Grid Lab Write-Up

Declaration of Time Spent

This project was completed by Edward Ekstrom and Spencer Weight

Edward worked on the following items

- Implementing Visualization
- Implementing the Bayes Rule
- Implementing the Grid and Grid Filter
- Debugging Tank Routing Code to Work Under Time Restrictions
- Running our Agent and Taking All of the Screenshots Used for this Write-up
- Wrote the text for the section titled "Discussion of Varying Parameters"

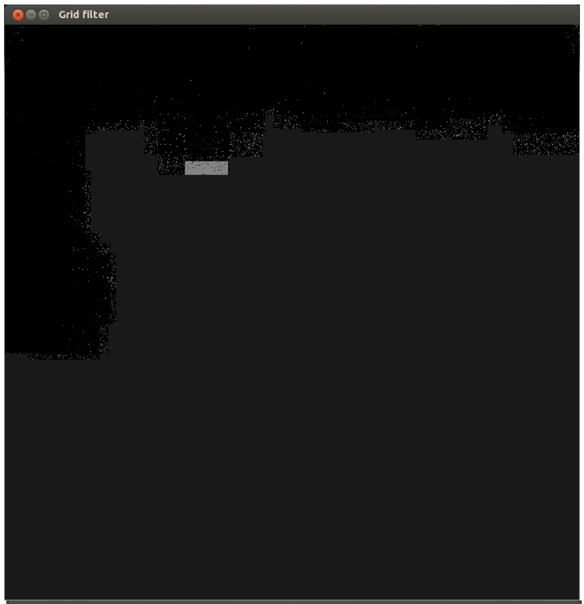
Edward spent 15 hours on the project.

Spencer worked on the following items

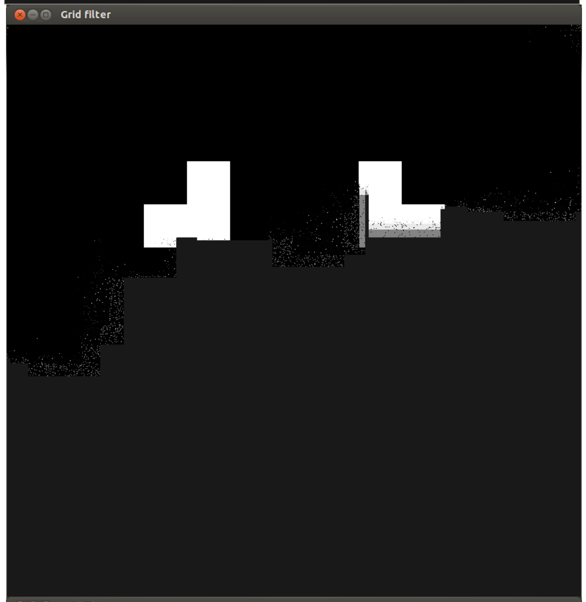
- Programmed the code that would move a tank along a specific path for mapping
- Helped with the debugging of some of the multi-tank pathing code.
- Wrote the text for the sections titled "Explanation of Visualizations", "How the tanks search the world", and "How the grid filter could be used against an opponent"

Spencer spent 10 hours on the project.

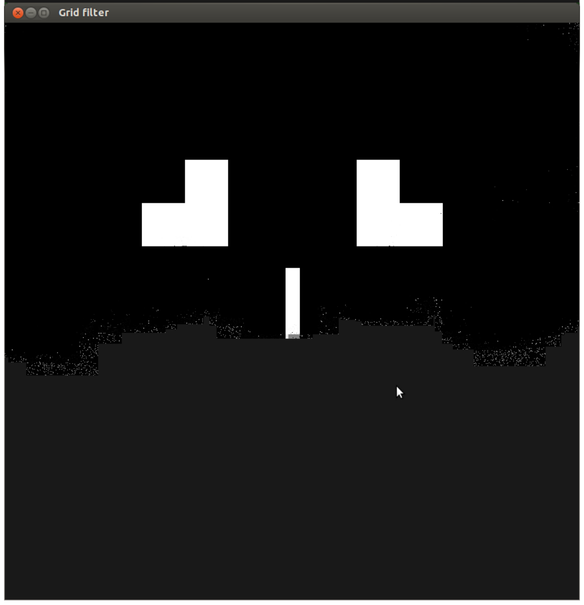
Explanation of Visualiztions



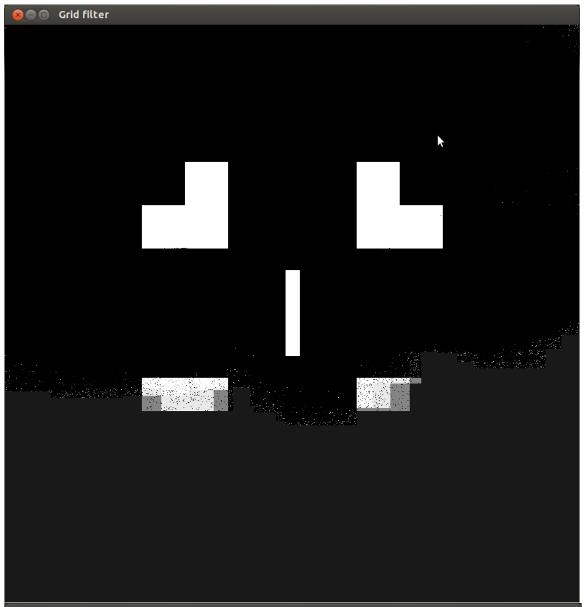
Stage 1 - This map is the “Four L’s” map. In this image, we had set the number of tanks per team to four and assigned the agent to control the red team’s tanks. The darkest black represents the absence of any obstacles, so if you look really close you can see that there is a black cloud that moves up from the left side to the top left corner and then to the top right corner. This black cloud is the first leg of the journey for the tanks on their exploration of the map. The light grey block that is visible is the top of one of the four L’s. The tanks did a quick pass and their sensors picked up that something might be there, but they are not completely sure yet.



Stage 2 - The path of the tanks is a path that snakes back and forth from the left to the right side of the map. After the first row of the path, the tanks turn around, move down a row, and return to the left side where the go down another row. In this image the tanks have completed the first 4 rows. As you can see, tanks have passed by the obstacles enough times and gathered enough sensor data to be sure that the obstacles are present. At least for the white parts.

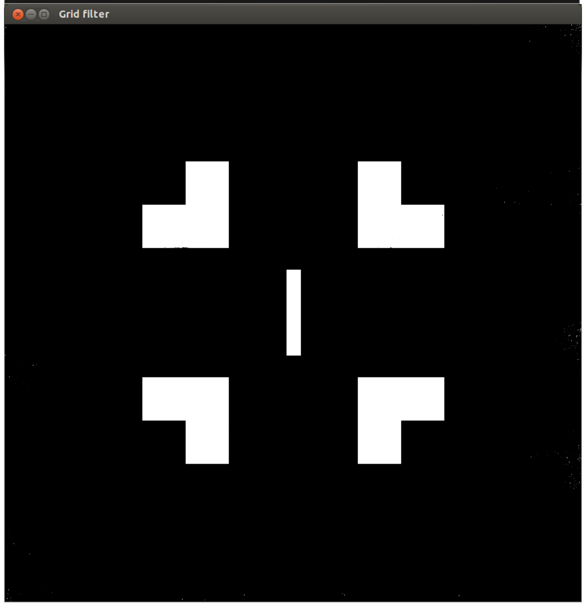


Stage 3 - At this point the 4 tanks have made enough passes to cover the top half of the map and they are sure that there are obstacles wherever there is a white shape. They have been gathering information with their sensors and plugging it into the Bayes rule to estimate the probability that an obstacle is present. The Bayes rule is being used to calculate $P(s_i,j = occupied \mid o_{i,j})$ for each pixel/cell of the world grid from the input of the sensor. White means that this value is close to 1.0 and black means this value is close to 0.0. Grey means somewhere in between.



Stage 4 - The tanks have continued further and are beginning to discover new objects through sensor use. The code that has been implemented to run the tanks' sensors is gathering these parts of the Bayes rule equation:

- *$P(o_{i,j} \mid s_{i,j} = occupied)$* -> this is the value of the truePos or trueNeg values passed in from the command line when running the program
- *$P(s_{i,j} = occupied)$* -> this value comes from what the value of the world grid cell was before taking into account the sensor data
- *$P(o_{i,j})$* -> this is the value that is calculated from solving for it in this equation:
    - *$P(s|o) = P(s,o)/P(o)$*



Stage 5 - The tanks have completed the map. The tanks' sensors were set to a width of 100 so they could see far enough into the obstacles to know that the obstacles are solid all the way through after enough readings. The tanks' sensors have passed over the map enough times to have a near perfect representation of the map. The world grid has been updated enough such that in each occupied cell the probability is high enough for the tank to recognize that there is something there.

Here is the block of code that calculates the probability that a cell in the world grid is occupied:
****

```
def updateGrid(self,pos,miniGrid):
    x0,y0 = pos
    x0+=self.dimensions[0]/2
    y0+=self.dimensions[1]/2
    x = x0
    y = y0
    for col in miniGrid:
        for row in col:
            prior = self.grid[x][y]
            if row == 1:
                occupied = self.truePos * prior
                unoccupied = (1 - self.trueNeg) * (1 - prior)
            else:
                occupied = (1 - self.truePos) * prior
                unoccupied = self.trueNeg * (1 - prior)

            self.grid[x][y] = occupied / (occupied + unoccupied)
            y+=1
        y=y0
        x+=1
```
****

The part of the code that is in bold is the part where the value for an individual cell is calculated. The elements of the Bayes rule can be picked out:
P(s = occupied) == prior == self.grid[x][y] // (before update)
P(o | s = occupied) == self.truePos
P(o | s = unoccupied) == self.trueNeg
P(o) == (occupied + unoccupied)
P(s = occupied | o) == occupied / (occupied + unoccupied) == self.grid[x][y] // (after update)

How the tanks search the world

Our first iteration of the pathing algorithm had just one tank doing all the exploring and later we implemented the algorithm to accommodate 4 tanks which is the limit that our code could handle successfully. The tanks are guided by attractive potential fields that are placed like bread crumbs. Each potential field is centered on a “cell” of an imaginary 8 by 8 grid on the world map. Once the tank reaches a potential field, the field shifts to the next cell in the direction that the tank is currently traveling. The tanks begin in the top left corner and then then proceed left to the top right corner. When the tanks reach the end of a row they move down to the next row and begin travelling towards the left. The unobstructed pattern that the tanks make is like this:

```
--------->
         |
<---------
|
--------->
```
And so on and so forth...

When the tanks following this path encounter an obstacle such that they are jammed up against it, the algorithm checks to see if the tanks are stuck by checking if they have changed positions since the last tick. If the tanks haven't changed position for 40 ticks then the algorithm shifts the location of the current potential field that the tanks are attracted to. The field is shifted up a row if the tanks are in the top half of the map, and the field is shifted down a row if the tanks are in the bottom half of the map. Shifting the field aids the tanks in circumventing obstacles by changing their vector of travel. Once the tanks are clear of the obstacle, they resume their normal path by moving towards the expected potential field.

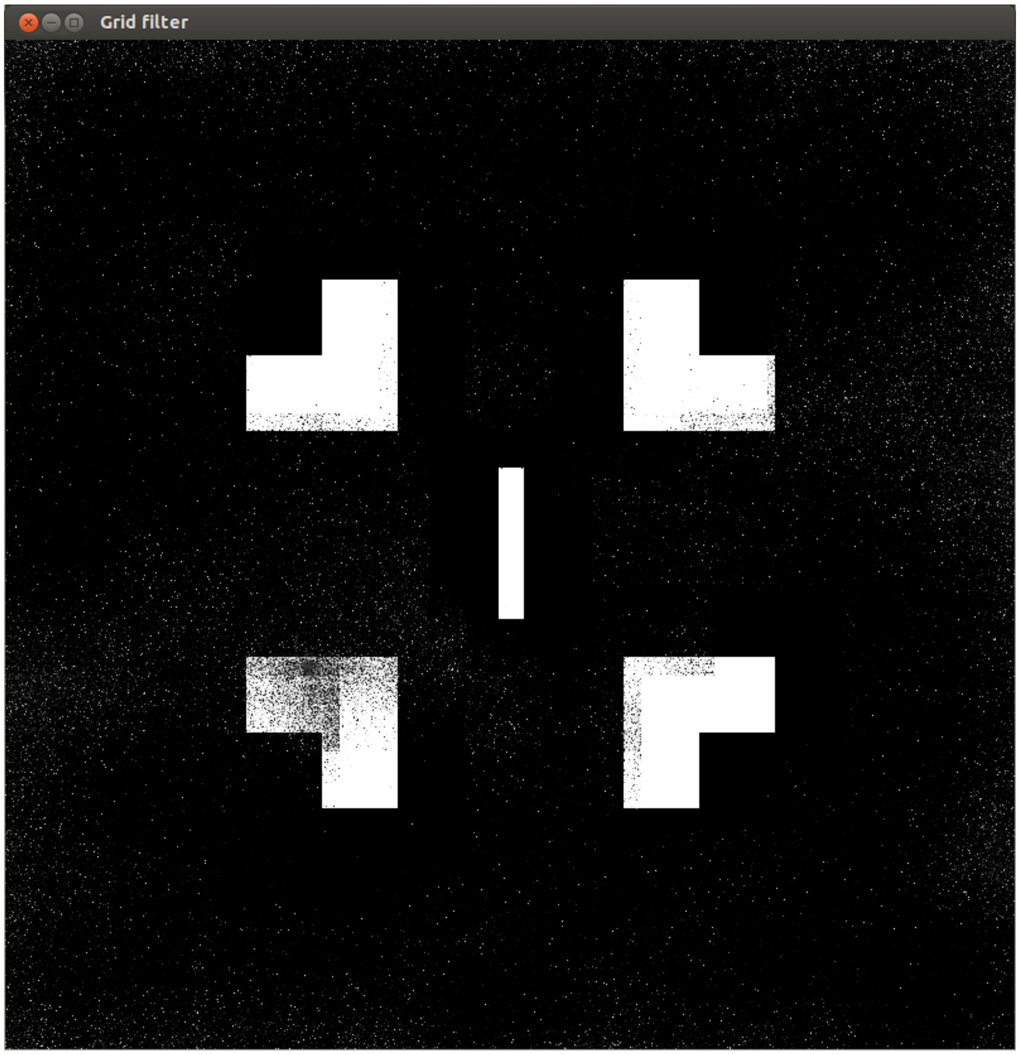
How the grid filter could be used against an opponent

The grid filter is useful against opponents in the sense that when the agent knows where all the obstacles are, then the agent can use that information to avoid the obstacles, and thus avoid running into them and getting stuck in a spot where the enemy could shoot them. If an opponent doesn't know the layout of the map, then they will be hindered by the various objects that get in the way. Thus becoming stationary, easy targets for tanks that use formulas to predict the movement of other tanks.

Discussion of varying parameters

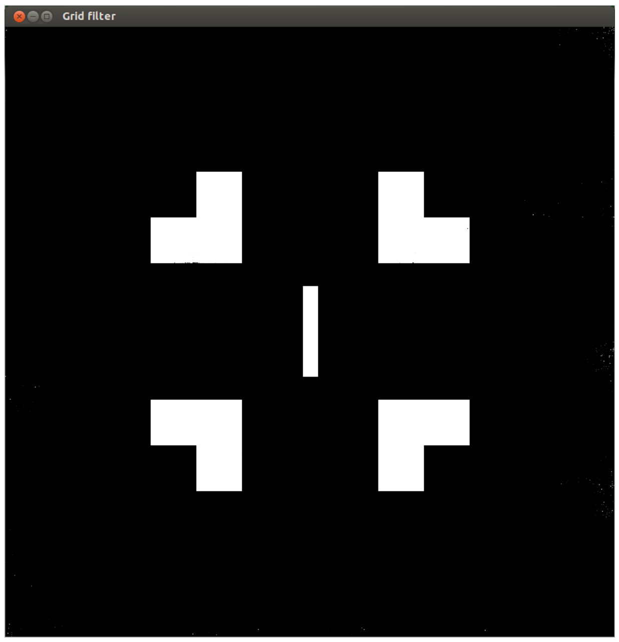
Noisier Readings:

We thought it would be an interesting experiment to see what would happen if we increased the likelihood of false-positives and false-negatives. For this experiment we ran the server with a true-positive value of .77 and a true-negative value of .70. These values are much lower than the values we had previously been testing with and so we hypothesized that our map would be much less clean than our previously drawn maps. Here is a visualization of our results:

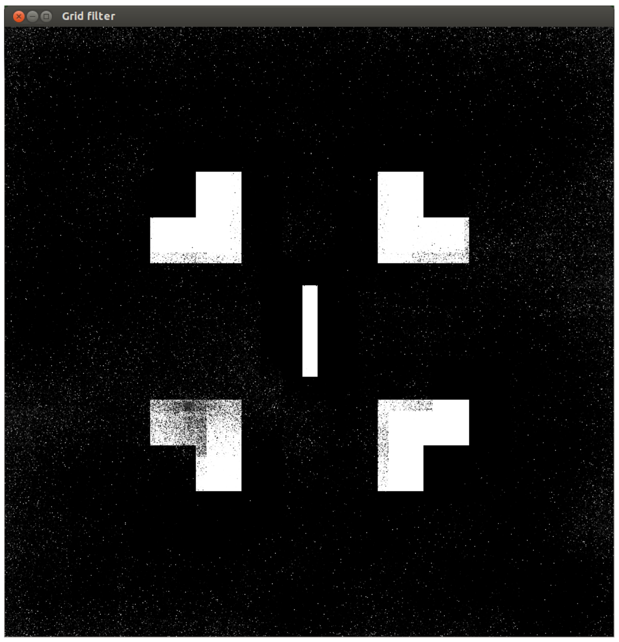


true-pos=.77, true-neg=.7

As you can see, the mapping has much more noise than our previous mappings. Here is a side by side so you can see the difference more clearly:



true-pos=.97, true-neg=.9



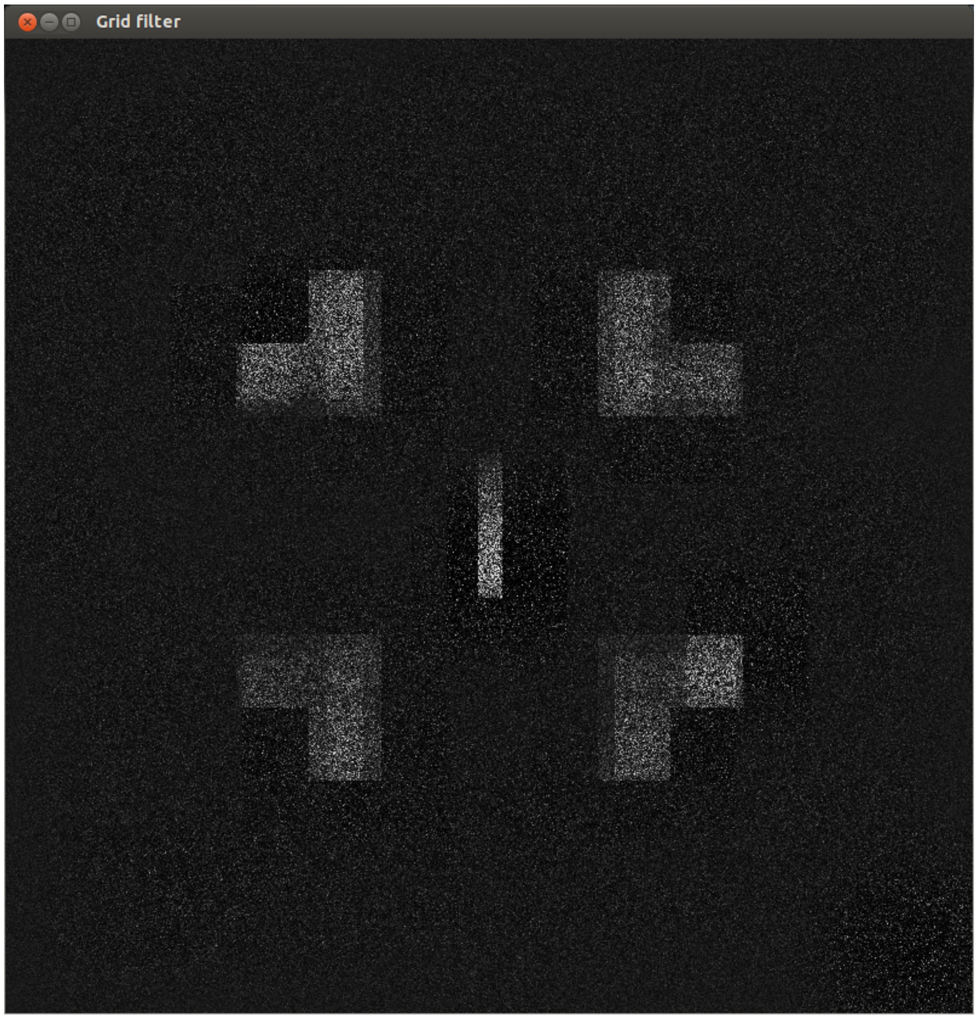
true-pos=.77, true-neg=.7

Our hypothesis was correct even though we thought it would be much noisier than it ended up being. It ended up being almost as clear as our mappings drawn with the higher true-positive and true-negative values.

It did an excellent job of finding the obstacles because the tanks spend a lot of time next to obstacles getting readings. This is because they get stuck on them and it takes a few seconds for our code that gets them around the obstacle to kick in. During this time they are able to get a lot of readings of the obstacle and so we are able to remove the effects of the noisy readings. The tanks were able to make it around the bottom left “L” with no problem and so there was significant noise on our drawing of that obstacle due to less readings.

There was a also relatively more noise in the open space of the map. Though, even with the noise in the open space it is easy to see that there is no obstacle there since the noise is just random dots every once and a while. It would be simple to write an algorithm that detects false-positive readings in the open space based on the surrounding area.

Sense our agent did so well mapping with true-positive = .77 and true-negative = .70, we decided to make it even more difficult. For this experiment we set true-positive to .57 and true-negative to .50. We hypothesized that you would not even be able to tell where obstacles were due to amount of false-positives and false-negatives. Here is a drawing of our results:



true-pos=.57, true-neg=.5

Surprisingly, our agent was still able to find the obstacles and our hypothesis was wrong. Looking at the image, it is easy to see where the obstacles are. Although it is still difficult to see some parts of the obstacles (ex. the top left side of the bottom two “L’s” and the top of the middle obstacle). Those parts of those three obstacles almost look the same as the rest of the map, only with a slightly lighter grey shading.

The area with no obstacles looks perfectly grey since it is wrong half the time. This ends up not mattering though since all we care about is the contrast between where there is an obstacle and where there is not an obstacle.

We conclude that it would be possible to use our agent with a true-positive value of .57 as long as the user is cautious to avoid obstacles with a large buffer just in case our agent failed to see the edges when making its drawing.

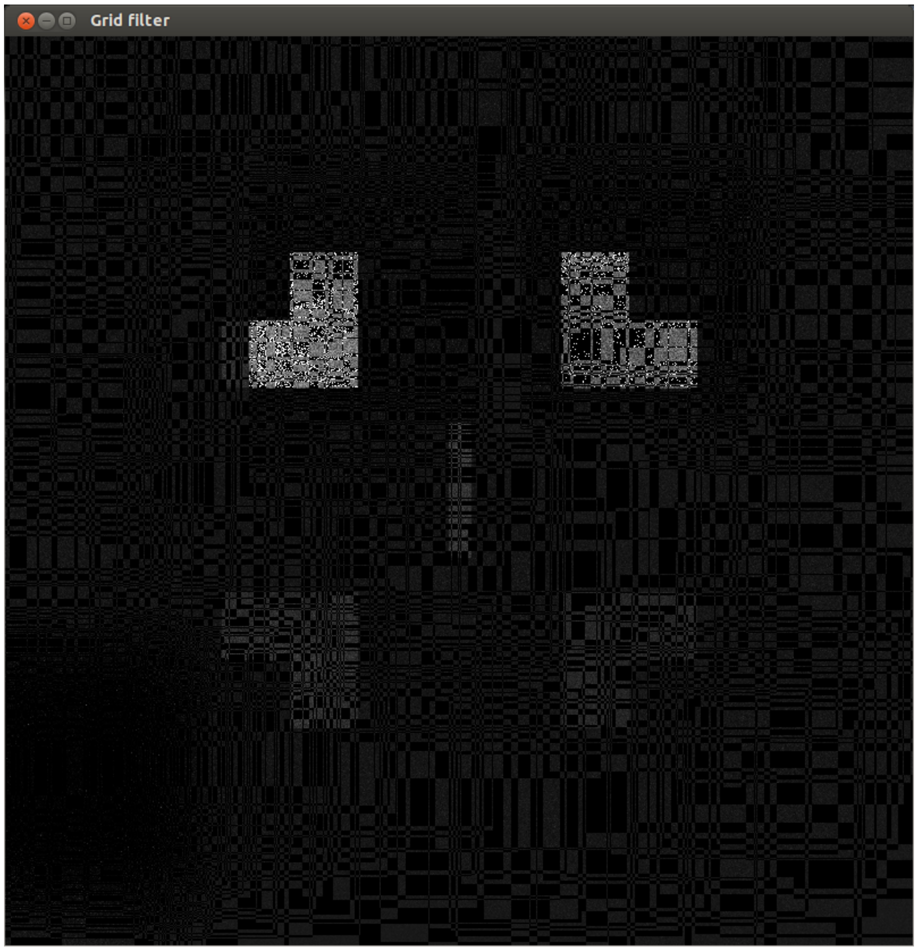
Larger Range:

Our next experiment was setting the range of our readings to 200 instead of 100. For this experiment we made it so our tanks would snake down the map at greater intervals since their reading radius was larger. We did not know if we would be able to control our tanks with the larger range since it would take longer to read in the get_occgrid() results and update our grid. We hypothesized that with this extra time requirement that we would lose control of our tanks and would have to use less of them.

Our hypothesis ended up being semi-correct. We did lose a little bit of control of our tanks, but we still had enough control to get them to their waypoints in a timely manner. We ended up being able to map the board approximately 2x faster with the larger range of our readings.

Incorrect Model of True-Positive and True-Negative:

Our final experiment was to give our agent incorrect values for true-positive and true-negative. Our hypothesis was that it would not have a great effect on our mapping as long as the value we gave it was within .2 of the actual value. To test this we set the servers true-positive to .67 and true-negative to .6 then tested our agent once telling it that true-positive was .87 and true negative was .8 and once telling it true-positive was .97 and true-negative was .9. Here are the results for incorrect true-positive=.87 and incorrect true-negative=.8 (+.2 of actual values):



We cannot explain why the mapping turned out so strange but it is still pretty easy to tell where the obstacles are, which is the important part. We do not even have a guess as to why the mapping has all of those grey squares where it should be black and all of the black squares on the obstacles where it should be white. The results for incorrect true-positive=.97 and incorrect true-negative=.9 (+.3 of actual values) were similar.

Summary of what was learned

Edward Ekstrom:

I learned the power of the Bayes rule in this lab. I also learned how to program under real-time restrictions. I had never had a program that was so greatly affected by an outside force (namely the server taking forever to return the results of get_occgrid). This made me think of ways I could update the grid less frequently and take readings less frequently so I would not lose control of my tanks. I was forced to come up with some pretty clever fixes in order to run fast enough to update the grid and not lose control of my tanks.

Spencer Weight:

From this project, I have gained a greater appreciation for the Bayes rule as a way to more accurately estimate the probability of an occurrence of an event. In a previous class I have taken, I had learned of the Bayes rule as a way to classify documents as part of the indexing process for a search engine. This lab has taught me that the Bayes rule is useful as a way of not only classifying documents, but is also useful as a way of classifying whether a cell on the world map can be classified as occupied or unoccupied. I also learned from this lab that in situations where it is important for the agent to be aware of its surroundings, knowing where obstacles are goes a long way towards helping the agent navigate around them.