جامعة الأخوين

AL AKHAWAYN
UNIVERSITY

Ayoub Mabkhout

Teammate: Chifaa Bouzid

Instructor: Dr. Tajjeeddine Rachidi

April 20, 2022

CSC 4301: Intro. to Artificial Intelligence Project #3

Bust the Ghost: A Probabilistic Minesweeper-like Game

Table of Contents

Introduction

This is a report detailing our implementation for 3rd A.I. mini project titled '*Bust the Ghost*'. *Bust the Ghost* is a Minesweeper-like game where the player has a censor that, once they select a tile, gives that tile a certain color as a function of the tile's distance from the ghost. Green means that the ghost is far away from that cell and Red means that the ghost is in that cell. The censor is, however, noisy and does not always give accurate measurements, creating the need for a methodology of probabilistic reasoning in order to infer the likely position of the Ghost from the color of cells in the grid.

The aim of this project is to build the game along with a Bayesian probabilistic inference algorithm in order to display the probability of the Ghost being in each cell and determine the cell that is most likely to contain the Ghost.

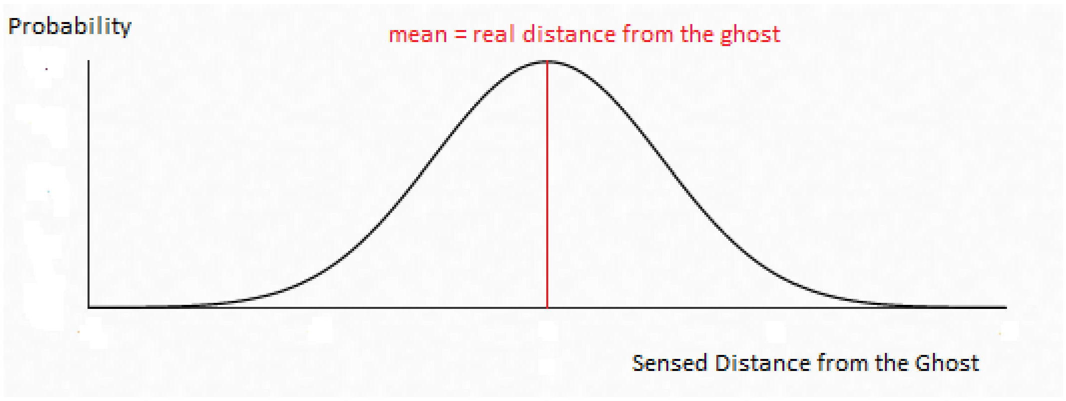
A demo video for the program can be found in the following link: https://youtu.be/ZSmXIHrsoPY

Program Structure

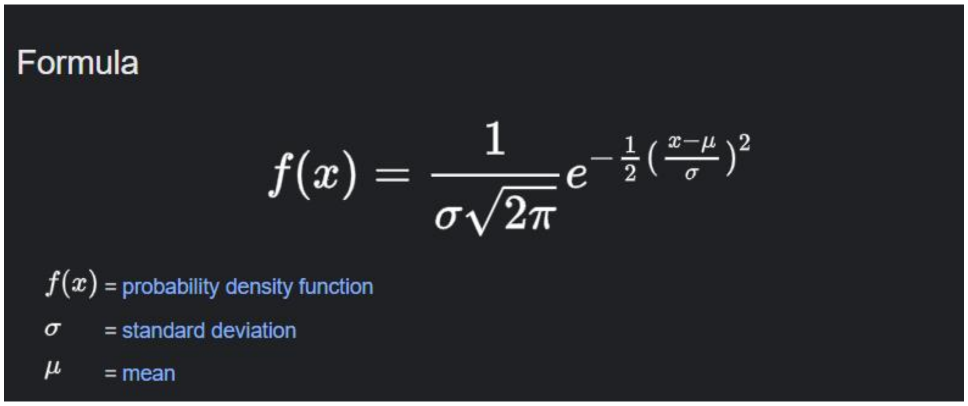
The program contains many C# scripts, but the two main ones are 'GridManager.cs' and 'Proba.cs'. The former contains the logic and flow of the game, including the algorithms for updating the probabilities and displaying them, the handling of the user input, and more. The latter is a probability library that we wrote in order to compute the probabilities and that is used in 'GridManager.cs'.

How to Simulate a Noisy Censor

In order to simulate a noisy censor, we opted for a normal distribution approach. We first set a value for a standard deviation. The censor then gets a the real distance from the clicked cell to the ghost cell and uses that to generate a 'sensed' distance close to that of the real distance plus or minus some uncertainty based on the value of the standard deviation.



The above graph shows the normal function for the probability distribution of distances likely to be sensed given the real distance as the mean and given the standard deviation that we have defined. The normal distribution function is defined as follow:



Formula

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$$

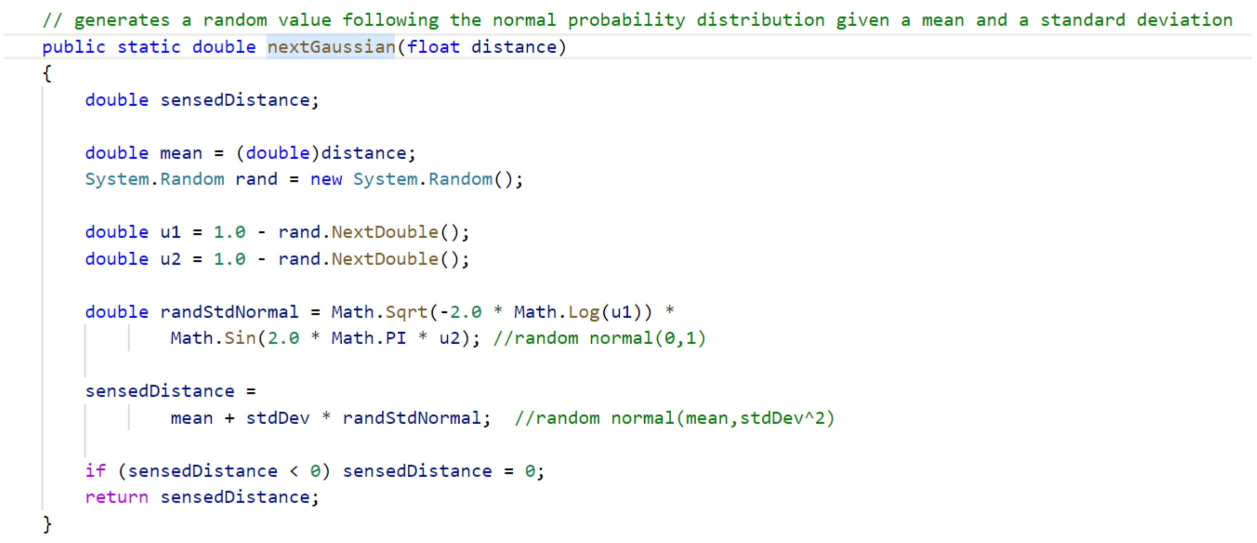
$f(x)$ = probability density function

σ = standard deviation

μ = mean

The standard deviation is the parameter that controls the amount of variance that we will have if we sample values from our normal distribution. The following link explains in more details how the standard deviation is thoroughly defined: https://en.wikipedia.org/wiki/68%E2%80%9395%E2%80%9399.7_rule

All that is left is to write a function that would take in the real distance (computed as a Manhattan distance) and generate a normally distributed distance. That distance in then rounded to an integer and then translated to the corresponding color. When clicking a tile for the first time, the distance between that tile and the ghost tile is given to our ‘NextGaussian()’ function as an argument and a color is displayed on that tile. The ‘NextGaussian()’ function is defined as follows:



```
// generates a random value following the normal probability distribution given a mean and a standard deviation
public static double nextGaussian(float distance)
{
    double sensedDistance;

    double mean = (double)distance;
    System.Random rand = new System.Random();

    double u1 = 1.0 - rand.NextDouble();
    double u2 = 1.0 - rand.NextDouble();

    double randStdNormal = Math.Sqrt(-2.0 * Math.Log(u1)) *
            Math.Sin(2.0 * Math.PI * u2); //random normal(0,1)

    sensedDistance =
            mean + stdDev * randStdNormal;  //random normal(mean,stdDev^2)

    if (sensedDistance < 0) sensedDistance = 0;
    return sensedDistance;
}
```

source: https://stackoverflow.com/questions/218060/random-gaussian-variables

This part is the only part that makes use of the real distance between the tile and the Ghost. All probabilistic inferencing for that tile and the tiles around it is based on estimates of the distances given the color of the tile that has been clicked on.

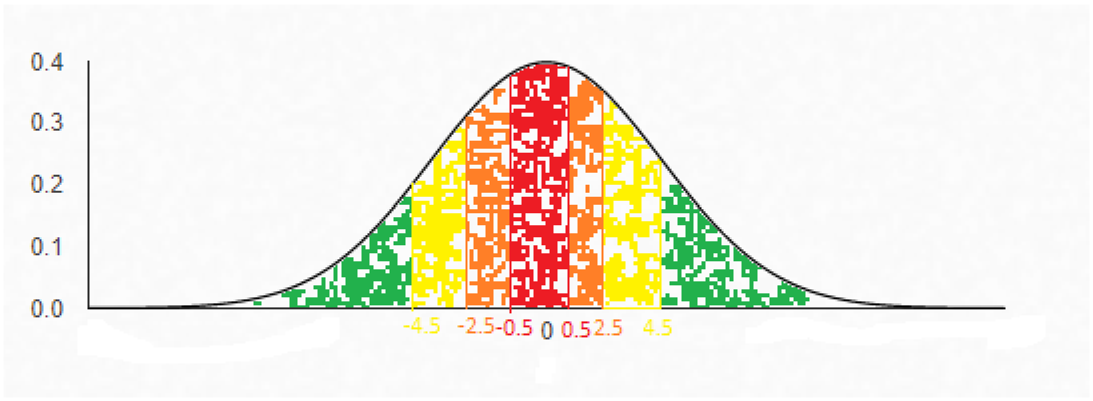
How do Determine P(Color/Distance from Ghost).

The way that we update the probability for a cell to contain the ghost is done in three steps. The first one is to compute the probability of the ghost being a certain distance away from the cell, or the probability of getting a certain color for the cell, given the estimated distance between that cell and the ghost cell. The second step is to multiply that by the prior probability for that cell containing, corresponds to **updating** the probability in the Bayesian inferencing. Once that is done for all cells, the last step is to normalize the probabilities so that they add up to one.

Note that, in the first step, distance and color are practically equivalent, as a color denotes a range of floating-point distances. For instance, the color orange corresponds to a distance of 1 to 2 cells away from the Ghost cell; that gives us a range of 0.51 to 2.49 from the ghost, since round any value in that range will give us either 1 or 2.

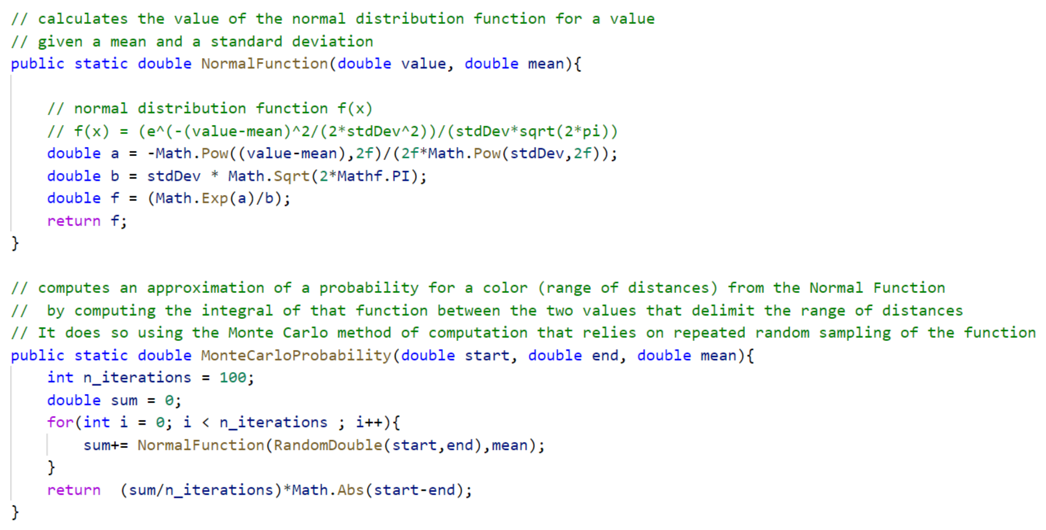
Now that we have established that the color of a cell is equivalent to a range of distances from the colored cell to the ghost cell, we can describe a methodology to get the value for probability of getting a color given an estimated distance from the Ghost. The method for estimating the distance of from the ghost will be discussed in a later part on the report.

That method would be to compute the integral of the normal function over the range that corresponds to that color. This gives us the probability of a value being in that range.



This is a graph that shows the probability distribution of colors in our normal distribution function.

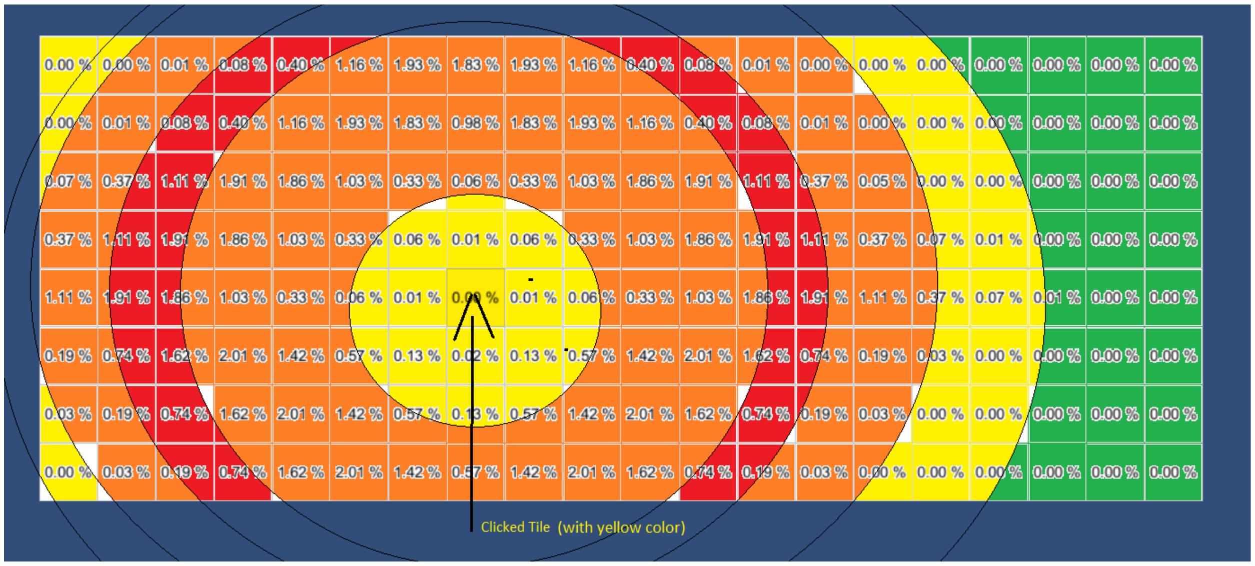
Since there is no readily available anti-derivative formula for the normal distribution function, we employed the Monte Carlo method of computing integrals, which is based on an iterative random sampling of the normal function within the specified range in order to get the average value in that range, which will then be multiplied by the range itself and will give us an accurate estimate of the area under the curve in the specified range.



```
// calculates the value of the normal distribution function for a value
// given a mean and a standard deviation
public static double NormalFunction(double value, double mean){

    // normal distribution function f(x)
    // f(x) = (e^(-(value-mean)^2/(2*stdDev^2))/(stdDev*sqrt(2*pi))
    double a = -Math.Pow((value-mean),2f)/(2f*Math.Pow(stdDev,2f));
    double b = stdDev * Math.Sqrt(2*Mathf.PI);
    double f = (Math.Exp(a)/b);
    return f;
}

// computes an approximation of a probability for a color (range of distances) from the Normal Function
//  by computing the integral of that function between the two values that delimit the range of distances
// It does so using the Monte Carlo method of computation that relies on repeated random sampling of the function
public static double MonteCarloProbability(double start, double end, double mean){
    int n_iterations = 100;
    double sum = 0;
    for(int i = 0; i < n_iterations ; i++){
        sum+= NormalFunction(RandomDouble(start,end),mean);
    }
    return  (sum/n_iterations)*Math.Abs(start-end);
}
```

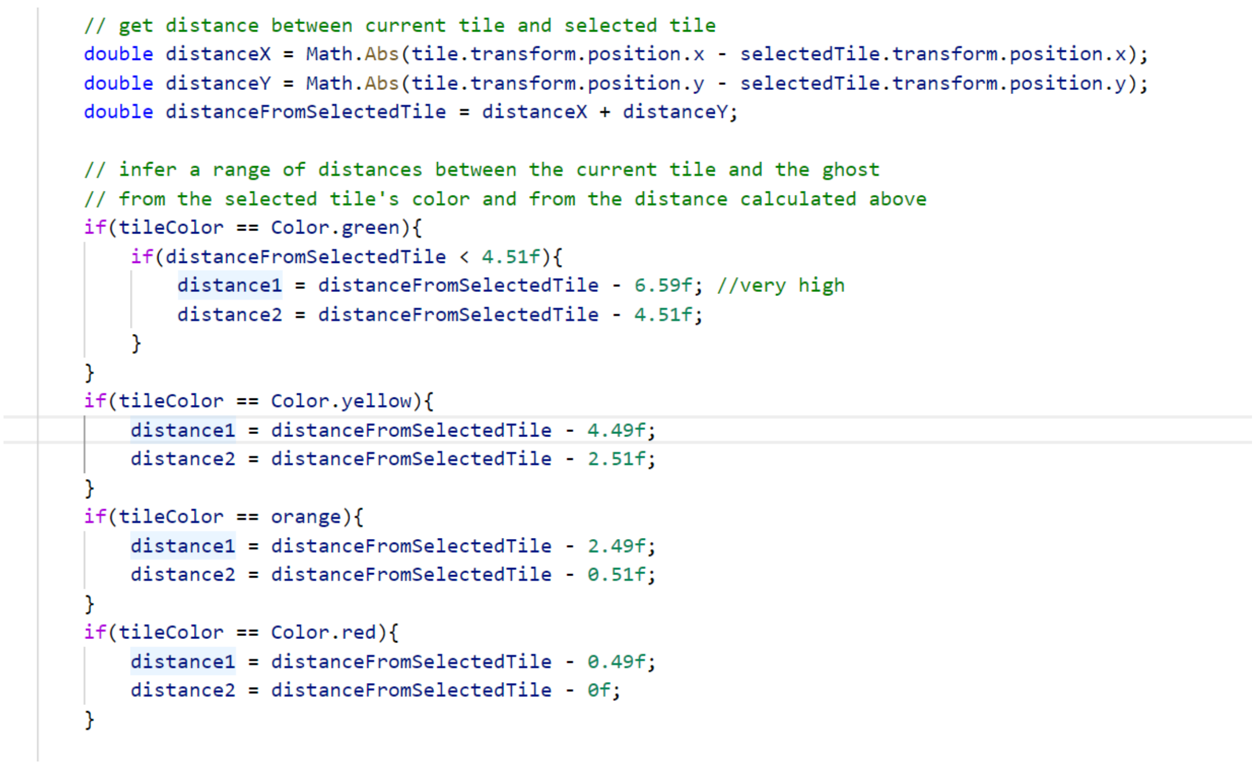
Updating the Probabilities of All Tiles

Once a tile is clicked and a color is generated for that tile, we can infer a probability of the ghost's distance from that cell. Here is one way to visualize it:



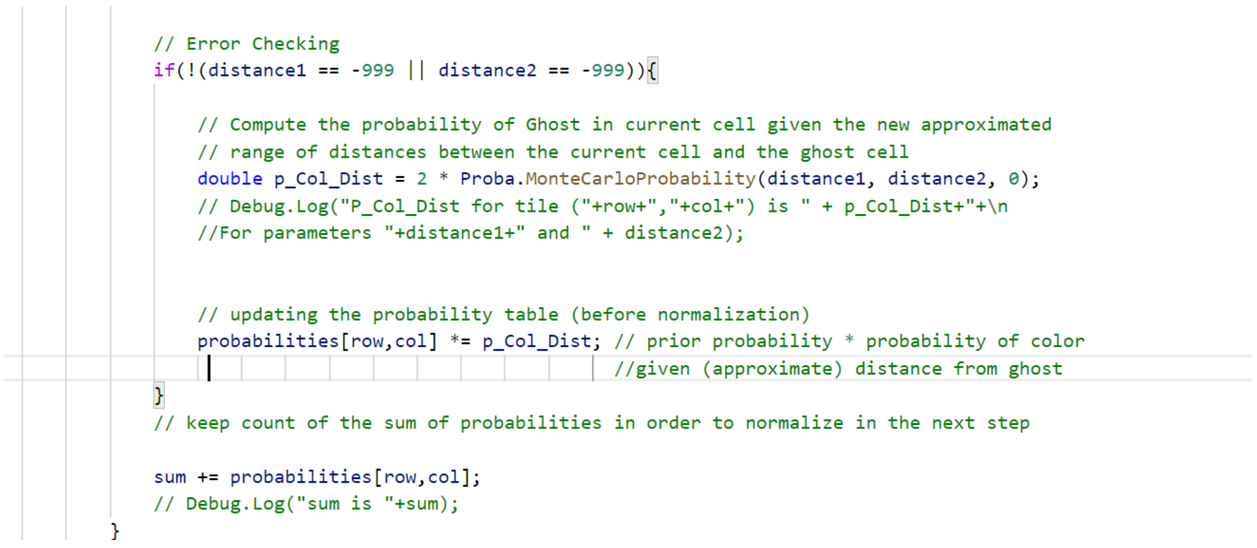
0.00 %	0.00 %	0.01 %	0.08 %	0.40 %	1.16 %	1.93 %	1.83 %	1.93 %	1.16 %	0.40 %	0.08 %	0.01 %	0.00 %	0.00 %	0.00 %	0.00 %	0.00 %	0.00 %	0.00 %
0.00 %	0.01 %	0.08 %	0.40 %	1.16 %	1.93 %	1.83 %	0.98 %	1.83 %	1.93 %	1.16 %	0.40 %	0.08 %	0.01 %	0.00 %	0.00 %	0.00 %	0.00 %	0.00 %	0.00 %
0.07 %	0.37 %	1.11 %	1.91 %	1.86 %	1.03 %	0.33 %	0.06 %	0.33 %	1.03 %	1.86 %	1.91 %	1.11 %	0.37 %	0.05 %	0.00 %	0.00 %	0.00 %	0.00 %	0.00 %
0.37 %	1.11 %	1.91 %	1.86 %	1.03 %	0.33 %	0.06 %	0.01 %	0.06 %	0.33 %	1.03 %	1.86 %	1.91 %	1.11 %	0.37 %	0.07 %	0.01 %	0.00 %	0.00 %	0.00 %
1.11 %	1.91 %	1.86 %	1.03 %	0.33 %	0.06 %	0.01 %	0.00 %	0.01 %	0.06 %	0.33 %	1.03 %	1.86 %	1.91 %	1.11 %	0.37 %	0.07 %	0.01 %	0.00 %	0.00 %
0.19 %	0.74 %	1.62 %	2.01 %	1.42 %	0.57 %	0.13 %	0.02 %	0.13 %	0.57 %	1.42 %	2.01 %	1.62 %	0.74 %	0.19 %	0.03 %	0.00 %	0.00 %	0.00 %	0.00 %
0.03 %	0.19 %	0.74 %	1.62 %	2.01 %	1.42 %	0.57 %	0.13 %	0.57 %	1.42 %	2.01 %	1.62 %	0.74 %	0.19 %	0.03 %	0.00 %	0.00 %	0.00 %	0.00 %	0.00 %
0.00 %	0.03 %	0.19 %	0.74 %	1.62 %	2.01 %	1.42 %	0.57 %	1.42 %	2.01 %	1.62 %	0.74 %	0.19 %	0.03 %	0.00 %	0.00 %	0.00 %	0.00 %	0.00 %	0.00 %

Here, the clicked tile showed a color of yellow, the red ring shows the distance at which we are most likely to find the ghost given that one piece of information. To compute the probability of the other tiles containing the ghost, the estimated distance that can used is the distance between each tile and the ring shown in red. That would be the difference in distances between each tile and the selected tile and the distance separating the selected tile from the ghost. Inside the loop iterating though each tile, we run the following logic:



```
// get distance between current tile and selected tile
double distanceX = Math.Abs(tile.transform.position.x - selectedTile.transform.position.x);
double distanceY = Math.Abs(tile.transform.position.y - selectedTile.transform.position.y);
double distanceFromSelectedTile = distanceX + distanceY;

// infer a range of distances between the current tile and the ghost
// from the selected tile's color and from the distance calculated above
if(tileColor == Color.green){
    if(distanceFromSelectedTile < 4.51f){
        distance1 = distanceFromSelectedTile - 6.59f; //very high
        distance2 = distanceFromSelectedTile - 4.51f;
    }
}
if(tileColor == Color.yellow){
    distance1 = distanceFromSelectedTile - 4.49f;
    distance2 = distanceFromSelectedTile - 2.51f;
}
if(tileColor == orange){
    distance1 = distanceFromSelectedTile - 2.49f;
    distance2 = distanceFromSelectedTile - 0.51f;
}
if(tileColor == Color.red){
    distance1 = distanceFromSelectedTile - 0.49f;
    distance2 = distanceFromSelectedTile - 0f;
}
```

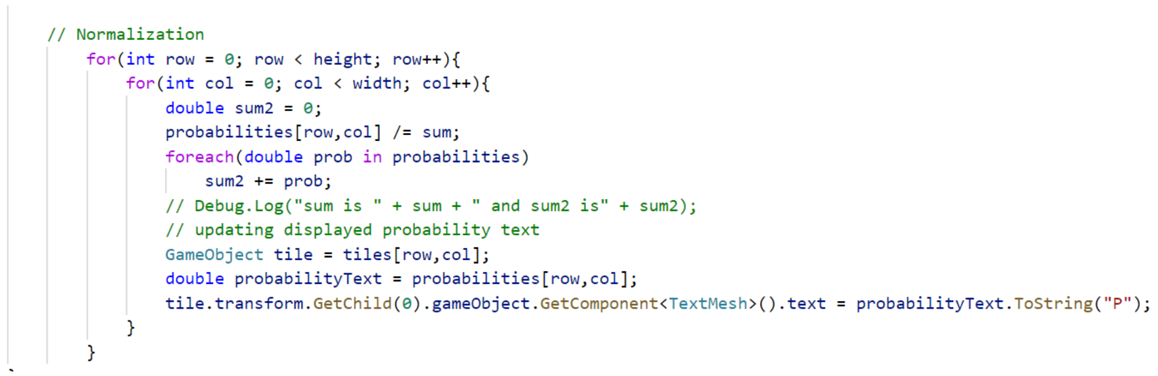
distance1 and distance2 are the variables that delimit the estimated range of distances between the current tile and the ghost ring. Those values are the ones used to compute our probabilities as follows.



```
        // Error Checking
        if(!(distance1 == -999 || distance2 == -999)){

            // Compute the probability of Ghost in current cell given the new approximated
            // range of distances between the current cell and the ghost cell
            double p_Col_Dist = 2 * Proba.MonteCarloProbability(distance1, distance2, 0);
            // Debug.Log("P_Col_Dist for tile ("+row+","+col+") is " + p_Col_Dist+"+\n
            //For parameters "+distance1+" and " + distance2);


            // updating the probability table (before normalization)
            probabilities[row,col] *= p_Col_Dist; // prior probability * probability of color
                                                  //given (approximate) distance from ghost
        }
        // keep count of the sum of probabilities in order to normalize in the next step

        sum += probabilities[row,col];
        // Debug.Log("sum is "+sum);
    }
```

If we click on a cell that gives us a color orange. The ghost is likely 1 to 2 cells away from the ghost. If we then compute the probability for a cell that is at a distance of 3 away from the ghost, it is similar to computing the probability of a cell to have a color orange (1 to 2 tiles away from the ghost) given that is it 1 to 2 tiles away from the red ghost ring.

We then normalize the computed probabilities for all tiles:



```
// Normalization
    for(int row = 0; row < height; row++){
        for(int col = 0; col < width; col++){
            double sum2 = 0;
            probabilities[row,col] /= sum;
            foreach(double prob in probabilities)
                sum2 += prob;
            // Debug.Log("sum is " + sum + " and sum2 is" + sum2);
            // updating displayed probability text
            GameObject tile = tiles[row,col];
            double probabilityText = probabilities[row,col];
            tile.transform.GetChild(0).gameObject.GetComponent<TextMesh>().text = probabilityText.ToString("P");
        }
    }
```

Discussion and Conclusion

While we managed to reach mostly satisfying results with our approach (see the Demo video), the game remains probabilistic, and with noisy censor measurements, we are not guaranteed to always have high probabilities converge to the right cell, it is not rare that the cell with the highest probability of containing the ghost be a cell adjacent to the one that actually contains the Ghost.