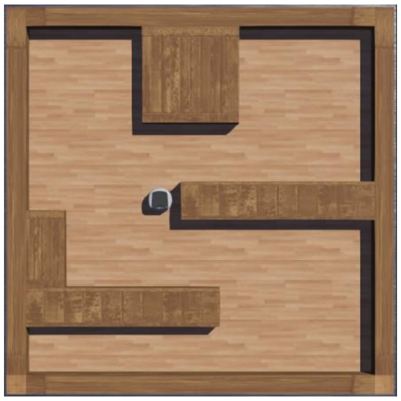
LAB 19 – Localization

(1) The goal of this lab is to be able to determine the location and orientation of the robot based on a given map and distance readings as the robot moves around in the environment. We will use the *Monte Carlo Localization* approach. Download the **Lab19_Localization.zip** file and unzip it. Load up the **LocalizationWorld** world.



The **MapperApp** will again make use of a simple internal webots display for showing the robot's position estimates. The **VectorMap** class now contains hard-coded vector maps that correspond to the three Webots worlds that we will be using.

The code for moving the robot around in the environment has already been written in the **Lab19Controller** file. It uses IR sensors to check for collisions and the front-facing ultrasonic sensor for checking distance measurements. You will NOT be modifying the **Lab19Controller** code. Instead, you will be writing your code in the **LocationEstimator** class that will continuously update position estimates as the robot moves and takes sensor readings.

In the **Lab19Controller** code, the following happens:

- Each time the robot begins spinning, it reads the compass. When it stops spinning, it reads the compass again. It then calls the following method in the **LocationEstimator** class, passing in the change in angle (i.e., the number of degrees that the robot turned):

  ```
  public void updateOrientation(double degrees)
  ```

- Each time the robot begins moving forward (i.e., upon starting or when it just finished spinning), it remembers its location. When it encounters an obstacle, it examines its location again and computes the distance travelled based on when it started moving forward. It then calls the following method in the **LocationEstimator** class, passing in the distance travelled (in cm):

  ```
  public void updateLocation(double distanceTraveled)
  ```

  It also takes a distance reading from the front-facing ultrasonic sensor and then calls the following method in the **LocationEstimator** class, passing in the distance from the robot's (x,y) location to the obstacle (in cm):

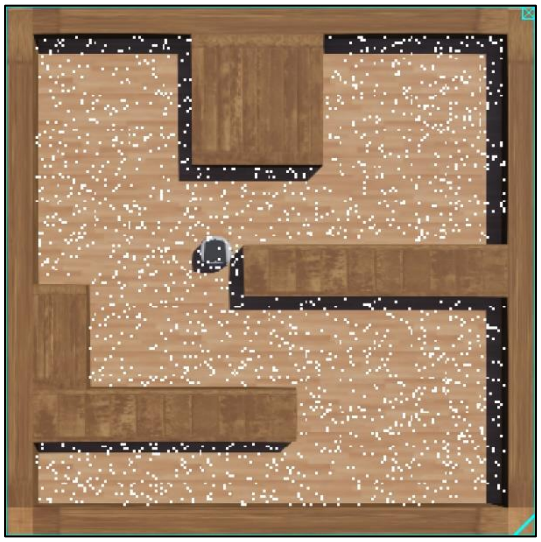
  ```
  public void estimateFromReading(double distanceReading)
  ```

- While moving forward … every UPDATE_FREQUENCY (which is set at **10**) number of iterations, it also calls the `updateLocation`() and `estimateFromReading`() methods as well.

In this assignment, you must write the code for those three methods, based on the functions described in slides 7 to 9 of the class notes.

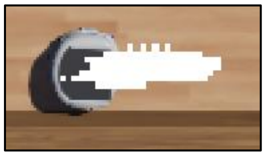
In the **LocationEstimator** class, the following has been provided … and you can make use of them:

- A NUM_SAMPLES constant to indicate how many random samples to use.
- A PERCENT_TOLERANCE constant which is the % error tollerance to use (see 4th **isGoodEstimate()** function parameter on slide 8)
- A **`ArrayList<Pose>`** `currentEstimates` attribute that should always hold the curresnt **Pose** estimates that are being used and displayed (i.e., all the white dots). It is not passed in as a parameter, but it is accessable from anywhere in your code.
- Use **vectorMap.getObstacles()** to get the list of obstacles, since it is not passed in as a parameter.
- An **isValidPose(Pose p)** function that determines whether or not the given **Pose** is valid (i.e., lies outside all obstacles and within the boundaries).
- A **resetEstimates()** method that computes NUM_SAMPLES random valid estimates and stores them in the **`currentEstimates`** variable. It makes use of **getValidPose()** which computes and returns a random **Pose** that is considered valid.
- An **intersectionPoint(int** x1**, int** y1**, int** x2**, int** y2**, int** x3**, int** y3**, int** x4**, int** y4**)** function that returns the intersection point of line segment (x1,y1)→(x2,y2) and line segment (x3,y3)→(x4,y4).
- A pre-defined function **java.awt.geom.Line2D.Double.linesIntersect(x1, y1, x2, y2, x3, y3, x4, y4)** which returns **true** if the line segments intersect and **false** otherwise.



When you are ready to test your code, run the simulator with the **LocalizationWorld** … a corresponding **VectorMap** will be automatically created. If you wrote the code properly, you should see some samples displayed. Make sure to translate and stretch the display screen so that it "perfectly" overlaps the environment as shown here on the right.

If your code is working properly … you will see the white dots converging into clusters at various locations … narrowing down over time until it ends up with a single cluster (see below) that closely follows the robot as it moves.

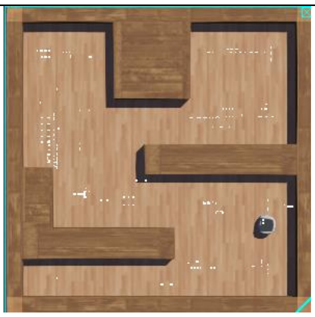


The single cluster may grow and shrink a little and it may even disappear, resulting in the robot generating all new samples again. This happens due to the **20%** randomness inherent to the algorithm as well as the inaccuracies of the robot sensors.

Once your code works, take a full world screen snapshot (called **Snapshot1.png**) of the robot after it finds a single cluster with the estimates overlayed on the map as shown in the above pictures.

Debugging Tips:

- If you are getting a lot of straight (i.e., pattern-like) lines like this … then you are likely typecasting wrong. Do not type-cast to integer in the middle of your calculations like this:

  ```
  p.x = p.x + (d * (int)…);
  ```

  instead, you must typecast after the entire calculation is done, like this:

  ```
  p.x = (int)(p.x + (d * …));
  ```

  Check everywhere that you are doing typecasting to make sure that you do not have this problem.

- If your estimates all seem to be sliding/shifting downwards or in some weird direction … you may have placed one of your closing braces (i.e., }) in the wrong spot … which alters the algorithm.

- If you are not seeing any white dots appear, then perhaps …
  - your display window is hidden. Press shift-function-F12 or go to Overlays menu to un hide it (i.e., **Hide All Display Overlays**).
  - the display window has been double-clicked and is open separate to the webots environment. Check your task bar.
  - the display window needs resetting. Go to the **worlds** directory and delete all the **.WBPROJ** files. They are hidden files … so you will have to enable windows to show hidden files. Then reload the environment (not recompile … but reload … the button next to the simulator time count field as shown here on the right).

- If you get random dots as "white noise" that never gets reduced, then perhaps you are not adding your good poses to the list … or maybe you are not identifying any good poses. Something is wrong with your code. Perhaps your computation of **p'** is wrong.

- If your code has a hard time keeping the located estimate or not getting it to converge:
  - Check to see if you are converting to radians when calculating **sin** and **cos** in the **updateLocation()** and **isGoodEstimate()** methods
  - Check to make sure that you are looping through ALL of the edges. Sometimes students get the FOR loop wrong. This, for example, is WRONG:

    ```
    for (int i=0; i<ob.numVertices()-1; i++) {
    ```

(2) Load up the **LocalizationNonConvex** world … its corresponding vector map will automatically be created. Run the simulation. You will notice that it is very difficult to get a single cluster to form. Try increasing the **NUM_SAMPLES** in the **LocationEstimator** code to a much larger number. You should be able to get it to work after a minute or two of running (real time, not simulator time). How long can it keep a single cluster when compared to the previous environment? Save a snapshot that shows the "best you can get" and call it **Snapshot2.png**.

(3) Load up the **LocalizationConvex** world … its corresponding vector map will automatically be created. Run the simulation. See if you can get a single cluster to form with the same number of samples as you used in the non-convex example. Is it harder or easier than the non-convex map to get it to converge to a single cluster? Why do you think this may be the case? Save a snapshot that shows the "best you can get" and call it **Snapshot3.png**.

Submit **ALL of your code**, including the files that were given to you … as well as the **3** snapshot **.png** files. Make sure that your name and student number is in the first comment line of your code.