Optimizing the Placement of Chemical and Biological Agent Sensors

D. Schafer

December 13, 2005

Abstract

A computer program was designed and implemented in Java to optimize the placement of chemical and biological (CB) agent sensors to best protect an installation. This was accomplished through the use of various optimization algorithms, combined with a Monte Carlo simulation method which determined relative utility functions for various sensor arrangements.

Contents

1 Introduction **4**

2 Design and Development **4**
2.1 Equipment 4
2.2 Procedures 4
2.2.1 Base Configuration 4
2.2.2 Sensor Arrangement 4
2.2.3 Plume Definition 5
2.2.4 Simulation Methodology 7
2.2.5 Evaluation Methods 7
2.2.6 Placement Algorithm 8
2.2.7 Allocation Algorithm 8

3 Data and Results **10**
3.1 Data 10
3.1.1 Program Speed 10
3.1.2 Sensor Grid Analysis 11
3.2 Results 14
3.3 Additional Work 14

4 Conclusions **14**

5 Acknowledgements **14**

6 Author Biography **15**

A Sample Code **15**

B Screenshots **21**

C Program Default Values **23**

D Mathematical Proofs **24**
D.1 Wind Velocity Spacing Relationship 24
D.2 Wind Velocity Spacing Relationship (General Case) 26
D.3 Circular Plume Analysis 27

References **30**

List of Figures

1 Different Sensor Configurations 5
2 Different Sensor Margins 6
3 Plume Diagram (from HPAC 4.04) 6
4 Bisection Method Function 9
5 Golden Section Function 9
6 Wind Velocity Spacing Relationship Diagram 24
7 Wind Speed vs. Optimal Spacing 25
8 Wind Velocity Spacing Relationship (General Case) Diagram 26
9 Circular Plume Analysis Diagram 27
10 Circular Plume Analysis Diagram - Rearrangement 28
11 P_{detect} vs. spacing 29
12 Derivative - P_{detect} vs. spacing 29

List of Tables

1 Optimization Time 10
2 Optimization Results (Small Plume) 11
3 Optimization Results (Large Plume) 12
4 Small Plume Threshold 12
5 Large Plume Threshold 12
6 Effect of Plume Spawn Region Width (Small Plume) 13
7 Effect of Plume Spawn Region Width (Large Plume) 13

1 Introduction

Weapons of Mass Effect pose a threat to any facility, especially military targets. Although nuclear attacks are often considered, chemical and biological attacks can be no less devastating. However, sensors exist that can detect such assaults, allowing the base to either avoid the attack (Detect-To-Warn situations) or treat any affected individuals and equipment (Detect-To-Treat situations). It is necessary to determine the most effective sensor placement to ensure that a base is well protected; unfortunately, as many of the tools that simulate such incidents are extremely detailed and complicated, running simulations on them to determine optimal sensor placement can take a prohibitively long amount of time. The development of a new tool that can both quickly simulate such attacks and optimize the placement of sensors to detect those attacks was needed for practical sensor optimization purposes.

The software to quickly simulate many attacks existed in a C++ application called "Sensor Geometry," which used a Monte Carlo simulation, randomly placing 500 potential biological attacks on a target, and then determined how effective the sensor arrangement was at detecting the attacks. However, using this tool to optimize sensor placement was still difficult, as thousands of individual scenarios would have to be manually entered to ensure accurate optimized results. A new application was developed, which used the original's basic methodology to automatically sample different sensor configurations until an optimal one could be determined.

2 Design and Development

2.1 Equipment

Although the original Sensor Geometry application was written in C++, conversion to Java had been a previously suggested enhancement[6]. The main benefit of Java was its platform independent capabilities, as one desired design feature for the new program was compatibility with many systems. Java is also an object-oriented language, which is useful for large-scale modeling problems. In order to ensure that older computers could run the program, all of the code was written to be compatible with an older, more common version of Java (J2SE 1.4.2, available since June 2003), so that users do not need to upgrade their Java installation.

The majority of the development work was done on a desktop computer using Windows 2000 Professional as its operating system, a 2.4 GHz Pentium 4 processor and 1 GB of RAM, but the application was also tested on computers running Windows XP, Mac OS X and Debian Linux, to ensure compatibility. Additionally, the program was written so that it could be run either as an application or as a web applet.

2.2 Procedures

2.2.1 Base Configuration

The major objective of the program is to determine the best way to place sensors to defend an installation, or "base." For the simulation, a base is considered to consist of three parts. The "defended area," which is shaded green in the diagrams, is the installation itself. The "safe area," colored blue, is a region that is not part of the base, but that cannot be the source of a biological assault. A real-life equivalent might be a large river on one side of the installation, or a fenced area surrounding the defended area. Many of the scenarios considered did not include safe areas. Finally, the "plume spawn area," in red, is the region where a chemical or biological attack might originate. Each of these regions is assumed to be a rectangle, though not necessarily all centered on the same point.

2.2.2 Sensor Arrangement

Many sensor configurations have been suggested. The three most commonly accepted arrangements are Perimeter, where all the sensors are along the outside of the base, Uniform, which has the sensors in a

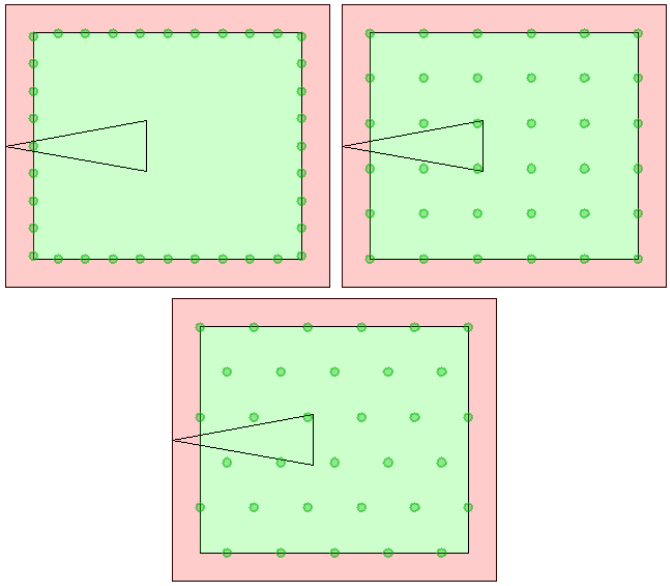


Figure 1: Different Sensor Configurations

grid across the base, and Dice-5, which has alternating rows slightly staggered (Figure 1). These three arrangements have mathematical justifications. A Perimeter setup has the smallest spacing between sensors while still surrounding the entire base. Uniform configurations are the most regular arrangement that covers the entire base, whereas Dice-5 arrangements place the sensors as close as possible to one another, a configuration that features prominently in optimal packing [4]. This may explain its popularity in sensor arrangements.

Additional sensors arrangements can be considered by introducing a margin, which pushes the grid further inside or outside the base (Figure 2). A positive margin restricts the sensors to a rectangle smaller than the defended region, and a negative margin allows sensors outside the defended area.

Although investigations had previously been made on other sensor arrangements, including Circular, Elliptical and Random configurations, it was observed that Perimeter, Uniform and Dice-5 were the better choices. Thus, in the optimization algorithm created, only these three configurations are considered.

One major difficulty in comparing the three arrangements is the variability of the number of sensors. Investigations showed that a sensor arrangement was best when it mirrored the dimensions of the defended region. For the nearly square defended region used in the data generation, it was discovered that the Uniform and Dice-5 layouts began dipping in efficiency when their number of rows and columns differed by more than two, so only configurations of form were used in the optimization routine, although the user can change this factor if they wish. This restriction forces Uniform and Dice-5 grid arrangements to skip some numbers that cannot be factored into two close integers.

2.2.3 Plume Definition

After a chemical or biological agent is released, it generally forms a plume, whose concentration contours are often in the shape of an ellipse (Figure 3). In the program, plumes (which are modeled as isosceles triangles, as algorithms to check area intersections for triangles are much faster than those for ellipses) are defined by four variables, two of which are placement-related, and two of which are dimension-related. The placement variables are the orientation of the plume, which corresponds to the wind direction and is between 0 and 360 degrees, and the spawn point of the plume, a Cartesian coordinate. The two dimension variables are the arc width of the plume (which, for the triangle, is the vertex angle) and the major axis length (which is

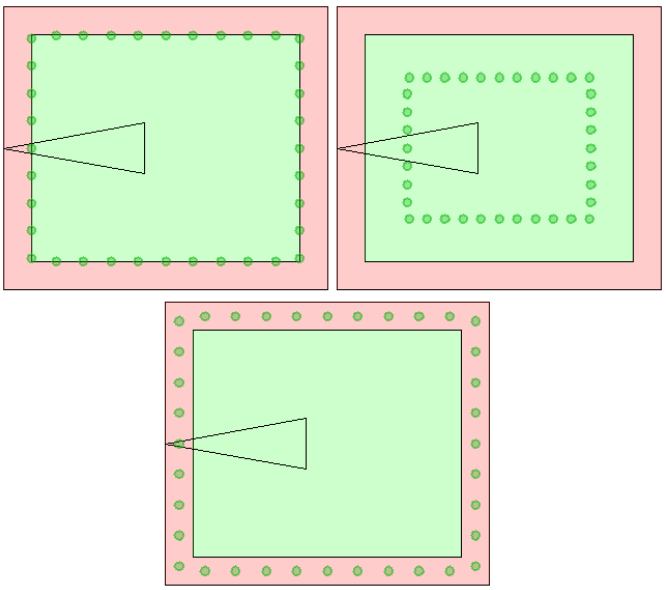


Figure 2: Different Sensor Margins

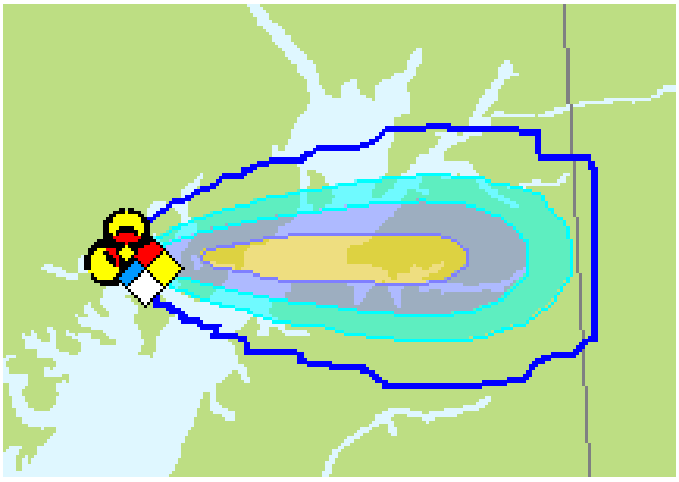


Figure 3: Plume Diagram (from HPAC 4.04)

the length of the altitude from the vertex). Although this format is easy to work with and configure, it does not lend itself towards developing realistic simulations, as plume dimensions in real life are determined by a number of factors of the weapon release. An algorithm had been previously developed [3] for converting real data to the program's plume entry format. This conversion is also used in the new version and enables reasonable simulation data, as actual wind speed data can be used to generate plumes.

In the original "Sensor Geometry" program, the plumes were dropped at a completely random orientation [7]. Although this is still an option, the new application also supports realistic wind data. The user can now input historical wind distribution information, allowing the program to tailor the distribution of plume directions accordingly. If no such data is available, the program reverts to the old functionality, choosing a direction between 0 and 360 degrees at random.

2.2.4 Simulation Methodology

The main methodology for the simulations existed in the original C++ program, but enhancements were made in the creation of the new application. The application uses a Monte Carlo simulation method, where a specified number of random attacks are made on the base. Each of these trials has a plume dropped randomly throughout the plume spawn area, although subject to two restrictions. All plumes must be dropped so that they do not originate inside the defended or safe area, as the program is designed to simulate external attacks. The safe area and defended area thus cannot completely cover the plume spawn area, as there would be no possible place where the plumes could be launched from. Additionally, any plume dropped that misses the defended area entirely is discarded, as detecting misses would not concern a base.

2.2.5 Evaluation Methods

After a scenario has been run, data is stored about each of the trials, including the number of sensors the plume contour intersected. However, as there are numerous ways of interpreting this data, five different scoring methods (also referred to as utility functions, or performance metrics) were used to rank sensor arrangements, some of which were normalized for easy comparison. The choice of which metric to use is left to the user, as different users will have different requirements for their sensor grid performances [5]. Variables that are used in formulae include:

The first scoring method, "One Hit," considers cases with at least one sensor triggered to be successes and non-detections to be failures. Its utility function is simply the ratio of plumes that were detected to the total number of trials. It is normalized between 0 and 1. The mathematical formula used to determine this score is:

$$Score_1 = 1 - \frac{N_0}{N}$$

The second scoring method, "Multi-Hit," considers plumes that are detected multiple times (at least twice) to be successes and gives those plumes a weight of $+1$. Plumes that are not detected are considered to be failures, and are given a weight of -1. Plumes detected once are not considered either way. Proponents of this scoring method argue that having only one sensor activate is not necessarily good, as false alarms are not uncommon with agent sensors [8]. The equation for "Multi-Hit," which is normalized between -1 and 1, is:

$$Score_2 = \frac{N_{>2} - N_0}{N}$$

The third scoring method, "Area-Weight," considers plumes that intersect large amounts of the base to be more important than those that barely hit the base. It also considers plumes that are detected multiple times to be successes, and plumes that are not detected to be failures, as in the "Multi-Hit" method, and it too is normalized between -1 and 1:

$$Score_3 = \frac{A_{>2} - A_0}{A}$$

The fourth method, "Power Law," considers each sensor triggered to be valuable, but offers diminishing returns on additional sensors. It uses a power law formula to reduce the value of each sensor, so that the first is worth 1, the second worth .5, the third worth .25, and so forth. This score is also normalized between 0 and 1:

$$Score_4 = \frac{\sum_{i=1}^{n} \frac{2^{D_i}-1}{2^{D_i-1}}}{2N}$$

The final method, "Avg. Hits," simply considers the average number of detections for each case. It is not normalized, and is difficult to use in comparisons. Although included in the program, no optimizations were done with this utility function, as it proved to lack sensitivity in its evaluations:

$$Score_5 = \frac{\sum_{i=1}^{n} D_i}{N}$$

2.2.6 Placement Algorithm

Two optimization approaches are considered to support two different potential users. Both optimization approaches allow the user to configure settings, including the maximum and minimum margin and number of sensors, and the scoring method to be used in the optimization. Initial attempts at designing the optimization algorithm showed that an exhaustive search of each sensor arrangement would take far too long, but investigations into better optimization algorithms succeeded in reducing the calculation time greatly. Appendix A contains selected code excerpts.

The first optimization method ("Placement") would be of most use to a base commander, who has received an allotment of sensors and wishes to distribute them for maximal effect. This optimization algorithm runs through the three possible configurations and tries different numbers of sensors with each margin. It chooses the distribution which scored highest on the chosen evaluation method, using a lower number of sensors to break a tie. It has no need, therefore, for threshold input and does not consider any arrangements that cannot logically be the solution. For Perimeter arrangements the algorithm does not try any grids with more than three sensors less than the maximum number of sensors. Although adding a sensor would seemingly always increase the utility, experimental data showed that not always to be the case and checking slightly smaller numbers ensures that no possible solution is discarded. For Uniform and Dice-5 simulations, the program also checks the arrangements with the most sensors, which also ensures that a slightly smaller but better arrangement is not discarded without a test.

2.2.7 Allocation Algorithm

The second optimization method ("Allocation") is meant for a commander who is deciding how many sensors to distribute to a base, and was the focus of development. Here, the user specifies a specific evaluation "threshold" that the sensor distribution must meet; for example, it must detect eighty percent of all attacks. The program then finds the smallest number of sensors needed to meet this goal. Initially, the algorithm for this method used a complete search that simply checked every data point to find the optimal one; however, this was unfeasibly slow. To increase the speed, the algorithm was refined by using more advanced searching methods, including Binary Search and Golden Section Search.

This method attempts to find the solution by first performing a binary search on the number of sensors to find the place where the utility is first greater than the threshold. Once the maximum utility value for the number of sensors has been determined, the "bisection method" algorithm [2] narrows its search field depending on whether the value was above or below the threshold. Figure 4 shows an example utility function , where $f(n)$ is the number of sensors, and n represents the maximum possible utility for that number of sensors. It is assumed that the intersection of the graph $f(n)$ and the horizontal line (the threshold) is between A and B. The utility function $f(n)$ is then evaluated at the midpoint of $[A, B]$, which is C, and the intersection region is narrowed to $[A, C]$, as is greater that the threshold. D, the midpoint of $[A, C]$, is then considered, and is evaluated. As D is less than the threshold, the possible intersection range is further

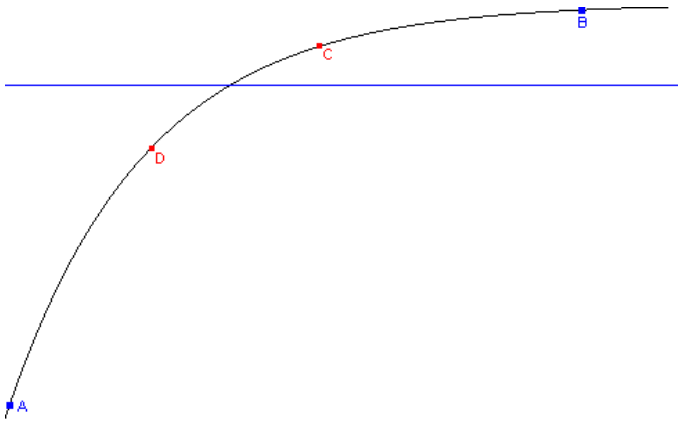


Figure 4: Bisection Method Function

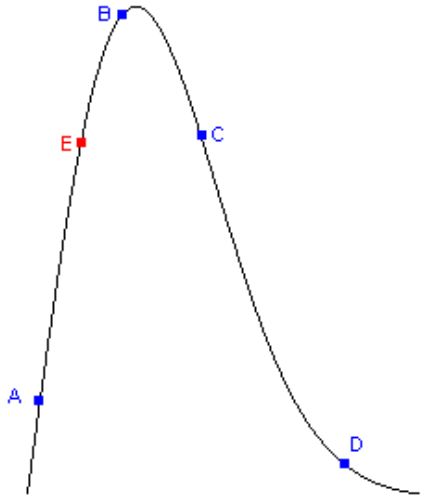


Figure 5: Golden Section Function

reduced to $[D, C]$. This could continue forever if any value for the number of sensors could be chosen, but because there can only be an integral sensor count, the search will end naturally when the search region reduces to size 1. The binary search (which runs in $O(\log_2(n))$ time) is possible because as the number of sensors increases, the utility function virtually always increases. Were this not true, a slower linear search (which runs in time) would be needed.

The method used to find the optimal margin (and thus the maximum utility value for a certain number of sensors) is a golden section search [10], which finds the maximum of a function. An example function $g(n)$ is shown in Figure 5, where m is a margin, and $g(m)$ is the utility function for that sensor grid (where the number of sensors is fixed).

The 2 end points, A and B, are considered at the start. It is assumed is that there is a single maximum in $[A, B]$. 2 interior points, C and D, are then chosen. As $B > C$, it is determined that the maximum must be in $[A, C]$. Two more points between A and C must be chosen, but because of the way the interior points are selected, B will be one of those two new interior points (the other is E). B is then compared to E, and the range is reduced to $[E, C]$. The search region is slowly reduced, and the method is cut off once the size

Trials	Placement Time (s)	Allocation Time (s)	
		Higher Threshold	Lower Threshold
500	97.531	100.734	70.203
1000	190.828	181.171	138.750
2500	451.300	493.816	356.252

Table 1: Optimization Time

of the search region has reached a certain accuracy level. This tolerance can be specified by the user, in order to ensure a selected level of accuracy. The following approach [11] shows how the points B and C are initially chosen:

$$\frac{B_x C_x}{A_x B_x} = \frac{A_x B_x}{B_x D_x} \tag{1}$$

Where B_x indicates the x-coordinate of the point B, etc. Using this ratio ensures that one of the two interior points is reused, regardless of how the algorithm narrows the search area. That equation simplifies to:

$$\left(\frac{B_x D_x}{A_x B_x}\right)^2 = \frac{B_x D_x}{A_x B_x} + 1 \tag{2}$$

$$\frac{B_x D_x}{A_x B_x} = \frac{1+\sqrt{5}}{2} \approx 1.618 \tag{3}$$

By using the golden ratio to select the interior points, only one calculation is required per iteration (as it can store the result of the calculation from the previous iteration), whereas using another search method would require two calculations of $g(m)$ [12]. Additionally, the code needed to create this program is not especially difficult [9]. As each calculation in this program's adaptation of this algorithm requires a scenario to be run, the time savings from using this search are quite substantial.

3 Data and Results

3.1 Data

3.1.1 Program Speed

One of the most important design requirements this project needed to meet in order to be useful was a speed threshold. The main reason that existing programs were not feasible to use for optimization was due to the length of time it took for them to finish. It was therefore necessary to examine the speed of the completed program.

Tests were done using 500, 1000, and 2500 trials per scenario, and the number of sensors logically had an approximately linear relationship with the speed of the program. These tests were done using the default values of the program, which are documented in Appendix C, and using Score 1 as the utility. Both "Placement" and "Allocation" trials were done. The results are shown in Table 1.

All of these timing trials were done on a Pentium 4 (2.4 GHz) machine with 1 GB of RAM. Timings will certainly vary depending on available computer components.

Considering that it can take over 100 seconds to do a single simulation of one plume using the standard agent modeling program HPAC (Hazard Prediction and Assessment Capability), these times are quite reasonable for entire optimizations.

Further analysis was done on the efficiency of running two optimizations at the same time on the same computer. It was discovered that the optimization time doubled in such situations; however, this work was done on a single processor, single core system. As each optimization runs in a separate thread, it would be

Scoring Method	Placement	Allocation	
		Higher Threshold	Lower Threshold
1 One Hit	(7,8) 56 sensors (Uniform) -1.019 margin. Score: 0.96	(6,5) 27 sensors (Dice-5) 0.056 margin. Score: 0.778	(3,5) 15 sensors (Uniform) 1.038 margin. Score: 0.522
2 Multi-Hit	(8,8) 60 sensors (Dice-5) -0.496 margin. Score: 0.744	(7,6) 42 sensors (Uniform) 0.0040 margin. Score: 0.502	(5,7) 35 sensors (Uniform) 0.366 margin. Score: 0.32
3 Area-Weight	(7,9) 60 sensors (Dice-5) 0.238 margin. Score: 0.972	(6,7) 39 sensors (Dice-5) 1.251 margin. Score: 0.802	(5,6) 30 sensors (Uniform) 2.927 margin. Score: 0.638
4 Power Law	(7,8) 56 sensors (Uniform) 1.038 margin. Score: 0.637	(5,7) 35 sensors (Uniform) 1.485 margin. Score: 0.505	18 sensors (Perimeter) 2.152 margin. Score: 0.306

Table 2: Optimization Results (Small Plume)

logical to assume that on a dual core or dual processor system, each optimization's thread would be assigned to a different core, so that the program's runtime would not change.

3.1.2 Sensor Grid Analysis

Initial results showed that the best geometry varied greatly, based on the user's input. Based on a user's specific needs, a different performance metric should be used. Forward fielded units, where an attack is somewhat likely would probably prefer a performance metric that considers any detection to be a success. Rear bases, where attacks are not expected, would greatly prefer multiple detections, as an alarm raised by a single sensor might be discarded as a false alarm. Some commanders might also consider detecting plumes that cover large parts of the base to be paramount, which the area-weighted utility function considers. Additionally, the power-law metric allows each sensor to be valued without greatly overweighting trials with huge numbers of detections. Thus, data was generated for each of the four main scoring methods. Additionally, trials were run both with a small plume, which used the default plume size with a major axis of 10 and an arc width of 20 degrees, and with a large plume, which extended the major axis length to 25 with the arc width fixed at 20 degrees.

Where not otherwise specified, all of the data was generated using the default parameters, as listed in Appendix C, but with 2500 trials, to ensure greater accuracy. The results for higher and lower polynomials are shown in Tables 2 and 3; the thresholds used are specified in Tables 4 and 5.

After analyzing this data, it was noted that the results tended not to differ from one another based on their grid preference, and thus, the next analysis was generated using Scoring Methods 1 and 3. The poor performance of Perimeter sensor grids in the small plume trials was postulated to be caused by the limited plume spawn area, as during these trials, it was assumed that the plume would spawn from within 2 km of the base. Thus, more analysis was done on the effect of the plume spawn area on the preferred setup. The width and height of the plume spawn region is equal to that of the defended region plus twice the "Plume Spawn Width," so that there is a border of the specified width surrounding the defended region. Other than these changes, the previously mentioned defaults were used. The results are shown in Tables 6 and 7.

Scoring Method	Placement	Allocation	
		Higher Threshold	Lower Threshold
1 One Hit	59 sensors (Perimeter) -1.104 margin. Score: 0.998	32 sensors (Perimeter) -0.415 margin. Score: 0.984	23 sensors (Perimeter) -1.104 margin. Score: 0.954
2 Multi-Hit	60 sensors (Perimeter) -0.542 margin. Score: 0.955	37 sensors (Perimeter) -0.496 margin. Score: 0.907	26 sensors (Perimeter) -0.629 margin. Score: 0.813
3 Area-Weight	60 sensors (Perimeter) 0.349 margin. Score: 1.0	24 sensors (Perimeter) 0.56 margin. Score: 0.995	16 sensors (Perimeter) 1.596 margin. Score: 0.908
4 Power Law	59 sensors (Perimeter) -0.204 margin. Score: 0.769	28 sensors (Perimeter) 0.877 margin. Score: 0.711	18 sensors (Perimeter) 0.268 margin. Score: 0.602

Table 3: Optimization Results (Large Plume)

Scoring Method	Higher Threshold	Lower Threshold
1	.75	.8
2	.5	.6
3	.5	.5
4	.25	.3

Table 4: Small Plume Threshold

Scoring Method	Higher Threshold	Lower Threshold
1	.98	.95
2	.9	.8
3	.99	.9
4	.7	.6

Table 5: Large Plume Threshold

Plume Spawn Width (km)	Score 1 (Threshold = .75)	Score 3 (Threshold = .80)
1	(5,6) 30 sensors (Uniform) 0.084 margin. Score: 0.833	(6,7) 39 sensors (Dice-5) 1.566 margin. Score: 0.811
2	(6,5) 27 sensors (Dice-5) 0.056 margin. Score: 0.778	(6,7) 39 sensors (Dice-5) 1.251 margin. Score: 0.802
3	(5,5) 25 sensors (Uniform) 0.349 margin. Score: 0.756	(6,7) 39 sensors (Dice-5) 1.631 margin. Score: 0.806
4	(4,6) 24 sensors (Uniform) 0.041 margin. Score: 0.754	(6,7) 42 sensors (Uniform) 1.612 margin. Score: 0.823
5	(5,5) 25 sensors (Uniform) 0.137 margin. Score: 0.751	(6,8) 45 sensors (Dice-5) 1.251 margin. Score: 0.843
6	(6,5) 27 sensors (Dice-5) -0.125 margin. Score: 0.764	(6,8) 45 sensors (Dice-5) 1.251 margin. Score: 0.821
7	28 sensors (Perimeter) 0.0040 margin. Score: 0.75	(6,8) 48 sensors (Uniform) 1.038 margin. Score: 0.83
8	(5,6) 28 sensors (Dice-5) -0.204 margin. Score: 0.753	(6,8) 48 sensors (Uniform) 0.911 margin. Score: 0.805
9	29 sensors (Perimeter) -0.125 margin. Score: 0.761	(6,8) 48 sensors (Uniform) 1.038 margin. Score: 0.823
10	28 sensors (Perimeter) -0.3 margin. Score: 0.768	(7,7) 49 sensors (Uniform) 1.287 margin. Score: 0.82

Table 6: Effect of Plume Spawn Region Width (Small Plume)

Plume Spawn Width (km)	Score 1 (Threshold = .95)	Score 3 (Threshold = .90)
1	26 sensors (Perimeter) -1.09 margin, Score: 0.954	20 sensors (Perimeter) 0.826 margin, Score: 0.957
2	23 sensors (Perimeter) -0.219 margin, Score: 0.959	19 sensors (Perimeter) 1.267 margin, Score: 0.958
3	17 sensors (Perimeter) -0.496 margin, Score: 0.95	18 sensors (Perimeter) 1.449 margin, Score: 0.961
4	16 sensors (Perimeter) -0.204 margin, Score: 0.953	18 sensors (Perimeter) 1.117 margin, Score: 0.966
5	16 sensors (Perimeter) -0.204 margin, Score: 0.95	17 sensors (Perimeter) 1.236 margin, Score: 0.955
6	16 sensors (Perimeter) -0.496 margin, Score: 0.95	17 sensors (Perimeter) 1.287 margin, Score: 0.958
7	(3,5) 15 sensors (Uniform) -0.014 margin, Score: 0.951	17 sensors (Perimeter) 1.812 margin, Score: 0.954
8	(3,5) 15 sensors (Uniform) 0.0040 margin, Score: 0.964	16 sensors (Perimeter) 1.94 margin, Score: 0.95
9	(3,5) 15 sensors (Uniform) -0.073 margin, Score: 0.955	(4,4) 16 sensors (Uniform) 0.481 margin, Score: 0.959
10	(3,5) 15 sensors (Uniform) -0.045 margin, Score: 0.965	(4,4) 16 sensors (Uniform) 0.78 margin, Score: 0.957

Table 7: Effect of Plume Spawn Region Width (Large Plume)

3.2 Results

The data gathered through optimization showed that the length of the plume relative to the size of the base played a major role in deciding which sensor arrangement would be best. In the initial optimization efforts, Uniform and Dice-5 grids were best against a small plume. However, Perimeter arrangements were clearly superior for the large plume situations, where the plume would undoubtedly stretch across the entire base. In these cases, the plume contour may be small enough to go undetected when it reaches the first edge, but it will be quite large when it reaches the second edge, and a Perimeter setup will likely have many sensors in the part of the second edge covered.

Data in the plume spawn region analysis showed a similar trend. With the large plumes, a Perimeter setup was preferred until the plume spawn region began getting large. Once the plume spawn region reached a large enough size, the plume contours were most likely not going to entirely cross the defended area, and the preferred grid changed to Uniform. With the small plumes, the plumes were never able to entirely cross the defended area regardless of the size of the plume spawn region, and thus the small plumes nearly always preferred the Uniform or Dice-5 configurations. The one exception was using a non-area-weighted scoring method with a large plume spawn region. Here, many plumes spawned far from the base, and thus were quite large when they reached even the first edge, meaning that they were detected by many Perimeter sensors. However, these plumes did not cover much of the base, which explains why they had little effect on the results for the area-weighted algorithm.

3.3 Additional Work

In addition to the work done on creating a computer simulation program, mathematical analysis of the problem was also performed. Three major discoveries were made during this work, each of whose full proofs appear in Appendix D. The first discovery related wind speed to sensor spacing, discovering that the spacing needed to achieve a specified performance was inversely proportional to the square root of the wind speed. This first proof assumed that the plume pointed directly at the base; however, a follow-up proof generalized this discovery for a plume pointing in any direction. The third mathematical model demonstrated the performance of a sensor grid in a windless situation, where the biological or chemical agent plume cloud assumes the form of a circle.

4 Conclusions

After analyzing the speed and efficiency of the program, the design requirements for the program were met. The application completes an entire analysis in a reasonable amount of time, and can perform different types of optimizations depending on the situation. The program is also flexible enough to deal with a variety of input parameters, allowing it to be used on many different bases.

Additionally, the analyses done with the program allowed some conclusions about the relative merits of sensor grid arrangements to be reached. In situations where the plume contour is expected to stretch through the entire base, a Perimeter configuration is preferable, whereas Uniform or Dice-5 is better in situations where only a limited part of the base will be affected. Additionally, the results found imply that Uniform and Dice-5 work well in similar situations, and that it is limits on the number of sensors that lead to preferences between these two grid types.

5 Acknowledgements

I would first like to thank my mentor, Dr. Gunn, for his numerous contributions to this project. Without his help, both in developing the original software and in making numerous suggestions for the new version, this project could not have gone forward. I would also like to thank everyone from the DTRA TDOA OR team, for their help with the project.

6 Author Biography

DANIEL L. SCHAFER is entering his senior year as part of the Class of 2006 at Thomas Jefferson High School for Science and Technology. He plans to continue his work in computer science in the future, pursuing a double major in Computer Science and Mathematics.

A Sample Code

This includes many samples of the actual code used in the creation of the application. Any code removed is either GUI modification code or repetitive code, and is replaced in the sample with an ellipsis.

`optimizeMinSensors()` is the optimization method which calls the methods needed to find the possible optimal configurations, then determines which one is best.

```
/**
     * This method calls the 3 optimizeMinSensor subfunctions:
     * {@link #optimizeMinSensorsPerimeter(int)}, and
     * {@link #optimizeMinSensorsUniformDice5(int, int)} twice
     * (once for Uniform and Dice-5).  It compares the 3 Scenarios returned,
     * and returns the one that has the least number of sensors.
     * <p>
     * Note that the {@link Scenario#clone()} calls are removed.  See that
     * javadoc entries for the resons behind this.
     *
     * @return  The scenario which has the least number of sensors among the 3
     *          returned by the optimizeMinSensors subfunctions.
     * @see Scenario#clone()
     */
    public Scenario optimizeMinSensors()
    {

        ...


        int currBestNumSensors = Integer.parseInt(getSensorControlTextField()[1].getText());
        Scenario currBestScenario = null;

        Scenario s1 = optimizeMinSensorsPerimeter(currBestNumSensors);
        if (!continueOptimization)
            return null;
        if (s1 != null)
        {
            ...
            if (s1.getSensorGrid().getNumSensors() < currBestNumSensors)
            {
                currBestNumSensors = s1.getSensorGrid().getNumSensors();
                currBestScenario = s1;
            }
        }

        Scenario s2 = optimizeMinSensorsUniformDice5(currBestNumSensors,
```

```
                SensorConstants.UNIFORM);
        ...
        if (s2 != null)
        {
            ...
            if (s2.getSensorGrid().getNumSensors() < currBestNumSensors)
            {
                currBestNumSensors = s2.getSensorGrid().getNumSensors();
                currBestScenario = s2;
            }
        }

        Scenario s3 = optimizeMinSensorsUniformDice5(currBestNumSensors,
                SensorConstants.DICE5);
        ...
        if (s3 != null)
        {
            ...
            if (s3.getSensorGrid().getNumSensors() < currBestNumSensors)
            {
                currBestNumSensors = s3.getSensorGrid().getNumSensors();
                currBestScenario = s3;
            }
        }

        ...

        return currBestScenario;
    }
```

`optimizeMinSensorsPeimeter()` is the optimization method which finds the smallest number of sensors needed to keep the score above a certain threshold, for Perimeter configurations.

```
/**
     * This method attempts to find a scenario that has a value greater than
     * the value held in the threshold text box, but with a few sensors as
     * possible.  If it cannot find such a scenario, it returns null.
     * <p>
     * This method only considers Perimeter sensor arrays.
     * <p>
     * This method runs a binary search over the number of sensors.  To
     * evaluate the score for each number of sensors, it finds the optimal
     * margin and score using {@link #findOptimalMargin}.  It then narrows
     * its search field as any binary search does: If the score is below the
     * threshold, it searches only above that number of sensors, otheriwse it
     * searches only below that number of sensors.
     *
     * @param m The maximum number of sensors for this method to consider.
     *          This is passed as a parameter so that if this method is
     *          part of a larger optimization, it can avoid checking scenarios
     *          with more sensors than the current best scenario.
```

```
 *
 * @return  A scenario whose value is greater than threshold, and which
 *          has as small a number of sensors as possible.  If no such
 *          scenario exists, this returns <code>null</code>.
 */
public Scenario optimizeMinSensorsPerimeter(int m)
{
    //LOOP variables
    int sensorMin = Integer.parseInt(
                            getSensorControlTextField()[0].getText());
    int sensorMax = m;
    double marginMin = Double.parseDouble(
                            getMarginControlTextField()[0].getText());
    double marginMax = Double.parseDouble(
                            getMarginControlTextField()[1].getText());

    double threshold = Double.parseDouble(
                            this.getThresholdTextField().getText());

    if (sensorMin < 1)
        sensorMin = 1;

    //GUI things
    ...

    double[] results = new double[2];

    int lower = sensorMin;
    int upper = sensorMax;
    int middle = 0;
    double upperValue = 0.0;
    double upperMargin = 0.0;
    boolean solutionFound = false;

    while (upper > lower || (upper == lower && !solutionFound))
    {
        middle = ((upper + lower)/2);
        ...

        results = findOptimalMargin(new Integer(middle),
                                    marginMin,marginMax);
        ...


        double middleScore = results[1];

        if (middleScore >= threshold)
        {
            upper = middle;
            upperMargin = results[0];
            upperValue = results[1];
```

```
            solutionFound = true;
        }
        else if (middleScore < threshold)
        {
            lower = middle + 1;
        }
        else //middleScore == threshold
        {
            upper = middle;
            upperMargin = results[0];
            upperValue = results[1];
            solutionFound = true;
            break;
        }

    }
    //The best score didn't pass the threshold, so return null
    if (!solutionFound)
    {
        ...

        return null;
    }

    ...


    SensorGrid sg = SensorGrid.createPerimeter(
                    myScenario.getDefendedArea(), middle,
                    1, false, results[1]);
    myScenario.setSensorGrid(sg);

    return myScenario;
}
```

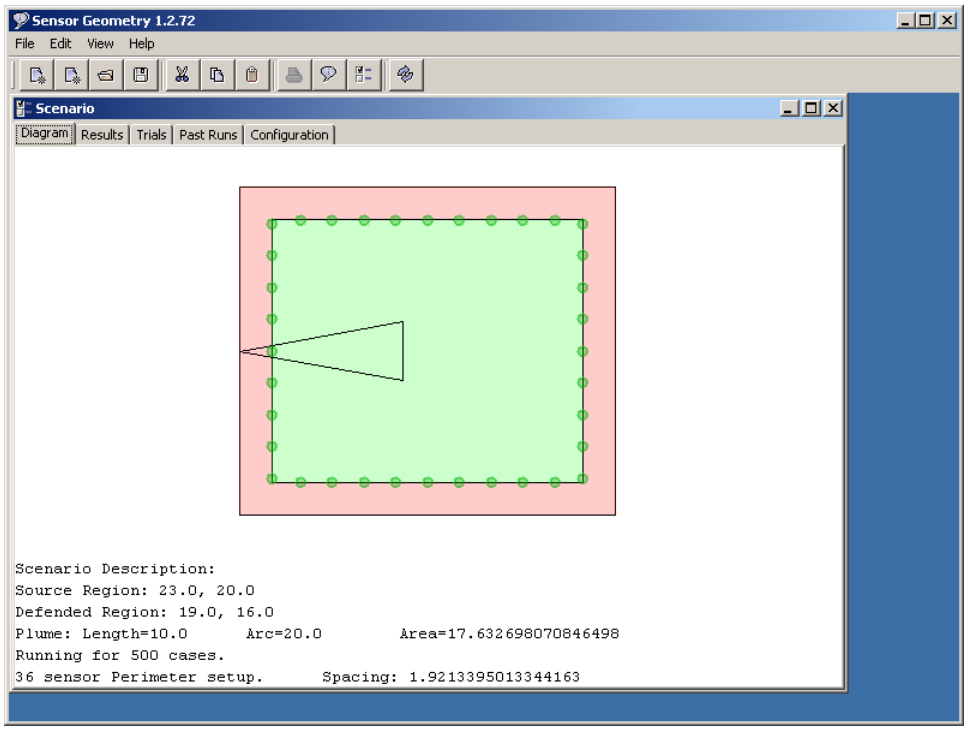
`findOptimalMargin()` is a helper method for the optimizeMinSensors() optimization, which finds the best possible margin given a certain number of sensors.

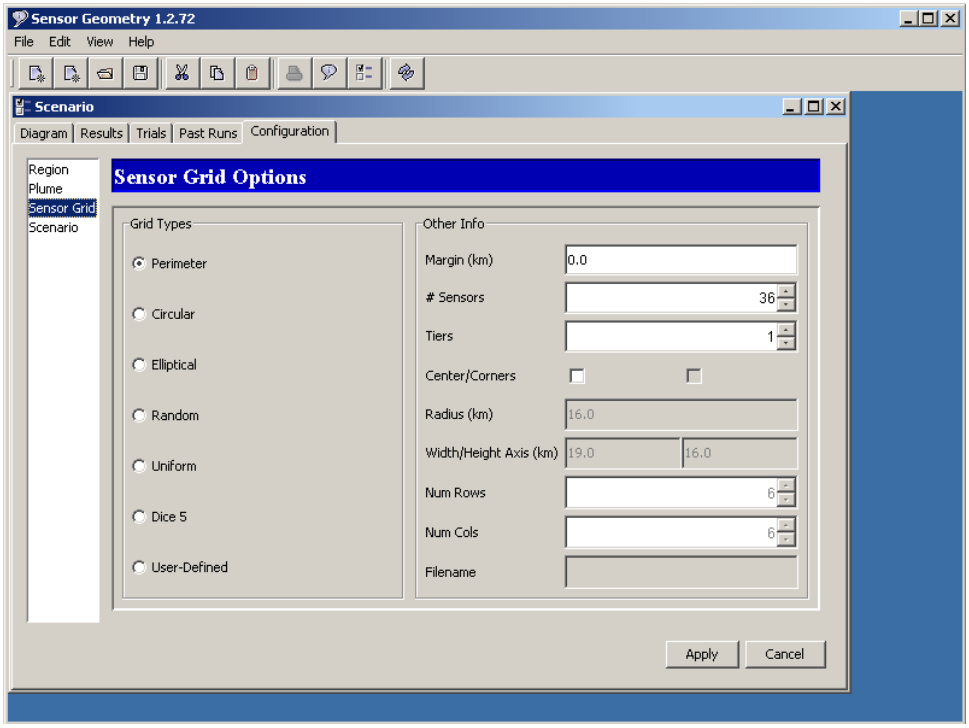
```
/**
 * This method finds the optimal margin for each number of sensors.  As the
 * graph of score against margin (with fixed number of sensors) is a
 * parabola, it tries to find the maximum value of this parabola.
 * It uses a Golden Ratio Search so that it only has to evaluate the
 * scenario at one point each time. The Golden ratio search cuts off
 * when it has narrowed the margin down to within <code>tolerance</code>.
 * <p>
 * The code for this method was simplified using the algorithm described at
 * http://www.cse.uiuc.edu/eot/modules/optimization/GoldenSection/.
 *
 * @param description   A description of the number of sensors or grid type
 *              for this function.  If description is an integer,
 *              it implies a Perimeter with that many sensors, if
```

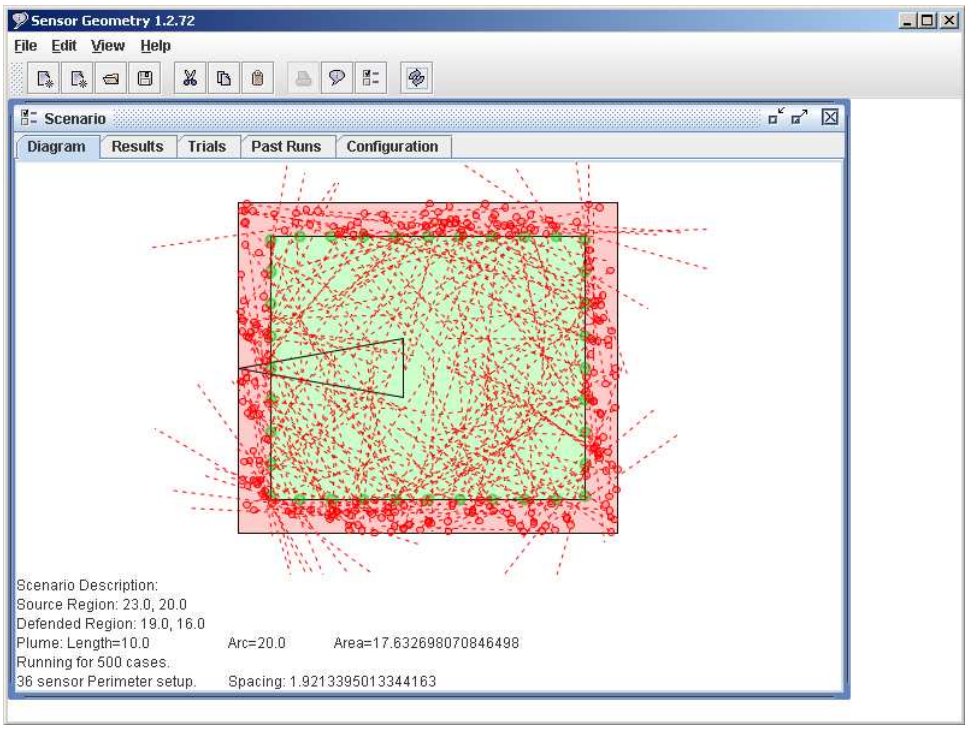
```
 *              it is a UniformDiceStorage, it implies a Uniform
 *              or Dice-5 arrangement.
 * @param minMargin     The minimum margin to consider.
 * @param maxMargin     The maximum margin to consider.
 *
 * @return  A double array of length 2.  Index 0 is the optimal margin,
 *          index 1 is the score when the margin is optimal.
 */
public double[] findOptimalMargin( Object description, double minMargin,
 double maxMargin)
{

    double[] best = new double[2];

    int scoreType = this.getScoringComboBox().getSelectedIndex();
    double tolerance = Double.parseDouble(
                      this.getToleranceTextField().getText());

    double middleOneMargin = 0;
    double middleOneScore = 0;
    double middleTwoMargin = 0;
    double middleTwoScore = 0;

    double r = (Math.sqrt(5) - 1) / 2;

    middleOneMargin = SensorConstants.decimalChange(
                            minMargin+(1-r)*(maxMargin-minMargin), 3);
    middleOneScore = evalFunction(
                            description, middleOneMargin, scoreType);
    middleTwoMargin = SensorConstants.decimalChange(
                            maxMargin-(1-r)*(maxMargin-minMargin), 3);
    middleTwoScore = evalFunction(
                            description, middleTwoMargin, scoreType);

    while (maxMargin - minMargin > tolerance)
    {
        ...

        if (middleTwoScore >= middleOneScore)
        {
            minMargin = middleOneMargin;
            middleOneMargin = middleTwoMargin;
            middleOneScore = middleTwoScore;
            middleTwoMargin = SensorConstants.decimalChange(
                    maxMargin-(1-r)*(maxMargin-minMargin), 3);
            middleTwoScore = evalFunction(description,
                    middleTwoMargin, scoreType);
        }
        if (middleTwoScore < middleOneScore)
        {
            maxMargin = middleTwoMargin;
```

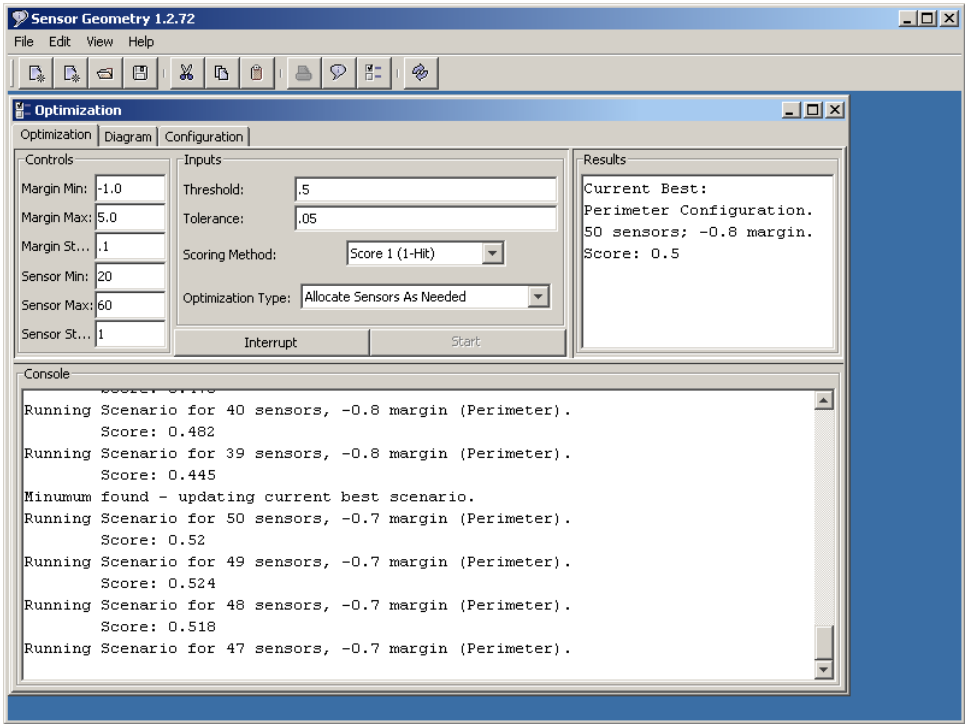
```
            middleTwoMargin = middleOneMargin;
            middleTwoScore = middleOneScore;
            middleOneMargin = SensorConstants.decimalChange(
                    minMargin+(1-r)*(maxMargin-minMargin), 3);
            middleOneScore = evalFunction(description,
                    middleOneMargin, scoreType);
        }
    }

    best[0] = middleOneMargin;
    best[1] = middleOneScore;

    Thread.yield();
    return best;
}
```

B Screenshots






Sensor Geometry 1.2.72
File Edit View Help
Scenario
Diagram Results Trials Past Runs Configuration
Scenario Description:
Source Region: 23.0, 20.0
Defended Region: 19.0, 16.0
Plume: Length=10.0 Arc=20.0 Area=17.632698070846498
Running for 500 cases.
36 sensor Perimeter setup. Spacing: 1.9213395013344163


Sensor Geometry 1.2.72
File Edit View Help
Optimization
Optimization Diagram Configuration
Controls
Margin Min: -1.0
Margin Max: 5.0
Margin St... .1
Sensor Min: 20
Sensor Max: 60
Sensor St... 1
Inputs
Threshold: .5
Tolerance: .05
Scoring Method: Score 1 (1-Hit)
Optimization Type: Allocate Sensors As Needed
Interrupt
Start
Results
Current Best:
Perimeter Configuration.
50 sensors; -0.8 margin.
Score: 0.5
Console
Running Scenario for 40 sensors, -0.8 margin (Perimeter).
Score: 0.482
Running Scenario for 39 sensors, -0.8 margin (Perimeter).
Score: 0.445
Minumum found - updating current best scenario.
Running Scenario for 50 sensors, -0.7 margin (Perimeter).
Score: 0.52
Running Scenario for 49 sensors, -0.7 margin (Perimeter).
Score: 0.524
Running Scenario for 48 sensors, -0.7 margin (Perimeter).
Score: 0.518
Running Scenario for 47 sensors, -0.7 margin (Perimeter).

C Program Default Values

This lists the default settings used by the program. These are loaded on startup, and in many cases were used in the data gathered for this report.

Parameter	Default Value	Parameter	Default Value
Defended Region Center	(0,0)	Sensor Grid Type	Perimeter
Defended Region Dimensions	19 x 16	Number of Sensors	36
Safe Region Center	(0,0)	Number of Tiers	1
Safe Region Dimensions	19 x 16	Margin	0
Plume Spawn Region Center	(0,0)	Center	False
Plume Spawn Region Dimensions	23 x 20	Number of Trials	500
Plume Major Axis Length	10	Random Seed	1
Plume Arc Width	20	Name	"Scenario"
Wind Distribution	Random		

There are also many default values for optimization purposes included in the program, which are shown below.

Parameter	Default Value	Parameter	Default Value
Margin Tolerance	.05	Margin Min	-2
Sensor Min	10	Margin Max	8
Sensor Max	60	Scoring Method	Score 1 (1-Hit)
Optimization Type	"Allocation"	Score Threshold	.5
Regularity	2		

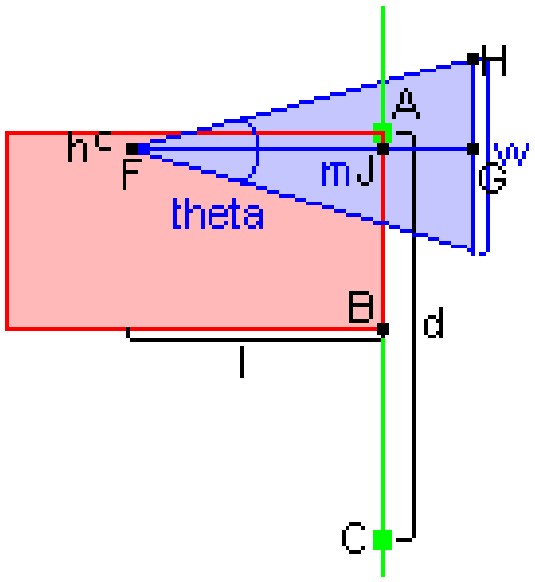


Figure 6: Wind Velocity Spacing Relationship Diagram

D Mathematical Proofs

The Sensor Geometry program is inspired by the Buffon's Needle problem, which calculates the odds that a needle dropped on a lined floor will land on a line. Proofs, often similar to the one used to prove the Buffon's Needle result, were created to develop mathematical conclusions regarding the program. In many cases, the mathematical proofs and the data agreed, which reinforced the validity of the application.

The first proof created found the relationship between the wind speed used to create the plume and the spacing needed to ensure a certain level of performance. The second proof expanded this idea, so that the plume could be more general in its placement, and supported the result of the original proof. The final proof examined the degenerate case where there is no wind, which leads the plume to expand in a circular contour.

D.1 Wind Velocity Spacing Relationship

When the Sernsor Geometry problem was initially postulated, one of the immediate hypotheses was that the needed spacing bewteen sensors would be related to wind speed. By examining the problem mathematically, it can be shown that this hypothesis is true for a reasonable general case, and that the needed distance between sensors is inversely proportional to the approximate square root of the wind speed.

Consider a regularly spaced Perimeter defense. Assuming a large enough base, we can neglect the effect of corners, and only consider sides. Consider the base Perimeter section AE. Let sensors be placed at points A, C, and E, where C is the midpoint of AE, and define points B and D to be the midpoints of AC and CE respectively.

In our proof, we need only consider the region between A and B, as it is identical (after any needed reflections) to the region BC, CD and DE.

Consider Figure 6:

$$\frac{d}{2} = AB$$

$$\frac{\theta}{2} = \angle HFG$$

$$m = GF$$

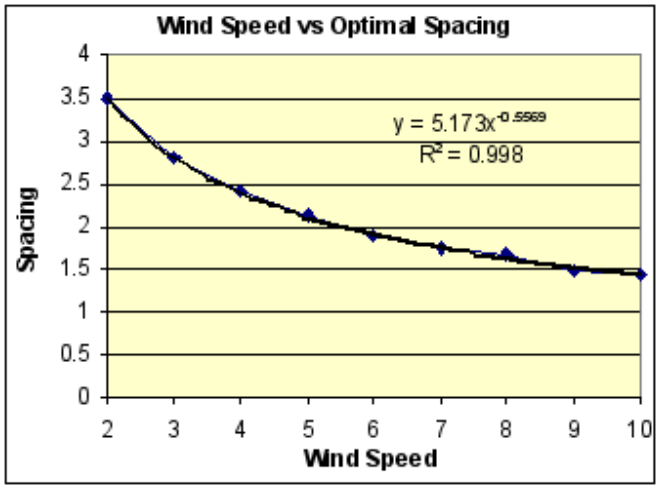


Figure 7: Wind Speed vs. Optimal Spacing

$$h = AJ$$
$$l = FJ$$
$$\frac{w}{2} = HG$$

Now, consider that Point F is placed randomly, such that $0 \leq l \leq m$ (such that the plume hits the defended region) and $0 \leq h \leq \frac{d}{2}$ (as this is the region we are considering).

Thus, the area of possible placement is $\int_0^m \int_0^{\frac{d}{2}} 1 \; dh \; dl = \frac{md}{2}$. Now, if A is going to be contained in $\triangle HFG$, then $0 \leq h \leq l \tan(\frac{\theta}{2})$, so the area where the plume is detected is $\int_0^m \int_0^{l \tan(\frac{\theta}{2})} 1 \, dh \, dl = \frac{m^2 \tan(\frac{\theta}{2})}{2}$.

$$P_{detect} = \frac{\frac{m^2 \tan(\frac{\theta}{2})}{2}}{\frac{md}{2}} = \frac{m \tan(\frac{\theta}{2})}{d} = \frac{w}{2d}$$

$\frac{w}{2} = av^b$, where a, b are constants and v is the wind speed. For most plume values, $b \approx -0.5$ [3].

$$P_detect = \frac{w}{2d} = \frac{av^b}{2d} \approx \frac{a}{d\sqrt{v}}$$

The work done in [1] (shown in Figure 7) agrees with this formula, although it suggests that $b \approx 0.55$ is a better estimate.

The above proof and corroborating data proves that the spacing needed for a specific detection probability is inversely proportional to the approximate square root of the wind speed. This proof, however, operates under 3 assumptions. The first is that corners of the defended area are negligent, which on a reasonably sized base, they are not. The second assumption is that the plume will point directly at the base. Finally, we assume that the plume's dimensions are such that it cannot intersect 2 sensors at once, and it is this assumption that is most worrisome. Nevertheless, given these conditions, the hypothesis regarding the relationship between wind speed and spacing has been proven. Further work will be needed in order to prove the same fact for the general case.

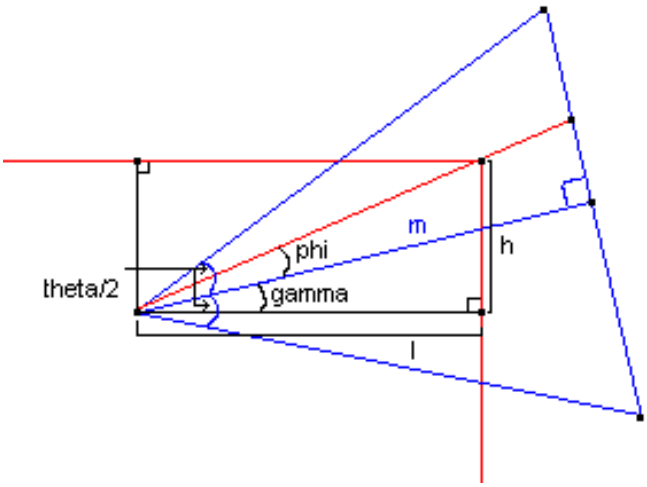


Figure 8: Wind Velocity Spacing Relationship (General Case) Diagram

D.2 Wind Velocity Spacing Relationship (General Case)

Consider Figure 8:

Now, consider that the plume spawn is placed randomly, such that $0 \leq l \leq m$ (such that the plume hits the defended region) and $0 \leq h \leq \frac{d}{2}$ (as this is the region we are considering). Also, $-\frac{\pi}{2} \leq \gamma \leq \frac{\pi}{2}$. Thus, the volume of possible placement (V_{pp}) is defined by the following equation:

$$V_{pp} = \int_0^m \int_0^{\frac{d}{2}} \int_{-\frac{\pi}{2}}^{\frac{\pi}{2}} 1 \; d\gamma \; dh \; dl$$

If the sensor is in the plume:

$$-\frac{\theta}{2} + \gamma \leq \tan^{-1}\left(\frac{h}{l}\right) \leq \frac{\theta}{2} + \gamma$$

Define $\phi = \tan^{-1}\left(\frac{h}{l}\right) - \gamma$ such that:

$$|\phi| \leq \frac{\theta}{2}$$

For the sensor to be inside the plume, the distance is also restricted:

$$\sqrt{h^2 + l^2} \leq \frac{m}{\cos(\theta)}$$

We can now use these restrictions to determine the volume of detected placements:

$$\begin{aligned} V_{detect} &= \int_{-\frac{\theta}{2}}^{\frac{\theta}{2}} \int_0^m \int_0^{\sqrt{\frac{m^2}{\cos^2(\phi)} - l^2}} 1 \; dh \; dl \; d\phi \\ &= \int_{-\frac{\theta}{2}}^{\frac{\theta}{2}} \int_0^m \sqrt{\frac{m^2}{\cos^2(\phi)} - l^2} \; dl \; d\phi \\ &= \int_{-\frac{\theta}{2}}^{\frac{\theta}{2}} 2m^2((\log(\cos(\phi)))(\frac{\pi}{2} - \phi)(\tan(\phi))) \; d\phi \end{aligned}$$

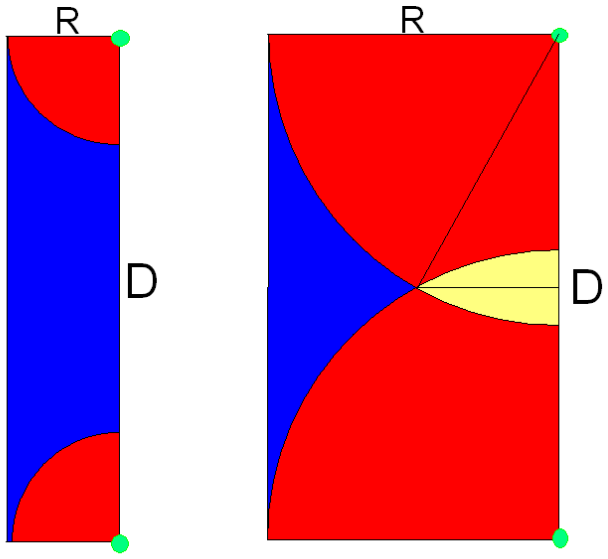


Figure 9: Circular Plume Analysis Diagram

$$= \frac{\pi m^2 \tan(\frac{\theta}{2})}{2}$$

Finally, the probability of detecting the sensor (P_{detect}) can be solved for.

$$P_{detect} = \frac{\frac{\pi m^2 \tan(\frac{\theta}{2})}{2}}{\frac{md\pi}{2}} = \frac{m \tan(\frac{\theta}{2})}{d}$$

This expression is the same as the previous proof, and the same algebra applies to prove that $d\sqrt{v}$ is constant.

D.3 Circular Plume Analysis

If a plume is released on a windless day, it will expand spherically, if terrain is not factored in. Considering just the 2D representation of this, it is clear the plume will be a circle. Examining once again an infinite base, consider Figure 9. In this diagram, D is the distance between sensors and R is the radius of the plume circle.

In both parts of Figure 9, the area that a plume can be dropped such that it will be detected is shaded red (or yellow, which is the area where it will be detected twice), and the area where it goes undetected is shaded blue.

Case 1 $D \geq 2R$

Here, the probability of the circle landing in the red is easy to compute:

$$P_{detect} = \frac{\frac{\pi R^2}{2}}{RD} = \frac{\pi R}{2D}$$

The ranging cases are:

$$D \to \infty \quad \Rightarrow \quad P_{detect} \to 0$$

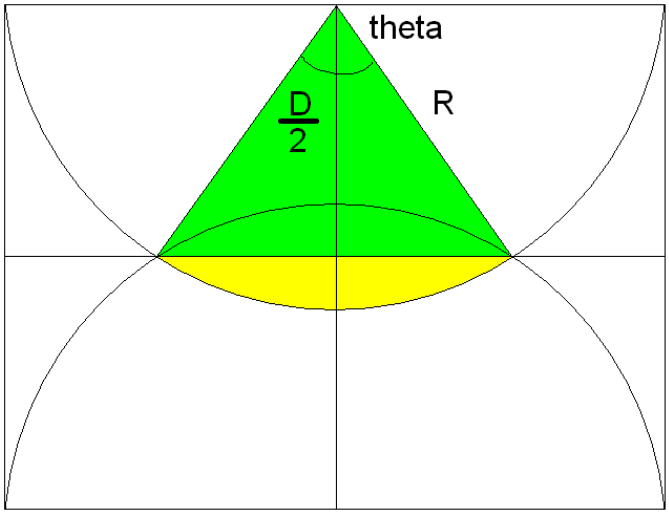


Figure 10: Circular Plume Analysis Diagram - Rearrangement

$$D \to 2R \quad \Rightarrow \quad P_{detect} \to \frac{\pi}{4}$$

Case 2 $D < 2R$
In this case, the probability is slightly more difficult to find.

$$P_{detect} = \frac{\frac{\pi R^2}{2} - A_{yellow}}{RD}$$

To solve for A_{yellow}, consider Figure 10. The area of the yellow region in this diagram is the same as the area of the yellow region in the original figure, and here,

$$\cos(\frac{\theta}{2}) = \frac{D}{2R}$$
$$\theta = 2\cos^{-1}(\frac{D}{2R})$$
$$A_yellow = \frac{R^2\theta}{2} - \frac{R^2\sin(\theta)}{2}$$
$$P_{detect} = \frac{\frac{\pi R^2}{2} - \frac{R^2}{2}(\theta - \sin(\theta))}{RD}$$

Inserting this into the previous equations,

$$P_{detect} = \frac{R}{2D}(\pi - \theta + \sin(\theta))$$
$$\theta = 2\cos^{-1}(\frac{D}{2R})$$

The ranging cases are:

$$D \to 2R \quad \Rightarrow \quad P_{detect} \to \frac{\pi}{4}$$
$$D \to 0 \quad \Rightarrow \quad P_{detect} \to 1$$

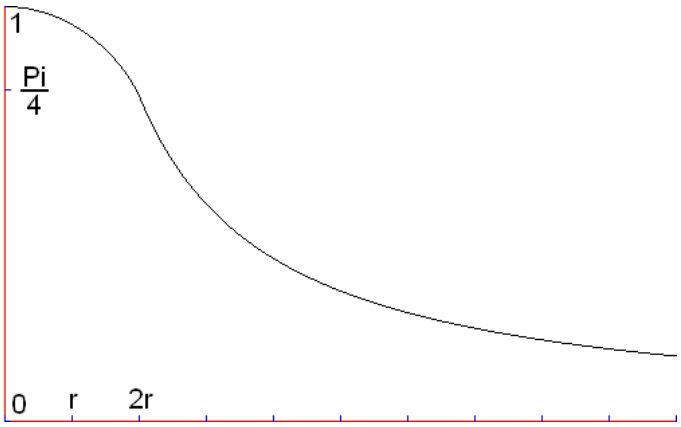


Figure 11: P_{detect} vs. spacing

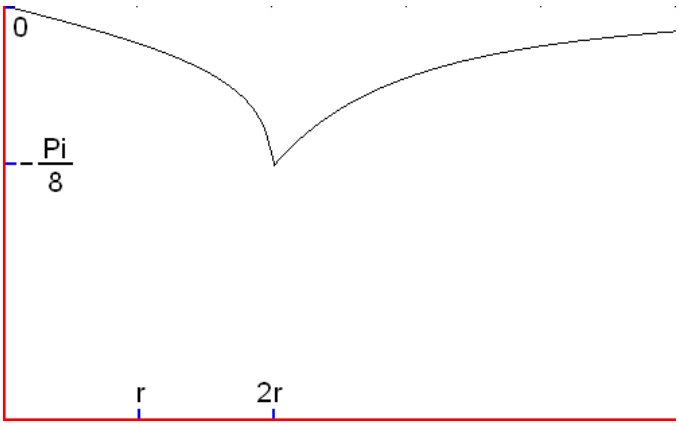


Figure 12: Derivative - P_{detect} vs. spacing

The ranging cases for bose Case 1 and Case 2 are reasonable, as the two agree at $D = 2R$. Also, as the spacing approaches 0, every plume is detected, and as the spacing becomes infinte, no plumes are detected.

The derivative of this equation is also continuous, although it does have a sharp point at. At that point, the derivative is at a minimum, implying that it is then that the loss or gain of one sensor would have the largest effect on the success rate of the array. The graph of P_{detect} vs. spacing is shown in Figure 11; the derivative of this graph is shown in Figure 12 (Graphs created using GCalc.net).

References

[1] A. Brown. Spacing optimization. Jul 2005.

[2] R. L. Burden and J. D. Faires. *Numerical Analysis Fifth Edition.* PWS Publishing Company, Boston, MA, fifth edition, 1993.

[3] T. DeLaPena. SEAP Report: Preliminary Performances of Sensor Geometries. Aug 2004.

[4] R.L. Graham and B.D. Luboachevsky. Repeated Patterns of Dense Packings of Equal Disks in a Square. *The Electronic Journal of Combinatorics*, 3, 1996.

[5] G. Gunn. Preliminary Investigation into Sensor Array Configurations. Feb 2005.

[6] G. Gunn, K. Gardner, J. Gerding, J. Han, and J. Hurd. A Monte Carlo Simulation for Analyzing the Performance of CB Sensor Arrays. In *73rd MORS Synopsium*, Jun 2005.

[7] G. Gunn, K. Gardner, J. Gerding, J. Han, J. Hurd, and G. Visco. Comparison of CB Sensor Array Configurations. In *73rd MORS Synopsium*, Jun 2005.

[8] G. Gunn, K. Gardner, J. Gerding, J. Han, G. Visco, and T. DeLaPena. Investigation into Performance Scoring for Arrays of CB Sensors. In *73rd MORS Synopsium*, Jun 2005.

[9] M. T. Heath. *Scientific Computing, An Introductory Survey, Second Edition.* McGraw-Hill, New York, NY, second edition, 2000.

[10] J. H. Matthews. Golden Ratio Search, 2004.

[11] W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery. *Numerical Recipes in C, The Art of Scientific Computing.* Cambridge University Press, Cambridge, second edition, 1999.

[12] W. L. Winston. *Operations Research Application and Algorithms Third Edition.* Duxbury Press, Belmont, third edition, 1993.