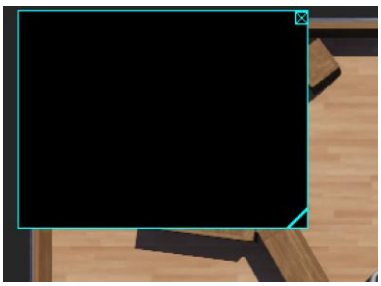
LAB 11 – Sensor Models

(1) Download the **Lab11_SensorModels.zip** file and unzip it. Load up the **CratesWorld** world. The world should appear as shown below. You may have to turn on **3D View** from the **Tools** menu.

The goal of this lab is to apply the sensor models to each reading as we create a map. At first, this may not seem to create a better map than what we had in **Lab10**, but we will have a follow-up lab to improve on the map further. In addition, we will use the less accurate ultrasonic (i.e., sonar) sensors along with the IR sensors that we used in the previous lab.

The code will make use of the code from **Lab10** (i.e., the **Mapping** lab). You will make use of a revised **MapperApp**, which allows for grayscale coloring of the occupancy grid. As with the previous lab, there will be two additional files in the controller directory (**MapperApp**.java and **Map**.java). This time, you will be altering the **Map.java** code.

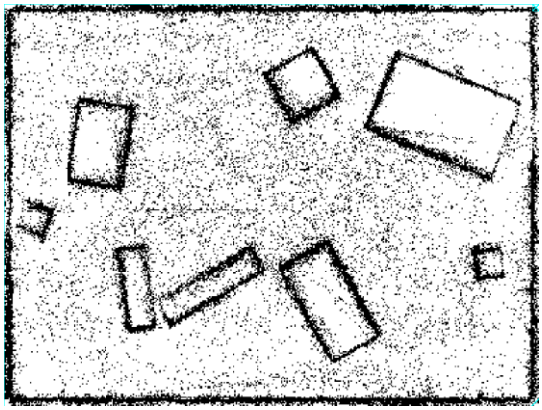
IMPORTANT:



This code makes use of a display window that will appear as black upon startup. The display window will show the map. It can be resized by grabbing the bottom right corner. You should NOT ever close it using the top right corner. If you close it accidentally, you can re-open it by clicking on the robot … then from the **Overlays** menu select **'e-Puck Overlays'** then **Display Devices** then **Show 'display' overlay**. If the window does not appear, check to make sure that **Hide All Display Overlays** is unchecked in the **Overlays** menu. Sometimes you have to reload the environment and sometimes even restart Webots to make the display appear. If this doesn't work, you can delete the WBPROJ file in the worlds directory that corresponds to the world that you are using, then re-open or reload the world. However, the WBPROJ files are hidden files, so you need to enable your windows/mac to show hidden files. On Windows pcs, check of **Hidden Items** under the **View** menu of the **File Explorer**. On a Mac, press **control+shift+period** to show hidden files.

To get a better understanding of how the sensor model mapping works, we will keep things simple and use the actual robot location and accurate compass sensor readings for this lab, instead of estimating its pose based on kinematic equations. This is like having a local (i.e., small scale) GPS position available at all times and will give us fairly accurate maps.

The **Lab11Controller** code has been mostly completed. It causes the robot to roam around in the environment avoiding obstacles as in **Lab10**. The robot starts at position ($\mathbf{x}_0$,$\mathbf{y}_0$, $\mathbf{a}_0$) = (0, 0, 90°). Each time the robot moves (except for spinning) it will call the `addSensorReadingsToMap()` function to record its sensor readings from that new location. The function is currently written in a way that gets the robot's location and orientation and then sends the object points for all the IR sensors to the mapper by calling the mapper's **addObjectPoint()** function that we used in **Lab10**. It should produce a map when you run, that looks like this shown here on the right →



However, the map WILL NOT have a transparent background anymore.

(2) To begin, in the `addSensorReadingsToMap()` function, you will need to replace the call to **addObjectPoint()** with a call to the **MapperApp** function called **applySensorModelReading()** which will send a single sensor model reading to the **MapperApp** for updates to the occupancy grid. The function has this signature:

```
applySensorModelReading(double sensorX, double sensorY, int sensorAngle,
                        double distance, double beamWidthInDegrees,
                        double distanceErrorAsPercent);
```

The point (`sensorX`, `sensorY`) represents the location of the sensor in the environment (i.e., not the location of the robot). You may recall from **Lab10** (slide 18) that this location is computed as follows:

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

$$sensorY = y + y_{off} \cdot \cos(\theta) + x_{off} \cdot \sin(\theta)$$

where (x_{off}, y_{off}) is the offset of the sensor with respect to the center of the robot (x,y) and the robot is facing angle **θ**. The position offsets are given to you for each sensor in the provided array where the first **9** values are the offsets of the **IR** sensors and the next **5** values are the offsets of the **Ultrasonic** sensors.

The `sensorAngle` is the angle (in degrees as an **int** … so you'll need to typecast) that the sensor makes in the environment (which is the robot's angle plus the sensor angle offset). The angle offsets are also given to you for each sensor in the provided array where the first **9** values are the angle offsets of the **IR** sensors and the next **5** values are the angle offsets of the **Ultrasonic** sensors … all in degrees.

The `distance` is the current reading (i.e., **d**) from the sensor.

The `beamWidthInDegrees` will be one of the two beam widths (i.e., for either the IR sensor or the sonar sensor … see slide 21).

The `distanceErrorAsPercent` is the percentage error in the distance reading for that sensor (e.g., 0.03 for a 3% error … see slide 21).

For now, we will just send the **9** IR sensor readings to the map and ignore the ultrasonic sensor readings. In the **Map** class, you need to write code for the similarly-called **applySensorModelReading()** method as well. In there, for now, you should take the incoming sensor reading and spread it across the **beam width only** (i.e., ignore the distance error) by using the pseudocode from slide 24 (which also makes use of slide 23).

The pseudocode indicates to `set grid at (objX, objY) = 1`. However, to change a cell in the occupancy grid, you should first check to make sure that it is within range by calling the **`isInsideGrid`**`(objX, objY)` function before attempting to set it. Then, as long as the computed location is inside the grid, you should call **`setGridValue`**`(objX, objY, (short)1)` to set the grid value to **1**.

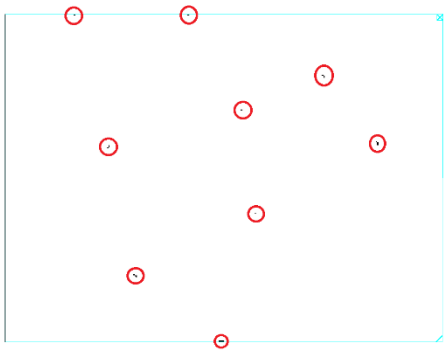
Now you are ready to test. Set the **MAX_SPEED** of the robot to **0** in the controller code. For now, we will stop the robot from displaying more than one set of sensor readings, to make sure that all is working. Comment out the first two lines from the Map class's **draw()** function:

```
// if (DISPLAY_COUNTER++ %250 != 0)
//     return;
```

Uncomment the following line at the end of the **while** loop in **Lab11Controller.java**:

```
if (count++ == 1) break;
```

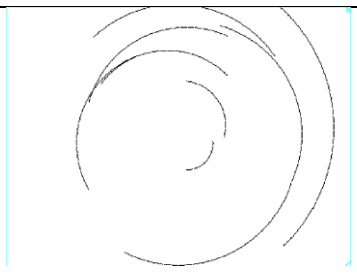
Now run your code. You should see something appear similar to what you see here (the red circles were added by me) →

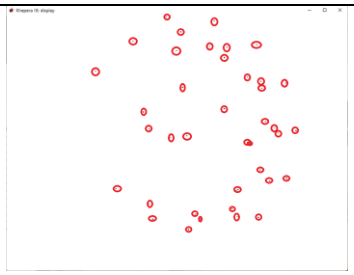
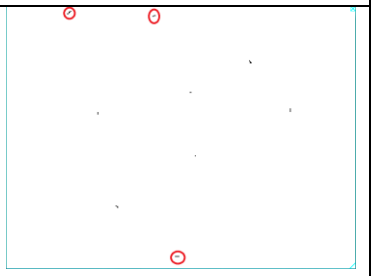


This map shows readings from **9** sensors applied to the map. You should notice that the reading gets a little wider as the distance from the robot is larger. Double-click on the map to get it into a separate window. Save a screen snapshot of the **MapperApp** window as **Snapshot1.png**.

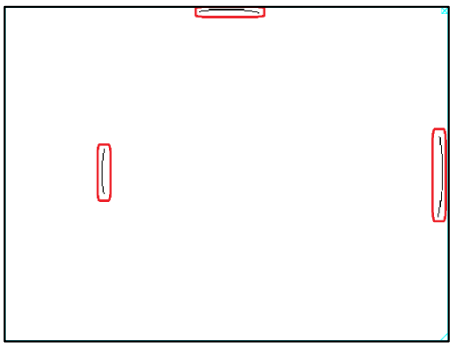
Debugging Tips:

WARNING: We are now using multiple java files. Unfortunately, in Webots … if some of your code does not compile, the simulator will still use your **previously-compiled code** and you may not realize it. Please make sure that ALL of your code compiles before you test anything.

• If you get an error like this shown here, then it means that you have an **infinite loop** somewhere. Print out your value for **ω** to make sure that it is positive, otherwise your FOR loop is going in the wrong direction.	**Forced Termination (because process didn't terminate itself after 1 second)**
• If your results only show one reading, such as what is here on the right … then you likely forgot to comment out the first two lines of the **draw()** method in the **Map** class.	
• If you get some very big circular arcs like this, you forgot to convert all of your angles to radians BEFORE calling the **cos()** and **sin()** functions in your calculation of (x_a, y_a) and (x_b, y_b).	

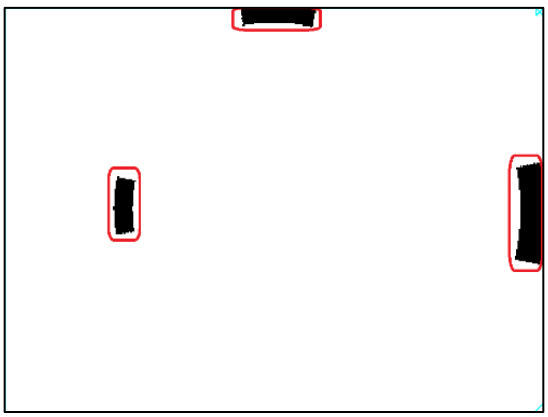
• If you get a bunch of 40 or so dots all over, then you forgot to convert to radians BEFORE calling the **cos()** and **sin()** functions in your computation of (**obX**, **objY**).	
• If you get something like this shown here → then you forgot to add the angle offsets for each sensor.	
• If the top and bottom sensor readings are not all the way at the top and bottom of the environment as shown here → then you forgot to add the **xOffset** and **yOffset** values.	

(3) Now go back to your **Lab11Controller** code and comment out the code that sends the IR readings and instead, send the **5** ultrasonic sensor readings, using the provided FOR loop just after the one for the IR sensors. Complete the FOR loop by calling **applySensorModelReading**(). Make sure to use the proper indexing when accessing the ultrasonic sensors (i.e., **0** to **5** … not **i**). Run your code again. You should see **3** of the **5** sensor readings displayed in a similar way to what you see here. If what you get is very different, then maybe you have the beam width, sensor offsets or your indices wrong. Save a screen snapshot of the **MapperApp** window as **Snapshot2.png**.



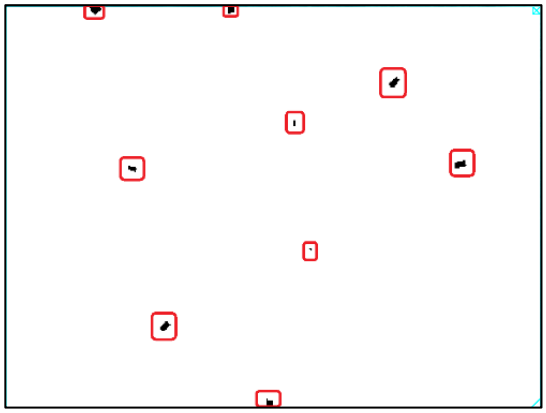
(4) Now you will apply the distance error. Go back to the **applySensorModelReading()** function in the **Map** class and change the code so that it applies the distance error as well as the beamWidth error. Do this by following the pseudocode in **slide 26** of the notes, with a value of **INC = 0.25**.

Run the code again. You should see something like what you see here → , where the readings are thicker. Save a screen snapshot of the **MapperApp** window as **Snapshot3.png**.



Now go back to the **Lab11Controller** code and comment out the sending of the ultrasonic sensor readings and uncomment the code that sends the IR readings. Then run again.

You should see something like what is shown here, where the readings are thicker. Save a screen snapshot of the **MapperApp** window as **Snapshot4.png**.



(5) If all of the above is working fine, you should be pretty sure that your application of the sensor models is working properly. It is time to start producing some maps.

Put back the first two lines in the Map class's **draw()** function:

```
if (DISPLAY_COUNTER++ %250 != 0)
   return;
```

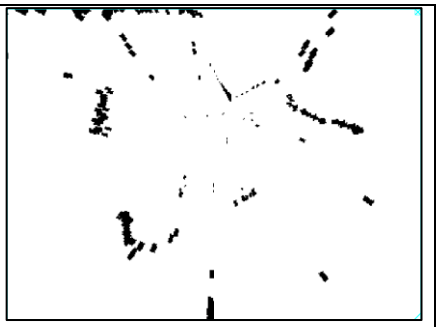
Comment out the following code at the end of the **while** loop in **Lab11Controller.java**:

```
if (count++ == 1) break;
```

Set the **MAX_SPEED** of the robot to **15**. Change your code so that it only sends IR sensor readings to the mapper if they are less than **30cm**. Make sure that the ultrasonic readings are NOT being sent. You can put the simulator on fast forward … but only run your code until the timer indicates **0:05:00:00.** Save a screen snapshot of the **MapperApp** window as **Snapshot5.png**.

Debugging Tips:

- If your code is running very slow … make sure that you added back in those two lines in the **draw()** function! Also, you likely forgot to put in the condition to only show readings that are less than **30cm**. Your map may wrongly look as shown like this →



Now change your code so that it only sends Ultrasonic sensor readings to the mapper if they are less than **30cm**. Do NOT send the IR readings. Run your code until the timer indicates **0:05:00:00.** Save a screen snapshot of the **MapperApp** window as **Snapshot6.png**.

Change your code so that it sends **both** the IR sensor readings **AND** the Ultrasonic sensor readings to the mapper if they are less than **30cm**. Run your code until the timer indicates **0:05:00:00.** Save a screen snapshot of the **MapperApp** window as **Snapshot7.png**.

You will notice that the maps are more rough-looking compared to mapping the exact points that we did in the last lab. But don't forget … in the last lab we assumed that the readings that we obtained were always accurate … which was not true. Now we are accounting for possible errors in the readings. You will also notice that the ultrasonic sensors are less precise.

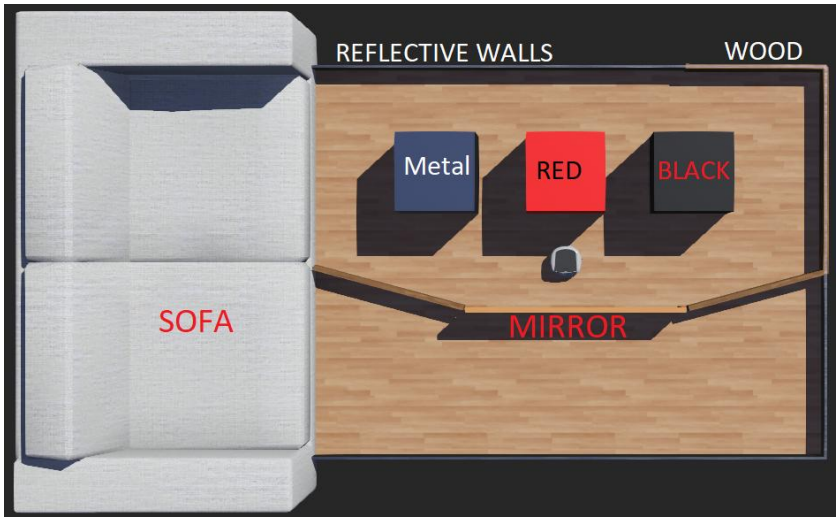
(6) Go to the **Lab11Controller** code and change the **3rd** parameter in the **MapperApp** constructor call (around the 7th line of the **main()** function code) to **false**. This will allow gray-scale mapping. In your **applySensorModelReading()** code that you wrote in the **Map** class, after you know that a computed object location is inside the grid, check the **isBinary** attribute of the **Map** class. If it is set to **true**, set the grid value as you already were doing. But if it is **false**, call **increaseGridValue**(**objX**, **objY**) instead. That will allow the locations of the occupancy grid to store larger and larger values as the same location gets more readings.

Change your code so that it only sends IR sensor readings to the mapper if they are less than **30cm** and does NOT send Ultrasonic sensor readings. Set the **MAX_SPEED** of the robot to **15**. Run your code until the timer indicates **0:10:00:00** (which is twice as long as when you ran the previous maps). You should see a gray-scale map now where the edges of the objects are darker while the noisier areas are lighter. The map will differ every time since some areas will be covered more than others. The more an area gets covered, the darked the edges will be in that part of the map. Save a screen snapshot of the **MapperApp** window as **Snapshot8.png**.

Now change your code so that it only sends Ultrasonic sensor readings to the mapper if they are less than **30cm** and does NOT send IR sensor readings.. Run your code until the timer again indicates **0:10:00:00**. You may see some rounding in the corners due to the wider sonar beams. Save a screen snapshot of the **MapperApp** window as **Snapshot9.png**.

You guessed it … change your code so that it sends **both** the IR sensor readings AND the Ultrasonic sensor readings to the mapper if they are less than **30cm**. But we will run the simulation **twice as long now** … until the timer indicates **0:20:00:00**. You will end up with a map with much darker edges now. Save a screen snapshot of the **MapperApp** window as **Snapshot10.png**.

(7) You may be thinking “The ultrasonic sensors are terrible!” Why would anyone use them since the beam with is so wide and the distance error is larger? Well … let us now see why. Load up the **HardToMapWorld** world:



It contains a big sofa on the left, a mirror, a portion of reflective walls and wooden walls, and three types of boxes: red, black and reflective metal. As you can guess, infrared light-based IR sensors may have some difficulties mapping metal and mirrors. Let’s see what happens.

Make sure that the **MAX_SPEED** is at **15**. Change your code so that it only sends IR sensor readings to the mapper if they are less than **30cm** and does NOT send Ultrasonic sensor readings. Run it again. The code will actually run a lot slower so it will take longer to create your map. Make sure to run in fast-forward mode. Run your code until the timer indicates **0:07:00:00**. Save a screen snapshot of the **MapperApp** window as **Snapshot11.png**. What do you notice about the map? Notice how well the wood is detected but how poorly the sofa and reflective walls are detected. Also, the robot seems to be identifying itself in the mirror … which results in it thinking that the object is inside of the mirror. And what is with those boxes? It doesn't seem to be able to map the metal box nor the red one, but it did better with the black one. Red and shiny objects are known for causing problems with IR sensors, as well as sunlight.

Now we will see how the sonar sensors do. Change your code so that it only sends Ultrasonic sensor readings to the mapper if they are less than **30cm**. It will run slower since it takes longer to update the map. You only need to run it until the timer indicates **0:04:00:00**. Save a screen snapshot of the **MapperApp** window as **Snapshot12.png**. What do you notice about the map? The sofa and all of the walls are detected properly (with the standard issues in the corners). The boxes are all detected properly, although we don't see the edges between the boxes because the robot rarely (if ever) travels in-between them. The mirror does not cause any issues either. Do you understand how the sonar sensors can have an advantage over IR sensors? We didn't even discuss the issues that pop up with glass doors and windows in buildings!!

(8) For some final fun … load up the **StillMessyRoom** world. Set your code to use all sensor readings from both types of sensors but only if they have a distance of less than **20cm**. Remember that this world needs **+15** added to the computation of **y** in the **addSensorReadingsToMap ()** function as follows:

```
double y = -(values[2]*100) + 15;
```

Run your code until the timer indicates **0:20:00:00**. Save a screen snapshot of the **MapperApp** window as **Snapshot13.png**. If you want … you can let the simulation run for **0:60:00:00** and see what you get … but don't hand that in.

Submit your **Lab11Controller.java** code along with the two **MapperApp** files as well as the **13** snapshot files. Make sure that your name and student number is in the first comment line of your code.