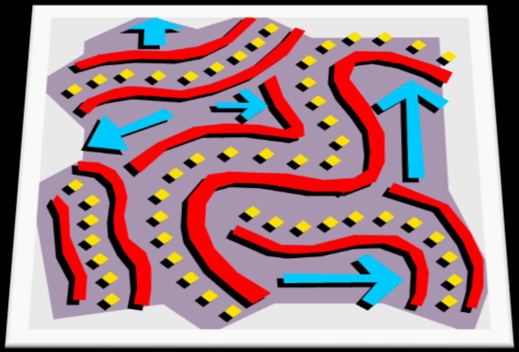
Mapping

Maps

- A robot's environment may change over time.

- A *map* is a stored representation of an environment at some particular time.

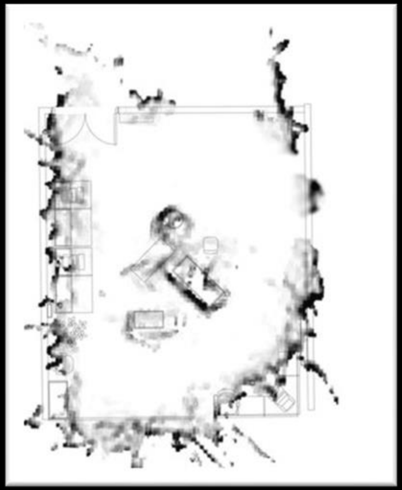
- Allows robot to:
  - plan navigation strategies
  - avoid obstacle collisions during travel
  - identify changes in the environment
  - identify accessible/inaccessible areas
  - verify its own position in the environment



- We will only consider 2-D maps in this course

Maps

- Realistically, maps are only estimates
  - often imprecise



- When navigating using a map, a robot must also rely on its sensors to avoid collisions since maps may be inaccurate or simply wrong.

- The goal when making a map is to make it as accurate as possible, although this is not easy.

- Before creating a map, we must decide on how it will be stored (i.e., represented) in memory.

Map Representation

- Maps can be represented as various types:
  - **Topological maps**

    - Keeps relations between obstacles (or free space) within env.

  - **Obstacle maps**

    - Keeps locations of obstacles and inaccessible locations in env.

  - **Free-space maps**

    - Keeps locations that robot is able to safely move to within env.

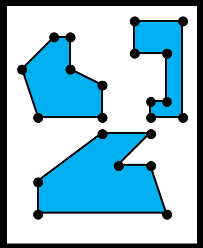
  - **Path/Road maps**

    - Keeps set of paths that robot can travel along safely in env.

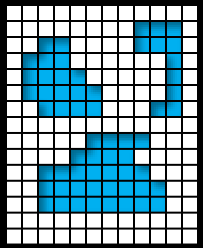
    - Usually used in industrial applications to move robots along known safe paths.

Map Representation

- Maps are stored in one of two main ways:
  - **Vector**
    - stored as collection of line segments and polygons
    - usually represents obstacle boundaries



  - **Raster**
    - storage in terms of fixed 2D grid of cells
    - each cell stores probability of occupancy (i.e., obstacle)

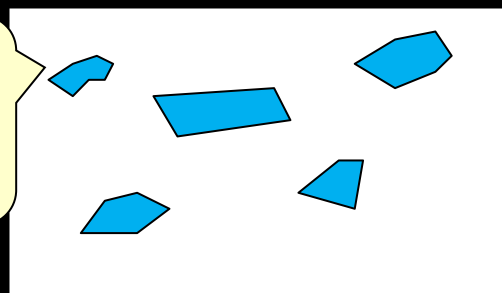


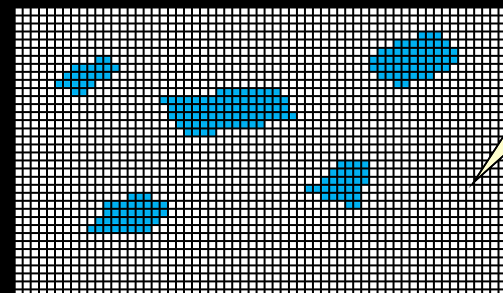
- Main differences lie in:
  - storage space requirements
  - algorithm complexity and runtime

Map Storage Space

- Large environments with few and simple obstacles take less space to store as vector:

Only need to store a few vertex coordinates and edge connections.

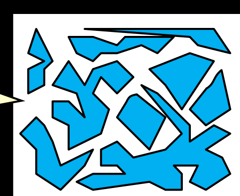


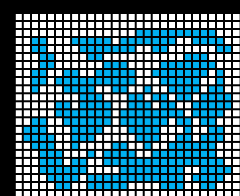


Both “occupied” and “empty” regions take up storage space.

- Smaller, obstacle-dense environments may be better stored as raster/grid:

Storing many vertices and edges may require more space than storing a small course grid.





Map Storage Space

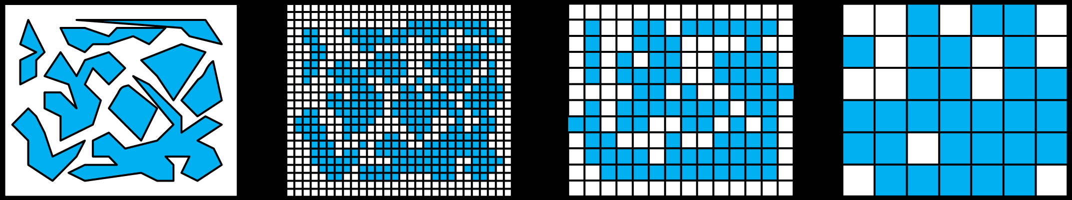
- Vector maps require the following storage space:
  - m obstacles with n vertices each requires storage of (x,y) vertex coordinates as well as edges (e.g., stored as linked list pointers)
  - Storage = $(m * n) * 2_{\text{integers}} + \underbrace{2_{\text{indices}} * (m * n)}$ = **O(mn)**

Optionally, edges don't have to be stored explicitly, but same time complexity.

- Raster maps require storage space that varies according to grid size (i.e., according to desired resolution):
  - a grid of size M x N takes **O(MN)**
- If m,n **<<** M,N then vector maps are more efficient

Map Storage Space

- Of course, much varies according to the resolution of the raster maps (i.e., depends on M & N).
- Resolution depends on desired accuracy. Notice the difference that it can make on the map:

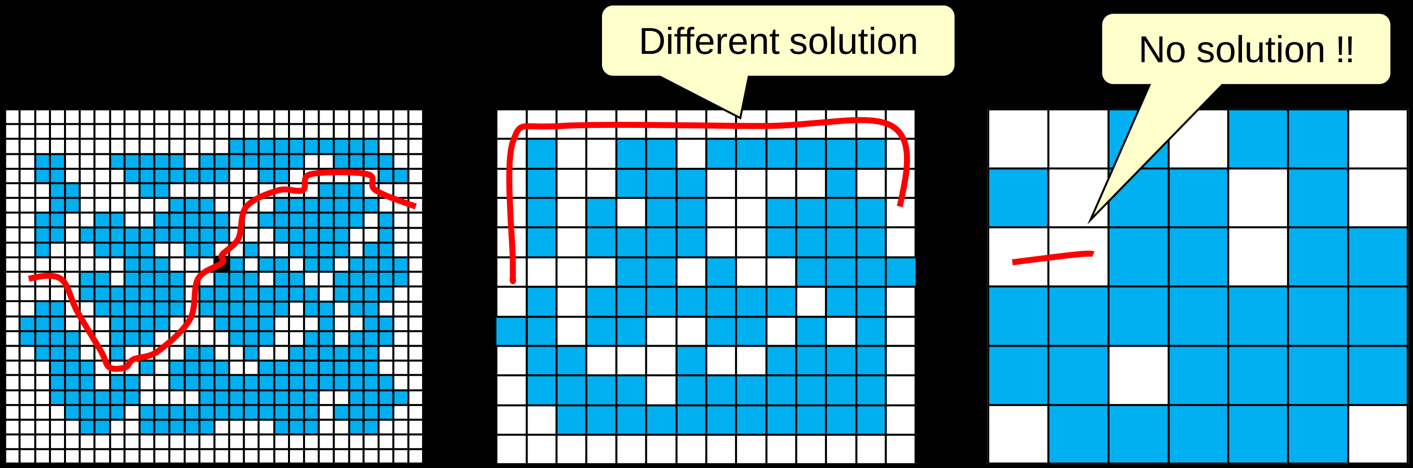


actual **28 x 24** **14 x 12** **7 x 6**

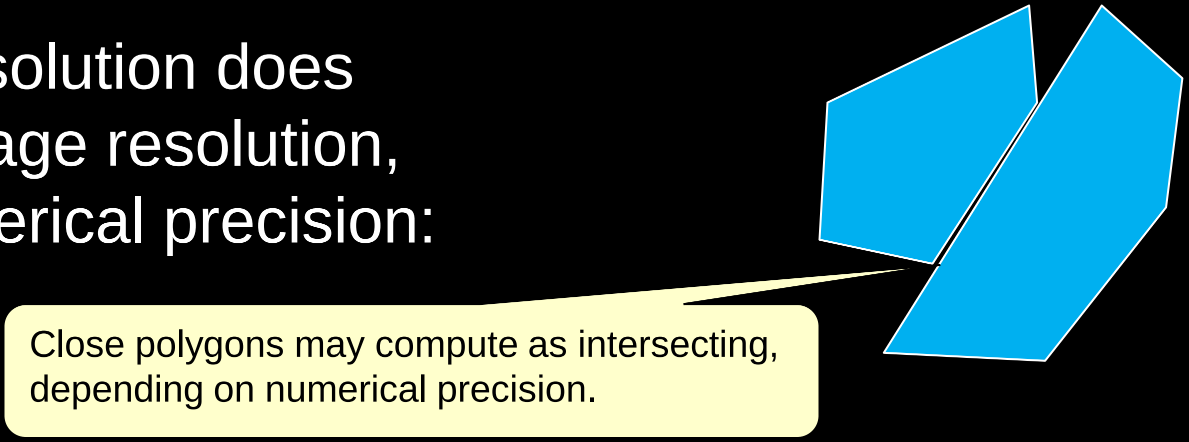
- As resolution decreases:
  - storage requirements are reduced
  - representation of "true" environment is compromised

Map Accuracy

- This decrease in accuracy can affect solutions to problems:

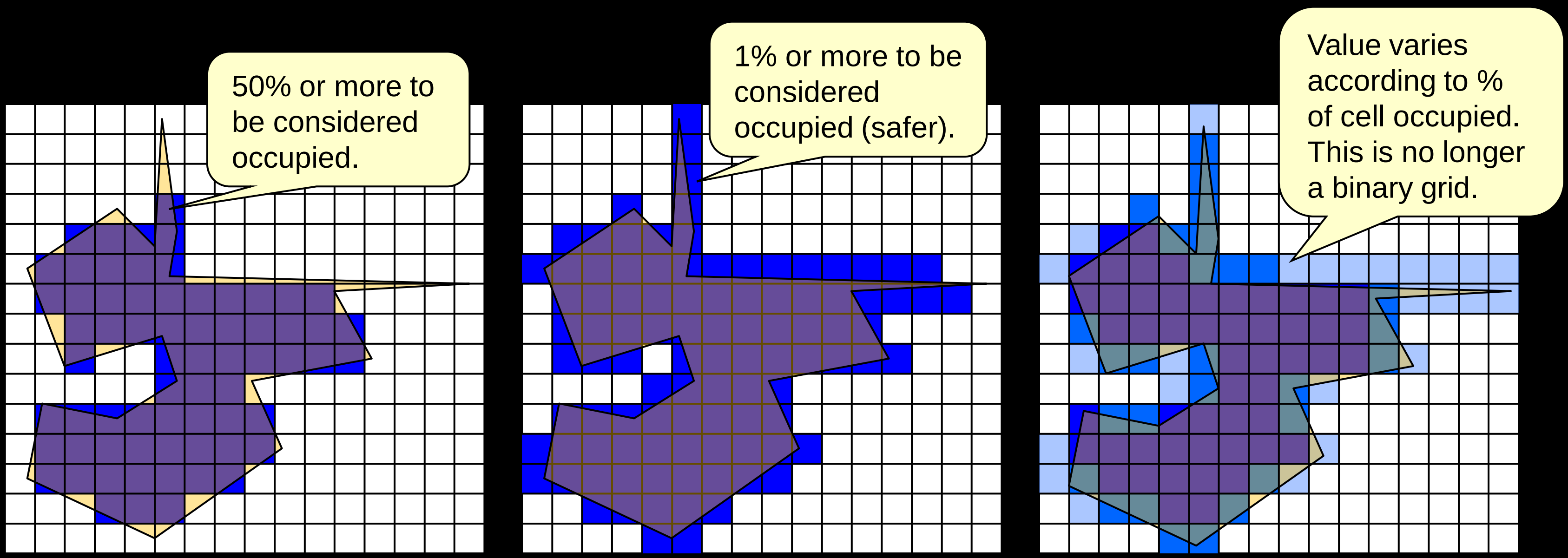


- With vector maps, solution does not depend on storage resolution, but instead on numerical precision:



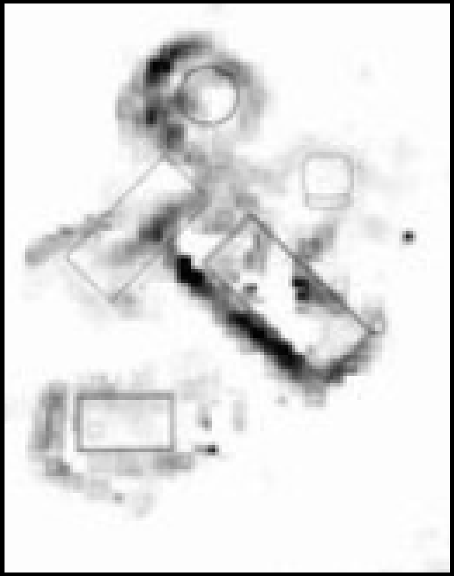
Map Accuracy

- Robot safety may be another issue with raster maps.
  - Assumptions about obstacle locations may lead to collisions
- Occupancy of grid cells can depend on some threshold indicating "*certainty*" that obstacle is at this location:



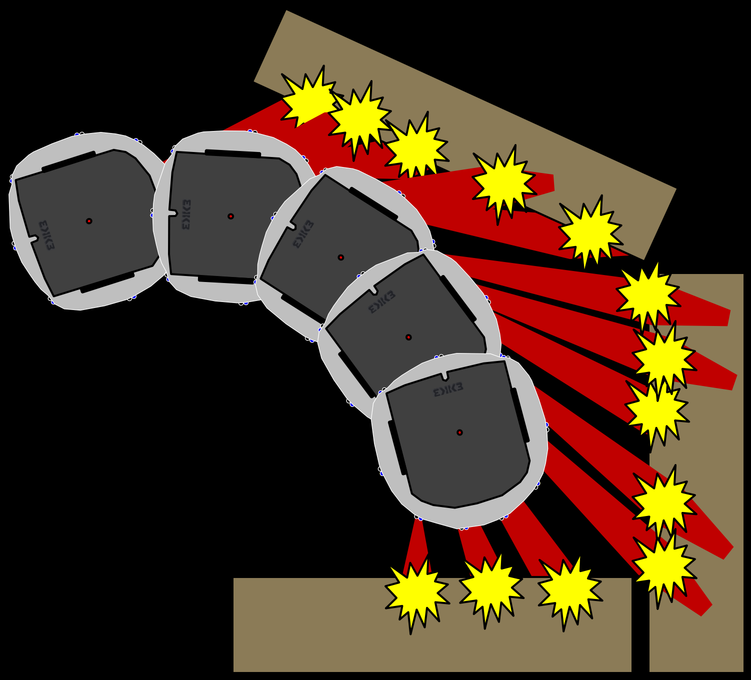
Map Accuracy

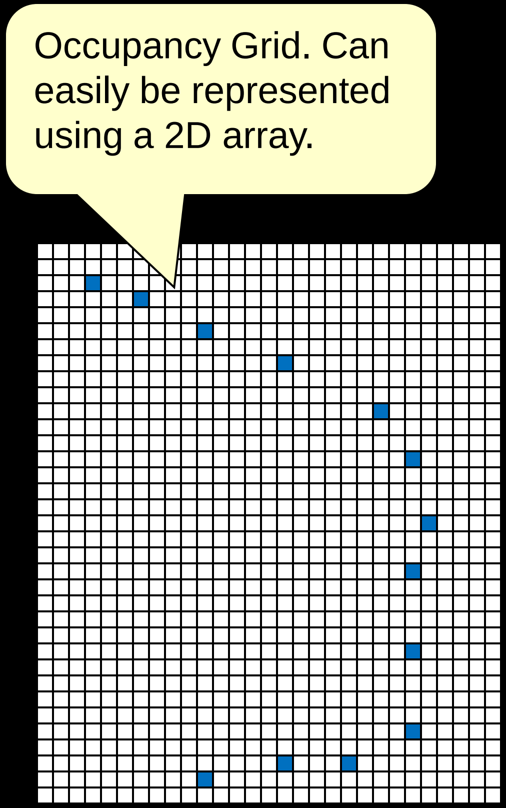
- In practice, many robots use such raster maps because they allow for “fuzziness” in terms of obstacle position.
  - They are commonly called ***occupancy grids*** (or ***certainty grids*** or ***evidence grids***).
- Still useful since most maps are constructed based on sensor data (which is already uncertain).
- The cell’s occupancy value indicates the probability that an obstacle is at that location.



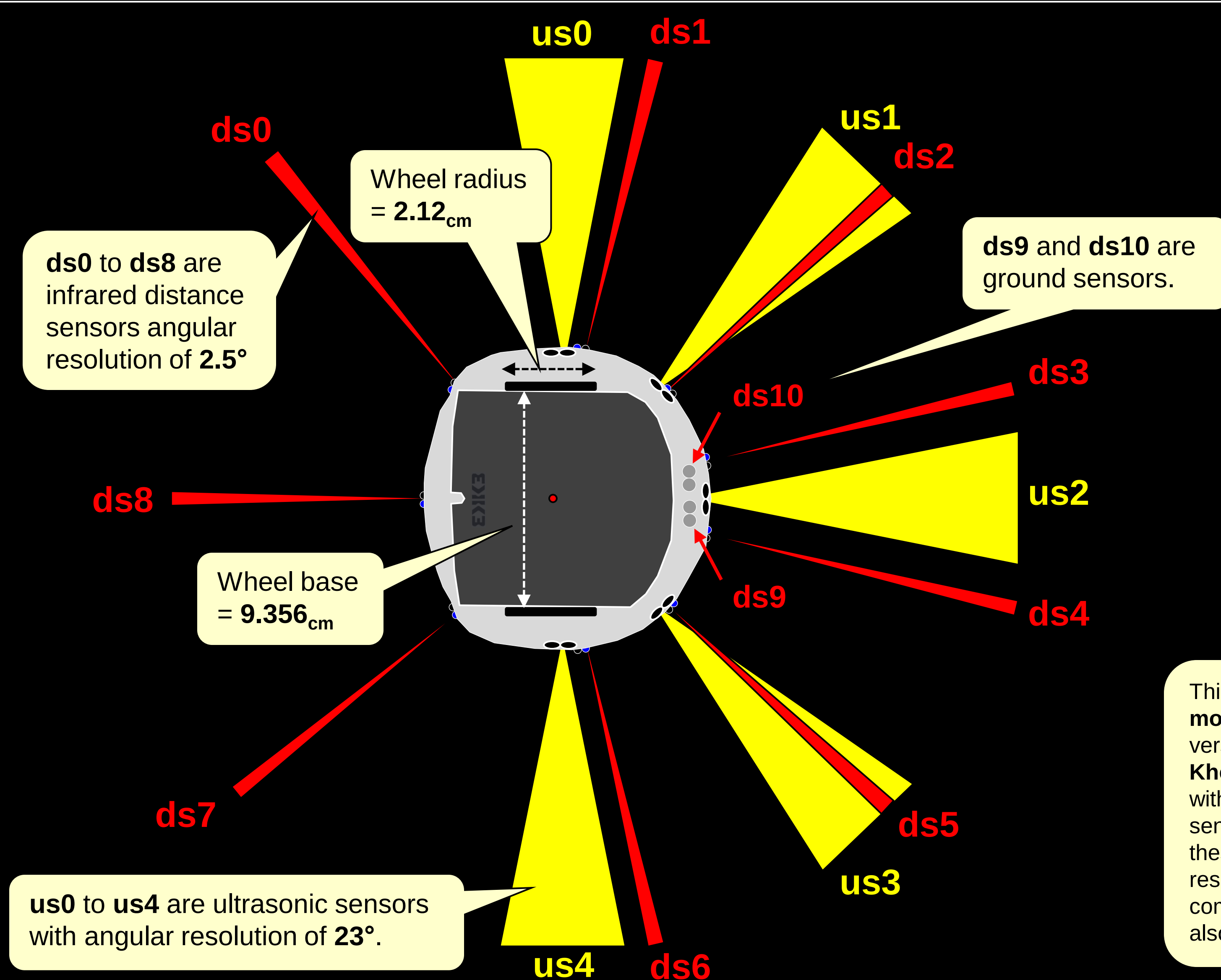
Multiple Readings

- A map is formed by merging together the results from multiple readings from various locations in the environment.



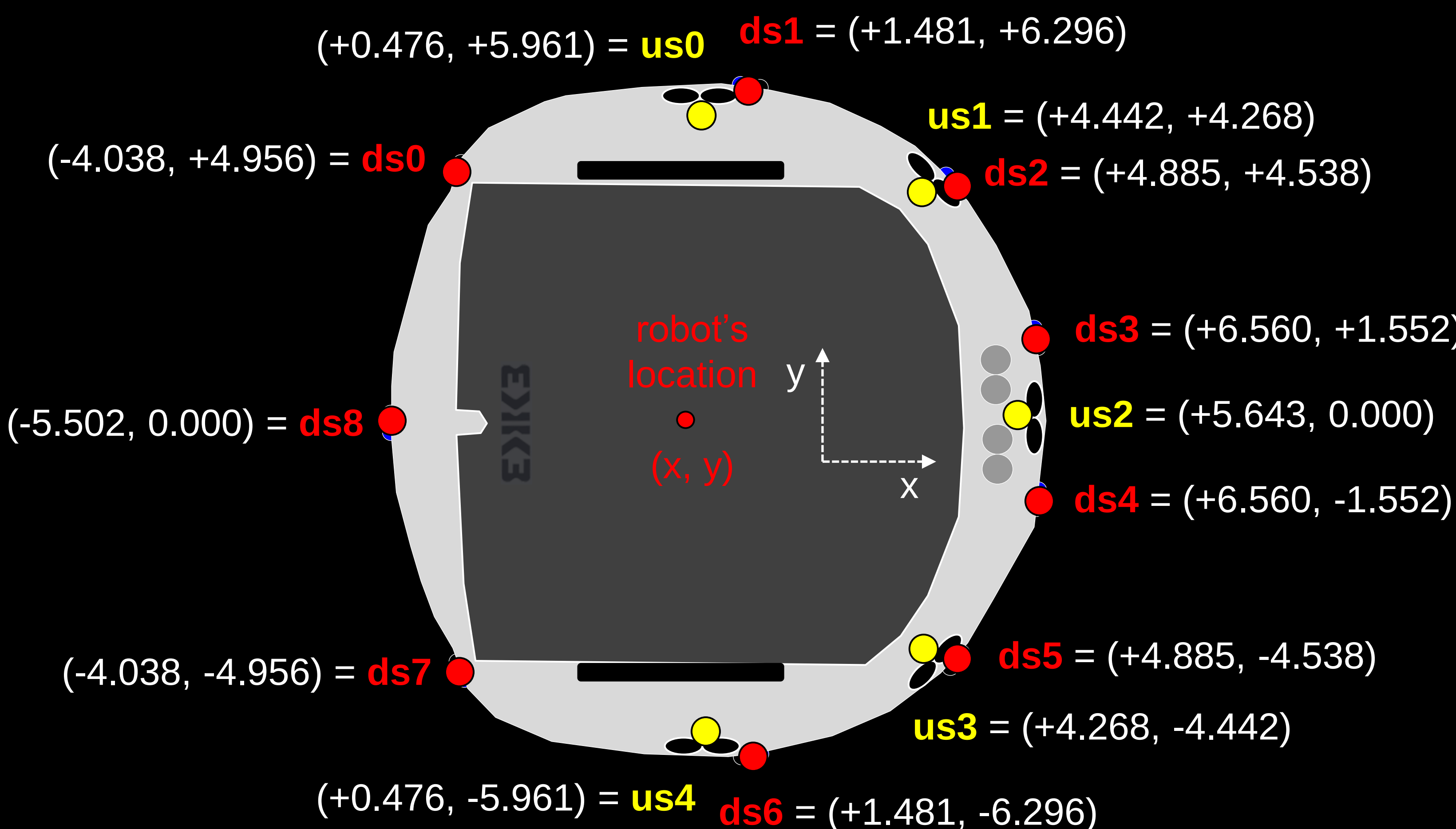


The Khepera III Robot

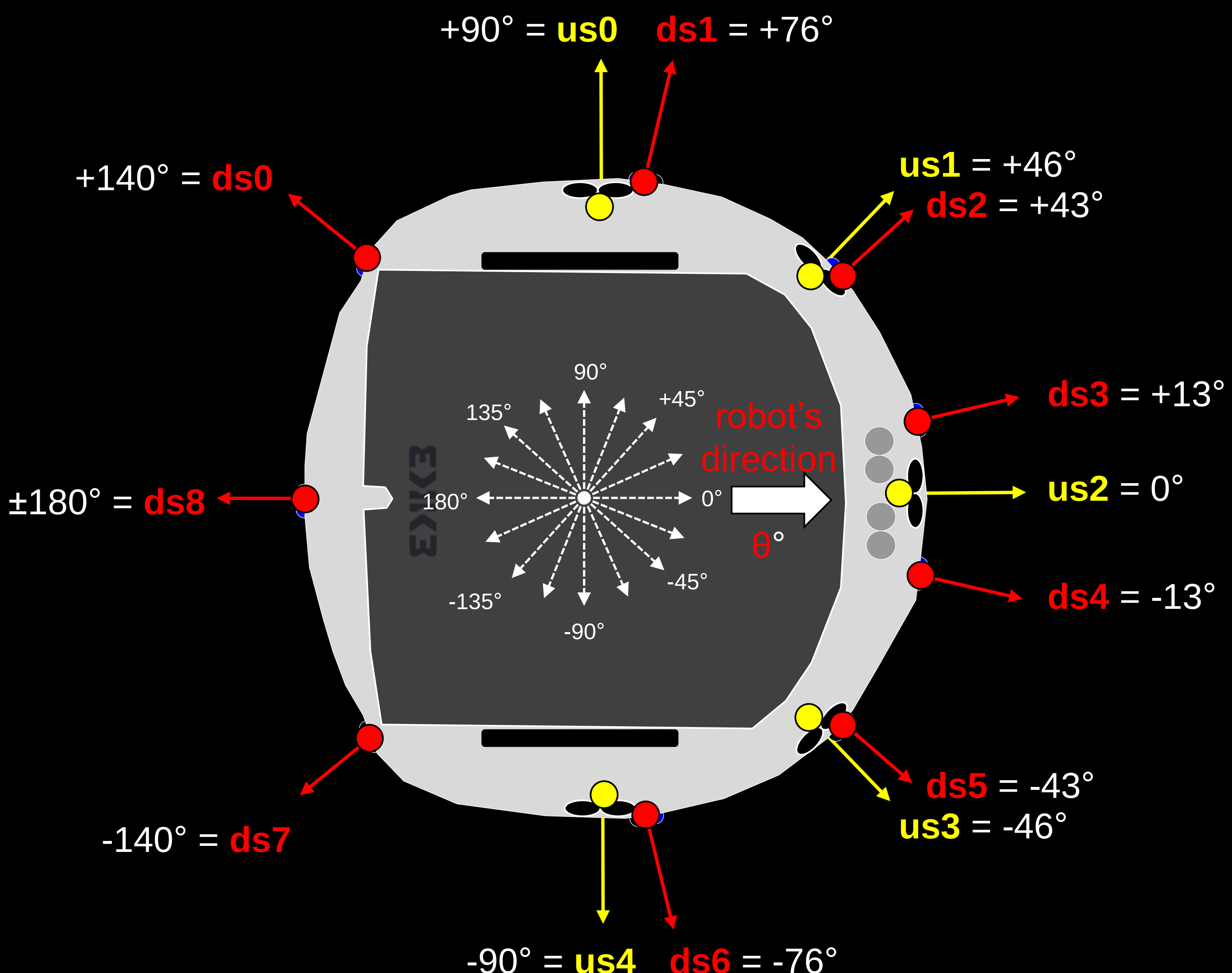


This is a **modified** version of the **Khepera III** with the sensors set to these angular resolutions. A compass was also added.

Sensors – Location Offsets



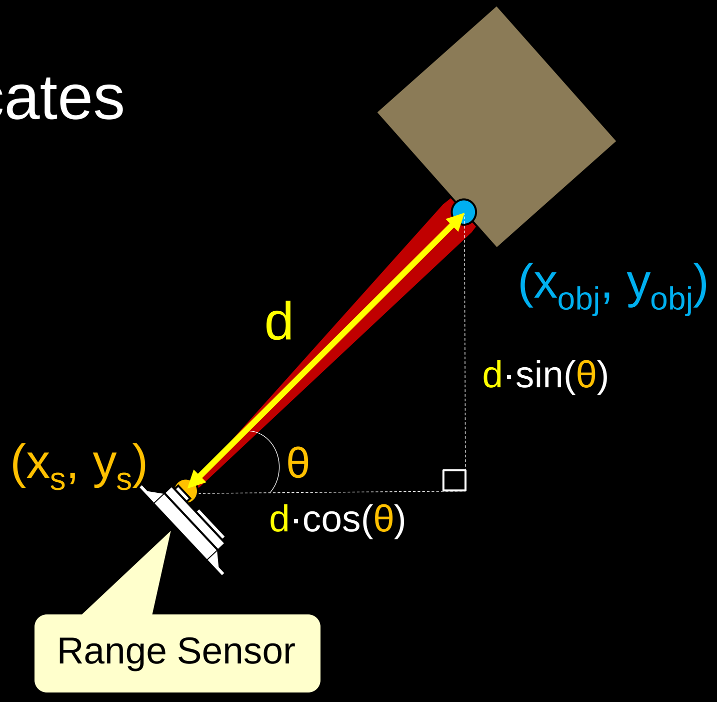
Sensors – Angle Offsets



Mapping Raw Sensor Data

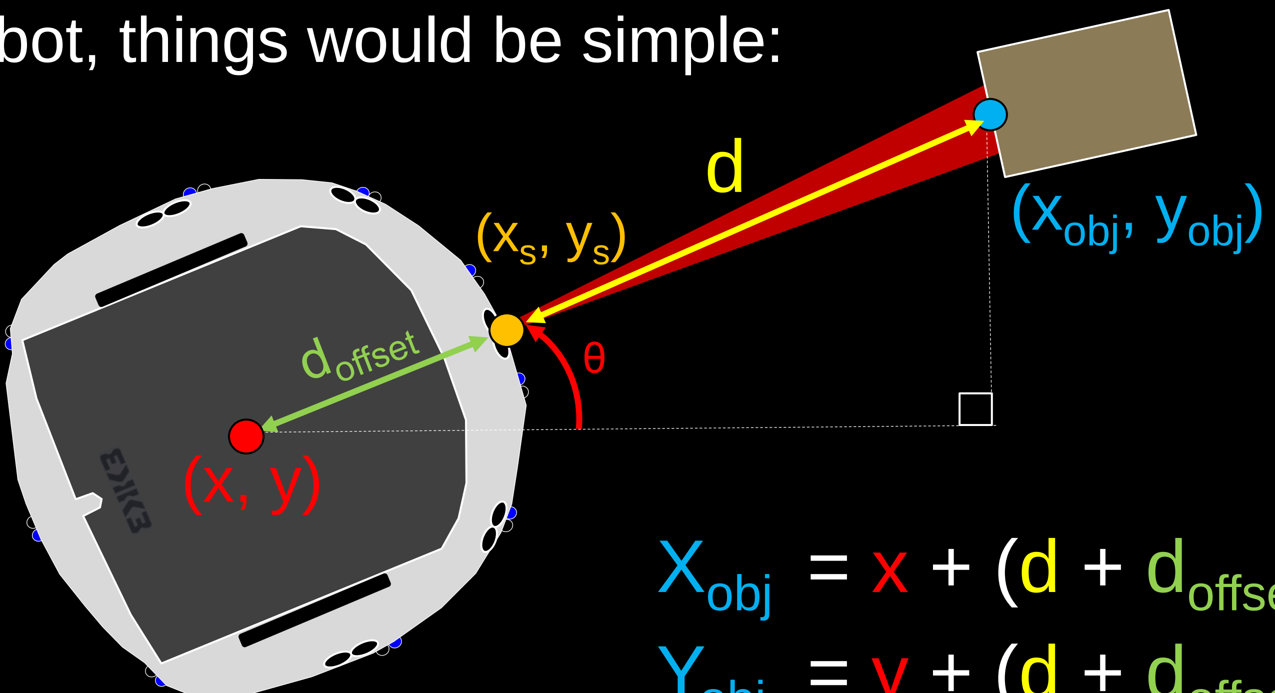
- Mapping involves taking lots of sensor readings.
- Assume that a range sensor is at location (x_s, y_s) in the environment and is tilted at angle θ with respect to the horizontal in the coordinate system.
- Assume that the sensor reading indicates an object at d_{cm} from the sensor.
- We now know that there is an object at location (x_{obj}, y_{obj}) where:

$$x_{obj} = x_s + d \cdot \cos(\theta)$$
$$y_{obj} = y_s + d \cdot \sin(\theta)$$



Computing Object Location

- The location (x_{obj}, y_{obj}) of an object needs to be computed with respect the robot’s pose (x, y, θ) because that is the only reference point that we have within the environment.
- If the sensor was pointing straight ahead at the front of the robot, things would be simple:

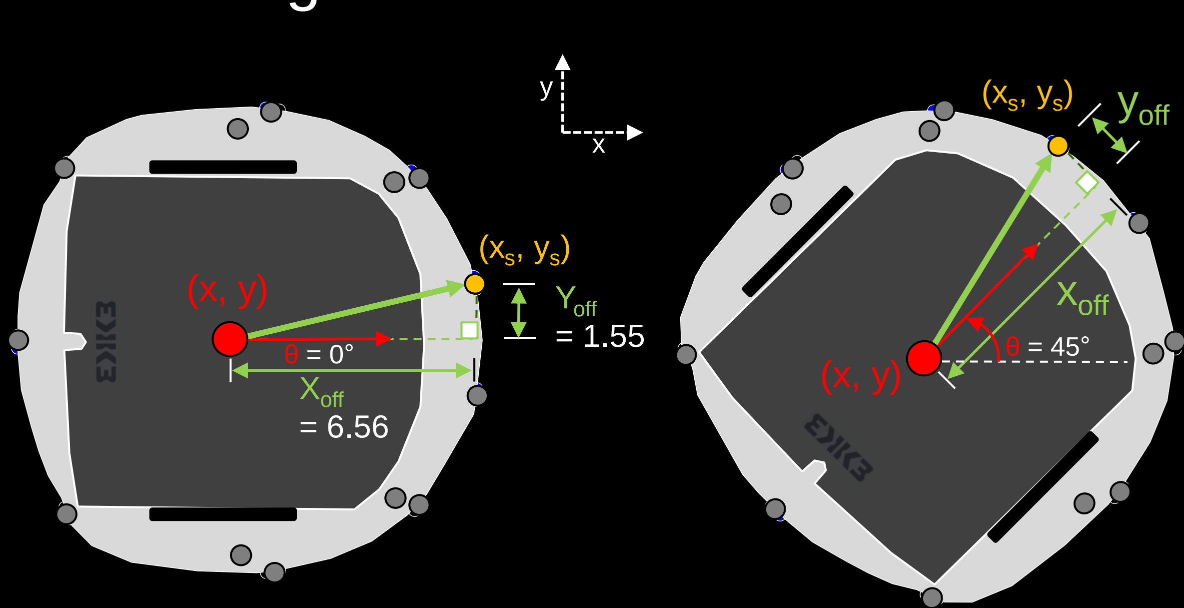


$$X_{obj} = x + (d + d_{offset})\cdot\cos(\theta)$$

$$Y_{obj} = y + (d + d_{offset})\cdot\sin(\theta)$$

Rotating Sensor Offsets

- To get reading with respect to the robot's position (i.e., center of robot), we need to *transform* (i.e., *rotate*) the sensor offsets according to direction the robot is currently facing.

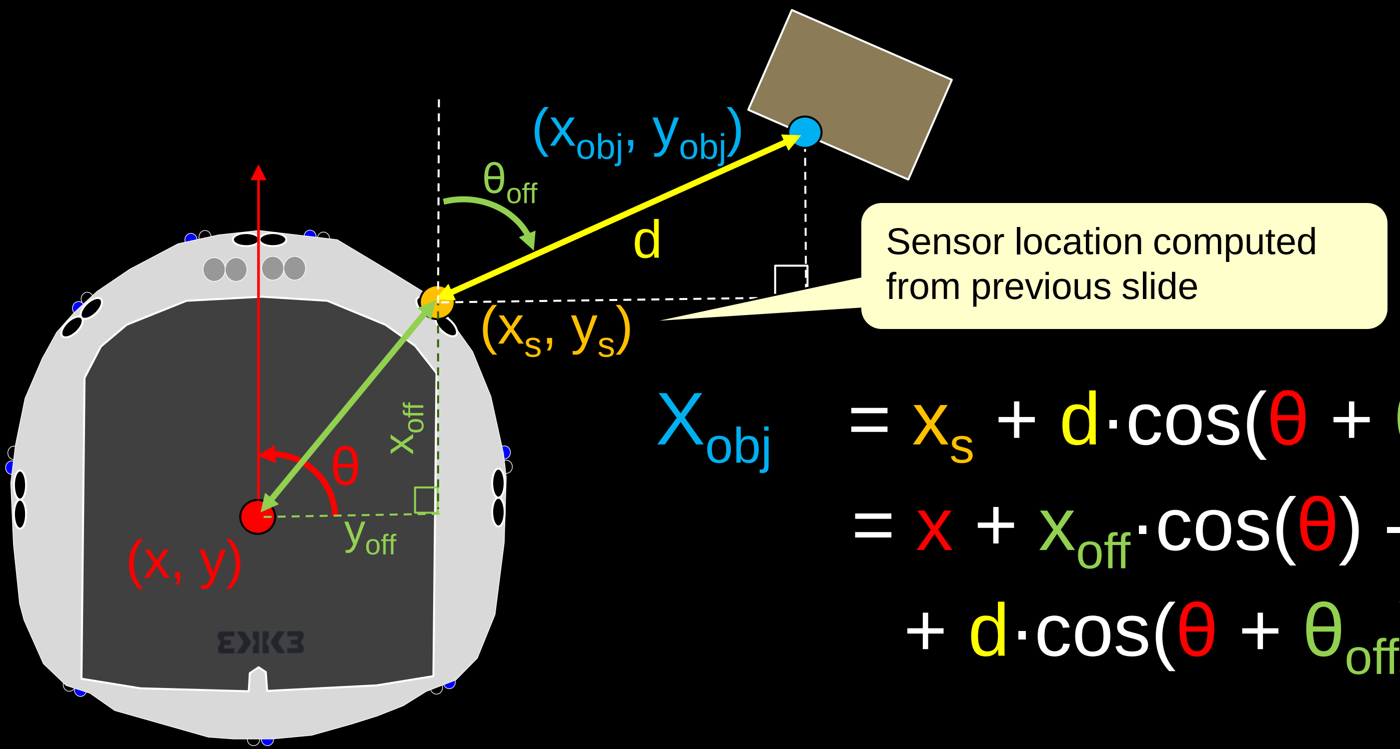


$$x_s = x + x_{off}\cdot\cos(\theta) - y_{off}\cdot\sin(\theta)$$
$$y_s = y + y_{off}\cdot\cos(\theta) + x_{off}\cdot\sin(\theta)$$

Sensor location computed as rotation around the robot's location (x, y).

Accounting for Sensor Offsets

- For each sensor, you must take into account it's offsets:



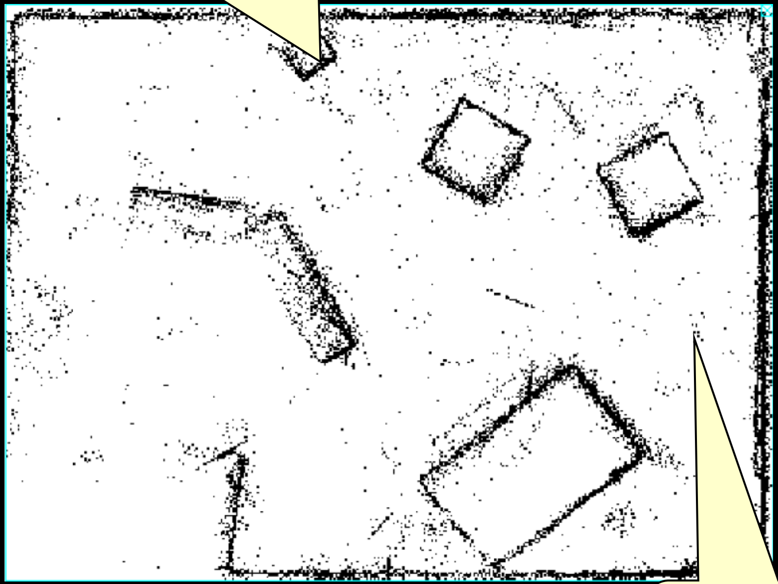
$$X_{obj} = x_s + d \cdot \cos(\theta + \theta_{off})$$
$$= x + x_{off} \cdot \cos(\theta) - y_{off} \cdot \sin(\theta)$$
$$+ d \cdot \cos(\theta + \theta_{off})$$

$$Y_{obj} = y_s + d \cdot \sin(\theta + \theta_{off})$$
$$= y + y_{off} \cdot \cos(\theta) + x_{off} \cdot \sin(\theta)$$
$$+ d \cdot \sin(\theta + \theta_{off})$$

Mapper App

- An embedded MapperApp class (included with your controller) allows you to display an occupancy grid map.
- Map points are filled in for each (x_{obj}, y_{obj}) computed.

Black dots indicate all the (x_{obj}, y_{obj}) computed over time.



Create the display for mapping based on the loaded world size (in cm). This code is given to you.

```
MapperApp  mapper;
mapper = new MapperApp(worldX, worldY, display);

mapper.addObjectPoint(Xobj, Yobj);
```

You do this each time you want to add a new object point to the map. Pass in the coordinate and it will appear on the map.

There will be lots of invalid/noisy readings

Reading the Sensor

- Here is how to set up the distance sensors on the Kheperra III robot:

```
import com.cyberbotics.webots.controller.DistanceSensor;

// Sensors are objects
static DistanceSensor   rangeSensors[];

// Set up the range sensors to be used
rangeSensors = new DistanceSensor[9];
for (int i=0; i<9; i++) {
   rangeSensors[i] = robot.getDistanceSensor("ds"+i);
   rangeSensors[i].enable(timeStep);
}
```

- Here is how to read distance **d** from the sensor:

```
// Read a distance reading d for sensor i in the array of sensors
double  d = rangeSensors[i].getValue() * 100;
```

Do this to read a single distance reading from one sensor in the array. Multiply by 100 to convert to centimeters.

Start the Lab …