Sensor Models and Mapping

Chapter 9

Objectives

- To understand how to create maps from raw sensor data.
- To understand simple sensor sensor models
- Investigate and model the errors associated with sensors and their data.
- To understand how occupancy grids are used to represent maps
- To understand how to extract obstacle features from raw sensor data as well as from grids.

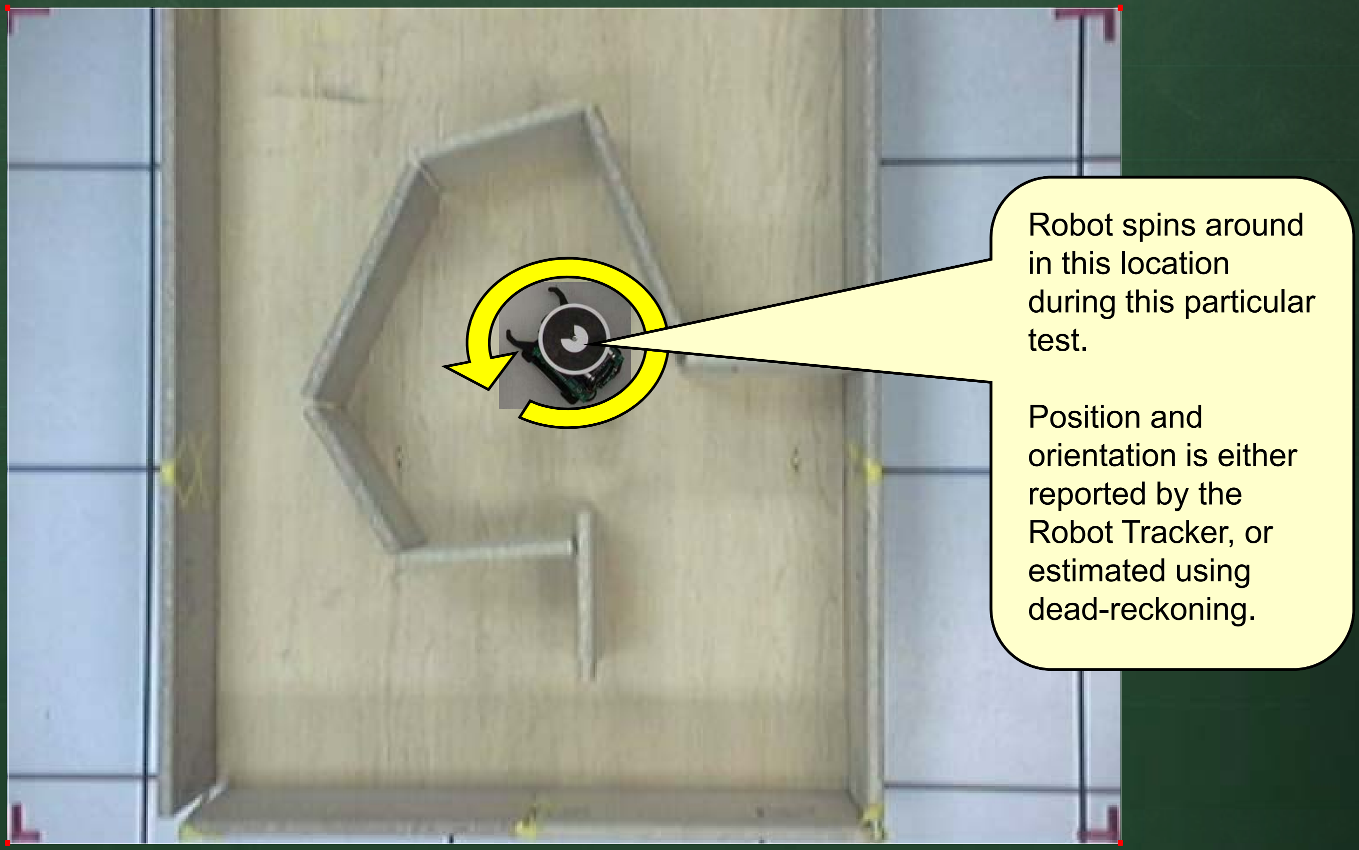
What's in Here ?

- **Mapping With Raw Data**
  - Mapping Raw Ping))) Data
  - Mapping Raw DIRRS+ Data
- **Mapping With Simplified Sensor Models**
  - Sensor Models
  - Applying a Sensor Model
  - Sensor Model Implementation
- **More Realistic Sensor Models**
  - Error Distribution
  - Applying a Gaussian Distribution
  - Converting to Probabilities
  - Bayesian Updating
  - Improving the Sensor Model
  - Odds Representation
  - Sensor Data Fusion
- **Extracting Obstacle Features From Raw Data**
  - Line Extraction
  - Split & Merge
  - Incremental
- **Extracting Obstacle Features From Occupancy Grids**
  - Thresholding
  - Border Tracing
  - Line Fitting

Mapping with Raw Data

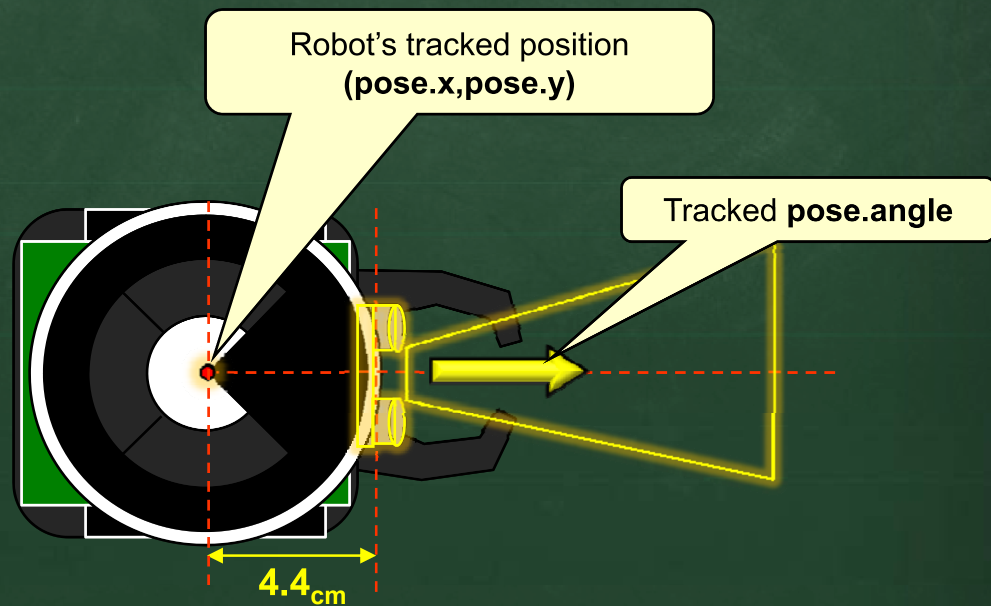
Mapping Raw Ping))) Data

- Consider computing the ranges to obstacles from a fixed location in the following environment:



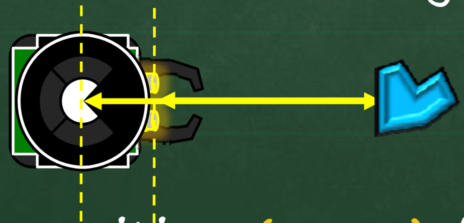
Mapping Raw Ping))) Data

- Assume that RobotTracker/RBC provides the position and angle of the robot accurately.
- We will make use of the *pose.x*, *pose.y* and *pose.angle* information and combine this with the sonar readings.



Mapping Raw Ping))) Data

- The sensor's position is 4.4_{cm} away from the robot's center (x, y) position at 0°
  - where x = pose.x and y = pose.y from the RobotTracker
- We need to adjust all distance readings by:
  - adding 4.4_{cm} since the objects are further away from the center of the robot than they are from the sensor



  - calculating a position (x_o, y_o) for the obstacle by incorporating the robot's pose (x, y, angle) and the range reading.

Mapping Raw Ping))) Data

- To convert range distances we must compute the position of the sensed obstacle.

- Assuming that (x,y) is provided from the Robot-Tracker in units of pixels, we need to either:

  – convert (x,y) into cm first, resulting in cm units for (x_o, y_o):

    x_o = (distance + 4.4_{cm}) * COS(angle) + (x / 3)

    y_o = (distance + 4.4_{cm}) * SIN(angle) + (y / 3)

    1cm ≈ 3pixels in Robot Tracker

  – convert the range readings into pixels first, resulting in pixel units for (x_o, y_o):

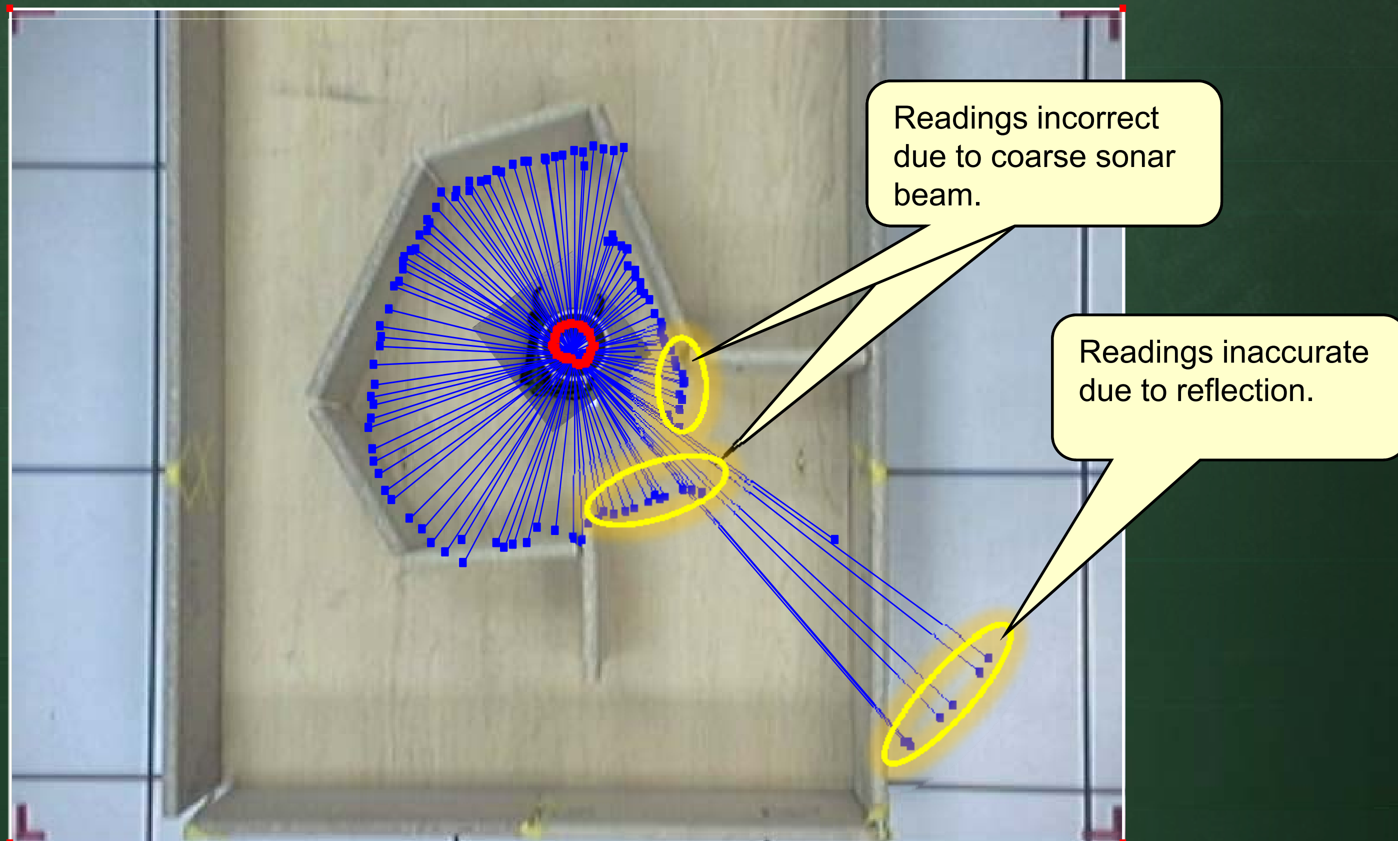
    x_o = [(distance + 4.4_{cm}) * 3] * COS(angle) + x

    y_o = [(distance + 4.4_{cm}) * 3] * SIN(angle) + y

Mapping Raw Ping))) Data

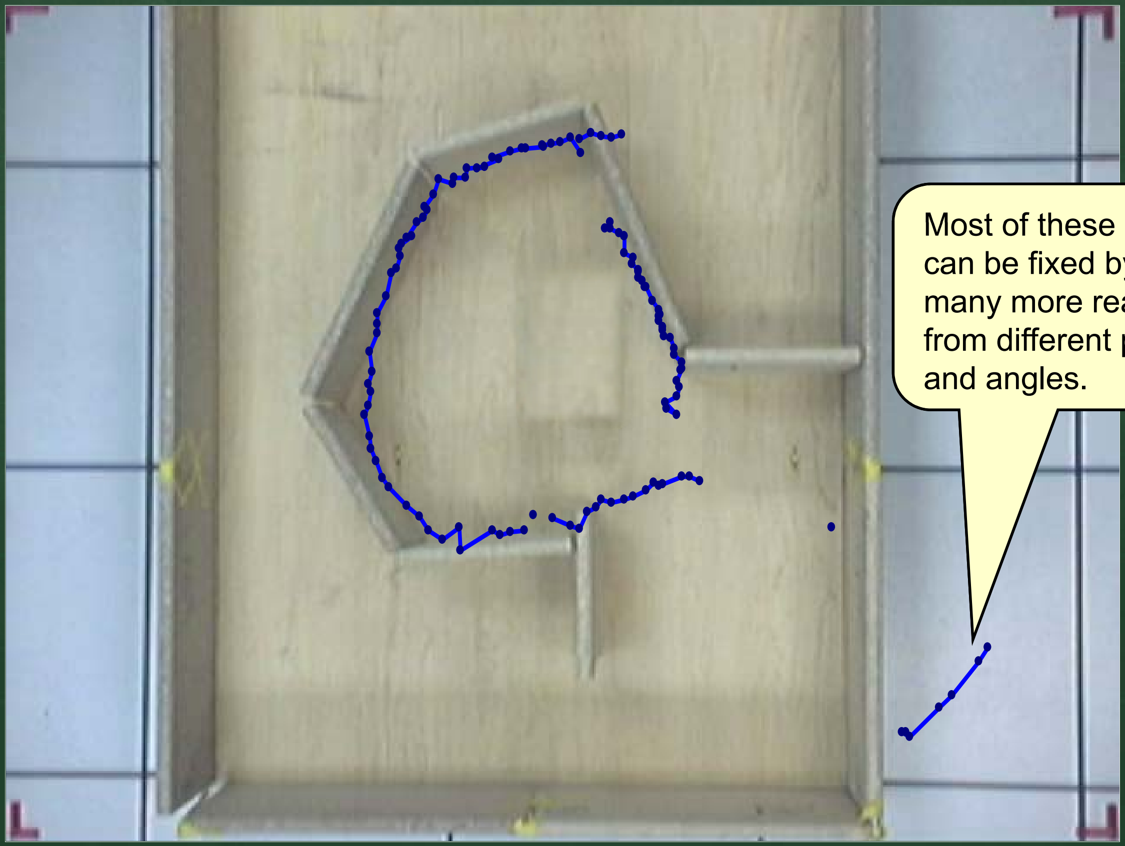
- Blue lines show readings to obstacle from robot's center position (x,y) to the computed obstacle position

(x_o, y_o)



Mapping Raw Ping))) Data

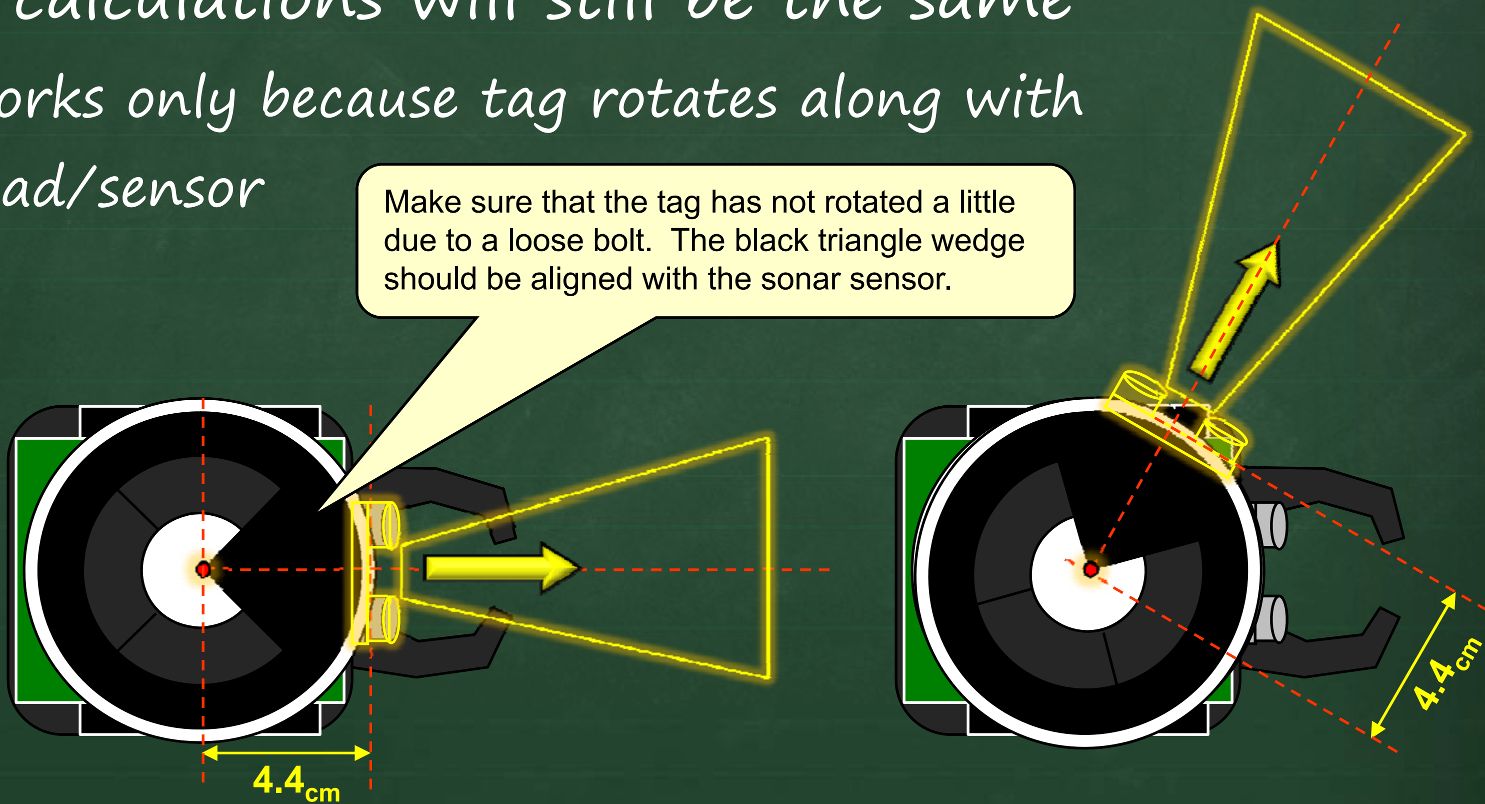
- Result is a "rough" outline of environment with some inaccurate readings.



Most of these problems can be fixed by taking many more readings from different positions and angles.

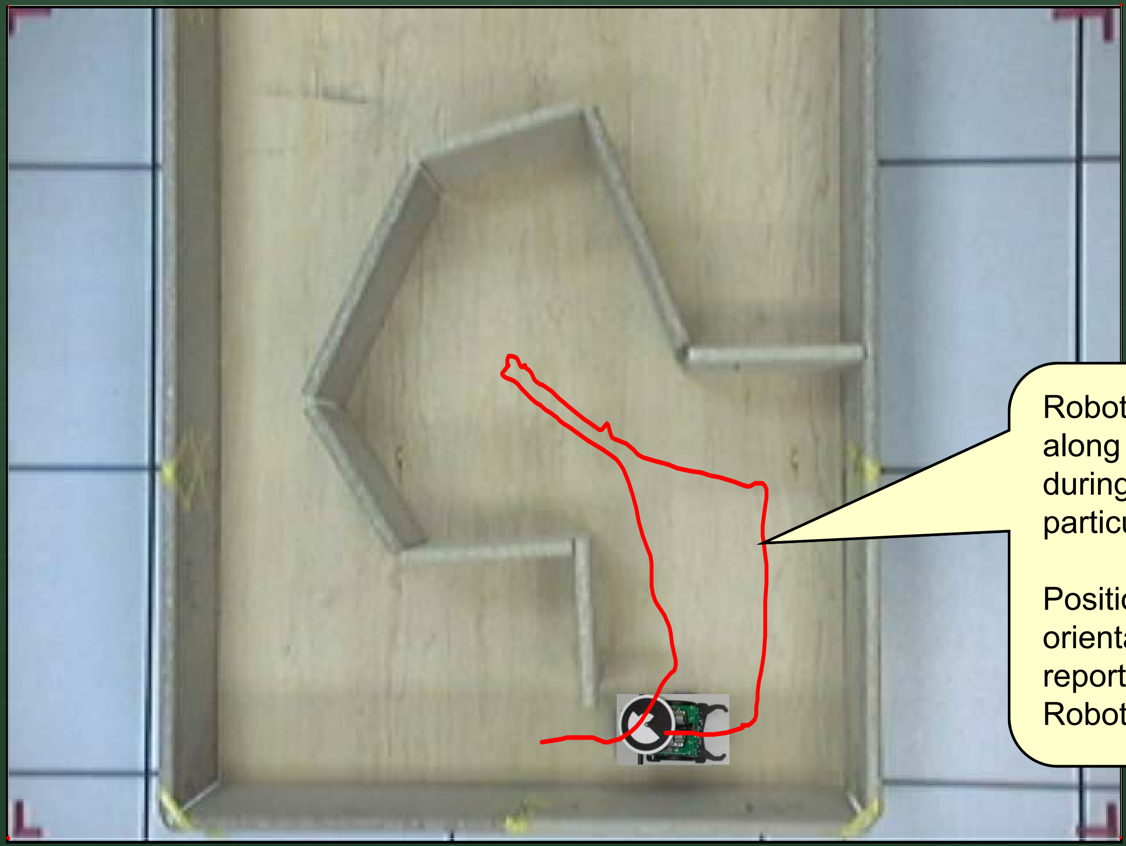
Changing Head Angle

- Even if the robot's head is not facing forward, then the calculations will still be the same
  - Works only because tag rotates along with head/sensor



Mapping Raw DIRRS+ Data

- Consider computing the ranges to obstacles along a path in the following environment:

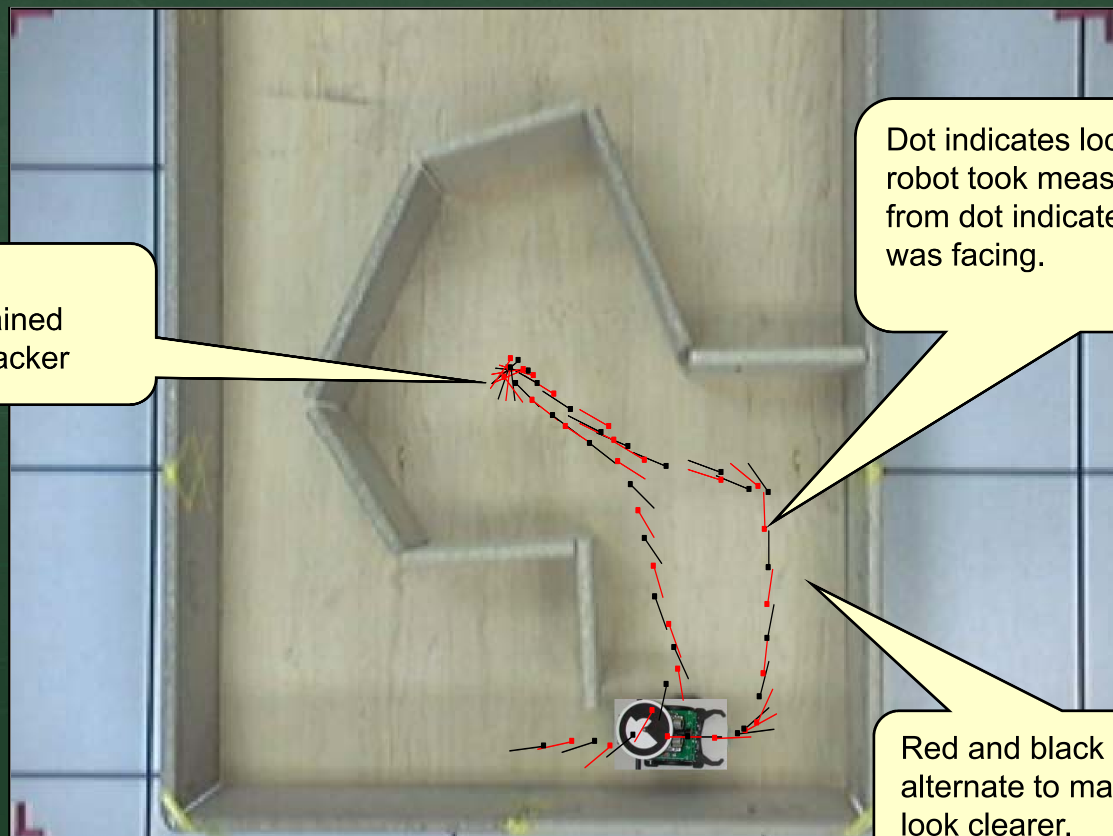


Robot travels along this path during this particular test.

Position and orientation is reported by the Robot Tracker.

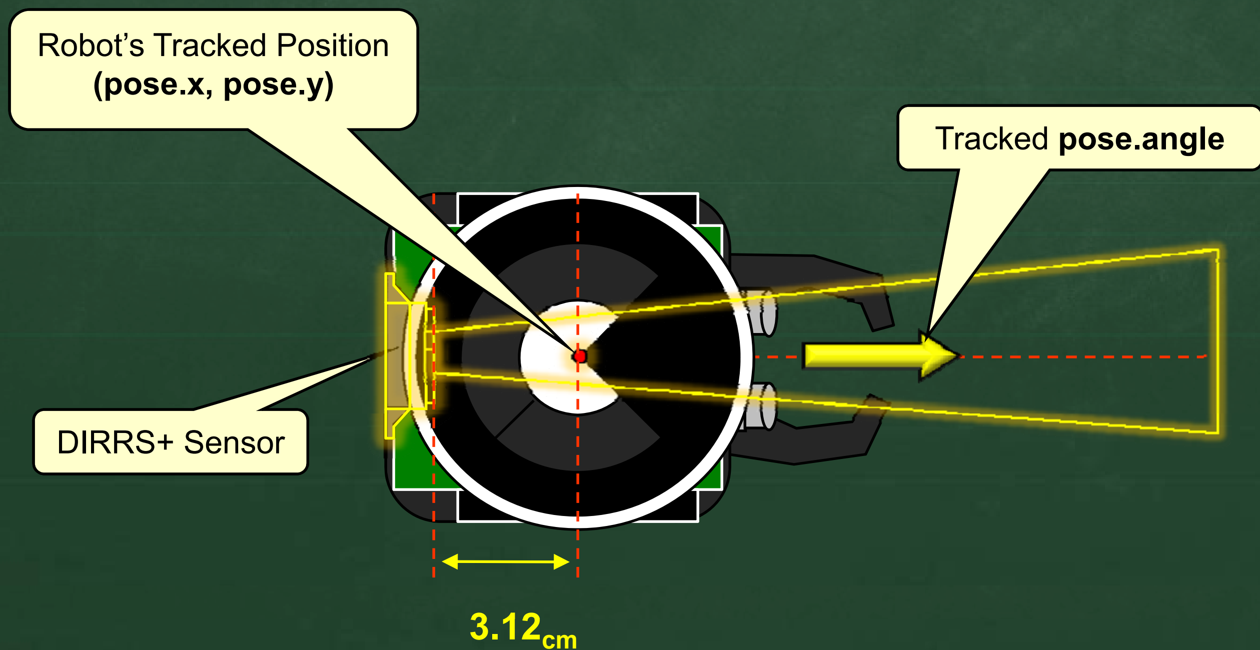
Mapping Raw DIRRS+ Data

- Take measurements along path at particular locations:



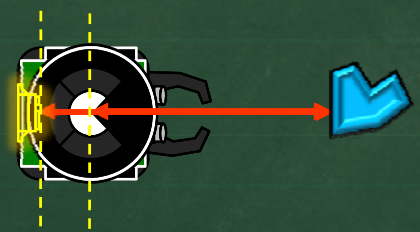
Mapping Raw DIRRS+ Data

- Assume that RobotTracker/RBC provides the position and angle of the robot accurately.
- We will make use of the pose.x, pose.y and pose.angle information and combine this with the IR readings.



Mapping Raw DIRRS+ Data

- The sensor's position is 3.12_{cm} behind the robot's center (x, y) position.
- We need to adjust all distance readings by:
  – subtracting 3.12_{cm} since the objects are closer to the center of the robot than they are from the sensor



  – calculating a position (x_o, y_o) for the obstacle by incorporating the robot's pose $(x, y, angle)$ and the range reading.

Mapping Raw DIRRS+ Data

- To convert range distances we must compute the position of the sensed obstacle.
- Assuming that (x,y) is provided from the Robot-Tracker in units of pixels, we need to either:
  - convert (x,y) into **cm** first, resulting in **cm** units for (x_o,y_o):

$$x_o = (distance - 3.12_{cm}) * COS(angle) + (x / 3)$$
$$y_o = (distance - 3.12_{cm}) * SIN(angle) + (y / 3)$$

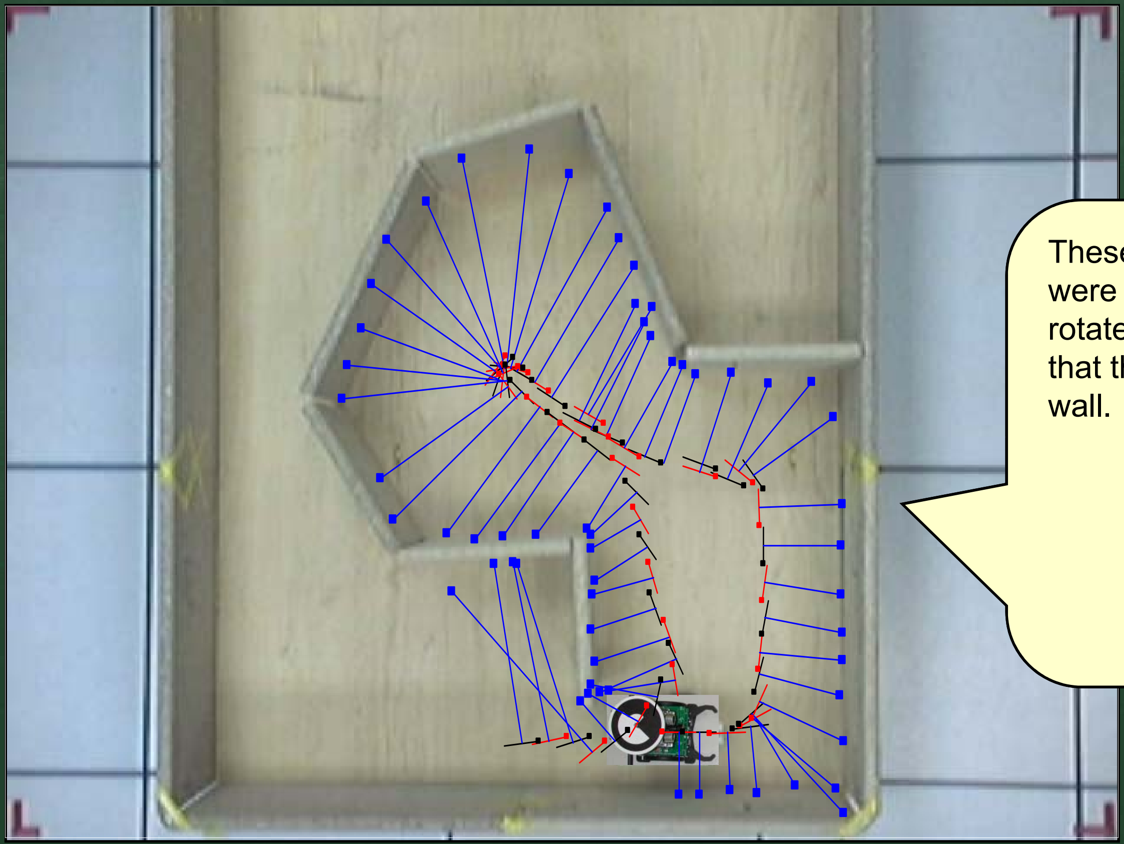
1cm ≈ 3pixels in RobotTracker

  - convert the range readings into pixels first, resulting in pixel units for (x_o, y_o):

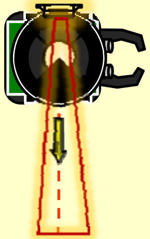
$$x_o = [(distance - 3.12_{cm}) * 3] * COS(angle) + x$$
$$y_o = [(distance - 3.12_{cm}) * 3] * SIN(angle) + y$$

Mapping Raw DIRRS+ Data

- Blue lines show readings to obstacle from robot's center location (x, y):

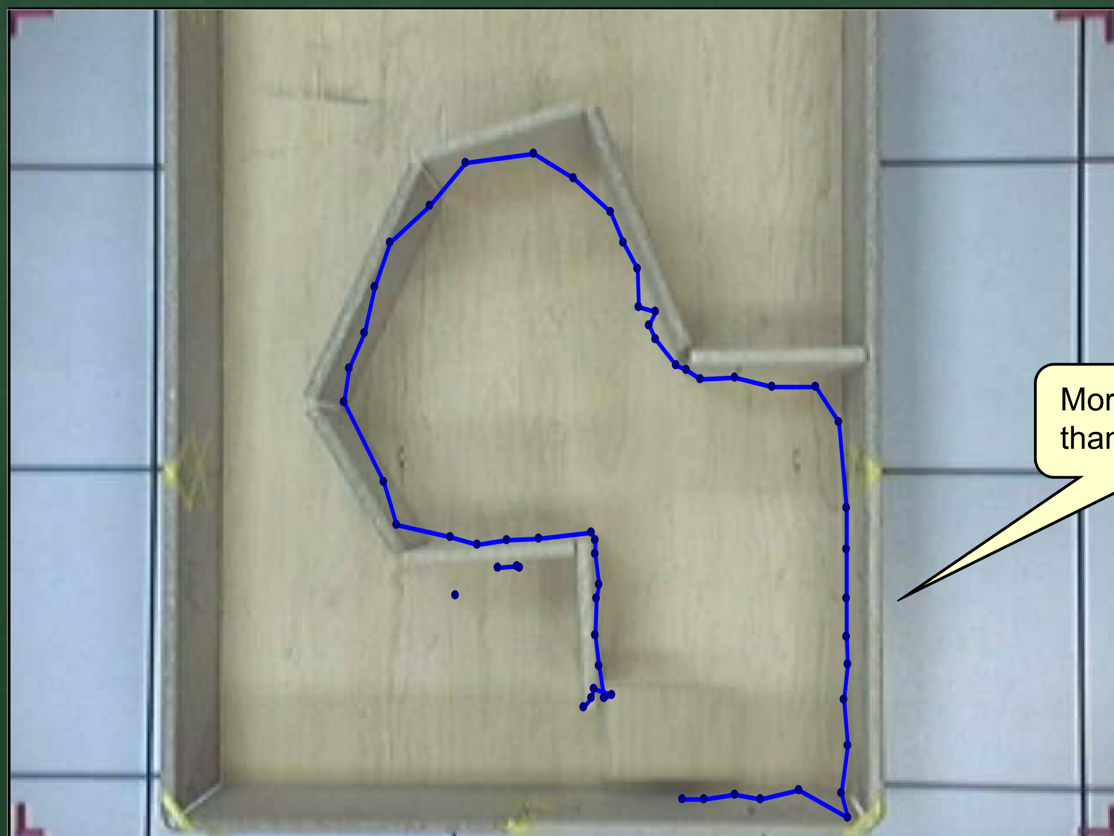


These particular readings were taken with the head rotated 90° to the right so that the sensor faced the wall.



Mapping Raw DIRRS+ Data

- Resulting map has reasonable accuracy:
  - Map can be refined by taking additional readings



More accurate than sonar.

Mapping With Simplified Sensor Models

Sensor Models

- Before using a sensor for mapping, a ***sensor model*** should be developed:
  - specifies how the sensor readings are to be interpreted
  - depends on physical parameters of sensor (e.g., beam width, accuracy, precision etc…)
  - must be able to deal reasonably with noisy data
- For range sensors, they all have similar common characteristics that must be dealt with:
  - *range errors* (distance accuracy)
  - *angular resolution* (beam width)
  - *noise* (invalid data)

Sensor Models

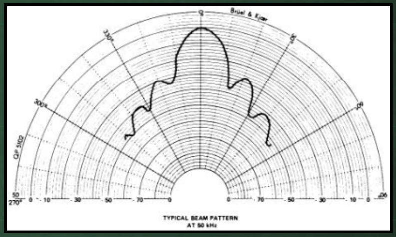
- Various ways to come up with a sensor model:
  - *Empirical*: Through testing
  - *Subjective*: Through Experience
  - *Analytical*: Through analysis of physical properties

- Once sensor model is determined, it is applied to each sensor reading so as to determine how it affects the map being built.

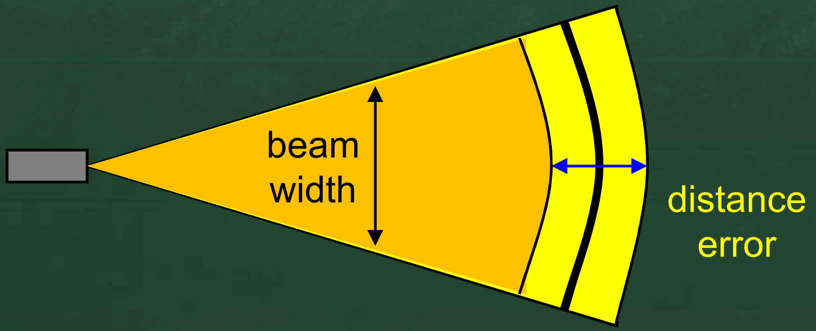
- We will consider our sensor models in terms of how they are used in generating occupancy grid maps.

Sensor Models

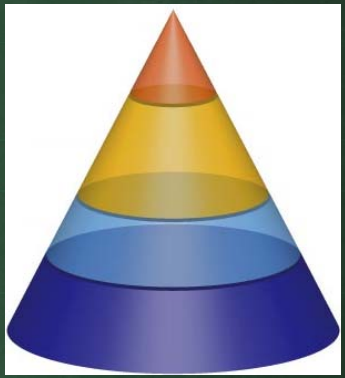
- Our sensor model will consider distance accuracy and beam width.
- Sensor beam width is not easy to model
  - has different width at different distances
  - different obstacles have different reflective effects



- Most models assume that beam is a cone-shaped wedge:



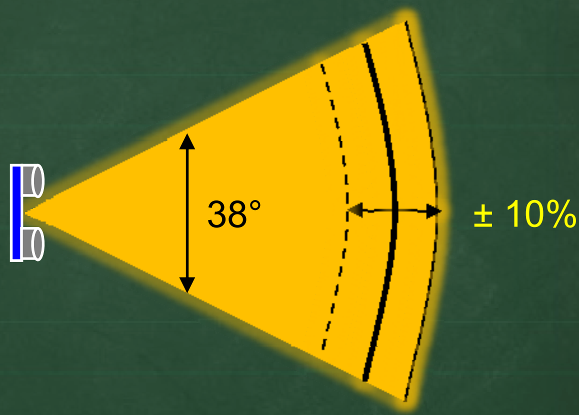
Sensor Models



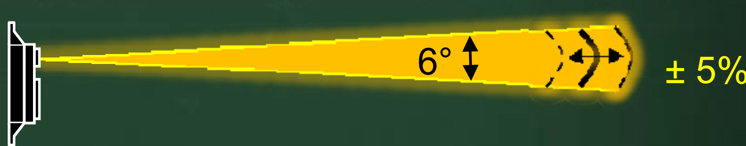
- We will assume that our sensors have beams with this simple cone shape:
  - actually an approximation of the true shape
  - simplifies calculations
  - will vary beam width and distance error with each sensor
- How do we pick beam width and distance error ?
  - beam width and distance error may vary between individual sensors of the same type
  - beam width and distance error usually obtained through experimentation
  - take average of many readings at certain distances

Sensor Models

- We will make the following assumptions regarding our two types of ranging sensors:
  - Ping))) sensor
    - beam width of 38°
    - distance error of ±10%

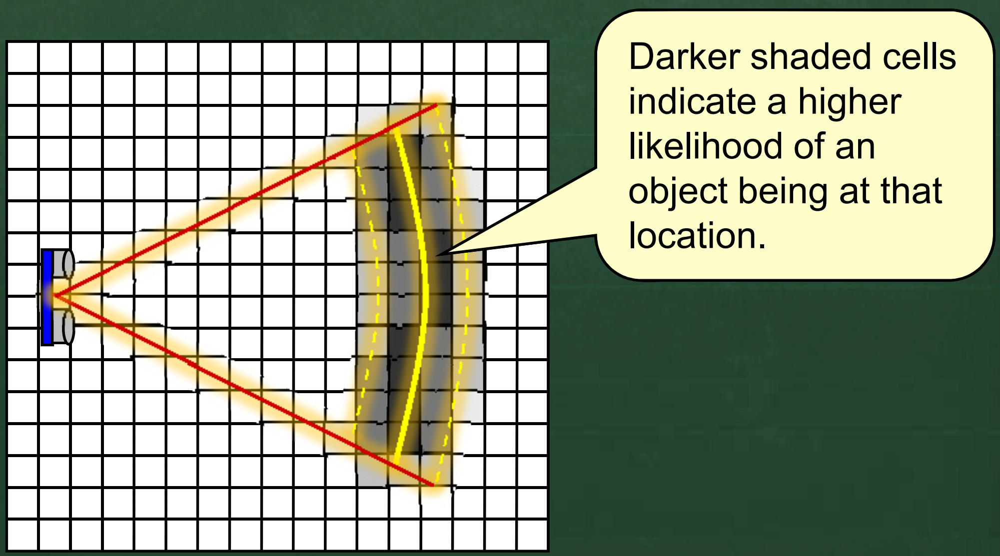


  - DIRRS+ sensor
    - beam width of 6°
    - distance error of ±5%



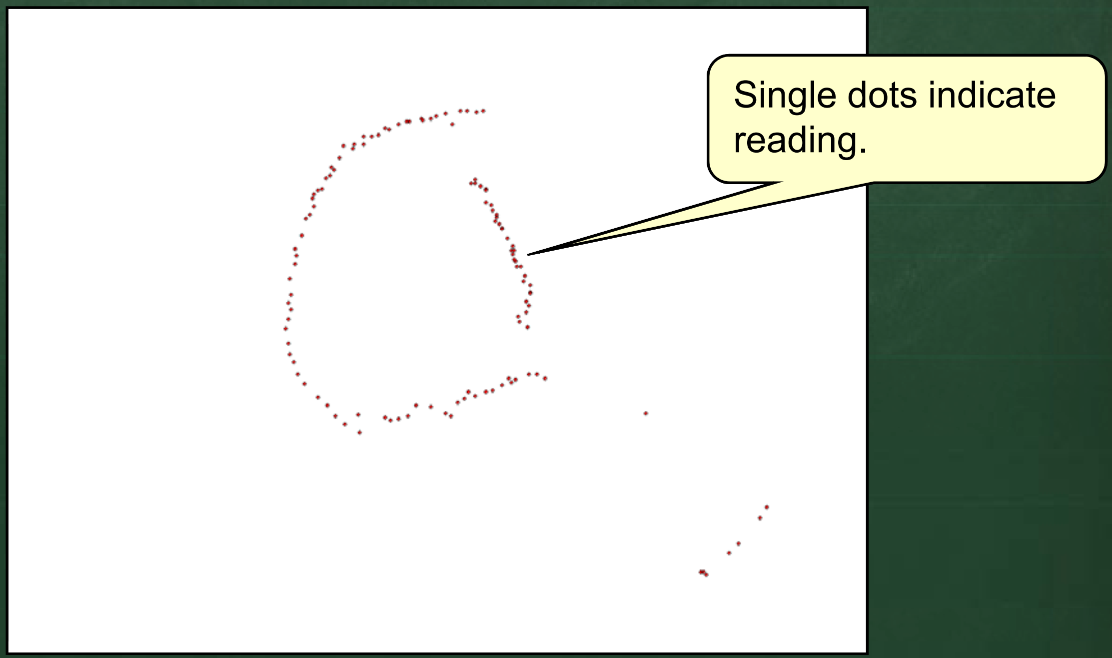
Sensor Models

- What does this all mean ?
- When we detect an obstacle, we will project the sensor model onto the grid and assign probabilities to the grid cells by taking into consideration the model.



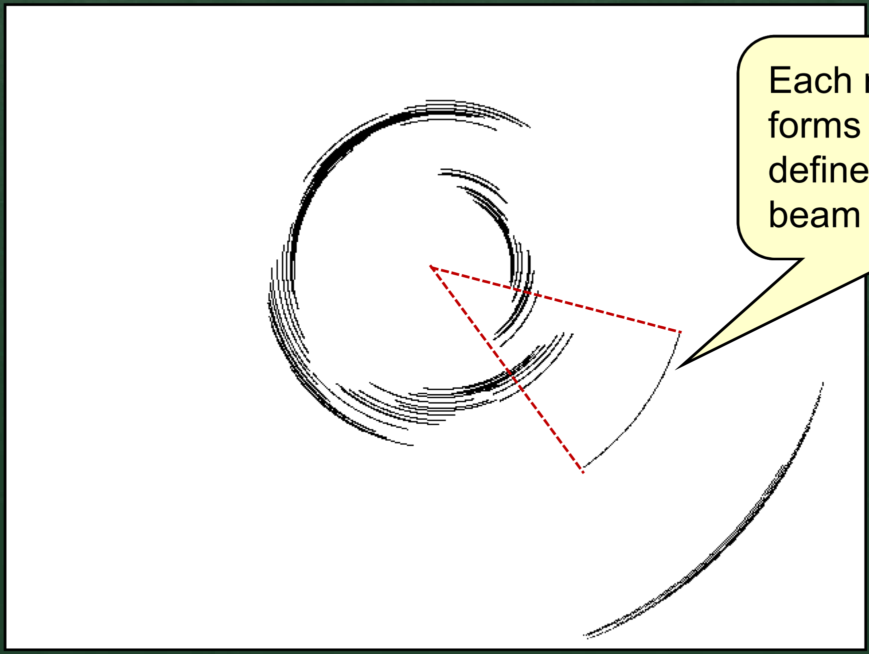
Applying a Sensor Model

- Consider binary map
  - every time sensor detects obstacle at some range, assume that object is there with 100% certainty.
  - did this with readings from our sonar sensor data earlier



Applying a Sensor Model

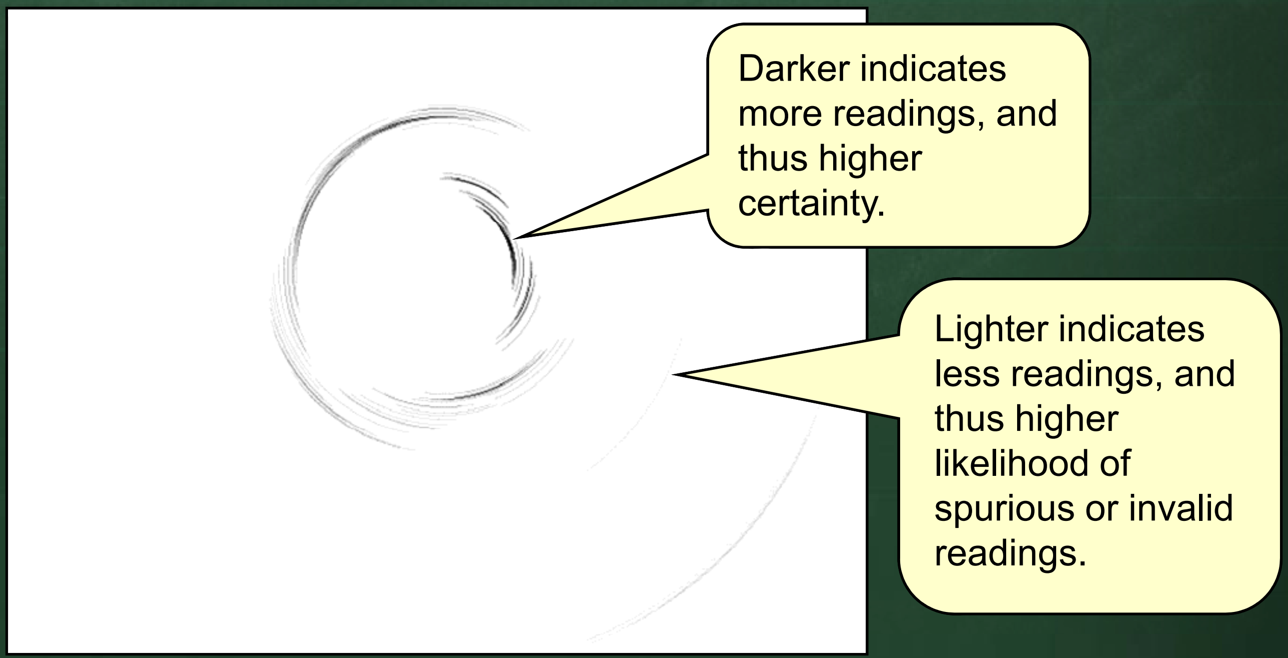
- Actual sensor data is not this precise.
  - must apply sensor model, assuming that object is anywhere within the specified 38° wedge.
  - simplest strategy assumes all readings indicate obstacle



Each reading now forms an arc defined by the beam angle.

Applying a Sensor Model

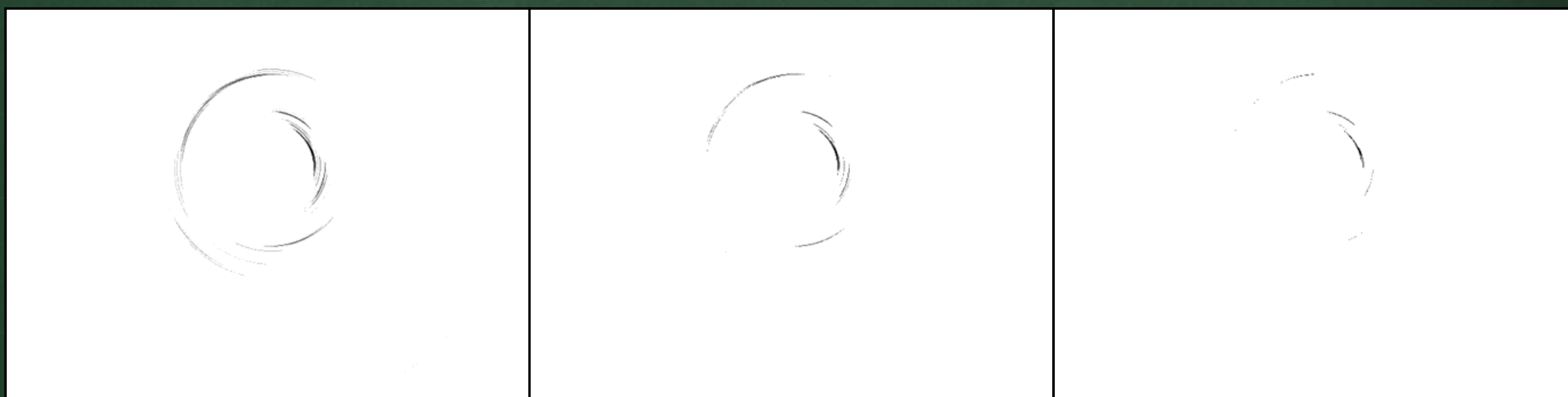
- Now consider a more realistic grayscale map:
  - keep a counter for each grid cell and increment every time sensor has reading at this location
  - over time, cells become darker with multiple readings



Applying a Sensor Model

- Can eliminate invalid readings by ignoring cells with counter below a certain threshold.

- Here are some results of applying a threshold:



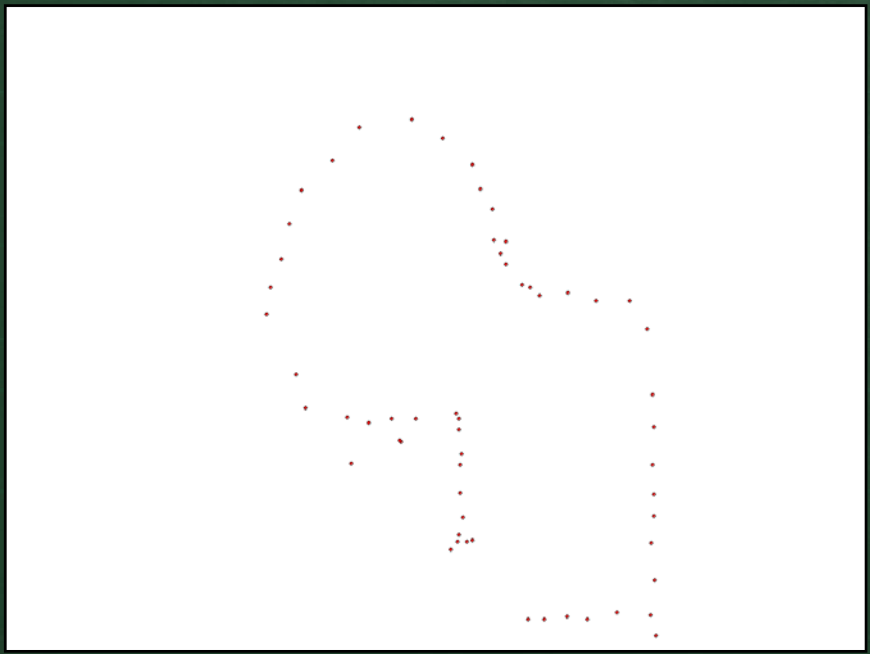
discard anything below 25% of max count.

discard anything below 50% of max count.

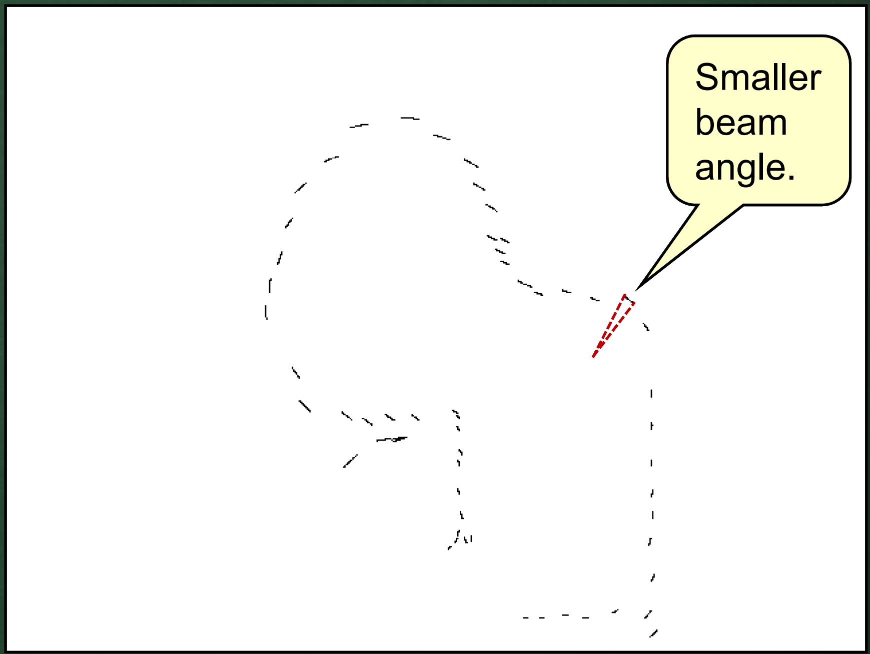
discard anything below 75% of max count.

Applying a Sensor Model

- Consider applying our IR model to our IR data:
  - More accurate than sonar with only 6° beam angle.



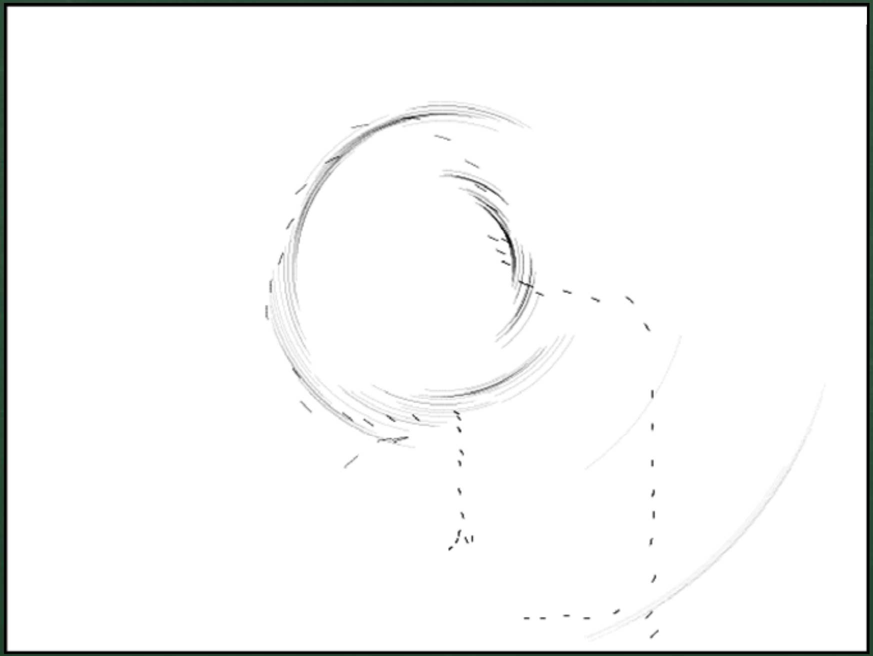
Data without model



Data with 6° model

Applying a Sensor Model

- Eventually, we want to "fuse" together the sensor data from our IR and sonar data.



- Before we do this, we must adjust our map data to account for distance errors too.

Sensor Model Implementation

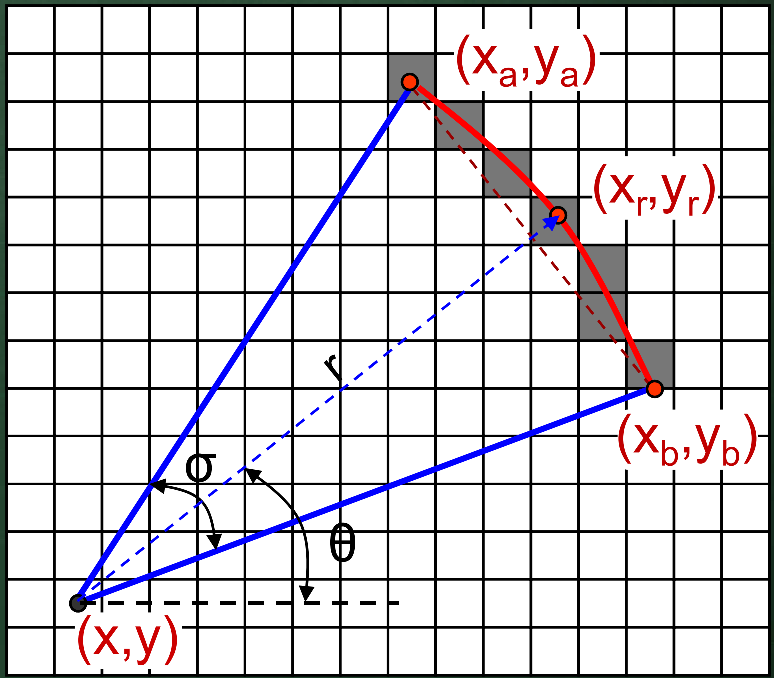
- How do we compute which cells are affected by our sensor model ?
  - Need to determine the cells covered by each arc.
- First, compute arc endpoints:

$x_r = x + r * COS(\theta)$
$y_r = y + r * SIN(\theta)$
$x_a = x + r * COS(\theta - \sigma/2)$
$y_a = y + r * SIN(\theta - \sigma/2)$
$x_b = x + r * COS(\theta + \sigma/2)$
$y_b = y + r * SIN(\theta + \sigma/2)$

θ is the sensor's direction.

r is the sensor's range reading.

σ is the fixed angular resolution of the sensor.



Sensor Model Implementation

- Then compute the angular interval so that we cover each cell along the arc once (roughly):

$$\omega = \sigma / \sqrt{(x_a - x_b)^2 + (y_a - y_b)^2}$$

- Now go to each grid cell along the arc and increment it accordingly:

```
FOR a = -σ/2 TO σ/2 BY ω DO {
    gridX = (x + cos(θ + a) * r);
    gridY = (y + sin(θ + a) * r);
    incrementCell(gridX, gridY);
}
```

Compute the grid cells at various angles along the wedge. We choose angles that should produce unique grid cells by choosing the appropriate angular interval.

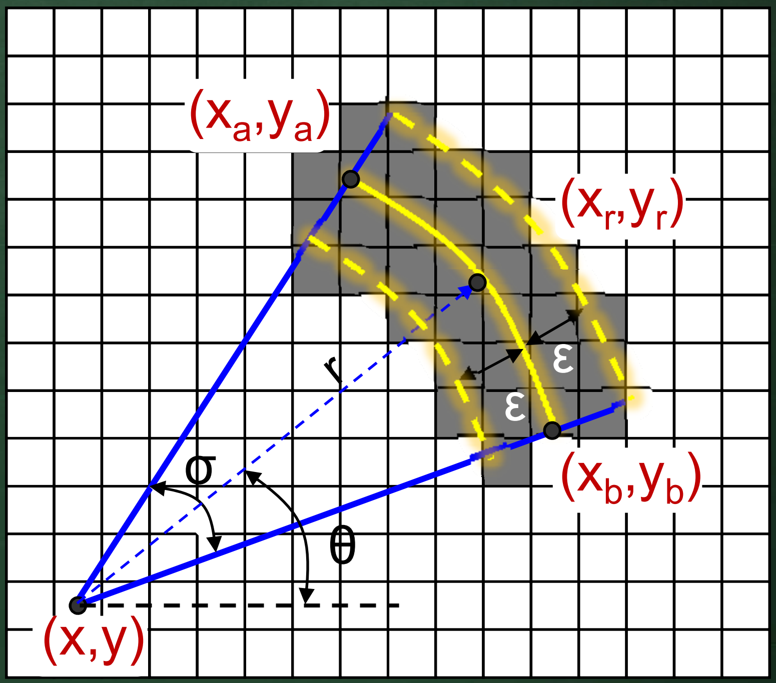
Sensor Model Implementation

- How do we account for the distance error ?
  - we iterate through ranges corresponding to the range ε defined by the distance error.

```
FOR r' = -ε TO ε DO {
   ω = computed as described below
   FOR a = -σ/2 TO σ/2 BY ω DO {
      gridX = (x + cos(θ+a) * (r+r'))
      gridY = (y + sin(θ+a) * (r+r'))
      incrementCell(gridX, gridY)
   }
}
```

$x_a = x + (r+r') * \mathrm{COS}(\theta - \sigma/2)$
$y_a = y + (r+r') * \mathrm{SIN}(\theta - \sigma/2)$
$x_b = x + (r+r') * \mathrm{COS}(\theta + \sigma/2)$
$y_b = y + (r+r') * \mathrm{SIN}(\theta + \sigma/2)$

$$\omega = \sigma / \sqrt{(x_a-x_b)^2 + (y_a-y_b)^2}$$

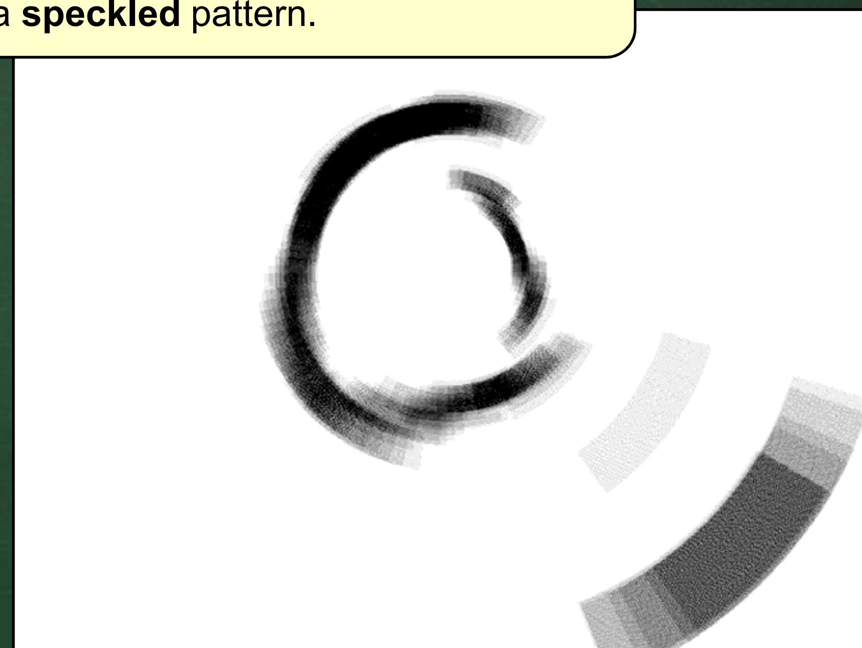


Sensor Model Implementation

- As a result, we end up with a blurred "band" indicating the distance error of the sensor:

Due to round-off inaccuracies, there will likely be some grid cells counted twice and some not counted during a single update. This may lead to a **speckled** pattern.

10% distance error (ε = ±5%)

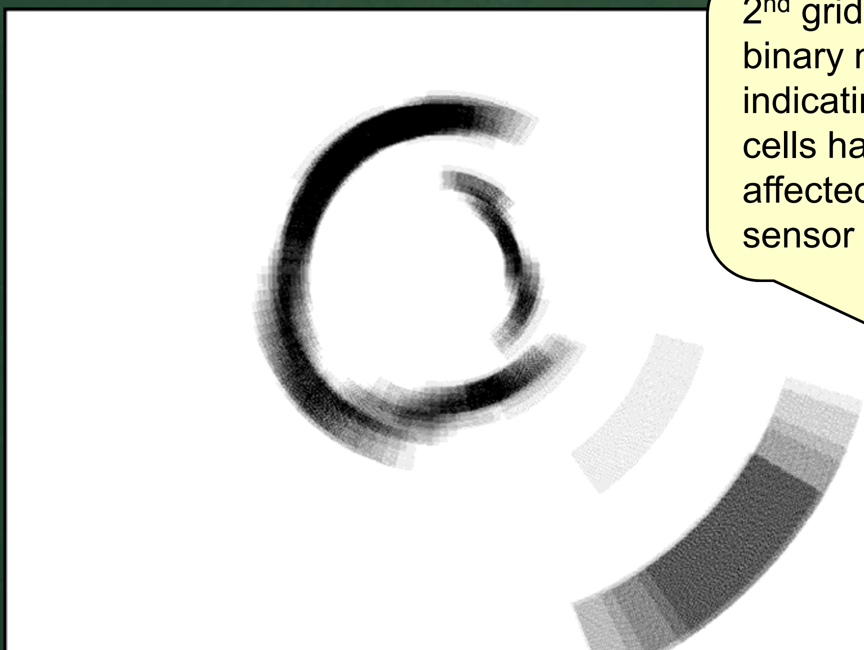


20% distance error (ε = ±10%)

Sensor Model Implementation

- Can eliminate speckled pattern by creating 2nd grid on which to compute sensor readings and then *merge it* onto the real grid when done.

2nd grid acts as a binary mask indicating which cells have been affected by the new sensor readings.

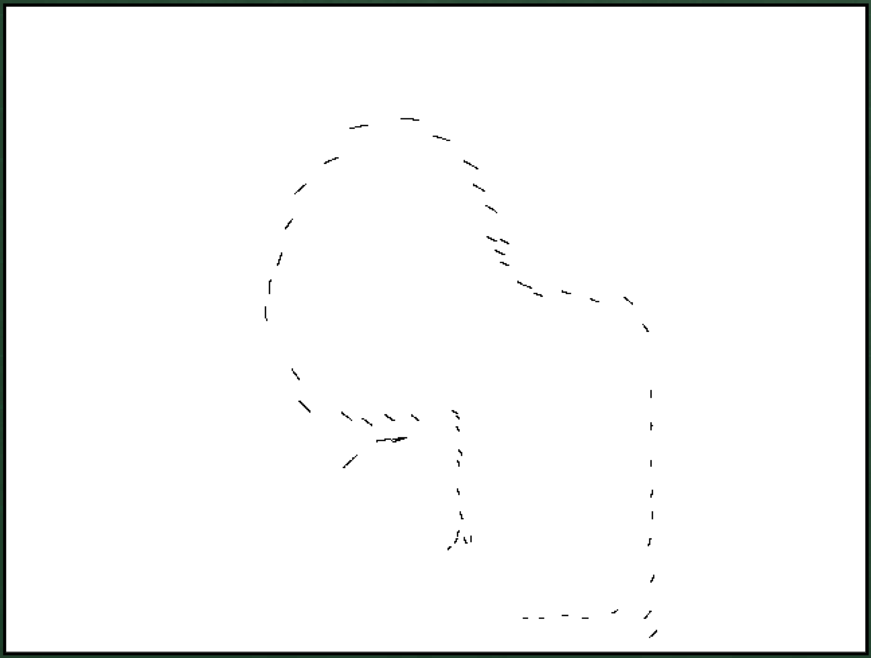


Without 2nd Grid

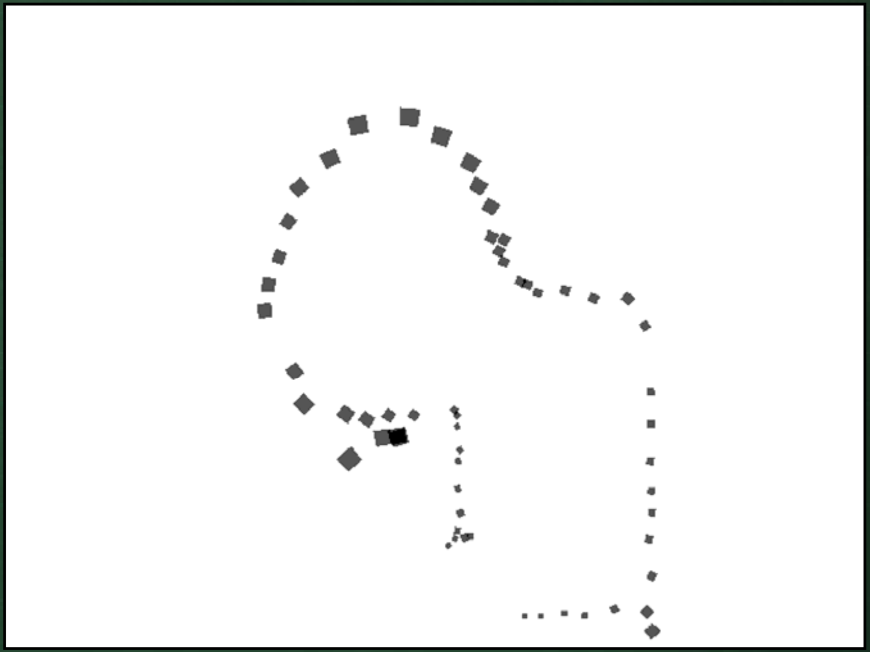
With 2nd Grid (used as a mask)

Sensor Model Implementation

- Here are the results for the IR sensor data when the ±5% distance error of the model is also applied:



Without Distance Error

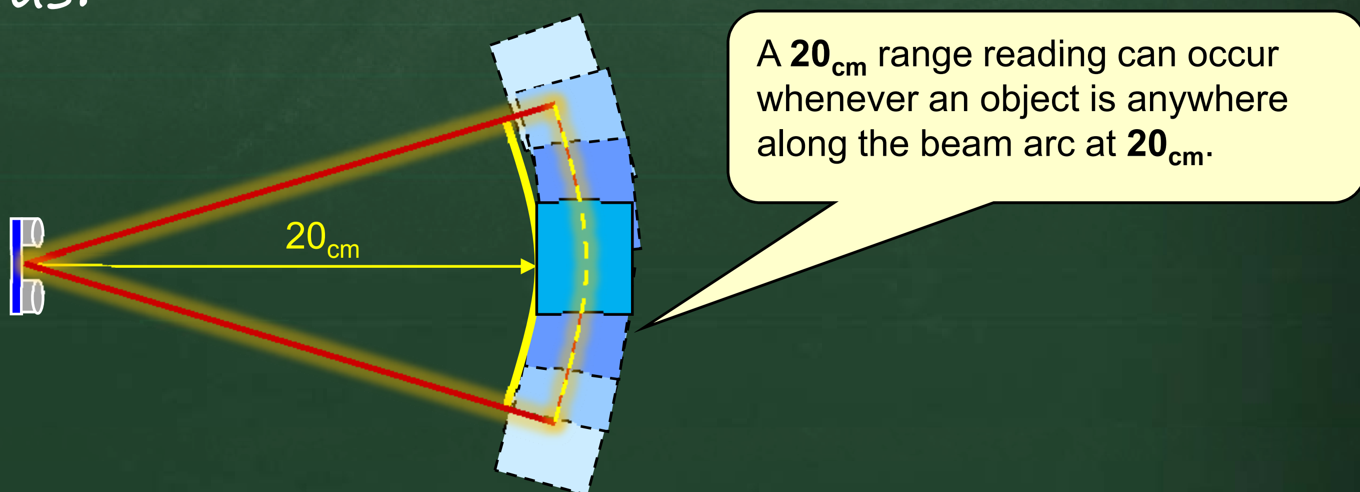


With Distance Error

More Realistic Sensor Models

Error Distribution

- As mentioned earlier, each sensor's readings are subject to angular errors as well as distance errors.
- Consider a sonar with 38° beam angle.
- When object is detected at, say 20_{cm}, it can actually be anywhere within the beam arc defined by the 20_{cm} radius:



Error Distribution

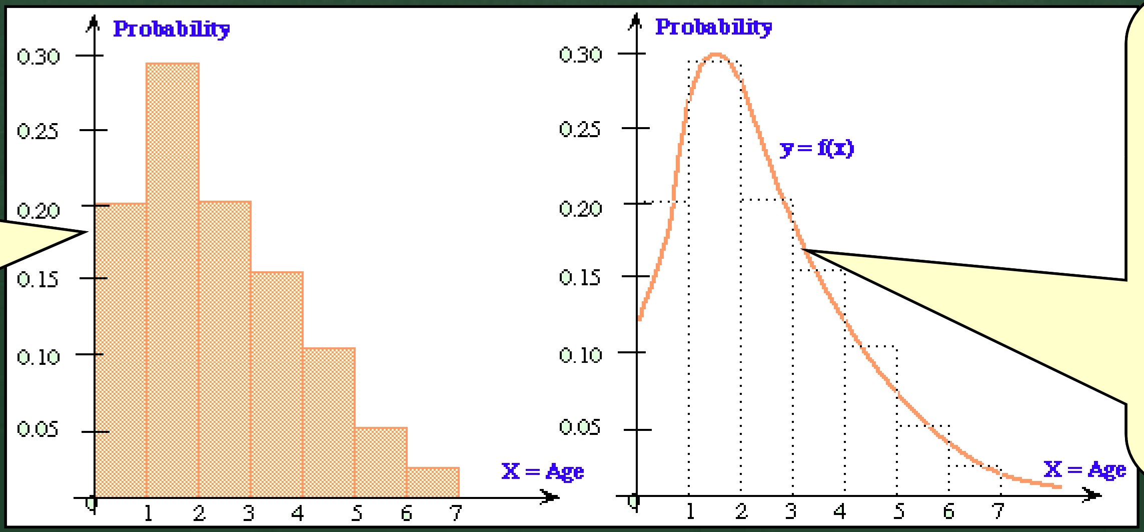
- The likelihood (or probability) that the object is centered across the arc is greater than if the object was off to the side of the arc.
- We can thus express the sensor reading itself as a set of probabilities across the grid.
- As a result, we end up with a probability distribution representing the likelihood that the object is centered at the detected angle.
- Assume that the location (along the arc) of the obstacle is a random variable.

Probability Density Functions

- Random variables operating in continuous spaces are called continuous random variables.
- Assume that all continuous random variables posses a *Probability Density Function* (PDF).

E.g.,

Probability distribution of a car being a certain age.



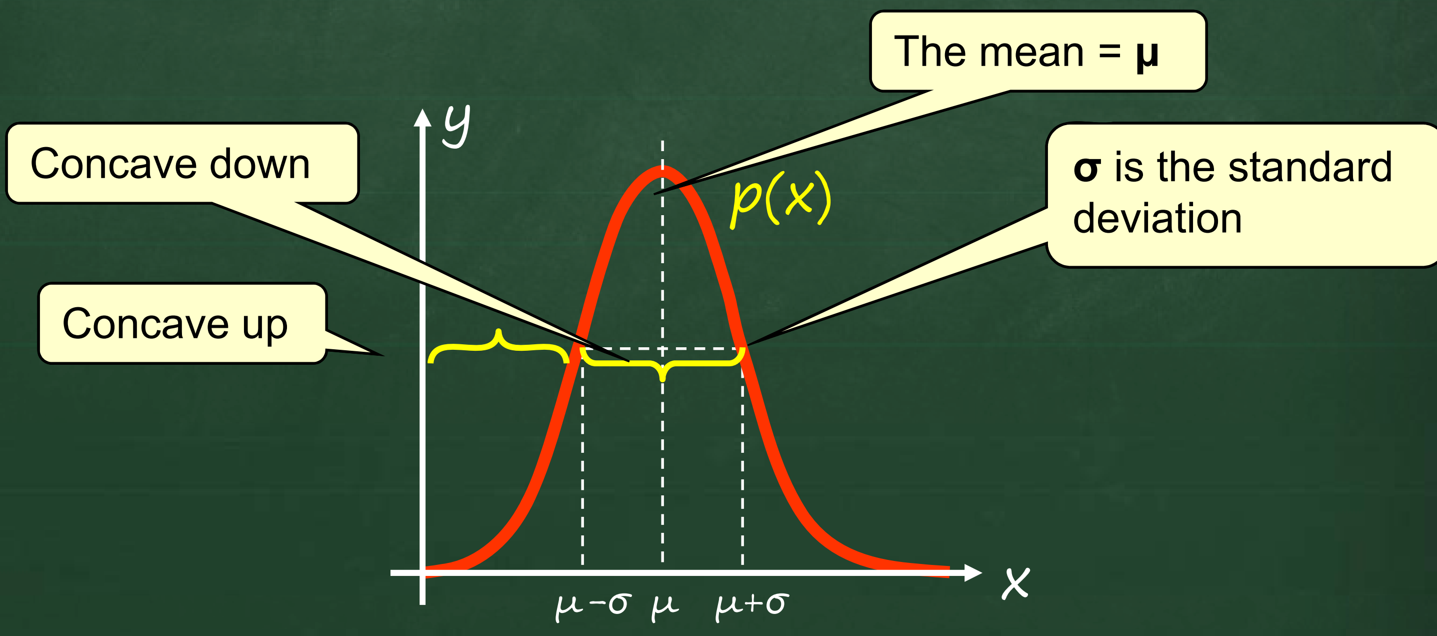
Probability Density Function for this distribution.

(Also known as the ***Probability Distribution Function***).

Probability Density Functions

- Common PDF is the 1-D ***normal distribution***:
  - given mean μ and variance σ^2 the normal distribution is

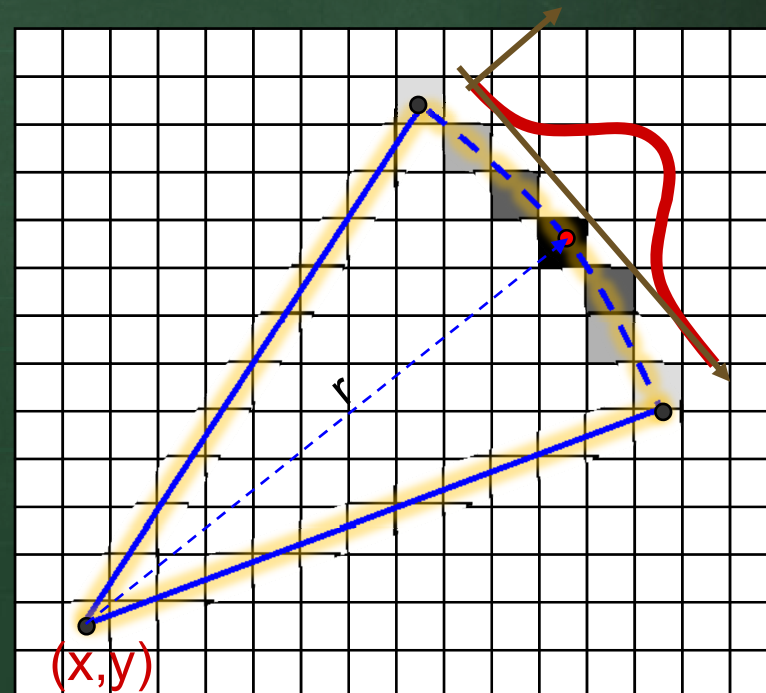
$$p(x) = (2\pi\sigma^2)^{-½} \exp\{ -½ (x-\mu)^2 / \sigma^2 \} = \frac{e^{-(x-\mu)^2 / 2\sigma^2}}{\sigma\sqrt{(2\pi)}}$$



Gaussian Distribution

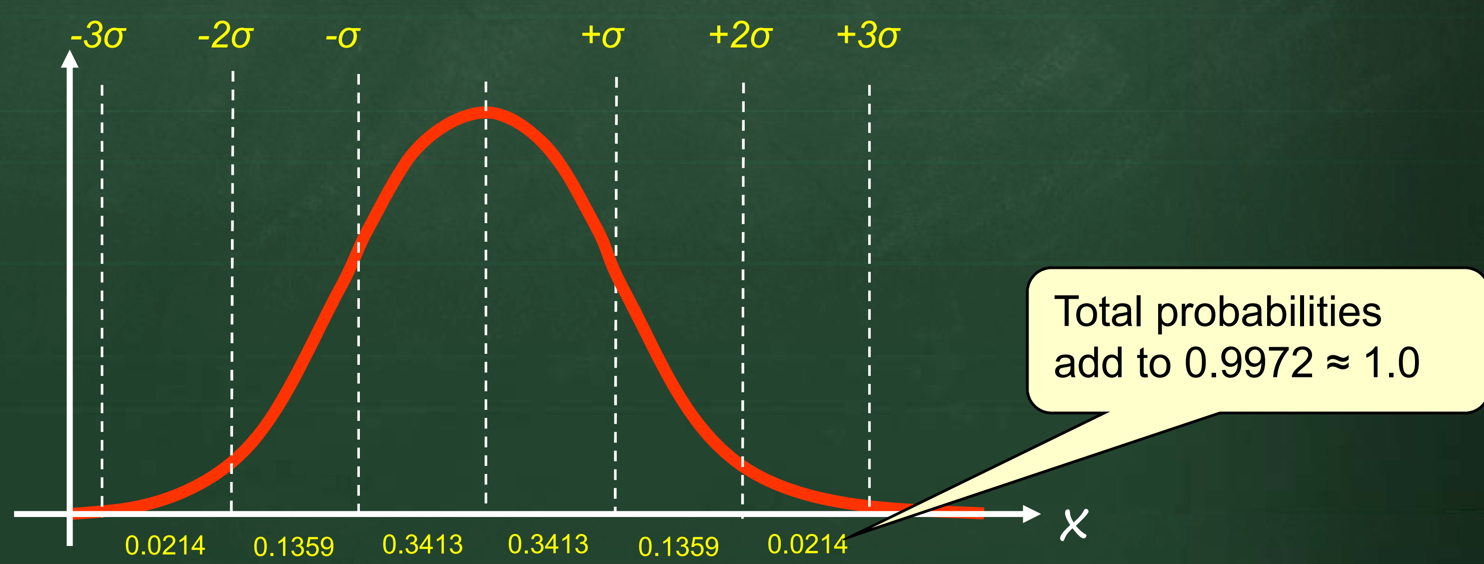
- A more realistic sensor model assigns probabilities to the cells according to some error distribution such as this **Gaussian** (or **Normal**) distribution.

- We can apply this distribution across our arc to assign probabilities that the cell is occupied:



Gaussian Distribution

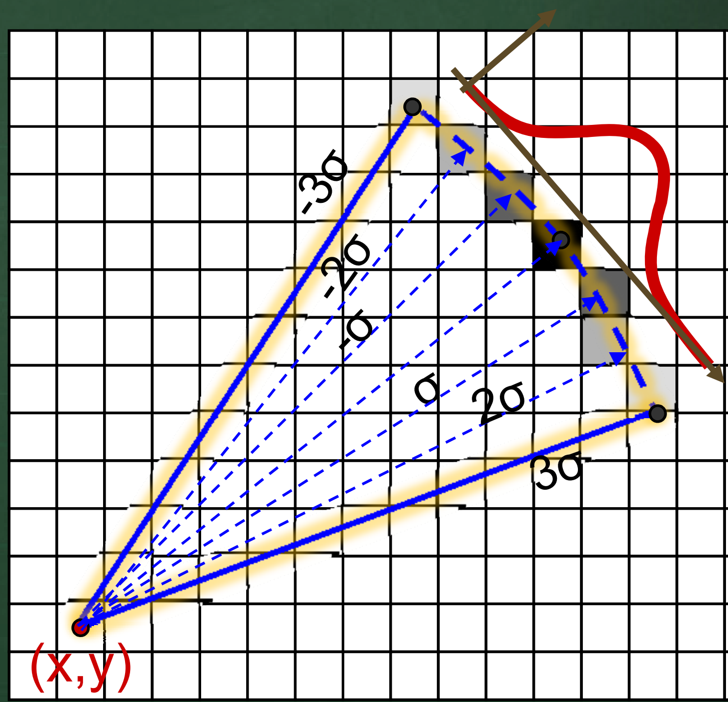
- How do we implement this on our grid ?
- Often the probabilities are approximated using what is known as the six-sigma rule. Which essentially divides the probabilities into 6 probability regions.



Applying Gaussian Distribution

- Hence, we can divide our arc into 6 "pieces" and apply the specific probabilities to the cells in each range.

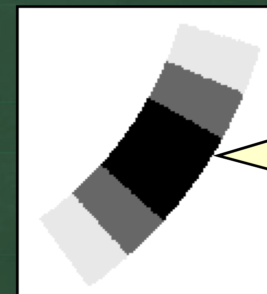
- Cells get probability according to how much they lie in each range



- Hence, we need to convert our grid into probabilities instead of using an integer counter.

Applying Gaussian Distribution

- Here are the results of applying the Gaussian distribution across the angular component of the data:



Angular distribution for single reading.

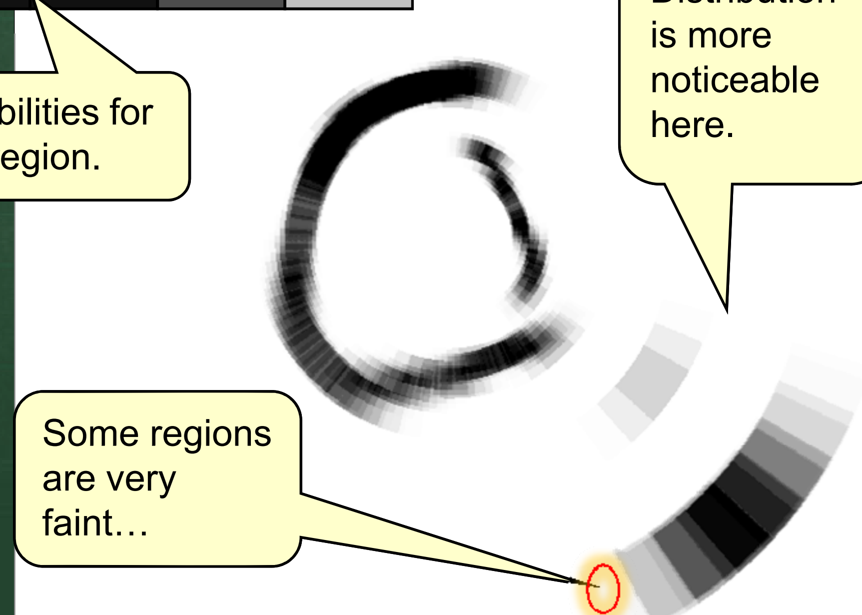
0.0214	0.1359	0.3413	0.3413	0.1359	0.0214

Probabilities for each region.

Without Gaussian Distribution on **Angle**

Distribution is more noticeable here.

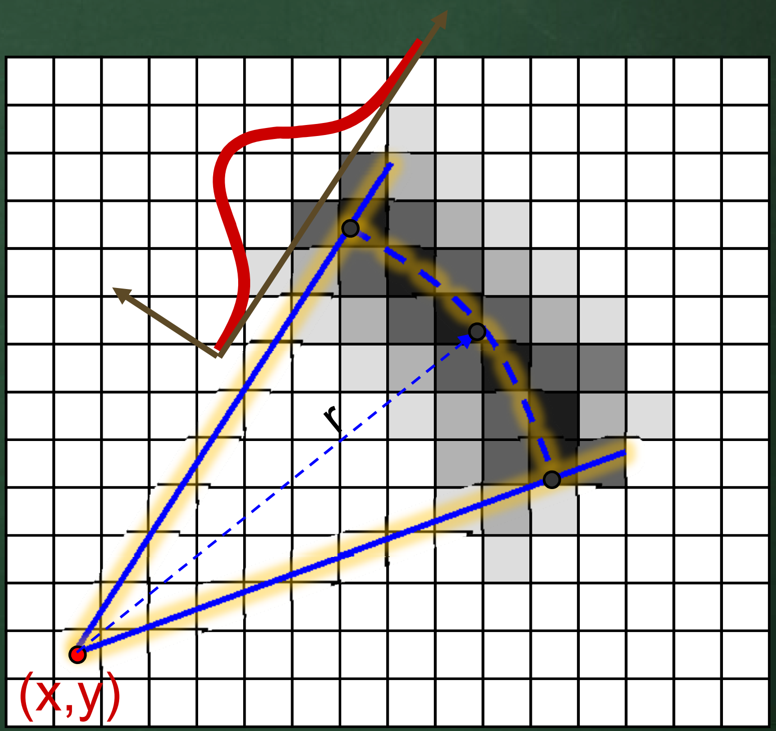
Some regions are very faint…



With Gaussian Distribution on **Angle**

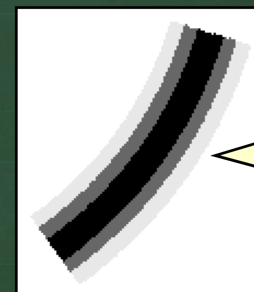
Applying Gaussian Distribution

- The data is still somewhat "band-like".
- Should also apply Gaussian distribution across the distance component.
- Allows for uncertainty in the distance as well.



Applying Gaussian Distribution

- Here are the results of applying the Gaussian distribution across the distance component of the data:



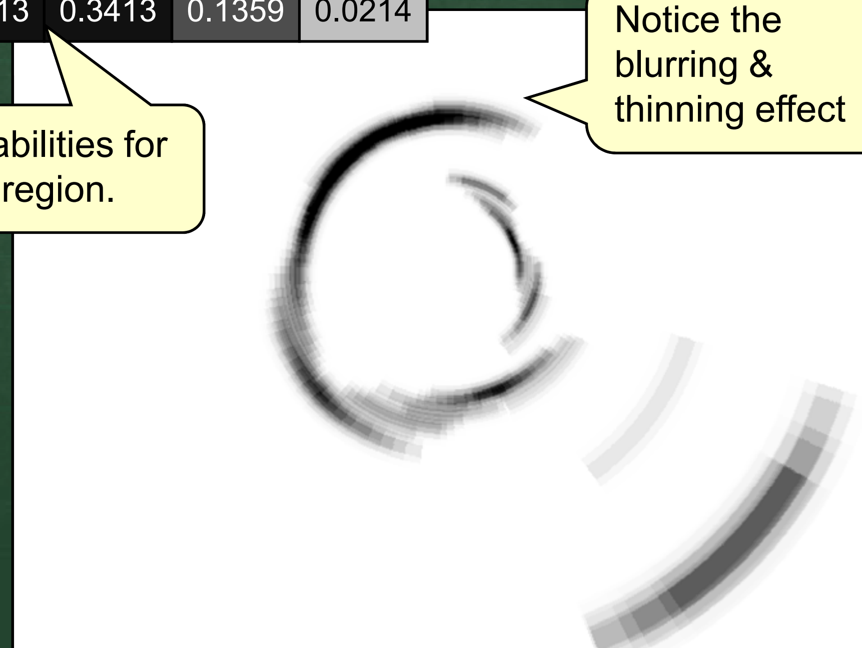
Distance distribution for single reading.

0.0214	0.1359	0.3413	0.3413	0.1359	0.0214

Probabilities for each region.

Notice the blurring & thinning effect

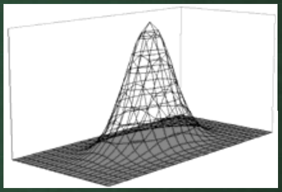
Without Gaussian Distribution on **Distance**

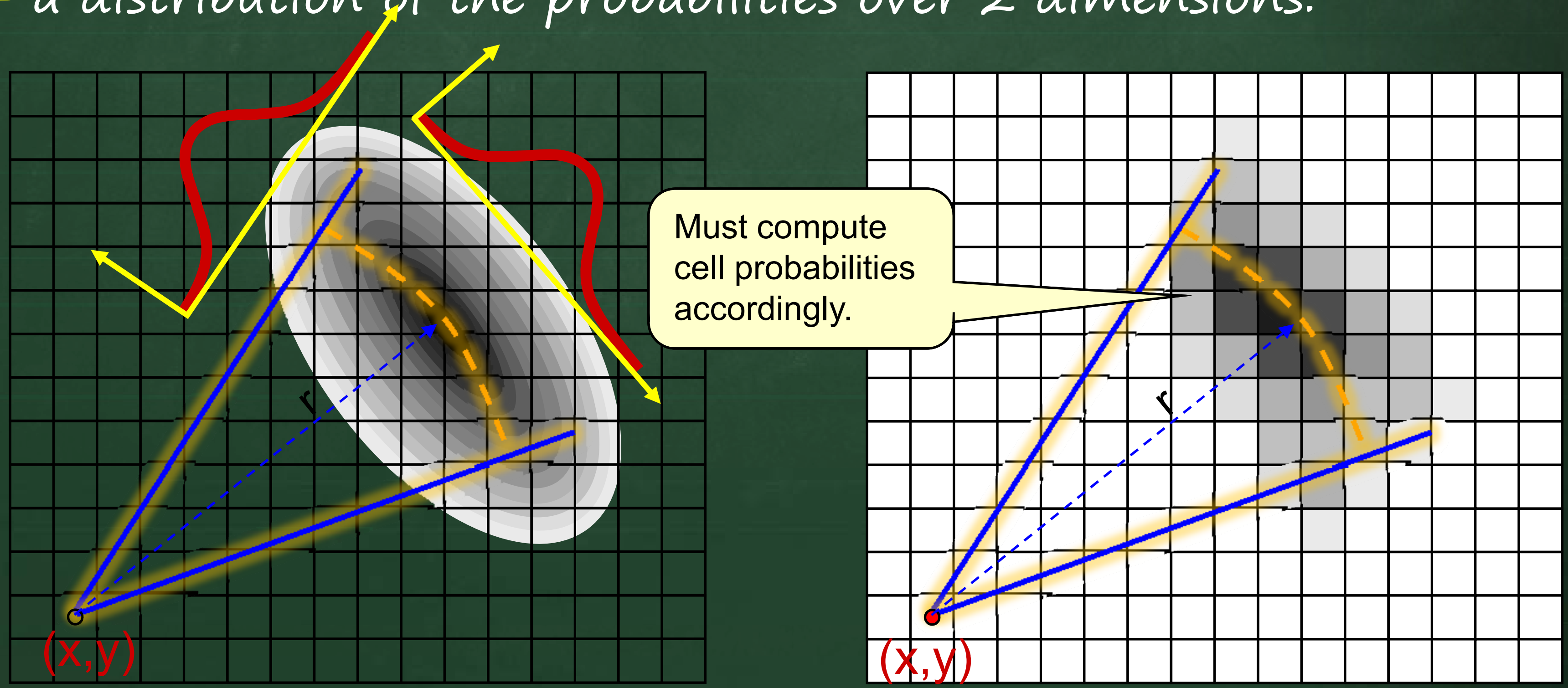


With Gaussian Distribution on **Distance**

Applying Gaussian Distribution

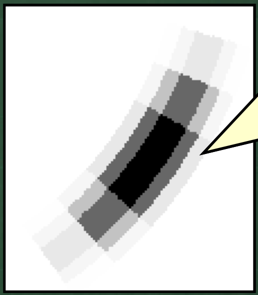
- We will apply the probability distribution over **both** angular and distance components
  - this is known as a *multivariate distribution*
  - a distribution of the probabilities over 2 dimensions.





Applying Gaussian Distribution

- Consider occupancy grid cells covered by single reading
  - 18,519 cells are affected by a single sonar sensor reading, in this particular example
  - all cells lie in one of the ranges below (which all add to 1.0) corresponding to a particular Gaussian probability value:



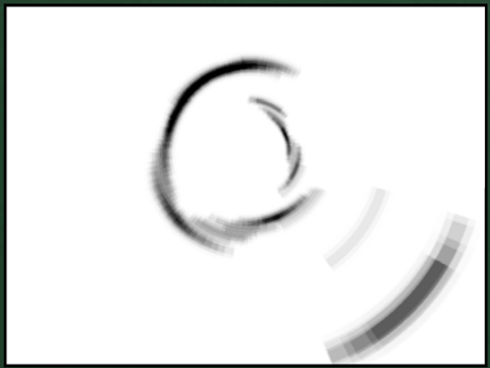
Combined angular & distance distribution for single reading.

0.0004579600	0.0029082602	0.0073038205	0.0073038205	0.0029082602	0.0004579600
0.0029082602	0.0184688115	0.0463826733	0.0463826733	0.0184688115	0.0029082602
0.0073038205	0.0463826733	0.1164856972	0.1164856972	0.0463826733	0.0073038205
0.0073038205	0.0463826733	0.1164856972	0.1164856972	0.0463826733	0.0073038205
0.0029082602	0.0184688115	0.0463826733	0.0463826733	0.0184688115	0.0029082602
0.0004579600	0.0029082602	0.0073038205	0.0073038205	0.0029082602	0.0004579600

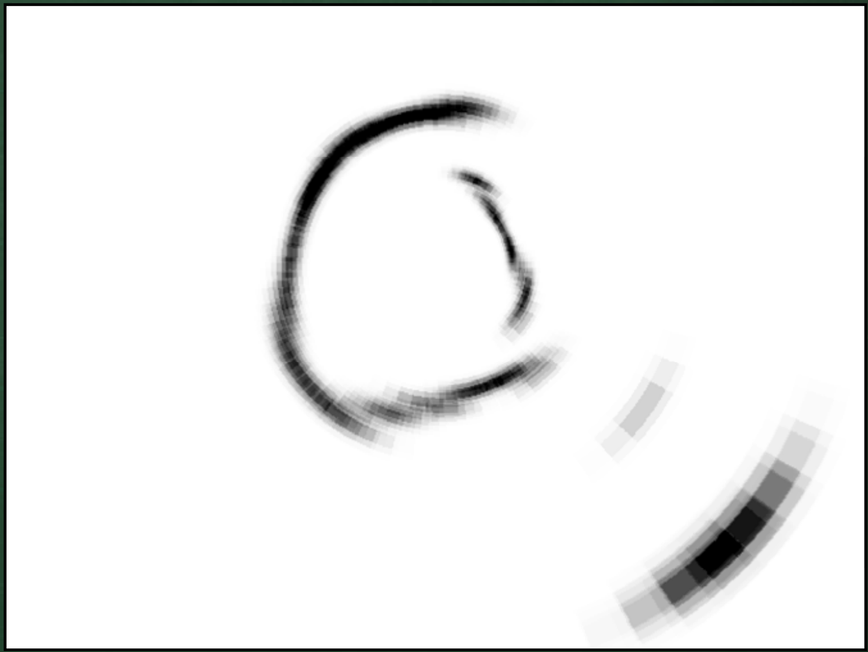
Applying Gaussian Distribution

- Here are the results of applying the Gaussian distribution across the distance component of the data:

With Gaussian distribution on angle



With Gaussian distribution on distance

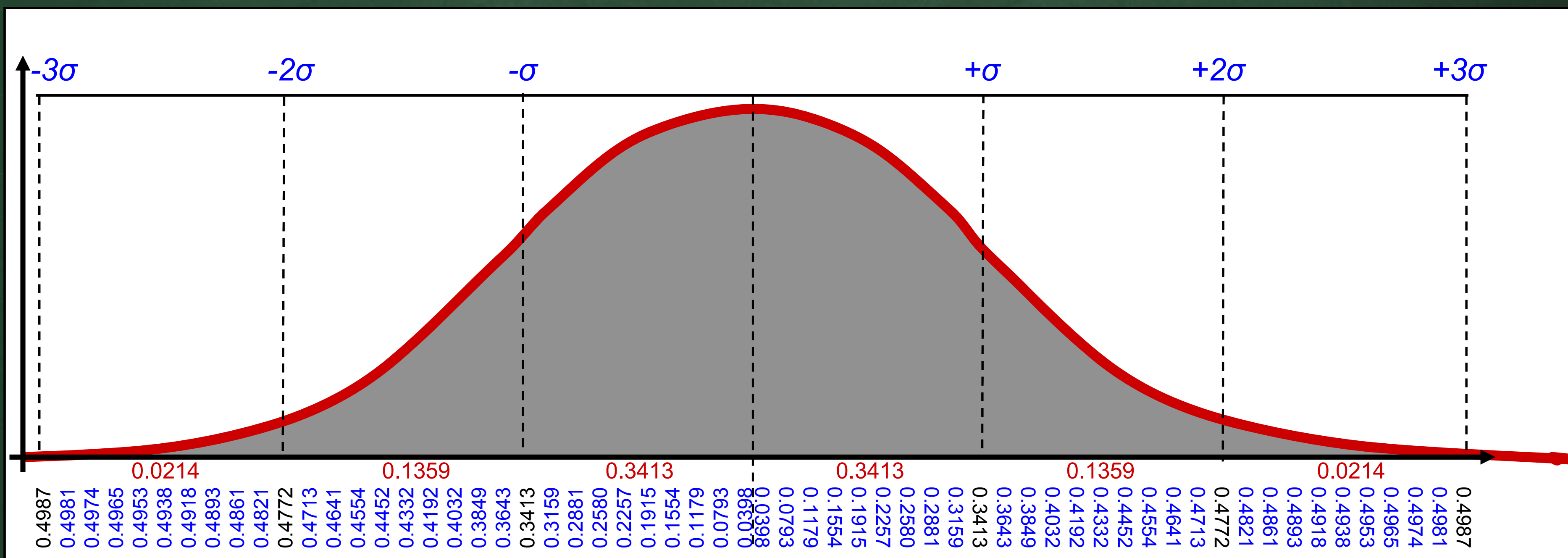


With Gaussian distribution on both angle and distance

Applying Gaussian Distribution

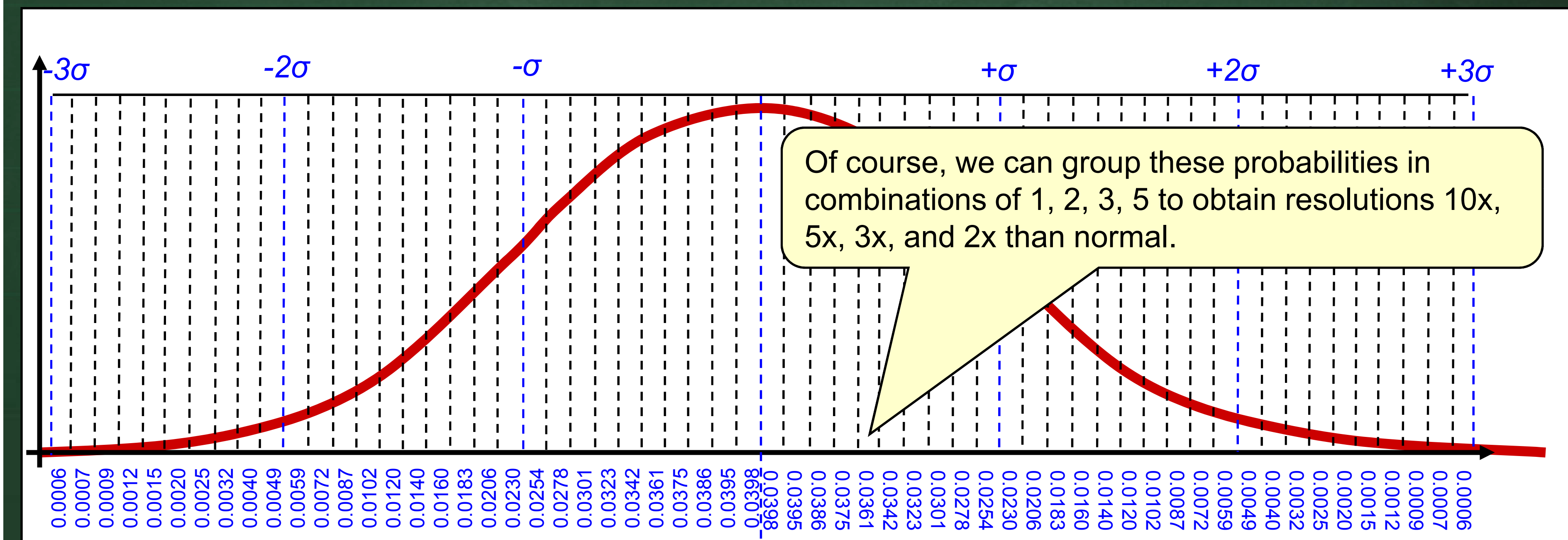
- In fact, we can increase resolution in our data if we *sub-divide* the 6-sigma areas.

- Can obtain areas of normal distribution curve from a statistics table:



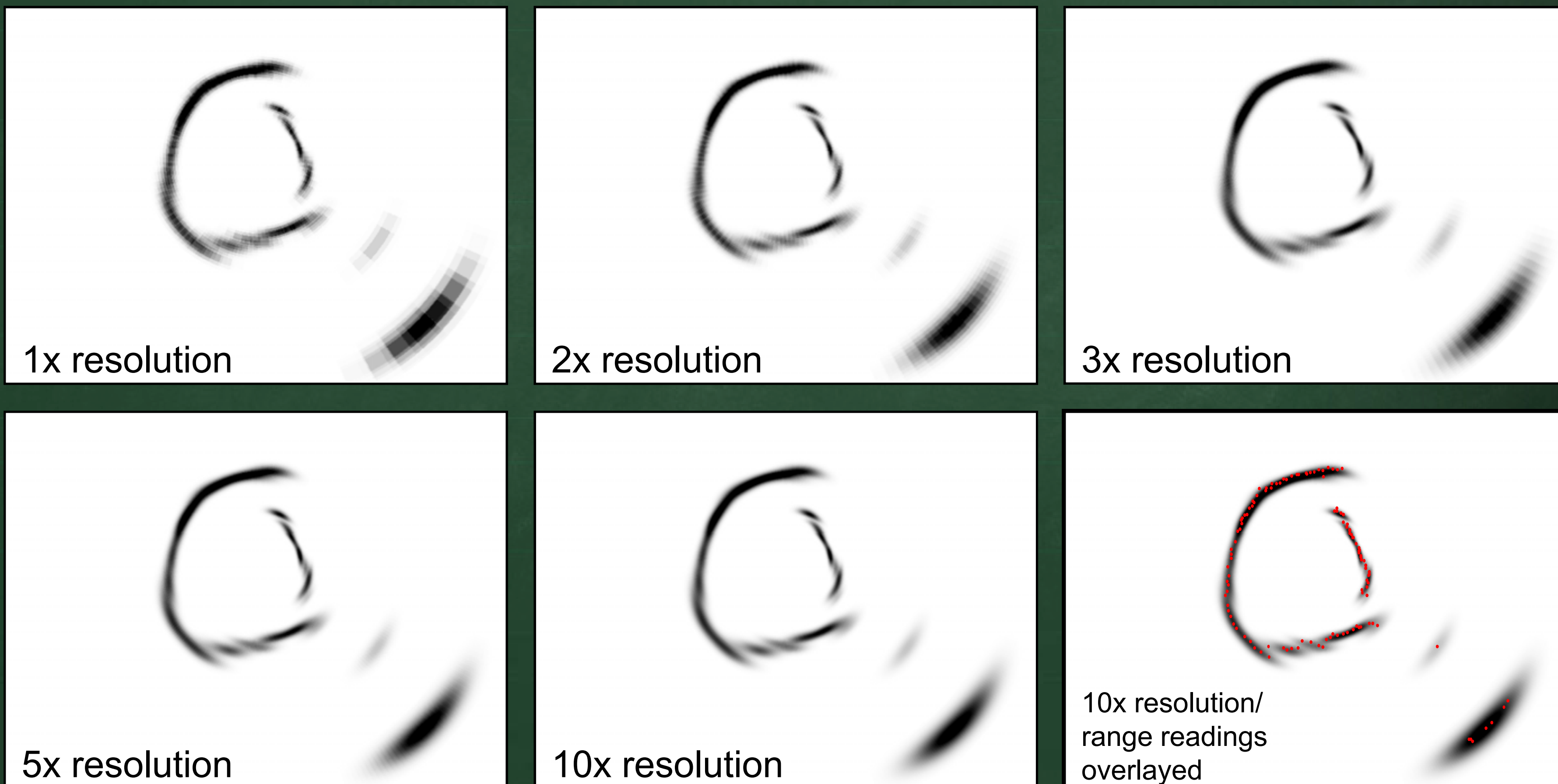
Applying Gaussian Distribution

- Can then subdivide into probability regions.
- With original 6-sigma, we had 6 probability regions.
- With this area data, we can have 60 regions:



Applying Gaussian Distribution

- Here are some of the various resolutions:



Applying Gaussian Distribution

- Pseudo code for processing data using 2x resolution:

distance error ± 10%

angular resolution = ± 19°

Probabilities for the 12 sigma sub-regions

Same size as occupancy grid

Compute number of samples to take along the arc

8.34 = 100/12 for 2x resolution of 6-sigma regions

This can be done in many ways. One way is to simply add the new probability values and then normalize everything later.

```
aRes = 19
dError = 0.10
SIGMA_PROBS = { 0.0049, 0.0166, 0.044, 0.0919, 0.1498, 0.1915, 0.1915, 0.1498, 0.0919, 0.044, 0.0166, 0.0049 }

FOR (each range reading d at pose p) DO {
    tempGrid = new grid

    FOR r = -d*dError TO d*dError BY 0.5 DO {
        distProb = SIGMA_PROBS [(r + d*dError) / (d*dError) / 2 * 100 / 8.34f]

        sX = (int)(p.x + COS(p.angle - aRes) * (d+r))
        sY = (int)(p.y + SIN(p.angle - aRes) * (d+r))
        dX = (int)(p.x + COS(p.angle + aRes) * (d+r))
        dY = (int)(p.y + SIN(p.angle + aRes) * (d+r))
        angularInterval = aRes / FLOOR(SQRT((sX-dX)*(sX-dX) + (sY-dY)*(sY-dY)))


        FOR a = -aRes TO aRes  BY angularInterval DO {
            objX = p.x + COS(p.angle + a) * (d+r)
            objY = p.y + SIN(p.angle + a) * (d+r)
            angProb = SIGMA_PROBS [(a + aRes) / (2*aRes) * 100 / 8.34f]
            tempGrid.setCell (objX,objY, angProb*distProb)
        }
    }
    merge tempGrid to fullGrid
}
```

Converting to Probabilities

- Until now, we have stored the likelihood of occupancy for each cell as a combined sum of the individual probabilities for each sensor reading.
- The result is not really a probability of the cell being occupied, it is biased towards positive occupancy readings.
- This does not allow us to distinguish cells that we know nothing about from cells we know that are likely not occupied.

Converting to Probabilities

- Given probabilities from 0.0 to 1.0, we can divide into three "groups" for occupancy:
  - > 0.5 → cell is occupied
  - = 0.5 → don't know if cell is occupied or empty
  - < 0.5 → cell is empty

- Adjust occupancy grid to maintain a probability for each cell, always in the range from 0 to 1.
  - Assume occupancy for each cell is initially unknown (i.e., have equal probability of being occupied or empty).
  - Change tempGrid.setCell(objX,objY, angProb*distProb) to tempGrid.setCell(objX,objY, angProb*distProb / 2.0 + 0.5)

Convert from 0-1.0 range to 0.5-1.0 range.

Converting to Probabilities

- As a result, the "white" areas become "medium gray", indicating uncertainty:

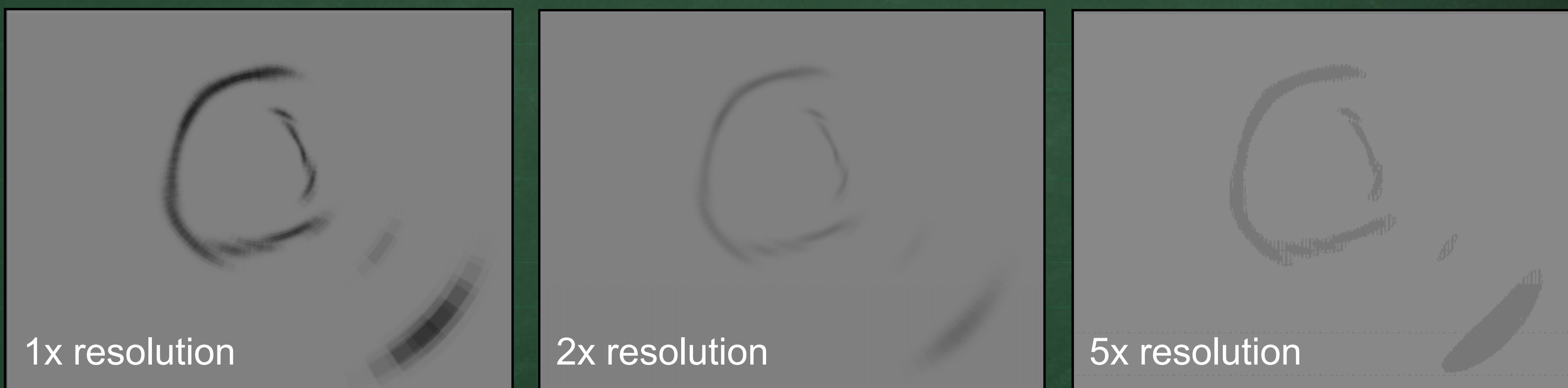
Unknown occupancy (i.e., probability of 0.5) appears as medium gray.

Before Normalized Probabilities

With Normalized Probabilities

Converting to Probabilities

- Be aware! The higher resolution sensor models attribute lower probabilities to each cell, so each reading has less of an influence on the occupancy:



- Must now consider how to combine probabilities from multiple sensor readings…

Bayesian Updating

- Can update cell values in various ways:
  - Bayesian
  - Dempster - Shafer
  - HIMM (Histogrammic in Motion Mapping)
- Most common used is *Bayesian*.
  - Essentially, it explains how we should change our existing beliefs in the light of new evidence.
  - e.g., when measuring the location of an obstacle, we take multiple readings and update the probability of an obstacle being at a certain location by updating our belief after each sensor reading.

Bayesian Updating

- *Bayes Rule* is as follows (assuming $p(R) > 0$):

$$p(L|R) = \frac{p(R|L)\, p(L)}{p(R)}$$

posterior probability distribution at location **L** given that sensor measurement **R** was just obtained.

probability value obtained during sensor reading **R** at grid location **L**.

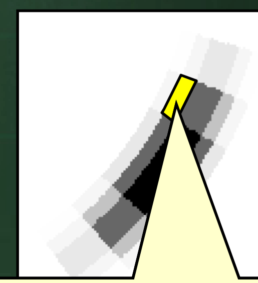
prior probability distribution at location **L** before reading **R** is incorporated (i.e., value at grid cell **L** before applying sensor reading.

p(R) is the probability that sensor reading **R** has occurred.

  - Here, $p(R) = p(R|L)p(L) + p(R|\bar{L})p(\bar{L})$

$$= p(R|L)p(L) + (1 - p(R|L))(1 - p(L))$$

- In our case, $p(R|\bar{L}) = 1 - p(R|L)$ since:
  - we only update cells affected by sensor reading
  - the Gaussian probabilities add to $0.9972 \approx 1$



Given that this cell is empty, the prob of sensor reading **R** is total of remaining probs in Gaussian distribution.

Bayesian Updating

- Can be re-stated in terms of each cell in the occupancy grid:

$$beliefCellOccupied = \frac{likelihoodCellOccupiedBasedOnReading * priorBeliefCellOccupied}{likelihoodThatReadingOccurred}$$

- Can be shortened to:

$$p(occupied|reading) = \frac{p(reading|occupied)\ p(occupied)}{p(reading|occupied)\ p(occupied) + p(reading|empty)\ p(empty)}$$

Bayesian Updating

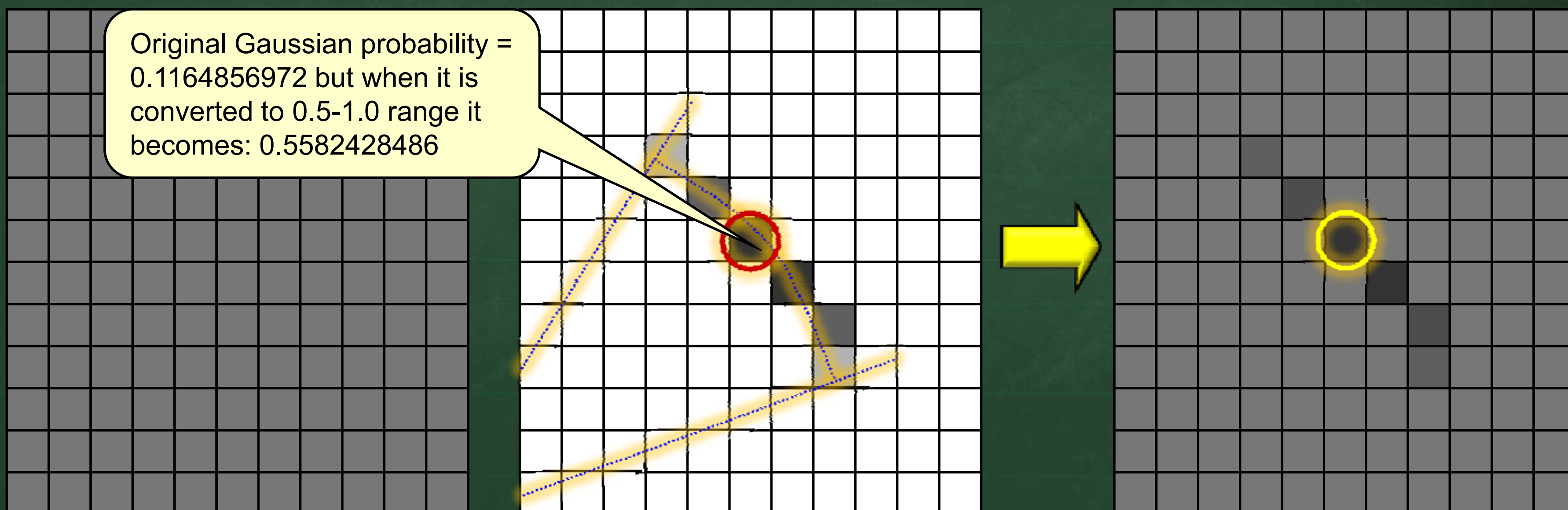
- How does this translate to code ?
  - It all takes place during the merging of the temporary and full grids from a single sensor reading:

```
FOR x = 0 TO WIDTH-1 DO {
   FOR y = 0 TO HEIGHT-1 DO {
      IF (tempGrid(x,y) ≠ 0.5) THEN {
         fullGrid(x,y) = (tempGrid(x,y) * fullGrid(x,y)) /
                         (tempGrid(x,y) * fullGrid(x,y) +
                         (1 - tempGrid(x,y))*(1 - fullGrid(x,y)))
      }
   }
}
```

Only merge cells that have been affected by this sensor reading.

Bayesian Updating

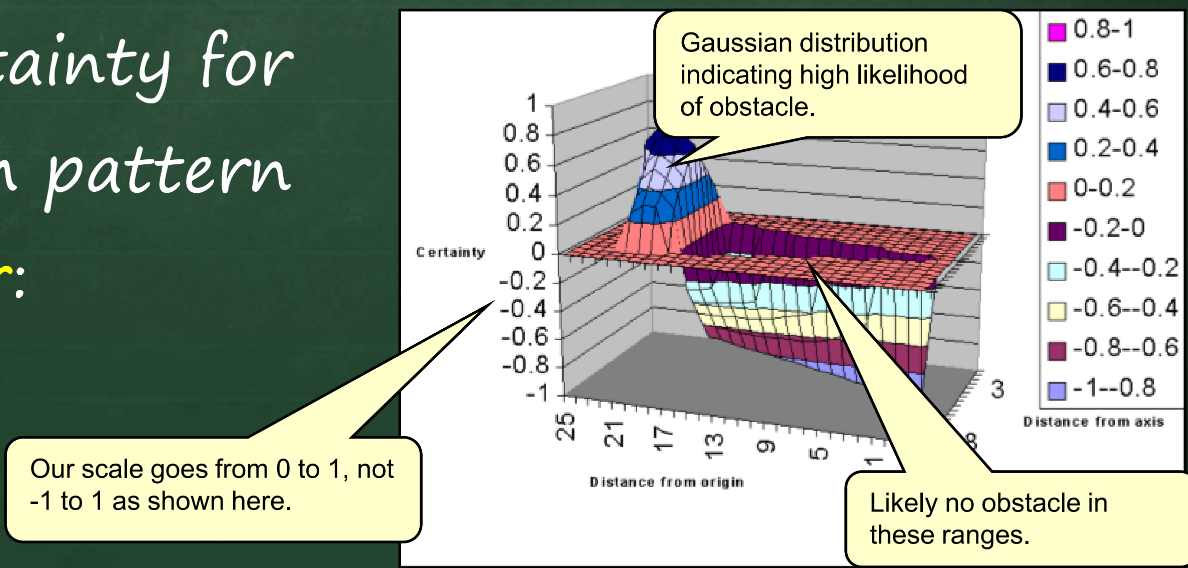
- For example:
  - assume initially unknown grid (all probabilities are 0.5)
  - sensor reading obtained with Gaussian distribution



p(occupied|reading) = $\dfrac{(0.5582428486 * 0.5)}{((0.5582428486 * 0.5) + (0.4417571514 * (1 - 0.5)))}$ = 0.5582428486

Improving the Sensor Model

- Recall that the sensor model applied a Gaussian distribution across the angle and distance.
- Our model should also consider that a particular range reading, *r*, implies that no obstacles are detected within beam pattern at distance < *r*.
- Must reduce certainty for cells within beam pattern with distance < *r*:

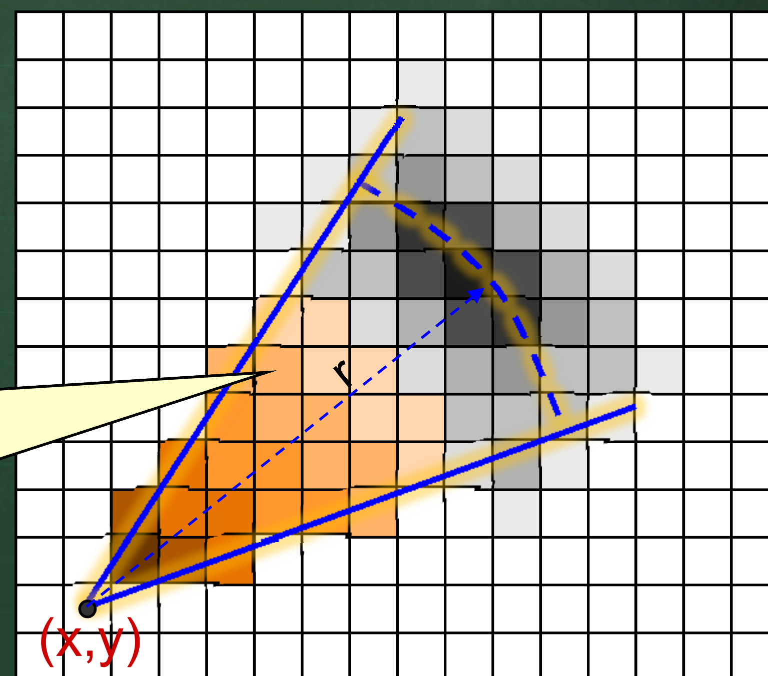


Improving the Sensor Model

- Now we can apply our improved sensor model in which cells within the beam have a high probability of being unoccupied for ranges preceding the sensor range reading:

There is a strong "negative" chance that there is NO obstacle in this area.

Stronger likelihood of no obstacle being present as distance to the source point decreases.



Improving the Sensor Model

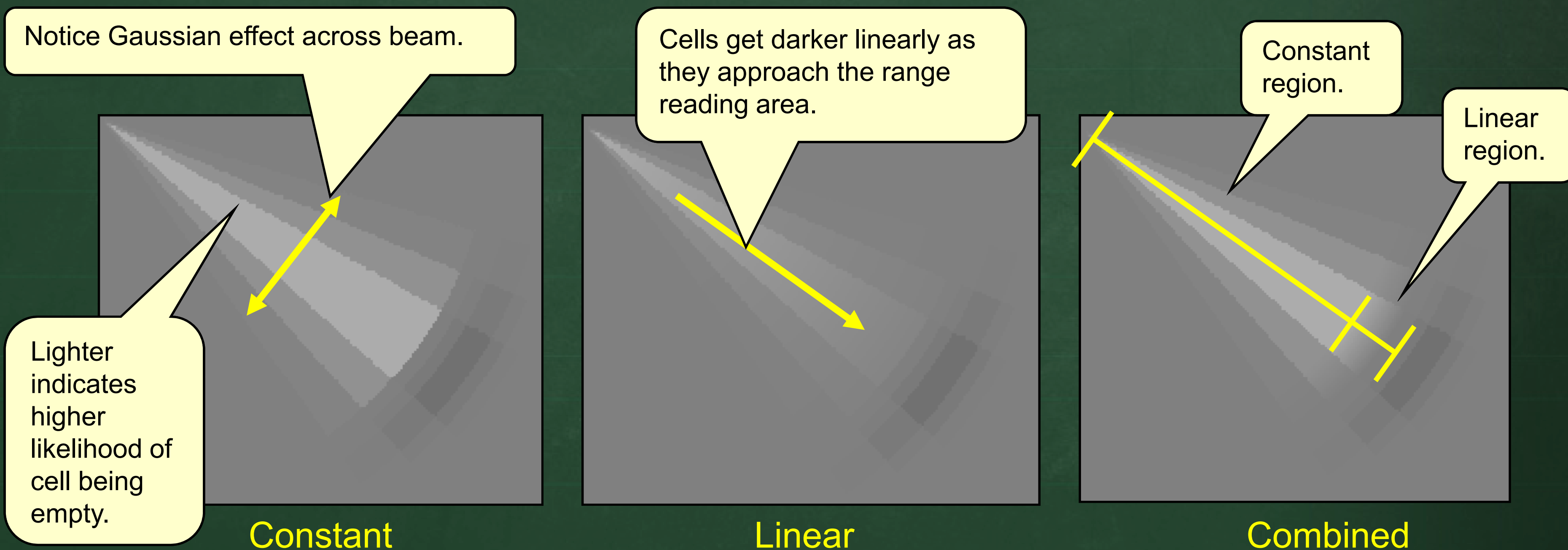
- We can modify the algorithm to “negate” the probabilities in these areas.
  - Simply make the changes in purple below to the algorithm (assuming probabilities are all added together)

```
…
FOR (each range reading d at pose p) DO {
    …
    FOR r = -d TO d*dError BY 0.5 DO {
        IF (r < -d*dError)
            distProb = -1.0;
        ELSE
            distProb = SIGMA_PROBS [(r + d*dError) / (d*dError) / 2 * 100 / 8.34f]

        …

        FOR a = -aRes TO aRes  BY angularInterval DO {
            …
            tempGrid.setCell (objX,objY, angProb*distProb / 2.0 + 0.5)
        }
    }
    merge tempGrid to fullGrid using Bayesian strategy
}
```

dError is the distance error which is 0.1 (i.e., ± 10%)

Improving the Sensor Model

- Here is the result from a single sensor reading by using a various models pertaining to emptiness:



Constant

IF (r < -**d***d**Error**)
distProb = -1.0

Linear

IF (r < -**d***d**Error**)
distProb = r / **d**

Combined

IF (r < -**d***2*d**Error**)
distProb = -1
ELSE IF (r < -**d***d**Error**)
distProb = **(r+d*dError) / (d*dError)**

Improving the Sensor Model

- Here are the comparisons of these models after multiple sensor readings:

Constant

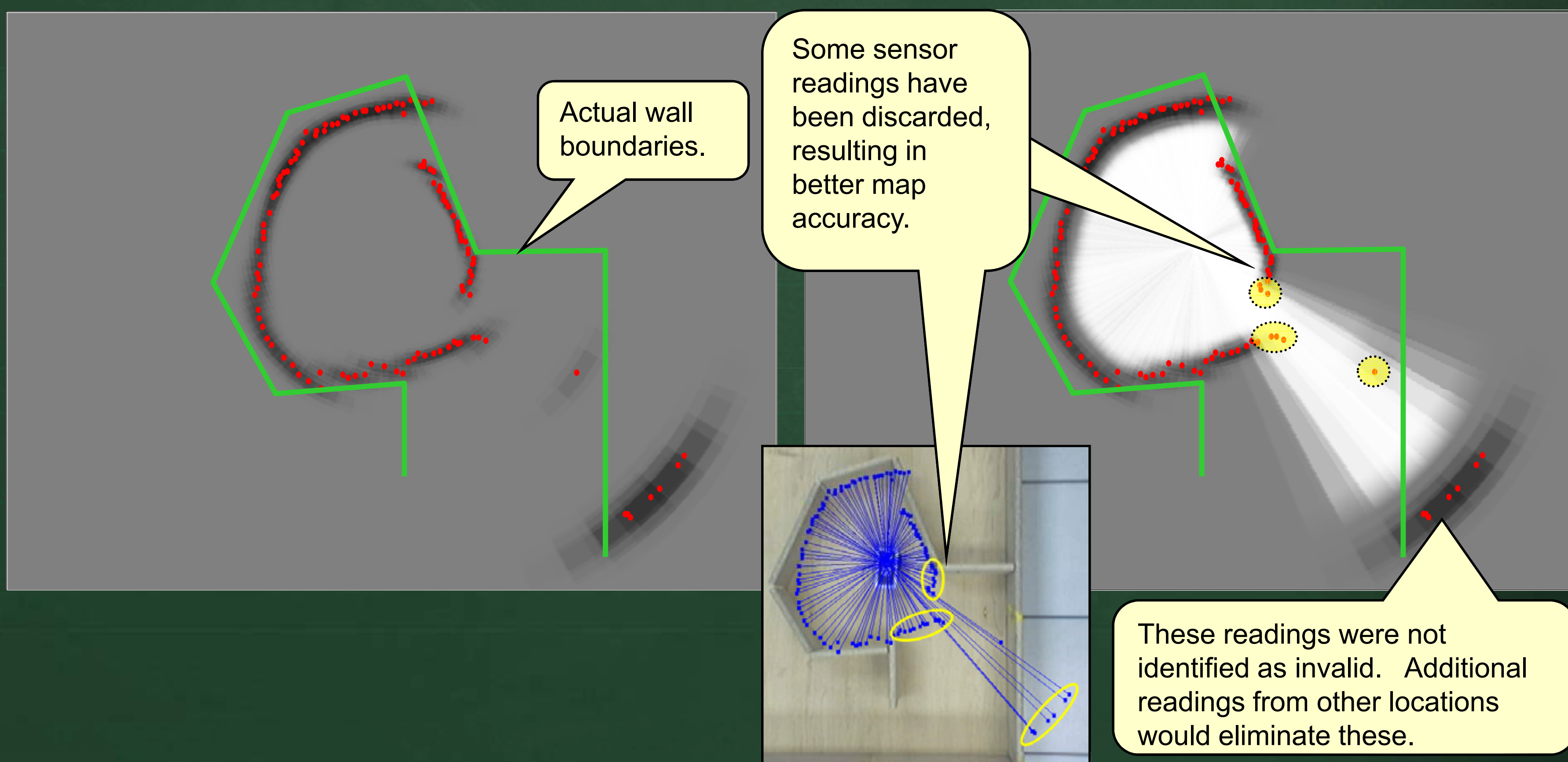
Linear

Combined

- The combined model allows for higher certainty regarding emptiness, and does so smoothly.

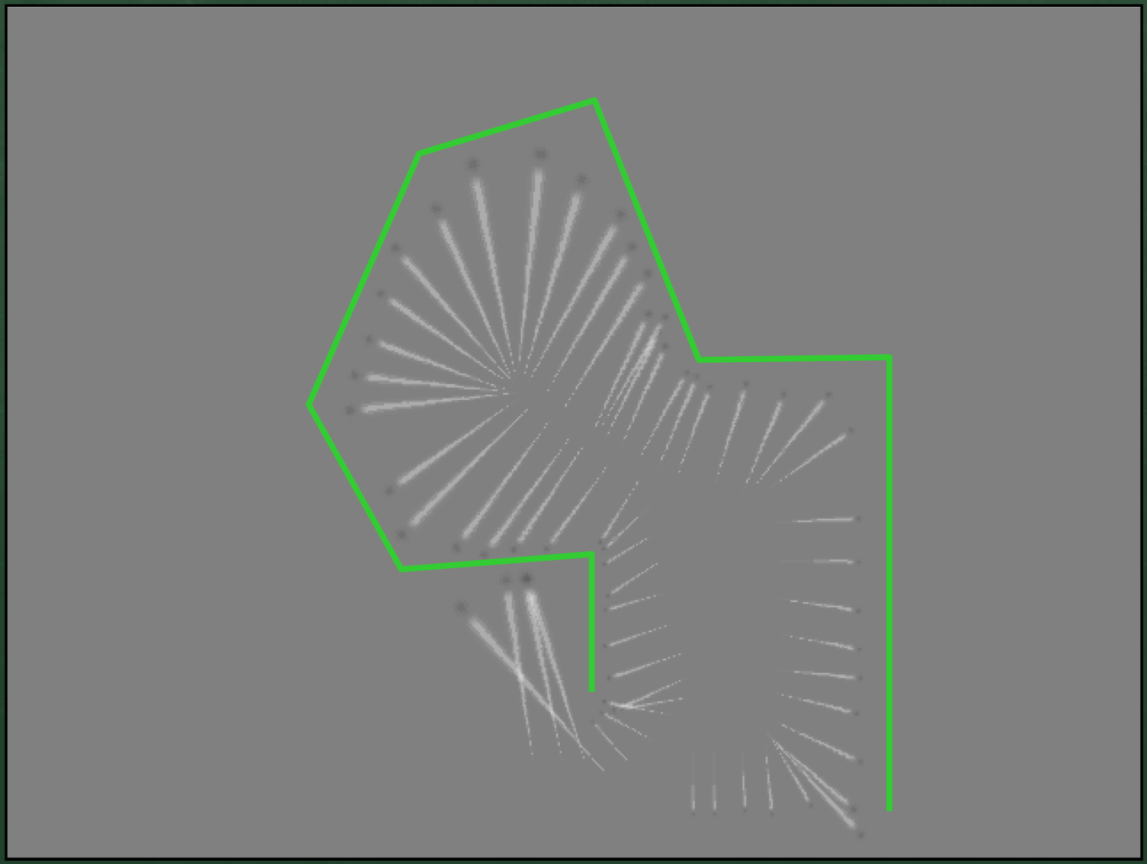
Improving the Sensor Model

- This sensor model improvement helps in identifying invalid and/or unlikely readings:



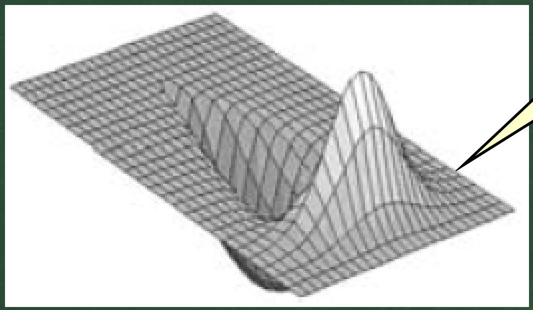
Improving the Sensor Model

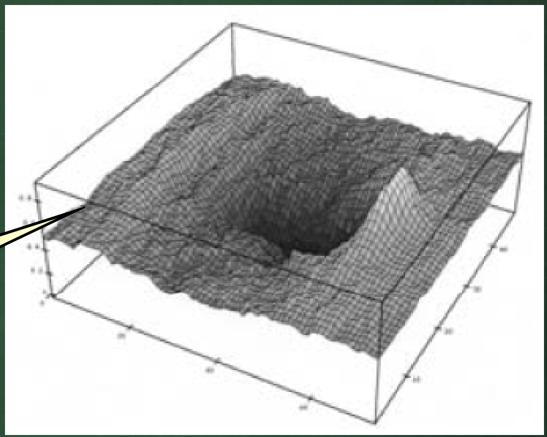
- We can similarly obtain the map corresponding to our IR sensor readings:



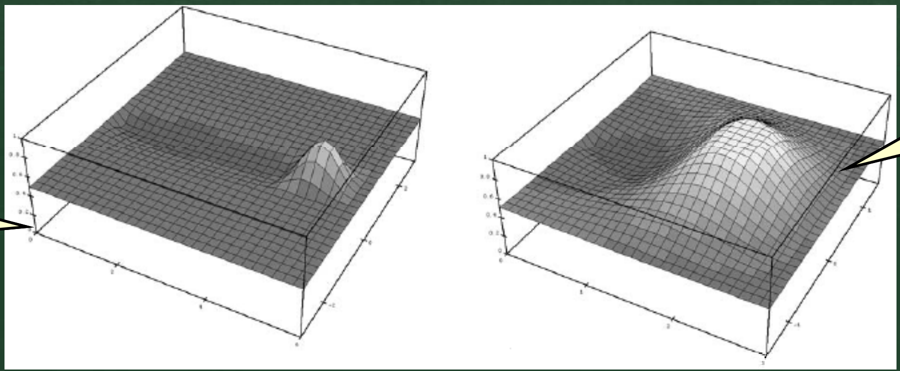
Learning the Sensor Model

- It is also possible to allow the sensor model to be
  - Dynamically computed:





  - Dynamically chosen:



Odds Representation

- Another popular way of maintaining the occupancy status for each cell is to store ***odds*** as opposed to probabilities.

$$odds(occupied \mid afterReading) = \frac{p(reading \mid occupied)}{p(reading \mid notOccupied)} \times odds(occupied \mid beforeReading)$$

- … in terms of coding for each cell [x][y] given Gaussian range probability r:

Advantage!!
Less calculations per cell….faster.

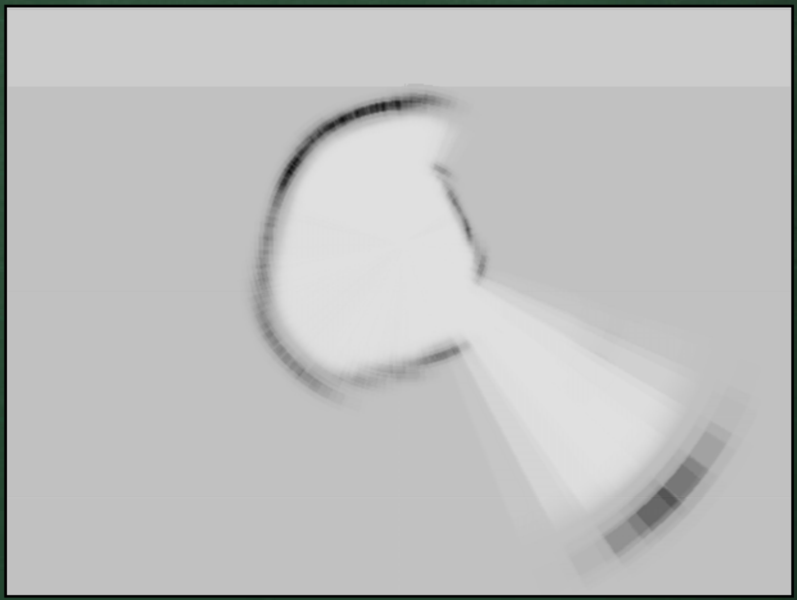
$$odds[x][y] = r \,/\, (1 - r) * odds[x][y]$$

- Odds representation ranges from 0 to ∞ instead of from 0 to 1.

Odds Representation

- As a result, our display strategy must be different…find maximum cell value first and distribute grey-scale values accordingly.

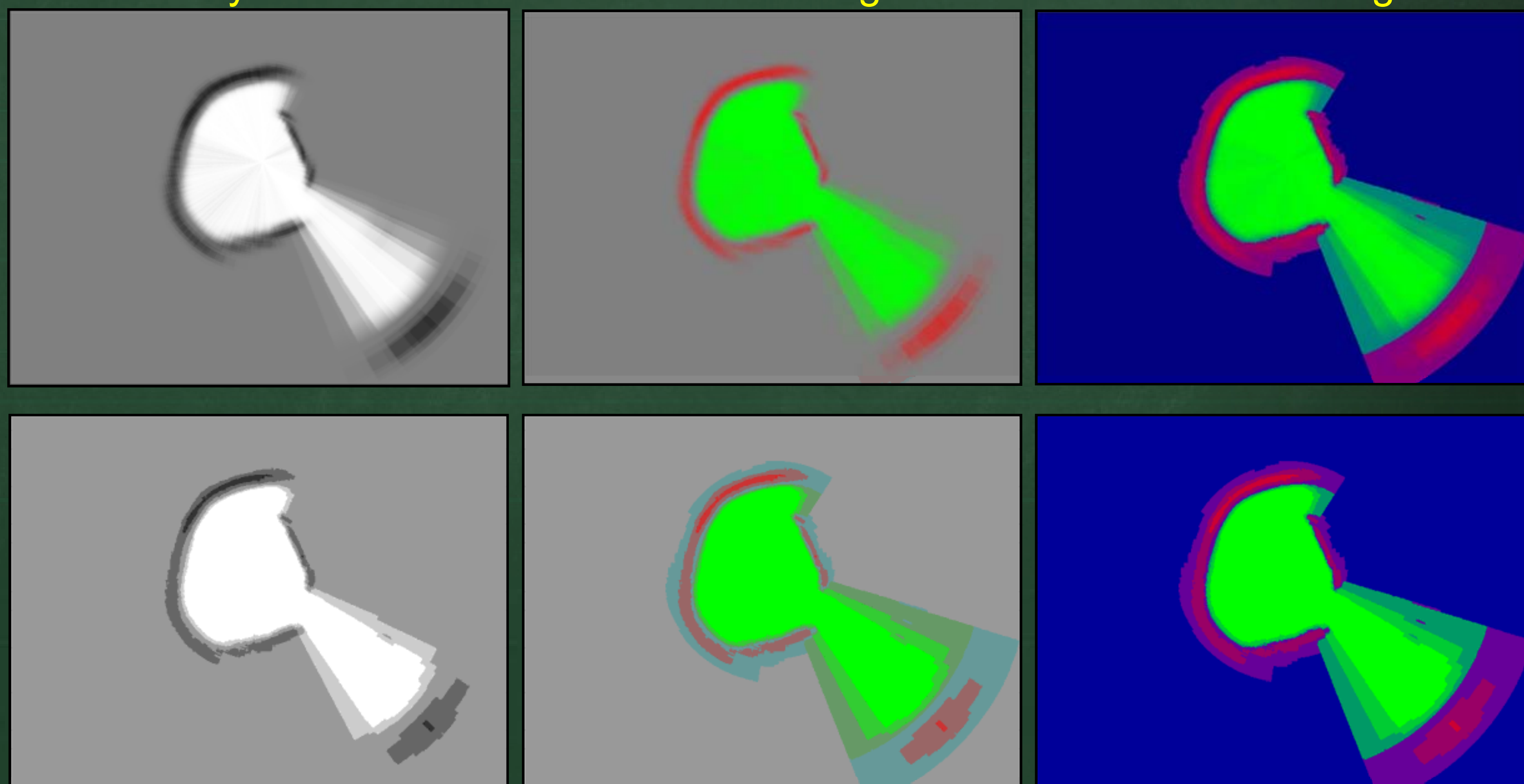
Probability Representation



Odds Representation

Display Options

- Of course, we can also manipulate thresholds and colors in various ways when displaying the map:



Gray-scale | 2-Color merge | 3-Color merge

Normal Color-Depth

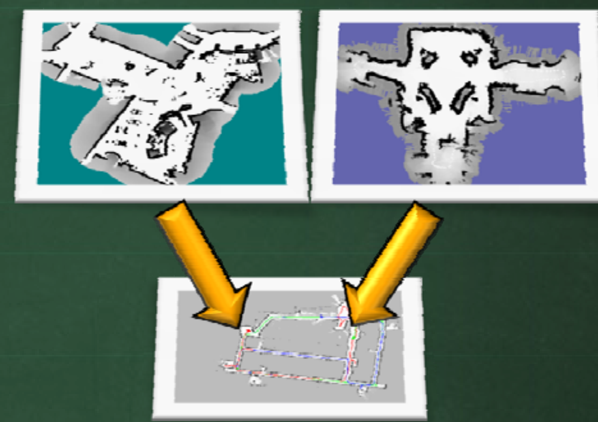
Reduced Color-Depth

Sensor Fusion

- Until now, we have only examined sensor fusion in regards to combining readings from a single sensor.
- If there are multiple sensors of the same type, it is easiest to combine them as if they were multiple readings from the same sensor.

- For different types of sensors, we may wish to incorporate a confidence level with each sensor.
  - some sensors are more accurate or reliable than others.
  - we may wish to give higher confidence to these

Sensor Fusion



- Sensor fusion is often done by
  - maintaining separate maps for different sensors over time and then combining the maps afterwards

Let $r_1, r_2, \ldots, r_n$ be n sensor readings from one sensor and $s_1, s_2, \ldots, s_m$ be m sensor readings from a different sensor

The probability that cell o is occupied after the readings can be stated as:

$$p(o \mid r_1, r_2, \ldots r_n, s_1, s_2, \ldots s_m) = p(o \mid r_1, r_2, \ldots r_n)\, \mathbf{w}(\mathbf{r}) + p(o \mid s_1, s_2, \ldots s_m)\, \mathbf{w}(\mathbf{s})$$

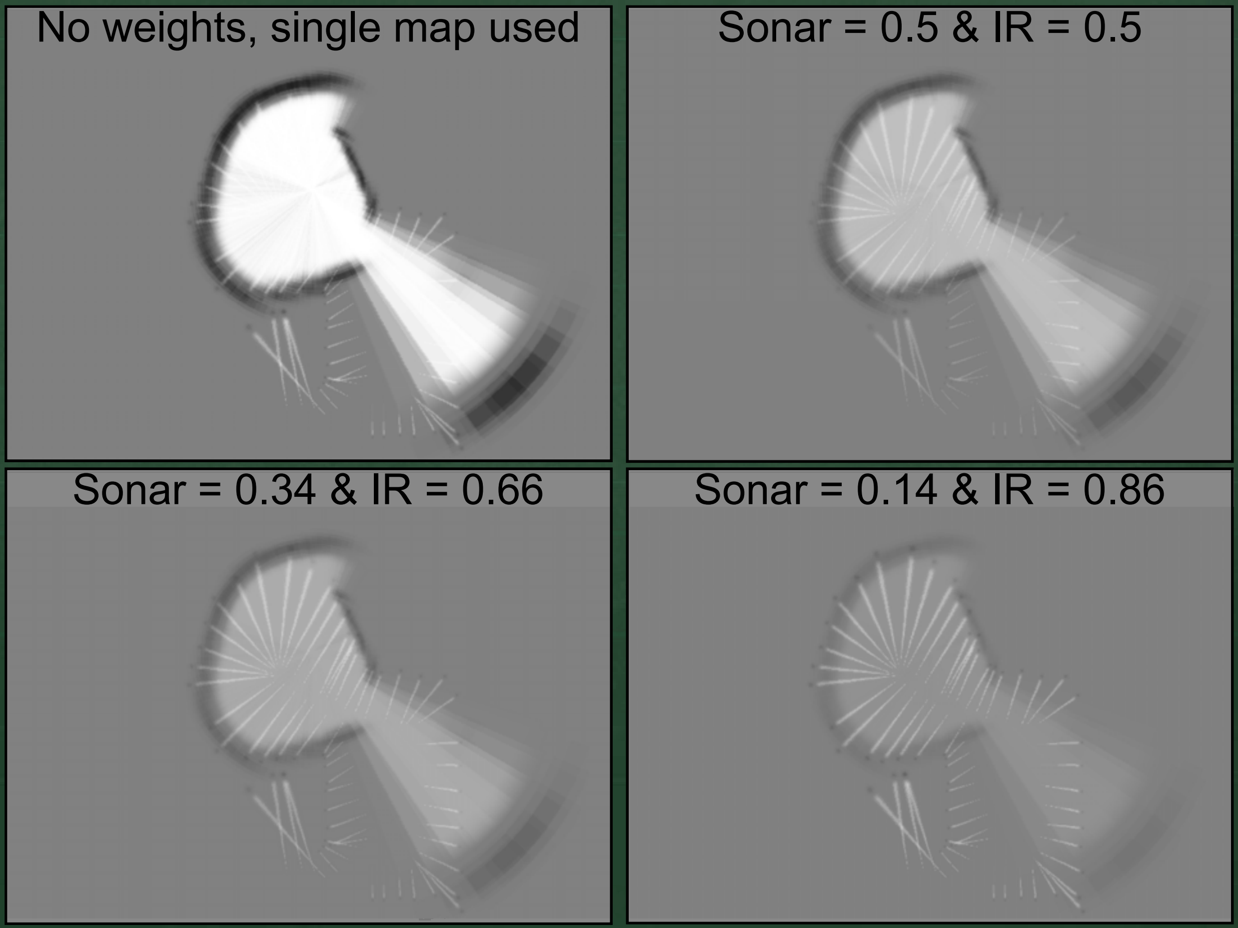
where $\mathbf{w}(\mathbf{r})$ and $\mathbf{w}(\mathbf{s})$ are the confidence weights of the sensors such that $\mathbf{w}(\mathbf{r}) + \mathbf{w}(\mathbf{s}) = 1$

Sensor Fusion

- We can choose w(r) and w(s) according to the trustworthiness of the sensors.
- E.g., we can choose weights according to:
  - Angular Resolution:
    - sonar with ±19° angular resolution may be roughly 1/6 as accurate as an IR with ±3° resolution. Thus, we might assign weights w(r) = 0.14 to the sonar and w(s) = 0.86 to the IR.
  - Distance Resolution:
    - sonar with ±10° distance resolution may be roughly 1/2 as accurate as an IR with ±5° resolution. Thus, we might assign weights w(r) = 0.34 to the sonar and w(s) = 0.66 to the IR.
  - Equal:
    - we can assign w(r) = w(s) = 0.5

Sensor Fusion

- Here are some fusion results:



Sensor Fusion

- Classifying (or categorizing) the data, we can see that some IR readings may be inhibited due to the false sonar readings (or vice versa):

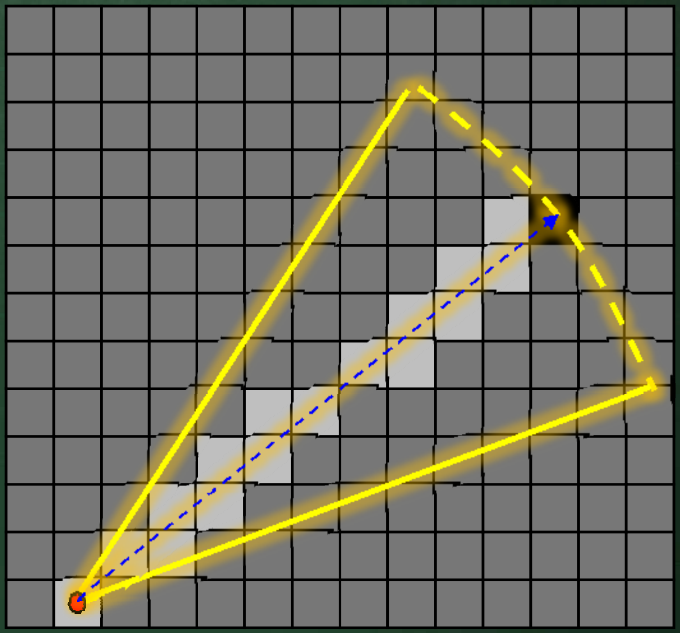
Other Updating Strategies

- The Dempster-Shafer method of updating cells
  - generalization of Bayesian strategy
  - provides similar performance to the Bayesian strategy
  - maintains p(occupied) + p(empty) + p(unknown) = 1.0
  - can distinguish conflicting evidence from lack of evidence

- We will not look any more into this.
- HIMM (Histogram In-Motion Mapping) method for updating cells:
  - faster than other two, but less accurate
  - meant for fast moving robot, performance can be poor when slow movements made

Other Updating Strategies

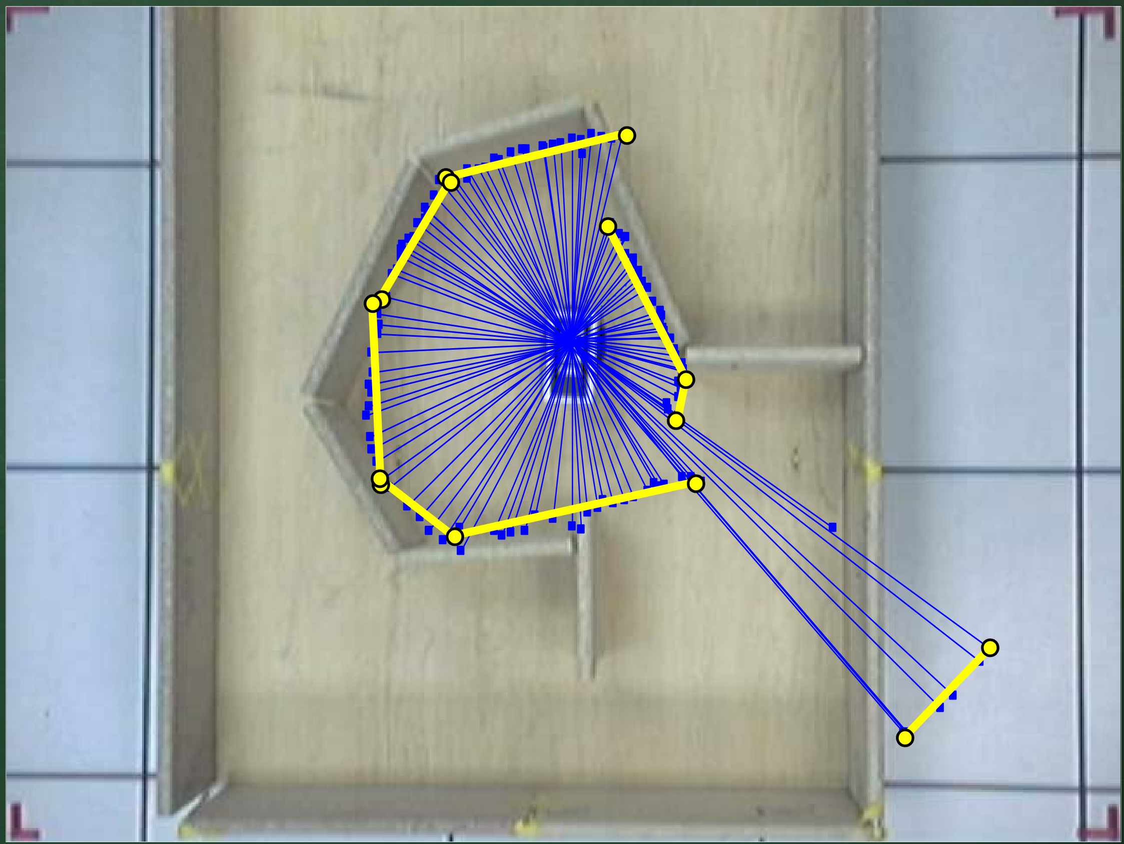
- HIMM updating rule:
  - if element is empty decrement its occupancy by 1
  - if element is occupied increment its occupancy by 3
- Believes that a region is occupied more than it believes that it is empty
- Same idea as counting
- Only updates along sonar axis beam (i.e., sonar is thin beam).
  - Will leave gaps since narrow beam reduces reading overlaps.



Extracting Obstacle Features From Raw Data

Line Extraction

- An alternative to using an occupancy grid:
  - extract the range readings as single points
  - combine/merge them to form a piecewise linear approximation (i.e., line segments) that represent the obstacle boundaries.



Line Extraction

- Advantages:
  - + less processing power
  - + quicker
  - + gives model of obstacles

- Disadvantages:
  - – data must come in sequence
  - – difficulty detecting and removing noise
  - – treats invalid readings as valid
  - – shapes are subjective to accuracy parameter which is difficult to choose.

Line Extraction

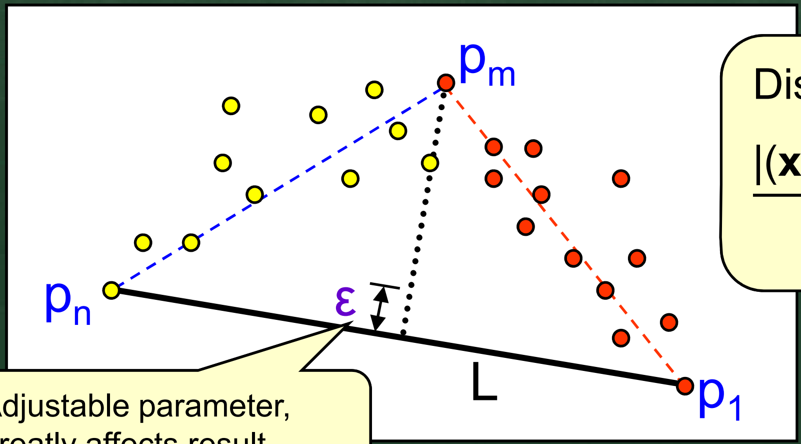
- Key issues:
  - How do we group range readings into line segments ?
  - How can we detect and eliminate noisy readings ?
- There are a variety of common techniques:
  - Split & Merge
  - Incremental
  - Line Regression
  - RANSAC (Random Sample Consensus)
  - Hough-Transform
  - EM (Expectation-Maximization)

Simplest and most popular.

Split & Merge

- Consider a set of range points:
  - $P = \{p_1, p_2, p_3, \ldots, p_n\}$ where $p_i = (x_i, y_i)$, $1 \le i \le n$
  - Let p_m be point of maximum distance from line $L = \overline{p_1 p_n}$
  - If distance from p_m to $L > \varepsilon$, split into two subsets:

$P' = \{p_1, p_2, p_3, \ldots, p_m\}$ $\qquad$ $P'' = \{p_m, p_{m+1}, p_{m+2}, \ldots, p_n\}$



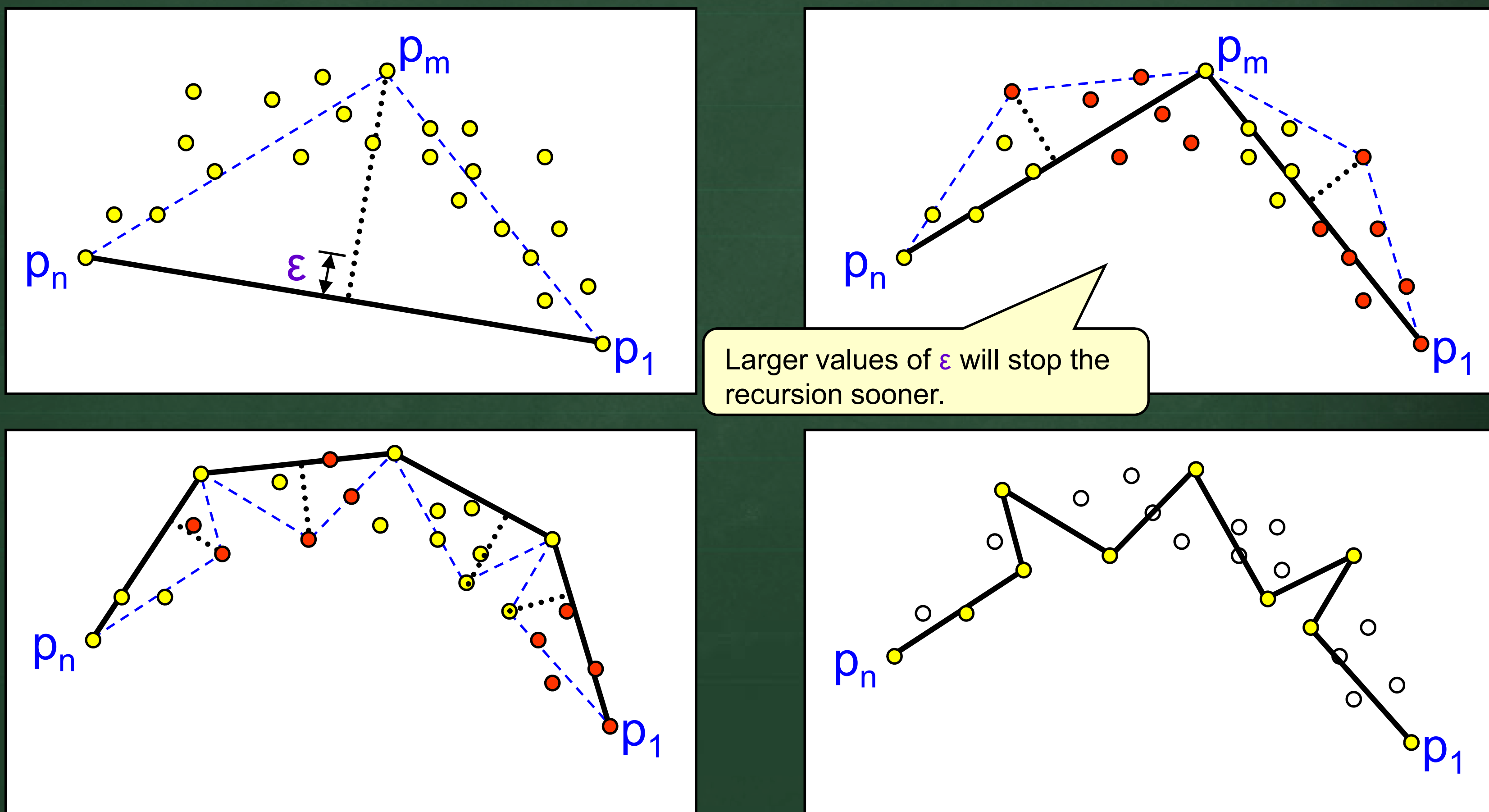
Adjustable parameter, greatly affects result.

Distance from $\mathbf{p_m}$ to $\mathbf{L}$ is:

$$\frac{|(x_n-x_1)(y_1-y_m)-(x_1-x_m)(y_n-y_1)|}{\sqrt{(x_n-x_1)^2+(y_n-y_1)^2}}$$

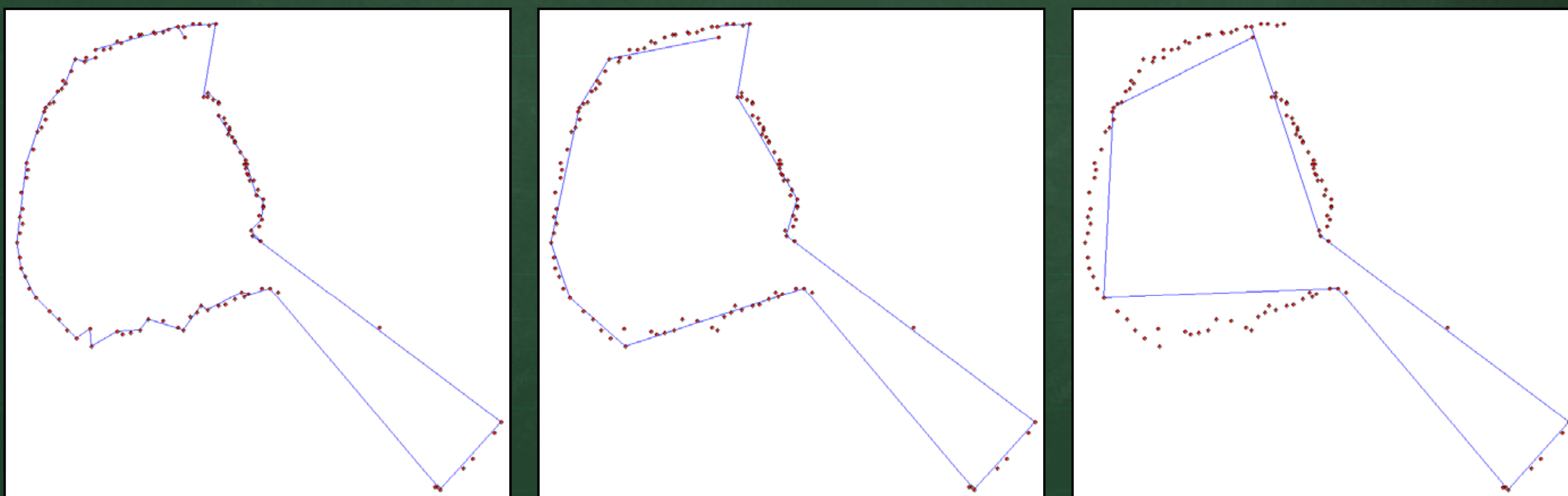
Split & Merge

- Do this recursively until no more splits can be made



Split & Merge

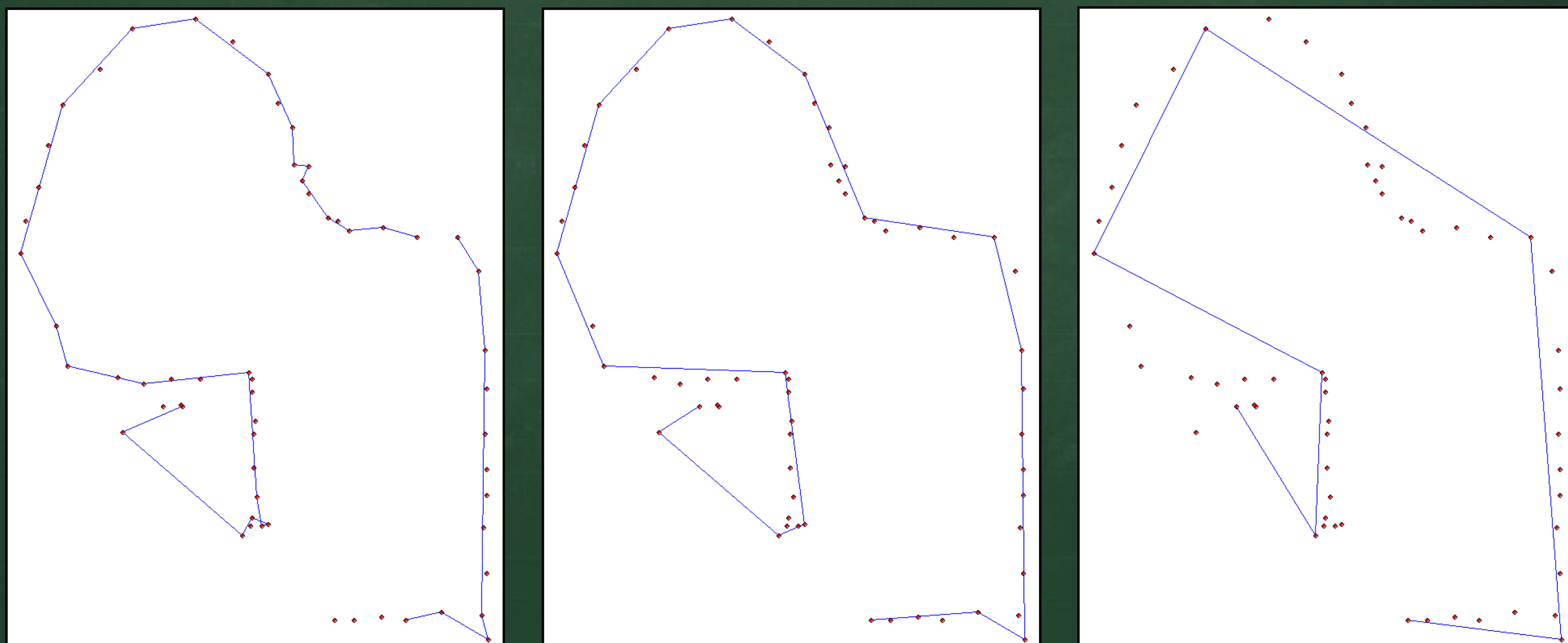
- Here are some results of the split phase on our sonar data for various thresholds:



ε ≈ 1 cm   ε ≈ 3 cm   ε ≈ 16 cm

Split & Merge

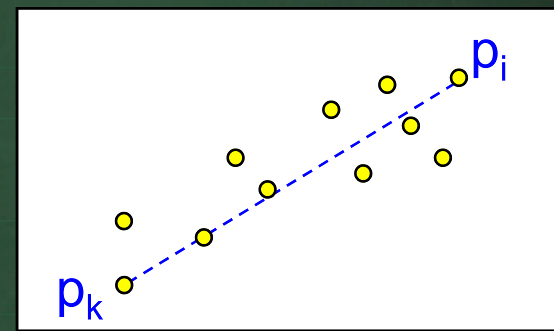
- Here are some results of the split phase on our IR data for various thresholds:

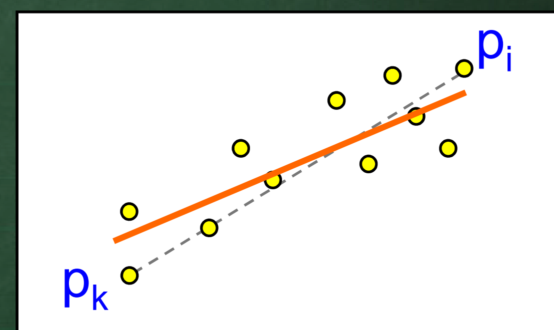


ε ≈ 1 cm   ε ≈ 3 cm   ε ≈ 16 cm

Split & Merge

- Consider a single line segment $\overline{p_i p_k}$ (i< k) obtained from our split phase.
- We would like to find a line that "best-fits" the data so as to not be biased towards the segment endpoints that were chosen during split phase.
- Can do this by "least squares method"





Split & Merge

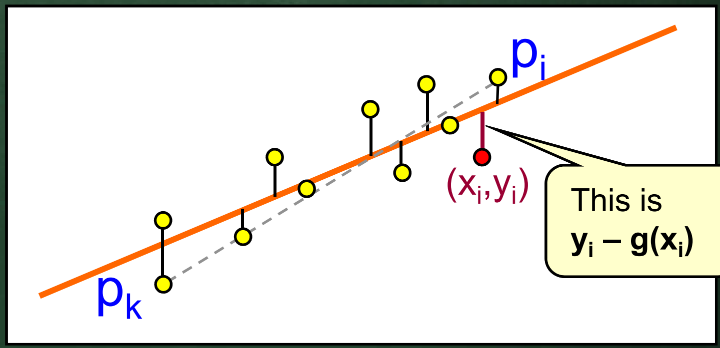
- Let $P = \{p_j, p_{j+1}, p_{j+2}, \ldots, p_k\}$ where $p_i = (x_i, y_i)$, $j \le i \le k$ be the set of points corresponding to a line segment chosen during the split phase.

- Let $g(x) = mx + b$ be a generic line equation

- Define the Mean Squared Error function in approximating the data as:

$$MSE = \frac{1}{k-j+1} \sum_{i=j}^{k} (y_i - g(x_i))^2$$

$$= \frac{1}{k-j+1} \sum_{i=j}^{k} (y_i - (mx_i+b))^2$$



- The "best-fit" line will minimize MSE.

Split & Merge

- After some joyful math work, we can obtain:

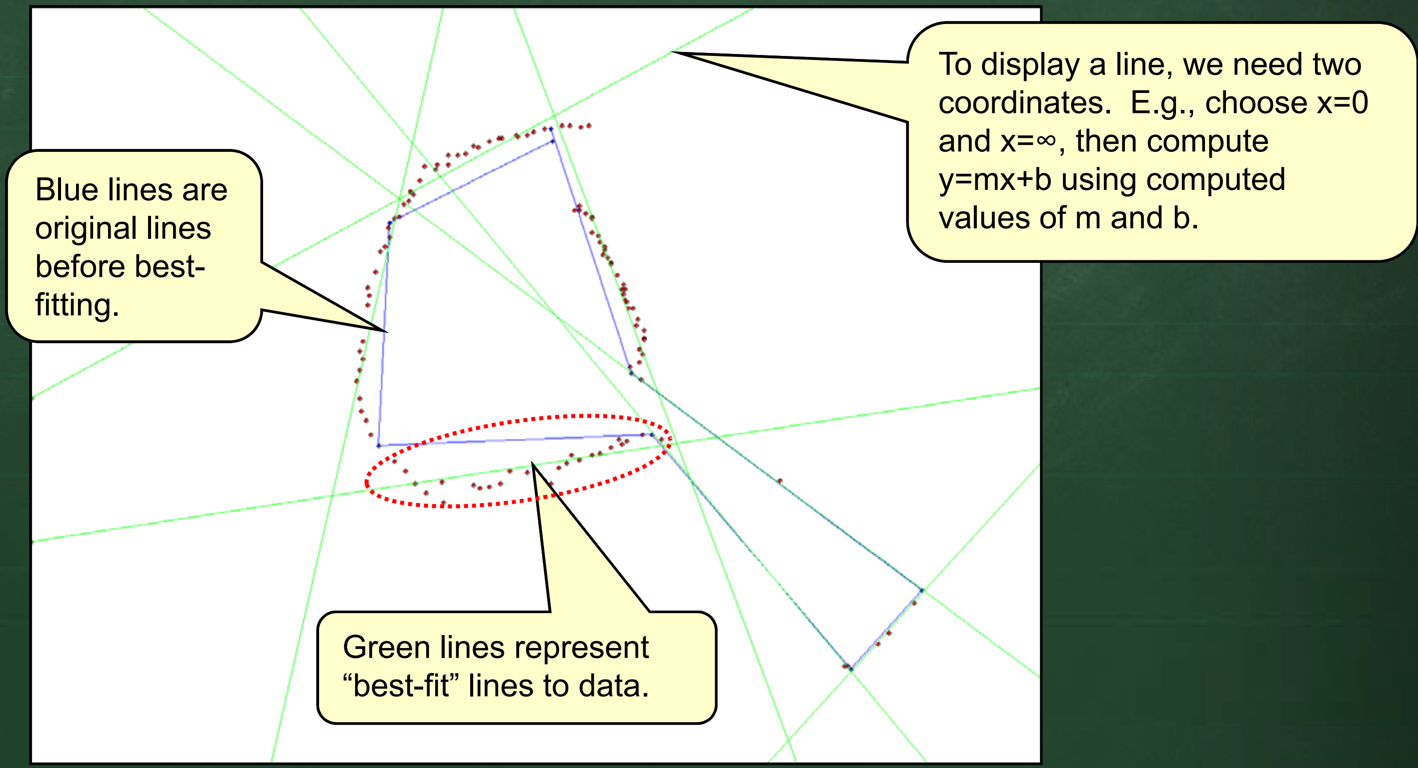
$$m = \frac{(\sum_{i=j}^{k} x_i)*(\sum_{i=j}^{k} y_i) - (k-j+1)*(\sum_{i=j}^{k} x_i y_i)}{(\sum_{i=j}^{k} x_i)^2 - (k-j+1)*(\sum_{i=j}^{k} x_i^2)}$$

$$b = \frac{(\sum_{i=j}^{k} x_i)*(\sum_{i=j}^{k} x_i y_i) - (\sum_{i=j}^{k} x_i^2)*(\sum_{i=j}^{k} y_i)}{(\sum_{i=j}^{k} x_i)^2 - (k-j+1)*(\sum_{i=j}^{k} x_i^2)}$$

- If we do this for each line segment, we end up with a set of intersecting lines

Split & Merge

- These line more closely fit the data.
- Next, determine intersections of adjacent lines.



Split & Merge

- Take two consecutive lines with equations:

$$y = m_1 x + b_1$$
$$y = m_2 x + b_2$$

- These intersect (provided that non-parallel) when:

$m_1 x + b_1 = m_2 x + b_2$ i.e., when $x = (b_2 - b_1) / (m_1 - m_2)$

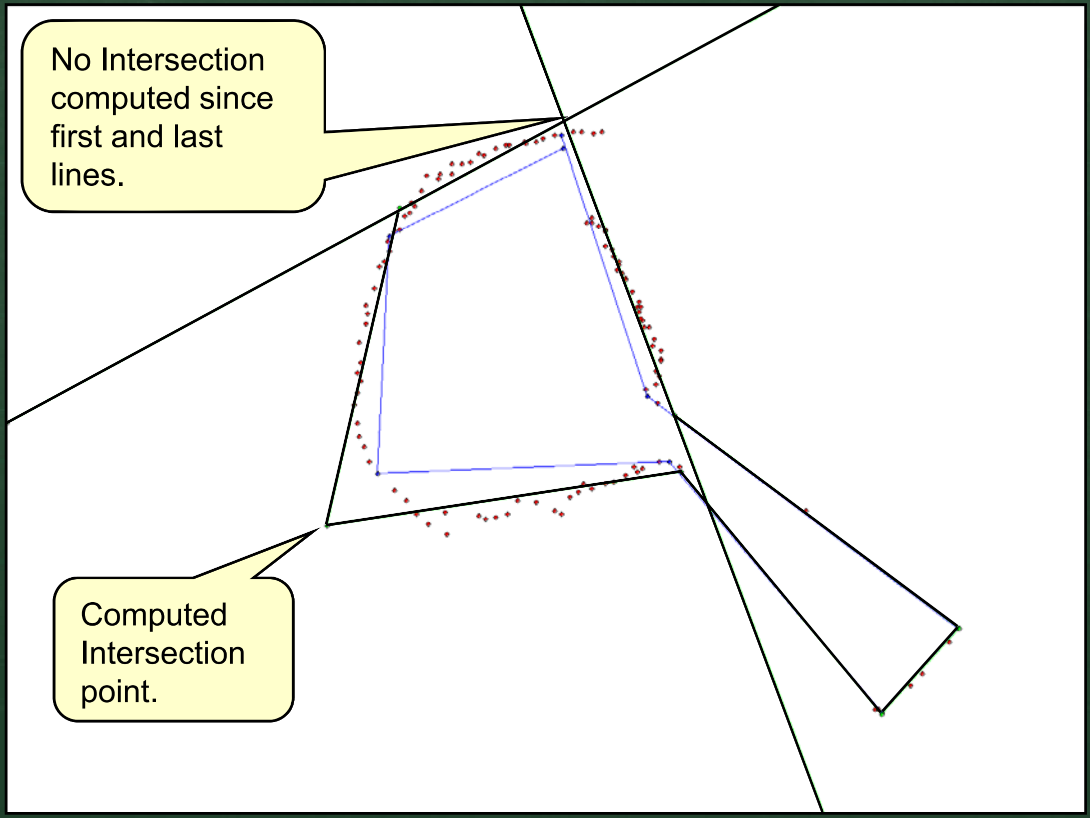
- So we do this for set of lines $L = \{L_1, L_2, L_3, \ldots, L_n\}$:

```
FOR i = 2 to N-1 DO {
      sx_i = (b_i - b_i-1) / (m_i-1 - m_i)
      sy_i = m_i*sx_i + b_i
      ex_i = (b_i+1 - b_i)/(m_i - m_i+1)
      ey_i = m_i*ex_i + b_i
}
```

Here $L_i = (sx_i, sy_i) \rightarrow (ex_i, ey_i)$

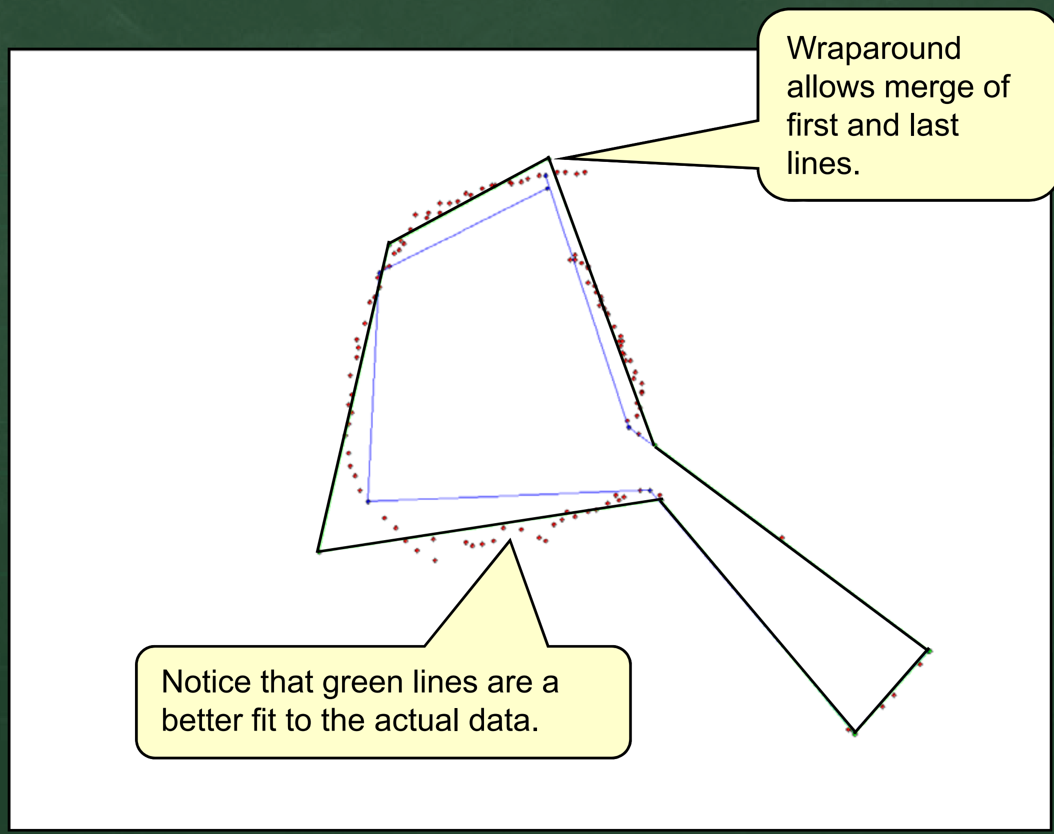
Split & Merge

- This will cover all adjacent lines except for the first and last lines:



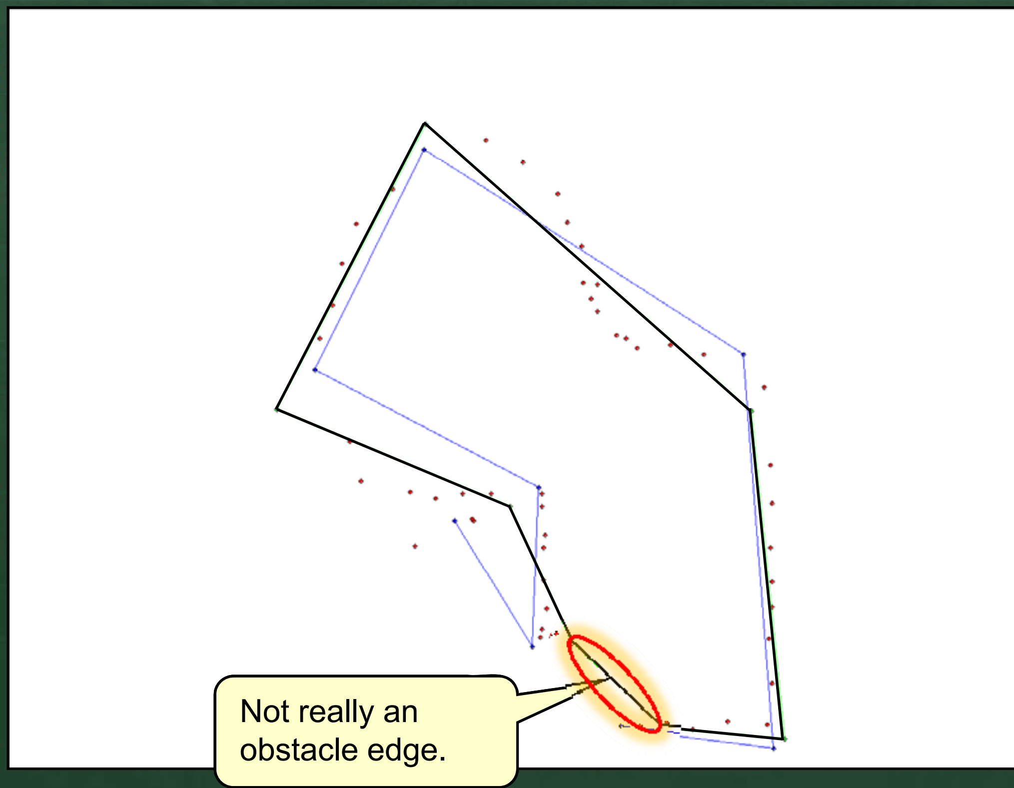
Split & Merge

- We can allow a wraparound by assuming that the first and last segments are adjacent:



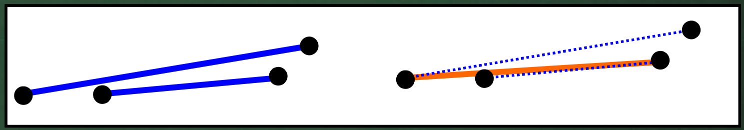
Split & Merge

- The wraparound may provide invalid line segments.
- It may be best to ignore first and last lines.

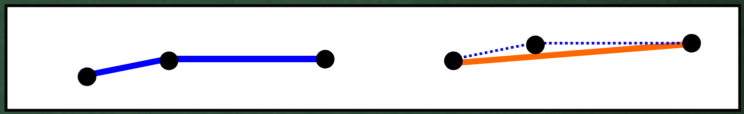


Split & Merge

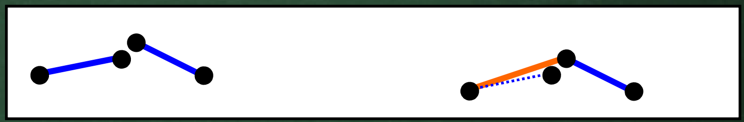
- Should also merge adjacent segments if:
  - they are nearly collinear



  - angle between them is small



  - they have very close endpoints

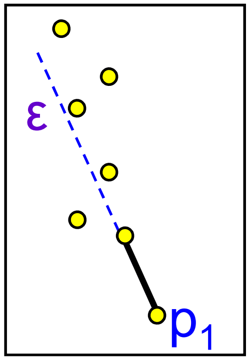


- Difficult to choose parameters in terms of when to split and when to merge:
  - usually chosen experimentally
  - will not perform the best in all cases

Incremental

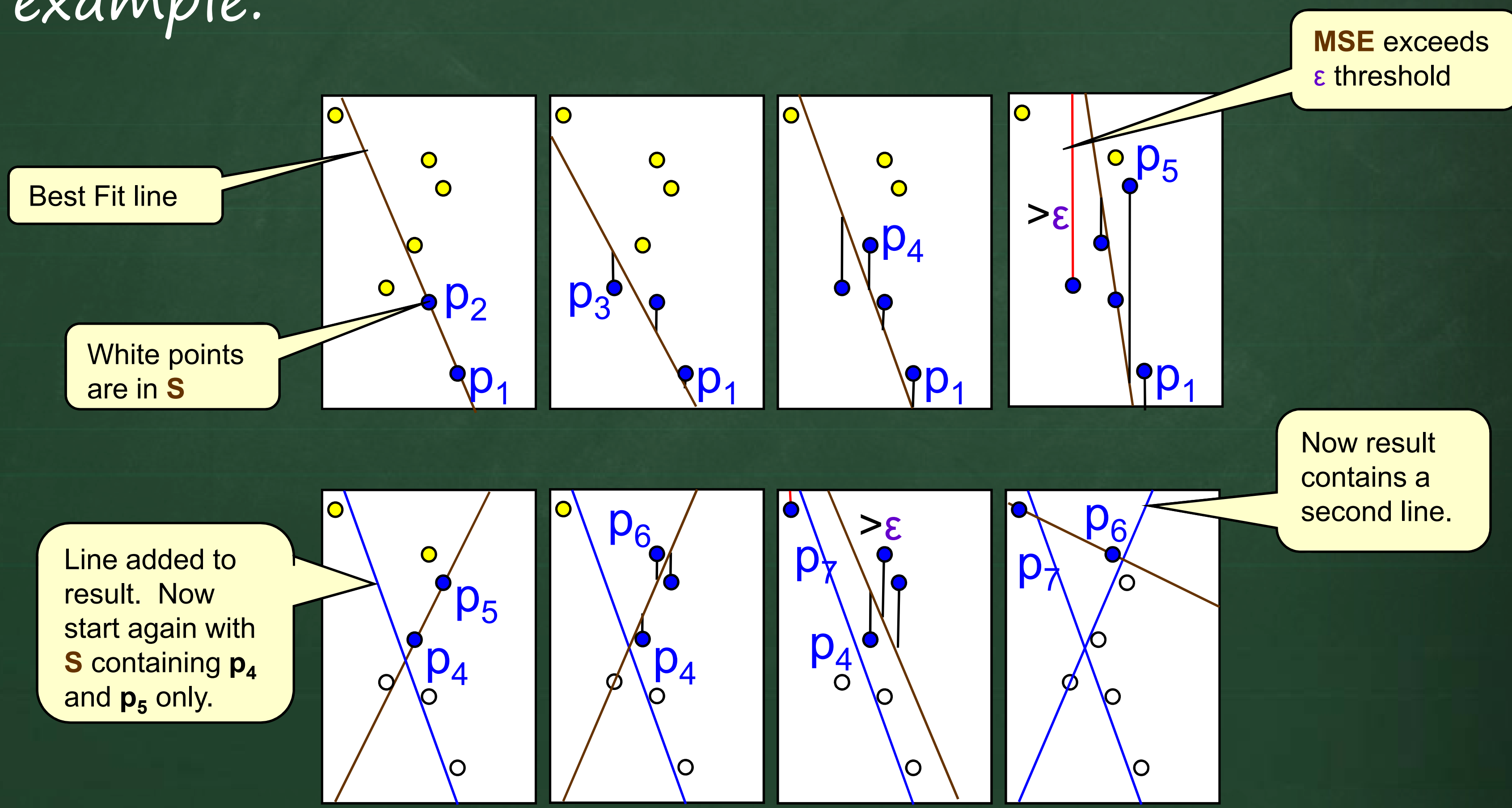
- The incremental algorithm for determining the lines is VERY similar and somewhat easier.

1. Start with a set S containing p_1
2. Add the next point p_i to S
3. Compute the best-fit line **L** for all points in S
   (choose x=0 and x=∞ to get two coordinates)
4. Determine the MSE for the points in S
   (use the best-fit line for comparison)
5. IF (MSE > ε)
   re-compute **L** with all points in S except p_i
   add **L** to the result and reset S to contain only p_{i-1} and p_i
6. IF i = size of point set THEN quit
7. Set $i = i+1$ and go back to 2



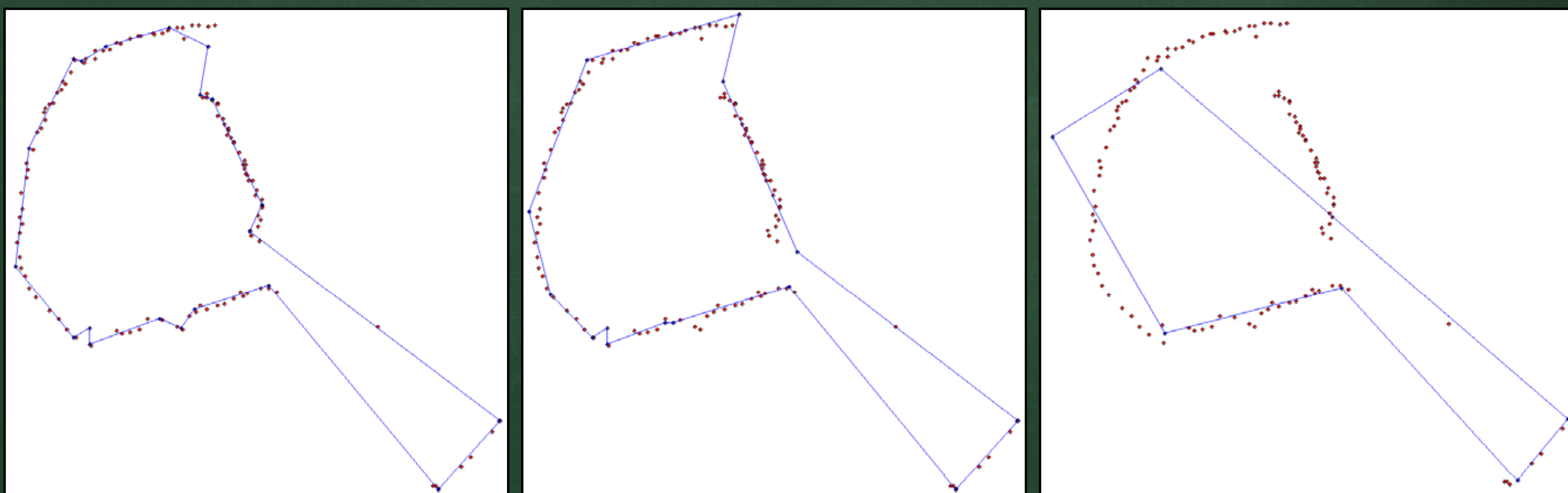
Incremental

- Here is how the algorithm works for a simple example:



Incremental

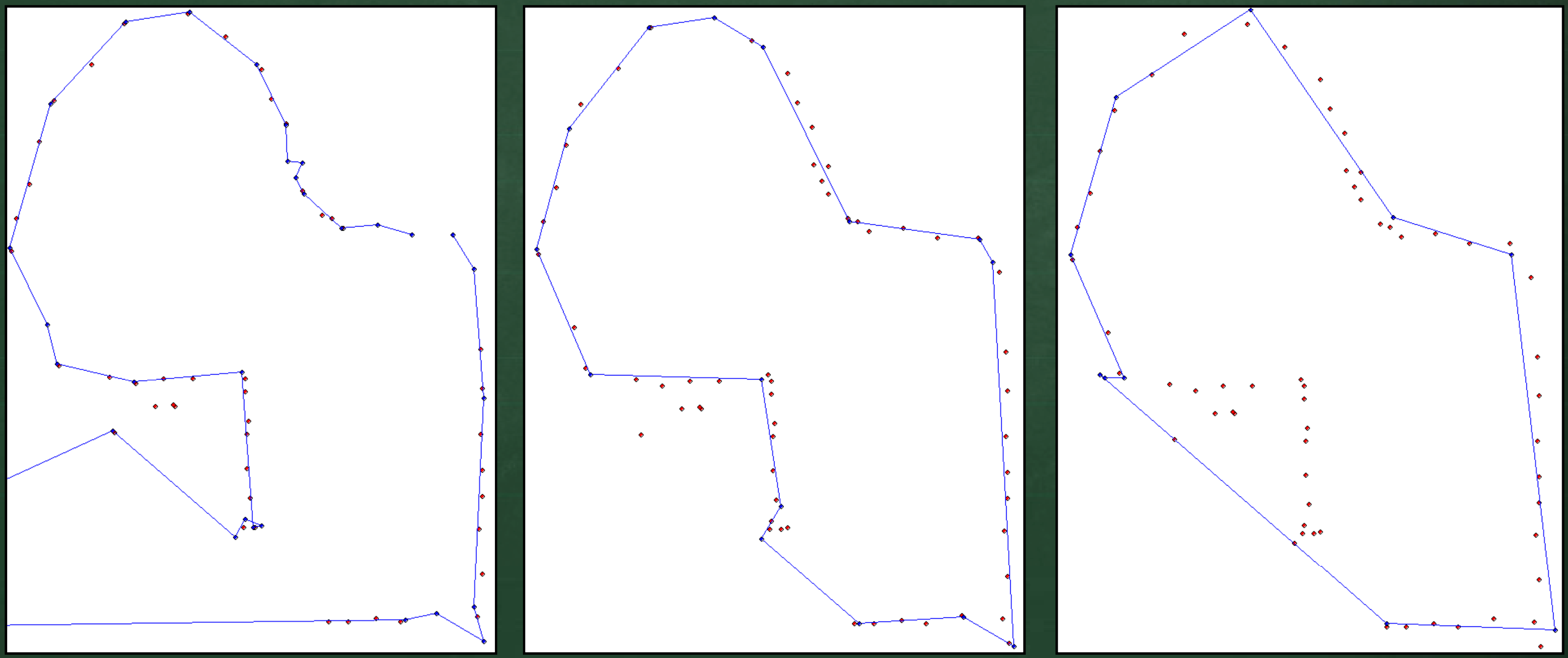
- Here are some results of the incremental algorithm on our sonar data for various thresholds:



ε ≈ 1.5 cm   ε ≈ 3 cm   ε ≈ 9 cm

Incremental

- Here are some results of the incremental algorithm on our IR data for various thresholds:



ε ≈ 3 cm   ε ≈ 9 cm   ε ≈ 18 cm

Other Schemes

- As mentioned, there are other techniques to compute these line segments.
- Both Incremental and Split&Merge are the most popular due to their simplicity and speed.
- Note that it is always necessary to determine the thresholds by examining data.
- Also, the technique only works when the data is ordered.
  - If data is given unordered, it must somehow be sorted first.

Extracting Obstacles Features from Occupancy Grids

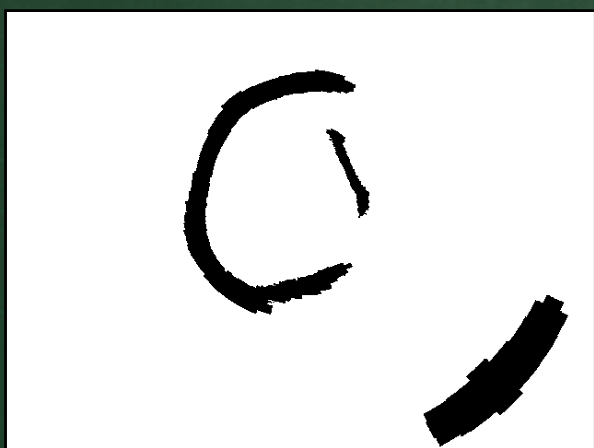
Extracting Features

- We just discussed how to extract features from raw sonar data.
  - this did not take into account errors in the data since it did not use any sensor models.
  - assumed that data was ordered along obstacle boundaries
- Using a certainty grid in place of raw data, we can take into account errors in the data:

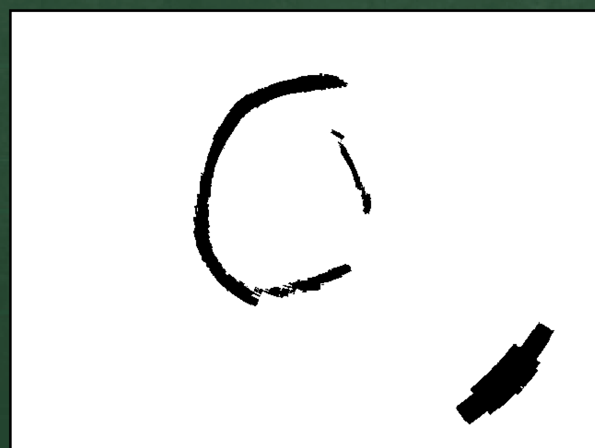
Thresholding

- Can apply image analysis techniques to extract important details.

- Robot must ultimately assume occupancy or not:
  - we must first obtain a binary grid by applying a threshold:



> 0.51 → 1
≤ 0.51 →0

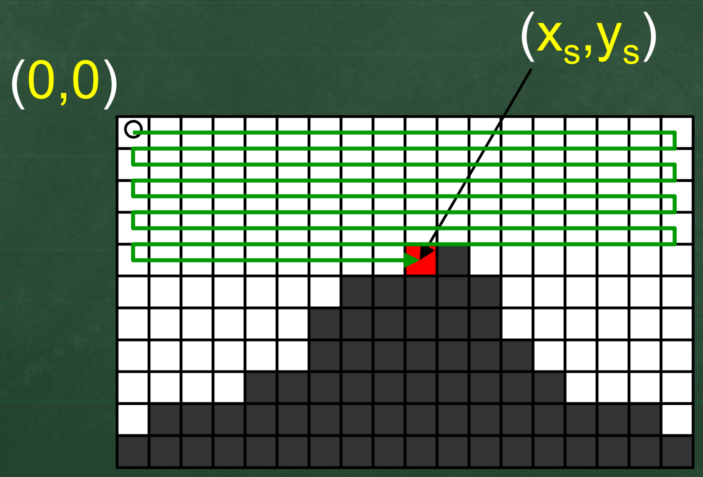


> 0.55 → 1
≤ 0.55 →0

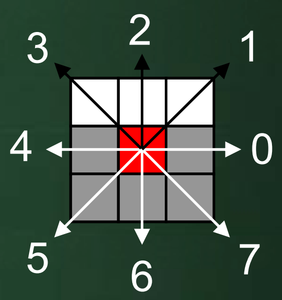
> 0.60 → 1
≤ 0.60 →0

Border Detection

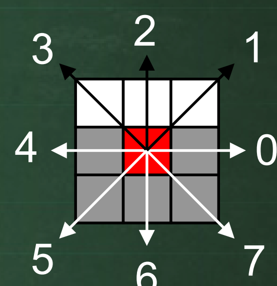
- We can then do a border detection on the resulting binary data as follows:
  - search the data from top left corner, scanning rows of pixels until the first black pixel is found at location (x_s,y_s)



  - define directions as follows around a pixel:



Border Detection



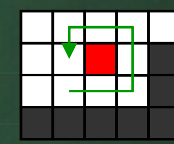
- Start with a direction d = 7 for pixel (x_s,y_s)
- Trace out border of obstacle by doing a traversal from the current point $(x_c,y_c) = (x_s,y_s)$:

1. Let the next position to be examined be d' where
   $d' = (d+7) \bmod 8$ if d is even, otherwise $d' = (d+6) \bmod 8$
2. Starting at position d', check pixels around (x_c,y_c) in counter-clockwise order (i.e., keep incrementing d') until a black pixel is found. Here are some scenarios:

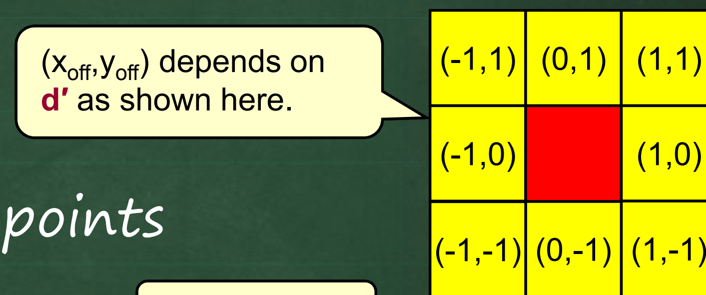
Border Detection

3. If there is no black pixel found, then (x_c,y_c) must be a single pixel in the image.

   (It should probably be ignored.)



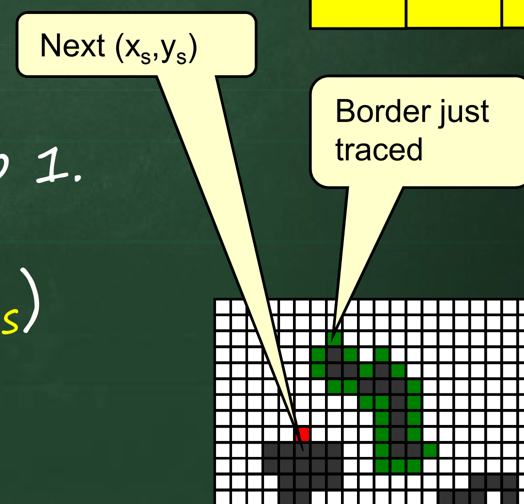
None found



4. Set $(x_c,y_c) = (x_c + x_{off},\ y_c + y_{off})$

5. Add point (x_c,y_c) to set of border points

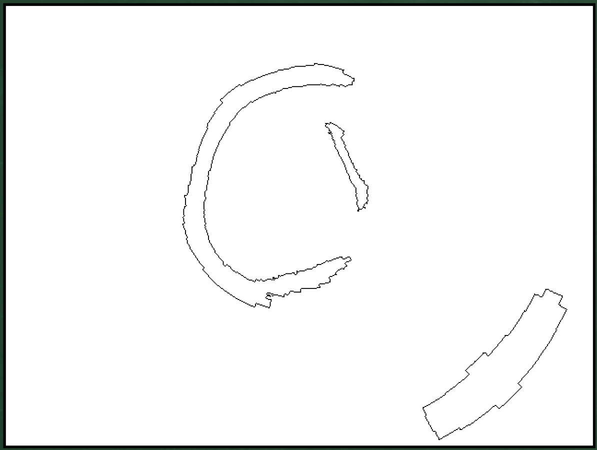
6. If $(x_c,y_c) == (x_s,y_s)$ then we are done otherwise set **d**=**d'** and go back to step 1.



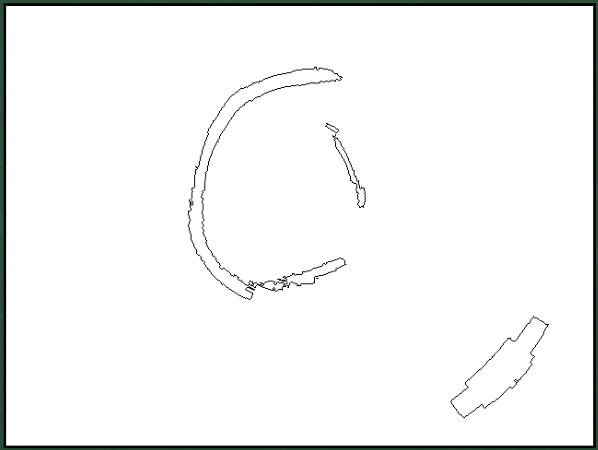
- Repeat by scanning for another (x_s,y_s)
  - must be black and not already a border
  - must have a white pixel on its left

Border Detection

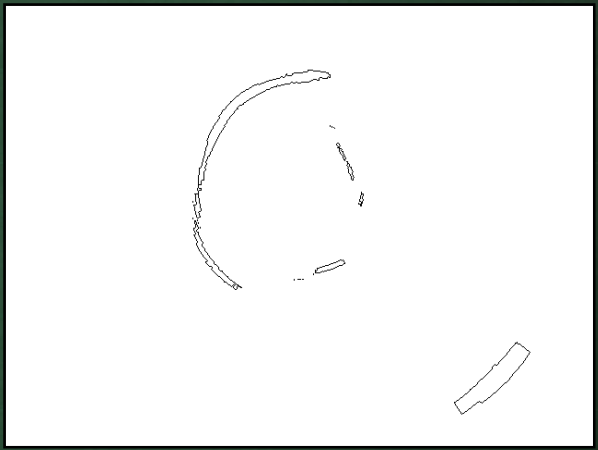
- Here are the results of doing border detection on our thresholded sonar data:



0.51 threshold
5 borders detected



0.55 threshold
14 borders detected

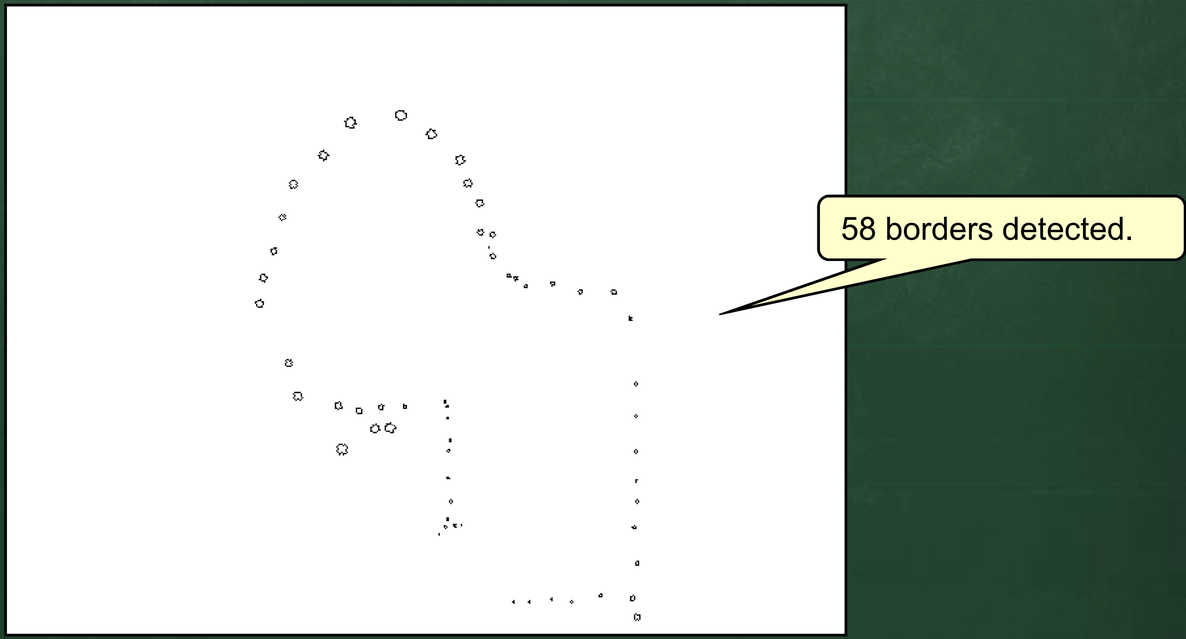


0.60 threshold
22 borders detected

- Notice that the number of borders (or obstacles) detected includes small (insignificant ones as well).

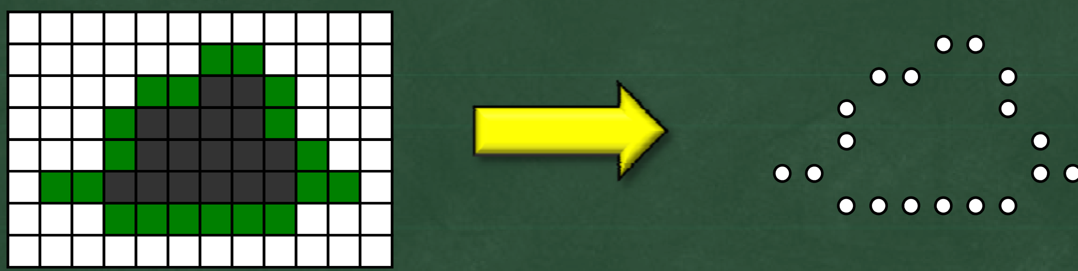
Border Detection

- Of course, the data must be significantly connected in order for the border detection to provide useful information.
- Here is the border detection result on the IR data:



Fitting Lines

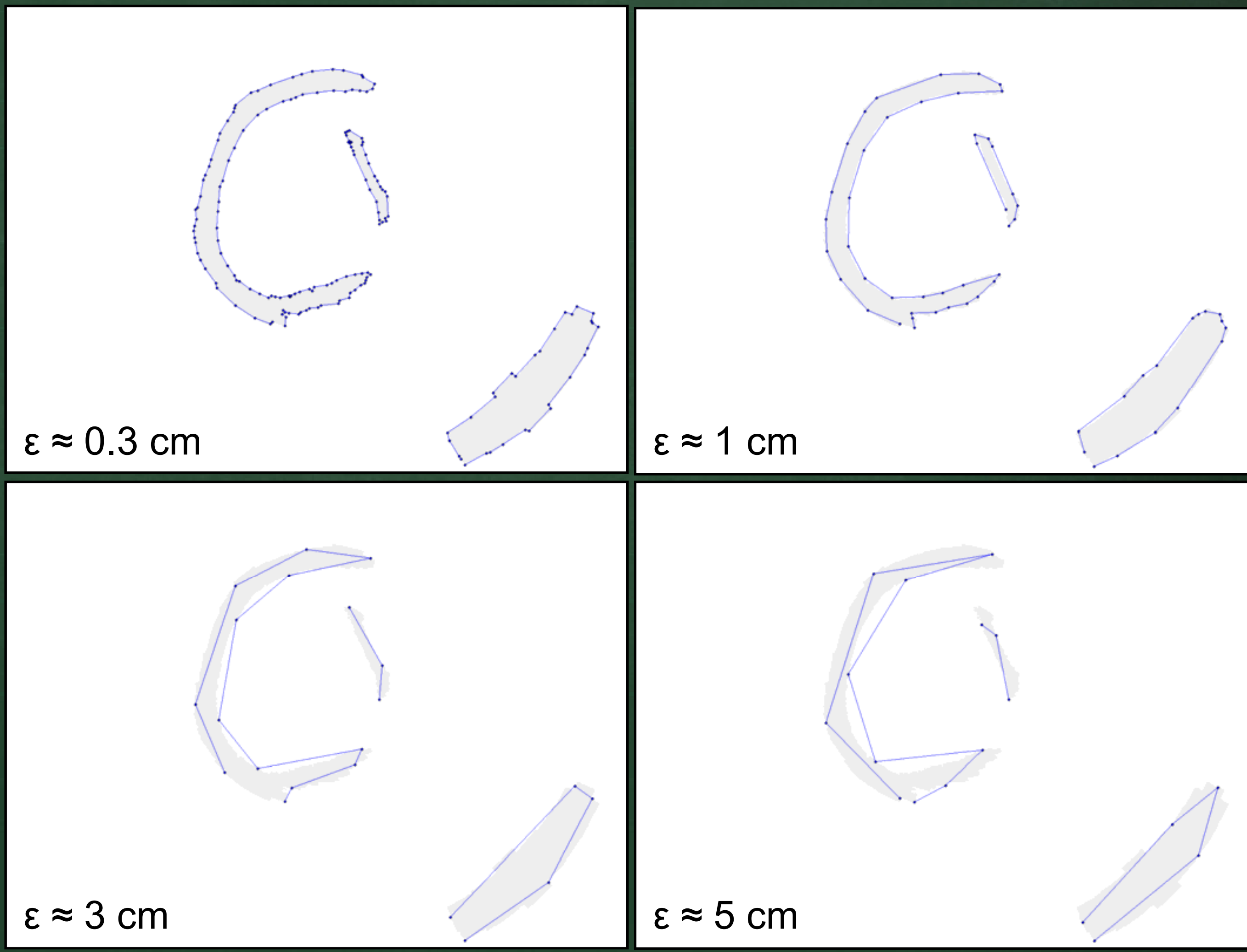
- Now we can take the set of border pixels from each border we found (i.e., a border was formed each time we found a new (x_s,y_s)):



- Then apply either the Split&Merge or Incremental segmentation algorithm:

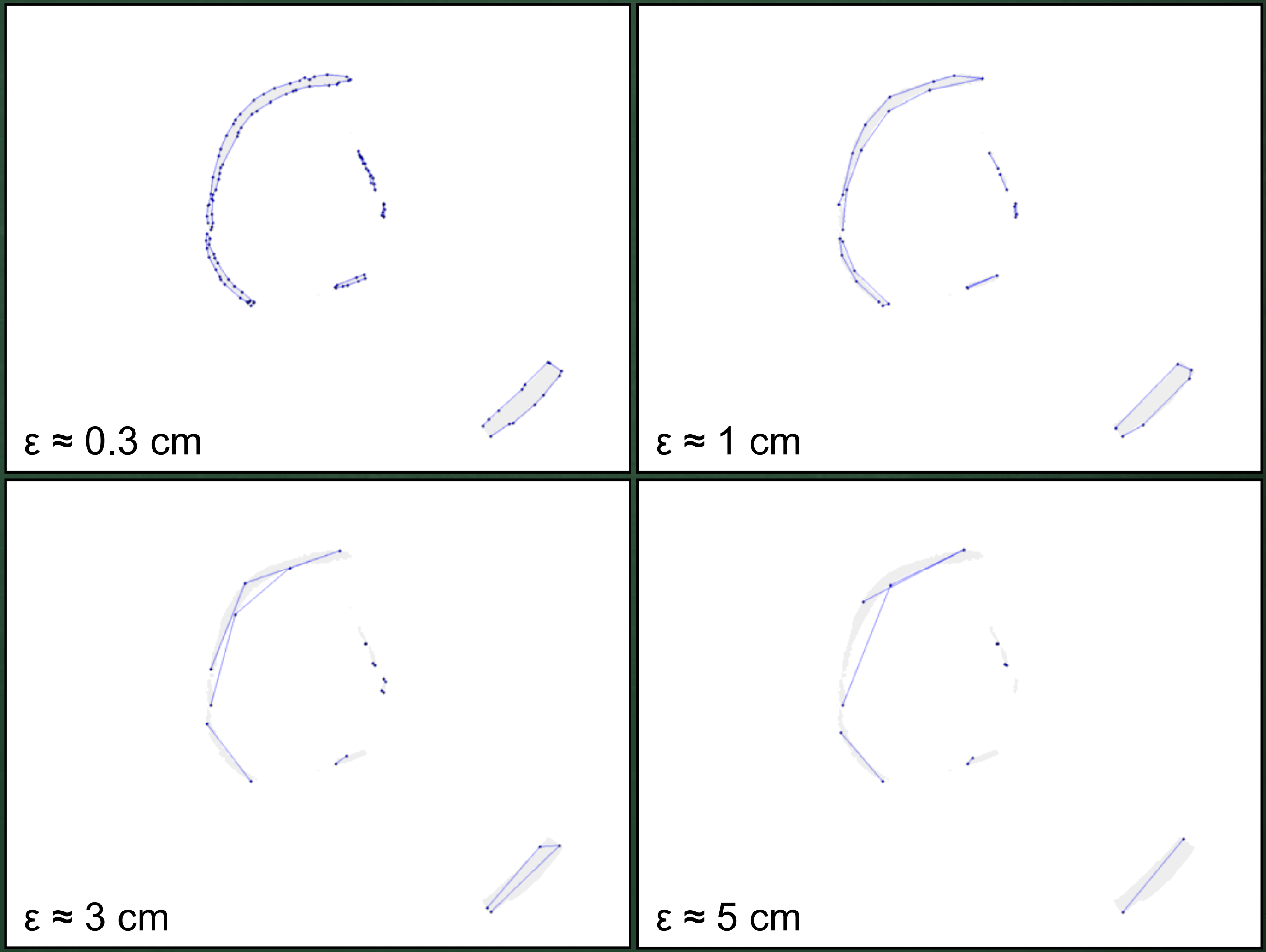
Fitting Lines

- Results:
  - sonar data
  - incremental algorithm
  - 0.51 threshold



Fitting Lines

- Results:
  - sonar data
  - incremental algorithm
  - 0.60 threshold

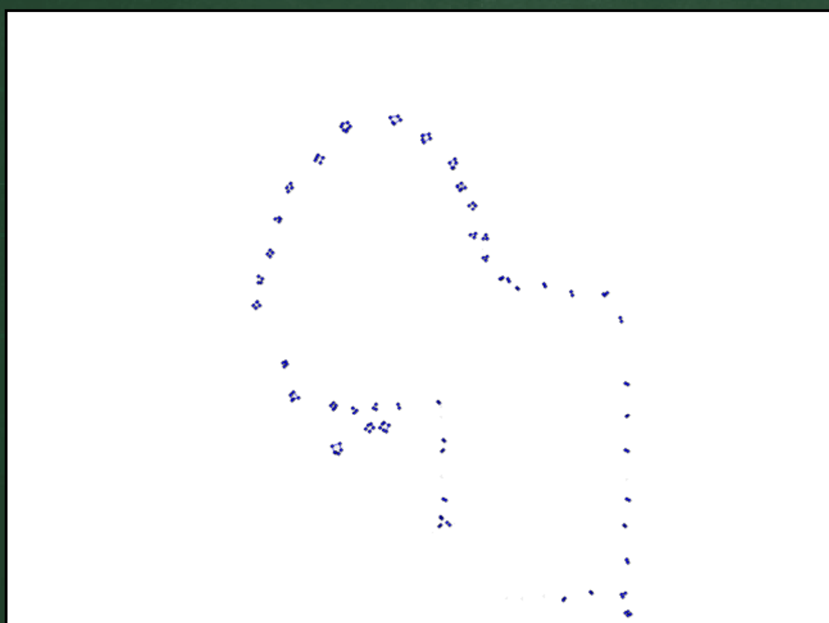


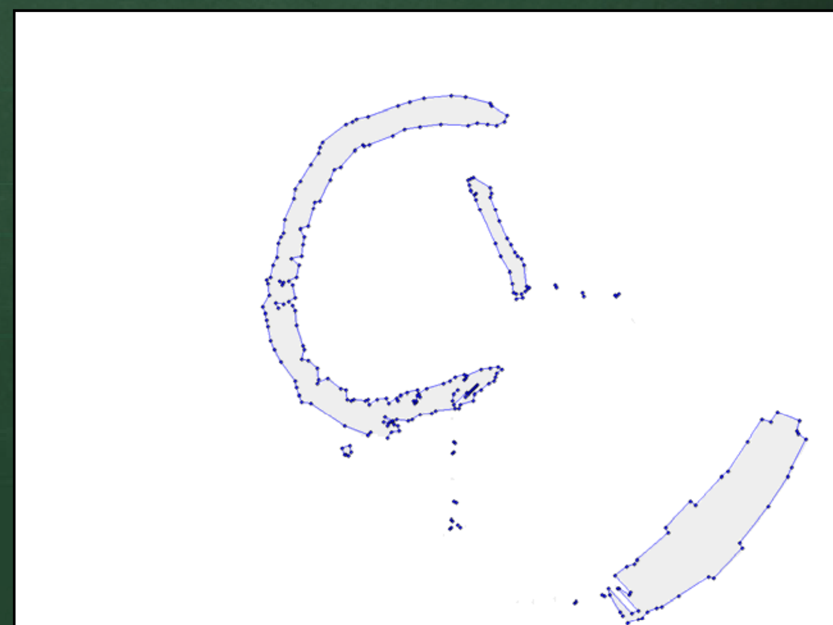
Fitting Lines

- More results:

  - IR data
  - incremental algorithm
  - 0.51 threshold

  - fused sonar and IR data
  - incremental algorithm
  - 0.51 threshold





Summary

- You should now know how to:
  - obtain raster (grid) & vector maps of raw sensor data
  - use sensor models to account for errors in readings
  - estimate likelihood of obstacle locations
  - perform simple sensor fusion from multiple sensors
  - convert maps from raster to vector