. The Delaunay triangulation of these points exists only if there are no points enclosed in this circle. The center of this circle is called a Voronoi point and is visible in Figure 3.1.2.

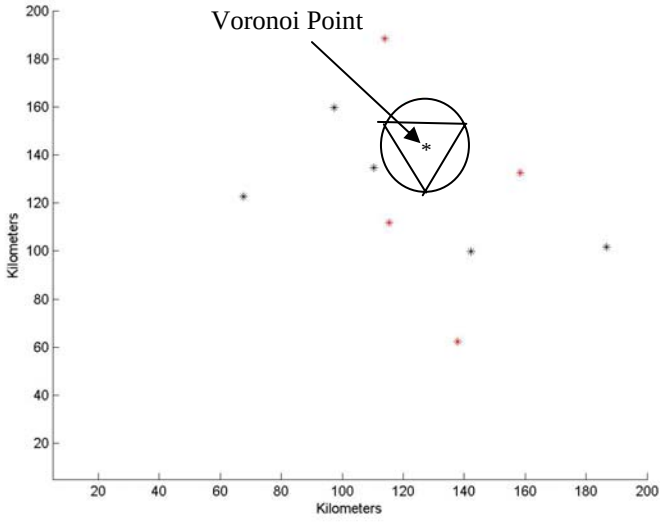


Figure 3.1.2 – Delaunay triangulation and the corresponding Voronoi point

After all of the Voronoi points are defined they must be connected in order to form the diagram. This is accomplished by connecting two points if and only if the Delaunay triangles associated with these points share a common edge. This method provides optimal results because each line in the diagram is equidistant to the pair of corresponding points. All of these lines and points form a complete Voronoi diagram, which can be seen in Figure 3.1.3.

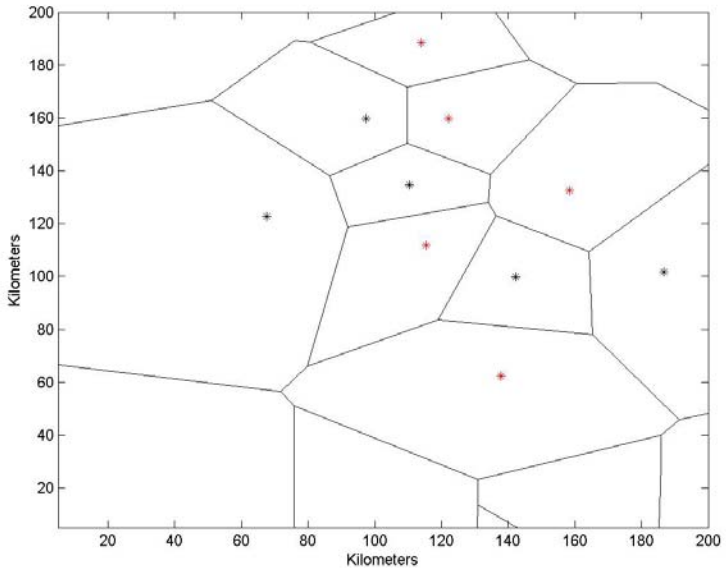


Figure 3.1.3 – Complete Voronoi diagram

Due to the fact that the Voronoi diagram only accounts for the locations of the threats and no-fly zones, the UAVs and targets must be manually connected into the diagram. This is done by connecting each UAV and each target to the three closest points in the diagram. The implementation of this is shown in Figure 3.1.4. In this figure the blue points are the UAVs and the green points are the targets.

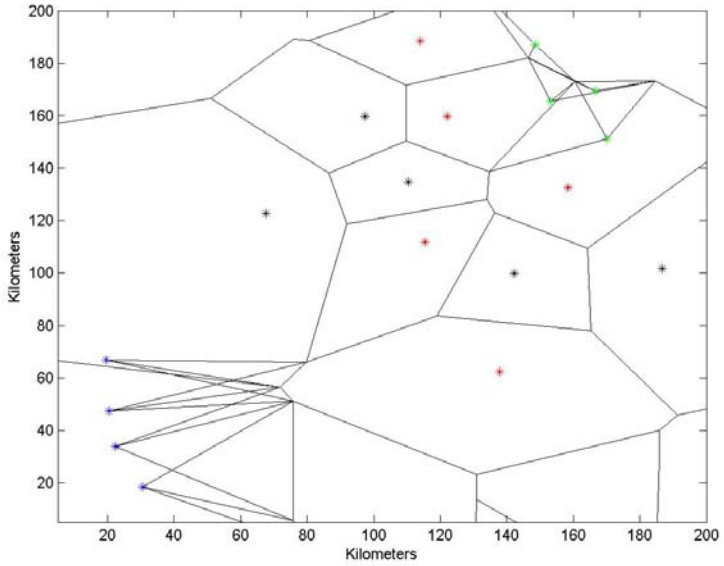


Figure 3.1.4 – Complete Voronoi diagram with UAVs and targets

A function was written in MATLAB named *"vrn_diag_gen"* to perform this which can be seen in Appendix A. The inputs for this function are the initial conditions of the battlefield as discussed in Chapter 1, which are the '*UAVS*', '*TARGETS*', '*ZONES*', and '*THREATS*' matrices. The '*UAVS*' matrix contains the initial x position, y position, speed, and altitude of each UAV. The '*TARGETS*' matrix contains the initial x position and y position of the targets. The '*ZONES*' matrix contains the initial x position, y position, and radius of each no-fly zone. The '*THREATS*' matrix contains the initial x position, y position, range and probability of kill of the threats. Using the positions of the threats and no-fly zones an initial Voronoi diagram is created. This is accomplished using the "*voronoi*" function in MATLAB. This can be seen in Algorithm 3.1.1. It should be noted, that due to the nature of the Voronoi diagram several points around the battlefield were added. This was needed so that the graph would completely encompass the area.

Algorithm 3.1.1

1. Do for all points (x,y)
2. Find Delaunay triangulation of all points (x_1, y_1), (x_2, y_2), (x_3, y_3)
3. Re-orient triangles so they are clockwise
4. If triangle edges for two points are the same
   record edge as Voronoi line (x_1, y_1) and (x_2, y_2),
5. Delaunay triangle defines a circle
6. If another point is not inside the circle,
   record point as a Voronoi point (x, y)
7. Go back to step 1

After all of the Voronoi lines and points have been defined the UAVs and targets must be connected into the diagram. This is accomplished using the "*connect_vrn*" function. This function inputs the positions of the UAVs or targets and the Voronoi points. It outputs the lines connecting the UAVs or targets to the three closet points and the associated distance of each line created. This process is outlined in the following:

Algorithm 3.1.2

1. Do for all points to be connected $(x_{i,} y_i)$
2. Find distance to all points in diagram $(x_{j,} y_j)$
   $d = \sqrt{\left(x_j - x_i\right)^2 + \left(y_j - y_i\right)^2}$
3. Record the closest 3 points (x_1, y_1), (x_2, y_2), (x_3, y_3)
   and their associated distance d_1, d_2, d_3
4. Go back to step 1

Every point, line, and distance associated with this diagram is output from the "*vrn_diag_gen*" function, which are the matrices '*all_pos*', '*all_lines_x*', '*all_lines_y*', and '*all_costs*' respectively.

The next step is the initial path selection for each permutation of UAV to target. Before this can be calculated the costs of the paths must be updated to account for the threats and no-fly zones on the battlefield. The subsequent equations are used to update the cost of each line.

$$c_j = \infty \tag{3.1.1}$$

$$c_j = p_i * w_T + c_{j_{OLD}} w_F \tag{3.1.2}$$

where c_j is the cost of traveling along line j, p_i is the probability of kill of threat i, w_T is the weighting factor applied to traveling through a threat, and w_F is the fuel weighting factor. Equation 3.1.1 shows the modification done to the cost of line j if it passes through a no-fly zone, which is shown in Figure 3.1.5.

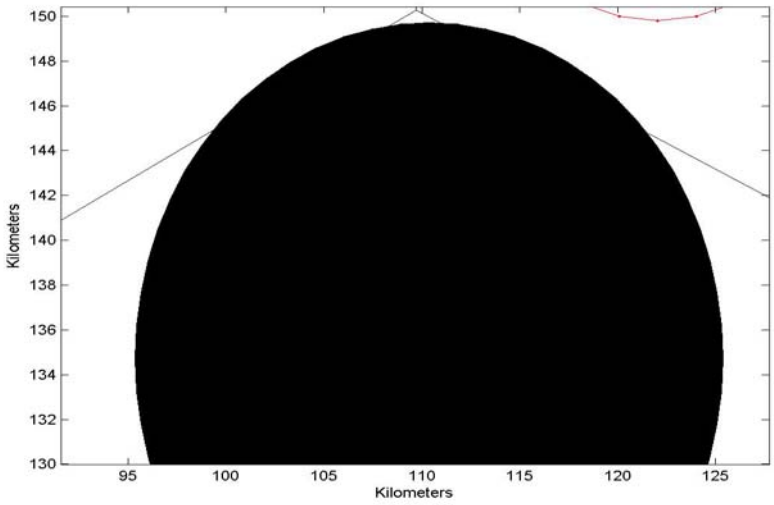


Figure 3.1.5 – Voronoi line passing through a no-fly zone's radius

Since entering a no-fly zone is prohibited, a cost of infinity is assigned to that particular line. Figure 3.1.6 shows line j passing through threat i.

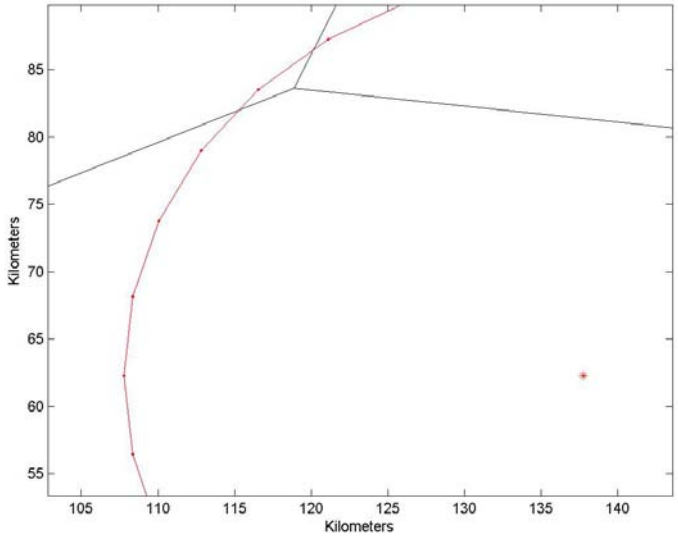


Figure 3.1.6 – Voronoi line passing through a threat's range

If this occurs a cost, proportional to the probability of kill of threat *i*, is added to the cost of that line, which can be seen in equation 3.1.2. The following algorithm is used to evaluate if a line passes through a threat or no-fly zone.

Algorithm 3.1.3

1. For all lines (x_s, y_s) and (x_f, y_f)
2. Find distances associated with that line to an obstacle

   start of line to center of obstacle, $d_{s,c} = \sqrt{(x_c - x_{Ps})^2 + (y_c - y_{Ps})^2}$

   finish of line to center of obstacle, $d_{f,c} = \sqrt{(x_c - x_{Pf})^2 + (y_c - y_{Pf})^2}$

   start of line to finish of line, $d_{s,f} = \sqrt{(x_{Ps} - x_{Pf})^2 + (y_{Ps} - y_{Pf})^2}$

   point of line perpendicular to obstacle, $d_{s,n} = \dfrac{d_{s,t}^2 + d_{s,f}^2 - d_{f,t}^2}{2 * d_{s,f}}$
3. If $d_{s,n} \le d_{s,f}$ and $d_{s,n} \ge 0$ then

   closest distance, $d_p = \sqrt{d_{s,c}^2 - d_{s,n}^2}$
4. Else If $d_{s,c} \le d_{f,c}$

closest distance, $d_p = d_{s,c}$

5. Else

   closest distance, $d_p = d_{f,c}$

6. Go back to step 2

If the closest point on each line is less than the radius or range of that obstacle the cost of that line is updated according to Equation 3.1.1 or 3.1.2.

After the costs of each line have been updated, Dijkstra's algorithm is implemented to find the lowest cost path from each UAV to each waypoint. Dijkstra's algorithm is a graph search algorithm that provides the optimal path from a starting node to every other node in the graph. In order for the graph to be searched it must be a directed graph, which means that each line of the graph must have a tail, head, and an associated cost. In this research each line in the graph has the ability to travel both from tail to head and from head to tail. The results for Dijkstra's algorithm are shown in Figure 3.1.7.

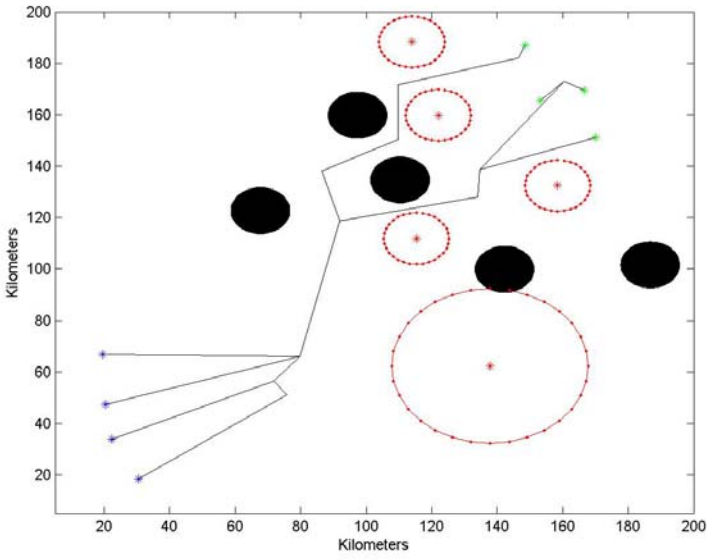


Figure 3.1.7 – Dijkstra's algorithm selected paths from each UAV to each target

A function was written in MATLAB to perform the path selection named "*cheapest_paths"* and can be found in Appendix A. The inputs of this function are the following matrices:

- *all_pos*
- *all_lines_x*
- *all_lines_y*
- *all_costs*
- *UAVS*
- *TARGETS*
- *ZONES*
- *THREATS*

Each of these matrices has been described previously. To place all of the lines in the proper format a function named "*set_THC"* was written. This function rearranges the lines and their associated costs into the '*THC'* matrix, where '*T'* is the tail of the line, '*H'* is the head of the line, and '*C'* is the cost of traveling along that line. Algorithm 3.1.4 shows the implementation of the "*set_THC"* function.

Algorithm 3.1.4

1. For all lines L_i
2. Place cost of L_i in $THC(i,3)$
3. If tail node is assigned a value
   place value in $THC(i,1)$
4. Else assign the node the lowest unused value
   place value in $THC(i,1)$
5. If head node is assigned a value
   place value in $THC(i,2)$

6. Else assign the node the lowest unused value
   place value in $THC(i,2)$
7. Go back to step 2

The '*THC*' matrix is then input into the function "*c_assign*", the purpose of this function is to assign new costs to each line based on if it enters a no-fly zone's radius or a threat's range. This is accomplished using Algorithm 3.1.3. After the costs for each line are updated the '*THC*' matrix is input into the "*dijk*" function, which performs Dijkstra's algorithm. The "*dijk*" function and its associated functions are from Kay's matlog, a logistics engineering MATLAB toolbox, which is available to download[27]. The outline of this algorithm is shown in Algorithm 2.1.1. This function provides the optimal path to travel from one node to another node within the graph. It also gives the cost associated with that path. The optimal paths and their associated costs for each permutation of UAV to target are stored in the matrices '*stored_paths*' and '*totalcost*' respectively, which are output from the function "*cheapest_paths*".

3.2 - Path Refinement and Task Allocation

This section will discuss the refinement of the initially selected paths and the task allocation for the group of UAVs. Since these selected paths are derived from a Voronoi diagram, they rarely travel as close as possible to the outer range of a threat or the outer radius of a no-fly zone. In addition, these paths have sharp corners that might not be flyable. These paths also do not account for a change in heading angle. Clearly, there is a need to refine these paths, which requires them to be optimized and developed into flyable paths. The lines in a Voronoi diagram are designed to yield the optimal paths to

avoid certain points on a battlefield. This yields a solution that has a tendency to avoid these points as much as possible and in many cases much further than is needed. After these paths have been optimized and developed into flyable paths a task must be assigned to each vehicle. This must yield a solution that leads to mission completion in an optimal manner. The final step in the path-planning process is a task allocation of the UAVs in order for them to visit the targets as needed to complete the mission. This leads to each UAV being assigned to visit a certain target along an optimal flyable path, which results in a globally optimal mission completion time and probability of mission completion.

As stated above, the nature of the Voronoi diagram is to avoid certain points as much as possible. This leads to paths that can be optimized. These paths can be improved by shortening them along the original path. This is accomplished according to whether a chosen path travels inside a threat's range or a no-fly zone's radius. The path is first split into several segments, which allows for an improved solution. The original path is then explored to see if it passes through a threat; if it does then the distance at it enters the threat is recorded. The shortened path will be allowed to enter that particular threat only that distance. The shortening of a path is accomplished by analyzing a line starting at the UAV's initial position and ending at the final position. This line is examined to see if it passes through a threat or no-fly zone using Algorithm 3.1.3.

This algorithm yields closest point on the line to the position of each threat and no-fly zone. This distance is compared with the corresponding range or radius associated with the obstacle. If this distance is greater than the range or the allowable entry distance of every threat and the radius of every no-fly zone. That line is recorded as the new optimized path. On the other hand, if the line intersects a threat or no-fly zone then the

previous point is evaluated. This process is repeated until the path has been shortened to the original starting point. An example of this can be seen in Figure 3.2.1.

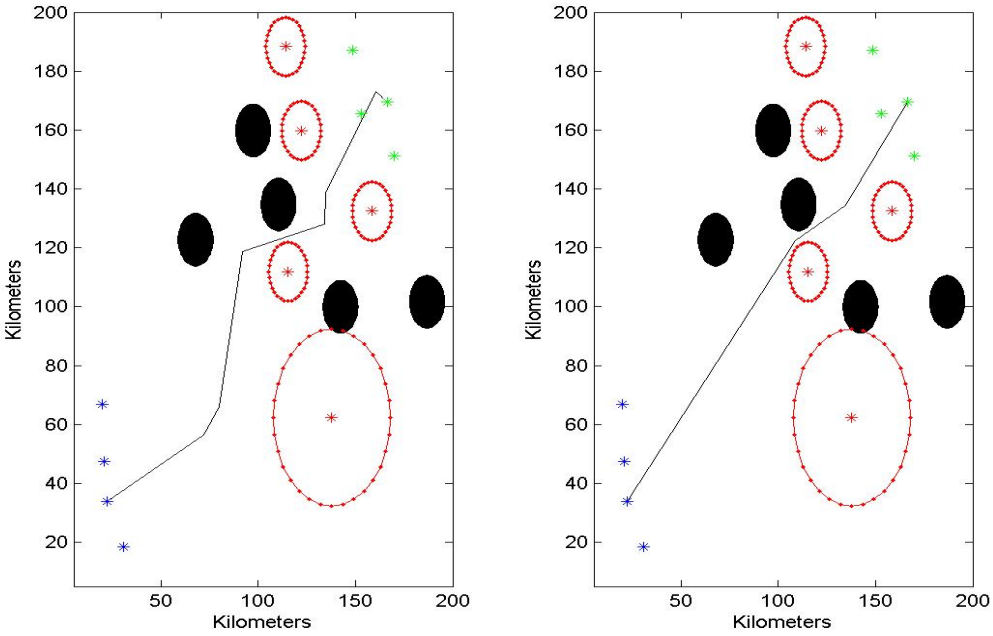


Figure 3.2.1 – Example of a shortened path

After an optimized path is found, this path must then be modified to account for the flight characteristics of the aircraft. There are two main changes that need to be made to each path. One is that each corner in the path must be filleted. This is done according to the minimum turn radius of the UAV, which is one kilometer. In order to fillet the sharp corners of the paths, a circle with the desired radius is placed into the corner. The radius of this circle is equal to the minimum turn radius of the aircraft. The two points on the circle tangent to each of the two lines form the fillet, which replaces the point at the corner. An example of this is shown in Figure 3.2.2.

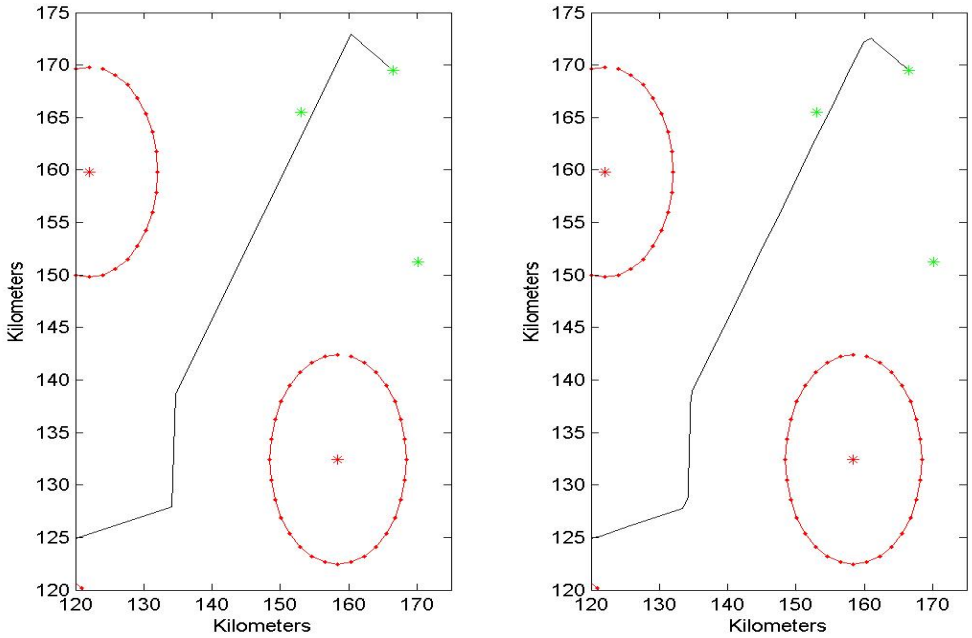


Figure 3.2.2 – Example of the filleted corner of a path

The second modification must be made in order to perform a heading angle change. This is done to account for the sudden change in heading angle a UAV experiences when its current heading angle and the heading angle proposed by the selected path are vastly different. In order to account for this change, the new path must first travel along a circle connected to its current path. Another circle is then placed connecting the first circle to the desired path. The intersection of these two circles is a transfer point at which the UAV leaves the circle connected to its current path and starts to follow the circle on the new path. The radii of these circles are equal to the minimum turn radius of the aircraft. These circles are fitted so that the starting point of the path does not change; merely the heading angle of the aircraft is corrected. An example of this can be seen in Figure 3.2.3.

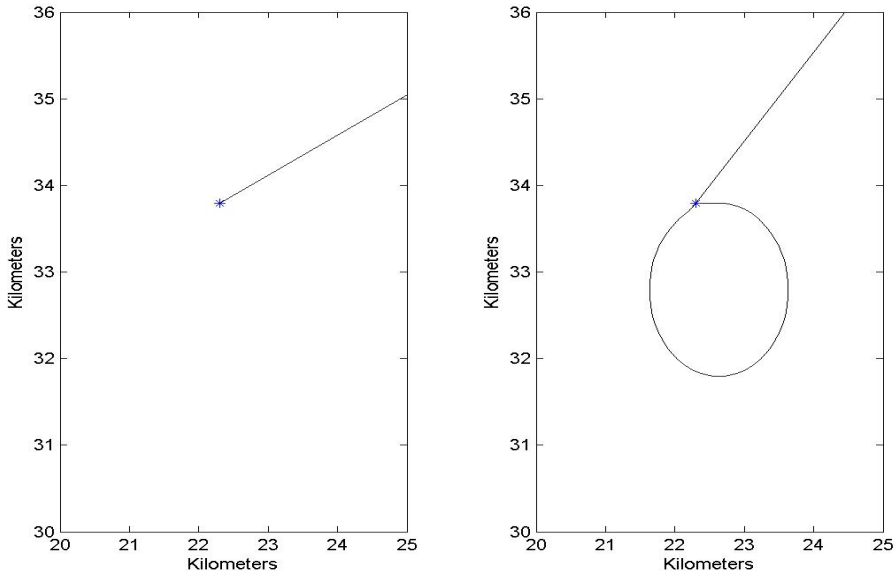


Figure 3.2.3 – Example of a heading angle correction

A function was written in MATLAB for the purpose of optimizing these paths and making them flyable. The function called "*path_shrtng*" can be found in Appendix A. This function inputs the matrices '*stored_paths*', '*all_pos*', '*ZONES*', '*THREATS*', and '*HEADING_ANGLE*'. All of these matrices have been described in the previous section except '*HEADING_ANGLE*', which is a vector containing the current heading angles for all of the UAVs. Other inputs to the function are the scalar numbers '*min_turn*', '*split_seg*', '*nuav*', and '*ntarg*'. These represent the minimum turn radius of the UAVs, the number of segments each line in the original path is split, the number of UAVs, and the number of targets. First this function splits each of the lines in the path into several segments as specified by the variable '*split_seg*'.

After each line is split the path is shortened using a function called "*shorten_paths*", which performs the optimization of the paths as described previously and is accomplished using Algorithm 3.1.3. The corners of the path must now be filleted

using the function "*fillet_path*". The purpose of this function is to add fillets to the corners of the path that are too sharp for the aircraft to follow. This is outlined in the Algorithm 3.2.1:

Algorithm 3.2.1

1. For all lines L_i
2. For all points (x_i, y_i)
3. Set α equal to the angle between (x_{i-1}, y_{i-1}) and (x_i, y_i)
3. If $\alpha \leq \alpha_{ALLOWABLE}$, $\alpha_{ALLOWABLE}$ is proportional to the minimum turn radius
   fillet corner of (x_{i-1}, y_{i-1}) and (x_i, y_i)
4. Go back to step 2

The final step in making the paths flyable is to make the heading angle correction. A function "*heading_angle_paths*" was written for the purpose of accomplishing this task. This process has been described previously and illustrated in the following algorithm:

Algorithm 3.2.2

1. For all paths P_i
2. If $\varphi_{OLD} - \varphi_{NEW} \geq 30°$
   apply heading angle change at beginning of path
3. Else dynamics will handle the change
4. Go back to step 1

After the paths are optimized and made flyable the costs of these paths are updated using the function "*update_cost*". These modified paths and updated costs are stored into the

'*Shortened_Paths_x'*, '*Shortened_Paths_y'*, and the '*totalcost'* matrices, which are the outputs of the "*path_shrtng"* function.

After an optimal flyable path for each permutation of UAV to each target has been developed, a task allocation must be performed in order to delegate which target each UAV should visit. These tasks must be allocated to achieve a global minimum mission cost as opposed to assigning each UAV its minimum path. This was formulated as a MMKP, which has been described in Chapter 2. The constraints placed on this problem for the purpose of this research are the following:

- Equal number of UAVs and targets
- Each target can only be visited once
- Each UAV can only visit one target

These constraints are applied to reduce the complexity of the MMKP algorithm. These constraints reduce the number of possible combinations of the task allocation to the factorial of the number of UAVs. The following algorithm was developed to achieve an optimal solution in a minimal amount of time, while accounting for the constraints of the problem.

Algorithm 3.2.3

1. Set minimum cost $Cost_M = \infty$
2. For $i = 1$ to $N_V!$
3. Initialize $Cost_C = 0$
4. For j to N_V
5. Find current cost
   $Cost_C = Cost_C + Cost_{i,j}$
6. Loop to step 4
7. If $Cost_C < Cost_M$

Assign new minimum tasks and cost

$Tasks_M = Tasks_C$, $Cost_M = Cost_C$

8. Go back to step 2

The final optimized, flyable paths are shown in Figure 3.2.4.

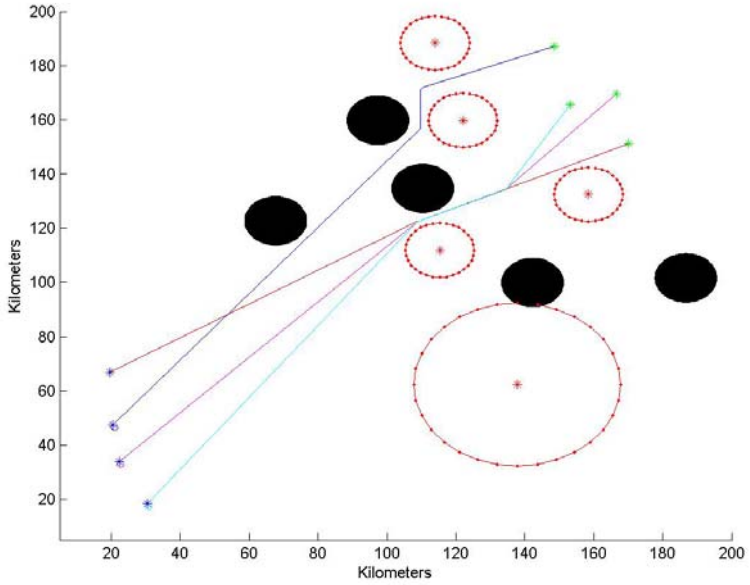


Figure 3.2.4 – Allocated tasks for each UAV to visit each target

A function written in MATLAB to perform this called "*mmkp_task_allocation*" is located in Appendix A. The inputs of this function are the matrices '*totalcost*', '*Shortened_Paths_x*', '*Shortened_Paths_y*'. This function finds the solution to the MMKP algorithm as stated above and returns the matrices '*Selected_Paths_x*' and '*Selected_Paths_y*'. These matrices contain the x and y locations of an optimized and flyable path for each UAV. These paths are designed such that the mission completion time is minimized and the probability of mission completion is maximized.

Chapter 4

Implementation of Six Degree of Freedom Aircraft Dynamics

4.1 - General Overview of Aircraft Dynamics

This section will review the dynamics of an aircraft, including a brief overview of aircraft forces, moments, equations of motion, and state variable modeling of the aircraft dynamics. To properly define the forces, moments, and the equations of motion that are associated with an aircraft, a non-rotating earth fixed axis system must be chosen as an initial point of reference. To derive these equations of motion the following assumptions must be made:

- The aircraft is a rigid body
- The earth is an inertial reference frame
- The aircraft mass and mass distributions are constant with respect to time
- The XZ plane is a plane of symmetry for the aircraft
- There are negligible gyroscopic effects from the engine
- The equations of motion are derived with respect to the stability axes
- There are only small perturbations
- There are only three primary control surfaces
    - Elevators
    - Ailerons
    - Rudder

The equations of motion of an aircraft come directly from Newton's second law with respect to the conservation of linear and angular momentum[28]. In order for these

equations to be derived they must relate the forces and moments associated with the aircraft to the dynamics and movement of the aircraft. The forces acting on an aircraft are modeled as F_{AX}, F_{AY}, and F_{AZ}, which can be seen in Figure 4.1.1[28].

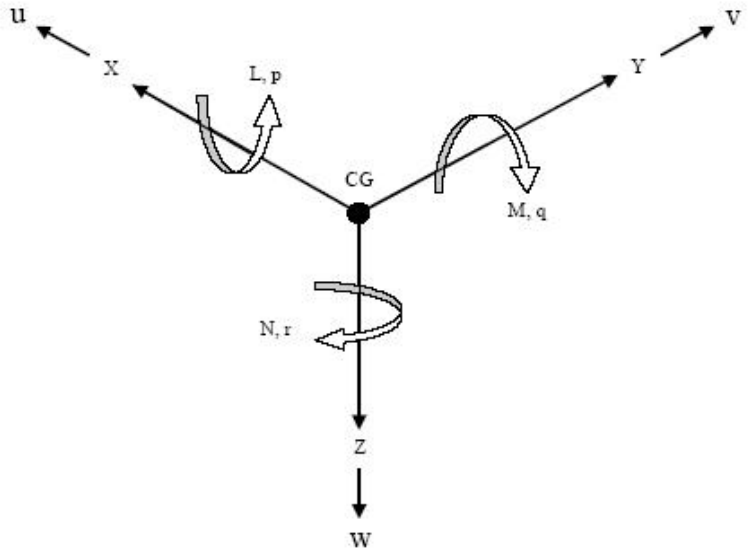


Figure 4.1.1 – Aircraft body axis forces and moments

This figure also shows the moments that act on an aircraft that are L_A, M_A, and N_A. It should be noted, all of these forces and moments are with respect to the body axis of the aircraft. Including the forces from thrust, applying Newton's second law with the conservation of linear momentum on an aircraft leads to equations 4.1.1 through 4.1.3[29].

$$m(\dot{U} - VR + WQ) = mg_x + F_{AX} + F_{TX} \tag{4.1.1}$$

$$m(\dot{V} - UR + WP) = mg_y + F_{AY} + F_{TY} \tag{4.1.2}$$

$$m(\dot{W} - UQ + VP) = mg_z + F_{AZ} + F_{TZ} \tag{4.1.3}$$

where m is the mass of the aircraft, U is the velocity in the x direction, V is the velocity in the y direction, and W is the velocity in the z direction. P, Q, and R are the angular

velocities with respect to the x, y, and z axes respectively. Also, g_x, g_y, and g_z are the components of gravity in the x, y, and z directions.

In equations 4.1.4 through 4.1.6 Newton's second law has been applied with the conservation of angular momentum, the moments from thrust have been included[29].

$$I_{XX}\dot{P} - I_{XZ}\dot{R} - I_{XZ}PQ + (I_{ZZ} - I_{YY})RQ = L_A + L_T \tag{4.1.4}$$

$$I_{YY}\dot{Q} + (I_{XX} - I_{ZZ})PR + I_{XZ}(P^2 - R^2) = M_A + M_T \tag{4.1.5}$$

$$I_{ZZ}\dot{R} - I_{XZ}\dot{P} + (I_{YY} - I_{XX})PQ + I_{XZ}Qr = N_A + N_T \tag{4.1.6}$$

I_{XX} , I_{YY} , and I_{ZZ} are the moments of inertia about of the x, y, and z axes. I_{XY} , I_{YZ} , and I_{XZ} are the products of inertia about the x, y, and z axes. The above equations form a non-linear system of equations that can be solved in terms of *U*, *V*, *W*, *P*, *Q*, and *R*. These equations are taken with respect to the body axis of the aircraft. In order to solve these equations they must be described according to a non-rotating earth fixed axis. This is accomplished through the use of Euler angles, φ , θ, and ϕ. The translation from the body axis to the earth axis can be done by using the following steps[29].

1. Consider the earth axis translated parallel to itself so that the origin coincides with the origin of the body axis of the aircraft or the CG.
2. Change the name the earth axis X`Y`Z` to $X_1Y_1Z_1$.
3. The axis system $X_1Y_1Z_1$ is rotated about Z_1 by the Euler angle φ to reach the axis system $X_2Y_2Z_2$.
4. The axis system $X_2Y_2Z_2$ is rotated about Y_2 by the Euler angle θ to reach the axis system $X_3Y_3Z_3$.
5. The axis system $X_3Y_3Z_3$ is rotated about X_3 by the Euler angle ϕ to reach the original axis system XYZ.

An illustration of this is shown in Figure 4.1.2[29].

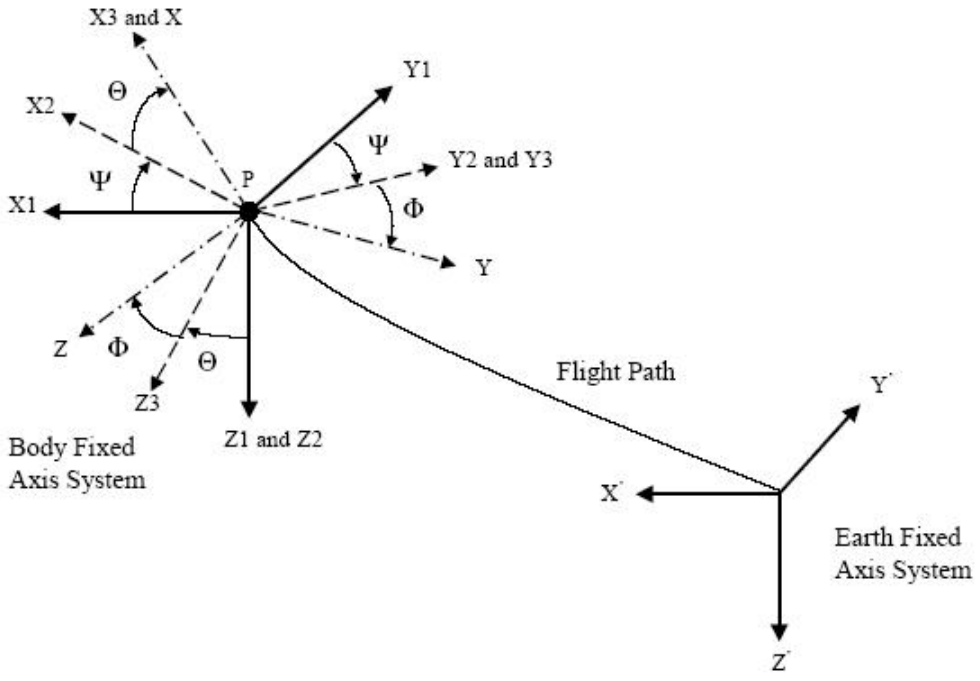


Figure 4.1.2 – Translation from the earth axis to the body axis

As shown the Euler angle φ is referred to as the heading angle, θ is the pitch angle, and ϕ is the bank angle of the aircraft. Using these angles, an aircraft's flight path can be described in terms of the earth and body axis velocities.

$$\begin{bmatrix} U_1 \\ V_1 \\ W_1 \end{bmatrix} = \begin{bmatrix} \cos\varphi & -\sin\varphi & 0 \\ \sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi \\ 0 & \sin\phi & \cos\phi \end{bmatrix} \begin{bmatrix} U \\ V \\ W \end{bmatrix} \tag{4.1.7}$$

In a similar fashion the angular velocities of the body axis can be expressed in terms of the Euler angles.

$$\begin{bmatrix} p \\ q \\ r \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\phi & \sin\phi\cos\theta \\ 0 & -\sin\phi & \cos\phi\cos\theta \end{bmatrix} \begin{bmatrix} \dot{\phi} \\ \dot{\theta} \\ \dot{\varphi} \end{bmatrix} \tag{4.1.8}$$

These equations are known as the kinematic equations, which yield the following expressions:

$$p = \dot{\phi} - \dot{\varphi}\sin\theta \tag{4.1.9}$$

$$q = \dot{\theta}\cos\theta + \dot{\varphi}\cos\theta\sin\phi \tag{4.1.10}$$

$$r = \dot{\varphi}\cos\theta\cos\phi - \dot{\theta}\sin\phi \tag{4.1.11}$$

This set of equations coupled with equations 4.1.1 through 4.1.3 and 4.1.4 through 4.1.6 are the equations of motion for an aircraft. Due to the fact that most of these values cannot be directly measured from the aircraft, they must be transferred into the polar coordinates α, β, and V, which are the angle of attack, sideslip angle, and aircraft velocity respectively. This conversion is shown in Figure 4.1.3[28].

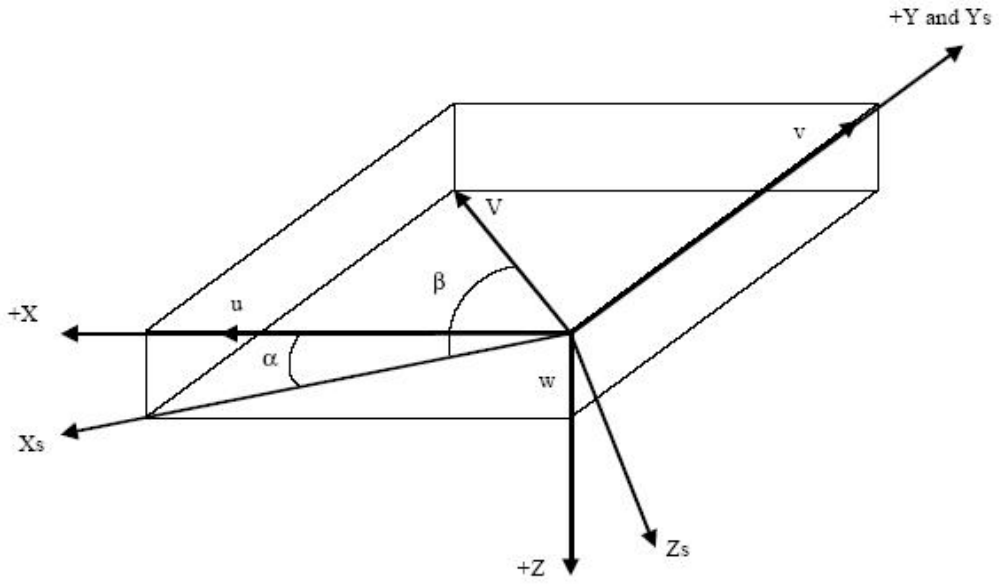


Figure 4.1.3 – Polar axis transformation for equations of motion

Using the aerodynamic coefficients the equations of motion then become the following:

$$\dot{V} = \frac{\bar{q}S}{m} C_{D_w} + g(\cos\phi\cos\theta\sin\alpha\cos\beta + \sin\phi\cos\theta\sin\beta - \sin\theta\cos\alpha\cos\beta) \tag{4.1.12}$$

$$\dot{\alpha} = q - \tan\beta(p\cos\alpha + r\sin\alpha) - \frac{\bar{q}S}{mV\cos\beta} C_L + \frac{g}{V\cos\beta}(\cos\theta\cos\phi\cos\alpha + \sin\theta\sin\alpha) \tag{4.1.13}$$

$$\dot{\beta} = p\sin\alpha - r\cos\alpha + \frac{\bar{q}S}{mV} C_{Y_w} + \frac{g}{V}\cos\beta\cos\theta\sin\phi - \frac{g}{V}\sin\beta(\cos\theta\cos\phi\sin\alpha - \sin\theta\cos\alpha) \tag{4.1.14}$$

$$\dot{p} = \frac{1}{I_{XX}}[I_{XY}(\dot{q} - pr) + I_{XZ}(\dot{r} + pq) + qr(I_{YY} - I_{ZZ}) + I_{YZ}(q^2 - r^2) + \bar{q}SbC_l] \tag{4.1.15}$$

$$\dot{q} = \frac{1}{I_{YY}}[I_{XY}(\dot{p} + qr) + I_{YZ}(\dot{r} - pq) + rp(I_{ZZ} - I_{XX}) + I_{XZ}(r^2 - p^2) + \bar{q}S\bar{c}C_m] \tag{4.1.16}$$

$$\dot{r} = \frac{1}{I_{ZZ}}[I_{XZ}(\dot{p} - qr) + I_{YZ}(\dot{q} + pr) + pq(I_{XX} - I_{YY}) + I_{XY}(p^2 - q^2) + \bar{q}SbC_n] \tag{4.1.17}$$

From these equations a state variable model of the longitudinal and lateral directional dynamics of the aircraft can be built. These models are of the form $\dot{x} = Ax + Bu$ and $y = Cx + Du$. The longitudinal dynamics these can be seen in the following equations[30]:

$$\dot{x}_{Long} = A_{Long}x_{Long} + B_{Long}u_{Long} \tag{4.1.18}$$

$$y_{Long} = C_{Long}x_{Long} + D_{Long}u_{Long} \tag{4.1.19}$$

where x_{Long} is a vector containing the states of the system α, u, q, and θ; u_{Long} is control input δ_E; and y_{Long} is a vector containing the outputs of the system a_Z, α, u, q, and θ. Substituting the state vectors and matrices yields the following:

$$\begin{bmatrix} \dot{\alpha} \\ \dot{u} \\ \dot{q} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} Z'_{\alpha} & Z'_{u} & Z'_{q} & Z'_{\theta} \\ X'_{\alpha} & X'_{u} & 0 & X'_{\theta} \\ M'_{\alpha} & M'_{u} & M'_{q} & M'_{\theta} \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} \alpha \\ u \\ q \\ \theta \end{bmatrix} + \begin{bmatrix} Z'_{\delta_E} \\ X'_{\delta_E} \\ M'_{\delta_E} \\ 0 \end{bmatrix} \begin{bmatrix} \delta_E \end{bmatrix} \tag{4.1.20}$$

$$\begin{bmatrix} a_Z \\ \alpha \\ u \\ q \\ \theta \end{bmatrix} = \begin{bmatrix} Z''_{\alpha} & Z''_{\alpha} & Z''_{\alpha} & Z''_{\alpha} \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \alpha \\ u \\ q \\ \theta \end{bmatrix} + \begin{bmatrix} Z''_{\delta_E} \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \begin{bmatrix} \delta_E \end{bmatrix} \tag{4.1.21}$$

All of the longitudinal dimensional stability derivatives used in the above equations are shown in Appendix B. Similarly the lateral directional dynamics are expressed as such:

$$\dot{x}_{LatDir} = A_{LatDir} x_{LatDir} + B_{LatDir} u_{LatDir} \tag{4.1.22}$$

$$y_{LatDir} = C_{LatDir} x_{LatDir} + D_{LatDir} u_{LatDir} \tag{4.1.23}$$

where x_{LatDir} is a vector containing the states of the system β, p, r, and ϕ; u_{LatDir} is a vector containing the control inputs δ_A and δ_R; and y_{LatDir} is a vector containing the outputs of the system a_Y, β, p, r, and ϕ. Substituting the lateral directional state vectors and matrices yields the following equations:

$$\begin{bmatrix} \dot{\beta} \\ \dot{p} \\ \dot{r} \\ \dot{\phi} \end{bmatrix} = \begin{bmatrix} Y_{\beta}^{'} & Y_{p}^{'} & Y_{r}^{'} & Y_{\phi}^{'} \\ L_{\beta}^{'} & L_{p}^{'} & L_{r}^{'} & 0 \\ N_{\beta}^{'} & N_{p}^{'} & N_{r}^{'} & 0 \\ 0 & 1 & \tan\theta & 0 \end{bmatrix} \begin{bmatrix} \beta \\ p \\ r \\ \phi \end{bmatrix} + \begin{bmatrix} Y_{\delta_A}^{'} & Y_{\delta_R}^{'} \\ L_{\delta_A}^{'} & L_{\delta_R}^{'} \\ N_{\delta_A}^{'} & N_{\delta_R}^{'} \\ 0 & 0 \end{bmatrix} \begin{bmatrix} \delta_A \\ \delta_R \end{bmatrix} \tag{4.1.24}$$

$$\begin{bmatrix} a_Y \\ \beta \\ p \\ r \\ \phi \end{bmatrix} = \begin{bmatrix} Y_{\beta}^{''} & Y_{p}^{''} & Y_{r}^{''} & Y_{\phi}^{''} \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \beta \\ p \\ r \\ \phi \end{bmatrix} + \begin{bmatrix} Y_{\delta_A}^{''} & Y_{\delta_R}^{''} \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} \delta_A \\ \delta_R \end{bmatrix} \tag{4.1.25}$$

The lateral directional dimensional stability derivatives used in these equations can be seen in Appendix B. From this point various control schemes can be designed to control the flight characteristics of an aircraft. A longitudinal control system can be implemented to control the pitching rate, pitch angle, airspeed, and altitude of the aircraft. Whereas a lateral directional control scheme has the ability to control the yaw rate, roll rate, bank angle, and the heading angle. For the purposes of this research a heading angle controller must be designed so the UAV can follow its assigned path.

4.2 - Implementation of Heading Angle Control Scheme

After the state equations have been derived to govern the heading angle of the aircraft, a control scheme must be designed to control it. This must be done within the SIMULINK environment in MATLAB. A simulation designed by Rauw named the Beaver aircraft simulator, provides an excellent way to simulate the dynamics of any general aviation aircraft[31]. This is due to the fact that the user can enter any desired aerodynamic coefficients with a user interface that can be seen in Figure 4.2.1. This also allows the user to enter the initial conditions, mass, and other geometric data of the aircraft.

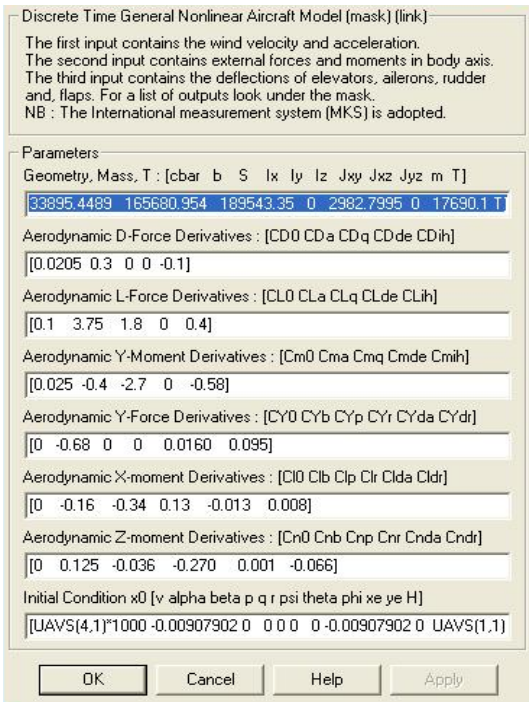


Figure 4.2.1 – Data entry user interface

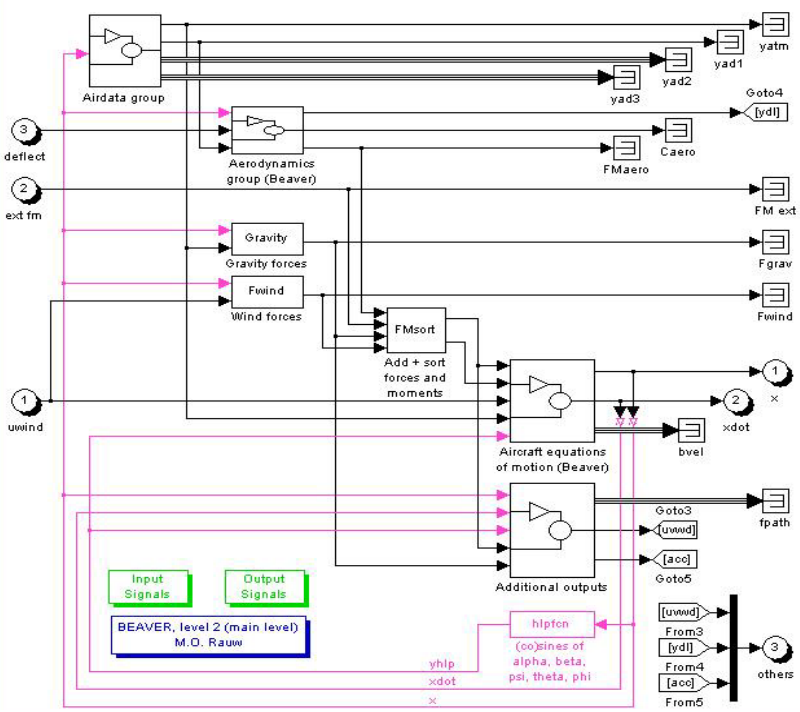


Figure 4.2.2 – The aircraft simulator control system

The SIMILINK block shown in Figure 4.2.2 simulates the longitudinal and lateral directional dynamics of an aircraft. This system inputs the control surface deflections of the elevators, rudder, and ailerons, and returns the current states of the aircraft. To give the UAV the ability to follow its given path, a control scheme must be designed to govern the control surface deflections, which is shown in Figure 4.2.3.

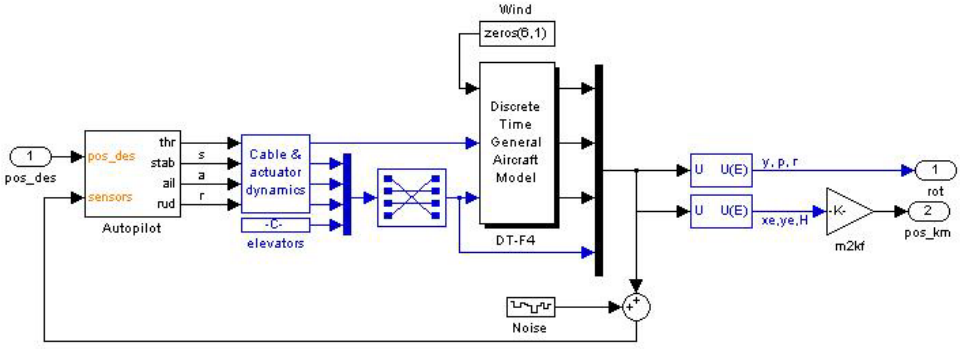


Figure 4.2.3 – The heading angle control scheme

Inside this main scheme is an autopilot controller seen in Figure 4.2.4. The autopilot block inputs the current and desired x and y positions for the aircraft.

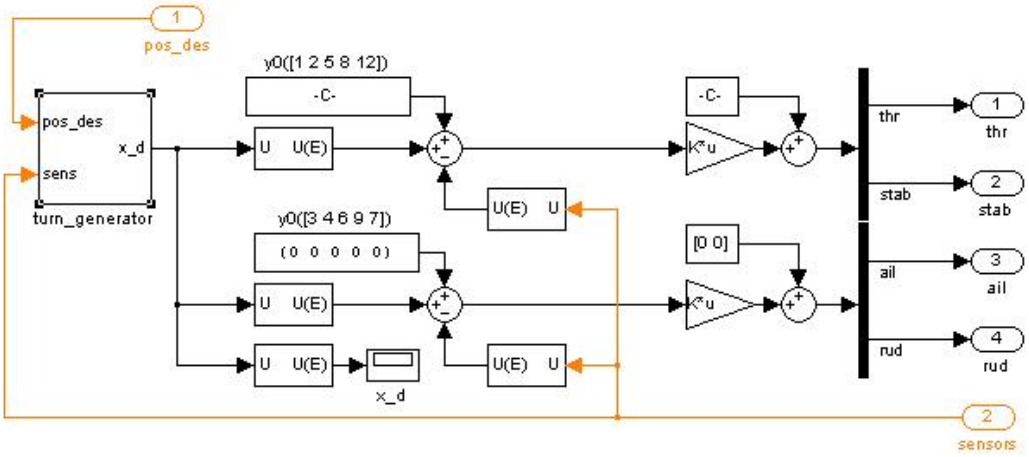


Figure 4.2.4 – The aircraft autopilot control block

This controller then uses a turn generator to follow the desired path by deflecting the proper control surfaces. The design for this is shown in Figure 4.2.5. This control scheme is an efficient and reliable way to navigate from one waypoint to the next.

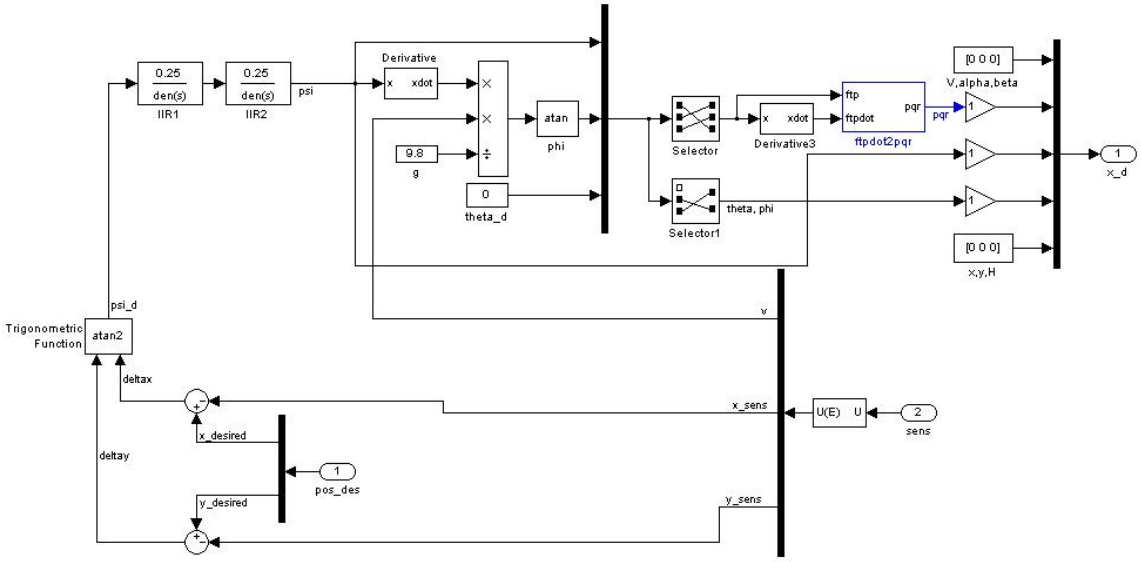


Figure 4.2.5 – The autopilot turn generator block

The beaver aircraft simulator outputs the state vector of the aircraft which contains the x location, y location, z location, velocity, angle of attack, sideslip angle, pitch rate, yaw rate, roll rate, pitch angle, bank angle, and the heading angle of the aircraft. These states of the aircraft, along with the desired x and y positions, are feedback into the aircraft dynamics forming the closed loop control design. In this research effort, since no actual UAV dynamics were available, F-4 dynamics were chosen due to their benign nature. The aerodynamics coefficients for an F-4 at subsonic cruise used are available in Roskam[29].

Chapter 5

Development of a SIMULINK scheme for Cooperating UAVs

5.1 - Implementation of the Path-Planning Process and Aircraft Dynamics

The SIMULINK environment not only provides an excellent way of executing MATLAB files, but it is advantageous in examining the inputs and outputs of a simulation. In addition, it provides several different ways to visualize the results of a simulation. This section will cover the implementation of the path-planning functions discussed in Chapter 3 and the heading angle control scheme discussed in Chapter 4. A SIMULINK file is constructed using a block diagram where each block has an input and an output. Each block contains code that is executed based on its inputs and returns an output, which is then sent to another block. This process is repeated to form a simulation. This can be seen in Figure 5.1.1, the main SIMULINK file for this simulation.

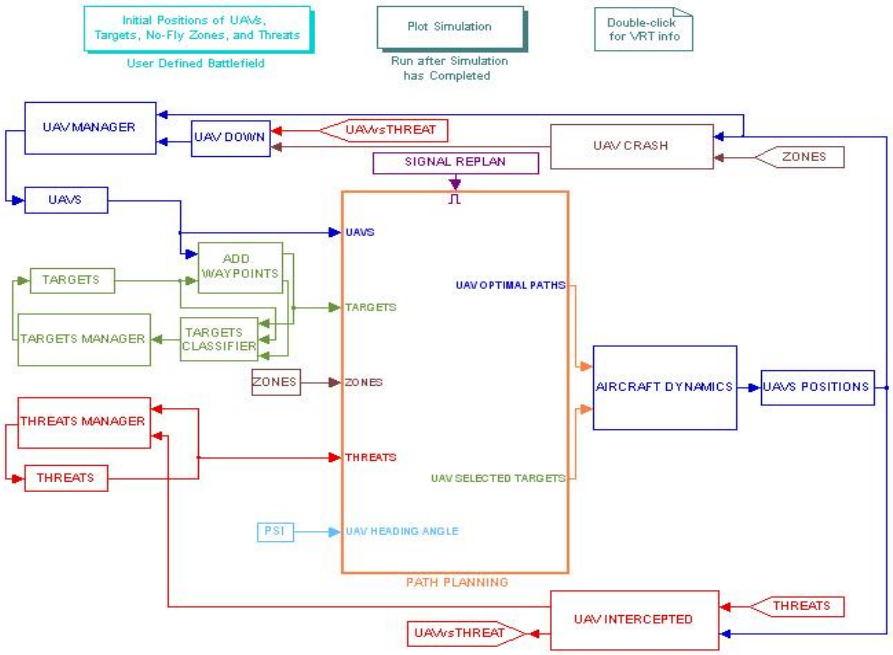


Figure 5.1.1 – Main block diagram for cooperating UAVs

The central block in this diagram labeled *"PATH PLANNING"* contains the MATLAB code discussed in Chapter 3. This code is implemented using an S-function, which stands for SIMULINK function. This function allows for the specification of the number of inputs and outputs to a block. Each S-function contains executable code. Figure 5.1.2 shows the S-function *"path_planning_s"* being used, which can also be seen in Appendix C.

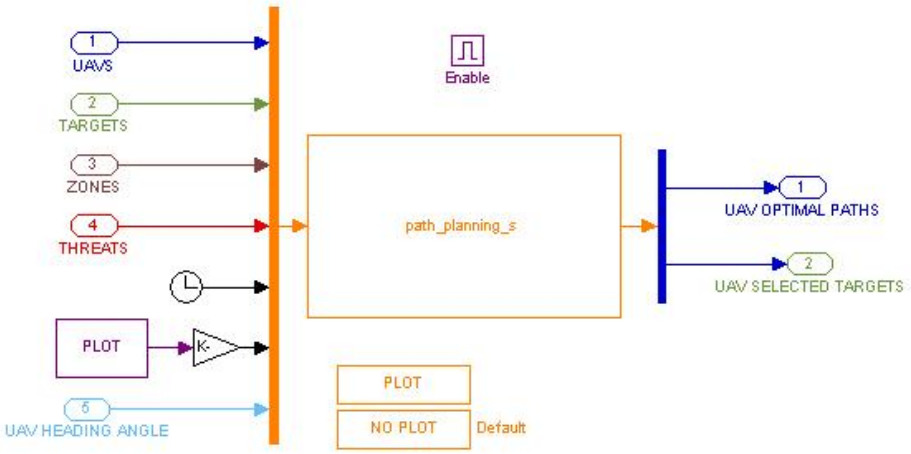


Figure 5.1.2 – Path planning s-function implementation

Since the S-function requires that the input and output be single vectors as opposed to matrices, the inputs are reshaped and combined into a vector of a fixed size using a multiplexer. A multiplexer combines several vectors and scalars into a single vector. In this case, the vectors *'UAVS'*, *'TARGETS'*, *'ZONES'*, *'THREATS'*, *'UAV_HEADING_ANGLE'* and the scalars current time of simulation and current plot number are all combined into a single vector. Also, inside this block the user can control if the current conditions of the battlefield are plotted when a replan occurs. Inside the S-function *"path_planning_s"* another function is called, *"path_planning"*, which can be

found in Appendix C. This function contains several reshape functions that transform the inputs into the desired matrix shape to execute the functions defined in Chapter 3.

As previously stated, this code yields an optimized, flyable path for each UAV to follow. In order for the heading angle control scheme to operate it must be given a smooth path instead of the locations of the waypoints. To accomplish this each waypoint must be assigned a time at which the UAV should be visiting it. This is estimated using the constant velocity of the aircraft. Along with the assigned waypoints, the selected targets that each UAV is assigned to visit is also output. This is done so that the targets are classified properly. After the waypoints and there associated times are output from the path-planning S-function they are then sent into a look-up table block, which is shown in Figure 5.1.3. This uses linear interpolation to provide a smooth path for the autopilot discussed in Chapter 4 to follow.

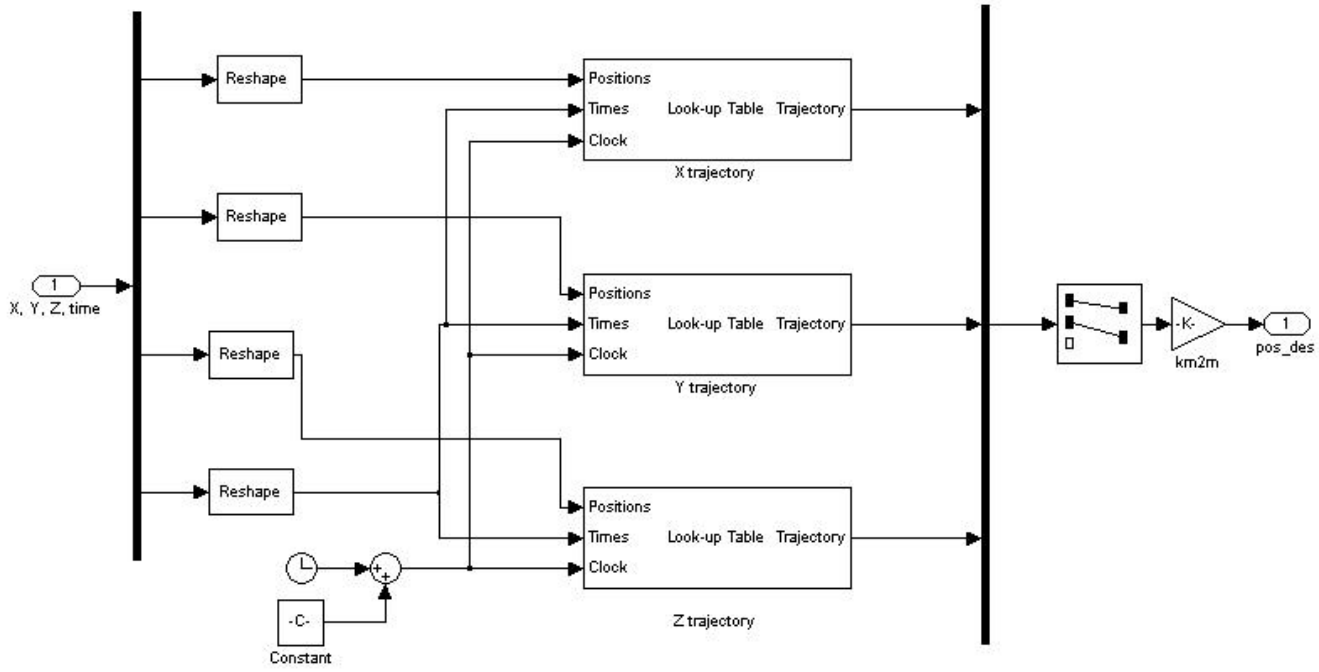


Figure 5.1.3 – Look-up table SIMULINK block

5.2 - Management of the No-Fly Zones and Threats

In a realistic battlefield environment, the UAVs must have the ability react to what is happening around them. This can include a threat popping up, vehicle entering a no-fly zone's radius, or a vehicle entering threat's range. If a UAV flies inside a threat's range the threat will fire and based on the probability of kill of the threat that vehicle may be destroyed. Also, if a UAV flies inside a no-fly zone's radius it is assumed that the aircraft is lost. In order to simulate a vehicle interacting with a threat or no-fly zone several S-functions were written, which are "*uav_crash_s*" and "*uav_intercepted_s*". These functions compare the current positions of the UAVs with the position and radius of each no-fly zone and the position and range of each threat. These functions can be seen in Appendix C.

The outputs of these two functions are vectors containing either zeros or ones. The value is a zero if the UAV is still operational or one if the UAV has been destroyed. The implementation of the "*uav_crash_s*" function is shown in Figure 5.2.1, while the "*uav_intercepted_s*" function can be seen in Figure 5.2.2.

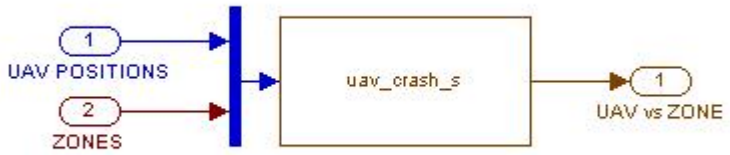


Figure 5.2.1 – Block comparing UAV positions to no-fly zone positions

Figure 5.2.2 – Block comparing UAV positions to threat positions

In addition to comparing the current locations of the UAVs to the threats and their associated ranges, a random number is generated when a vehicle passes inside a threat's range. If this number is within the specifications for the probability of kill of that threat the vehicle is destroyed. Otherwise, the vehicle remains operational and continues on its current path. Either way when a threat has fired, it is no longer present on the battlefield and will have no further effect on any UAV. If a threat has fired or a vehicle is destroyed, a replan is signaled for the entire group based on the battlefield changing.

Another component of a dynamic battlefield environment is a pop-up threat. This is a threat that is unknown for the initial plan, but is discovered during the simulation. The simulation of this occurrence is important because a realistic battlefield will never remain constant. A block was created for the purpose of simulating a pop-up threat, which is shown in Figure 5.2.3.

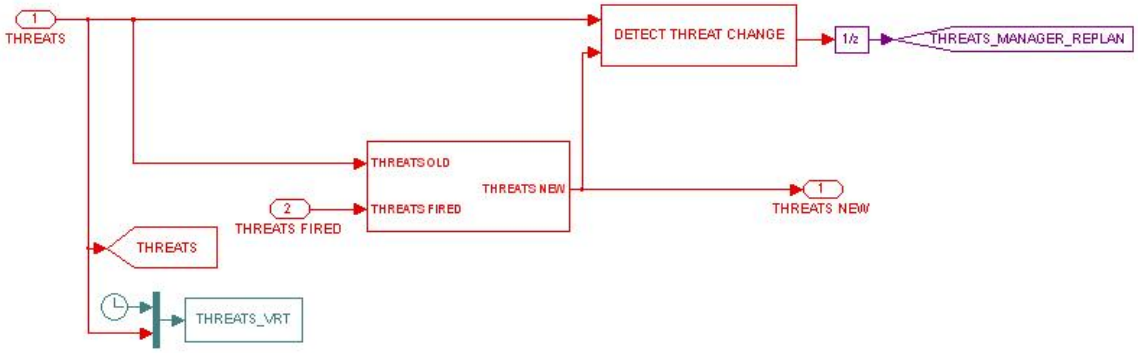


Figure 5.2.3 – Threats manager

These blocks compare the old values of the '*THREATS*' vector to the new values of the vector. If a change occurs a replan is signaled for the group according to the new information.

5.3 - Management of the UAVs and Targets

The dynamics of the battlefield extend to the UAVs and targets as well as the threats and no-fly zones. A simulation for cooperating UAVs must have the ability to simulate a vehicle being destroyed or a target changing states, i.e. classified, destroyed, or assessed. These are extremely important when creating a realistic simulation of cooperating UAVs. Whether a UAV is operational or not is controlled by the SIMULINK block shown in Figure 5.3.1.

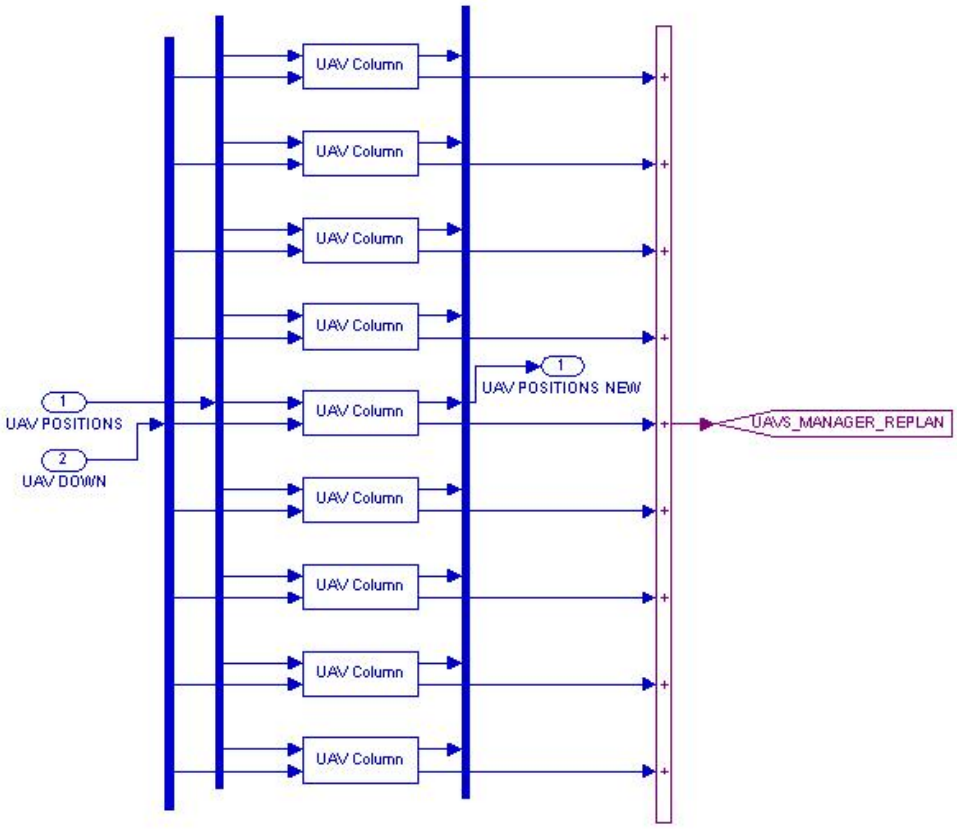


Figure 5.3.1 – UAVs manager

These blocks input the vectors from the previous section based if a vehicle entered a threat's range or no-fly zone's radius. If these values are all zero then no vehicle is destroyed, but if a vehicle is destroyed a replan is signaled. Also, each vehicle has a limited amount of fuel, therefore if a vehicle's fuel runs out that vehicle is considered lost and a replan is signaled.

In any battlefield, each target must be acted upon by several UAVs. The states that a target can have are the following:

- Identified
- Classified as a valid / invalid target
- Attacked
- Assessed as destroyed / not destroyed

Each target must have all of these actions performed on it, except when a target is classified as an invalid target. This is implemented with the code contained in the SIMULINK block shown in Figure 5.3.2. Inside this block is an S-function named "*target_classifier_s"*, which can be seen in Appendix C. This calls the function "*target_classifier"* that changes the state of a target based on if it is visited.

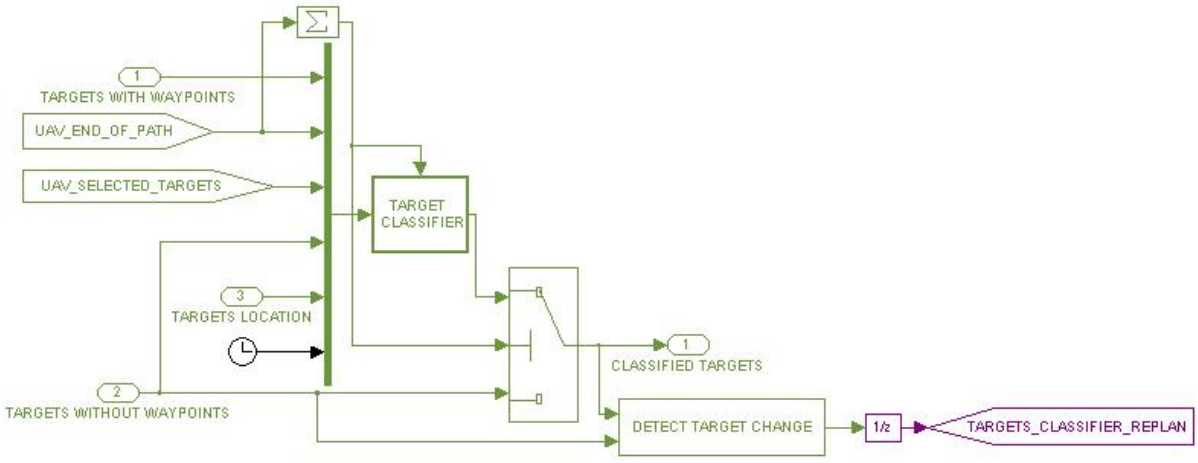


Figure 5.3.2 – Targets classifier SIMULINK block

Every target is initialized to the state of identified not classified. After a target is visited for the first time, a random number is generated. Based on this number a target is either classified as a valid target or classified as an invalid target. An invalid target is immediately deleted and no further action is required. After a target has been classified as a real target it must be attacked. In order to ensure that the desired target has been destroyed, a battle damage assessment (BDA) must be preformed. If the BDA reveals that the target has not been destroyed the target must be attacked again. This process is repeated until the target has been assessed as destroyed. For the purpose of this simulation a random number is generated between 0 and 1. If that number is less than 0.85 the BDA is deemed successful and the target is deleted.

In much the same way as a threat can be discovered during the simulation a target can pop-up while the UAVs are acting on the current targets. The SIMULINK block in Figure 5.3.3 has the ability to simulate this occurrence.

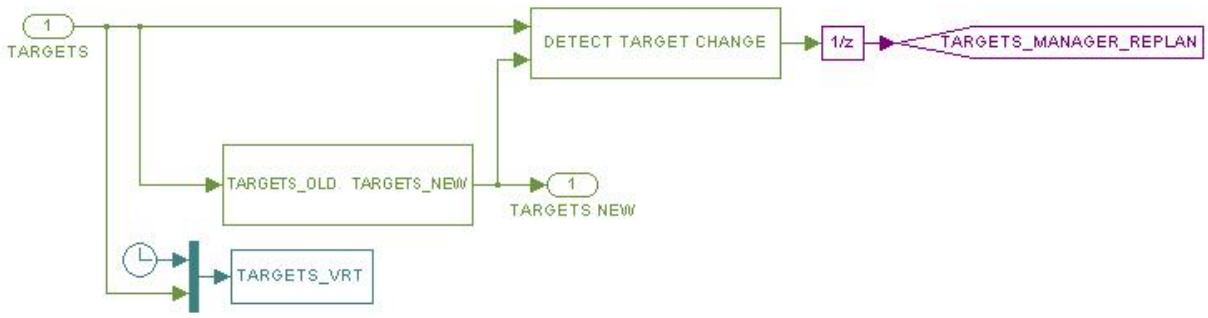


Figure 5.3.3 – Targets manager

If this happens a replan for the group of UAVs is signaled. This block compares the old values of the '*TARGETS*' vector to the new value and detects a change.

Due to the nature of the MMKP algorithm outlined in Chapter 3 the number of UAVs must be equal to the number of targets. The code contained in the block shown in Figure 5.3.4 calls an S-function named "*place_waypoints_s*", which is shown in Appendix C.

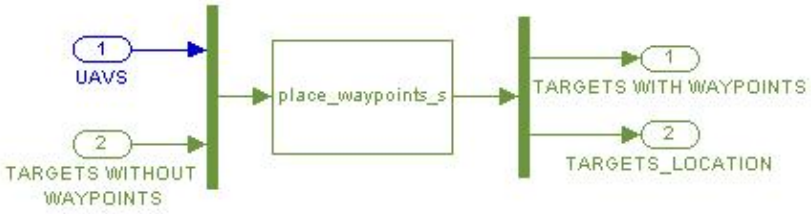


Figure 5.3.4 – Add waypoints SIMULINK block

This function calls a function that alters the '*TARGETS*' vector if needed. If the number of UAVs exceeds the number of targets, waypoints are placed at the most valuable targets. This is done to ensure that these targets will be visited the most. Otherwise, if the number of UAVs is less than the number of targets, the least valuable targets are temporarily deleted until all of the valuable targets have been serviced. After every dynamic reaction a replan for the group of UAVs is signaled, which can be seen in figure 5.3.5. This shows the SIMULINK block that gathers all of the replan signals and activates the central path-planning algorithm.

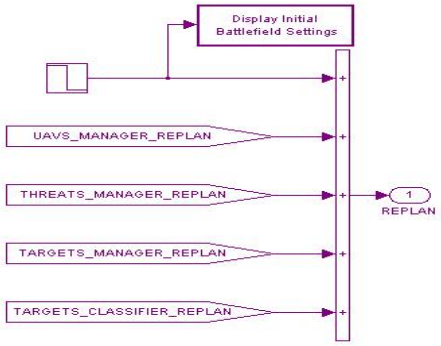


Figure 5.3.5 – Signal replan SIMULINK block

Chapter 6

Comparison with Other Available Path Generation Methods

6.1 - Implementation of Grid and Visibility Graph

In addition to the path generation technique presented in Chapter 3, several other methods have been used by previous researchers such as a grid[7] or a visibility graph[5,6,8]. These two methods provide an excellent comparison for evaluating the efficiency and calculation speed of the Voronoi diagram method. This is important to evaluate the level of optimization and computational complexity. A simulation is desired that not only has real-time application abilities, but also results in an optimal solution for the mission. This can be evaluated by using a grid or a visibility graph. A grid involves the overlaying of a grid on the battlefield. In a visibility graph every point on the battlefield is entered and lines are drawn between these points, if and only if there is a clear line of sight.

The overlaying of a grid onto the battlefield provides a simple comparison to the more complicated methods. This is accomplished using the MATLAB code seen in Appendix D. An example of this is shown in Figure 6.1.1. After the grid has been generated the same path-planning process is used that has been described in Chapter 3. The UAVs and targets positions are connected into the grid through the three closest nodes. Dijkstra's algorithm is then implemented to find the lowest cost path for each permutation of UAV to target. This method can provide different results because it does not take into account the locations of any no-fly zones or threats when the possible paths are generated. In the same fashion as before, the selected paths are refined into optimized flyable paths before the tasks are allocated using the MMKP algorithm.

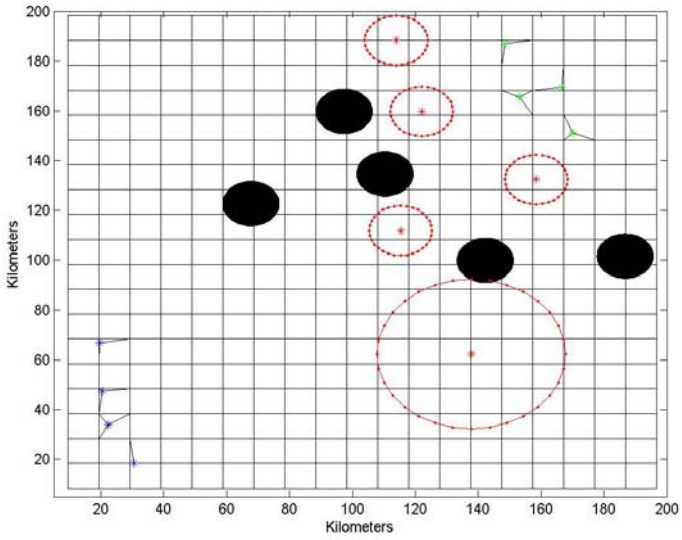


Figure 6.1.1 – Grid path generation

A visibility graph provides a completely different comparison than the previous two methods. There are several advantages and disadvantages with this method. The major disadvantage is the computational complexity that it brings to Dijkstra's algorithm. The MATLAB code written to implement this method can be seen in Appendix D. An example of a visibility graph is shown in Figure 6.1.2.

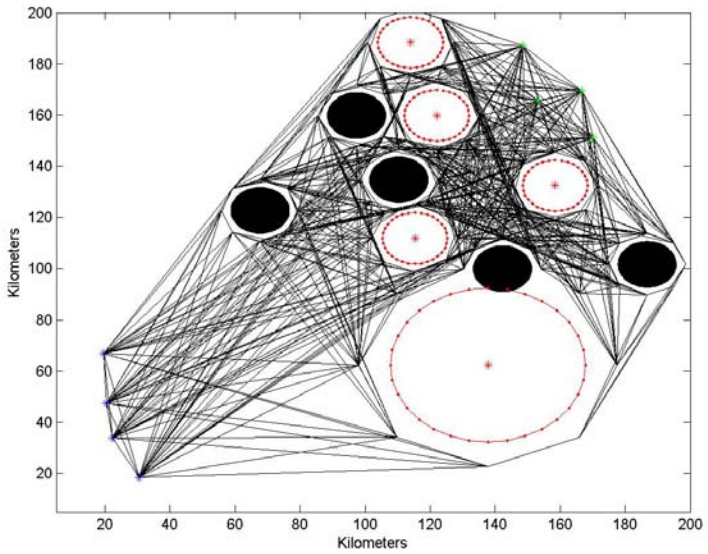


Figure 6.1.2 – Visibility graph path generation

It is apparent from the graph that the complexity greatly exceeds the other two possible path generation methods, which is a large hindrance on finding the lowest cost path for each UAV to each target. Also, it should be noted that a safety factor of 10% of each threat's range and no-fly zone's radius was used in creation of these paths. As opposed to the previous two methods, the UAV and target locations are included in the generation of these possible paths. In theory this approach should yield an already optimized solution. This is because it is an exhaustive search as opposed to approximate solutions.

Some of the advantages of a visibility graph are that is provides a more complete possible path solution. This leads to fewer calculations after Dijkstra's algorithm. Due to the fact that this path will be the shortest possible path, it will not have to be optimized during the refinement step, but these paths still need to be made flyable. These paths are defined according to points on the battlefield, which are the outer lying radii of the no-fly zones, ranges of the threats, positions of the UAVs, and positions of the targets. Since the radii and ranges are spherical the points must be placed at equal intervals along this sphere. This leads to the paths passing as close as possible to a threat or no-fly zone penetrating it.

The generation of the visibility lines in this graph is accomplished using Algorithm 3.1.3. After every point is generated, they are exhaustively searched to every other point to see if the line connecting the two points passes through a threat's range or no-fly zone's radius. If the line does not, it is recorded as a possible line of travel. Clearly, this process leads to the generation of paths that cannot be optimized.

6.2 - Comparison of the Path Generation Methods

To evaluate the original Voronoi based method for path generation it must be compared with the two methods discussed in the previous section. The comparison of these methods involves the evaluation of several factors.

- Calculation time of each replan
- Total estimated cost of each replan
- Simulation time at which each replan occurred
- Total Number of replans needed to complete the mission
- Total mission completion time (simulation and calculation)

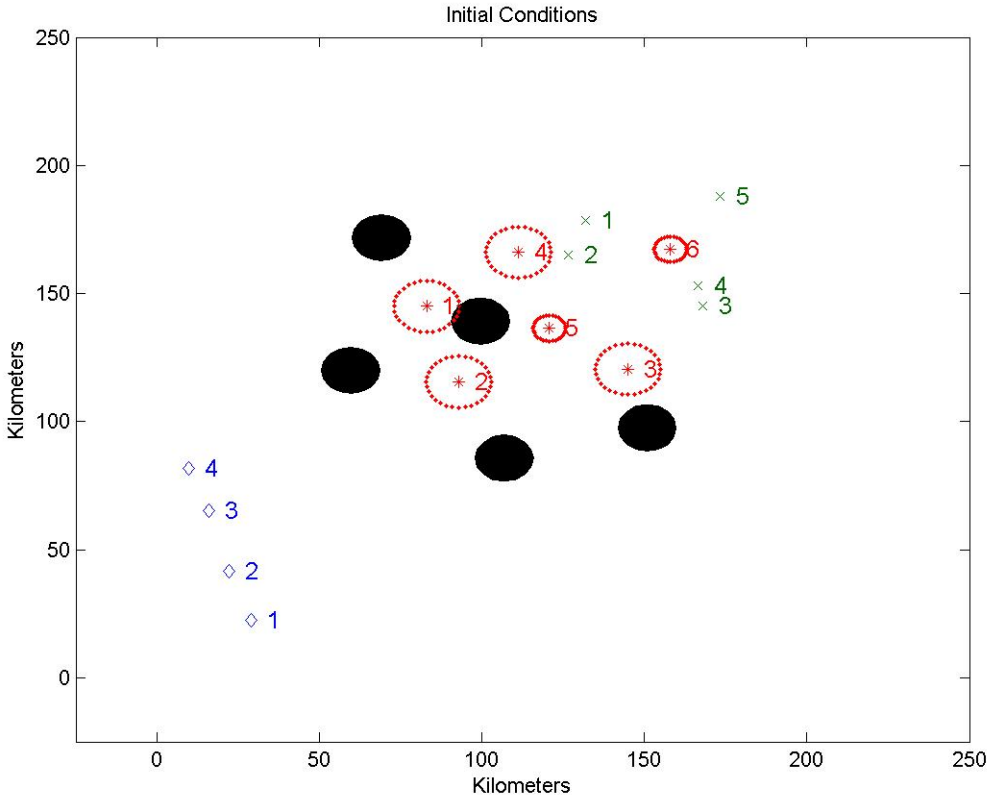
Each method was used with the same initial conditions of the battlefield. In addition, for comparison purposes, all of the random variables involved in classifying a target were removed. If these variables were left in place it would be difficult to draw any conclusions on which method is more effective. The comparison between these methods is shown in Table 6.2.1, which shows the total calculation time and total simulation time it took to complete the mission.

Table 6.2.1 – Comparison of total simulation time for possible path generation methods

Path Generation Method	Total Time (sec)	Simulaiton Time (sec)
Grid	180	1778
Voronoi Diagram	174	1918
Visibilty Graph	176	1715

It should be noted for the purpose of this comparison no targets were placed inside of a threat's range. Although the simulation is setup to allow this, it would have introduced randomness into the results, which is undesirable. To perform a fair and

unbiased comparison the initial conditions of the battlefield must be exactly the same, which are shown in Figure 6.2.1.




```
UAV 1 exists at location 29 x, location 22 y, altitude 2 km, and is flying at 130 m/s.
UAV 2 exists at location 22 x, location 42 y, altitude 2 km, and is flying at 130 m/s.
UAV 3 exists at location 16 x, location 65 y, altitude 2 km, and is flying at 130 m/s.
UAV 4 exists at location 10 x, location 82 y, altitude 2 km, and is flying at 130 m/s.
Target 1 indicated to be at location 132 x, location 179 y , and with an estimated value of 70.
Target 2 indicated to be at location 127 x, location 165 y , and with an estimated value of 80.
Target 3 indicated to be at location 168 x, location 145 y , and with an estimated value of 100.
Target 4 indicated to be at location 167 x, location 153 y , and with an estimated value of 90.
Target 5 indicated to be at location 173 x, location 188 y , and with an estimated value of 40.
No-Fly Zone 1 exists at location 69 x, location 172 y, and with a radius of 9 km.
No-Fly Zone 2 exists at location 100 x, location 139 y, and with a radius of 9 km.
No-Fly Zone 3 exists at location 151 x, location 98 y, and with a radius of 9 km.
No-Fly Zone 4 exists at location 107 x, location 86 y, and with a radius of 9 km.
No-Fly Zone 5 exists at location 60 x, location 120 y, and with a radius of 9 km.
Threat 1 exists at location 83 x, location 145 y, with a range of 10 km, and has a probability of kill of 80%.
Threat 2 exists at location 93 x, location 116 y, with a range of 10 km, and has a probability of kill of 80%.
Threat 3 exists at location 145 x, location 120 y, with a range of 10 km, and has a probability of kill of 80%.
Threat 4 exists at location 111 x, location 166 y, with a range of 10 km, and has a probability of kill of 80%.
Threat 5 exists at location 121 x, location 136 y, with a range of 5 km, and has a probability of kill of 50%.
Threat 6 exists at location 158 x, location 167 y, with a range of 5 km, and has a probability of kill of 50%.
```

Figure 6.2.1 – Initial conditions of the battlefield

In addition, all of the pop-up targets and threats were removed to ensure a fair comparison. These particular conditions were chosen because no UAV can travel directly to a target. This ensures that each path generation method is used instead of a

UAV traveling along a straight line to a target. Each UAV was given the same initial heading angle, cruise speed, and altitude, which are zero degrees, 130 meters per second, and two kilometers. The cruise speed and altitude of each UAV are held constant throughout the simulation.

To understand what occurred during the simulation every time a replan is signaled a figure is plotted showing the current positions of everything on the battlefield as well as the assigned paths for each UAV. In these the blue represents the UAVs positions and selected paths, the green points are the targets positions, the black circles are the no-fly zones and the red represents the threats positions and ranges.

Figures 6.2.2 -6.2.22 shows the figures plotted for the grid path generation, the Voronoi diagram path generation, and the visibility graph path generation from top to bottom. In addition to a figure being plotted the current action is printed to the MATLAB command line, so that a log of the simulation can be kept. These simulation logs can be seen for the grid, Voronoi diagram, and visibility graph path generation in Figures 6.2.23, 6.2.24, and 6.2.25 respectively. Table 6.2.2 shows the purpose of each replan, Table 6.2.3 shows what point in the simulation each replan is signaled. Table 6.2.4 contains the actual calculation time for each replan, while Table 6.2.5 show the assigned minimum cost for the current mission. This information was recorded to provide more in-depth comparison between the three methods.

Table 6.2.2 – Current actions for path generation methods

	Grid	Voronoi Diagram	Visibility Graph
Replan	Current Action	Current Action	Current Action
1	Initial Plan	Initial Plan	Initial Plan
2	Target 2 identified by UAV 4	Target 2 identified by UAV 4	Target 2 identified by UAV4
3	Target 2 classified by UAV 4	Target 4 identifed by UAV 3	Target 2 classified by UAV 4
4	Target 1 identified by UAV 3	Target 4 classifed by UAV 3	Target 2 attacked by UAV 4
5	Target 1 classified by UAV 3	Target 4 attacked by UAV 3	Target 2 assessed by UAV 4
6	Target 3 identified by UAV 3	Target 4 assessed by UAV 3	Target 3 identified by UAV 1
7	Target 3 classified by UAV 2	Target 3 identified by UAV 2	Target 3 classified by UAV 1
8	Target 4 identified by UAV 1	Target 3 classified by UAV 2	Target 4 identified by UAV 3
9	Target 4 classified by UAV 1	Target 3 attacked by UAV 2	Target 4 classified by UAV 3
10	Target 2 attacked by UAV 4	Target 3 assessed by UAV 2	Target 3 attacked by UAV 1
11	Target 4 attacked by UAV 1	Target 2 classified by UAV 4	Target 3 assessed by UAV 1
12	Target 4 assessed by UAV 1	Target 2 attacked by UAV 1	Target 4 attacked by UAV 3
13	Target 1 attacked by UAV 4	Target 2 assessed by UAV 1	Target 4 assessed by UAV 3
14	Target 3 attacked by UAV 2	Target 1 identified by UAV 4	Target 1 identified by UAV 4
15	Target 1 assessed by UAV 4	Target 1 classified by UAV 4	Target 5 identified by UAV 2
16	Target 2 assessed by UAV 3	Target 1 attacked by UAV 1	Target 5 classified by UAV 2
17	Target 3 assessed by UAV 2	Target 5 identified by UAV 2	Target 5 attacked by UAV 2
18	Target 5 identified by UAV 1	Target 5 classified by UAV 2	Target 1 classified by UAV 4
19	Target 5 classified by UAV 1	Target 5 attacked by UAV 3	Target 5 assessed by UAV 3
20	Target 5 attacked by UAV 1	Target 5 assessed by UAV 3	Target 1 attacked by UAV 4
21	Target 5 assessed by UAV 1	Target 1 assessed by UAV 1	Target 1 assessed by UAV 1

Table 6.2.3 – Time when replan is signaled for path generation methods

	Grid	Voronoi Diagram	Visibility Graph
Replan	Signaled (sec)	Signaled (sec)	Signaled (sec)
1	0	0	0
2	1308	1370	1188
3	1324	1379	1203
4	1354	1394	1250
5	1370	1439	1265
6	1384	1455	1353
7	1399	1479	1368
8	1425	1519	1388
9	1441	1562	1404
10	1459	1602	1412
11	1510	1647	1453
12	1525	1661	1467
13	1561	1706	1482
14	1573	1770	1537
15	1618	1786	1555
16	1626	1794	1571
17	1678	1806	1617
18	1703	1822	1628
19	1719	1841	1663
20	1763	1856	1705
21	1778	1918	1715

Table 6.2.4 – Actual replan calculation times for path generation methods

	Grid	Voronoi Diagram	Visibility Graph
Replan	Calculation (sec)	Calculation (sec)	Calculation (sec)
1	1.64	0.74	1.02
2	0.45	0.16	0.92
3	0.44	0.19	0.94
4	0.47	0.14	0.94
5	0.49	0.19	0.92
6	0.47	0.14	0.89
7	0.47	0.17	0.92
8	0.48	0.19	0.89
9	0.45	0.22	0.91
10	0.49	0.20	0.95
11	0.50	0.16	0.89
12	0.50	0.17	0.92
13	0.45	0.20	0.94
14	0.52	0.16	0.89
15	0.48	0.13	0.89
16	0.48	0.13	0.92
17	0.45	0.13	0.91
18	0.44	0.13	0.89
19	0.45	0.13	0.94
20	0.38	0.16	0.91
21	1.53	0.75	1.00

Table 6.2.5 – Replan current total cost for path generation methods

	Grid	Voronoi Diagram	Visibility Graph
Replan	Totalcost (m)	Totalcost (m)	Totalcost (m)
1	2968.30	2623.10	2266.30
2	57.23	48.04	199.40
3	53.51	55.71	202.11
4	26.04	78.14	92.23
5	33.60	31.63	146.79
6	33.62	61.39	61.00
7	41.09	93.59	69.02
8	24.48	63.48	51.62
9	26.23	48.19	47.65
10	72.32	118.42	64.86
11	33.66	83.59	49.77
12	68.71	86.81	49.91
13	47.51	91.15	180.55
14	62.22	29.24	136.57
15	86.03	25.05	127.41
16	140.27	25.15	79.28
17	155.51	31.89	35.40
18	140.31	38.50	44.53
19	191.50	35.25	113.20
20	100.05	128.07	80.06
21	2096.80	2207.80	2204.10

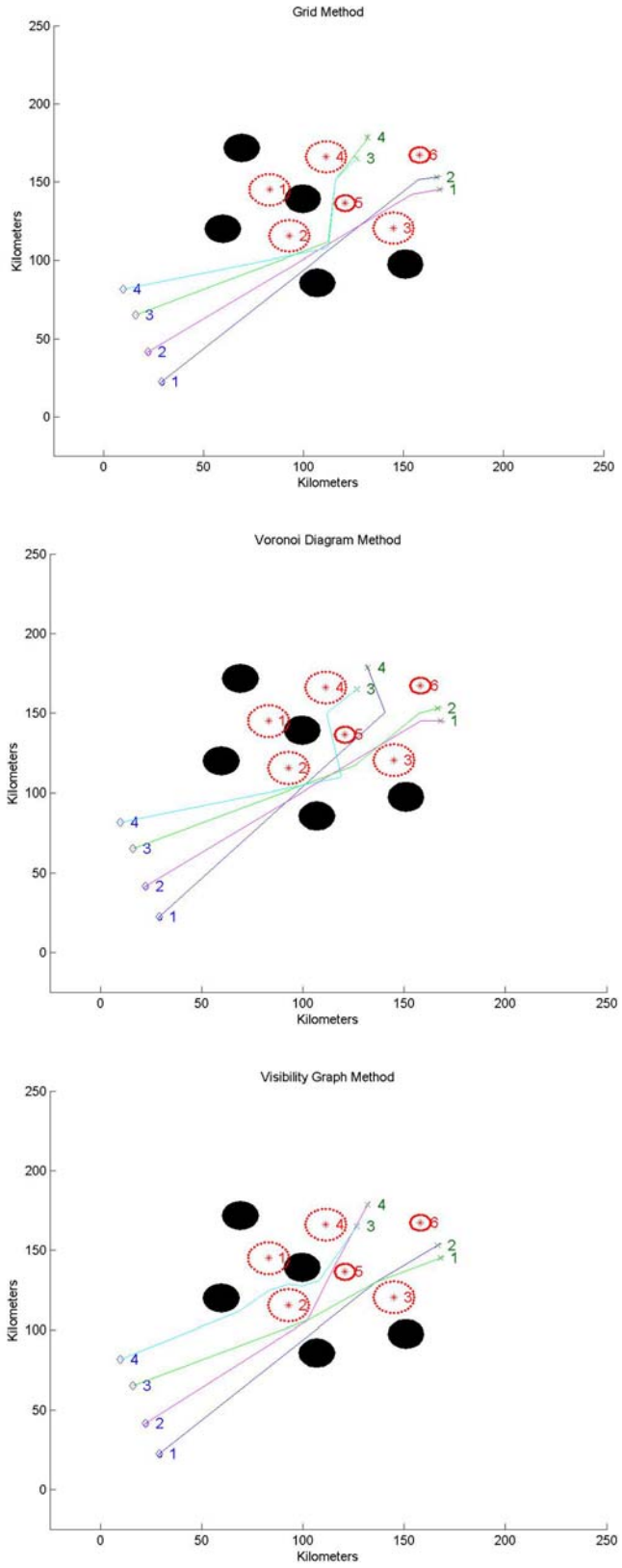


Figure 6.2.2 – 1st replan of the simulation for all three methods

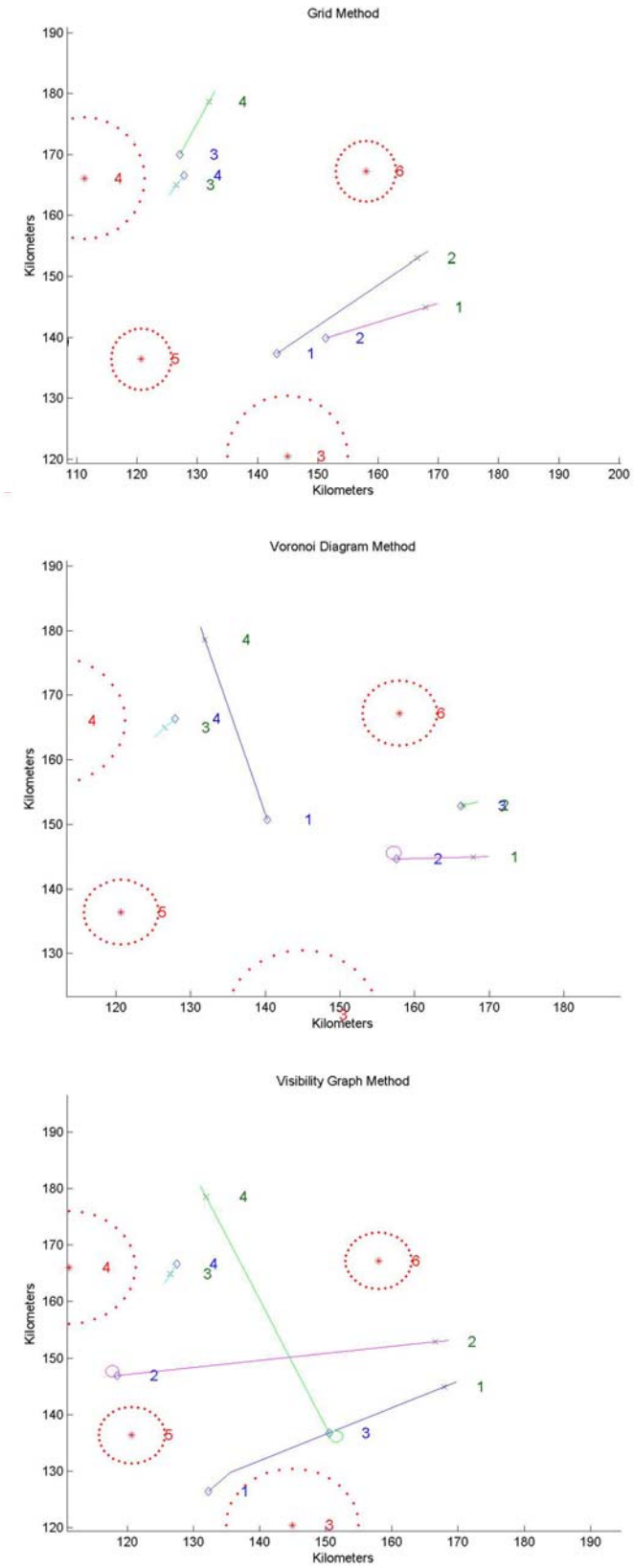


Figure 6.2.3 – 2nd replan of the simulation for all three methods

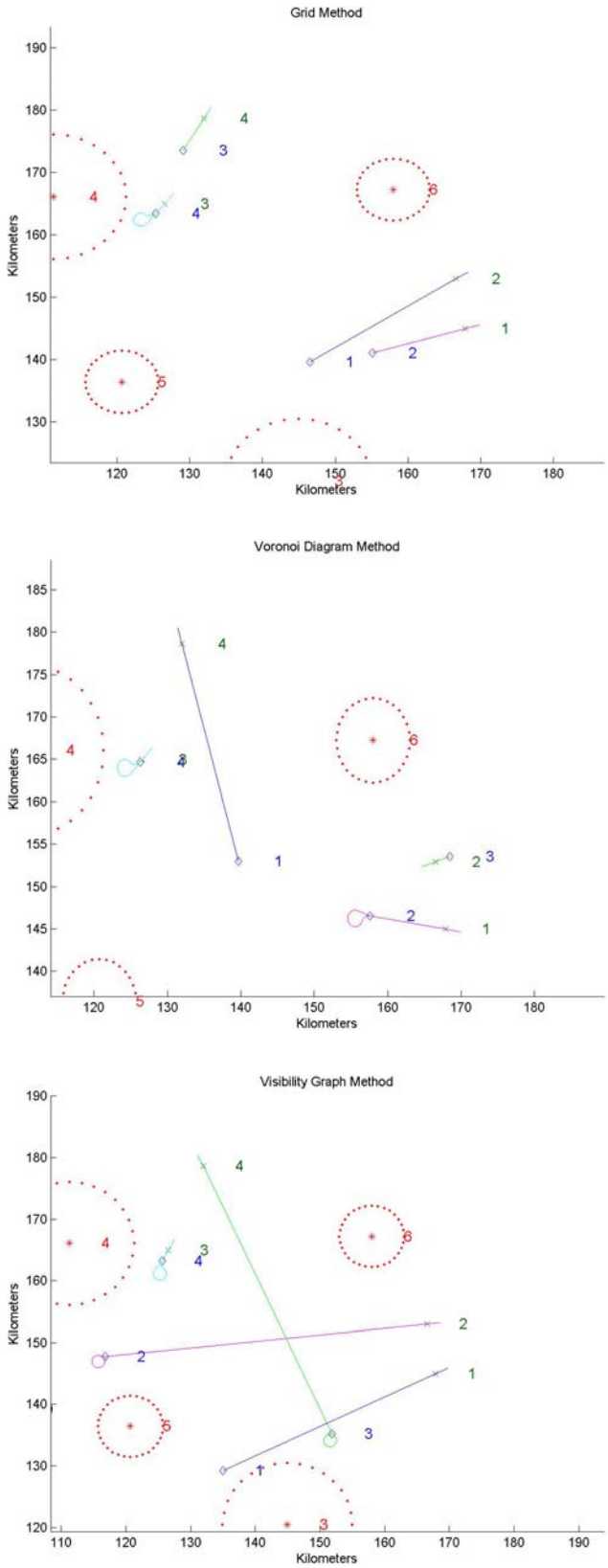


Figure 6.2.4 – 3rd replan of the simulation for all three methods

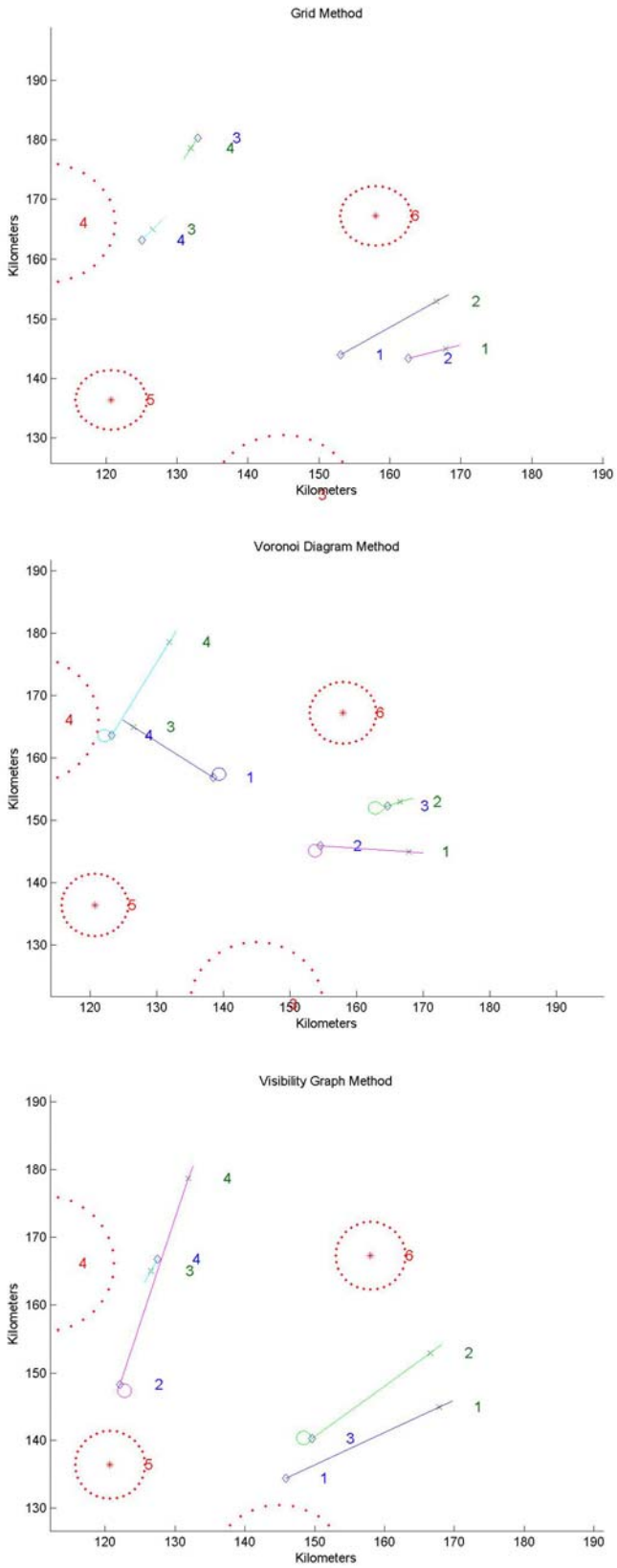


Figure 6.2.5 – 4th replan of the simulation for all three methods

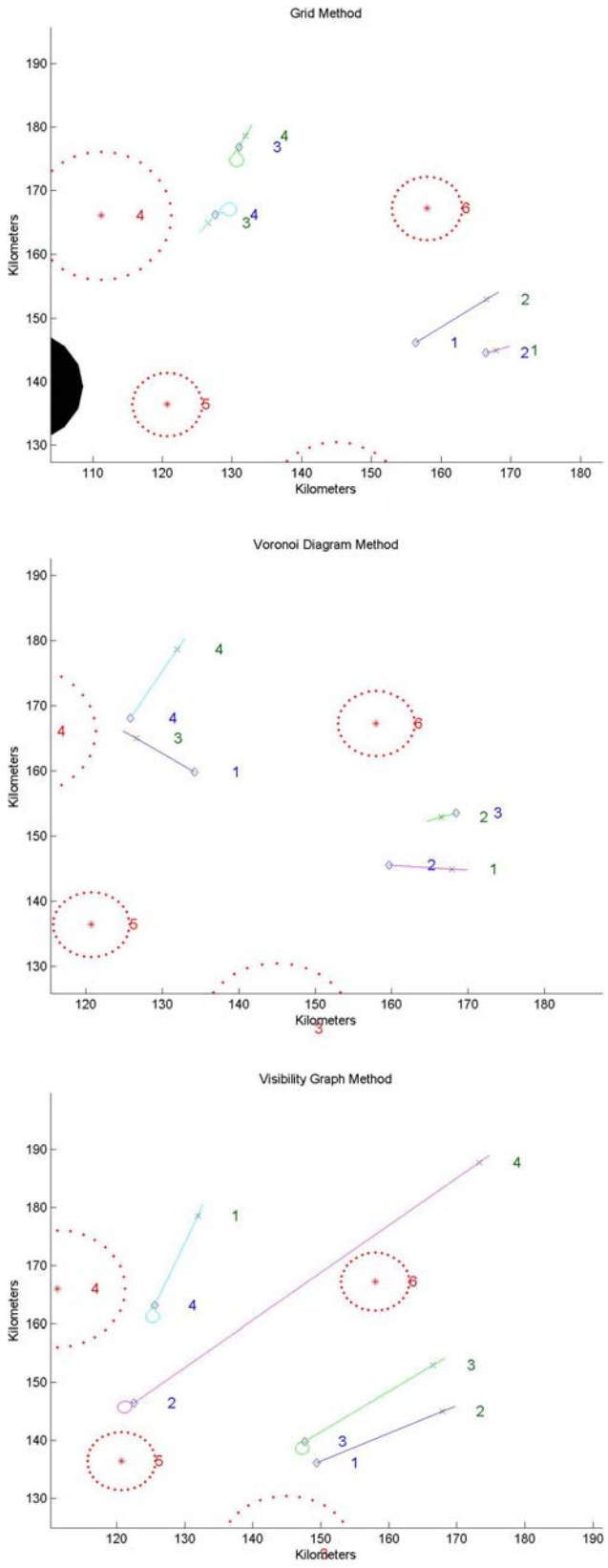


Figure 6.2.6 – 5th replan of the simulation for all three methods

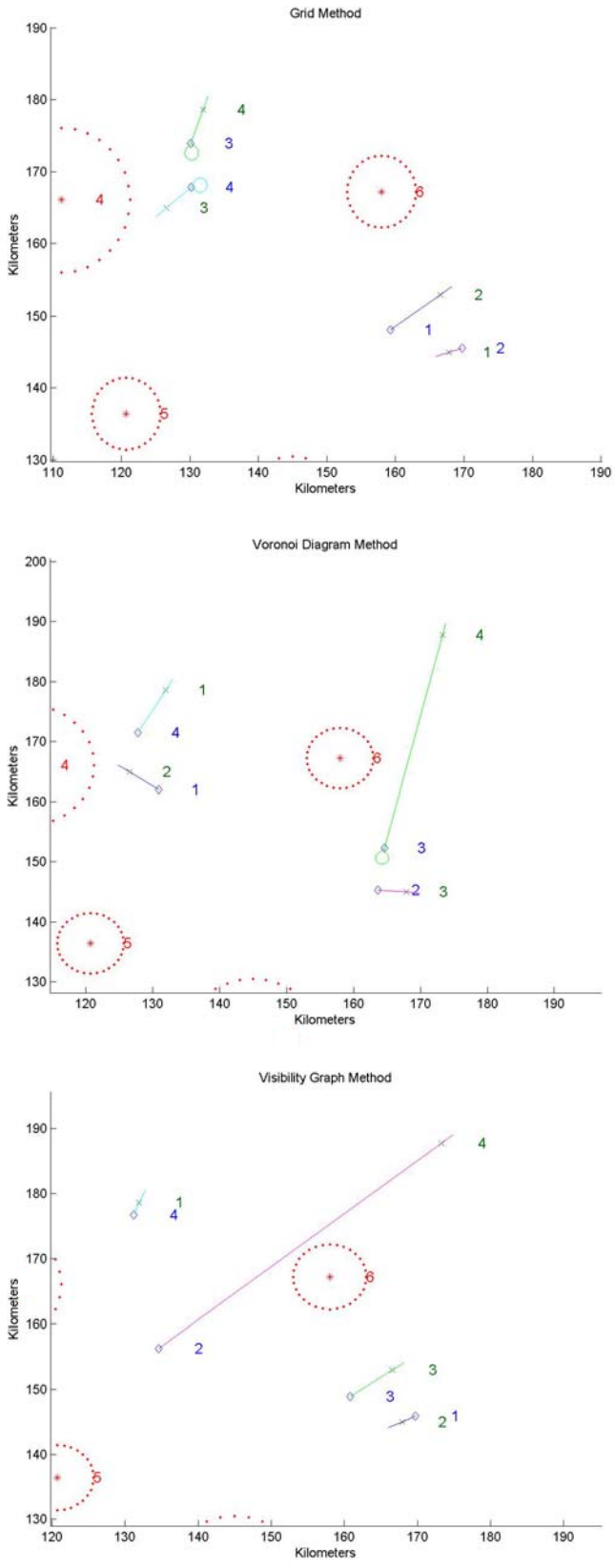


Figure 6.2.7 – 6th replan of the simulation for all three methods

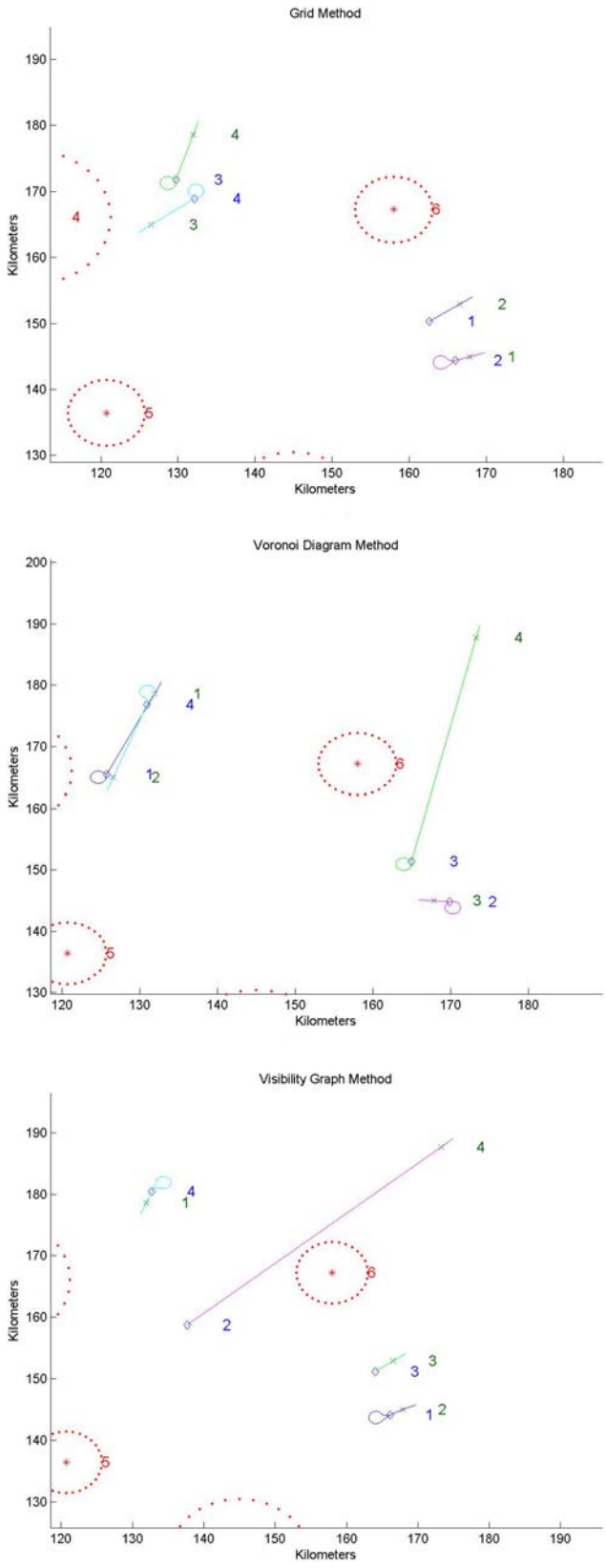


Figure 6.2.8 – 7th replan of the simulation for all three methods

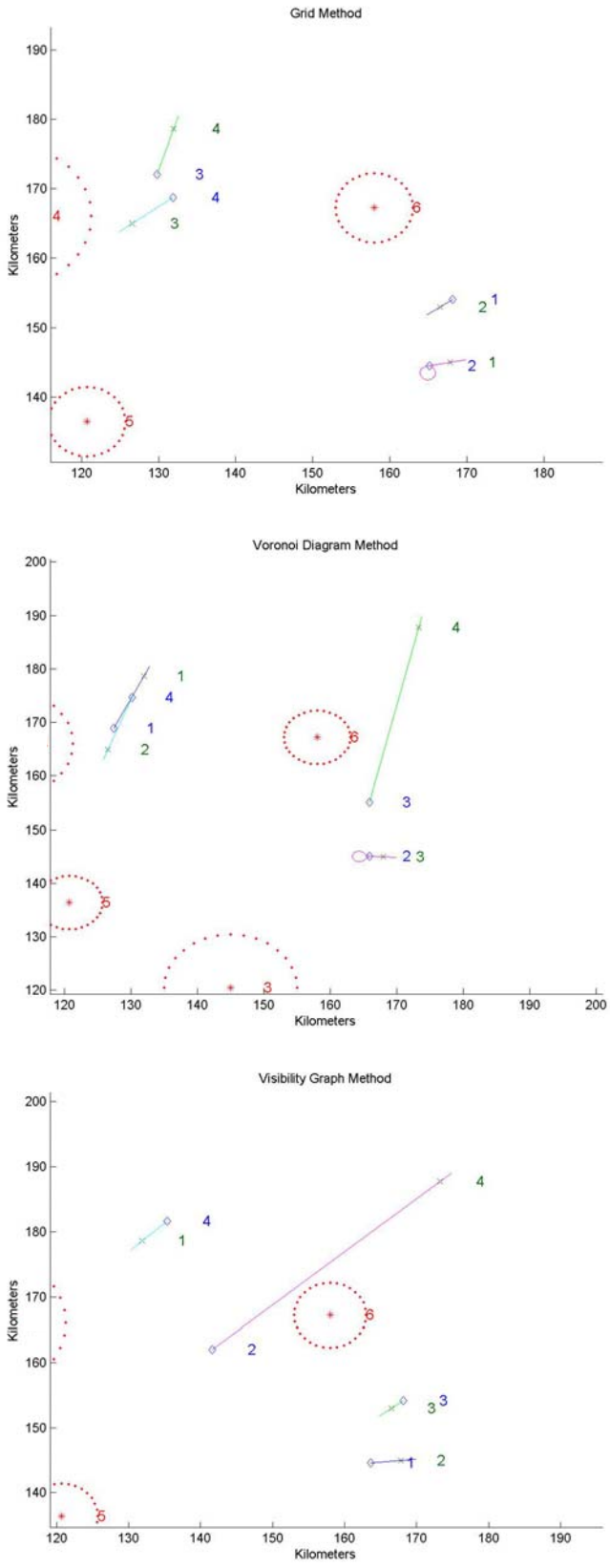


Figure 6.2.9 – 8th replan of the simulation for all three methods

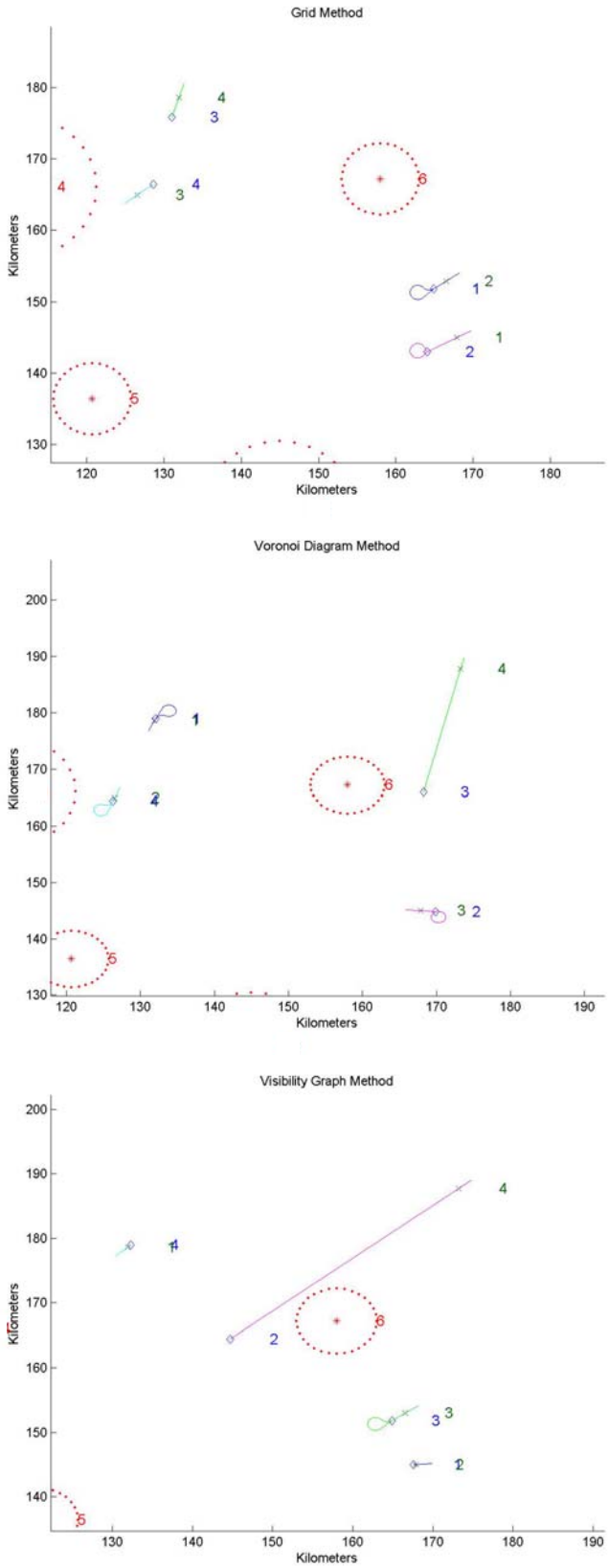


Figure 6.2.10 – 9th replan of the simulation for all three methods

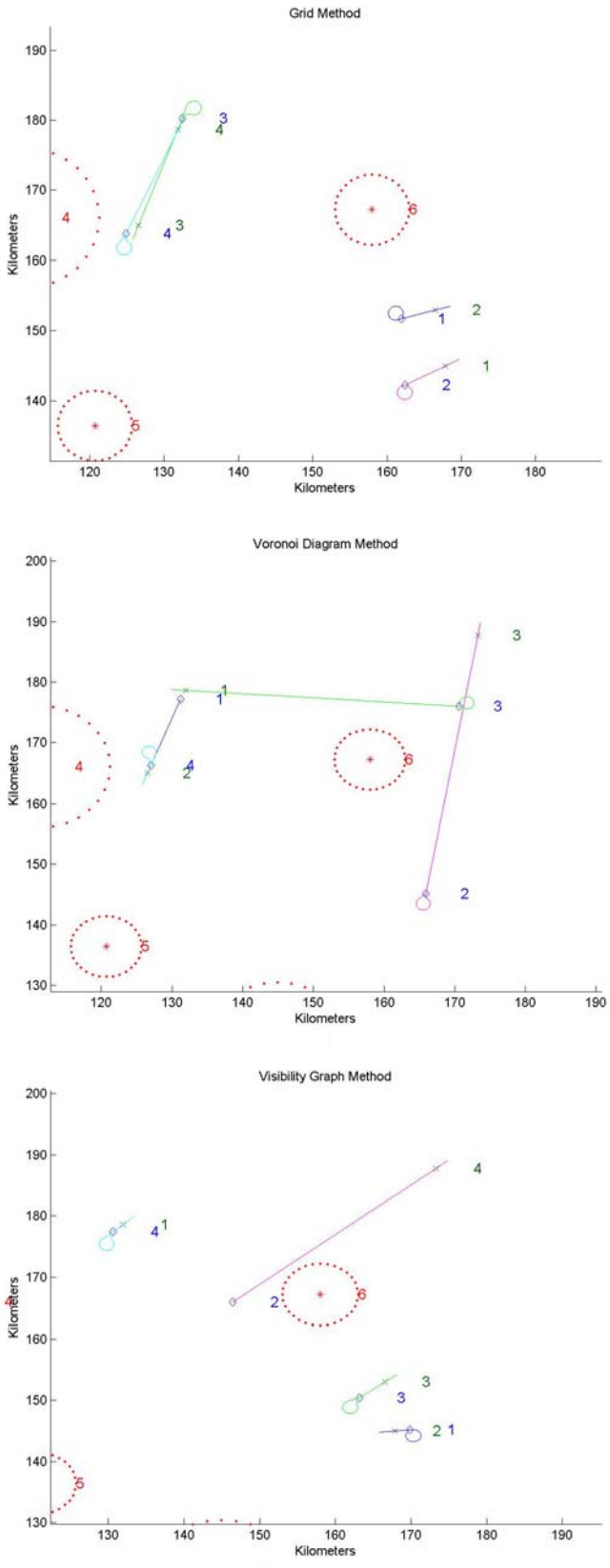


Figure 6.2.11 – 10th replan of the simulation for all three methods

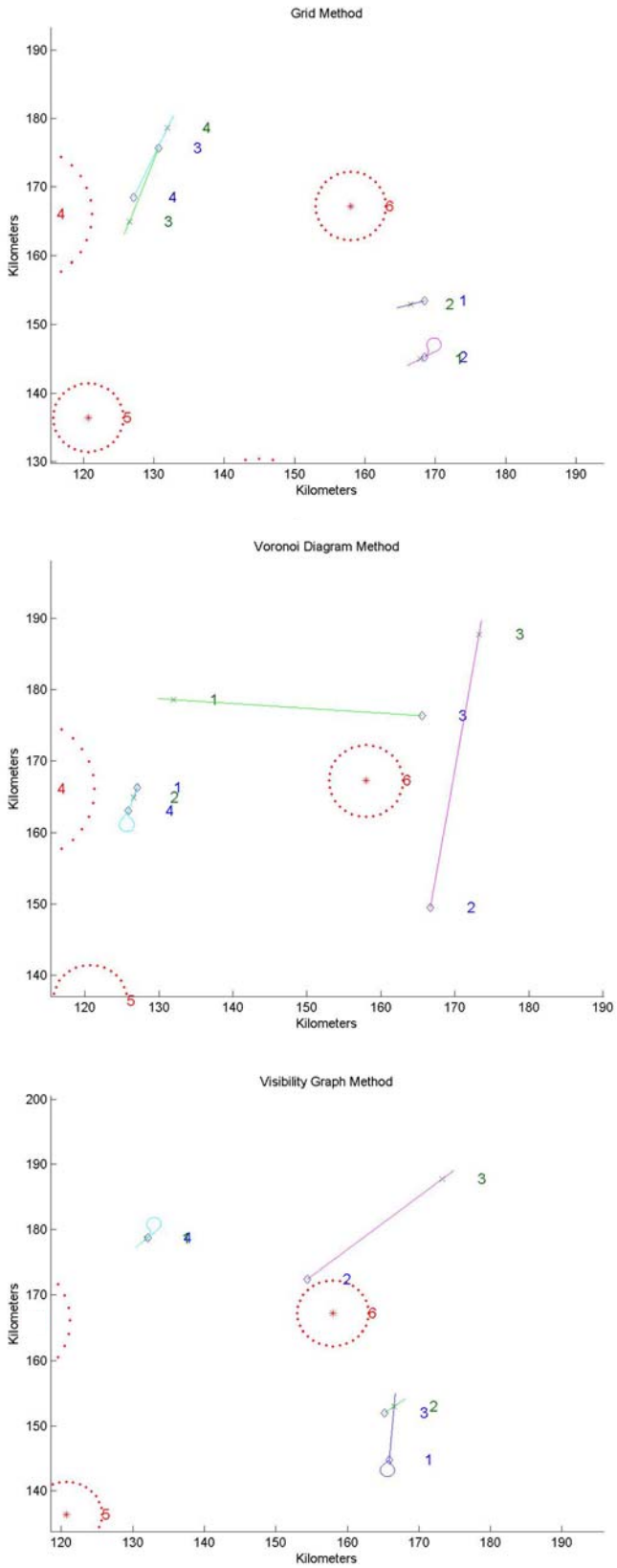


Figure 6.2.12 – 11th replan of the simulation for all three methods

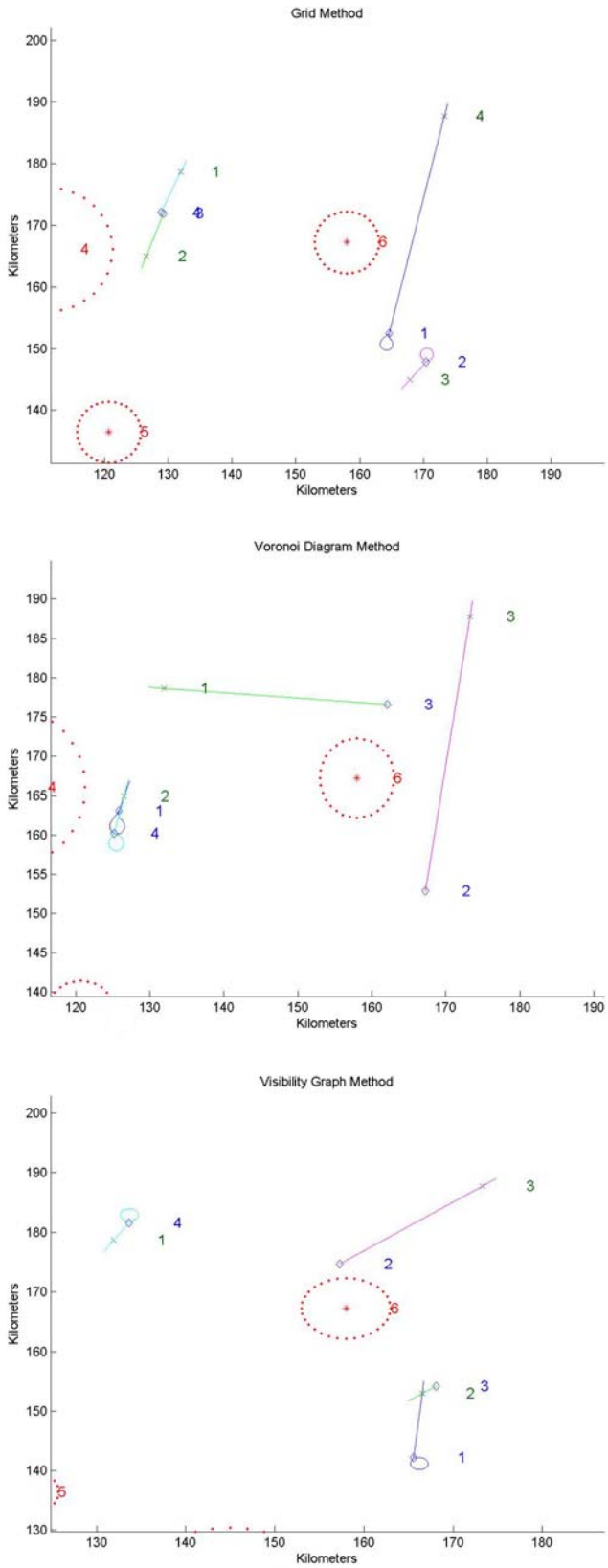


Figure 6.2.13 – 12th replan of the simulation for all three methods

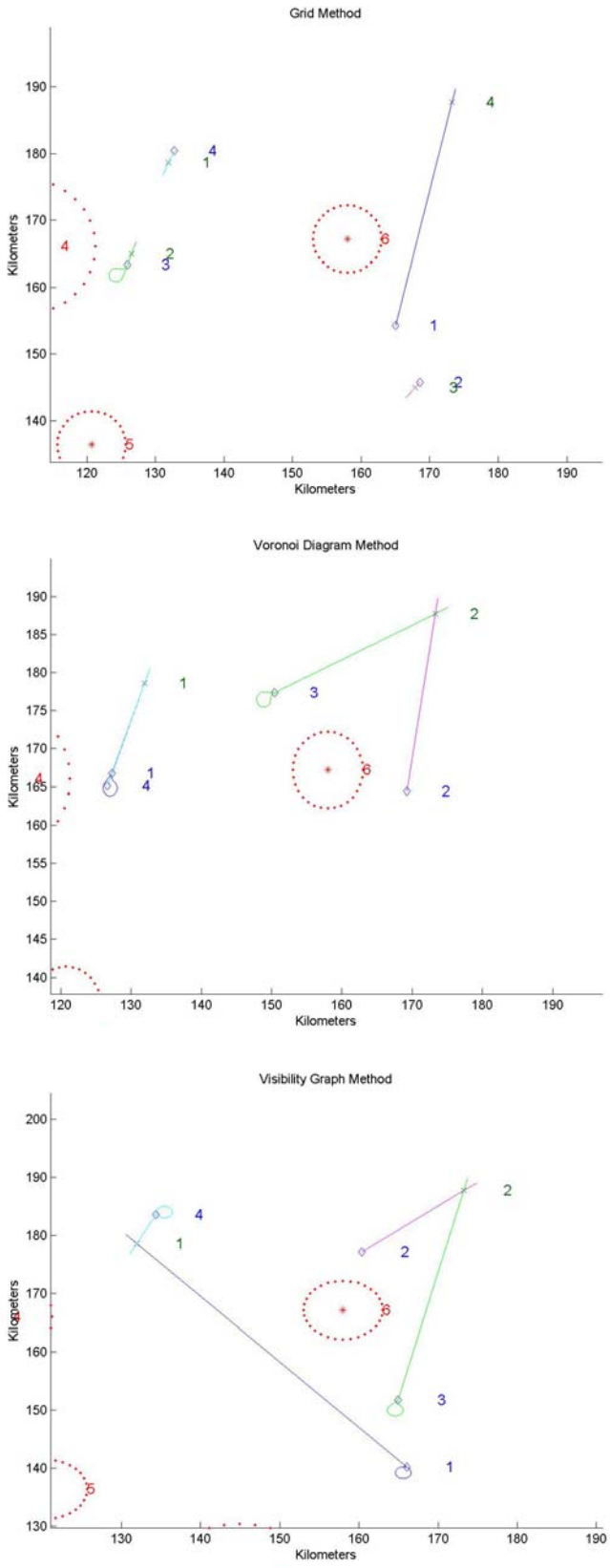


Figure 6.2.14 – 13th replan of the simulation for all three methods

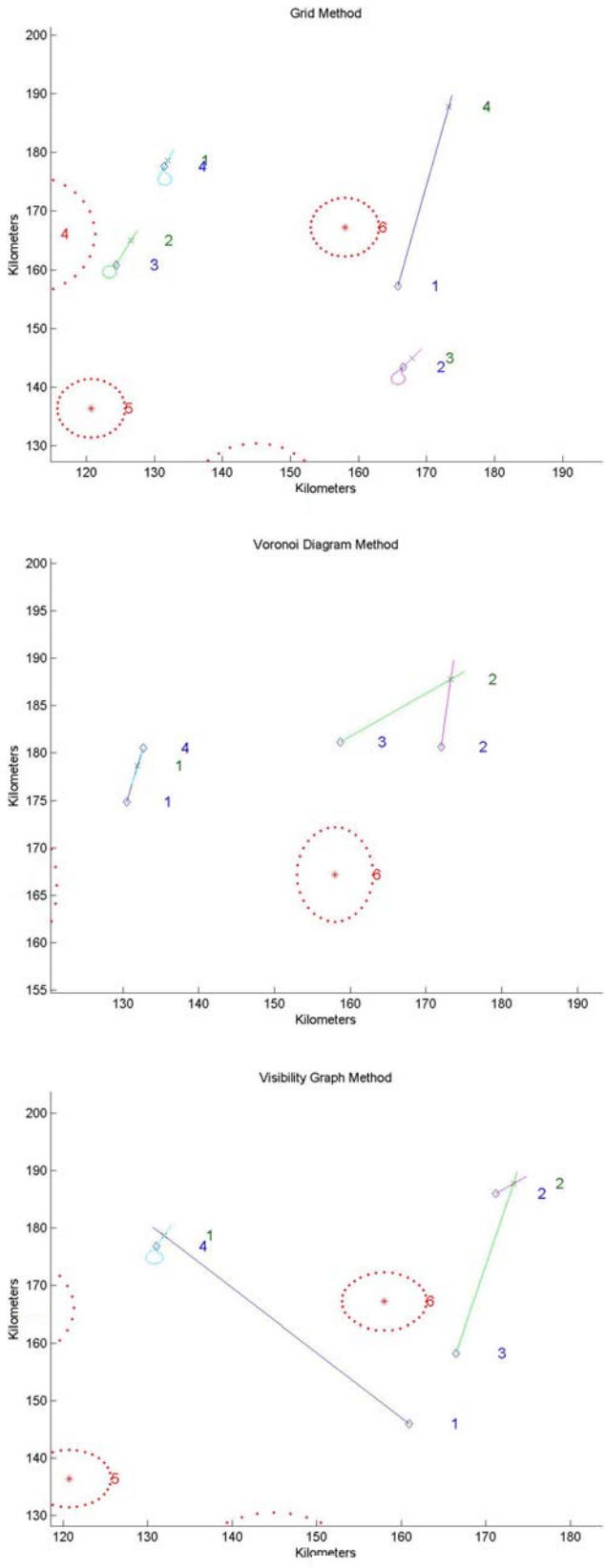


Figure 6.2.15 – 14th replan of the simulation for all three methods

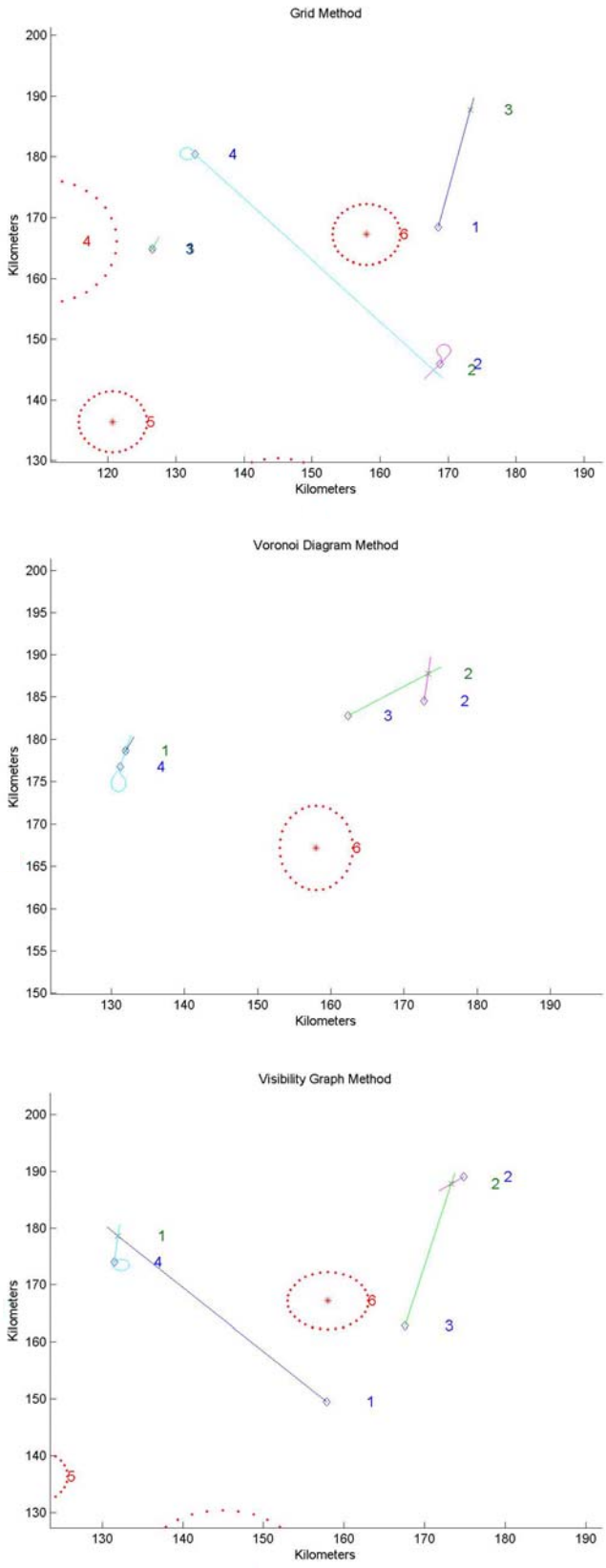


Figure 6.2.16 – 15th replan of the simulation for all three methods

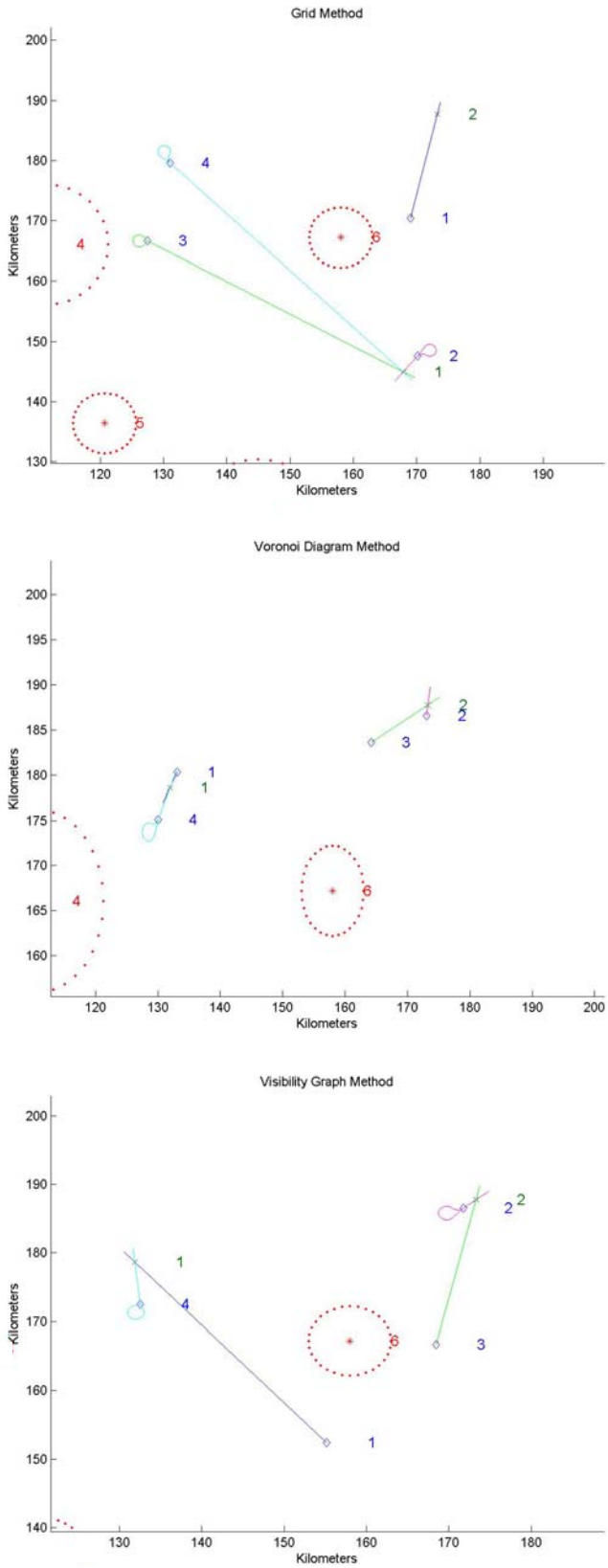


Figure 6.2.17 – 16th replan of the simulation for all three methods

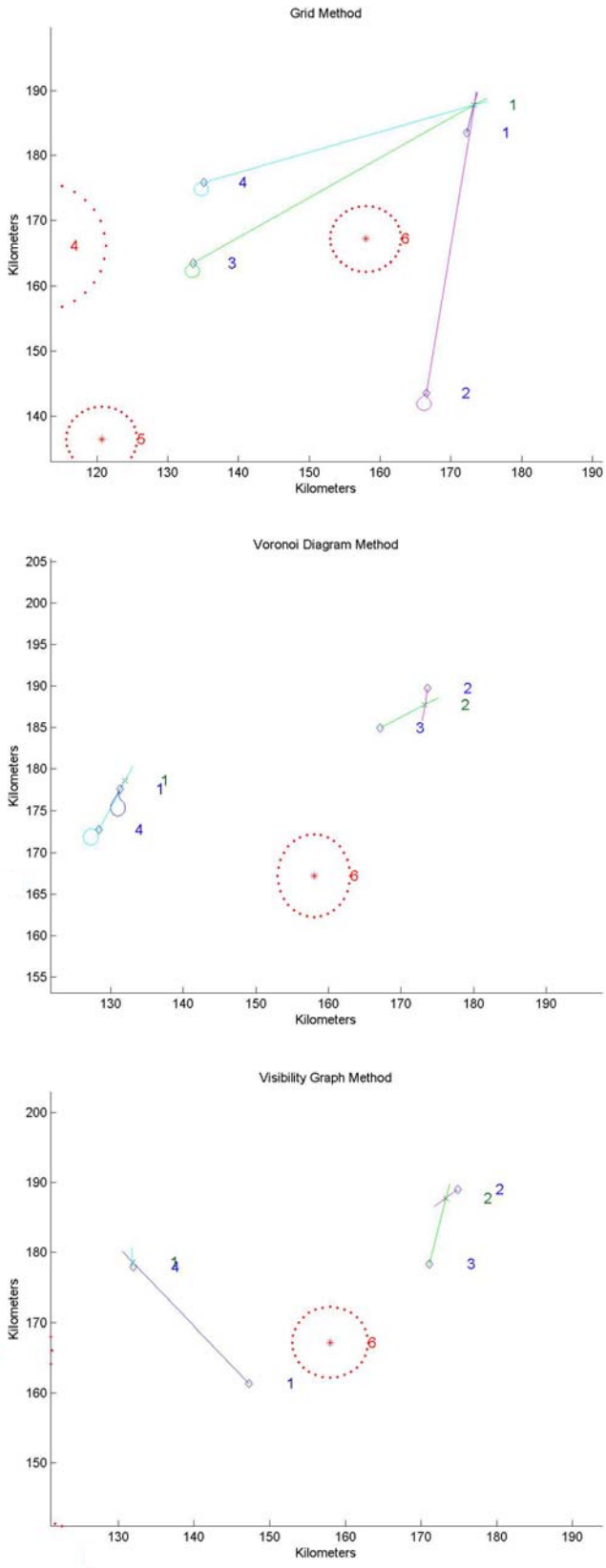


Figure 6.2.18 – 17th replan of the simulation for all three methods

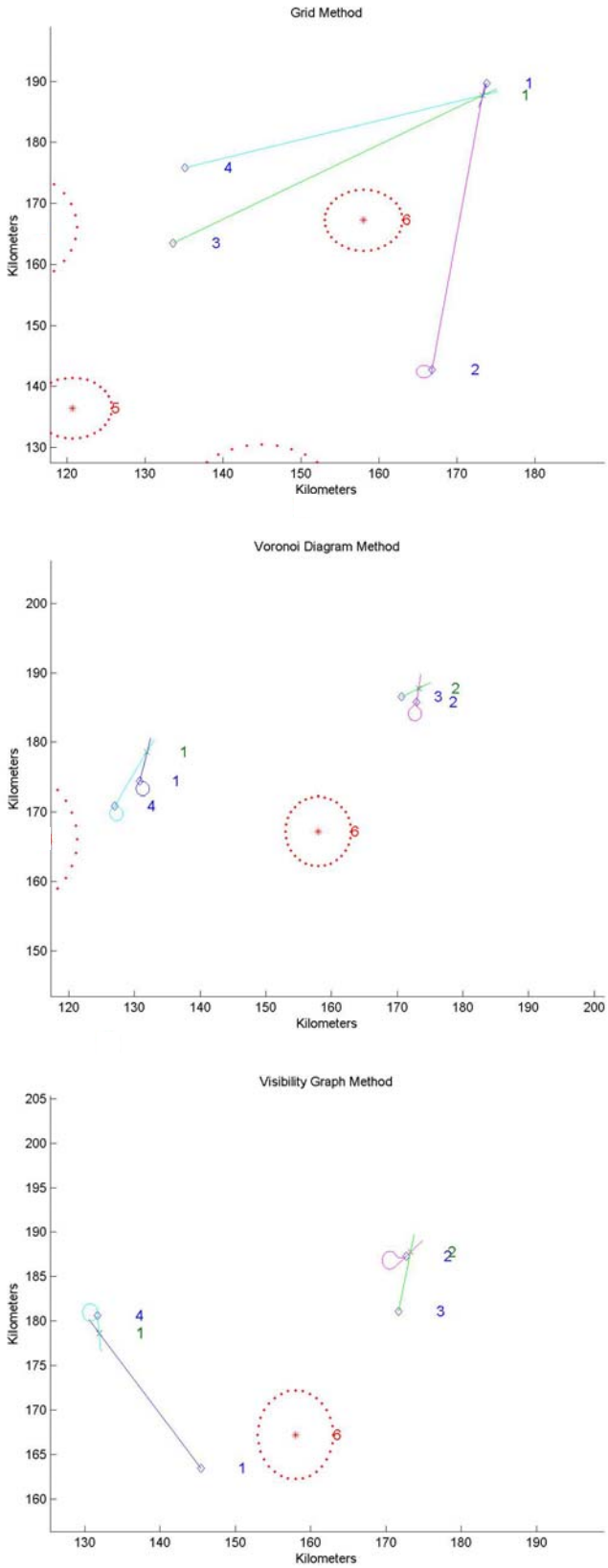


Figure 6.2.19 – 18th replan of the simulation for all three methods

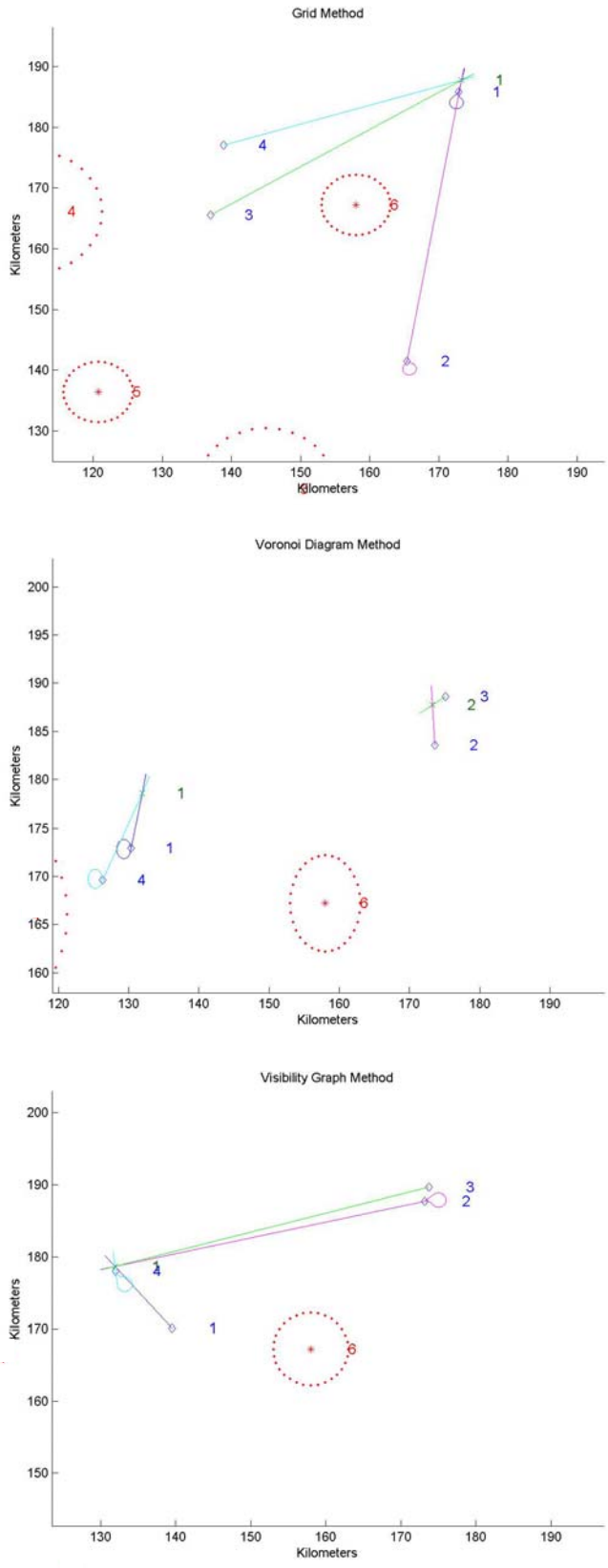


Figure 6.2.20 – 19th replan of the simulation for all three methods

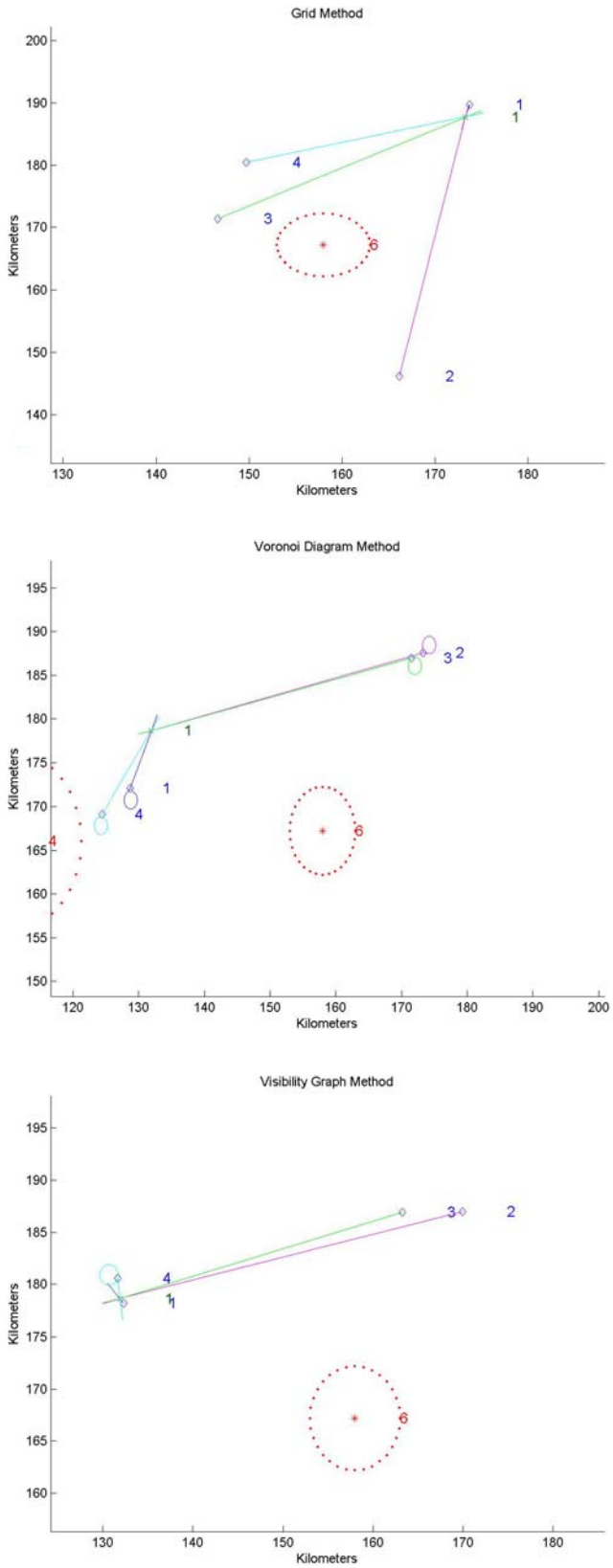


Figure 6.2.21 – 20th replan of the simulation for all three methods

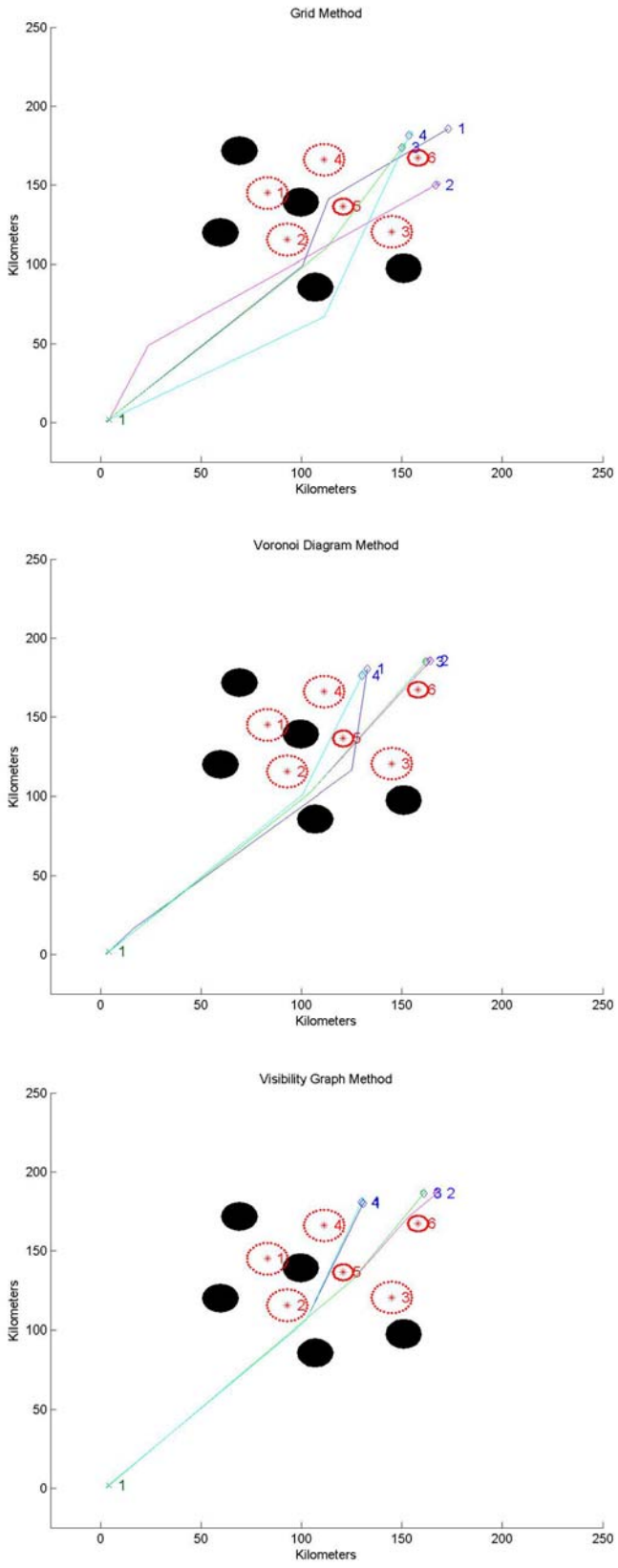


Figure 6.2.22 – 21st replan of the simulation for all three methods

```
UAV 1 exists at location 29 x, location 22 y, altitude 2 km, and is flying at 130 m/s.
UAV 2 exists at location 22 x, location 42 y, altitude 2 km, and is flying at 130 m/s.
UAV 3 exists at location 16 x, location 65 y, altitude 2 km, and is flying at 130 m/s.
UAV 4 exists at location 10 x, location 82 y, altitude 2 km, and is flying at 130 m/s.
Target 1 indicated to be at location 132 x, location 179 y , and with an estimated value of 70.
Target 2 indicated to be at location 127 x, location 165 y , and with an estimated value of 80.
Target 3 indicated to be at location 168 x, location 145 y , and with an estimated value of 100.
Target 4 indicated to be at location 167 x, location 153 y , and with an estimated value of 90.
Target 5 indicated to be at location 173 x, location 188 y , and with an estimated value of 40.
No-Fly Zone 1 exists at location 69 x, location 172 y, and with a radius of 9 km.
No-Fly Zone 2 exists at location 100 x, location 139 y, and with a radius of 9 km.
No-Fly Zone 3 exists at location 151 x, location 98 y, and with a radius of 9 km.
No-Fly Zone 4 exists at location 107 x, location 86 y, and with a radius of 9 km.
No-Fly Zone 5 exists at location 60 x, location 120 y, and with a radius of 9 km.
Threat 1 exists at location 83 x, location 145 y, with a range of 10 km, and has a probability of kill of 80%.
Threat 2 exists at location 93 x, location 116 y, with a range of 10 km, and has a probability of kill of 80%.
Threat 3 exists at location 145 x, location 120 y, with a range of 10 km, and has a probability of kill of 80%.
Threat 4 exists at location 111 x, location 166 y, with a range of 10 km, and has a probability of kill of 80%.
Threat 5 exists at location 121 x, location 136 y, with a range of 5 km, and has a probability of kill of 50%.
Threat 6 exists at location 158 x, location 167 y, with a range of 5 km, and has a probability of kill of 50%.
Target 2 (value 80) identified as a target at time 1308 by UAV 4.
Target 2 (value 80) classified not attacked at time 1324 by UAV 4.
Target 1 (value 70) identified as a target at time 1354 by UAV 3.
Target 1 (value 70) classified not attacked at time 1370 by UAV 3.
Target 3 (value 100) identified as a target at time 1384 by UAV 2.
Target 3 (value 100) classified not attacked at time 1399 by UAV 2.
Target 4 (value 90) identified as a target at time 1425 by UAV 1.
Target 4 (value 90) classified not attacked at time 1441 by UAV 1.
Target 2 (value 80) attacked not assested at time 1459 by UAV 4.
Target 4 (value 90) attacked not assested at time 1510 by UAV 1.
Target 4 (value 0) assested as destroyed at time 1525 by UAV 1.
Target 1 (value 70) attacked not assested at time 1561 by UAV 4.
Target 3 (value 100) attacked not assested at time 1573 by UAV 2.
Target 1 (value 0) assested as destroyed at time 1618 by UAV 4.
Target 2 (value 0) assested as destroyed at time 1626 by UAV 3.
Target 3 (value 0) assested as destroyed at time 1678 by UAV 2.
Target 5 (value 40) identified as a target at time 1703 by UAV 1.
Target 5 (value 40) classified not attacked at time 1719 by UAV 1.
Target 5 (value 40) attacked not assested at time 1763 by UAV 1.
Target 5 (value 0) assested as destroyed at time 1778 by UAV 1.
```

Figure 6.2.23 – Log of the simulation for the grid method

```
UAV 1 exists at location 29 x, location 22 y, altitude 2 km, and is flying at 130 m/s.
UAV 2 exists at location 22 x, location 42 y, altitude 2 km, and is flying at 130 m/s.
UAV 3 exists at location 16 x, location 65 y, altitude 2 km, and is flying at 130 m/s.
UAV 4 exists at location 10 x, location 82 y, altitude 2 km, and is flying at 130 m/s.
Target 1 indicated to be at location 132 x, location 179 y , and with an estimated value of 70.
Target 2 indicated to be at location 127 x, location 165 y , and with an estimated value of 80.
Target 3 indicated to be at location 168 x, location 145 y , and with an estimated value of 100.
Target 4 indicated to be at location 167 x, location 153 y , and with an estimated value of 90.
Target 5 indicated to be at location 173 x, location 188 y , and with an estimated value of 40.
No-Fly Zone 1 exists at location 69 x, location 172 y, and with a radius of 9 km.
No-Fly Zone 2 exists at location 100 x, location 139 y, and with a radius of 9 km.
No-Fly Zone 3 exists at location 151 x, location 98 y, and with a radius of 9 km.
No-Fly Zone 4 exists at location 107 x, location 86 y, and with a radius of 9 km.
No-Fly Zone 5 exists at location 60 x, location 120 y, and with a radius of 9 km.
Threat 1 exists at location 83 x, location 145 y, with a range of 10 km, and has a probability of kill of 80%.
Threat 2 exists at location 93 x, location 116 y, with a range of 10 km, and has a probability of kill of 80%.
Threat 3 exists at location 145 x, location 120 y, with a range of 10 km, and has a probability of kill of 80%.
Threat 4 exists at location 111 x, location 166 y, with a range of 10 km, and has a probability of kill of 80%.
Threat 5 exists at location 121 x, location 136 y, with a range of 5 km, and has a probability of kill of 50%.
Threat 6 exists at location 158 x, location 167 y, with a range of 5 km, and has a probability of kill of 50%.
Target 2 (value 80) identified as a target at time 1370 by UAV 4.
Target 4 (value 90) identified as a target at time 1379 by UAV 3.
Target 4 (value 90) classified not attacked at time 1394 by UAV 3.
Target 4 (value 90) attacked not assested at time 1439 by UAV 3.
Target 4 (value 0) assested as destroyed at time 1455 by UAV 3.
Target 3 (value 100) identified as a target at time 1479 by UAV 2.
Target 3 (value 100) classified not attacked at time 1519 by UAV 2.
Target 3 (value 100) attacked not assested at time 1562 by UAV 2.
Target 3 (value 0) assested as destroyed at time 1602 by UAV 2.
Target 2 (value 80) classified not attacked at time 1647 by UAV 4.
Target 2 (value 80) attacked not assested at time 1661 by UAV 1.
Target 2 (value 0) assested as destroyed at time 1706 by UAV 1.
Target 1 (value 70) identified as a target at time 1770 by UAV 4.
Target 1 (value 70) classified not attacked at time 1786 by UAV 4.
Target 1 (value 70) attacked not assested at time 1794 by UAV 1.
Target 5 (value 40) identified as a target at time 1806 by UAV 2.
Target 5 (value 40) classified not attacked at time 1822 by UAV 2.
Target 5 (value 40) attacked not assested at time 1841 by UAV 3.
Target 5 (value 0) assested as destroyed at time 1856 by UAV 3.
Target 1 (value 0) assested as destroyed at time 1918 by UAV 1.
```

Figure 6.2.24 – Log of the simulation for the Voronoi diagram method

```
UAV 1 exists at location 29 x, location 22 y, altitude 2 km, and is flying at 130 m/s.
UAV 2 exists at location 22 x, location 42 y, altitude 2 km, and is flying at 130 m/s.
UAV 3 exists at location 16 x, location 65 y, altitude 2 km, and is flying at 130 m/s.
UAV 4 exists at location 10 x, location 82 y, altitude 2 km, and is flying at 130 m/s.
Target 1 indicated to be at location 132 x, location 179 y , and with an estimated value of 70.
Target 2 indicated to be at location 127 x, location 165 y , and with an estimated value of 80.
Target 3 indicated to be at location 168 x, location 145 y , and with an estimated value of 100.
Target 4 indicated to be at location 167 x, location 153 y , and with an estimated value of 90.
Target 5 indicated to be at location 173 x, location 188 y , and with an estimated value of 40.
No-Fly Zone 1 exists at location 69 x, location 172 y, and with a radius of 9 km.
No-Fly Zone 2 exists at location 100 x, location 139 y, and with a radius of 9 km.
No-Fly Zone 3 exists at location 151 x, location 98 y, and with a radius of 9 km.
No-Fly Zone 4 exists at location 107 x, location 86 y, and with a radius of 9 km.
No-Fly Zone 5 exists at location 60 x, location 120 y, and with a radius of 9 km.
Threat 1 exists at location 83 x, location 145 y, with a range of 10 km, and has a probability of kill of 80%.
Threat 2 exists at location 93 x, location 116 y, with a range of 10 km, and has a probability of kill of 80%.
Threat 3 exists at location 145 x, location 120 y, with a range of 10 km, and has a probability of kill of 80%.
Threat 4 exists at location 111 x, location 166 y, with a range of 10 km, and has a probability of kill of 80%.
Threat 5 exists at location 121 x, location 136 y, with a range of 5 km, and has a probability of kill of 50%.
Threat 6 exists at location 158 x, location 167 y, with a range of 5 km, and has a probability of kill of 50%.
Target 2 (value 80) identified as a target at time 1187 by UAV 4.
Target 2 (value 80) classified not attacked at time 1203 by UAV 4.
Target 2 (value 80) attacked not assested at time 1250 by UAV 4.
Target 2 (value 0) assested as destroyed at time 1265 by UAV 4.
Target 3 (value 100) identified as a target at time 1353 by UAV 1.
Target 3 (value 100) classified not attacked at time 1368 by UAV 1.
Target 4 (value 90) identified as a target at time 1388 by UAV 3.
Target 4 (value 90) classified not attacked at time 1404 by UAV 3.
Target 3 (value 100) attacked not assested at time 1412 by UAV 1.
Target 3 (value 0) assested as destroyed at time 1452 by UAV 1.
Target 4 (value 90) attacked not assested at time 1467 by UAV 3.
Target 4 (value 0) assested as destroyed at time 1482 by UAV 3.
Target 1 (value 70) identified as a target at time 1537 by UAV 4.
Target 5 (value 40) identified as a target at time 1555 by UAV 2.
Target 5 (value 40) classified not attacked at time 1571 by UAV 2.
Target 5 (value 40) attacked not assested at time 1617 by UAV 2.
Target 1 (value 70) classified not attacked at time 1628 by UAV 4.
Target 5 (value 0) assested as destroyed at time 1663 by UAV 3.
Target 1 (value 70) attacked not assested at time 1705 by UAV 4.
Target 1 (value 0) assested as destroyed at time 1715 by UAV 1.
```

Figure 6.2.25 – Log of the simulation for the visibility graph method

Chapter 7

Implementation and Discussion of Search Scheme in SIMULINK

7.1 - Implementation of a SIMULINK Based Search Scheme

As discussed previously, there are two types of cooperating UAV problems. One has been covered in the preceding chapters, a bombing type UAV that has knowledge of the entire battlefield before launch. For the purpose of this research effort, that type of vehicle is the main concentration. This will eventually be developed into model aircraft. The other type of UAV of interest to the Air Force is a disposable UAV, such as the Predator. These UAVs will perform a search and destroy mission. It is evident from inspection of any war that both scenarios are extremely realistic and important.

The search and destroy mission starts with the assumption that everything about an area is unknown, except the position of the UAVs and the size of the area. The only goal of these inexpensive vehicles is to search out and destroy targets. Unlike the other mission where there are no-fly zones and threats to be avoided. This scenario is mainly for the suppression of enemy defenses or any other mission in which an area needs to be cleared. Essentially, the threats and the targets become one in the same. For simplicity each target is assumed to be incapable of destroying a vehicle. This assumption is made because if a target is considered a threat it would immediately attack the UAV, not giving the vehicle a chance to communicate the information it has gathered.

Due to these assumptions about the battlefield the original control scheme needed to be completely redesigned. The SIMULINK scheme for this is shown in Figure 7.1.1.

In this control system, the general architecture stays the same with a central path-planning block that contains the main decision making algorithms.

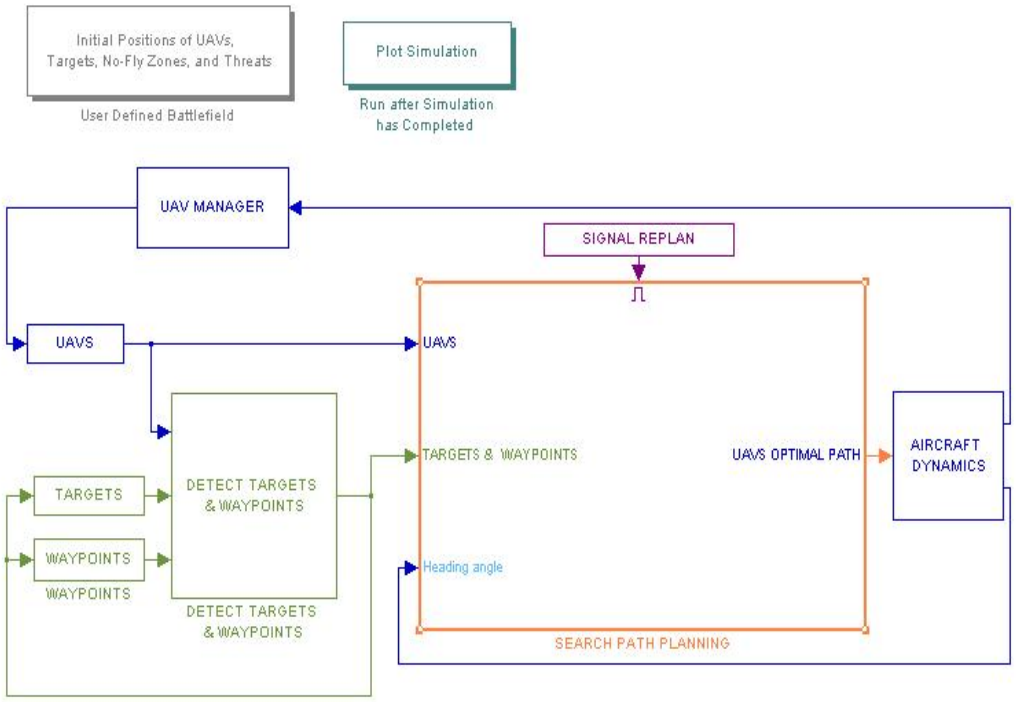


Figure 7.1.1 – Search control scheme in SIMULINK

Also, the heading angle control design and the UAV manager stayed the same. All of the other blocks were either replaced or removed. Instead of a targets manager, a targets and waypoints manager was created. It was designed this way to assign the targets and waypoints interchangeably for each vehicle.

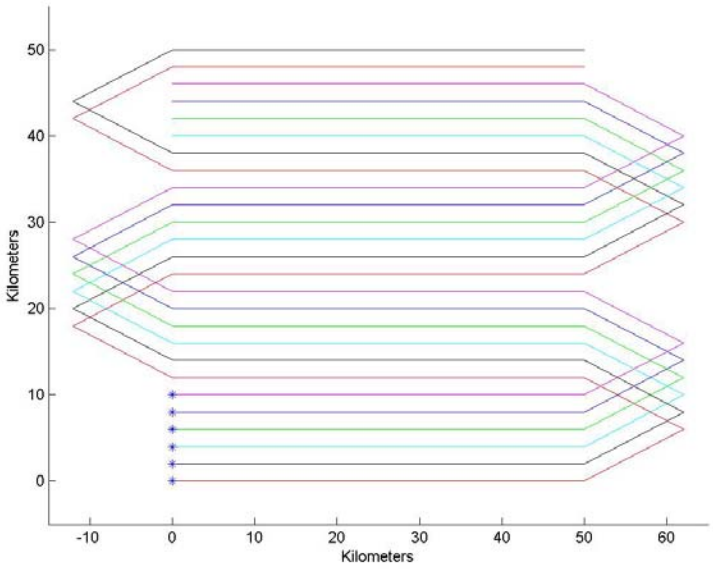


Figure 7.1.2 – Serpentine search pattern

These waypoints were assigned such that the field is searched using a serpentine pattern, an example of this can be seen in Figure 7.1.2. This allows the entire area to be searched efficiently. This was accomplished by assigning each UAV to visit a point directly across from it. After that point the path sweeps around to search another area of the battlefield traveling the opposite direction. This process is repeated until the entire area is searched.

During the search of this area a target can be discovered, when this happens a number of vehicles must be assigned to perform an action on this target. A target in a search and destroy mission can have 5 states.

- Undetected
- Detected
- Classified as a valid / invalid target
- Attacked

- Assessed as destroyed / not destroyed

This is similar to the previous case but with the addition of the first state, undetected, since there is not knowledge of the battlefield a target cannot be identified, merely detected. A target is considered detected if it travels within 1,000 meters of the vehicle. In order for the target to change to any other state it must be within 10 meters of the vehicle. This is the same distance that the UAV must travel within a waypoint for the vehicle to be assigned to its next waypoint. The implementation of this in SIMULINK is shown in Figure 7.1.3.

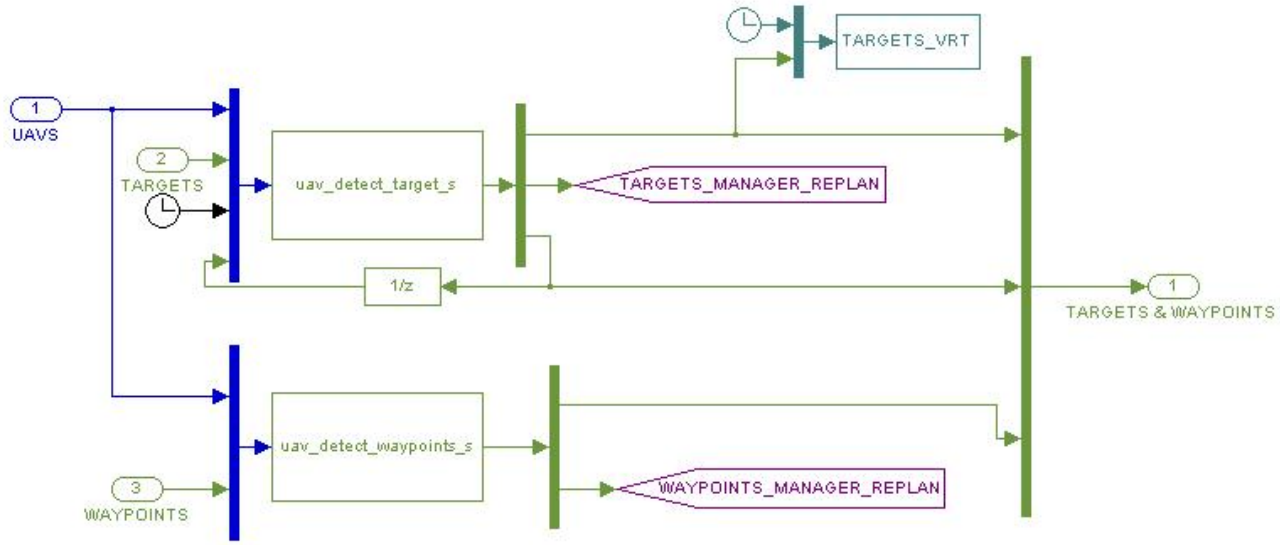


Figure 7.1.3 – Detect targets and waypoints SIMULINK block

In much the same fashion as before, if a target changes states, UAV becomes lost, or a waypoint is visited a replan is signaled for the entire group of vehicles. If a target changes states, an appropriate number of vehicles are sent to the target to perform all of the needed tasks. This decision is made by the central path-planning block shown in Figure 7.1.4.

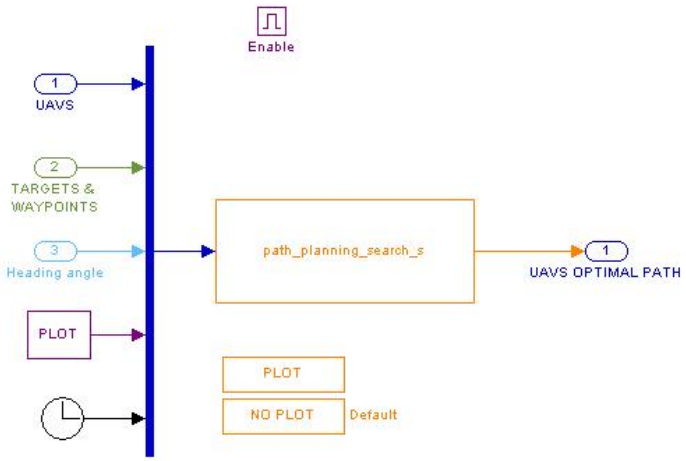


Figure 7.1.4 – Path planning SIMULINK block

Since there are no threats or no-fly zones the possible paths for each UAV to take to reach each target are straight lines. The only modification that must be made before tasks are assigned is each path must be flyable. This is accomplished by using the previously mentioned method in Algorithm 3.2.2. After each path is flyable the UAVs with the shortest paths are selected to visit the target. If a vehicle is not assigned to visit a target it continues on its current path. After each vehicle is assigned a path it is then input into the same heading angle autopilot designed in Chapter 4.

Upon initialization of the battlefield a function named "*waypoint_gen*", seen in Appendix E, is called, which defines the set of waypoints for each UAV to follow. The inputs of this function are the number, position, the minimum turn radius of the UAVs, and the size of the area to be searched. This function yields the locations of the waypoints that each vehicle is assigned to visit.

The block seen in Figure 7.1.3 calls two S-functions, "*uav_detect_targets_*s" and "*uav_detect_waypoints_s"*, both are located in Appendix E. The first calls a function

"*uav_detect_target"*, which inputs the current location of each UAV, the location of each target, and the state of each target. This function compares the positions to evaluate if the target should change states and if required it changes the state. The output of this function is the updated target states. The second S-function calls "*uav_detect_waypoints"*, which evaluates if a waypoint has been visited. This function compares the current locations of the UAVs to the locations of the waypoints. If a waypoint is visited, the function assigns the next waypoint to the UAV.

The central path-planning block calls the S-function "*path_planning_search_s"*, which is shown in Appendix E. This function is invoked when a replan is signaled. All of the current information is input to this block which calls the "*path_planning_search"* function. This function contains the following algorithm:

Algorithm 7.1.1

1. If target *i* is present
2. Calculate flyable path for each UAV to target *i*
3. Assign N_{TASKS} UAVs to visit target
   N_{TASKS} is current state of the target
4. If UAV not assigned to visit a target
   continue to current waypoint
5. Go back to step 1

This assigns each UAV a path based on its current waypoints or a target changing states. The heading angle control system designed previously is then applied so that each UAV can follow the selected path.

7.2 - Results of a Search Simulation

To visualize the results of the simulation a similar method was adapted to that in Chapter 6. Each time an action occurs on the battlefield and a replan is signaled, a figure is plotted that shows the current position and path for each UAV as well as the current position of the detected targets. The first several of these replans are shown in Figures 7.2.1 - 7.2.6. In the figures shown a target is detected and destroyed. After the target has been destroyed the UAVs proceed to their next assigned waypoint. In addition, a statement was printed to the command line of MATLAB for the purpose of keeping a simulation log, which is shown in Figure 7.2.7.

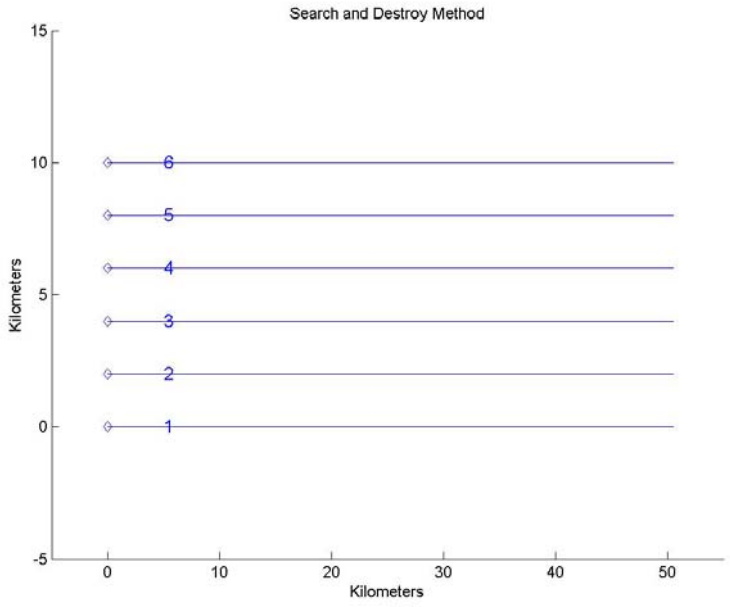


Figure 7.2.1 – 1st replan for search simulation

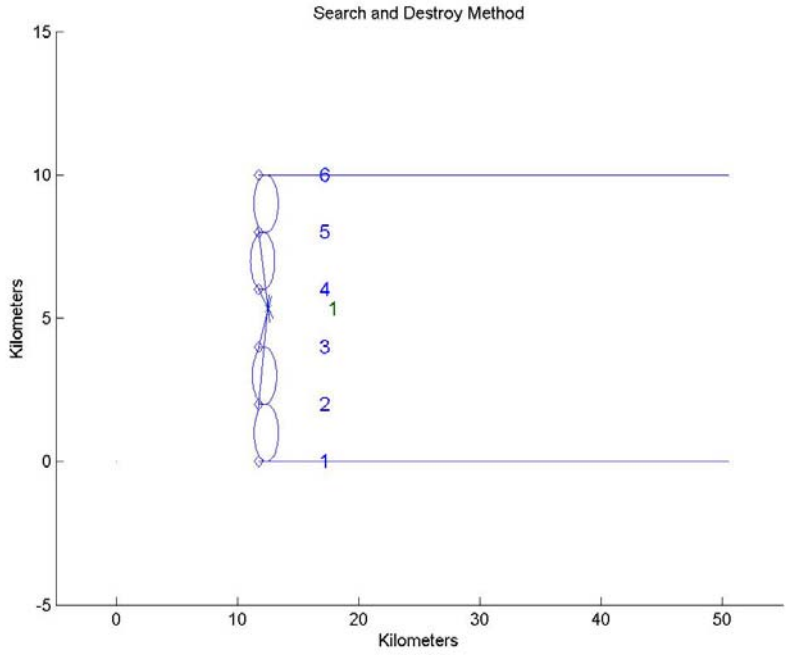


Figure 7.2.2 – 2nd replan for search simulation

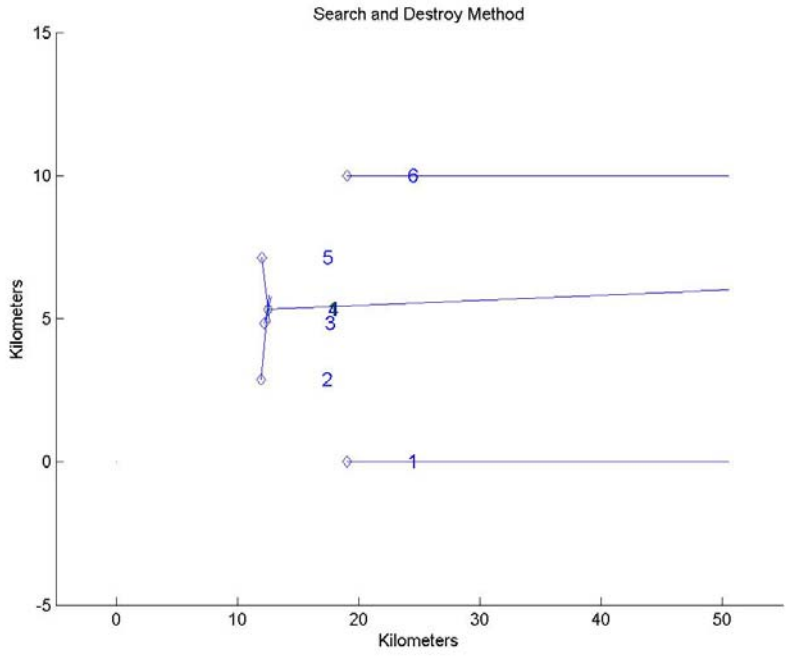


Figure 7.2.3 – 3rd replan for search simulation

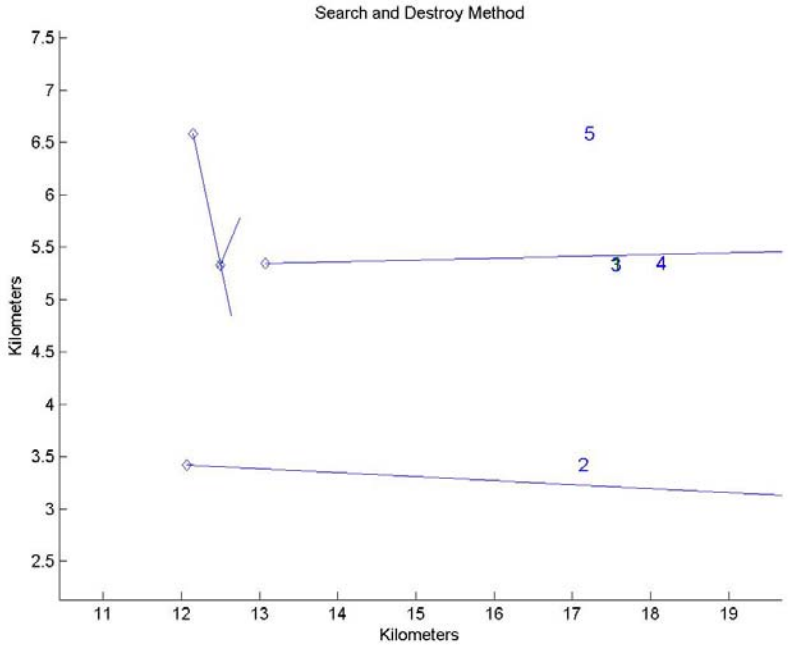


Figure 7.2.4 – 4th replan for search simulation

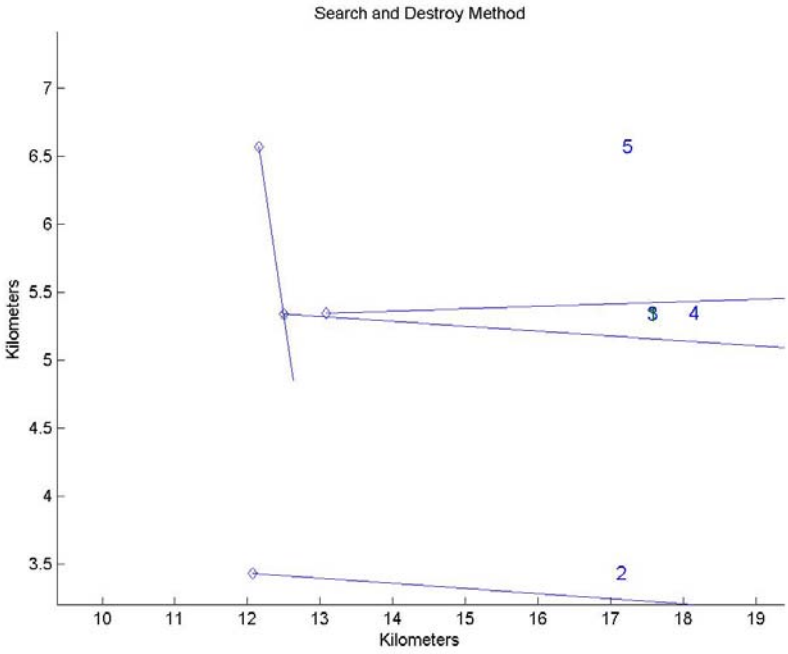


Figure 7.2.5 – 5th replan for search simulation

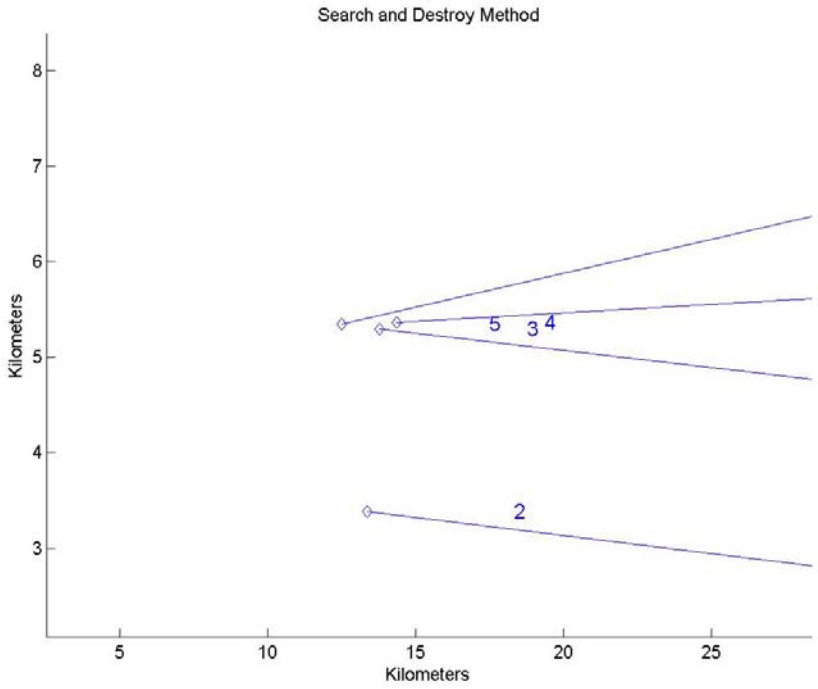


Figure 7.2.6 – 6th replan for search simulation

```
UAV 1 exists at location 0 x, location 0 y, altitude 2 km, and is flying at 130 m/s.
UAV 2 exists at location 0 x, location 2 y, altitude 2 km, and is flying at 130 m/s.
UAV 3 exists at location 0 x, location 4 y, altitude 2 km, and is flying at 130 m/s.
UAV 4 exists at location 0 x, location 6 y, altitude 2 km, and is flying at 130 m/s.
UAV 5 exists at location 0 x, location 8 y, altitude 2 km, and is flying at 130 m/s.
UAV 6 exists at location 0 x, location 10 y, altitude 2 km, and is flying at 130 m/s.
Target 1 (value 100) detected at time 91 by UAV 4.
Target 1 (value 100) classified as a target at time 119 by UAV 4.
Target 1 (value 100) classified not attacked at time 121 by UAV 3.
Target 1 (value 100) attacked not assested at time 122 by UAV 3.
Target 1 (value 100) assested as destroyed at time 126 by UAV 5.
```

Figure 7.2.7 – Log for search simulation

Chapter 8

Conclusions and Recommendations

8.1 - Conclusions

This research is the first step in the process of implementing cooperating UAVs onto a real battlefield. The importance of these vehicles is becoming increasingly apparent. UAVs are being used more in real world applications such as the war in Iraq to searching missions in Afghanistan. Clearly, these vehicles are the way of the future. They have lower operational cost, present less risk of loss of human life, and far greater maneuverability capabilities.

As has been presented in this thesis, the cooperating UAVs problem is exceedingly complex. Many different researchers have attempted this problem as shown in Chapter 2. This paper presents simulations that have the ability to replan and avoid obstacles in a battlefield environment, as well as a pure search and destroy mission. The simulation discussed can react to a dynamic environment such as targets popping up, threats popping up, classifying targets, loss of a UAV, and firing of a threat. In a realistic battlefield scenario, if a vehicle cannot react properly to the environment it is inhabiting it serves no purpose.

Any cooperating UAV simulation must have the capability to find, classify, destroy, and perform a battle damage assessment on each target. This must be accomplished using real-time computations, which this paper shows can be accomplished. The three path generation methods presented in this document are grid, Voronoi diagram, and visibility graph. Chapter 6 provides a comparison of these methods.

In the example, several factors must be compared to determine the best solution to the problem. The visibility graph method provided superior results for the cost of the initial plan, the Voronoi diagram method was 16% greater and the grid method was 31% greater. The visibility graph also had a lower total simulation time than the other two methods, 3.6% less than the grid method and 12% less than the Voronoi diagram method. This shows that the visibility graph is the optimal method of the three.

Comparing the average of the individual replan calculations the Voronoi diagram method provided a 76% decrease from the visibility graph method, while the decrease was 38% for the grid method. This difference becomes less evident when comparing the total calculation time of the simulation, which the Voronoi diagram method is 1% less than the visibility graph method and 3% less than the grid method.

While the individual replan computation time is significantly reduced by using the Voronoi diagram method, the optimization of the simulation suffers. From these comparisons one may draw the conclusion that the visibility graph provides best results, which is because it is an exhaustive solution as opposed to an approximate solution. If the battlefield complexity is low, the visibility graph should be used. As the complexity of the battlefield increases, this method is not feasible and the Voronoi diagram method should be used. For the given in scenario in Chapter 6, it is the conclusion that the visibility graph method would be the best option.

There are several possible reasons for error involved in the gathering of this data. These are human error in recording the total calculation time of the simulation, which could be in the range of 5%. Another source of error could be the calculation time of

each replan. Each time this was calculated the computer could be running different processes that could result in varying processor speed.

In addition, a control scheme to simulate a search and destroy mission was designed. This simulation was created to show the other purpose of UAVs. The goal in a search and destroy mission is to search out targets on a battlefield when there is no prior knowledge of the given area. These vehicles must clear the battlefield of targets using a market-based bidding procedure to assign each UAV a task to accomplish the desired mission. Chapter 7 shows a simulation that accomplishes this desired mission by destroying the given target.

8.2 - Recommendations

Both of the above scenarios are realistic, but in order to build model aircraft that can perform these simulations it must be coded on an airborne processor. To choose the proper method the battlefield must be clearly defined. The grid and Voronoi diagram methods lend themselves to a dynamic environment, while the visibility graph would be better applicable toward a static environment. Initially, a search and destroy mission with no obstacles would be easier to implement before introducing threats and no-fly zones into the problem. Some topics of future research could include the addition of timing constraints, collision avoidance, or a 3-D environment, into the simulation. In conclusion, this thesis has presented the initial steps necessary to implement cooperating UAVs on a model battlefield.

References

[1] B.S. Papadales, R.T. Leitner, “New Trends in High Altitude Unmanned Aircraft”, W.J. Schafer Associates, Inc., 1992.

[2] J.P. Nalepka, M.M. Duquette, “A Multi-purpose Simulation Environment for UAV Research”, Austin, TX, AIAA Modeling and Simulation Technologies Conference, August 2003.

[3] T.M. McLain, “Coordinated Control of Unmanned Air Vehicles”, Wright-Patterson Air Force Base, OH, Air Vehicles Directorate, 1999.

[4] P.R. Chandler, M. Pachter, D. Swaroop, J.M. Fowler, J.K. Howlett, S. Rasmussen, C. Schumacher, K. Nygard, “Complexity in UAV Cooperative Control”, Anchorage, AK, American Control Conference, May 2002.

[5] J. Bellingham, M. Tillerson, A. Richards, J.P. How, “Multi-task Allocation and Path Planning for Cooperating UAVs”, Conference on Coordination, Control and Optimization, November 2001.

[6] A. Richards, J. Bellingham, M. Tillerson, J. How, “Co-ordination and Control of Multiple UAVs”, Monterey, CA, AIAA Guidance, Navigation, and Control Conference, August 2002.

[7] S.A. Bortoff, “Path-Planning for Unmanned Air Vehicles”, Dayton, OH, AFRL / VAAD, 1999.

[8] A. Richards, J. Bellingham, M. Tillerson, J. How, “Coordination and Control of Multiple UAVs”, Monterey, CA, AIAA Guidance, Navigation, and Control Conference, August 2002.

[9] S. Li, J.D. Boskovic, S. Seereeram, R. Prasanth, J. Amin, R.K. Mehra, R.W. Beard, T.W. McLain, “Autonomous Hierarchical Control of Multiple Unmanned Combat Air Vehicles (UCAVs)”, Anchorage, AK, American Control Conference, May 2002.

[10] P.R. Chandler, M. Pachter, S.R. Rasmussen, C. Schumacher, “Distributed Control for Multiple UAVs with Strongly Coupled Tasks”, Austin, TX, AIAA Guidance, Navigation, and Control Conference, August 2003.

[11] M. Moser, D.P. Jokanovic, N. Shiratori, “An Algorithm for the Multidimensional Multiple-Choice Knapsack Problem”, IEICE Trans. Fundamentals, Vol. E80-A, No. 3, March 1997, pp. 582-589

[12] W. Kang, A. Sparks, “Task Assignment in the Cooperative Control of Multiple UAVs”, Austin, TX, AIAA Guidance, Navigation, and Control Conference, August 2003.

[13] T.W. McLain, R.W. Beard, “Trajectory Planning for Coordinated Rendezvous of Unmanned Air Vehicles”, Denver, CO, AIAA Guidance, Navigation, and Control Conference, 2000.

[14] S. Rasmussen, P. Chandler, J.W. Mitchell, C. Schumacher, A. Sparks, “Optimal vs. Heuristic Assignment of Cooperative Autonomous Unmanned Air Vehicles”, Austin, TX, AIAA Guidance, Navigation, and Control Conference, August 2003.

[15] G. Chen, J.B. Cruz, Jr., “Genetic Algorithm for Task Allocation in UAV Cooperative Control”, Austin, TX, AIAA Guidance, Navigation, and Control Conference, August 2003.

[16] P.R. Chandler, M. Pachter, S.R. Rasmussen, C. Schumacher, “Multiple Task Assignment for a UAV Team”, Monterey, CA, AIAA Guidance, Navigation, and Control Conference, August 2002.

[17] J. Pike, “Iraqi Air Defense Equipment”, GlobalSecurity.org, December 2002.

[18] S.J. Rasmussen, C. Schumacher, P.R. Chandler, “Investigation of Single vs. Multiple Task Tour Assignments for UAV Cooperative Control”, Monterey, CA, AIAA Guidance, Navigation, and Control Conference, August 2002.

[19] J.W. Curtis, R. Murphey, “Simultaneous Area Search and Task Assignment for a Team of Cooperative Agents”, Austin, TX, AIAA Guidance, Navigation, and Control Conference, August 2003.

[20] M.L. Baum, K.M. Passino, “A Search-Theoretic Approach to Cooperative Control for Uninhabited Air Vehicles”, Monterey, CA, AIAA Guidance, Navigation, and Control Conference, August 2002.

[21] G.L. Slater, “Cooperation Between UAVs in a Search and Destroy Mission”, Austin, TX, AIAA Guidance, Navigation, and Control Conference, August 2003.

[22] P.R. Chandler, M. Pachter, “Hierarchical Control for Autonomous Teams”, Montreal, Canada, AIAA Guidance, Navigation, and Control Conference, August 2001.

[23] S. Rasmussen, J.W. Mitchell, C. Schulz, C. Schumacher, P. Chandler, “A Multiple UAV Simulation for Researchers”, Austin, TX, AIAA Modeling and Simulation Technologies Conference, August 2003.

[24] D.Enns, D. Bugajski, S. Pratt, “Guidance and Control for Cooperative Search”, Anchorage, AK, American Control Conference, May 2002.

[25] G.Hui, http://www.ece.northwestern.edu/~guanghui/Transportation/spt/section3_1, Northwestern University, Chicago, IL, 1995.

[26] M.M. Akbar, E.G. Manning, G.C. Shoja, S. Khan, "Heuristic Solutions for the Multiple-Choice Multi-Dimension Knapsack Problem", Victoria, BC, Canada, Department of CD, PANDA Lab, Uvic, 2002.

[27] M.G. Kay, http://www.ie.ncsu.edu/kay/matlog/, "Matlog: Logistics Engineering MATLAB Toolbox", Department of Industrial Engineering, North Carolina State University, Raleigh, NC, 2003.

[28] B. Seanor, "Flight Testing of a Remotely Piloted Vehicle for Aircraft Parameter Estimation Purposes", Dissertation West Virginia University, MAE Department, Morgantown, WV, 2002.

[29] J. Roskam, "Airplane Flight Dynamics and Automatic Flight Controls", Design, Analysis and Reseach Corporation, Lawrence, KS, 1995, pgs. 1-34.

[30] B.L. Stevens, F.L. Lewis, "Aircraft Control and Simulation", John Wiley & Sons, Inc., New York, NY, 1992. pgs. 94, 103-109.

[31] M.O. Rauw, http://home.wanadoo.nl/dutchroll/author.html, "The Flight Dynamics and Control Toolbox", BL, Haarlem, Netherlands1992.

Appendix A

Path-Planning and Task Allocation MATLAB Files

Path Generation Related Functions

Vrn_Diag_gen

```
%Authored by Zachary Spritzer and Matthew Lechliter

function [all_pos,all_lines_x,all_lines_y,all_costs]=vrn_diag_gen(UAVS,TARGETS,ZONES,THREATS)

%INPUTS:
%
%UAVS - is a 4xn matrix where n is number of UAVs, the first row is the
%initial x position of the UAVs, the second row is the initial y position
%of the UAVs, the third row is the initial altitude of the UAVs, and
%the fourth row is the intial Velocity of the UAVs.
%
%TARGETS - is a 2xn matrix where n is the number of Targets, the first row
%is the x position of the targets and the second row is the y position of
%the targets.
%
%ZONES - is a 3xn matrix where n is the number of No-Fly Zones, the first
%row is the x position of the no-fly zones, the second row is the y
%position of the no-fly zones, and the third row is the radius or range of
%the no-fly zones.
%
%THREATS - is a 4xn matrix where n is the number of Threats, the first row
%is the x position of the threats, the second row is the y position of the
%threats, the third row is the range of the threats, and the fourth row is
%the level of danger of the threats.
%
%OUTPUTS:
%
%all_pos - is a 2xn matrix where n is the number of unique voronoi points,
%uav points, and target points.  Where the first row is the x position and
%the second row is the y position of all of these unique points.
%
%all_lines_x - is a 2xn matrix where n is the number of all of the lines
%for the voronoi, uavs, and targets.  The first row is the ending point's
%x position for the nth line and the second row is the starting point's
%x position for the nthe line.
%
%all_lines_y - is a 2xn matrix where n is the number of all of the lines
%for the voronoi, uavs, and targets.  The first row is the ending point's
%y position for the nth line and the second row is the starting point's
%y position for the nthe line.
%
%all_costs - is a 1xn row where n is the number of all of the lines
%for the voronoi, uavs, and targets.  This row is the costs for all of the
%lines of all_lines_x and all_lines_y

max_x=max([TARGETS(1,:),UAVS(1,:),ZONES(1,:),THREATS(1,:)])+25;
min_x=min([TARGETS(1,:),UAVS(1,:),ZONES(1,:),THREATS(1,:)])-25;
max_y=max([TARGETS(2,:),UAVS(2,:),ZONES(2,:),THREATS(2,:)])+25;
min_y=min([TARGETS(2,:),UAVS(2,:),ZONES(2,:),THREATS(2,:)])-25;

VRNPTS=[ZONES([1,2],:) THREATS([1,2],:) ...
```

```
   [(((max_y-min_y)*[1:4]/4)+min_y);(min_x)*ones(1,4)] ...
   [(((max_y-min_y)*[1:4]/4)+min_y);(min_x)*ones(1,4)] ...
   [(((max_x-min_x)*[1:4]/4)+min_x);(min_y)*ones(1,4)] ...
   [(((max_x-min_x)*[1:4]/4)+min_x);(max_y)*ones(1,4)]];


[vx,vy] = voronoi(VRNPTS(1,:),VRNPTS(2,:));

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Taking unique numbers from vx and vy
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
[vxyn]= 1e-6*unique(round(1e6*[vx(:),vy(:)]),'rows');


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Connecting UAV's into voronoi
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
[line_cost_uav,uavx,uavy]=connect_vrn(vxyn,UAVS([1,2],:));

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Connecting the targets into the voronoi
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
[line_cost_targ,targx,targy]=connect_vrn(vxyn,TARGETS([1,2],:));

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Generation for voronoi line costs
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
nvlines=size(vx,2);
line_cost_vrn=zeros(1,nvlines);
for i=1:nvlines,
   line_cost_vrn(1,i)=sqrt((vx(1,i)-vx(2,i))^2+(vy(1,i)-vy(2,i))^2);
end
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Stacking unique positions, lines for x and y, and costs of those lines
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
all_pos=[UAVS([1,2],:) vxyn(:,[1,2])' TARGETS([1,2],:)];
all_lines_x=[uavx([1,2],:) vx([1,2],:) targx([1,2],:)];
all_lines_y=[uavy([1,2],:) vy([1,2],:) targy([1,2],:)];
all_costs=[line_cost_uav(1,:) line_cost_vrn(1,:) line_cost_targ(1,:)];
```

Voronoi

```
function [vxx,vy] = voronoi(x,y,arg3,arg4)
%VORONOI Voronoi diagram.
%   VORONOI(X,Y) plots the Voronoi diagram for the points X,Y.
%   Cells that contain a point at infinity are unbounded and
%   are not plotted.
%
%   VORONOI(X,Y,TRI) uses the triangulation TRI instead of
%   computing it via DELAUNAY.
%
%   H = VORONOI(...,'LineSpec') plots the diagram with color and linestyle
%   specified and returns handles to the line objects created in H.
%
```

```
%   [VX,VY] = VORONOI(...) returns the vertices of the Voronoi
%   edges in VX and VY so that plot(VX,VY,'-',X,Y,'.') creates the
%   Voronoi diagram.
%
%   For the topology of the voronoi diagram, i.e. the vertices for
%   each voronoi cell, use the function VORONOIN as follows:
%
%       [V,C] = VORONOIN([X(:) Y(:)])
%
%   See also VORONOIN, DELAUNAY, CONVHULL.

%   Copyright 1984-2002 The MathWorks, Inc.
%   $Revision: 1.15 $  $Date: 2002/06/05 20:05:17 $

error(nargchk(2,4,nargin));

if nargin==2,
  tri = delaunay(x,y);
  ls = '';
elseif nargin==3,
  if isstr(arg3),
    tri = delaunay(x,y);
    ls = arg3;
  else
    tri = arg3;
    ls = '';
  end
else
  tri = arg3;
  ls = arg4;
end

% re-orient the triangles so that they are all clockwise
xt = x(tri); yt=y(tri);
ot = xt(:,1).*(yt(:,2)-yt(:,3)) + ...
     xt(:,2).*(yt(:,3)-yt(:,1)) + ...
     xt(:,3).*(yt(:,1)-yt(:,2));
bt = find(ot<0);
tri(bt,[1 2]) = tri(bt,[2 1]);

n = prod(size(x));
ntri = size(tri,1);
t = (1:ntri)';
T = sparse(tri,tri(:,[3 1 2]),t(:,ones(1,3)),n,n); % Triangle edge if T(i,j)
E = (T & T').*T; % Voronoi edge if E(i,j)

[i,j,v] = find(triu(E));
[i,j,vv] = find(triu(E'));
c1 = circle(tri(v,:),x,y);
c2 = circle(tri(vv,:),x,y);

vx = [c1(:,1) c2(:,1)].';
vy = [c1(:,2) c2(:,2)].';

if nargout<2
  if isempty(ls),
```

```
    co = get(gcf,'defaultaxescolororder');
    h = plot(vx,vy,'-',x,y,'.','color',co(1,:));
  else
    [l,c,m,msg] = colstyle(ls); error(msg)
    if isempty(m), m = '.'; end
    h = plot(vx,vy,ls,x,y,[c m]);
  end
  if ~ishold,
    view(2), axis([min(x(:)) max(x(:)) min(y(:)) max(y(:))])
  end
  if nargout==1, vxx = h; end
else
  vxx = vx;
end

function c = circle(tri,x,y)
%CIRCLE Return center and radius for circumcircles
%   C = CIRCLE(TRI,X,Y) returns a N-by-3 vector containing [xcenter(:)
%   ycenter(:) radius(:)] for each triangle in TRI.

% Reference: Watson, p32.
x = x(:); y = y(:);

x1 = x(tri(:,1)); x2 = x(tri(:,2)); x3 = x(tri(:,3));
y1 = y(tri(:,1)); y2 = y(tri(:,2)); y3 = y(tri(:,3));

% Set equation for center of each circumcircle:
%    [a11 a12;a21 a22]*[x;y] = [b1;b2] * 0.5;

a11 = x2-x1; a12 = y2-y1;
a21 = x3-x1; a22 = y3-y1;

b1 = a11 .* (x2+x1) + a12 .* (y2+y1);
b2 = a21 .* (x3+x1) + a22 .* (y3+y1);

% Solve the 2-by-2 equation explicitly
idet = a11.*a22 - a21.*a12;

% Add small random displacement to points that are either the same
% or on a line.
d = find(idet == 0);
if ~isempty(d), % Add small random displacement to points
  delta = sqrt(eps);
  x1(d) = x1(d) + delta*(rand(size(d))-0.5);
  x2(d) = x2(d) + delta*(rand(size(d))-0.5);
  x3(d) = x3(d) + delta*(rand(size(d))-0.5);
  y1(d) = y1(d) + delta*(rand(size(d))-0.5);
  y2(d) = y2(d) + delta*(rand(size(d))-0.5);
  y3(d) = y3(d) + delta*(rand(size(d))-0.5);
  a11 = x2-x1; a12 = y2-y1;
  a21 = x3-x1; a22 = y3-y1;
  b1 = a11 .* (x2+x1) + a12 .* (y2+y1);
  b2 = a21 .* (x3+x1) + a22 .* (y3+y1);
  idet = a11.*a22 - a21.*a12;
end
```

```
idet = 0.5 ./ idet;

xcenter = ( a22.*b1 - a12.*b2) .* idet;
ycenter = (-a21.*b1 + a11.*b2) .* idet;

radius = (x1-xcenter).^2 + (y1-ycenter).^2;

c = [xcenter ycenter radius];
```

Connect_Vrn

```
%Authored by Zachary Spritzer and Matthew Lechliter

function [line_cost_uav,uavx,uavy]=connect_vrn(vxyn,UAVS)

%Inputs:
%
%vxyn - is a nx2 matrix with first column defining all of the unique x
%positions of the voronoi diagram or grid and the second column defining
%all of the unique y positions of the voronoi diagram or grid.
%
%UAVS - is a 2xn matrix with the first row defining the x position of the
%UAV and the second row defining the y position of the UAV.
%
%Outputs:
%
%line_cost_uav - is a vector containing the cost of the lines of connecting
%the UAV's into the voronoi diagram or grid
%
%uavx - is a 2xn matrix with first row defining ending point and second row
%defining starting point for the x coordinates.
%
%uavy - is a 2xn matrix with first row defining ending point and second row
%defining starting point for the y coordinates.
nuav=size(UAVS,2);
nvxynpts=size(vxyn,1);
du=zeros(1,nvxynpts-1);
uavx=zeros(2,nuav*3);
uavy=zeros(2,nuav*3);
line_cost_uav=zeros(1,nuav*3);
for k=1:nuav,
   for j=2:nvxynpts,
      du(1,j-1)=sqrt((UAVS(1,k)-vxyn(j,1))^2+(UAVS(2,k)-vxyn(j,2))^2);
   end
   mdu=sort(du,2);
   for i=1:3,
      mdu_loc=find(du==mdu(1,i));
      uavx(1,3*(k-1)+i)=vxyn(mdu_loc+1,1);
      uavy(1,3*(k-1)+i)=vxyn(mdu_loc+1,2);
      uavx(2,3*(k-1)+i)=UAVS(1,k);
      uavy(2,3*(k-1)+i)=UAVS(2,k);
      line_cost_uav(1,3*(k-1)+i)=mdu(1,i);
   end
end
```

Path Selection Related Functions

Cheapest_Paths

```
%Authored by Zachary Spritzer and Matthew Lechliter

function
[stored_paths,totalcost]=cheapest_paths(all_pos,all_lines_x,all_lines_y,all_costs,UAVS,TARGETS,ZONE
S,THREATS)
%
%INPUTS:
%
%all_pos - is a 2xn matrix where n is the number of unique voronoi points,
%uav points, and target points.  Where the first row is the x position and
%the second row is the y position of all of these unique points.
%
%all_lines_x - is a 2xn matrix where n is the number of all of the lines
%for the voronoi, uavs, and targets.  The first row is the ending point's
%x position for the nth line and the second row is the starting point's
%x position for the nthe line.
%
%all_lines_y - is a 2xn matrix where n is the number of all of the lines
%for the voronoi, uavs, and targets.  The first row is the ending point's
%y position for the nth line and the second row is the starting point's
%y position for the nthe line.
%
%all_costs - is a 1xn row where n is the number of all of the lines
%for the voronoi, uavs, and targets.  This row is the costs for all of the
%lines of all_lines_x and all_lines_y.
%
%UAVS - is a 4xn matrix where n is number of UAVs, the first row is the
%initial x position of the UAVs, the second row is the initial y position
%of the UAVs, the third row is the initial altitude of the UAVs, and
%the fourth row is the intial Velocity of the UAVs.
%
%TARGETS - is a 2xn matrix where n is the number of Targets, the first row
%is the x position of the targets and the second row is the y position of
%the targets.
%
%ZONES - is a 3xn matrix where n is the number of No-Fly Zones, the first
%row is the x position of the no-fly zones, the second row is the y
%position of the no-fly zones, and the third row is the radius or range of
%the no-fly zones.
%
%THREATS - is a 4xn matrix where n is the number of Threats, the first row
%is the x position of the threats, the second row is the y position of the
%threats, the third row is the range of the threats, and the fourth row is
%the level of danger of the threats.
%
%OUTPUTS:
%
%stored_paths - is a mxn matrix where m is the number of uavs times the
```

```
%number of targets and n is the length of the longest path.  The first row
%being the first path for the first uav and the last row being the last
%path for the last uav. The paths are output by node numbers coming from
%the implementation of dijkstra's algorithm.
%
%totalcost - is a mxn matrix where m is the number of uavs and n is the
%number of possible paths for each uav.  The element (m,n) of this matrix
%is the cost for the mth uav to take the nth path.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Making THC matrix for dijkstra's algorithm
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
[THC]=set_THC(all_pos,all_lines_x,all_lines_y,all_costs);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Cost Assignment for all lines
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
[THC]= c_assign(all_pos,THC,ZONES,THREATS);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Adding the reverse of the THC matrix onto the end, so that the
%reverse of the lines is possible
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
THC=[THC(:,[1,2,3]); THC(:,[2,1,3])];

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Implementing Dijkstra's algorithm
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
nuav=size(UAVS,2);
ntarg=size(TARGETS,2);
A = list2adj(THC);
totalcost=zeros(nuav,ntarg);
for i=1:nuav,
   for j=1:ntarg,
      [totalcost(i,j),path] = dijk(A,i,size(all_pos,2) - j + 1);
      stored_paths((i-1)*ntarg+j,[1:size(path,2)])=path(1,[1:size(path,2)]);
   end
end
```

Set_THC

```
%Authored by Zachary Spritzer, Matthew Lechliter, and Elena Lucci

function [THC]=set_THC(all_pos,all_lines_x,all_lines_y,all_costs)
%
%INPUTS:
%
%all_pos - is a 2xn matrix where n is the number of unique voronoi points,
%uav points, and target points.  Where the first row is the x position and
%the second row is the y position of all of these unique points.
%
%all_lines_x - is a 2xn matrix where n is the number of all of the lines
%for the voronoi, uavs, and targets.  The first row is the ending point's
%x position for the nth line and the second row is the starting point's
```

```
%x position for the nthe line.
%
%all_lines_y - is a 2xn matrix where n is the number of all of the lines
%for the voronoi, uavs, and targets.  The first row is the ending point's
%y position for the nth line and the second row is the starting point's
%y position for the nthe line.
%
%all_costs - is a 1xn row where n is the number of all of the lines
%for the voronoi, uavs, and targets.  This row is the costs for all of the
%lines of all_lines_x and all_lines_y.
%
%OUTPUTS:
%
%THC - is a nx3 matrix where n is the number of possible lines to be chosen
%the first column is the tail of the line or starting point, the second
%column is the head of the line or the ending point, and the third column
%is the cost of the line. With updated costs due to no-fly zones and
%threats.



THC=zeros(size(all_lines_x,2),3);
THC(:,3)=all_costs(:);
for i=1:(2*size(all_lines_x,2))
   P=(round(all_pos(1,:)*100)== round(all_lines_x(i)*100)) &
(round(all_pos(2,:)*100)==round(all_lines_y(i)*100));
   if  any(P)
      num=find(P);
      if (rem(i,2))~=0
         bz=((fix(i./2))+1);
         THC(bz,1)=num;
      else THC((i/2),2)=num;
      end
   else
      if (rem(i,2))~=0
         tz=(fix((i./2))+1);
         THC(tz,1)=i;
      else THC((i/2),2)=i;
      end
   end
end
```

C_assign

```
%Authored by Zachary Spritzer and Matthew Lechliter

function [THC]= c_assign(all_pos,THC,ZONES,THREATS)
%
%INPUTS:
%
%all_pos - is a 2xn matrix where n is the number of unique voronoi points,
%uav points, and target points.  Where the first row is the x position and
%the second row is the y position of all of these unique points.
```

```
%
%THC - is a nx3 matrix where n is the number of possible lines to be chosen
%the first column is the tail of the line or starting point, the second
%column is the head of the line or the ending point, and the third column
%is the cost of the line.
%
%ZONES - is a 3xn matrix where n is the number of No-Fly Zones, the first
%row is the x position of the no-fly zones, the second row is the y
%position of the no-fly zones, and the third row is the radius or range of
%the no-fly zones.
%
%THREATS - is a 4xn matrix where n is the number of Threats, the first row
%is the x position of the threats, the second row is the y position of the
%threats, the third row is the range of the threats, and the fourth row is
%the level of danger of the threats.
%
%OUTPUTS:
%
%THC - is a nx3 matrix where n is the number of possible lines to be chosen
%the first column is the tail of the line or starting point, the second
%column is the head of the line or the ending point, and the third column
%is the cost of the line. With updated costs due to no-fly zones and
%threats.
szthc=size(THC,1);
nzones=size(ZONES,2);
nthrts=size(THREATS,2);

for i=1:szthc,
   start=THC(i,1);finish=THC(i,2);
   SF=sqrt(((all_pos(1,finish)-all_pos(1,start))^2)+((all_pos(2,finish)-all_pos(2,start))^2));
   for j=1:nzones,
      SC=sqrt(((ZONES(1,j)-all_pos(1,start))^2)+((ZONES(2,j)-all_pos(2,start))^2));
      FC=sqrt(((ZONES(1,j)-all_pos(1,finish))^2)+((ZONES(2,j)-all_pos(2,finish))^2));
      SN=(SC^2+SF^2-FC^2)/(2*SF);
      if SN<SF & SN>0,PC=sqrt(SC^2-SN^2);
      else
         if SC<FC,PC=SC;
         else
            PC=FC;
         end
      end
      if PC < ZONES(3,j),THC(i,3)=1e30*THC(i,3);
      end
   end
   for j=1:nthrts,
      SC=sqrt(((THREATS(1,j)-all_pos(1,start))^2)+((THREATS(2,j)-all_pos(2,start))^2));
      FC=sqrt(((THREATS(1,j)-all_pos(1,finish))^2)+((THREATS(2,j)-all_pos(2,finish))^2));
      SN=(SC^2+SF^2-FC^2)/(2*SF);
      if SN<SF & SN>0,PC=sqrt(SC^2-SN^2);
      else
         if SC<FC,PC=SC;
         else
            PC=FC;
         end
      end
      if PC < THREATS(3,j),THC(i,3)=(THREATS(4,j)*100)+THC(i,3);
```

```
      end
   end
end
```

Dijk

```
function [D,P] = dijk(A,s,t)
%DIJK Shortest paths from nodes 's' to nodes 't' using Dijkstra algorithm.
% [D,P] = dijk(A,s,t)
%    A = n x n node-node weighted adjacency matrix of arc lengths
%       (Note: A(i,j) = 0   => Arc (i,j) does not exist;
%              A(i,j) = NaN => Arc (i,j) exists with 0 weight)
%    s = FROM node indices
%      = [] (default), paths from all nodes
%    t = TO node indices
%      = [] (default), paths to all nodes
%    D = |s| x |t| matrix of shortest path distances from 's' to 't'
%      = [D(i,j)], where D(i,j) = distance from node 'i' to node 'j'
%    P = |s| x n matrix of predecessor indices, where P(i,j) is the
%       index of the predecessor to node 'j' on the path from 's(i)' to
%       'j',where P(i,i) = 0 and P(i,j) = NaN is 'j' not on path to 's(i)'
%       (use PRED2PATH to convert P to paths)
%      = path from 's' to 't', if |s| = |t| = 1
%
%  (If A is a triangular matrix, then computationally intensive node
%   selection step not needed since graph is acyclic (triangularity is a
%   sufficient, but not a necessary, condition for a graph to be acyclic)
%   and A can have non-negative elements)
%
%  (If |s| >> |t|, then DIJK is faster if DIJK(A',t,s) used, where D is now
%   transposed and P now represents successor indices)
%
%  (Based on Fig. 4.6 in Ahuja, Magnanti, and Orlin, Network Flows,
%   Prentice-Hall, 1993, p. 109.)

% Copyright (c) 1994-2002 by Michael G. Kay
% Matlog Version 6 19-Sep-2002

% Input Error Checking ************************************************
error(nargchk(1,3,nargin))

[n,cA] = size(A);

if nargin < 2 | isempty(s), s = (1:n)'; else s = s(:); end
if nargin < 3 | isempty(t), t = (1:n)'; else t = t(:); end

if ~any(any(tril(A) ~= 0))       % A is upper triangular
  isAcyclic = 1;
elseif ~any(any(triu(A) ~= 0))   % A is lower triangular
  isAcyclic = 2;
else                             % Graph may not be acyclic
  isAcyclic = 0;
end
```

```
if n ~= cA
  error('A must be a square matrix');
elseif ~isAcyclic & any(any(A < 0))
  error('A must be non-negative');
elseif any(s < 1 | s > n)
  error(['"s" must be an integer between 1 and ',num2str(n)]);
elseif any(t < 1 | t > n)
  error(['"t" must be an integer between 1 and ',num2str(n)]);
end
% End (Input Error Checking) ************************************************

A = A';    % Use transpose to speed-up FIND for sparse A

D = zeros(length(s),length(t));
if nargout > 1, P = NaN*ones(length(s),n); end

for i = 1:length(s)
  j = s(i);

  Di = Inf*ones(n,1); Di(j) = 0;

  isLab = logical(zeros(length(t),1));
  if isAcyclic ==  1
    nLab = j - 1;
  elseif isAcyclic == 2
    nLab = n - j;
  else
    nLab = 0;
    UnLab = 1:n;
    isUnLab = logical(ones(n,1));
  end

  if nargout > 1, P(i,s(i)) = 0; end  % Change from NaN to indicate no pred

  while nLab < n & ~all(isLab)
    if isAcyclic
      Dj = Di(j);
    else  % Node selection
      [Dj,jj] = min(Di(isUnLab));
      j = UnLab(jj);
      UnLab(jj) = [];
      isUnLab(j) = 0;
    end

    nLab = nLab + 1;
    if length(t) < n, isLab = isLab | (j == t); end

    [jA,kA,Aj] = find(A(:,j));
    Aj(isnan(Aj)) = 0;

    if isempty(Aj), Dk = Inf; else Dk = Dj + Aj; end

    if nargout > 1, P(i,jA(Dk < Di(jA))) = j; end
    Di(jA) = min(Di(jA),Dk);

    if isAcyclic == 1       % Increment node index for upper triangular A
```

```
      j = j + 1;
    elseif isAcyclic == 2   % Decrement node index for lower triangular A
      j = j - 1;
    end
  end
  D(i,:) = Di(t)';
end

if nargout > 1 & length(s) == 1 & length(t) == 1
  P = pred2path(P,s,t);
End
```

Additional Dijkstra Functions

```
function [i,j,c] = adj2list(A)
%ADJ2LIST Node-node weighted adjacency matrix to arc list representation.
%    IJC = adj2list(A)
% [i,j,c] = adj2list(A)
%    A = m x m node-node weighted adjacency matrix of arc lengths
%   IJC = n x 2-3 matrix arc list [i j c], where
%    i = n-element vector of arc tails nodes
%    j = n-element vector of arc head nodes
%    c = n-element vector of arc weights
%
% Note: All A(i,j) = A(j,i) => [i -j c] (symmetric A)
%       A(i,j) = 0   => Arc (i,j) does not exist
%       A(i,j) = NaN => Arc (i,j) exists with 0 weight
%       Wrapper for [i,j,c] = FIND(C); c(ISNAN(c)) = 0)
%
% See also LIST2INCID, LIST2ADJ, and ADJ2INCID

% Copyright (c) 1994-2002 by Michael G. Kay
% Matlog Version 6 19-Sep-2002

% Input Error Checking ************************************************************
[rA,cA] = size(A);
if rA ~= cA
  error('"A" must be a square matrix.');
end
% End (Input Error Checking) ******************************************************

if all(all(triu(A)==tril(A)')), A = triu(A); issym = 1; else issym = 0; end

[i,j,c] = find(A);
if issym, j = -j; end
c(isnan(c)) = 0;

if nargout == 1
  i = [i j c];
end



function y = isint(x,TolInt)
%ISINT True for integer elements (within tolerance).
%      y = isint(x,TolInt)
```

```
%       = abs(x-round(x)) < TolInt
% TolInt = integer tolerance
%       = [0.01*sqrt(eps)], default

% Copyright (c) 1994-2002 by Michael G. Kay
% Matlog Version 6 19-Sep-2002

% Input Error Checking ************************************************************
error(nargchk(1,2,nargin));
if nargin < 2 | isempty(TolInt), TolInt = 0.01*sqrt(eps); end
% End (Input Error Checking) ******************************************************

y = abs(x-round(x)) < TolInt;



function A = list2adj(IJC,m,spA)
%LIST2ADJ Arc list to node-node weighted adjacency matrix representation.
%    A = list2adj(IJC,m,spA)
%  IJC = n x 2-5 matrix arc list [i j c u l], where
%    i = n-element vector of arc tails nodes
%    j = n-element vector of arc head nodes
%    c = (optional) n-element vector of arc costs, where n = number of arcs
%      = (default) ONES(n,1)
%    u = (optional) ignored
%    l = (optional) ignored
%    m = (optional) scalar size of A if greater than max{max(i),max(abs(j))}
%  spA = (optional) make A sparse matrix if n <= spA x m x m
%      = 1, always make A sparse
%      = 0.1 (default), A sparse if 10% arc density
%      = 0, always make A full matrix
%    A = m x m node-node weighted adjacency matrix
%
% Transforms: If j(k) > 0, then [i(k) j(k) c(k)] -> A[i(k),j(k)]  = c(k)
%             If j(k) < 0, then [i(k) j(k) c(k)] -> A[i(k),-j(k)] = c(k) and
%                                                   A[-j(k),i(k)] = c(k)
%
% Note: Weights of any duplicate arcs added together in A
%      c(k) = 0 => A(i(k),j(k)) = NaN
%      Wrapper for c(c==0) = NaN; A = SPARSE(i,j,c,m,m);
%
% See also LIST2INCID, ADJ2LIST, and ADJ2INCID

% Copyright (c) 1994-2002 by Michael G. Kay
% Matlog Version 6 19-Sep-2002

% Input Error Checking ************************************************************
error(nargchk(1,3,nargin))

[n,cIJC] = size(IJC);
if cIJC < 2 | cIJC > 5, error('IJC must be a 2-3 column matrix.'), end

[i,j,c] = mat2vec(IJC);
if isempty(c), c = ones(n,1); end

jsgn = sign(j); j = abs(j);
```

```
minIJ = min(min([i j]));
if isempty(minIJ) | minIJ < 1 | any(~isint(i)) | any(~isint(j))
  error('All elements of "i" and "j" must be nonzero integers.');
end

if nargin < 2 | isempty(m)
  m = max(max([i j]));
elseif length(m(:)) ~= 1 | ~isint(m) | m < max(max([i j]))
  error('"n" must be >= max{max(i),max(abs(j))}.');
end

if nargin < 3 | isempty(spA)
  spA = 0.1;
elseif length(spA(:)) ~= 1 | spA < 0
  error('"spA" must be non-negative scalar.');
end
% End (Input Error Checking) ************************************************

if any(jsgn < 0)                                        % Add elements from undirected arcs
  jsgn(jsgn < 0 & i == j) = 1;
  i = [i; j(jsgn < 0)];
  j = [j; i(jsgn < 0)];
  c = [c; c(jsgn < 0)];
end

c(c==0) = NaN;
A = sparse(i,j,c,m,m);

if n > spA * m * m, A = full(A); end



function varargout = mat2vec(X)
%MAT2VEC Convert columns of matrix to vectors.
% [X(:,1),X(:,2),...] = mat2vec(X)
%
% (Additional output vectors assigned as empty)

% Copyright (c) 1994-2002 by Michael G. Kay
% Matlog Version 6 19-Sep-2002

% Input Error Checking ******************************************************
if ~isnumeric(X)
  error('X must be numeric.')
end
% End (Input Error Checking) ************************************************

varargout = cell(1,max(1,nargout));
X = num2cell(X,1);
varargout(1,1:min(nargout,size(X,2))) = X(1,1:min(nargout,size(X,2)));



function rte = pred2path(P,s,t)
%PRED2PATH Convert predecessor indices to shortest paths from node 's' to 't'.
%   rte = pred2path(P,s,t)
```

```
%     P = |s| x n matrix of predecessor indices (from DIJK)
%     s = FROM node indices
%       = [] (default), paths from all nodes
%     t = TO node indices
%       = [] (default), paths to all nodes
%   rte = |s| x |t| cell array of paths (or routes) from 's' to 't', where
%         rte{i,j} = path from s(i) to t(j)
%                  = [], if no path exists from s(i) to t(j)
%
% (Used with output of DIJK)

% Copyright (c) 1994-2002 by Michael G. Kay
% Matlog Version 6 19-Sep-2002

% Input Error Checking ************************************************
error(nargchk(1,3,nargin));

[rP,n] = size(P);

if nargin < 2 | isempty(s), s = (1:n)'; else s = s(:); end
if nargin < 3 | isempty(t), t = (1:n)'; else t = t(:); end

if any(P < 0 | P > n)
  error(['Elements of P must be integers between 1 and ',num2str(n)]);
elseif any(s < 1 | s > n)
  error(['"s" must be an integer between 1 and ',num2str(n)]);
elseif any(t < 1 | t > n)
  error(['"t" must be an integer between 1 and ',num2str(n)]);
end
% End (Input Error Checking) ************************************************

rte = cell(length(s),length(t));

[ans,idxs] = find(P==0);

for i = 1:length(s)
%   if rP == 1
%     si = 1;
%   else
%     si = s(i);
%     if si < 1 | si > rP
%       error('Invalid P matrix.')
%     end
%   end
  si = find(idxs == s(i));
  for j = 1:length(t)
    tj = t(j);
    if tj == s(i)
      r = tj;
    elseif P(si,tj) == 0
      r = [];
    else
      r = tj;
      while tj ~= 0
        if tj < 1 | tj > n
          error('Invalid element of P matrix found.')
```

```
          end
          r = [P(si,tj) r];
          tj = P(si,tj);
        end
        r(1) = [];
      end
      rte{i,j} = r;
    end
end

if length(s) == 1 & length(t) == 1
  rte = rte{:};
end

%rte = t;
while 0%t ~= s
  if t < 1 | t > n | round(t) ~= t
    error('Invalid "pred" element found prior to reaching "s"');
  end
  rte = [P(t) rte];
  t = P(t);
end
```

Path Refinement Related Functions

Path_Shrtng

```
%Authored by Zachary Spritzer and Matthew Lechliter

function
[Shortened_Paths_x,Shortened_Paths_y,totalcost]=path_shrtng(stored_paths,all_pos,ZONES,THREATS,m
in_turn,split_seg,nuav,ntarg,HEADING_ANGLE)

%INPUTS:
%
%stored_paths - is a mxn matrix where m is the number of uavs times the
%number of targets and n is the length of the longest path.  The first row
%being the first path for the first uav and the last row being the last
%path for the last uav. The paths are output by node numbers coming from
%the implementation of dijkstra's algorithm.
%
%all_pos - is a 2xn matrix where n is the number of unique voronoi points,
%uav points, and target points.  Where the first row is the x position and
%the second row is the y position of all of these unique points.
%
%ZONES - is a 3xn matrix where n is the number of No-Fly Zones, the first
%row is the x position of the no-fly zones, the second row is the y
%position of the no-fly zones, and the third row is the radius or range of
%the no-fly zones.
%
%THREATS - is a 4xn matrix where n is the number of Threats, the first row
%is the x position of the threats, the second row is the y position of the
%threats, the third row is the range of the threats, and the fourth row is
```

```
%the level of danger of the threats.
%
%min_turn - minimum turning radius for the UAVs
%
%split_seg - number of segments to Split the voronoi lines into for the
%purpose of a more near-optimal solution
%
%nuav - number of UAVs
%
%ntarg - number of targets

%OUTPUTS:
%
%Shortened_Paths - is a nxmx2 matrix where n is the length of the longest
%path and m is the number of UAVs multiplied by the number of targets.
%The element (nxmx1) x position of the mth uav at point n.  The element
%(nxmx2) y position of the mth uav at point n.
%
%totalcost - is a mxn matrix where m is the number of uavs and n is the
%number of possible paths for each uav.  The element (m,n) of this matrix
%is the cost for the mth uav to take the nth path.
%
%Stored_Pos - is a nxmx2 matrix where n is the length of the longest
%path and m is the number of UAVs multiplied by the number of targets.
%The element (nxmx1) x position of the mth uav at point n.  The element
%(nxmx2) y position of the mth uav at point n.


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Splitting the voronoi lines into more segments for the purpose of a more
%near-optimal solution
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
szpths=size(stored_paths,2);

split_vect=[(0:(1/split_seg):(1- 1/split_seg))]';

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Finding the corresponding x and y coordinates
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Stored_Pos_x=ones(szpths,nuav*ntarg);
Stored_Pos_y=ones(szpths,nuav*ntarg);
stored_paths(:,szpths+1)=0;
for i=1:nuav*ntarg,
   mnz=min(find(stored_paths(i,:)==0));
   Stored_Pos_x(1:mnz-1,i)=all_pos(1,stored_paths(i,1:mnz-1))';
   Stored_Pos_y(1:mnz-1,i)=all_pos(2,stored_paths(i,1:mnz-1))';
   Stored_Pos_x(mnz:end,i)=ones((szpths-mnz+1),1)*all_pos(1,stored_paths(i,mnz-1))';
   Stored_Pos_y(mnz:end,i)=ones((szpths-mnz+1),1)*all_pos(2,stored_paths(i,mnz-1))';

end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Stored_Pos_x_new=ones((((szpths-1)*split_seg)+1),nuav*ntarg);
Stored_Pos_y_new=ones((((szpths-1)*split_seg)+1),nuav*ntarg);
for k=1:nuav*ntarg,
      j=1;
```

```
   for i=1:(szpths -1),
       Stored_Pos_x_new([j:(j + (split_seg -1))],k)=
ones(split_seg,1)*Stored_Pos_x(i,k)+split_vect*(Stored_Pos_x(i+1,k)-Stored_Pos_x(i,k));
       Stored_Pos_y_new([j:(j + (split_seg -1))],k)=
ones(split_seg,1)*Stored_Pos_y(i,k)+split_vect*(Stored_Pos_y(i+1,k)-Stored_Pos_y(i,k));
       j=j+ split_seg;
   end
   Stored_Pos_x_new((((szpths-1)*split_seg)+1),k)=Stored_Pos_x(szpths,k);
   Stored_Pos_y_new((((szpths-1)*split_seg)+1),k)=Stored_Pos_y(szpths,k);
end


Shortened_Paths_x_end=ones(500,1)*Stored_Pos_x(szpths,:);
Shortened_Paths_y_end=ones(500,1)*Stored_Pos_y(szpths,:);
Shortened_Paths_x=[Stored_Pos_x_new;Shortened_Paths_x_end];
Shortened_Paths_y=[Stored_Pos_y_new;Shortened_Paths_y_end];



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Shortening the paths
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
for i=1:nuav*ntarg,

[Shortened_Paths_x(:,i),Shortened_Paths_y(:,i)]=shorten_paths(Shortened_Paths_x(:,i),Shortened_Paths_y
(:,i),ZONES,THREATS,Stored_Pos_x(:,i),Stored_Pos_y(:,i));
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Putting fillets into the shortened paths
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
for i=1:nuav*ntarg,

[Shortened_Paths_x(:,i),Shortened_Paths_y(:,i)]=fillet_path([Shortened_Paths_x(:,i),Shortened_Paths_y(:,i
)],min_turn);
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Adding initial path based on heading angle
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
for i=1:nuav,
   for j=1:ntarg,
      [Shortened_Paths_x(:,((i-1)*ntarg)+j),Shortened_Paths_y(:,((i-1)*ntarg)+j)]=...
         heading_angle_paths([Shortened_Paths_x(:,((i-1)*ntarg)+j),Shortened_Paths_y(:,((i-
1)*ntarg)+j)],min_turn,HEADING_ANGLE(i,1),72);
   end
end


Shortened_Paths_x_old=Shortened_Paths_x;
Shortened_Paths_y_old=Shortened_Paths_y;
Shortened_Paths_x=[];
Shortened_Paths_y=[];
for j=1:size(Shortened_Paths_x_old,1)-1,
```

```
   if Shortened_Paths_x_old(j,:)==Shortened_Paths_x_old(j+1,:) &
Shortened_Paths_y_old(j,:)==Shortened_Paths_y_old(j+1,:),
       Shortened_Paths_x(j,:)=Shortened_Paths_x_old(j,:);
       Shortened_Paths_y(j,:)=Shortened_Paths_y_old(j,:);
       break
   else
      Shortened_Paths_x(j,:)=Shortened_Paths_x_old(j,:);
      Shortened_Paths_y(j,:)=Shortened_Paths_y_old(j,:);
   end
end


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Updating the Costs
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%s
zsp_perm=size(Shortened_Paths_x,2);
permcost=zeros(nuav*ntarg,1);


for z=1:szsp_perm,
   [permcost(z,1)]=update_cost([Shortened_Paths_x(:,z),Shortened_Paths_y(:,z)],THREATS);
end
totalcost=reshape(permcost,ntarg,nuav)';
```

Shorten_Paths

```
%Authored by Zachary Spritzer and Matthew Lechliter

function [shr_x,shr_y]=shorten_paths(sp_x,sp_y,Z,T,spo_x,spo_y)

%INPUTS:
%
%sp - is a nxmx2 matrix where n is the length of the longest
%path and m is the number of UAVs.  The element (nxmx1) x position of the
%mth uav at point n.  The element (nxmx2) y position of the mth uav at
%point n.
%
%Z - is a 3xn matrix where n is the number of No-Fly Zones, the first
%row is the x position of the no-fly zones, the second row is the y
%position of the no-fly zones, and the third row is the radius or range of
%the no-fly zones.
%
%T - is a 4xn matrix where n is the number of Threats, the first row
%is the x position of the threats, the second row is the y position of the
%threats, the third row is the range of the threats, and the fourth row is
%the level of danger of the threats.
%
%spo - is a nxmx2 matrix where n is the length of the longest
%path and m is the number of UAVs.  The element (nxmx1) x position of the
%mth uav at point n.  The element (nxmx2) y position of the mth uav at
%point n. This matrix is the original matrix without the voronoi segements
%split up.
%
```

```
%OUTPUTS:
%
%shr - is a nxmx2 matrix where n is the length of the longest
%path and m is the number of UAVs.  The element (nxmx1) x position of the
%mth uav at point n.  The element (nxmx2) y position of the mth uav at
%point n.
spo=[spo_x,spo_y];
sp=[sp_x,sp_y];
SC=0;FC=0;SF=0;SN=0;
for j=1:size(T,2),
   PC=[];
   for i=1:size(spo,1)-1,
      SC=sqrt(((T(1,j)-spo(i,1))^2)+((T(2,j)-spo(i,2))^2));
      FC=sqrt(((T(1,j)-spo(i+1,1))^2)+((T(2,j)-spo(i+1,2))^2));
      SF=sqrt(((spo(i+1,1)-spo(i,1))^2)+((spo(i+1,2)-spo(i,2))^2));
      SN=(SC^2+SF^2-FC^2)/(2*SF);
      if SN<SF & SN>0
         PC(i)=sqrt(SC^2-SN^2);
      else
         if SC<FC
            PC(i)=SC;
         else
            PC(i)=FC;
         end
      end
      mPC=min(PC);
      if  mPC< T(3,j),
         T(3,j)=mPC*.995;
      end
   end
end

ZT=[Z([1:3],:) T([1:3],:)];
szzt=size(ZT,2);
szsp=size(sp,1);
shr=ones(szsp,2);
for i=1:2,
   shr(:,i)=sp(szsp,i);
end
shr(1,:)=sp(1,:);
a=1;
PC=zeros(1,szzt);
while shr(a,:)~=sp(szsp,:),
   for i=1:szsp,
      if shr(a,:)==sp(i,:)
         pck=i;
         break
      end
   end
   for i=szsp:-1:pck+1,
      SF=sqrt(((shr(a,1)-sp(i,1))^2)+((shr(a,2)-sp(i,2))^2));
      for j=1:szzt,
         SC=sqrt(((ZT(1,j)-shr(a,1))^2)+((ZT(2,j)-shr(a,2))^2));
         FC=sqrt(((ZT(1,j)-sp(i,1))^2)+((ZT(2,j)-sp(i,2))^2));
         SN=(SC^2+SF^2-FC^2)/(2*SF);
         if SN<SF & SN>0
```

```
            PC(1,j)=sqrt(SC^2-SN^2);
         else
            if SC<FC
               PC(1,j)=SC;
            else
               PC(1,j)=FC;
            end
         end
      end
      if PC(1,:)>ZT(3,:),
         a=a+1;
         shr(a,:)=sp(i,:);
         break
      end
   end
end
shr_x=shr(:,1);
shr_y=shr(:,2);
```

Fillet_Path

```
%Authored by Matthew Lechliter

function [Shortened_Paths_fillet_x,Shortened_Paths_fillet_y]=fillet_path(Shortened_Paths,min_turn)

%INPUTS:
%
%Shortened_Paths - is a nxmx2 matrix where n is the length of the longest
%path and m is the number of UAVs multiplied by the number of targets.
%The element (nxmx1) x position of the mth uav at point n.  The element
%(nxmx2) y position of the mth uav at point n.
%
%min_turn - minimum turning radius for the UAVs

%OUTPUTS:
%
%Shortened_Paths_fillet - is a nxmx2 matrix where n is the length of the
%longest path with the addition of fillets ((2*old size)-1) and m is the
%number of UAVs multiplied by the number of targets.  The element (nxmx1)
%x position of the mth uav at point n.  The element (nxmx2) y position of
%the mth uav at point n.


Shortened_Paths_fillet=Shortened_Paths*0;
Shortened_Paths_fillet(:,1)=Shortened_Paths(size(Shortened_Paths,1),1);
Shortened_Paths_fillet(:,2)=Shortened_Paths(size(Shortened_Paths,1),2);
Shortened_Paths_fillet(1,:)=Shortened_Paths(1,:);

fillet_counter=2;
for j=2:size(Shortened_Paths,1)-1,
   if Shortened_Paths(j,:)==Shortened_Paths(j+1,:),
      break
   end
   start=Shortened_Paths(j-1,:);
```

```
   middle=Shortened_Paths(j,:);
   finish=Shortened_Paths(j+1,:);
   SM=sqrt(sum((middle-start).^2));
   MF=sqrt(sum(((finish-middle).^2)));
   SF=sqrt(sum(((finish-start).^2)));
   alpha=acos((SM^2+MF^2-SF^2)/(2*SM*MF));
   Fillet=min_turn/tan(alpha/2);
   if Fillet>=SM
      Shortened_Paths_fillet(fillet_counter,:)=Shortened_Paths(j-1,:);
   else
      Shortened_Paths_fillet(fillet_counter,:)=Shortened_Paths(j-1,:)+(Shortened_Paths(j,:)-
Shortened_Paths(j-1,:))*((SM-Fillet)/SM);
   end
   if Fillet>=MF,
      Shortened_Paths_fillet(fillet_counter+1,:)=Shortened_Paths(j+1,:);
   else
      Shortened_Paths_fillet(fillet_counter+1,:)=Shortened_Paths(j,:)+(Shortened_Paths(j+1,:)-
Shortened_Paths(j,:))*(Fillet/MF);
   end
   fillet_counter=fillet_counter+2;
end
Shortened_Paths_fillet_x=Shortened_Paths_fillet(:,1);
Shortened_Paths_fillet_y=Shortened_Paths_fillet(:,2);
```

Heading_Angle_Paths

```
%Authored by Matthew Lechliter

function
[Shortened_Paths_heading_angle_x,Shortened_Paths_heading_angle_y]=heading_angle_paths(Shortened_
Paths,min_turn,HEADING_ANGLE,num_segs);

warning off MATLAB:divideByZero

if HEADING_ANGLE < 0,
   HEADING_ANGLE=pi*2+HEADING_ANGLE;
end

delta_x = Shortened_Paths(2,1) - Shortened_Paths(1,1);
delta_y = Shortened_Paths(2,2) - Shortened_Paths(1,2);

NEW_HEADING_ANGLE=(atan(abs(delta_y)/abs(delta_x)));
if delta_x>=0 & delta_y>=0,
   NEW_HEADING_ANGLE=NEW_HEADING_ANGLE;
end
if delta_x<0 & delta_y>=0,
   NEW_HEADING_ANGLE=pi-NEW_HEADING_ANGLE;
end
if delta_x<0 & delta_y<0,
   NEW_HEADING_ANGLE=pi+NEW_HEADING_ANGLE;
end
if delta_x>=0 & delta_y<0,
   NEW_HEADING_ANGLE=2*pi-NEW_HEADING_ANGLE;
end
```

```
% x and y are the initial positions of the UAV
x=Shortened_Paths(1,1);
y=Shortened_Paths(1,2);

% Rotated heading angle
ROTATED_HEADING_ANGLE=HEADING_ANGLE-NEW_HEADING_ANGLE;

% Rotated NEW_HEADING_ANGLE is 0 degrees
ROTATED_NEW_HEADING_ANGLE=0;

% This section ensures that ROTATED_HEADING_ANGLE is between -pi and pi
if abs(ROTATED_HEADING_ANGLE) > pi
   if ROTATED_HEADING_ANGLE > 0
      ROTATED_HEADING_ANGLE = ROTATED_HEADING_ANGLE-2*pi;
   else
      ROTATED_HEADING_ANGLE = ROTATED_HEADING_ANGLE+2*pi;
   end
end

if abs(ROTATED_HEADING_ANGLE) < pi/5.5
   small_ang=1;
else
   small_ang=0;
   %  Equation found by numerical methods, used to find the location of the
   %  first point to break from the old path onto the first circle

init_dist=0.082565052*(abs(ROTATED_HEADING_ANGLE)/pi*(2*min_turn))^3+0.020254038*(abs(R
OTATED_HEADING_ANGLE)/pi*(2*min_turn))^2+0.629231718*(abs(ROTATED_HEADING_ANGL
E)/pi*(2*min_turn));

   %  xu and yu are the coordinates of the first point that breaks from the
   %  old path and onto the new path following the circles
   xu = x+init_dist*cos(ROTATED_HEADING_ANGLE);
   yu = y+init_dist*sin(ROTATED_HEADING_ANGLE);


   if ROTATED_HEADING_ANGLE >= 0
      ccw = -1;
   else
      ccw = 1;
   end

   %  Finds the locations of the center of both circles, based on whether
   %  the angle made by the intersection of the old and new heading angles
   %  is positive or negative

   xc1 = (x+min_turn*cos(ROTATED_NEW_HEADING_ANGLE + ccw*.5*pi));
   yc1 = (y+min_turn*sin(ROTATED_NEW_HEADING_ANGLE + ccw*.5*pi));

   xc2 = (xu+min_turn*cos(ROTATED_HEADING_ANGLE - ccw*.5*pi));
   yc2 = (yu+min_turn*sin(ROTATED_HEADING_ANGLE - ccw*.5*pi));

   % dx_c2 and dy_c2 are the delta x and delta y between the position of the
   % center of the first break off point and the center of the first circle
   dx_c2 = xu - xc2;
   dy_c2 = yu - yc2;
```

```
% c2_angle is the angle made by the horizon (x-axis) and the line between
% the break off point and center of the first circle
c2_angle=(atan(abs(dy_c2)/abs(dx_c2)));
if dx_c2>=0 & dy_c2>=0,
   c2_angle=c2_angle;
end
if dx_c2<0 & dy_c2>=0,
   c2_angle=pi-c2_angle;
end
if dx_c2<0 & dy_c2<0,
   c2_angle=pi+c2_angle;
end
if dx_c2>=0 & dy_c2<0,
   c2_angle=2*pi-c2_angle;
end

% dx_cc and dy_cc are the delta x and delta y between the position of the
% center of the final circle and the center of the first circle
dx_cc = (xc1 - xc2);
dy_cc = (yc1 - yc2);

% cc_angle is the angle made by the horizon (x-axis) and the line between
% the position of the center of the final circle and the center of the first circle
cc_angle=(atan(abs(dy_cc)/abs(dx_cc)));
if dx_cc>=0 & dy_cc>=0,
   cc_angle=cc_angle;
end
if dx_cc<0 & dy_cc>=0,
   cc_angle=pi-cc_angle;
end
if dx_cc<0 & dy_cc<0,
   cc_angle=pi+cc_angle;
end
if dx_cc>=0 & dy_cc<0,
   cc_angle=2*pi-cc_angle;
end

if ccw == 1
   if abs(ROTATED_HEADING_ANGLE)>pi/2
      cc_point = (2*pi-cc_angle);
      c2_point = -(2*pi-c2_angle);
   else
      cc_point = (2*pi-cc_angle);
      c2_point = (c2_angle);
   end
else
   if abs(ROTATED_HEADING_ANGLE)>pi/2
      cc_point = ccw*(cc_angle);
      c2_point = -1*ccw*(c2_angle);
   else
      cc_point = ccw*(cc_angle);
      c2_point = ccw*(2*pi-c2_angle);
   end
end
```

```
    counter = 1;
    for i = (ccw*2*pi/num_segs:ccw*2*pi/num_segs:cc_point+c2_point)+pi/2-c2_angle
       x_c2(1,counter)=min_turn*sin(i)+xc2;
       y_c2(1,counter) = min_turn*cos(i)+yc2;
       counter = counter + 1;
    end

    dx_c1 = x - xc1;
    dy_c1 = y - yc1;

    c1_angle=(atan(abs(dy_c1)/abs(dx_c1)));
    if dx_c1>=0 & dy_c1>=0,
       c1_angle=c1_angle;
    end
    if dx_c1<0 & dy_c1>=0,
       c1_angle=pi-c1_angle;
    end
    if dx_c1<0 & dy_c1<0,
       c1_angle=pi+c1_angle;
    end
    if dx_c1>=0 & dy_c1<0,
       c1_angle=2*pi-c1_angle;
    end

    cc_angle=cc_angle+ccw*pi;

    counter = 1;
    for i = (-ccw*2*pi/num_segs:-ccw*2*pi/num_segs:(cc_angle-c1_angle))-(cc_angle-pi/2)
       x_c1(1,counter)=min_turn*sin(i)+xc1;
       y_c1(1,counter) = min_turn*cos(i)+yc1;
       counter = counter + 1;
    end

    %  Rotation back to original coordinates
    [t,r] = cart2pol(xu - x,yu - y);
    t = t + NEW_HEADING_ANGLE;
    [xu_temp,yu_temp] = pol2cart(t,r);

    Shortened_Paths_heading_angle_x_temp(1) = x;
    Shortened_Paths_heading_angle_y_temp(1) = y;
    Shortened_Paths_heading_angle_x_temp(2) = xu_temp + x;
    Shortened_Paths_heading_angle_y_temp(2) = yu_temp + y;

    for i = 1:size(x_c2,2)
       [t,r] = cart2pol(x_c2(i) - x,y_c2(i) - y);
       t = t + NEW_HEADING_ANGLE;
       [x_c2_temp,y_c2_temp] = pol2cart(t,r);
       Shortened_Paths_heading_angle_x_temp(size(Shortened_Paths_heading_angle_x_temp,2)+1) =
(x_c2_temp + x);
       Shortened_Paths_heading_angle_y_temp(size(Shortened_Paths_heading_angle_y_temp,2)+1) =
(y_c2_temp + y);
    end


    for i = 1:size(x_c1,2)
       [t,r] = cart2pol(x_c1(i) - x,y_c1(i) - y);
```

```
        t = t + NEW_HEADING_ANGLE;
        [x_c1_temp,y_c1_temp] = pol2cart(t,r);
        Shortened_Paths_heading_angle_x_temp(size(Shortened_Paths_heading_angle_x_temp,2)+1) =
(x_c1_temp + x);
        Shortened_Paths_heading_angle_y_temp(size(Shortened_Paths_heading_angle_y_temp,2)+1) =
(y_c1_temp + y);
    end
end

if small_ang==0,
    sze = size(Shortened_Paths,1);
    Shortened_Paths_heading_angle_x=ones(sze,1)*Shortened_Paths(end,1);
    Shortened_Paths_heading_angle_y=ones(sze,1)*Shortened_Paths(end,2);

    szpts=size(Shortened_Paths_heading_angle_x_temp,2);

    Shortened_Paths_heading_angle_x([1:szpts],1)=Shortened_Paths_heading_angle_x_temp';
    Shortened_Paths_heading_angle_x([szpts+1:sze],1)=Shortened_Paths([1:sze-szpts],1);
    Shortened_Paths_heading_angle_y([1:szpts],1)=Shortened_Paths_heading_angle_y_temp';
    Shortened_Paths_heading_angle_y([szpts+1:sze],1)=Shortened_Paths([1:sze-szpts],2);
else
    Shortened_Paths_heading_angle_x=Shortened_Paths(:,1);
    Shortened_Paths_heading_angle_y=Shortened_Paths(:,2);
End
```

Update_Cost

```
%Authored by Zachary Spritzer and Matthew Lechliter

function [permcost]=update_cost(Shortened_Paths,THREATS)

%INPUTS:
%
%Shortened_Paths - is a nxmx2 matrix where n is the length of the longest
%path and m is the number of UAVs multiplied by the number of targets.
%The element (nxmx1) x position of the mth uav at point n.  The element
%(nxmx2) y position of the mth uav at point n.
%
%THREATS - is a 4xn matrix where n is the number of Threats, the first row
%is the x position of the threats, the second row is the y position of the
%threats, the third row is the range of the threats, and the fourth row is
%the level of danger of the threats.

%OUTPUTS:
%
%permcost - cost associated with the nth UAV going to the mth TARGET


szsp_num=size(Shortened_Paths,1)-1;
nthrts=size(THREATS,2);
permcost=0;
```

```
for i=1:szsp_num,
   start_x=Shortened_Paths(i,1);start_y=Shortened_Paths(i,2);
   finish_x=Shortened_Paths(i+1,1);finish_y=Shortened_Paths(i+1,2);
   SF=sqrt(((finish_x-start_x)^2)+((finish_y-start_y)^2));
   for j=1:nthrts,
      SC=sqrt(((THREATS(1,j)-start_x)^2)+((THREATS(2,j)-finish_y)^2));
      FC=sqrt(((THREATS(1,j)-finish_x)^2)+((THREATS(2,j)-finish_y)^2));
      SN=(SC^2+SF^2-FC^2)/(2*SF);
      if SN<SF & SN>0,PC=sqrt(SC^2-SN^2);
      else
         if SC<FC,PC=SC;
         else
            PC=FC;
         end
      end
      if PC < THREATS(3,j),SF=SF+(THREATS(4,j)*100);
      end
   end
   permcost=permcost+SF;
end
```

Task Allocation Related Functions

%Authored by Zachary Spritzer and Matthew Lechliter

MMKP_Task_Allocation

```
function
[Selected_Paths_x,Selected_Paths_y]=mmkp_task_allocation(totalcost,Shortened_Paths_x,Shortened_Path
s_y,nuav)

%INPUTS:
%
%totalcost - is a mxn matrix where m is the number of uavs and n is the
%number of possible paths for each uav.  The element (m,n) of this matrix
%is the cost for the mth uav to take the nth path.
%
%Shortened_Paths - is a nxmx2 matrix where n is the length of the longest
%path and m is the number of UAVs multiplied by the number of targets.
%The element (nxmx1) x position of the mth uav at point n.  The element
%(nxmx2) y position of the mth uav at point n.
%
%nuav - number of UAVs

%OUTPUTS:
%
%Selected_Pos - is a nxmx2 matrix where n is the length of the longest
%path and m is the number of UAVs.  The element (nxmx1) x position of the
%mth uav at point n.  The element (nxmx2) y position of the mth uav at
%point n.
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%MMKP algorithm
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
 [bestcomb,mincost]=mmkp_new(totalcost);

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Taking the results from mmkp
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Selected_Paths_x=zeros(size(Shortened_Paths_x,1),nuav);
Selected_Paths_y=zeros(size(Shortened_Paths_x,1),nuav);
for i=1:nuav,
   Selected_Paths_x(:,i)=Shortened_Paths_x(:,(nuav)*(i-1)+bestcomb(1,i));
   Selected_Paths_y(:,i)=Shortened_Paths_y(:,(nuav)*(i-1)+bestcomb(1,i));
End
```

MMKP_New

```
%Authored by Zachary Spritzer, Matthew Lechliter, and Elena Lucci

function [bestcomb,mincost]=mmkp_new(totalcost)
%Inputs:
%
%totalcost - is a nxm matrix where n is the total number of uav's and m is
%the total number of targets or paths. Where the element nxm is the cost
%associated with uav "n" choosing target or path "m".
%
%Outputs:
%
%bestcomb - is a 1xn row with n equal to the number or uav's where each
%element of the row represents which path the uav should select to give the
%optimal solution.
%
%mincost - is a scalar number which is sum of the optimal costs for all
%the uav's paths.
nuav=size(totalcost,1);
mincost=inf;
C_new=perms(1:nuav);
for j=1:size(C_new,1),
   sc=0;
   for i=1:nuav,
      sc=sc+totalcost(i,C_new(j,i));
   end
   if sc < mincost
      bestcomb=C_new(j,:);
      mincost = sc;
   end
end
```

Appendix B

Longitudinal Dimensional and Lateral Directional Stability Derivatives

Longitudinal Dimensional Stability Derivatives

$$X_u = \frac{-\bar{q}_1 S(C_{Du} + 2C_{D1})}{mU_1}(\sec^{-1}) \qquad X_{Tu} = \frac{-\bar{q}_1 S(C_{Txu} + 2C_{Tx1})}{mU_1}(\sec^{-1})$$

$$X_\alpha = \frac{-\bar{q}_1 S(C_{D\alpha} - C_{L1})}{m}(ft \cdot \sec^{-2}) \qquad X_{\delta_E} = \frac{-\bar{q}_1 S C_{D_{\delta E}}}{m}(ft \cdot \sec^{-2})$$

$$Z_u = \frac{-\bar{q}_1 S(C_{Lu} + 2C_{L1})}{mU_1}(\sec^{-1}) \qquad Z_\alpha = \frac{-\bar{q}_1 S(C_{L\alpha} + C_{D1})}{m}(ft \cdot \sec^{-2})$$

$$Z_{\dot{\alpha}} = \frac{-\bar{q}_1 S\bar{c} C_{L\dot{\alpha}}}{2mU_1}(ft \cdot \sec^{-1}) \qquad Z_q = \frac{-\bar{q}_1 S\bar{c} C_{Lq}}{2mU_1}(ft \cdot \sec^{-1})$$

$$Z_{\delta E} = \frac{-\bar{q}_1 S C_{L\delta E}}{m}(ft \cdot \sec^{-2}) \qquad M_u = \frac{\bar{q}_1 S(C_{Mu} + 2C_{M1})}{U_1 I_{YY}}(ft^{-1} \cdot \sec^{-1})$$

$$M_{Tu} = \frac{\bar{q}_1 S\bar{c}(C_{MTu} + 2C_{MT1})}{U_1 I_{YY}}(ft^{-1} \cdot \sec^{-1}) \qquad M_\alpha = \frac{\bar{q}_1 S\bar{c} C_{M\alpha}}{I_{YY}}(\sec^{-2})$$

$$M_{T\alpha} = \frac{\bar{q}_1 S\bar{c} C_{MT\alpha}}{I_{YY}}(\sec^{-2}) \qquad M_{\dot{\alpha}} = \frac{\bar{q}_1 S\bar{c}^2 C_{M\dot{\alpha}}}{2I_{YY}U_1}(\sec^{-1})$$

$$M_{\delta E} = \frac{\bar{q}_1 S\bar{c} C_{M\delta E}}{I_{YY}}(\sec^{-2}) \qquad M_q = \frac{\bar{q}_1 S\bar{c}^2 C_{Mq}}{2I_{YY}U_1}(\sec^{-1})$$

Modified Longitudinal Dimensional Statbility Derivatives

$$X'_\alpha = g\cos\gamma_1 + X_\alpha \qquad X'_u = X_u \qquad X'_\theta = -g\cos\gamma_1$$

$$X'_{\delta E} = X_{\delta E} \qquad Z'_\alpha = \frac{g\sin\gamma_1 + Z_\alpha}{U_1 - Z_{\dot{\alpha}}} \qquad Z'_u = \frac{Z_u}{U_1 - Z_{\dot{\alpha}}}$$

$$Z'_q = \frac{U_1 + Z_q}{U_1 - Z_{\dot{\alpha}}} \qquad Z'_\theta = \frac{-g\sin\gamma_1}{U_1 - Z_{\dot{\alpha}}} \qquad Z'_{\delta E} = \frac{Z_{\delta E}}{U_1 - Z_{\dot{\alpha}}}$$

$$M'_\alpha = M_{\dot{\alpha}} Z'_\alpha + M_\alpha \qquad M'_u = M_{\dot{\alpha}} Z'_u + M_u \qquad M'_q = M_{\dot{\alpha}} Z'_q + M_q$$

$$M'_\theta = M_{\dot{\alpha}} Z'_\theta \qquad M'_{\delta E} = M_{\dot{\alpha}} Z'_{\delta E} + M_{\delta E}$$

$$Z''_\alpha = U_1 Z'_\alpha - g\sin\gamma_1 \qquad Z''_u = U_1 Y'_u \qquad Y''_q = U_1(Z'_q - 1)$$

$$Z''_\theta = U_1 Z'_\theta + g\cos\gamma_1 \qquad Z''_{\delta E} = U_1 Z'_{\delta E}$$

Lateral Directional Dimensional Stability Derivatives

$$Y_\beta = \frac{\bar{q}_1 S C_{Y\beta}}{m}(ft \cdot \sec^{-2}) \qquad Y_P = \frac{\bar{q}_1 S b C_{Yp}}{2mU_1}(ft \cdot \sec^{-1})$$

$$Y_r = \frac{\bar{q}_1 S b C_{Yr}}{2mU_1}(ft \cdot \sec^{-1}) \qquad Y_{\delta_A} = \frac{\bar{q}_1 S C_{Y\delta A}}{m}(ft \cdot \sec^{-2})$$

$$Y_{\delta R} = \frac{\bar{q}_1 S C_{Y\delta R}}{m}(\,ft\cdot\sec^{-2}) \qquad L_\beta = \frac{\bar{q}_1 SbC_{L\beta}}{I_{XX}}(\sec^{-2})$$

$$L_p = \frac{\bar{q}_1 Sb^2 C_{Lp}}{2I_{XX}U_1}(\sec^{-1}) \qquad L_r = \frac{\bar{q}_1 Sb^2 C_{Lr}}{2I_{XX}U_1}(\sec^{-1})$$

$$L_{\delta A} = \frac{\bar{q}_1 SbC_{L\delta A}}{I_{XX}}(\sec^{-2}) \qquad L_{\delta R} = \frac{\bar{q}_1 SbC_{\delta R}}{I_{XX}}(\sec^{-2})$$

$$N_\beta = \frac{\bar{q}_1 SbC_{N\beta}}{I_{ZZ}}(\sec^{-2}) \qquad N_{T\beta} = \frac{\bar{q}_1 SbC_{NT\beta}}{I_{ZZ}}(\sec^{-2})$$

$$N_p = \frac{\bar{q}_1 Sb^2 C_{np}}{2I_{ZZ}U_1}(\sec^{-1}) \qquad N_r = \frac{\bar{q}_1 Sb^2 C_{nr}}{2I_{ZZ}U_1}(\sec^{-1})$$

$$N_{\delta A} = \frac{\bar{q}_1 SbC_{N\delta A}}{I_{ZZ}}(\sec^{-2}) \qquad N_{\delta R} = \frac{\bar{q}_1 SbC_{N\delta R}}{I_{ZZ}}(\sec^{-2})$$

Modified Lateral Directional Dimensional Statbility Derivatives

Given: $A_1 = \dfrac{I_{XZ}}{I_{XX}}$ and $B_1 = \dfrac{I_{XZ}}{I_{ZZ}}$

$$Y_\beta^{'} = \frac{Y_\beta}{U_1} \qquad Y_p^{'} = \frac{Y_p}{U_1} \qquad Y_r^{'} = \frac{Y_r}{U_1} - 1$$

$$Y_\phi^{'} = \frac{g\cos\Theta_1}{U_1} \qquad Y_{\delta A}^{'} = \frac{Y_{\delta A}}{U_1} \qquad Y_{\delta R}^{'} = \frac{Y_{\delta R}}{U_1}$$

$$L_\beta^{'} = \frac{A_1 N_\beta + L_\beta}{1 - A_1 B_1} \qquad L_p^{'} = \frac{A_1 N_p + L_p}{1 - A_1 B_1} \qquad L_r^{'} = \frac{A_1 N_r + L_r}{1 - A_1 B_1}$$

$$L_{\delta A}^{'} = \frac{A_1 N_{\delta A} + L_{\delta A}}{1 - A_1 B_1} \qquad L_{\delta R}^{'} = \frac{A_1 N_{\delta R} + L_{\delta R}}{1 - A_1 B_1} \qquad N_\beta^{'} = \frac{B_1 L_\beta + N_\beta}{1 - A_1 B_1}$$

$$N_p^{'} = \frac{B_1 L_p + N_p}{1 - A_1 B_1} \qquad N_r^{'} = \frac{B_1 L_r + N_r}{1 - A_1 B_1} \qquad N_{\delta A}^{'} = \frac{B_1 L_{\delta A} + N_{\delta A}}{1 - A_1 B_1}$$

$$N_{\delta R}^{'} = \frac{B_1 L_{\delta R} + N_{\delta R}}{1 - A_1 B_1} \qquad Y_\beta^{''} = U_1 Y_\beta^{'} \qquad Y_p^{''} = U_1 Y_p^{'}$$

$$Y_r^{''} = U_1(Y_r^{'} + 1) \qquad Y_\phi^{''} = U_1 Y_\phi^{'} - g\cos\Theta_1 \qquad Y_{\delta A}^{''} = U_1 Y_{\delta A}^{'}$$

$$Y_{\delta R}^{''} = U_1 Y_{\delta R}^{'}$$

Appendix C

Simulation Implementation MATLAB Files

Initialization and Display Functions

Define_Battlefield

```
%Authored by Zachary Spritzer and Matthew Lechliter

function [UAVS,TARGETS,THREATS,ZONES,n_uav,n_targ,n_zones,n_threats]=define_battlefield

UAVS=zeros(4,9);
TARGETS=zeros(4,9);
THREATS=zeros(4,15);
ZONES=zeros(3,10);

n_uav=menu('Enter the number of UAVs for this simulation','                    1
',...
   '2','3','4','5','6','7','8','9');
n_targ=menu('Enter the number of TARGETs for this simulation','                    1
',...
   '2','3','4','5','6','7','8','9');
n_zones=menu('Enter the number of NO-FLY ZONEs for this simulation','                    1
',...
   '2','3','4','5','6','7','8','9','10');
n_threats=menu('Enter the number of THREATs for this simulation','                    1
',...
   '2','3','4','5','6','7','8','9','10','11','12','13','14','15');



Vel_UAV=0.26;
menu('Using the crosshairs and clicking on the plot','Place UAVs at desired positions');
axis([5 200 5 200]);
grid on;

for i=1:n_uav
   [UAVS(1,i),UAVS(2,i)]=ginput(1);
   plot(UAVS(1,i),UAVS(2,i),'bd');
   text(UAVS(1,i)+5,UAVS(2,i),{i},'FontSize',12,'Color','b');
   axis([5 200 5 200]);
   grid on;
   UAVS(3,i)=2;
   UAVS(4,i)=Vel_UAV ;
   hold on;
end

hold on;

menu('Using the crosshairs and clicking on the plot','Place TARGETs at desired positions');
for i=1:n_targ
   tar=menu('Select Target Value - Scale 10-100','10','20','30','40','50','60','70','80','90','100');
   TARGETS(3,i)=10*tar;
   TARGETS(4,i)=1;
   [TARGETS(1,i),TARGETS(2,i)]=ginput(1);
   plot(TARGETS(1,i),TARGETS(2,i),'x','Color',[0,.4,0]);
   text(TARGETS(1,i)+5,TARGETS(2,i),{i},'FontSize',12,'Color',[0,0.4,0]);
   axis([5 200 5 200]);
```

```
   grid on;
   hold on;
end

hold on;

menu('Using the crosshairs and clicking on the plot','Place NO-FLY ZONEs at desired positions');

for i=1:n_zones
   ZONES(3,i)=9;
   [ZONES(1,i),ZONES(2,i)]=ginput(1);
   axis([5 200 5 200]);
   grid on;
   t_nfz = (1/16:1/16:1)'*2*pi;
   x_nfz = ZONES(3,i)*sin(t_nfz)+ZONES(1,i);
   y_nfz = ZONES(3,i)*cos(t_nfz)+ZONES(2,i);
   fill(x_nfz,y_nfz,'k');
end

menu('Using the crosshairs and clicking on the plot','Place THREATs at desired positions');
hold on;

for i=1:n_threats
   thr=menu('Select Threat Type','KS-19 100mm AntiAircraft Artillery - Range 4000 meters, 40%
Probability of Kill',...
      'SA-7 Grail - Man-Portable SAM - Range 5000 meters, 50% Probabilty of Kill',...
      'Crotale SAM - Range 10,000 meters, 80% Probability of Kill',...
      'SA-2 - Range 30,000 meters, 80% Probabilty of Kill');
   if thr == 1
      THREATS(3,i)=4;
      THREATS(4,i)=.4;
   end
   if thr == 2
      THREATS(3,i)=5;
      THREATS(4,i)=.5;
   end
   if thr == 3
      THREATS(3,i)=10;
      THREATS(4,i)=.8;
   end
   if thr == 4
      THREATS(3,i)=30;
      THREATS(4,i)=.8;
   end
   [THREATS(1,i),THREATS(2,i)]=ginput(1);
   plot(THREATS(1,i),THREATS(2,i),'r*');
   text(THREATS(1,i)+5,THREATS(2,i),{i},'FontSize',12,'Color','r')
   axis([5 200 5 200]);
   grid on;
   t_threat = (1/32:1/32:1)'*2*pi;
   x_threat = THREATS(3,i)*sin(t_threat)+THREATS(1,i);
   y_threat = THREATS(3,i)*cos(t_threat)+THREATS(2,i);
   plot(x_threat,y_threat,'r.');
   hold on;
end
```

Display_Initial_S

```
%Authored by Zachary Spritzer and Matthew Lechliter

function [sys,x0,str,ts] = display_initial_s(t,x,u,flag,T)
% Dispatch the flag. The switch function controls the calls to
% S-function routines at each simulation stage.
switch flag,

  case 0
    [sys,x0,str,ts] = mdlInitializeSizes(T); % Initialization

  case 3
    mdlOutputs(u); % Calculate outputs

  case { 1, 2, 4, 9 }
    sys = []; % Unused flags

  otherwise
    error(['Unhandled flag = ',num2str(flag)]); % Error handling
end;

%=============================================================
% Function mdlInitializeSizes initializes the states, sample
% times, state ordering strings (str), and sizes structure.
%=============================================================
function [sys,x0,str,ts] = mdlInitializeSizes(T)
% Call function simsizes to create the sizes structure.
sizes = simsizes;
% Load the sizes structure with the initialization information.
sizes.NumContStates= 0;
sizes.NumDiscStates= 0;
sizes.NumOutputs=    0;
sizes.NumInputs=     36+36+30+60;
sizes.DirFeedthrough=1;
sizes.NumSampleTimes=1;
% Load the sys vector with the sizes information.
sys = simsizes(sizes);
%
x0 = []; % No continuous states
%
str = []; % No state ordering
%
ts = [T 0]; % Inherited sample time
% End of mdlInitializeSizes.
%=============================================================
% Function mdlOutputs performs the calculations.
%=============================================================
function mdlOutputs(u)


display_initial(u);

% End of mdlOutputs.
```

Display_Initial

```
%Authored by Zachary Spritzer and Matthew Lechliter

function display_initial(u)

UAVS=u([1:4*9],1);
UAVS=reshape(UAVS,4,9);
a=4*9;
TARGETS=u([a+1:a+4*9]);
TARGETS=reshape(TARGETS,4,9);
a=a+4*9;
ZONES=u([a+1:a+3*10]);
ZONES=reshape(ZONES,3,10);
a=a+3*10;
THREATS=u([a+1:a+4*15]);
THREATS=reshape(THREATS,4,15);

for i=1:9
   if abs(sum(UAVS(:,i)))>0 & abs(sum(UAVS(:,i)))~=0.26
      disp(sprintf('UAV %d exists at location %d x, location %d y, altitude %d km, and is flying at %d m/s.
\n',...
         i,round(UAVS(1,i)),round(UAVS(2,i)),round(UAVS(3,i)),round(UAVS(4,i)*1000)));
  end
end

for i=1:9
   if abs(sum(TARGETS(:,i)))>0
      disp(sprintf('Target %d indicated to be at location %d x, location %d y , and with an estimated value
of %d. \n',...
         i,round(TARGETS(1,i)),round(TARGETS(2,i)),round(TARGETS(3,i))));
  end
end

for i=1:10
   if abs(sum(ZONES(:,i)))>0
      disp(sprintf('No-Fly Zone %d exists at location %d x, location %d y, and with a radius of %d km.
\n',...
         i,round(ZONES(1,i)),round(ZONES(2,i)),round(ZONES(3,i))));
  end
end

for i=1:15
   if abs(sum(THREATS(:,i)))>0
      disp(sprintf('Threat %d exists at location %d x, location %d y, with a range of %d km, and has a
probability of kill of %d%%. \n',...

i,round(THREATS(1,i)),round(THREATS(2,i)),round(THREATS(3,i)),round(THREATS(4,i)*100)));
  end
end
```

Plot_UAV

```
%Authored by Zachary Spritzer and Matthew Lechliter

function
plot_uav(UAVS,TARGETS,ZONES,THREATS,uav_path_x,uav_path_y,n_plots,uavs_existing,targ_existi
ng,threats_existing)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Plotting results
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
figure(n_plots);
hold on;
% for i=1:2,
%     subplot(1,2,i),
   for i=1:size(UAVS,2)
      if uavs_existing(1,i)==1
         plot(UAVS(1,i),UAVS(2,i),'bd');
         text(UAVS(1,i)+5,UAVS(2,i),{i},'FontSize',12,'Color','b');
         axis([5 200 5 200]);
         hold on;
      end
   end
   for i=1:size(TARGETS,2)
      if targ_existing(1,i)==1
         plot(TARGETS(1,i),TARGETS(2,i),'x','Color',[0,.4,0]);
         text(TARGETS(1,i)+5,TARGETS(2,i),{i},'FontSize',12,'Color',[0,0.4,0]);
         axis([5 200 5 200]);
         hold on;
      end
   end
   for i=1:size(THREATS,2)
      if threats_existing(1,i)==1
         plot(THREATS(1,i),THREATS(2,i),'r*');
         text(THREATS(1,i)+5,THREATS(2,i),{i},'FontSize',12,'Color','r')
         axis([5 200 5 200]);
         hold on;
      end
   end
   hold on;
% end

%Plotting Threats and range
for i=1:size(THREATS,2)
   if threats_existing(1,i)==1
      t_threat = (1/32:1/32:1)'*2*pi;
      x_threat = THREATS(3,i)*sin(t_threat)+THREATS(1,i);
      y_threat = THREATS(3,i)*cos(t_threat)+THREATS(2,i);
%       for i=1:2,
%           subplot(1,2,i),
         plot(x_threat,y_threat,'r.');hold on;
%       end
   end
end

%Plotting No fly Zones
for i=1:size(ZONES,2)
```

```
   t_nfz = (1/16:1/16:1)'*2*pi;
   x_nfz = ZONES(3,i)*sin(t_nfz)+ZONES(1,i);
   y_nfz = ZONES(3,i)*cos(t_nfz)+ZONES(2,i);
%     for i=1:2,
%         subplot(1,2,i),
       fill(x_nfz,y_nfz,'k');hold on;
%     end
end


%Plotting shortened paths
for i=1:size(uav_path_x,1)
%     subplot(1,2,2),
   plot(uav_path_x(i,:),uav_path_y(i,:),'b-');hold on;
end

% subplot(1,2,2),
title('Voronoi Diagram Method');hold on;
% for i=1:2,
%     subplot(1,2,i),
   axis([-25 250 -25 250]);hold on;
      xlabel('Kilometers')
   ylabel('Kilometers')
% end
```

Path Planning Related Functions

Path_Planning_S

```
%Authored by Zachary Spritzer and Matthew Lechliter

function [sys,x0,str,ts] = path_planning_s(t,x,u,flag,T)
% Dispatch the flag. The switch function controls the calls to
% S-function routines at each simulation stage.
switch flag,

  case 0
    [sys,x0,str,ts] = mdlInitializeSizes(T); % Initialization

  case 3
    sys = mdlOutputs(u); % Calculate outputs

  case { 1, 2, 4, 9 }
    sys = []; % Unused flags

  otherwise
    error(['Unhandled flag = ',num2str(flag)]); % Error handling
end;

%=============================================================
% Function mdlInitializeSizes initializes the states, sample
% times, state ordering strings (str), and sizes structure.
%=============================================================
```

```
function [sys,x0,str,ts] = mdlInitializeSizes(T)
% Call function simsizes to create the sizes structure.
sizes = simsizes;
% Load the sizes structure with the initialization information.
sizes.NumContStates= 0;
sizes.NumDiscStates= 0;
sizes.NumOutputs=    9*100*4+9;
sizes.NumInputs=     36+36+30+60+1+1+9;
sizes.DirFeedthrough=1;
sizes.NumSampleTimes=1;
% Load the sys vector with the sizes information.
sys = simsizes(sizes);
%
x0 = []; % No continuous states
%
str = []; % No state ordering
%
ts = [T 0]; % Inherited sample time
% End of mdlInitializeSizes.
%=============================================================
% Function mdlOutputs performs the calculations.
%=============================================================
function sys = mdlOutputs(u)

[sys]=path_planning(u);

% End of mdlOutputs.
```

Path_Planning

```
%Authored by Zachary Spritzer and Matthew Lechliter

function [out]=path_planning(in)

UAVS_long=in([1:36],1);
UAVS_long=reshape(UAVS_long,4,9);
TARGETS_long=in([37:72]);
TARGETS_long=reshape(TARGETS_long,4,9);
ZONES_long=in([73:102]);
ZONES_long=reshape(ZONES_long,3,10);
THREATS_long=in([103:162]);
THREATS_long=reshape(THREATS_long,4,15);
TIME=in(163);
n_plots=in(164);
HEADING_ANGLE=in([165:173]);

uavs_existing=zeros(1,9);
for i=1:9
   if abs(sum(UAVS_long(:,i)))>0 & abs(sum(UAVS_long(:,i)))~=0.26
      uavs_existing(1,i)=1;
  end
end
[UAVS]=filter_zeros(UAVS_long,9);
n_uav=size(UAVS,2);
```

```
targ_existing=zeros(1,9);
for i=1:9
   if TARGETS_long(3,i)~=0,
      targ_existing(1,i)=1;
  end
end
[TARGETS_temp]=filter_zeros(TARGETS_long,9);
TARGETS=[TARGETS_temp(1,:);TARGETS_temp(2,:)];
n_targ=size(TARGETS,2);

[ZONES]=filter_zeros(ZONES_long,10);
n_zones=size(ZONES,2);

threats_existing=zeros(1,15);
for i=1:15
   if THREATS_long(3,i)~=0
      threats_existing(1,i)=1;
  end
end
[THREATS]=filter_zeros(THREATS_long,15);
n_threats=size(THREATS,2);

ZONES_REAL=ZONES;
THREATS_REAL=THREATS;

ZONES(3,:)=1.15*ZONES_REAL(3,:);
THREATS(3,:)=1.15*THREATS_REAL(3,:);



split_seg=10;
min_turn=1;
[all_pos,all_lines_x,all_lines_y,all_costs]=vrn_diag_gen(UAVS,TARGETS,ZONES,THREATS);
[stored_paths,totalcost]=cheapest_paths(all_pos,all_lines_x,all_lines_y,all_costs,UAVS,TARGETS,ZONE
S,THREATS);
[Shortened_Paths_x,Shortened_Paths_y,totalcost]=path_shrtng(stored_paths,all_pos,ZONES,THREATS,m
in_turn,split_seg,n_uav,n_targ,HEADING_ANGLE);
[Selected_Paths_x,Selected_Paths_y]=mmkp_task_allocation(totalcost,Shortened_Paths_x,Shortened_Path
s_y,n_uav);
[uav_path_x,uav_path_y,time_uav,altitude_uav]=vrt_sim_convert(Selected_Paths_x,Selected_Paths_y,UA
VS,min_turn*2);

if n_plots~=0,

plot_uav(UAVS_long,TARGETS_long,ZONES_REAL,THREATS_long,uav_path_x,uav_path_y,n_plots,
uavs_existing,targ_existing,threats_existing);
end

disp(sprintf('Path Planning ran at time %d. \n',round(TIME)));

bestcomb=zeros(1,9);
for i=1:n_uav,
   for j=1:n_targ,
```

```
      if round(Selected_Paths_x(end,i)*10)==round(TARGETS(1,j)*10) &
round(Selected_Paths_y(end,i)*10)==round(TARGETS(2,j)*10)
         bestcomb(1,i)=j;
         break
      end
   end
end

%Making into vector
uav_x=zeros(9,100);
uav_y=zeros(9,100);
uav_time=zeros(9,100);
uav_alt=zeros(9,100);
selected_targets=zeros(9,1);
szpath=size(uav_path_x,2);
counter=1;
for i=1:9,
   if uavs_existing(1,i)==1
      selected_targets(i,1)=bestcomb(1,counter);
      uav_x(i,[1:szpath])=uav_path_x(counter,:);
      uav_y(i,[1:szpath])=uav_path_y(counter,:);
      uav_time(i,[1:szpath])=time_uav(counter,:)+TIME;
      uav_alt(i,[1:szpath])=altitude_uav(counter,:);
      counter=counter+1;
   end
end
sys_temp=[];
for i=1:9;
   sys_temp=[sys_temp,uav_x(i,:),uav_y(i,:),uav_alt(i,:),uav_time(i,:)];
end
out=[sys_temp,selected_targets'];
```

Filter_Zeros

```
%Authored by Zachary Spritzer and Matthew Lechliter

function [A]=filter_zeros(A_long,n)

A=[];
counter=1;
for i=1:n
   if abs(sum(A_long(:,i)))>0 & abs(sum(A_long(:,i)))~=0.26
      A(:,counter)=A_long(:,i);
      counter=counter+1;
  end
end
```

VRT_sim_convert

```
%Authored by Zachary Spritzer and Matthew Lechliter

function [uav_path_x,uav_path_y,time_uav,altitude_uav]=vrt_sim_convert(shr_x,shr_y,UAVS,distpast)
```

```
%
%INPUTS:
%
%shr - is a nxmx2 matrix where n is the length of the longest
%path and m is the number of UAVs.  The element (nxmx1) x position of the
%mth uav at point n.  The element (nxmx2) y position of the mth uav at
%point n.
%
%UAVS - is a 4xn matrix where n is number of UAVs, the first row is the
%initial x position of the UAVs, the second row is the initial y position
%of the UAVs, the third row is the initial altitude of the UAVs, and
%the fourth row is the intial Velocity of the UAVs.
%
%
%OUTPUTS:
%
%uav_path_x - is a mxn matrix where m is the number of uavs and m is the
%length of the longest path. These are the x coordinates of the paths.
%
%uav_path_y - is a mxn matrix where m is the number of uavs and m is the
%length of the longest path. These are the y coordinates of the paths.
%
%time_uav - is a mxn matrix where m is the number of uavs and m is the
%length of the longest path. These values correspond to the time at which
%the uavs are at coordinates x and y in uav_path_x and uav_path_y.
%
%altitude_uav - is a mxn matrix where m is the number of uavs and m is the
%length of the longest path. These values correspond to the altitudes that
%the uavs are at when they are at coordinates x and y in uav_path_x and
%uav_path_y.
%
%Threat_range_vrt - is a 1xn vector where n is the number of threats, where
%the first row is the range of the threats at the altitude where the uavs
%are flying.
%
%Zone_range_vrt - is a 1xn vector where n is the number of zones, where
%the first row is the range of the zones at the altitude where the uavs
%are flying.

nuav=size(shr_x,2);
szshrpth=size(shr_x,1);
shr_x=[[shr_x];[shr_x(szshrpth,:)]];
shr_y=[[shr_y];[shr_y(szshrpth,:)]];
uav_path_x=zeros(nuav,szshrpth+1);
uav_path_y=zeros(nuav,szshrpth+1);
for i=1:nuav,
   for j=1:szshrpth,
      if [shr_x(j+1,i),shr_y(j+1,i)]==[shr_x(j,i),shr_y(j,i)] | j==szshrpth,
         lst_pnt_x=shr_x(j,i);
         nxtlst_pnt_x=shr_x(j-1,i);
         lst_pnt_y=shr_y(j,i);
         nxtlst_pnt_y=shr_y(j-1,i);
         dist_pnts=sqrt(((lst_pnt_x-nxtlst_pnt_x)^2)+((lst_pnt_y-nxtlst_pnt_y)^2));
         last_x=lst_pnt_x+((lst_pnt_x-nxtlst_pnt_x)*(distpast/dist_pnts));
         last_y=lst_pnt_y+((lst_pnt_y-nxtlst_pnt_y)*(distpast/dist_pnts));
         uav_path_x(i,[j+1:szshrpth+1])=last_x;
```

```
            uav_path_y(i,[j+1:szshrpth+1])=last_y;
            uav_path_x(i,j)=shr_x(j,i);
            uav_path_y(i,j)=shr_y(j,i);
            break
        else
            uav_path_x(i,j)=shr_x(j,i);
            uav_path_y(i,j)=shr_y(j,i);
        end
    end
end

%Initializing matrixes
time_uav_temp=zeros(nuav,szshrpth+1);
time_uav=zeros(nuav,szshrpth+1);
altitude_uav=zeros(nuav,szshrpth+1);

%Time matrix
for i=1:nuav,
    for j=1:szshrpth,
        shr_dist(i,j)=sqrt((uav_path_x(i,j)-uav_path_x(i,j+1))^2+(uav_path_y(i,j)-uav_path_y(i,j+1))^2);
        time_uav_temp(i,j+1)=shr_dist(i,j)/UAVS(4,i);
    end
    time_uav(i,[2:szshrpth+1])=sum(time_uav_temp(i,:));
    for j=2:szshrpth+1,
        time_uav(i,j)=time_uav(i,j-1)+time_uav_temp(i,j);
    end
end

time_uav=time_uav*1.01;

%Altitude matrix
for i=1:nuav,
    for j=1:szshrpth+1,
        altitude_uav(i,j)=UAVS(3,i);
    end
end
```

No-Fly Zone Related Functions

UAV_Crash_S

```
%Authored by Zachary Spritzer and Matthew Lechliter

function [sys,x0,str,ts] =uav_crash_s(t,x,u,flag,T)
% Dispatch the flag. The switch function controls the calls to
% S-function routines at each simulation stage.
switch flag,

    case 0
        [sys,x0,str,ts] = mdlInitializeSizes(T); % Initialization

    case 3
        sys = mdlOutputs(u); % Calculate outputs
```

```
   case { 1, 2, 4, 9 }
      sys = []; % Unused flags

   otherwise
      error(['Unhandled flag = ',num2str(flag)]); % Error handling
end;

%=============================================================
% Function mdlInitializeSizes initializes the states, sample
% times, state ordering strings (str), and sizes structure.
%=============================================================
function [sys,x0,str,ts] = mdlInitializeSizes(T)
% Call function simsizes to create the sizes structure.
sizes = simsizes;
% Load the sizes structure with the initialization information.
sizes.NumContStates= 0;
sizes.NumDiscStates= 0;
sizes.NumOutputs=    9;
sizes.NumInputs=     57;
sizes.DirFeedthrough=1;
sizes.NumSampleTimes=1;
% Load the sys vector with the sizes information.
sys = simsizes(sizes);
%
x0 = []; % No continuous states
%
str = []; % No state ordering
%
ts = [T 0]; % Inherited sample time
% End of mdlInitializeSizes.
%=============================================================
% Function mdlOutputs performs the calculations.
%=============================================================
function sys = mdlOutputs(u);


[sys]=uav_crash(u);

% End of mdlOutputs.
```

UAV_Crash

```
%Authored by Zachary Spritzer and Matthew Lechliter

function [sys]=uav_crash(u)

uav_pos=reshape(u([1:27],1),3,9);
zone_pos=reshape(u([28:57],1),3,10);

uav_shot_down=zeros(9,1);

for i=1:9,
   for j=1:10,
```

```
      dist_uav_zone=sqrt(((uav_pos(1,i)-zone_pos(1,j))^2)+((uav_pos(2,i)-zone_pos(2,j))^2));
      if dist_uav_zone < zone_pos(3,j),
         uav_shot_down(i,1)=1;
      end
   end
end
sys=[uav_shot_down];
```

Threat Related Functions

UAV_Intercepted_S

```
%Authored by Zachary Spritzer and Matthew Lechliter

function [sys,x0,str,ts] =uav_intercepted_s(t,x,u,flag,T)
% Dispatch the flag. The switch function controls the calls to
% S-function routines at each simulation stage.
switch flag,

   case 0
      [sys,x0,str,ts] = mdlInitializeSizes(T); % Initialization

   case 3
      sys = mdlOutputs(u); % Calculate outputs

   case { 1, 2, 4, 9 }
      sys = []; % Unused flags

   otherwise
      error(['Unhandled flag = ',num2str(flag)]); % Error handling
end;

%=============================================================
% Function mdlInitializeSizes initializes the states, sample
% times, state ordering strings (str), and sizes structure.
%=============================================================
function [sys,x0,str,ts] = mdlInitializeSizes(T)
% Call function simsizes to create the sizes structure.
sizes = simsizes;
% Load the sizes structure with the initialization information.
sizes.NumContStates= 0;
sizes.NumDiscStates= 0;
sizes.NumOutputs=    24;
sizes.NumInputs=     87;
sizes.DirFeedthrough=1;
sizes.NumSampleTimes=1;
% Load the sys vector with the sizes information.
sys = simsizes(sizes);
%
x0 = []; % No continuous states
%
str = []; % No state ordering
%
```

```
ts = [T 0]; % Inherited sample time
% End of mdlInitializeSizes.
%=============================================================
% Function mdlOutputs performs the calculations.
%=============================================================
function sys = mdlOutputs(u);


[sys]=uav_intercepted(u);

% End of mdlOutputs.
```

UAV_Intercepted

```
%Authored by Zachary Spritzer and Matthew Lechliter

function [sys]=uav_intercepted(u)

uav_pos=reshape(u([1:27],1),3,9);
threat_pos=reshape(u([28:87],1),4,15);

uav_shot_down=zeros(9,1);
threats_fired=zeros(15,1);
for i=1:9,
   for j=1:15,
      dist_uav_threat=sqrt(((uav_pos(1,i)-threat_pos(1,j))^2)+((uav_pos(2,i)-threat_pos(2,j))^2));
      if dist_uav_threat < threat_pos(3,j),
         threats_fired(j,1)=1;
         uav_chance=rand;
         if uav_chance <= threat_pos(4,j),
            uav_shot_down(i,1)=1;
         end
      end
   end
end
sys=[uav_shot_down; threats_fired];
```

Target Related Functions

Target_Classifier_S

```
%Authored by Zachary Spritzer and Matthew Lechliter

function [sys,x0,str,ts] = target_classifier_s(t,x,u,flag,T)
% Dispatch the flag. The switch function controls the calls to
% S-function routines at each simulation stage.
switch flag,

   case 0
      [sys,x0,str,ts] = mdlInitializeSizes(T); % Initialization
```

```
   case 3
      sys = mdlOutputs(u); % Calculate outputs

   case { 1, 2, 4, 9 }
      sys = []; % Unused flags

   otherwise
      error(['Unhandled flag = ',num2str(flag)]); % Error handling
end;

%=============================================================
% Function mdlInitializeSizes initializes the states, sample
% times, state ordering strings (str), and sizes structure.
%=============================================================
function [sys,x0,str,ts] = mdlInitializeSizes(T)
% Call function simsizes to create the sizes structure.
sizes = simsizes;
% Load the sizes structure with the initialization information.
sizes.NumContStates= 0;
sizes.NumDiscStates= 0;
sizes.NumOutputs=    36;
sizes.NumInputs=     100;
sizes.DirFeedthrough=1;
sizes.NumSampleTimes=1;
% Load the sys vector with the sizes information.
sys = simsizes(sizes);
%
x0 = []; % No continuous states
%
str = []; % No state ordering
%
ts = [T 0]; % Inherited sample time
% End of mdlInitializeSizes.
%=============================================================
% Function mdlOutputs performs the calculations.
%=============================================================
function sys = mdlOutputs(u);


[sys]=target_classifier(u);


% End of mdlOutputs.
```

Target_Classifier

```
%Authored by Zachary Spritzer and Matthew Lechliter

function [sys]=target_classifier(u)


TARGETS_OLD=u([1:36],1);
TARGETS_OLD=reshape(TARGETS_OLD,4,9);
```

```
END_OF_PATH=u([37:45],1);

SELECTED_TARGETS=u([46:54],1);

TARGETS_REAL=u([55:90],1);
TARGETS_REAL=reshape(TARGETS_REAL,4,9);

target_location=u([91:99],1);

clock=round(u(100,1));

uav_complete=find(END_OF_PATH==1);
nuav_complete=size(uav_complete,2);
action=0;
for i=1:nuav_complete,
   target_real_location=target_location(SELECTED_TARGETS(uav_complete(1,i),1));
   action=TARGETS_REAL(4,target_real_location);
   if TARGETS_REAL(4,target_real_location) < 4,
      TARGETS_REAL(4,target_real_location)=TARGETS_REAL(4,target_real_location)+1;
   else
      TARGETS_REAL(:,target_real_location)=0;
   end
   if action==1,
      target_present=rand;
      if target_present <= 1.1,
         disp(sprintf('Target %d (value %d) indentified as a target at time %d by UAV %d. \n',...
         target_real_location,TARGETS_REAL(3,target_real_location),clock,uav_complete(1,i)));
      else
         disp(sprintf('Target %d (value %d) indentified as NOT a target at time %d by UAV %d.',...
         target_real_location,TARGETS_REAL(3,target_real_location),clock,uav_complete(1,i)));
         disp(sprintf('Target %d has been removed from target status at time %d.\n',...
         target_real_location,clock));
         TARGETS_REAL(:,target_real_location)=0;
      end
   end
   if action==2, disp(sprintf('Target %d (value %d) classified not attacked at time %d by UAV %d. \n',...
         target_real_location,TARGETS_REAL(3,target_real_location),clock,uav_complete(1,i))); end
   if action==3, disp(sprintf('Target %d (value %d) attacked not assested at time %d by UAV %d. \n',...
         target_real_location,TARGETS_REAL(3,target_real_location),clock,uav_complete(1,i))); end
   if action==4,
      target_destroyed=rand;
      if target_destroyed <= 1.1,
         disp(sprintf('Target %d (value %d) assested as destroyed at time %d by UAV %d. \n',...
            target_real_location,TARGETS_REAL(3,target_real_location),clock,uav_complete(1,i)));
      else
         disp(sprintf('Target %d (value %d) assested as NOT destroyed at time %d by UAV %d. \n',...
            target_real_location,TARGETS_REAL(3,target_real_location),clock,uav_complete(1,i)));
         TARGETS_REAL(4,target_real_location)=3;
      end
   end
end

if sum(sum(TARGETS_REAL))==0,
   TARGETS_REAL(:,1)=[4 2 3 1]';
end
```

```
sys=reshape(TARGETS_REAL,36,1);
```

Place_Waypoints_S

```
%Authored by Zachary Spritzer and Matthew Lechliter

function [sys,x0,str,ts] =place_waypoints_s(t,x,u,flag,T)
% Dispatch the flag. The switch function controls the calls to
% S-function routines at each simulation stage.
switch flag,

   case 0
      [sys,x0,str,ts] = mdlInitializeSizes(T); % Initialization

   case 3
      sys = mdlOutputs(u); % Calculate outputs

   case { 1, 2, 4, 9 }
      sys = []; % Unused flags

   otherwise
      error(['Unhandled flag = ',num2str(flag)]); % Error handling
end;

%=============================================================
% Function mdlInitializeSizes initializes the states, sample
% times, state ordering strings (str), and sizes structure.
%=============================================================
function [sys,x0,str,ts] = mdlInitializeSizes(T)
% Call function simsizes to create the sizes structure.
sizes = simsizes;
% Load the sizes structure with the initialization information.
sizes.NumContStates= 0;
sizes.NumDiscStates= 0;
sizes.NumOutputs=    9*4+9;
sizes.NumInputs=     9*4+9*4;
sizes.DirFeedthrough=1;
sizes.NumSampleTimes=1;
% Load the sys vector with the sizes information.
sys = simsizes(sizes);
%
x0 = []; % No continuous states
%
str = []; % No state ordering
%
ts = [T 0]; % Inherited sample time
% End of mdlInitializeSizes.
%=============================================================
% Function mdlOutputs performs the calculations.
%=============================================================
function sys = mdlOutputs(u);


[sys]=place_waypoints(u);
```

```
% End of mdlOutputs.
```

Place_Waypoints

```
%Authored by Zachary Spritzer and Matthew Lechliter

function [sys]=place_waypoints(u)

UAVS=u([1:36],1);
UAVS=reshape(UAVS,4,9);


uavs_existing=zeros(1,9);
for i=1:9
   if abs(sum(UAVS(:,i)))>0 & abs(sum(UAVS(:,i)))~=0.26
      uavs_existing(1,i)=1;
  end
end

TARGETS_REAL=u([37:72],1);
TARGETS_REAL=reshape(TARGETS_REAL,4,9);
n_uav=0;n_targ=0;

TARGETS=zeros(4,9);
targets_location=zeros(1,9);
for i=1:9
   if abs(sum(UAVS(:,i)))>0 & abs(sum(UAVS(:,i)))~=0.26
      n_uav=n_uav+1;
   end
   if abs(sum(TARGETS_REAL(:,i)))>0
      n_targ=n_targ+1;
   end
end

if n_uav < n_targ
   for i = 1:n_uav
      A=TARGETS_REAL(3,:);
      B=sort(A);
      Column=find(A==B(1,size(B,2)));
      TARGETS(1,i) = TARGETS_REAL(1,Column(1,1));
      TARGETS(2,i) = TARGETS_REAL(2,Column(1,1));
      TARGETS(3,i) = TARGETS_REAL(3,Column(1,1));
      TARGETS(4,i) = TARGETS_REAL(4,Column(1,1));
      targets_location(1,i)=Column(1,1);
      TARGETS_REAL(3,Column(1,1))=0;
   end
else
   counter=1;
   for i=1:9
      if abs(sum(TARGETS_REAL(:,i)))>0
         TARGETS(:,counter)=TARGETS_REAL(:,i);
         targets_location(1,counter)=i;
         counter=counter+1;
```

```
        end
    end
end

if n_uav > n_targ
    for i=1:(n_uav-n_targ)
        A=TARGETS_REAL(3,:);
        B=sort(A);
        Column=find(A==B(1,size(B,2)));
        TARGETS(1,n_targ+i) = i*.01+TARGETS_REAL(1,Column(1,1));
        TARGETS(2,n_targ+i) = i*.01+TARGETS_REAL(2,Column(1,1));
        TARGETS(3,n_targ+i) = 0;
        TARGETS(4,n_targ+i) = 0;
        TARGETS_REAL(3,Column(1,1))=0.5*TARGETS_REAL(3,Column(1,1));
        targets_location(1,i+n_targ)=Column(1,1);
    end
end
TARGETS=[TARGETS,zeros(4,9-size(TARGETS,2))];


sys=[reshape(TARGETS,36,1);targets_location'];
```

Appendix D

Grid and Visibility Graph MATLAB Files

Grid Related Functions

Path_Planning_Grid_S

```
%Authored by Zachary Spritzer

function [sys,x0,str,ts] = path_planning_grid_s(t,x,u,flag,T)
% Dispatch the flag. The switch function controls the calls to
% S-function routines at each simulation stage.
switch flag,

  case 0
    [sys,x0,str,ts] = mdlInitializeSizes(T); % Initialization

  case 3
    sys = mdlOutputs(u); % Calculate outputs

  case { 1, 2, 4, 9 }
    sys = []; % Unused flags

  otherwise
    error(['Unhandled flag = ',num2str(flag)]); % Error handling
end;

%=============================================================
% Function mdlInitializeSizes initializes the states, sample
% times, state ordering strings (str), and sizes structure.
%=============================================================
function [sys,x0,str,ts] = mdlInitializeSizes(T)
% Call function simsizes to create the sizes structure.
sizes = simsizes;
% Load the sizes structure with the initialization information.
sizes.NumContStates= 0;
sizes.NumDiscStates= 0;
sizes.NumOutputs=    9*100*4+9;
sizes.NumInputs=     36+36+30+60+1+1+9;
sizes.DirFeedthrough=1;
sizes.NumSampleTimes=1;
% Load the sys vector with the sizes information.
sys = simsizes(sizes);
%
x0 = []; % No continuous states
%
str = []; % No state ordering
%
ts = [T 0]; % Inherited sample time
% End of mdlInitializeSizes.
%=============================================================
% Function mdlOutputs performs the calculations.
%=============================================================
function sys = mdlOutputs(u)

[sys]=path_planning_grid(u);

% End of mdlOutputs
```

Path_Planning_Grid

```
%Authored by Zachary Spritzer

function [out]=path_planning_grid(in)

UAVS_long=in([1:36],1);
UAVS_long=reshape(UAVS_long,4,9);
TARGETS_long=in([37:72]);
TARGETS_long=reshape(TARGETS_long,4,9);
ZONES_long=in([73:102]);
ZONES_long=reshape(ZONES_long,3,10);
THREATS_long=in([103:162]);
THREATS_long=reshape(THREATS_long,4,15);
TIME=in(163);
n_plots=in(164);
HEADING_ANGLE=in([165:173]);

uavs_existing=zeros(1,9);
for i=1:9
   if abs(sum(UAVS_long(:,i)))>0 & abs(sum(UAVS_long(:,i)))~=0.26
      uavs_existing(1,i)=1;
  end
end
[UAVS]=filter_zeros(UAVS_long,9);
n_uav=size(UAVS,2);

targ_existing=zeros(1,9);
for i=1:9
   if TARGETS_long(3,i)~=0,
      targ_existing(1,i)=1;
  end
end
[TARGETS_temp]=filter_zeros(TARGETS_long,9);
TARGETS=[TARGETS_temp(1,:);TARGETS_temp(2,:)];
n_targ=size(TARGETS,2);

[ZONES]=filter_zeros(ZONES_long,10);
n_zones=size(ZONES,2);

threats_existing=zeros(1,15);
for i=1:15
   if THREATS_long(3,i)~=0
      threats_existing(1,i)=1;
  end
end
[THREATS]=filter_zeros(THREATS_long,15);
n_threats=size(THREATS,2);

ZONES_REAL=ZONES;
THREATS_REAL=THREATS;

ZONES(3,:)=1.15*ZONES_REAL(3,:);
```

```
THREATS(3,:)=1.15*THREATS_REAL(3,:);



split_seg=10;
min_turn=1;
sz_grid=20;
[all_pos,all_lines_x,all_lines_y,all_costs]=grid_gen(UAVS,TARGETS,ZONES,THREATS,sz_grid);
[stored_paths,totalcost]=cheapest_paths(all_pos,all_lines_x,all_lines_y,all_costs,UAVS,TARGETS,ZONE
S,THREATS);
[Shortened_Paths_x,Shortened_Paths_y,totalcost]=path_shrtng(stored_paths,all_pos,ZONES,THREATS,m
in_turn,split_seg,n_uav,n_targ,HEADING_ANGLE);
[Selected_Paths_x,Selected_Paths_y]=mmkp_task_allocation(totalcost,Shortened_Paths_x,Shortened_Path
s_y,n_uav);
[uav_path_x,uav_path_y,time_uav,altitude_uav]=vrt_sim_convert(Selected_Paths_x,Selected_Paths_y,UA
VS,min_turn*2);
if n_plots~=0,

plot_uav(UAVS_long,TARGETS_long,ZONES_REAL,THREATS_long,uav_path_x,uav_path_y,n_plots,
uavs_existing,targ_existing,threats_existing);
end

disp(sprintf('Path Planning ran at time %d. \n',round(TIME)));

bestcomb=zeros(1,9);
for i=1:n_uav,
   for j=1:n_targ,
      if round(Selected_Paths_x(end,i)*10)==round(TARGETS(1,j)*10) &
round(Selected_Paths_y(end,i)*10)==round(TARGETS(2,j)*10)
         bestcomb(1,i)=j;
         break
      end
   end
end

%Making into vector
uav_x=zeros(9,100);
uav_y=zeros(9,100);
uav_time=zeros(9,100);
uav_alt=zeros(9,100);
selected_targets=zeros(9,1);
szpath=size(uav_path_x,2);
counter=1;
for i=1:9,
   if uavs_existing(1,i)==1
      selected_targets(i,1)=bestcomb(1,counter);
      uav_x(i,[1:szpath])=uav_path_x(counter,:);
      uav_y(i,[1:szpath])=uav_path_y(counter,:);
      uav_time(i,[1:szpath])=time_uav(counter,:)+TIME;
      uav_alt(i,[1:szpath])=altitude_uav(counter,:);
      counter=counter+1;
   end
end
sys_temp=[];
for i=1:9;
```

```
   sys_temp=[sys_temp,uav_x(i,:),uav_y(i,:),uav_alt(i,:),uav_time(i,:)];
end
out=[sys_temp,selected_targets'];
```

Grid_Gen

```
%Authored by Zachary Spritzer

function
[all_pos,all_lines_x,all_lines_y,all_costs]=grid_gen(UAVS,TARGETS,ZONES,THREATS,sz_grid)

%INPUTS:
%
%UAVS - is a 4xn matrix where n is number of UAVs, the first row is the
%initial x position of the UAVs, the second row is the initial y position
%of the UAVs, the third row is the initial altitude of the UAVs, and
%the fourth row is the intial Velocity of the UAVs.
%
%TARGETS - is a 2xn matrix where n is the number of Targets, the first row
%is the x position of the targets and the second row is the y position of
%the targets.
%
%ZONES - is a 3xn matrix where n is the number of No-Fly Zones, the first
%row is the x position of the no-fly zones, the second row is the y
%position of the no-fly zones, and the third row is the radius or range of
%the no-fly zones.
%
%THREATS - is a 4xn matrix where n is the number of Threats, the first row
%is the x position of the threats, the second row is the y position of the
%threats, the third row is the range of the threats, and the fourth row is
%the level of danger of the threats.
%
%OUTPUTS:
%
%all_pos - is a 2xn matrix where n is the number of unique voronoi points,
%uav points, and target points.  Where the first row is the x position and
%the second row is the y position of all of these unique points.
%
%all_lines_x - is a 2xn matrix where n is the number of all of the lines
%for the voronoi, uavs, and targets.  The first row is the ending point's
%x position for the nth line and the second row is the starting point's
%x position for the nthe line.
%
%all_lines_y - is a 2xn matrix where n is the number of all of the lines
%for the voronoi, uavs, and targets.  The first row is the ending point's
%y position for the nth line and the second row is the starting point's
%y position for the nthe line.
%
%all_costs - is a 1xn row where n is the number of all of the lines
%for the voronoi, uavs, and targets.  This row is the costs for all of the
%lines of all_lines_x and all_lines_y

max_x=max([TARGETS(1,:),UAVS(1,:),ZONES(1,:),THREATS(1,:)])+10;
min_x=min([TARGETS(1,:),UAVS(1,:),ZONES(1,:),THREATS(1,:)])-10;
```

```
max_y=max([TARGETS(2,:),UAVS(2,:),ZONES(2,:),THREATS(2,:)])+10;
min_y=min([TARGETS(2,:),UAVS(2,:),ZONES(2,:),THREATS(2,:)])-10;


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Generating Grid
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%Generating Grid points
vxyn=zeros(2,sz_grid^2);
grid_x_pnts=min_x+(((max_x-min_x)*[0:(sz_grid-1)])/(sz_grid-1));
grid_y_pnts=min_y+(((max_y-min_y)*[0:(sz_grid-1)])/(sz_grid-1));
for i=1:sz_grid,
   vxyn(1,[(i-1)*sz_grid+1:(i-1)*sz_grid+sz_grid])=grid_x_pnts;
   vxyn(2,[(i-1)*sz_grid+1:(i-1)*sz_grid+sz_grid])=ones(1,sz_grid)*grid_y_pnts(1,i);
end

%Generating Grid Lines
sz_lines=(sz_grid-1)*sz_grid;
vx=zeros(2,(sz_lines)*2);
vy=vx;
for i=1:sz_grid,
   vx(1,[(i-1)*((sz_grid-1)*2)+1:(i-1)*((sz_grid-1)*2)+(sz_grid-1)])=grid_x_pnts(1,[2:sz_grid]);
   vx(2,[(i-1)*((sz_grid-1)*2)+1:(i-1)*((sz_grid-1)*2)+(sz_grid-1)])=grid_x_pnts(1,[1:(sz_grid-1)]);
   vy(1,[(i-1)*((sz_grid-1)*2)+1:(i-1)*((sz_grid-1)*2)+(sz_grid-1)])=ones(1,sz_grid-1)*grid_y_pnts(1,i);
   vy(2,[(i-1)*((sz_grid-1)*2)+1:(i-1)*((sz_grid-1)*2)+(sz_grid-1)])=ones(1,sz_grid-1)*grid_y_pnts(1,i);

   vx(1,[(i-1)*((sz_grid-1)*2)+(sz_grid-1)+1:(i-1)*((sz_grid-1)*2)+(sz_grid-1)*2])=ones(1,sz_grid-
1)*grid_x_pnts(1,i);
   vx(2,[(i-1)*((sz_grid-1)*2)+(sz_grid-1)+1:(i-1)*((sz_grid-1)*2)+(sz_grid-1)*2])=ones(1,sz_grid-
1)*grid_x_pnts(1,i);
   vy(1,[(i-1)*((sz_grid-1)*2)+(sz_grid-1)+1:(i-1)*((sz_grid-1)*2)+(sz_grid-
1)*2])=grid_y_pnts(1,[2:(sz_grid)]);
   vy(2,[(i-1)*((sz_grid-1)*2)+(sz_grid-1)+1:(i-1)*((sz_grid-1)*2)+(sz_grid-
1)*2])=grid_y_pnts(1,[1:(sz_grid-1)]);

end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Connecting UAV's into grid
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
[line_cost_uav,uavx,uavy]=connect_vrn(vxyn',UAVS([1,2],:));

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Connecting the targets into the grid
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
[line_cost_targ,targx,targy]=connect_vrn(vxyn',TARGETS([1,2],:));

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Generation for grid line costs
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
nvlines=size(vx,2);
line_cost_vrn=zeros(1,nvlines);
for i=1:nvlines,
   line_cost_vrn(1,i)=sqrt((vx(1,i)-vx(2,i))^2+(vy(1,i)-vy(2,i))^2);
end
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Stacking unique positions, lines for x and y, and costs of those lines
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
all_pos=[UAVS([1,2],:) vxyn([1,2],:) TARGETS([1,2],:)];
all_lines_x=[uavx([1,2],:) vx([1,2],:) targx([1,2],:)];
all_lines_y=[uavy([1,2],:) vy([1,2],:) targy([1,2],:)];
all_costs=[line_cost_uav(1,:) line_cost_vrn(1,:) line_cost_targ(1,:)];
```

Visibility Related Functions

Path_Planning_Vis_S

```
%Authored by Zachary Spritzer

function [sys,x0,str,ts] = path_planning__vis_s(t,x,u,flag,T)
% Dispatch the flag. The switch function controls the calls to
% S-function routines at each simulation stage.
switch flag,

  case 0
    [sys,x0,str,ts] = mdlInitializeSizes(T); % Initialization

  case 3
    sys = mdlOutputs(u); % Calculate outputs

  case { 1, 2, 4, 9 }
    sys = []; % Unused flags

  otherwise
    error(['Unhandled flag = ',num2str(flag)]); % Error handling
end;

%=============================================================
% Function mdlInitializeSizes initializes the states, sample
% times, state ordering strings (str), and sizes structure.
%=============================================================
function [sys,x0,str,ts] = mdlInitializeSizes(T)
% Call function simsizes to create the sizes structure.
sizes = simsizes;
% Load the sizes structure with the initialization information.
sizes.NumContStates= 0;
sizes.NumDiscStates= 0;
sizes.NumOutputs=    9*100*4+9;
sizes.NumInputs=     36+36+30+60+1+1+9;
sizes.DirFeedthrough=1;
sizes.NumSampleTimes=1;
% Load the sys vector with the sizes information.
sys = simsizes(sizes);
%
x0 = []; % No continuous states
%
str = []; % No state ordering
%
```

```
ts = [T 0]; % Inherited sample time
% End of mdlInitializeSizes.
%=============================================================
% Function mdlOutputs performs the calculations.
%=============================================================
function sys = mdlOutputs(u)

[sys]=path_planning_vis_graph(u);

% End of mdlOutputs.
```

Path_Planning_Vis_Graph

```
%Authored by Zachary Spritzer

function [out]=path_planning_vis_graph(in)

UAVS_long=in([1:36],1);
UAVS_long=reshape(UAVS_long,4,9);
TARGETS_long=in([37:72]);
TARGETS_long=reshape(TARGETS_long,4,9);
ZONES_long=in([73:102]);
ZONES_long=reshape(ZONES_long,3,10);
THREATS_long=in([103:162]);
THREATS_long=reshape(THREATS_long,4,15);
TIME=in(163);
n_plots=in(164);
HEADING_ANGLE=in([165:173]);

uavs_existing=zeros(1,9);
for i=1:9
   if abs(sum(UAVS_long(:,i)))>0 & abs(sum(UAVS_long(:,i)))~=0.26
      uavs_existing(1,i)=1;
  end
end
[UAVS]=filter_zeros(UAVS_long,9);
n_uav=size(UAVS,2);

targ_existing=zeros(1,9);
for i=1:9
   if TARGETS_long(3,i)~=0,
      targ_existing(1,i)=1;
  end
end
[TARGETS_temp]=filter_zeros(TARGETS_long,9);
TARGETS=[TARGETS_temp(1,:);TARGETS_temp(2,:)];
n_targ=size(TARGETS,2);

[ZONES]=filter_zeros(ZONES_long,10);
n_zones=size(ZONES,2);

threats_existing=zeros(1,15);
for i=1:15
   if THREATS_long(3,i)~=0
```

```
      threats_existing(1,i)=1;
  end
end
[THREATS]=filter_zeros(THREATS_long,15);
n_threats=size(THREATS,2);

ZONES_REAL=ZONES;
THREATS_REAL=THREATS;

ZONES(3,:)=1.15*ZONES_REAL(3,:);
THREATS(3,:)=1.15*THREATS_REAL(3,:);

split_seg=10;
min_turn=1;
points=8;
[all_pos,all_lines_x,all_lines_y,all_costs]=vis_line_gen(UAVS,TARGETS,ZONES,THREATS,points);
[stored_paths,totalcost]=cheapest_paths_vis(all_pos,all_lines_x,all_lines_y,all_costs,UAVS,TARGETS,ZO
NES,THREATS);

[Shortened_Paths_x,Shortened_Paths_y,totalcost]=path_shrtng_vis(stored_paths,all_pos,ZONES,THREAT
S,min_turn,n_uav,n_targ,HEADING_ANGLE);
[Selected_Paths_x,Selected_Paths_y]=mmkp_task_allocation(totalcost,Shortened_Paths_x,Shortened_Path
s_y,n_uav);
[uav_path_x,uav_path_y,time_uav,altitude_uav]=vrt_sim_convert(Selected_Paths_x,Selected_Paths_y,UA
VS,min_turn*2);
if n_plots~=0,

plot_uav(UAVS_long,TARGETS_long,ZONES_REAL,THREATS_long,uav_path_x,uav_path_y,n_plots,
uavs_existing,targ_existing,threats_existing);
end

disp(sprintf('Path Planning ran at time %d. \n',round(TIME)));

bestcomb=zeros(1,9);
for i=1:n_uav,
   for j=1:n_targ,
      if round(Selected_Paths_x(end,i)*10)==round(TARGETS(1,j)*10) &
round(Selected_Paths_y(end,i)*10)==round(TARGETS(2,j)*10)
         bestcomb(1,i)=j;
         break
      end
   end
end

%Making into vector
uav_x=zeros(9,100);
uav_y=zeros(9,100);
uav_time=zeros(9,100);
uav_alt=zeros(9,100);
selected_targets=zeros(9,1);
szpath=size(uav_path_x,2);
counter=1;
for i=1:9,
   if uavs_existing(1,i)==1
      selected_targets(i,1)=bestcomb(1,counter);
      uav_x(i,[1:szpath])=uav_path_x(counter,:);
```

```
      uav_y(i,[1:szpath])=uav_path_y(counter,:);
      uav_time(i,[1:szpath])=time_uav(counter,:)+TIME;
      uav_alt(i,[1:szpath])=altitude_uav(counter,:);
      counter=counter+1;
   end
end
sys_temp=[];
for i=1:9;
   sys_temp=[sys_temp,uav_x(i,:),uav_y(i,:),uav_alt(i,:),uav_time(i,:)];
end
out=[sys_temp,selected_targets'];
```

Vis_line_gen

```
%Authored by Zachary Spritzer

function
[all_pos,all_lines_x,all_lines_y,all_costs]=vis_line_gen(UAVS,TARGETS,ZONES,THREATS,points);



n_threats=size(THREATS,2);
n_zones=size(ZONES,2);
n_uav=size(UAVS,2);
n_targets=size(TARGETS,2);
all_pos=zeros(2,points*(n_threats+n_zones));

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Generating all the points on each No-Fly Zone and Threat
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
t=(1/points:1/points:1)'*2*pi;
for i=1:n_zones,
   x=ZONES(3,i)*1.15*sin(t)+ZONES(1,i);
   y=ZONES(3,i)*1.15*cos(t)+ZONES(2,i);
   all_pos(1,[(i-1)*points+1:(i-1)*points+points])=x';
   all_pos(2,[(i-1)*points+1:(i-1)*points+points])=y';
end

for i=1:n_threats,
   x=THREATS(3,i)*1.15*sin(t)+THREATS(1,i);
   y=THREATS(3,i)*1.15*cos(t)+THREATS(2,i);
   all_pos(1,[(i-1)*points+1+points*n_zones:(i-1)*points+points+points*n_zones])=x';
   all_pos(2,[(i-1)*points+1+points*n_zones:(i-1)*points+points+points*n_zones])=y';
end

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Adding UAV and Target positions into all_pos
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
all_pos=[UAVS(1,:) all_pos(1,:) TARGETS(1,:);UAVS(2,:) all_pos(2,:) TARGETS(2,:)];


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Generating visibilty lines
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
ZONES_THREATS=[ZONES([1:3],:) THREATS([1:3],:)];
n_zones_threats=size(ZONES_THREATS,2);
a=1;
for i=1:size(all_pos,2),
   for j=1:size(all_pos,2),
      if i~=j,
         SF=sqrt(((all_pos(1,i)-all_pos(1,j))^2)+((all_pos(2,i)-all_pos(2,j))^2));
         for k=1:n_zones_threats ,
            SC=sqrt(((ZONES_THREATS(1,k)-all_pos(1,i))^2)+((ZONES_THREATS(2,k)-
all_pos(2,i))^2));
            FC=sqrt(((ZONES_THREATS(1,k)-all_pos(1,j))^2)+((ZONES_THREATS(2,k)-
all_pos(2,j))^2));
            SN=(SC^2+SF^2-FC^2)/(2*SF);
            if SN<SF & SN>0
               PC(1,k)=sqrt(SC^2-SN^2);
            else
               if SC<FC
                  PC(1,k)=SC;
               else
                  PC(1,k)=FC;
               end
            end
         end
         if PC(1,:)>ZONES_THREATS(3,:),
            all_lines_x(1,a)=all_pos(1,j);
            all_lines_x(2,a)=all_pos(1,i);
            all_lines_y(1,a)=all_pos(2,j);
            all_lines_y(2,a)=all_pos(2,i);
            a=a+1;
         end
      end
   end
end


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Generating straight line cost for visibilty lines
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Since there is an equal weight assigned to each line within a threat there
%is no additional weighting needed for these lines since entering a threat
%is associated with a probability of kill not how long a UAV is in the
%threat's range.
for i=1:size(all_lines_x,2);
   all_costs(1,i)=sqrt((all_lines_x(1,i)-all_lines_x(2,i))^2+(all_lines_y(1,i)-all_lines_y(2,i))^2);
end
```

Path_shrtng_vis

```
%Authored by Zachary Spritzer

function
[Shortened_Paths_x,Shortened_Paths_y,totalcost]=path_shrtng_vis(stored_paths,all_pos,ZONES,THREAT
S,min_turn,nuav,ntarg,HEADING_ANGLE)
```

```
%INPUTS:
%
%stored_paths - is a mxn matrix where m is the number of uavs times the
%number of targets and n is the length of the longest path.  The first row
%being the first path for the first uav and the last row being the last
%path for the last uav. The paths are output by node numbers coming from
%the implementation of dijkstra's algorithm.
%
%all_pos - is a 2xn matrix where n is the number of unique voronoi points,
%uav points, and target points.  Where the first row is the x position and
%the second row is the y position of all of these unique points.
%
%ZONES - is a 3xn matrix where n is the number of No-Fly Zones, the first
%row is the x position of the no-fly zones, the second row is the y
%position of the no-fly zones, and the third row is the radius or range of
%the no-fly zones.
%
%THREATS - is a 4xn matrix where n is the number of Threats, the first row
%is the x position of the threats, the second row is the y position of the
%threats, the third row is the range of the threats, and the fourth row is
%the level of danger of the threats.
%
%min_turn - minimum turning radius for the UAVs
%
%split_seg - number of segments to Split the voronoi lines into for the
%purpose of a more near-optimal solution
%
%nuav - number of UAVs
%
%ntarg - number of targets

%OUTPUTS:
%
%Shortened_Paths - is a nxmx2 matrix where n is the length of the longest
%path and m is the number of UAVs multiplied by the number of targets.
%The element (nxmx1) x position of the mth uav at point n.  The element
%(nxmx2) y position of the mth uav at point n.
%
%totalcost - is a mxn matrix where m is the number of uavs and n is the
%number of possible paths for each uav.  The element (m,n) of this matrix
%is the cost for the mth uav to take the nth path.
%
%Stored_Pos - is a nxmx2 matrix where n is the length of the longest
%path and m is the number of UAVs multiplied by the number of targets.
%The element (nxmx1) x position of the mth uav at point n.  The element
%(nxmx2) y position of the mth uav at point n.



szpths=size(stored_paths,2);
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Finding the corresponding x and y coordinates
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Stored_Pos_x=ones(szpths,nuav*ntarg);
Stored_Pos_y=ones(szpths,nuav*ntarg);
stored_paths(:,szpths+1)=0;
```

```
for i=1:nuav*ntarg,
   mnz=min(find(stored_paths(i,:)==0));
   Stored_Pos_x(1:mnz-1,i)=all_pos(1,stored_paths(i,1:mnz-1))';
   Stored_Pos_y(1:mnz-1,i)=all_pos(2,stored_paths(i,1:mnz-1))';
   Stored_Pos_x(mnz:end,i)=ones((szpths-mnz+1),1)*all_pos(1,stored_paths(i,mnz-1))';
   Stored_Pos_y(mnz:end,i)=ones((szpths-mnz+1),1)*all_pos(2,stored_paths(i,mnz-1))';
end

Shortened_Paths_x_end=ones(500,1)*Stored_Pos_x(szpths,:);
Shortened_Paths_y_end=ones(500,1)*Stored_Pos_y(szpths,:);
Shortened_Paths_x=[Stored_Pos_x;Shortened_Paths_x_end];
Shortened_Paths_y=[Stored_Pos_y;Shortened_Paths_y_end];



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Adding initial path based on heading angle
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
for i=1:nuav,
   for j=1:ntarg,
      [Shortened_Paths_x(:,((i-1)*ntarg)+j),Shortened_Paths_y(:,((i-1)*ntarg)+j)]=...
         heading_angle_paths([Shortened_Paths_x(:,((i-1)*ntarg)+j),Shortened_Paths_y(:,((i-
1)*ntarg)+j)],min_turn,HEADING_ANGLE(i,1),72);
   end
end


Shortened_Paths_x_old=Shortened_Paths_x;
Shortened_Paths_y_old=Shortened_Paths_y;
Shortened_Paths_x=[];
Shortened_Paths_y=[];
for j=1:size(Shortened_Paths_x_old,1)-1,
   if Shortened_Paths_x_old(j,:)==Shortened_Paths_x_old(j+1,:) &
Shortened_Paths_y_old(j,:)==Shortened_Paths_y_old(j+1,:),
      Shortened_Paths_x(j,:)=Shortened_Paths_x_old(j,:);
      Shortened_Paths_y(j,:)=Shortened_Paths_y_old(j,:);
      break
   else
     Shortened_Paths_x(j,:)=Shortened_Paths_x_old(j,:);
     Shortened_Paths_y(j,:)=Shortened_Paths_y_old(j,:);
   end
end


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%Updating the Costs
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%s
zsp_perm=size(Shortened_Paths_x,2);
permcost=zeros(nuav*ntarg,1);


for z=1:szsp_perm,
   [permcost(z,1)]=update_cost([Shortened_Paths_x(:,z),Shortened_Paths_y(:,z)],THREATS);
end
totalcost=reshape(permcost,ntarg,nuav)';
```

Appendix E

Search and Destroy MATLAB Files

Path Planning Related Functions

Path_Planning_Search_S

```
%Authored by Zachary Spritzer

function [sys,x0,str,ts] = path_planning_search_s(t,x,u,flag,T)
% Dispatch the flag. The switch function controls the calls to
% S-function routines at each simulation stage.
switch flag,

  case 0
    [sys,x0,str,ts] = mdlInitializeSizes(T); % Initialization

  case 3
    sys = mdlOutputs(u); % Calculate outputs

  case { 1, 2, 4, 9 }
    sys = []; % Unused flags

  otherwise
    error(['Unhandled flag = ',num2str(flag)]); % Error handling
end;

%=============================================================
% Function mdlInitializeSizes initializes the states, sample
% times, state ordering strings (str), and sizes structure.
%=============================================================
function [sys,x0,str,ts] = mdlInitializeSizes(T)
% Call function simsizes to create the sizes structure.
sizes = simsizes;
% Load the sizes structure with the initialization information.
sizes.NumContStates= 0;
sizes.NumDiscStates= 0;
sizes.NumOutputs=    9*100*4+9;
sizes.NumInputs=     36+36+180*3+9+1+81+1;
sizes.DirFeedthrough=1;
sizes.NumSampleTimes=1;
% Load the sys vector with the sizes information.
sys = simsizes(sizes);
%
x0 = []; % No continuous states
%
str = []; % No state ordering
%
ts = [T 0]; % Inherited sample time
% End of mdlInitializeSizes.
%=============================================================
% Function mdlOutputs performs the calculations.
%=============================================================
function sys = mdlOutputs(u)

[sys]=path_planning_search(u);
```

Path_Planning_Search

```
%Authored by Zachary Spritzer

function [out]=path_planning_search(in)

UAVS=reshape(in([1:36],1),4,9);
TARGETS=reshape(in([37:72],1),4,9);
uav_action=reshape(in([73:153],1),9,9);
waypoints_x=reshape(in([154:333],1),20,9);
waypoints_y=reshape(in([334:513],1),20,9);
waypoints_checked=reshape(in([514:693],1),20,9);
HEADING_ANGLE=in([694:702],1);
n_plots=in(703,1);
TIME=round(in(704,1));
disp(sprintf('Path Planning Search ran at %d. \n',TIME))
uavs_existing=zeros(9,1);
for i=1:9,
   if UAVS(3,i)~=0,
      uavs_existing(i,1)=1;
   end
end

targets_present=zeros(9,1);
for i=1:9,
   if TARGETS(4,i)~=0,
      targets_present(i,1)=1;
   end
end
n_targ=sum(targets_present(:,1));

Selected_paths_x=zeros(9,100);
Selected_paths_y=zeros(9,100);

%If no targets are present
for i=1:9,
   if uavs_existing(i,1)~=0,
      Selected_paths_x(i,1)=UAVS(1,i);
      Selected_paths_y(i,1)=UAVS(2,i);
      for j=1:20,
         if waypoints_checked(j,i)==0,
            Selected_paths_x(i,[2:100])=waypoints_x(j,i);
            Selected_paths_y(i,[2:100])=waypoints_y(j,i);
            break
         end
      end
   end
end


%If targets are present
if sum(targets_present(:,1))~=0,
   num_target_visits=zeros(9,1);
```

```
   visit_target_costs=zeros(9,9);
   Selected_paths_x_temp=zeros(9*9,100);
   Selected_paths_y_temp=zeros(9*9,100);
   uav_assignment=zeros(9,1);

   for i=1:9,
      if targets_present(i,1)==1,
         for j=1:9
            if uavs_existing(i,1)~=0,
               Selected_paths_x_temp((j-1)*9+i,1)=UAVS(1,j);
               Selected_paths_y_temp((j-1)*9+i,1)=UAVS(2,j);
               Selected_paths_x_temp((j-1)*9+i,[2:100])=TARGETS(1,i);
               Selected_paths_y_temp((j-1)*9+i,[2:100])=TARGETS(2,i);

               [Selected_paths_x_temp((j-1)*9+i,:),Selected_paths_y_temp((j-1)*9+i,:)]=...
                  heading_angle_paths([Selected_paths_x_temp((j-1)*9+i,:);Selected_paths_y_temp((j-
1)*9+i,:)]...
                  ,1,HEADING_ANGLE(j,1),72);

               %Defining Costs
               for n=1:99,
                  visit_target_costs(j,i)=visit_target_costs(j,i)+sqrt(((Selected_paths_x_temp((j-1)*9+i,n)-
Selected_paths_x_temp((j-1)*9+i,n+1))^2)+...
                     ((Selected_paths_y_temp((j-1)*9+i,n)-Selected_paths_y_temp((j-1)*9+i,n+1))^2));
               end
            end
         end
         num_target_visits(i,1)=4-TARGETS(4,i);
         visit_target_costs_temp=round(visit_target_costs*100);
         uav_to_target=round(sort(visit_target_costs(:,i)*100));
         for k=1:num_target_visits(i,1);
            if uav_action(j,i)==0,
               uav_assignment(find(visit_target_costs_temp(:,i)==uav_to_target(k)),1)=i;
            end
         end
      end
   end
   for i=1:9,
      if uav_assignment(i,1)~=0,
         Selected_paths_x(i,:)=Selected_paths_x_temp((i-1)*9+uav_assignment(i,1),:);
         Selected_paths_y(i,:)=Selected_paths_y_temp((i-1)*9+uav_assignment(i,1),:);
      end
   end
end

[uav_path_x,uav_path_y,time_uav,altitude_uav]=path_times(Selected_paths_x,Selected_paths_y,UAVS,0.
5,uavs_existing);
time_uav=time_uav+ones(size(time_uav,1),size(time_uav,2))*TIME;
if n_plots~=0,
   plot_uav(UAVS,TARGETS,uav_path_x,uav_path_y,n_plots,uavs_existing,targets_present);
end


sys_temp=[];
for i=1:9;
   sys_temp=[sys_temp,uav_path_x(i,:),uav_path_y(i,:),altitude_uav(i,:),time_uav(i,:),uavs_existing(i,1)];
```

```
end

out=[sys_temp];


% End of mdlOutputs.
```

Path_Times

```
%Authored by Zachary Spritzer

function
[uav_path_x,uav_path_y,time_uav,altitude_uav]=path_times(Selected_paths_x,Selected_paths_y,UAVS,di
stpast,uavs_existing)
%
%INPUTS:
%
%Selected_paths_x - is a n*m matrix where n=9 and m=90 path length.
%
%UAVS - is a 4xn matrix where n is number of UAVs, the first row is the
%initial x position of the UAVs, the second row is the initial y position
%of the UAVs, the third row is the initial altitude of the UAVs, and
%the fourth row is the intial Velocity of the UAVs.
%
%
%OUTPUTS:
%
%uav_path_x - is a mxn matrix where m is the number of uavs and n is the
%length of the longest path. These are the x coordinates of the paths.
%
%uav_path_y - is a mxn matrix where m is the number of uavs and n is the
%length of the longest path. These are the y coordinates of the paths.
%
%time_uav - is a mxn matrix where m is the number of uavs and n is the
%length of the longest path. These values correspond to the time at which
%the uavs are at coordinates x and y in uav_path_x and uav_path_y.
%
%altitude_uav - is a mxn matrix where m is the number of uavs and n is the
%length of the longest path. These values correspond to the altitudes that
%the uavs are at when they are at coordinates x and y in uav_path_x and
%uav_path_y.

uav_path_x=zeros(9,100);
uav_path_y=zeros(9,100);


for i=1:9,
   if uavs_existing(i,1)~=0,
      for j=1:100,
         if Selected_paths_x(i,j+1) == Selected_paths_x(i,j+2) & Selected_paths_y(i,j+1) ==
Selected_paths_y(i,j+2),
            lst_pnt_x=Selected_paths_x(i,j+1);
```

```
            nxtlst_pnt_x=Selected_paths_x(i,j);
            lst_pnt_y=Selected_paths_y(i,j+1);
            nxtlst_pnt_y=Selected_paths_y(i,j);
            dist_pnts=sqrt(((lst_pnt_x-nxtlst_pnt_x)^2)+((lst_pnt_y-nxtlst_pnt_y)^2));
            last_x=lst_pnt_x+((lst_pnt_x-nxtlst_pnt_x)*(distpast/dist_pnts));
            last_y=lst_pnt_y+((lst_pnt_y-nxtlst_pnt_y)*(distpast/dist_pnts));
            uav_path_x(i,[j+1:100])=last_x;
            uav_path_y(i,[j+1:100])=last_y;
            uav_path_x(i,j)=Selected_paths_x(i,j);
            uav_path_y(i,j)=Selected_paths_y(i,j);
            break
         else
            uav_path_x(i,j)=Selected_paths_x(i,j);
            uav_path_y(i,j)=Selected_paths_y(i,j);
         end
      end
   end
end

%Initializing matrixes
time_uav=zeros(9,100);
time_uav_temp=zeros(9,100);

%Time matrix
for i=1:9,
   if uavs_existing(i,1)~=0,
      for j=1:98,
         if uav_path_x(i,j) == uav_path_x(i,j+1) & uav_path_y(i,j) == uav_path_y(i,j+1),
            break
          end
          shr_dist(i,j)=sqrt((uav_path_x(i,j)-uav_path_x(i,j+1))^2+(uav_path_y(i,j)-uav_path_y(i,j+1))^2);
          time_uav_temp(i,j+1)=shr_dist(i,j)/UAVS(4,i);
       end

      time_uav(i,[2:100])=sum(time_uav_temp(i,:));

      for j=2:100,
         time_uav(i,j)=time_uav(i,j-1)+time_uav_temp(i,j);
      end
   end
end
time_uav=time_uav*1.01;

%Altitude matrix
altitude_uav=zeros(9,100);

for i=1:9,
   altitude_uav(i,:)=UAVS(3,i);
end
```

Waypoint_Gen

```
%Authored by Zachary Spritzer

function
[waypoint_x_pos,waypoint_y_pos,waypoint_pos_checked,waypoint_start]=waypoint_gen(UAVS,grid_lim
its,search_rad,n_uav)

%Limits of the battlefield
min_x=grid_limits(1,1);
max_x=grid_limits(1,2);
min_y=grid_limits(1,3);
max_y=grid_limits(1,4);

%Number of points equal to the in increments of the search radius of the
%vehicles from min to max y
gridypnts=min_y:search_rad*2:max_y;


n_waypoints=2*ceil(size(gridypnts,2)/n_uav);

waypoint_x_pos=zeros(9,n_waypoints);
waypoint_y_pos=zeros(9,n_waypoints);
waypoint_x_pos(1:n_uav,1)=min_x;
waypoint_x_pos(1:n_uav,2)=max_x;
n_points=0;
n_uav_points=0;

%Generating orignial x and y points
while n_points<size(gridypnts,2),
   for j=1:n_uav,
      n_points=n_points+1;
      if n_points>size(gridypnts,2),break;end
      waypoint_y_pos(j,[n_uav_points*2+1,n_uav_points*2+2])=ones(1,2)*gridypnts(1,n_points);
      if n_uav_points>=1,

waypoint_x_pos(j,[n_uav_points*2+1,n_uav_points*2+2])=[waypoint_x_pos(j,n_uav_points*2),waypoint_
x_pos(j,n_uav_points*2-1)];
      end
   end
   n_uav_points=n_uav_points+1;
end

%Adding corners to the paths with the minimum turn radius
waypoint_x_pos_temp=zeros(9,20);
waypoint_y_pos_temp=zeros(9,20);
for i=1:9,
   n_temp_points=1;
   for j=1:n_waypoints-1,
      if (waypoint_x_pos(i,j) == waypoint_x_pos(i,j+1)) & (waypoint_y_pos(i,j) ~=
waypoint_y_pos(i,j+1)),
         waypoint_x_pos_temp(i,n_temp_points)=waypoint_x_pos(i,j);
         waypoint_y_pos_temp(i,n_temp_points)=waypoint_y_pos(i,j);
         n_temp_points=n_temp_points+1;
         if waypoint_x_pos(i,j) == min_x,
            waypoint_x_pos_temp(i,n_temp_points)=min_x-((waypoint_y_pos(i,j+1)-waypoint_y_pos(i,j)));
```

```
            waypoint_y_pos_temp(i,n_temp_points)=((waypoint_y_pos(i,j+1)-
waypoint_y_pos(i,j))/2)+waypoint_y_pos(i,j);
         else
            waypoint_x_pos_temp(i,n_temp_points)=max_x+(waypoint_y_pos(i,j+1)-waypoint_y_pos(i,j));
            waypoint_y_pos_temp(i,n_temp_points)=((waypoint_y_pos(i,j+1)-
waypoint_y_pos(i,j))/2)+waypoint_y_pos(i,j);
         end
         n_temp_points=n_temp_points+1;
         waypoint_x_pos_temp(i,n_temp_points:n_waypoints+n_temp_points-j-
1)=waypoint_x_pos(i,j+1:n_waypoints);
         waypoint_y_pos_temp(i,n_temp_points:n_waypoints+n_temp_points-j-
1)=waypoint_y_pos(i,j+1:n_waypoints);
      else
         waypoint_x_pos_temp(i,n_temp_points)=waypoint_x_pos(i,j);
         waypoint_y_pos_temp(i,n_temp_points)=waypoint_y_pos(i,j);
         n_temp_points=n_temp_points+1;
      end
   end
end

figure(102)
hold on
plot(waypoint_x_pos_temp(1,1:14),waypoint_y_pos_temp(1,1:14),'r')
plot(waypoint_x_pos_temp(2,1:14),waypoint_y_pos_temp(2,1:14),'k')
plot(waypoint_x_pos_temp(3,1:11),waypoint_y_pos_temp(3,1:11),'c')
plot(waypoint_x_pos_temp(4,1:11),waypoint_y_pos_temp(4,1:11),'g')
plot(waypoint_x_pos_temp(5,1:11),waypoint_y_pos_temp(5,1:11),'b')
plot(waypoint_x_pos_temp(6,1:11),waypoint_y_pos_temp(6,1:11),'m')
for i=1:6
   plot(UAVS(1,i),UAVS(2,i),'b*');
end
axis([-15 65 -5 55])
xlabel('Kilometers')
ylabel('Kilometers')
hold off

waypoint_x_pos=waypoint_x_pos_temp;
waypoint_y_pos=waypoint_y_pos_temp;
waypoint_start=zeros(2,9);
waypoint_start(1,:)=waypoint_x_pos(:,2)';
waypoint_start(2,:)=waypoint_y_pos(:,2)';
waypoint_start=reshape(waypoint_start,18,1);

waypoint_x_pos_temp=reshape(waypoint_x_pos',20*9,1);
waypoint_y_pos_temp=reshape(waypoint_y_pos',20*9,1);
n_waypoints=size(waypoint_x_pos,2);
waypoint_x_pos=zeros(20*9,1);
waypoint_y_pos=zeros(20*9,1);
waypoint_pos_checked=zeros(20,9);
waypoint_pos_checked(1,:)=1;
waypoint_pos_checked=reshape(waypoint_pos_checked,20*9,1);

for i=1:9
   waypoint_x_pos((i-1)*20+1:(i-1)*20+n_waypoints)=waypoint_x_pos_temp((i-1)*n_waypoints+1:(i-
1)*n_waypoints+n_waypoints,1);
```

```
   waypoint_y_pos((i-1)*20+1:(i-1)*20+n_waypoints)=waypoint_y_pos_temp((i-1)*n_waypoints+1:(i-
1)*n_waypoints+n_waypoints,1);
end
```

Target and Waypoint Related Functions

UAV_Detect_Target_S

```
%Authored by Zachary Spritzer

function [sys,x0,str,ts] =uav_detect_target_s(t,x,u,flag,T)
% Dispatch the flag. The switch function controls the calls to
% S-function routines at each simulation stage.
switch flag,

   case 0
      [sys,x0,str,ts] = mdlInitializeSizes(T); % Initialization

   case 3
      sys = mdlOutputs(u); % Calculate outputs

   case { 1, 2, 4, 9 }
      sys = []; % Unused flags

   otherwise
      error(['Unhandled flag = ',num2str(flag)]); % Error handling
end;

%=============================================================
% Function mdlInitializeSizes initializes the states, sample
% times, state ordering strings (str), and sizes structure.
%=============================================================
function [sys,x0,str,ts] = mdlInitializeSizes(T)
% Call function simsizes to create the sizes structure.
sizes = simsizes;
% Load the sizes structure with the initialization information.
sizes.NumContStates= 0;
sizes.NumDiscStates= 0;
sizes.NumOutputs=    36+81+1;
sizes.NumInputs=     36+36+1+81;
sizes.DirFeedthrough=1;
sizes.NumSampleTimes=1;
% Load the sys vector with the sizes information.
sys = simsizes(sizes);
%
x0 = []; % No continuous states
%
```

```
str = []; % No state ordering
%
ts = [T 0]; % Inherited sample time
% End of mdlInitializeSizes.
%=============================================================
% Function mdlOutputs performs the calculations.
%=============================================================
function sys = mdlOutputs(u);


[sys]=uav_detect_target(u);

% End of mdlOutputs.
```

UAV_Detect_Target

```
%Authored by Zachary Spritzer

function [sys]=uav_detect_target(u)

uav_pos=reshape(u([1:36],1),4,9);
target_pos=reshape(u([37:72],1),4,9);
target_pos_old=target_pos;
clock=round(u(73,1));
uav_action=reshape(u([74:154],1),9,9);

uavs_existing=zeros(9,1);
for i=1:9,
   if uav_pos(3,i)~=0,
      uavs_existing(i,1)=1;
   end
end

targets_present=zeros(9,1);
for i=1:9,
   if target_pos(1,i)~=0,
      targets_present(i,1)=1;
   end
end

for i=1:9,
   if uavs_existing(i,1)~=0,
      for j=1:9,
         if targets_present(j,1)~=0,
            dist_uav_target=sqrt(((uav_pos(1,i)-target_pos(1,j))^2)+((uav_pos(2,i)-target_pos(2,j))^2));
            if dist_uav_target < 1 & uav_action(i,j)==0,
               action=target_pos(4,j);
               if action==0,
                  disp(sprintf('Target %d (value %d) indentified at time %d by UAV %d. \n',...
                     j,target_pos(3,j),clock,i));
                  target_pos(4,j)=1;
               end
               if dist_uav_target < 0.1 & uav_action(i,j)==0,
                  if action==1,
                     target_present=rand;
                     if target_present <= .9,
```

```
                    disp(sprintf('Target %d (value %d) indentified as a target at time %d by UAV %d. \n',...
                        j,target_pos(3,j),clock,i));
                    target_pos(4,j)=2;
                else
                    disp(sprintf('Target %d (value %d) indentified as NOT a target at time %d by UAV
%d.',...
                        j,target_pos(3,j),clock,i));
                    disp(sprintf('Target %d has been removed from target status at time %d.\n',...
                        j,clock));
                    target_pos(:,j)=0;
                end
            end
            if action==2, disp(sprintf('Target %d (value %d) classified not attacked at time %d by UAV
%d. \n',...
                    j,target_pos(3,j),clock,i));
                target_pos(4,j)=3;
            end
            if action==3, disp(sprintf('Target %d (value %d) attacked not assested at time %d by UAV
%d. \n',...
                    j,target_pos(3,j),clock,i));
                target_pos(4,j)=4;
            end
            if action==4,
                target_destroyed=rand;
                if target_destroyed <= .85,
                    disp(sprintf('Target %d (value %d) assested as destroyed at time %d by UAV %d. \n',...
                        j,target_pos(3,j),clock,i));
                    target_pos(:,j)=0;
                else
                    disp(sprintf('Target %d (value %d) assested as NOT destroyed at time %d by UAV %d.
\n',...
                        j,target_pos(3,j),clock,i));
                    target_pos(4,j)=3;
                end
            end
        end
    end
  end
end

plan=(target_pos_old~=target_pos);
replan=sum(sum(plan));

sys=[reshape(target_pos,36,1);replan;reshape(uav_action,9*9,1)];
```

UAV_Detect_Waypoints_S

```
%Authored by Zachary Spritzer

function [sys,x0,str,ts] =uav_detect_waypoints_s(t,x,u,flag,T)
% Dispatch the flag. The switch function controls the calls to
% S-function routines at each simulation stage.
```

```
switch flag,

   case 0
      [sys,x0,str,ts] = mdlInitializeSizes(T); % Initialization

   case 3
      sys = mdlOutputs(u); % Calculate outputs

   case { 1, 2, 4, 9 }
      sys = []; % Unused flags

   otherwise
      error(['Unhandled flag = ',num2str(flag)]); % Error handling
end;

%=============================================================
% Function mdlInitializeSizes initializes the states, sample
% times, state ordering strings (str), and sizes structure.
%=============================================================
function [sys,x0,str,ts] = mdlInitializeSizes(T)
% Call function simsizes to create the sizes structure.
sizes = simsizes;
% Load the sizes structure with the initialization information.
sizes.NumContStates= 0;
sizes.NumDiscStates= 0;
sizes.NumOutputs=    180*3+1;
sizes.NumInputs=     36+180*3;
sizes.DirFeedthrough=1;
sizes.NumSampleTimes=1;
% Load the sys vector with the sizes information.
sys = simsizes(sizes);
%
x0 = []; % No continuous states
%
str = []; % No state ordering
%
ts = [T 0]; % Inherited sample time
% End of mdlInitializeSizes.
%=============================================================
% Function mdlOutputs performs the calculations.
%=============================================================
function sys = mdlOutputs(u);
[sys]=uav_detect_waypoints(u);
% End of mdlOutputs.
```

UAV_Detect_Waypoints

```
%Authored by Zachary Spritzer

function [sys]=uav_detect_waypoints(u)

uav_pos=reshape(u([1:36],1),4,9);
waypoint_x=reshape(u([37:216],1),20,9);
```

```
waypoint_y=reshape(u([217:396],1),20,9);
waypoints_checked=reshape(u([397:576],1),20,9);
waypoints_checked_old=waypoints_checked;

for i=1:9,
   for j=1:20,
      if waypoints_checked(j,i) == 1,
         dist_uav_waypoint=sqrt(((uav_pos(1,i)-waypoint_x(j+1,i))^2)+((uav_pos(2,i)-
waypoint_y(j+1,i))^2));
         if dist_uav_waypoint < .1,
            waypoints_checked(j+1,i)=1;
         end
         break
      end
   end
end

plan=(waypoints_checked_old~=waypoints_checked);
replan=sum(sum(plan));

sys=[reshape(waypoint_x,180,1);reshape(waypoint_y,180,1);reshape(waypoints_checked,180,1);replan];
```