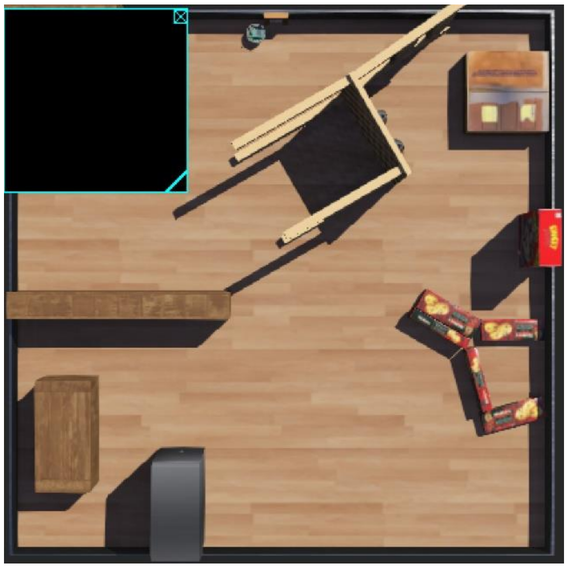
LAB 8 – Odometry Correction

(1) Download the **Lab8_OdometryCorrection.zip** file and unzip it. Load up the **JunkRoom** 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 understand how the robot's estimated position can change over time and how it can be improved by using an accurate compass. The code will make use of the code from **Lab 2** (i.e., the **WallFollowing** lab) and you will need to use your solutions from **Lab 4** (i.e., the **PositionEstimation** lab).



You will make use of a **TrackerApp** class which will display the robot's position estimates in comparison with its actual position. There will be two additional files in the controller directory (**Trace**.java and **TrackerApp**.java). These will automatically be compiled and used by your program. You should not change these files.

IMPORTANT:

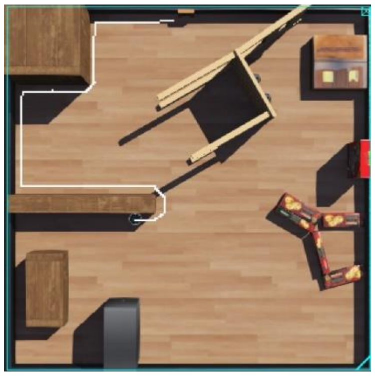
This code makes use of a display window that will appear as black upon startup. The display window will show the trace of the robot. It can be resized by grabbing the bottom right corner. You should NOT ever close it using the top right corner. If you don't see it (or close it accidentally), you can open it by selecting the robot … then from the **Overlays** menu select **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.

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.

You can also double-click the display window to move it out of the webots environment and into its own window. Closing that detached window will make the display overlay on the webots environment again.

(2) The **Lab8Controller** code has been started for you. It causes the robot to follow the environment boundary as in **Lab2**. When you run the code, it will bring up the Display window as shown above. You will want to stretch the display window so that it perfectly overlays the world that you have loaded.

The robot will start at logical position ($\mathbf{x}_0$,$\mathbf{y}_0$,$\mathbf{a}_0$) = (91.5, 192.9, 180°) … but the robot's translation field gives values in meters and needs to be offset. The adjustments will be made for you when displaying, so you do not need to worry about converting to **cm** or offsetting anything. Your job is to add the forward kinematic equations (given in part 3) that are based on **Lab 4** so that the robot continuously computes an estimate of its position.



To begin … you will first add code so that the robot's **actual** location is shown on the **Display** window as the robot moves (see slide 10 in the notes). A **displayEstimate()** function has been set up for you to write your code in. The function takes the **TrackerApp** object and the robot's **translationField** (see slide 9) as parameters. Test your code to make sure that as the robot moves, the white trace starts forming and looks like what is shown here →

(3) Now you will work on calculating and displaying the position estimate. Recall that every time the robot changes its motor speeds (i.e., whenever it changes its **currentMode** in the code) then the ICC circle changes … so you will need to recompute and update the location <u>when the mode changes</u>. Some **static** variables have been set up (i.e., **estimateX**, **estimateY**, **A**, **R**, **TD**) that can be set and accessed from any function. Notice that the code between the two **SWITCH** statements will ensure that whenever the **currentMode** changes, the robot's position estimate (**estimateX**, **estimateY**, **A**°) is recomputed and displayed. The **updateEstimate()** function should be used to change the **estimateX**, **estimateY** and **A** values depending on the mode that the robot was just in. The code examines the value of **previousMode** to decide which calculation to use. Recall that there are two calculations:

Here is the code for computing the estimate when <u>curving</u> or <u>pivoting</u>:

```
R = WHEEL_BASE*(LeftReading/(RightReading-LeftReading)) + WHEEL_BASE/2;
TD = (RightReading-LeftReading)*WHEEL_RADIUS/WHEEL_BASE/Math.PI*180;
estimateX = estimateX +
        (R*Math.cos(Math.toRadians(TD))*Math.sin(Math.toRadians(A))) +
        (R*Math.cos(Math.toRadians(A))*Math.sin(Math.toRadians(TD))) -
        (R*Math.sin(Math.toRadians(A)));
estimateY = estimateY +
        (R*Math.sin(Math.toRadians(TD))*Math.sin(Math.toRadians(A))) -
        (R*Math.cos(Math.toRadians(A))*Math.cos(Math.toRadians(TD))) +
        (R*Math.cos(Math.toRadians(A)));
A = A + TD;
if (A > 180)  A = A - 360;
if (A < -180) A = 360 + A;
```

Here is the code for computing the estimate when moving <u>straight</u>:

```
estimateX = estimateX + (LeftReading * WHEEL_RADIUS) * Math.cos(Math.toRadians(A));
estimateY = estimateY + (LeftReading * WHEEL_RADIUS) * Math.sin(Math.toRadians(A));
```

And here is the code for computing the estimate when <u>spinning</u>:

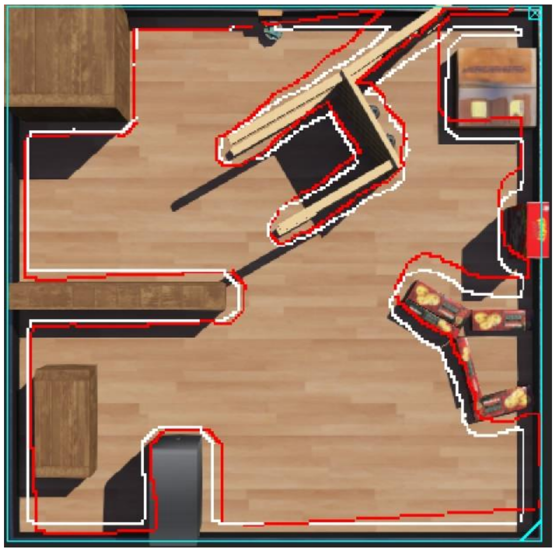
```
A = A+((RightReading-LeftReading)*WHEEL_RADIUS/WHEEL_BASE/Math.PI*180);
```

Due to some rounding errors, it is possible that for the most general formula (i.e., <u>curving</u> and <u>pivoting</u>) that the **leftReading** and **rightReading** encoder values could be the same. If that is the case, you will need to use the <u>straight</u> movement formula for that situation, otherwise you may get some **NaN** (i.e., not a number) results due to the radius calculation being infinite … and your estimates will stop showing up in the app.

Also, as long as the robot is **not** in SPIN_LEFT mode, your code will display this <u>estimate</u> by adding code to the **displayEstimate()** function. (**Note**: we don't add points on the SPIN_LEFT mode because it would create many overlapping points since the coordinate stays the same)

Your initial starting location and angle estimate has been set before the main **while** loop.

The **displayEstimate()** function should display the actual location as well as the estimate by calling the two **TrackerApp** functions mentioned in slide 10 of the notes. Don't forget to negate the **y** value of the <u>actual</u> location (see bottom of slide 9) … but not the estimate … because the estimate is using the correct coordinate system already. Once all is working fine, you should see something VERY similar to what is shown here →



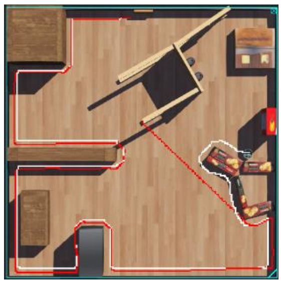
<u>Speedup Tips:</u>

When testing, you will want to press the **fast forward button >>** due to the slow movement of the robot. Make sure to remove any repeating print statements so that your code runs faster. If it is still very slow … try this:

1. Go to the **Tools** menu, then **Preferences...** at the bottom. Select the **OpenGL** Tab.
2. Set **Ambient Occlusion** to **Disabled**.
3. Set **Texture Quality** to **Low**.
4. Check off the two boxes labelled **Disable shadows** and **Disable anti-aliasing**.

<u>Debugging Tips:</u>

- If your robot ends up losing its estimate altogether after cornering the cracker boxes … then you forgot to handle the case where the curved motion formula had equal **leftReading** and **rightReading** encoder values. The image will look something like this →



- If your code seems very inaccurate, it could be that you got something wrong in the formulas or are using them in the wrong spot in the code.

Once the robot reaches the mirror at the top (i.e., where it started from), then press the pause button. Double-click on the **Display** window to get it into a separate window. The image will have a black background … it will not show the objects in the environment nor the robot … just the traces. Save a snapshot of this "separate" display window to a file called: **Snapshot1.png**.

Load up the **MessyRoom** world, also provided for you in the given files. Run your code again. You should notice that the robot's estimated position starts getting skewed (i.e., crooked) at the left horizontal box with some additional inaccuracies around the rounded areas. Take a screen snapshot of the separate **Display** window and save it to a file called: **Snapshot2.png**.

(4) Comment out (but do not delete) the code for calculating the angle. Now, instead of using formulas to calculate the angle, set the angle to the compass reading instead. A function called **getCompassReadingInDegrees()** has been set up for you that you can write your code in. When you are done, run your code in the **MessyRoom** again and you should see that the compass corrects the skew in the angle of the position estimates. Take a screen snapshot of the separate **Display** window and save it to a file called: **Snapshot3.png**.

(5) The compass is giving fairly accurate readings. Adjust your code so that every compass reading gives you readings in multiples of 5 degrees (i.e., -15, -10, -5, 0, 5, 10, 15, etc..). This will be a more realistic compass. Run your code in the **MessyRoom** again and you should see that the compass no longer works as well. Take a screen snapshot of the separate **Display** window and save it to a file called: **Snapshot4.png**.

Submit all of your code including all your .JAVA source code files as well as your **4 Snapshot** files. DO NOT ZIP YOUR FILES. Make sure that your name and student number is in the first comment line of your code.