

# Lab 4 Practical Activity

## Introduction

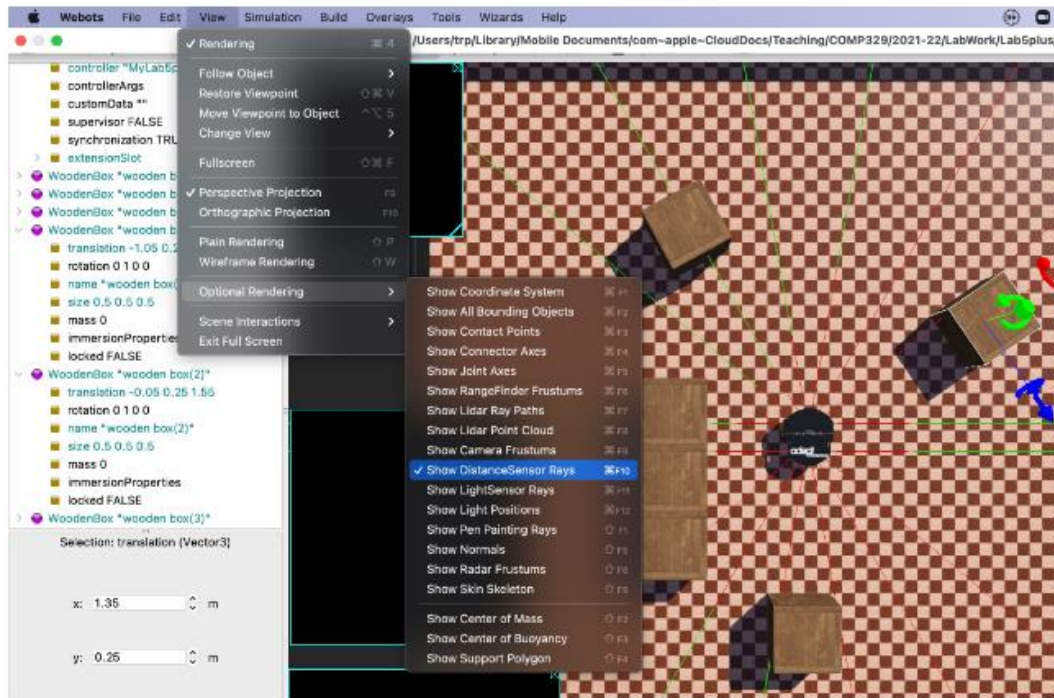
This week, we will look at the use of Sonar sensors, and illustrate in a principled way how the readings can be obtained, as well as demonstrating how to make use of them to perceive the environment. As in previous labs, we'll be reusing the [Pioneer 3-DX, Links to an external site.](#) and illustrating how to locate the positioning of each of the sensors on the robot body, when it is modelled as a cylinder. This lab builds upon the material on PID controllers and the notion of wall following, as well as reinforcing the material covered in past engagement tasks.

## Step 1 - getting started

For this lab, we will want to create a larger arena than before, and exploit the fact that by setting the tile size, we can use that to estimate distances. Create an environment with the following characteristics:

- The size of the floor should be 6m by 6m
- Set the wall height to 0.4m
- Set the floor Tile Size to 0.2 x 0.2 (note each tile has four squares, which will be 10cm square)
- Add a Pioneer 3dx robot and position it in the centre of the arena (i.e. at **x:0m, y:0m, z=0.0975m**)
  - Attach a Display to the extensionSlot of the robot, called **sensorDisplay**, and make it 500x500 pixels large
  - Set the supervisor flag to true - we'll be using a dead reckoning navigator api
  - Add Camera to the extensionSlot of the robot (use a similar approach to adding a display). Set its translation **z** position to 0.21 (i.e. move it above the robot so that the camera is not occluded by the robot)
- Add six boxes taken from the PROTO / objects / factory nodes. Resize each to be 0.5x0.5x0.5 and place at the following locations:
  - Translation **x:-0.95, y:1.55, z:0.25**;  
Rotation **x:0, y:1, z:0, angle 0.5**
  - Translation **x:-1.05, y:0.15, z:0.25**;  
Rotation **x:0, y:1, z:0, angle 0**
  - Translation **x:-1.05, y:-0.35, z:0.25**;  
Rotation **x:0, y:1, z:0, angle 0**
  - Translation **x:-1.05, y:-0.85, z:0.25**;  
Rotation **x:0, y:1, z:0, angle 0**
  - Translation **x:1.35, y:0.85, z:0.25**; Rotation **x:0, y:1, z:0, angle 0.6**
  - Translation **x:-0.05, y:-1.55, z:0.25**;  
Rotation **x:0, y:1, z:0, angle 0**

Don't forget to save this world before testing any code. It is also possible to activate optional rendering of additional information in the 3D view (see image below); a good one for this lab is to enable "**Show DistanceSensor Rays**" to visualise the beam axis of each sonar sensor of the robot. You can also activate "**Show Coordinate System**" to illustrate each of the three dimension (x, y and z) to better understand the placement of each box.



[Download the Controllers, Pose, Navigator and the stub file for our new class](#) ↓, the Pioneer Proximity Sensor Class from the following zip file and move the relevant files and directory into your project. Set the controller property of the robot to refer to your chosen controller.

[Download the Controllers, Pose, Navigator and the stub file for our new class](#) Download Download the Controllers, Pose, Navigator and the stub file for our new class, the Pioneer Proximity Sensor Class from the following zip file and move the relevant files and directory into your project. Set the controller property of the robot to refer to your chosen controller.

In this lab, we are adding a [CameraLinks to an external site.](#) to visualise what is in front of the robot (which will appear in a **Camera Overlay**, in a similar way to that used by the display overlays). It is possible to add a variety of different sensors to the robot and to position them with respect to the robot; and in fact multiple sensors can be added. To initialise the camera, we simply obtain the device object (note that the way this is done is slightly different when using python to that using Java), and enable it by setting the sample period (typically using the basic time step used by the control loop). Add the following code fragment to the controller class, after defining the **timeStep** but before the control loop, and don't forget to import the Camera library file if using Java. The video feed from the camera will appear automatically in the associated camera overlap, but can also be processed

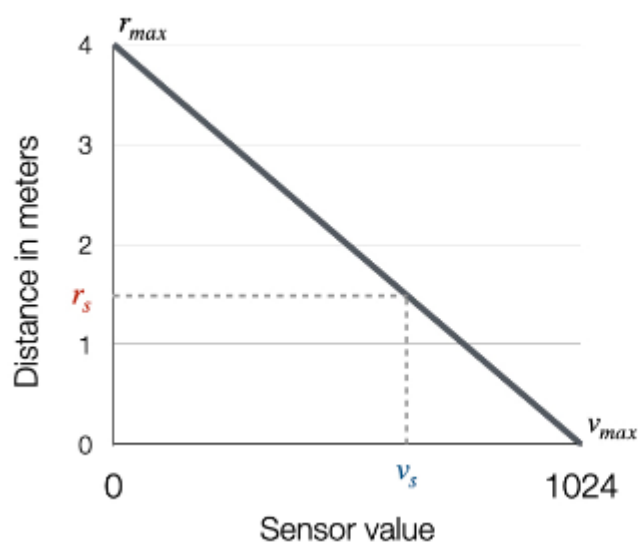
using image analysis techniques. In this lab, we simply display the overlay, and do not retain or use the device object once it has been enabled.

<pre>import com.cyberbotics.webots.controller.Camera a; ... Camera camera = robot.getCamera("camera"); if (camera != null)     camera.enable(timeStep);</pre>	<pre>camera = robot.getDevice('camera') if camera is not None:     camera.enable(timeStep)</pre>
---	--

Run the project and check that everything is working. All going well, you should see the robot slowly turn around anticlockwise, and watch the world rotate from the camera. If you want, you could increase the resolution of the camera overlay by changing the width and height properties in the scene tree.

## Step 2 - understanding the distance sensors in webots

The class [DistanceSensorLinks to an external site.](#) is used to model different range sensors, including infra-red, sonar and laser sensors. It uses a lookup table to define the values returned for different distances, and the standard deviation for a gaussian function that adds noise to the returned value. Other values are then linearly interpolated from this function. Although it is therefore possible to build our own model of the sensor by retrieving the lookup table, what we want here is to know the range of values returned by the sensor, and the operating range (in terms of distance). From this, we can build a simple function that converts one from the other.



Webots Sensor Model

$$r_s = r_{max} - \left( \frac{r_{max}}{v_{max}} \times v_s \right)$$

Example:

$$2.5 = 4 - \left( \frac{4}{1024} \times 640 \right)$$

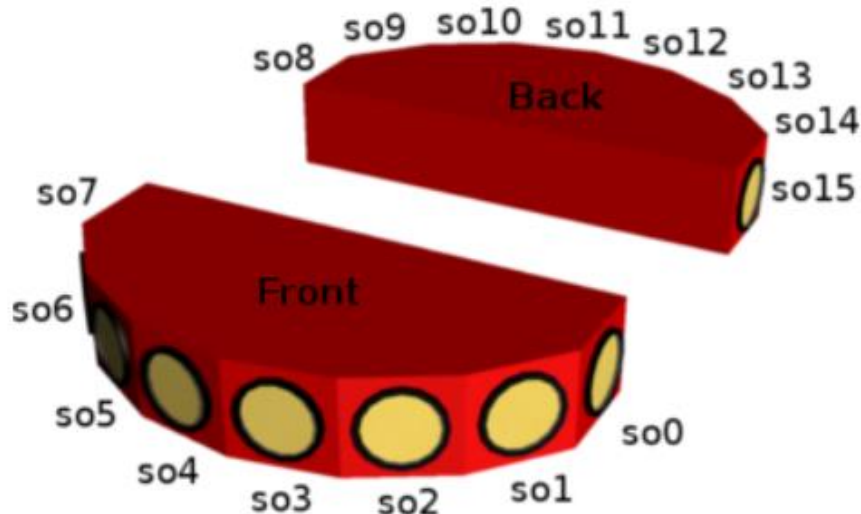
If we can assume that the function is linear and that we can obtain the ranges of both input and output values, we can construct a function that maps  $v_s$  (i.e. a returned value for a sensor) into a distance  $r_s$ . In the above figure, we assume that the lower values for both distance and sensor values are 0 (this is the case given the sensors used, but should always be checked when modelling other sensors in webots by inspecting the lookup table). The gradient can be determined using the formula  $m = \frac{\Delta y}{\Delta x}$ , where in this case  $\Delta y = -r_{max}$  when  $\Delta x = v_{max}$  (note that the gradient is negative), and the intercept  $r = v_{max}$ . Therefore our function is simply  $r_s = r_{max} - \left(\frac{r_{max}}{v_{max}} \times v_s\right)$ . To implement this, we can make use of the **DistanceSensor** methods `getMaxValue()` and `getLookupTable()`. The following code fragment illustrates how, for some sensor `ps[0]`, we can acquire the parameters `r_max` and `v_max` (note, we are not adding this to our project yet):

<pre>// determine max range from lookup table double[] lt = ps[0].getLookupTable(); r_max=0.0; // start with a minimum value for (int i=0; i&lt; lt.length; i++) {     if ((i%3)==0) r_max=lt[i]; } v_max=ps[0].getMaxValue();</pre>	<pre># Determine max range from lookup table lt = self.ps[0].getLookupTable() r_max = 0.0. # start with a minimum value for i in range(len(lt)):     if ((i%3) == 0):         r_max = lt[i] v_max = self.ps[0].getMaxValue()</pre>
--	--

Finally, to find the distance of some object given the return result of the sensor `ps[i]` as follows:

```
d = rMax - (rMax/vMax * ps[i].getValue());
```

In this step we will make use of the range sensors to detect nearby obstacles to determine if the robot should move left or right to avoid them. If you take a look at the details of the [Pioneer 3-DX](#), you will notice that it has 16 sonar sensors positioned around its body, such that the angle between each is 20 degrees, except for the four side sensors (s0, s7, s8, and s15), where the angle is 40 degrees. This is illustrated in the figure below:



However, these sensors are not positioned around a cylinder (which simplifies the way we could model the robot), but rather the robot is shaped such that its length is longer than its width ([as illustrated in the Pioneer-DX data sheetLinks to an external site.](#)). However, for simplicity we will model our robot as a cylinder with the sensors having the bearings (with respect to the robot's pose) as given above. In the notes on Sensors and Perception, we explored one method of determining the location of each sensor by using the rotation matrix, given the bearing of each sensor with respect to the robot pose, assuming that each sensor was a given distance from the robot's centre or origin. We can also use the trigonometric form directly, where

$x = r \cos \theta$  and  $y = r \sin \theta$  for some radius  $r$  representing the distance from the robot's centre, and for the sensor bearing  $\theta$ . In the example below, we assume that we have an array **psAngleDeg** which contains the bearing of each sensor (note in the example below, the bearings are stored as degrees, and thus need converting into radians). We create an array of **Poses** for each sensor, located in the robot's coordinate system:

```
double[] psAngleDeg = { 90, 50, 30, 10, -10, -30, -50, -90,
                       -90, -130, -150, -170, 170, 150, 130, 90};
// Determine the pose (relative to the robot of each of the sensors)
psPose = new Pose[psAngleDeg.length]; // Allocate the pose array
for (int i=0; i< psAngleDeg.length; i++) {
    double theta = Math.toRadians(psAngleDeg[i]);
    psPose[i] = new Pose(Math.cos(theta)*this.radius,
                        Math.sin(theta)*this.radius,
                        theta);
}
```

Note that this does have the consequence that we assume that the sensors so0 and so15 are colocated, as are so7 and so8.

In Step 3 (below) we will make use of the code fragments (above) to create a class that can visualise the readings from different sensors

## Step 3 - creating a class to retrieve sensor readings

For this step we will add the code to our new class, called the **PioneerProxSensors1** (**pioneer\_proxsensor1.py**), which contains all of the relevant information for the sensor. In a larger project, it would make sense to have separate classes for the sensor model and the view itself, but to keep things simple, for now we will construct the one file / class. However, before constructing this class, we will do some initialisation in the controller.

In the main method, we start by setting up the proximity detectors by creating the array containing the distance sensor objects, and then retrieving each object by constructing algorithmically its name. In each case, we then enable the sensor by providing the tilmestep as the sampling period. We then create an array of poses with each pose corresponding to the location of the sensor itself with respect to the robot coordinate system, by making the simplifying solution that the robot itself can be modelled as a circle of a given radius, and that each sensor is positioned along the circumference of that circle (i.e. the perimeter of the robot). The orientation of the pose reflects the orientation of each sensor. Note that, as specified earlier, this has the effect of co-locating two sets of sensors ( **so0** / **so15**, and **so7** / **so8**). Add the following code fragments to the constructor of the Pioneer Proximity Sensor class, after the definition of the radius:



<pre>//----- // set up proximity detectors ps = new DistanceSensor[MAX_NUM_SENSORS];  for (int i = 0; i &lt; MAX_NUM_SENSORS; i++) {     String sensorName = "so" + i;     this.ps[i] = robot.getDistanceSensor(sensorName);     this.ps[i].enable(timestep); }  // The following array determines the orientation of each sensor, based on the // details of the Pioneer Robot Stat sheet. Note that the positions may be slightly // inaccurate as the pioneer is not perfectly round. Also these values are in degrees // and so may require converting to radians. Finally, we assume that the front of the // robot is between so3 and so4. As the angle between these is 20 deg, we assume that // they are 10 deg each from the robot heading double[] psAngleDeg = { 90, 50, 30, 10, -10, -30, -50, -90,                         -90, -130, -150, -170,                         170, 150, 130, 90};  double[] psAngles = new double[MAX_NUM_SENSORS];  //----- // Determine the pose (relative to the robot) of each of the sensors this.psPose = new Pose[psAngleDeg.length]; // Allocate the pose array for (int i=0; i&lt; psAngleDeg.length; i++) {     double theta = Math.toRadians(psAngleDeg[i]);     this.psPose[i] = new Pose(Math.cos(theta)*this.radius,                              Math.sin(theta)*this.radius,                              theta); }</pre>	<pre># ----- - # set up proximity detectors self.ps = [] for i in range(self.MAX_NUM_SENSORS):     sensor_name = 'so' + str(i)     self.ps.append(robot.getDevice(sensor_name))     self.ps[i].enable(timestep)  # The following array determines the orientation of each sensor, based on the # details of the Pioneer Robot Stat sheet. Note that the positions may be slightly # inaccurate as the pioneer is not perfectly round. Also these values are in degrees # and so may require converting to radians. Finally, we assume that the front of the # robot is between so3 and so4. As the angle between these is 20 deg, we assume that # they are 10 deg each from the robot heading  ps_degAngles = [     90, 50, 30, 10, -10, -30, -50, -90,     -90, -130, -150, -170, 170, 150, 130, 90 ] ps_angles = [None] * len(ps_degAngles) for i in range(len(ps_angles)):     ps_angles[i] = math.radians(ps_degAngles[i])  # ----- -- # Determine the poses of each of the sensors self.ps_pose = [] for i in ps_angles:     p = pose.Pose(math.cos(i) * self.radius, math.sin(i) * self.radius, i)     self.ps_pose.append(p)</pre>
---	--

Now we have set up the sensors, we need to determine the max range of a sensor, and create a method for querying each sensor. In step 2, we examined a simple approach to obtain the maximum value returned from the sensor, and the maximum range, given an assumption that the sensor was linear (please note that not all sensors exhibit this linear behaviour). Add the following fragments (based on the code above) to the constructor to obtain these values. It gets the lookup table, and then displays its entries as triples (note the use of the modulo operator), as well as determining the max range. After querying the max range, these parameters are then displayed in the console

<pre>//----- // determine max range from lookup table double[] lt = ps[0].getLookupTable(); System.out.println("Lookup Table has "+lt.length+" entries"); this.maxRange=0.0; for (int i=0; i&lt; lt.length; i++) {     if ((i%3)==0) this.maxRange=lt[i];     System.out.print(" "+lt[i]+",");     if ((i%3)==2) System.out.println("\n"); } this.maxValue=ps[0].getMaxValue();</pre>	<pre># ----- - # Determine max range from lookup table lt = self.ps[0].getLookupTable() print(f"Lookup Table has {len(lt)} entries")  self.max_range = 0.0 for i in range(len(lt)):     if ((i%3) == 0):         self.max_range = lt[i]         print(f" {lt[i]}", end='')     if ((i%3) == 2):         print("") # Newline self.max_value = self.ps[0].getMaxValue()</pre>
---	---

The one thing we still need to do is to create an object of this class within the controller code. Open **MyLab5Controller.java** (or **my\_lab5\_controller.py**), and after the initialisation of the camera device add the following lines to create our object. Note that for the python version, we also have to import the class:

```
PioneerProxSensors1 prox_sensors = new PioneerProxSensors1(robot, "sensor_display", robot_pose);
```

```
import pioneer_proxsensors1 as pps
...
prox_sensors = pps.PioneerProxSensors(robot, "sensor_display", robot_pose);
```

If you compile this, then the following output should appear in the console. Note that each line of the lookup table reports a distance, a return value for that distance, and the standard deviation for a gaussian function that adds noise to the returned value. Given the two lines, we can see that this generates the linear function that we have assumed.

```
Lookup Table has 6 entries
0.0, 1024.0, 0.01,
5.0, 0.0, 0.01,
Max Range: 5.0
```

The next task is to create a method which can query one of the proximity sensors and test this out. We'll also include a couple of methods that can be used to retrieve the max range and number of sensors from the model. Add the following code in the sensor model class under the comment "**Insert public methods here**".

```
public double get_maxRange() {
    return maxRange;
}

public int get_number_of_sensors() {
    return MAX_NUM_SENSORS;
}

public double get_value(int sensorID) {
    if (sensorID < MAX_NUM_SENSORS)
        return this.maxRange - (this.maxRange/this.max_value * this.ps[sensorID].getValue());
    else
        System.err.println("Out of range error in get SensorValue");
    return this.maxRange;
}
```

```
def get_maxRange(self):
    return self.max_range

def get_number_of_sensors(self):
    return len(self.ps)

def get_value(self, i):
    if (i < len(self.ps)):
        return self.max_range - (self.max_range/self.max_value * self.ps[i].getValue())
    else:
        print("Out of range error in get_value")
    return None
```

We can now test out the return value of one of the sensors as the robot rotates, to ensure things are working. The eventual aim is to visualise the sensors graphically, but we will do this in the next step. For now, modify the controller code by adding the following lines to the code inside the control loop, immediately after the while statement (prior to changing the variable **state**).

```
// Testing out the front proximity sensors
System.out.println(String.format("Sensor readings for so3 | so4: %.3f | %.3f",
    prox_sensors.get_value(3), prox_sensors.get_value(4)));
```

```
# Testing out the front proximity sensors
print(f"Sensor readings for so3 | so4: "+
    f"{prox_sensors.get_value(3):.3f} | {prox_sensors.get_value(4):.3f}")
```

Run the code and verify that the sensor readings look realistic. In cases where the beam hits an obstacle at an acute angle, the value for max\_range (i.e. 5.0) will be returned.

## Step 4 - extending the class to visualise the sensor readings

In this step, we will make use of the display to render a depiction of the robot, such that its orientation reflects that of the robot itself, with "wedges" drawn

to indicate the range of each sensor, reflecting the sensor readings. The constructor of the Pioneer Proximity Sensor class takes three arguments, two of which have already been used. However, the third (`display_name`) has yet to be used. This argument provides the name of some display (or is empty if no display is used) which we should use. Therefore, the last responsibility of the constructor is to obtain a display object corresponding to this name, and determine the display's dimensions. From these dimensions, we can then determine a scale factor that can be used to map any graphics we want to display to the display itself, without necessarily knowing the size of the display.

In previous labs, we did something similar to map the arena size to that of the display. However, in this case, we want to ensure that the display is able to display some depiction of the robot (i.e. a circle with header indicator) and the wedges (representing `max_range`) within its bounds. Thus, in the worst case, if all of the sensors were returning max range, then the display needs to be big enough for two wedges plus the full diameter of the robot. This we use this size to determine the scale factor that will later be used when visualising the wedges. Add the following code fragments to the end of the constructor of the sensor class:

<pre> this.display = this.robot.getDisplay(display_name); if (this.display != null) {     this.device_width = this.display.getWidth();     this.device_height = this.display.getHeight(); } this.scaleFactor = Math.min(this.device_width, this.device_height) / (2 * (this.maxRange + this.radius)); } else {     this.device_width = 0;     this.device_height = 0;     this.scaleFactor = 0.0; } </pre>	<pre> self.display = robot.getDevice(display_name) if self.display is not None:     self.device_width = self.display.getWidth()     self.device_height = self.display.getHeight()     self.scalefactor = min(self.device_width, self.device_height) / (2 * (self.max_range + self.radius)) </pre>
--	---

The sensor class has a number of pre-defined methods to assist in completing the lab. In previous labs, we have already seen the methods **mapx**, **mapy**, and **scale**. We have also seen the method **set\_pose**, used to update the pose of the class, and the method **paint** is used to draw a depiction of the robot on the display. In this class, the robot depiction is drawn in the centre of the display, and does not move, other than to rotate as the robot itself rotates. We can test this, by adding the following code fragment to the control loop of the controller class, at the top of the while loop (e.g. just after we test the sensor readings from the previous step. The code fragments obtain the current pose (as before we are using the supervisor mode to get the real pose), pass this through to the sensor object, and then we re-paint the display.

<pre> robot_pose = nav.get_real_pose(); prox_sensors.set_pose(robot_pose); prox_sensors.paint(); // Render sensor Display </pre>	<pre> robot_pose = nav.get_real_pose() prox_sensors.set_pose(robot_pose) prox_sensors.paint() # Render sensor Display </pre>
--	--



Try to compile the code and check that it works. You should now see the robot drawn in the middle of the display, with the heading depiction which rotates as the robot rotates. Note how small it is - this is due to need to also display the sensor wedges (which are coming soon).

The final two helper methods that are already in the sensor class source use the rotation matrix to calculate new coordinates when rotating around the origin by theta ( $\theta$ ) radians:

```
private double rotX(double x, double y, double theta) {
    return Math.cos(theta)*x - Math.sin(theta)*y;
}
private double rotY(double x, double y, double theta) {
    return Math.sin(theta)*x + Math.cos(theta)*y;
}
```

```
def rotx(self, x, y, theta):
    return math.cos(theta) * x - math.sin(theta) * y
def roty(self, x, y, theta):
    return math.sin(theta) * x + math.cos(theta) * y
```

We can use these to complete the **paint** method to visualise the sensor readings using a wedge or triangle anchored at the location of each sensor, with respect to the sensor's bearing. To display each triangle, we will use the **Display** method **fillPolygon()** which takes three arguments; an array of x values, an array of y values, and the number of values to consider in each array. The values define a path of vertices for the polygon itself which is then filled. If the first vertex coordinates are not the same as the last ones, then the path is automatically closed. We will use this mechanism to define a set of coordinates for the three vertices of each triangle, and an additional set of coordinates to display the distance as characters.

Create the following arrays for the coordinates after drawing the robot in the **paint** method. Note that we will create two sets of arrays for each axis - one for the coordinates with respect to the robot coordinate system, and one for coordinates on the display. The latter coordinates will be generated using the scaling and rotation helper methods described above.

```
// For each sensor, get the value of the sensor
// and display the distance
// Note that we assume a array of four values.
// Each polygon only requires
// three, but the fourth is used to determine
// a position for the text
int[] xArc = {0, 0, 0, 0};
int[] yArc = {0, 0, 0, 0};
double[] xd = {0.0, 0.0, 0.0, 0.0};
double[] yd = {0.0, 0.0, 0.0, 0.0};
double d; // distance measured
```

```
# For each sensor, get the value of the sensor
# and display the distance
# Note that we assume an array of three values.
# The label will be
# managed seperately
xarc = [0, 0, 0]
yarc = [0, 0, 0]
x = [0.0, 0.0, 0.0]
y = [0.0, 0.0, 0.0]
```

Start by creating a loop for each sensor at the end of the **paint** method. Note that we will use the function described in step 2 to map the return value of each sensor into a distance value. The triangle will represent the sensor cone from each sensor, based on some field of view. The angle of the field of view was selected to look aesthetically good, and doesn't accurately reflect the actual field of view of the sensor. In this case, an angle of  $\frac{\pi}{18}$  rads either side of the sensor axis will be used. Thus, the first coordinate corresponds to the sensor position; the second coordinate is the far point to the right of the field of view, and the third coordinate is the far point to the left of the field of view. A fourth coordinate (in the Java version) is also generated, along the sensor axis but just beyond the polygon to display the sensor range value (note that the values 0.3 are used - these were found empirically).

```

// Draw triangles of 2*Math.PI/36.0 either side
either side of the sensor orientation
for (int i = 0; i < ps.length ; i++) {

    d = this.get_value(i);

    xd[0] = psPose[i].getX();
    xd[1] = psPose[i].getX() + (d * Math.cos(psPo
se[i].getTheta())-(Math.PI/18.0));
    xd[2] = psPose[i].getX() + (d * Math.cos(psPo
se[i].getTheta()+(Math.PI/18.0));
    xd[3] = psPose[i].getX() + ((d+0.3) * Math.co
s(psPose[i].getTheta()));

    yd[0] = psPose[i].getY();
    yd[1] = psPose[i].getY() + (d * Math.sin(psPo
se[i].getTheta())-(Math.PI/18.0));
    yd[2] = psPose[i].getY() + (d * Math.sin(psPo
se[i].getTheta()+(Math.PI/18.0));
    yd[3] = psPose[i].getY() + ((d+0.3) * Math.si
n(psPose[i].getTheta()));

```

```

# Draw triangles of 2*Math.PI/36.0 either s
ide
# of the sensor orientation
for i in range(len(self.ps)):
    d = self.get_value(i)

    p = self.ps_pose[i]

    x[0] = p.x
    x[1] = p.x + (d * math.cos(p.theta - ma
th.pi/18.0))
    x[2] = p.x + (d * math.cos(p.theta + ma
th.pi/18.0))

    y[0] = p.y
    y[1] = p.y + (d * math.sin(p.theta - ma
th.pi/18.0))
    y[2] = p.y + (d * math.sin(p.theta + ma
th.pi/18.0))

```

Note that these are the coordinates based on the robot coordinate system. They need to be mapped into the coordinates for the display, taking into account the orientation of the robot.

```

// Need to rotate each point using the rotati
on matrix and the robot orientation
for (int j=0; j<4; j++) {
    xArc[j] = mapX(rotX(xd[j], yd[j], this.robo
t_pose.getTheta()));
    yArc[j] = mapY(rotY(xd[j], yd[j], this.robo
t_pose.getTheta()));
}

// Only use the first three points for the po
lygon
this.display.setColor(DARKGREY);
this.display.fillPolygon(xArc, yArc, 3);

```

```

for j in range(len(x)):
    xarc[j] = self.mapx(self.rotx(x[j],
y[j], self.robot_pose.theta))
    yarc[j] = self.mapy(self.rotx(x[j],
y[j], self.robot_pose.theta))

self.display.setColor(self.DARKGREY)
self.display.fillPolygon(xarc, yarc)

```

Finally, we draw the polygon (based on the first three of the four coordinates in the Java version), and draw the text of the range (using the fourth set of coordinates). Note that the code fragment for Java belongs to the same loop and draws the label adjacent to the end of the wedge. In contrast, the python version creates a separate loop (note how the indentation changes):

```

// Use the fourth point for the distance stri
ng, with a slight offset as the
// coordinates are for the location of the bo
ttom left of the string
this.display.drawText(String.format("%.02f",
d), xArc[3]-10, yArc[3]);
}

```

```

for i in range(len(self.ps)):
    # for display clarity, skip sensors 8 &
    15 as they overla with 7 and 0
    if not (i%8):
        continue

    d = self.get_value(i)

    p = self.ps_pose[i]

    x_label = p.x + ((d+self.LABEL_OFFSET)
* math.cos(p.theta))
    y_label = p.y + ((d+self.LABEL_OFFSET)
* math.sin(p.theta))
    x1 = self.mapx(self.rotx(x_label, y_lab
el, self.robot_pose.theta))
    y1 = self.mapy(self.rotx(x_label, y_lab
el, self.robot_pose.theta))

    self.display.setColor(self.DARKGREY)
    self.display.drawText('{0:.2f}'.format
(d), x1, y1)

```

Compile and test this out. You should see something similar to the view below (note that some measurements will vary slightly due to the gaussian noise). If everything works, you can go back to the top of the control loop in the controller class and remove



between the two wheels causing the robot to arc to the left or the right (proportionally to the magnitude of the control). This controller will be determined by the PID controller (**pid**), which responds to the set point and the error, and returns a controller that we use with our **set\_velocity** method. Whilst this can be used to follow walls and slightly irregular shapes, we also need a control logic to handle the case where we enter the inside of a corner (such that we need to turn to avoid the oncoming wall) or when we pass a corner and have to turn to follow the wall. This is handled by the **follow\_wall** method (which also determines if we are following a wall on the left or right).

### ***Setting the velocity with a base and control***

The principle behind this method is simple; set the velocity of both wheels to be the base velocity, but then adjust the velocity of the two wheels given the control. The one danger with a naive approach to this algorithm is that if there are no bounds placed on the control, then the speed settings for the motors can exceed the permissible settings, resulting in warnings that the requested velocity exceeds the maximum velocity:

```
WARNING: Pioneer3dx (PROTO) > DEF RIGHT_WHEEL HingeJoint > RotationalMotor: The requested velocity 31.348 exceeds 'maxVelocity' = 12.3.
```

Therefore we need to mitigate against that, by changing the overall speed given the maximum permissible speed (which can be obtained using the device method **getMaxVelocity**). The approach taken in the code below checks to see if either motor should run at a faster velocity than the max velocity, and if such scales down the velocities of both motors equally. Thus the **magnitude** of any rotation is maintained, but the **rate** of rotation (with respect to time) is reduced.

Add the following method to the navigation code:

```
public void set_velocity(double base, double control) {
    // base gives the velocity of the wheels in m/s
    // control is an adjustment on the main velocity

    double base_av = (base/this.WHEEL_RADIUS);
    double lv = base_av;
    double rv = base_av;

    if (control != 0) {
        double control_av = (control/this.WHEEL_RADIUS);

        // Check if we exceed max velocity and compensate
        double correction = 1;
        lv = base_av - control_av;
        rv = base_av + control_av;

        if (lv > this.max_vel) {
            correction = this.max_vel / lv;
            lv = lv * correction;
            rv = rv * correction;
        }

        if (rv > this.max_vel) {
            correction = this.max_vel / rv;
            lv = lv * correction;
            rv = rv * correction;
        }
    }

    this.left_motor.setVelocity(lv);
    this.right_motor.setVelocity(rv);
}
```

```
def set_velocity(self, base, control=0):
    # base gives the velocity of the wheels in m/s
    # control is an adjustment on the main velocity

    base_av = (base/self.WHEEL_RADIUS)

    if (control != 0):
        control_av = (control/self.WHEEL_RADIUS)

        # Check if we exceed max velocity and compensate
        correction = 1
        lv = base_av - control_av
        rv = base_av + control_av

        if (lv > self.max_vel):
            correction = self.max_vel / lv
            lv = lv * correction
            rv = rv * correction

        if (rv > self.max_vel):
            correction = self.max_vel / rv
            lv = lv * correction
            rv = rv * correction

    else:
        lv = rv = base_av

    self.left_motor.setVelocity(lv)
    self.right_motor.setVelocity(rv)
```

If we try to compile this, we will get an error as we have not defined the max velocity. This can be obtained as a constant (for efficiency) within the constructor. Add the following lines in the constructor of the navigation class, after enabling the motors (for Java programmers, remember that you will also need to create an instance variable of type double for max\_vel).

```
this.max_vel = this.left_motor.getMaxVelocity()
- 0.1; // Fudge: just under max vel
```

```
self.max_vel = self.left_motor.getMaxVelocity() - 0.1 # Fudge: just under max vel
```

We can now add our PID controller method in the navigation class (e.g. after the definition of the **set\_velocity** method).

```
private double pid(double error) {
    double kp = 0.6; // proportional weight (may need tuning)
    double kd = 3.0; // differential weight (may need tuning)
    double ki = 0.0; // integral weight (may need tuning)

    double prop = error;
    double diff = error - this.prev_error;
    this.total_error += error;

    double control = (kp * prop) + (ki * this.total_error) + (kd * diff);
    this.prev_error = error;

    return control;
}
```

```
def pid(self, error):
    kp = 0.6 # proportional weight (may need tuning)
    kd = 3.0 # differential weight (may need tuning)
    ki = 0.0 # integral weight (may need tuning)

    prop = error
    diff = error - self.prev_error
    self.total_error += error
    control = (kp * prop) + (ki * self.total_error) + (kd * diff)
    self.prev_error = error

    return control
```



Check to see if there are any errors in your code (this is mainly for the Java programmers, as the python will not be interpreted until it is called). We will not see any new behaviour as yet; this is going to happen after the final task, which is to add the wall following logic.

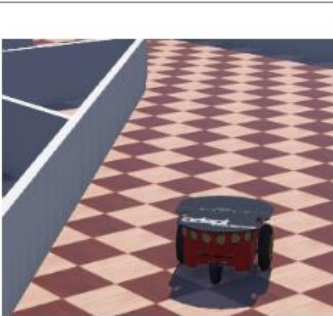


### ***Logic to follow the walls***

Wall following is a useful tool to assist in exploring an environment, and is a key component in many obstacle avoidance algorithms such as the Bug1 and Bug2 algorithms. A simple way of doing this is to make use of the proximity sensors to determine our proximity to the walls, and using this within a wall following logic. However, there are three cases to consider when writing a wall following algorithm:

1. **The robot needs to use a proximity sensor to ensure that it stays a certain distance from the wall.** This is typically done using a feedback controller, such as the **PID controller** that we have just implemented. This not only monitors the change between the current and desired distance to the wall, but also smooths the control behaviour to avoid oscillatory behaviour.
2. **A strategy is needed when a wall ends (e.g. if it angles to the left, or if it is a free standing wall).** A simple strategy is to move in an arc to the left until a new wall is detected, and then to follow that new wall.
3. **An obstacle detection algorithm is needed to detect any obstacle in front of the robot.** This is important if it approaches an inside corner; in which case the robot would have to turn to the right, but without colliding with the wall.

We have three cases we need to consider (do not use the code below, as the full method is given later):

The navigation object has been written to use these three simple behaviours, as follows (you don't need to write any code, as it has been written):

	<p>The standard case is where we are running parallel to the wall. Here, we want to make use of the PID controller to maintain a consistent distance between the robot and the wall. Note that we need to determine a distance value (<b>wall_dist</b>) between the wall and the robot.</p> <pre># Running approximately parallel to the wall if (wall_dist &lt; self.prox_sensors.max_range):     error = wall_dist - set_point     control = self.pid(error)     # adjust for right wall     self.set_velocity(robot_linearvelocity, control*direction_coeff)</pre>
	<p>This is the case where the wall has ended and the robot is about to move into space. In this case we need to rotate towards where the wall was in the hope of rejoining it.</p> <pre>else:     # No wall, so turn     self.set_velocity(robot_linearvelocity, 0.08*direction_coeff)</pre>
	<p>The final case is where the robot moves towards an inside corner. The robot can detect this because there is an obstacle in front of it. To deal with this case, the robot will need to turn away from the wall it is following.</p> <pre># Approaching a wall, turn if(min(self.prox_sensors.get_value(1),     self.prox_sensors.get_value(2),     self.prox_sensors.get_value(3),     self.prox_sensors.get_value(4),     self.prox_sensors.get_value(5),     self.prox_sensors.get_value(6)) &lt; set_point):     self.set_velocity(robot_linearvelocity/3, -0.2*direction_coef f)</pre>

In order to implement this method, we will need to ensure that the navigator class has an instance of the proximity sensor class, in order to sample the proximity sensors. Start by opening the controller class and modifying the constructor for the navigator. Note that we currently instantiate that before creating the instance of the proximity sensor, so we need to swap the order of this. This is illustrated in the code below (where the old navigation class constructor has been commented out).

<pre>// PioneerNav2 nav = new PioneerNav2(robot, robot_pose); PioneerProxSensors1 prox_sensors = new PioneerProxSensors1(robot, "sensor_display", robot_pose); PioneerNav2 nav = new PioneerNav2(robot, robot_pose, prox_sensors);</pre>	<pre># nav = pn.PioneerNavigation(robot, robot_pose) prox_sensors = pps.PioneerProxSensors(robot, "sensor_display", robot_pose) nav = pn.PioneerNavigation(robot, robot_pose, prox_sensors)</pre>
--	---

We can now modify the constructor itself to take new arguments and store these for later use. Note that we need to add an additional entry to our list of enum states for our wall following action; in which case add the entry **FOLLOW\_WALL** to the end (the python version will need the value 3). Remember that in the Java version we also need to declare the instance variable **prox\_sensors** with the type **PioneerProxSensors1** (this is not shown below).

```

public PioneerNav2(Supervisor robot, Pose init_
pose, PioneerProxSensors1 ps) {
    this.prox_sensors = ps; // reference to proxim
ity sensors

```

```

def __init__(self, robot, init_pose, ps):
    self.prox_sensors = ps # reference to
proximity sensors

```

We can now implement our wall following method - note that this is just one of a variety of approaches for implementing our control logic. The Boolean variable **right** determines whether or not the robot should follow the left or right wall. For the cases where the robot has passed an "outside" corner, or is approaching an "inside" corner, then main difference is whether the robot turns left or right. This is modelled by simply changing whether the control adjustment is positive or negative by setting the appropriate value to the **direction\_coefficient** variable. For wall following, this is determined by the choice of sensor values. Note that we currently use the two sensors attached to the sides (either on the left or right) of the robot. Also the rate of rotation for the corners has been hand tuned; this could be done using another PID controller with the front sensors.

```

public void follow_wall(double robot_linearveloci
ty, double set_point, boolean right) {
    int direction_coeff = 1;
    double error;
    double control;
    double wall_dist;

    if (right) direction_coeff = -1; // invert the
values for the control

    if (Math.min(this.prox_sensors.get_value(1),
        Math.min(this.prox_sensors.get_value(2),
            Math.min(this.prox_sensors.get_value
(3),
                Math.min(this.prox_sensors.get_value
(4),
                    Math.min(this.prox_sensors.get_valu
e(5),
                        this.prox_sensors.get_value
(6)))))) < set_point)
        this.set_velocity(robot_linearvelocity/3, -0.
2*direction_coeff);

    else {
        if (!right) wall_dist = Math.min(this.prox_se
nsors.get_value(1),
            this.prox_se
nsors.get_value(0));
        else wall_dist = Math.min(this.prox_sensors.g
et_value(7),
            this.prox_sensors.g
et_value(8));
        // Running approximately parallel to the wall
        if (wall_dist < this.prox_sensors.get_maxRang
e()) {
            error = wall_dist - set_point;
            control = this.pid(error);
            // adjust for right wall
            this.set_velocity(robot_linearvelocity, con
trol*direction_coeff);
        } else {
            // No wall, so turn
            this.set_velocity(robot_linearvelocity, 0.0
8*direction_coeff);
        }
    }
    this.state = MoveState.FOLLOW_WALL;
}

```

```

def follow_wall(self, robot_linearvelocity, set
_point, right=False):
    if right:
        direction_coeff = -1
    else:
        direction_coeff = 1

    # Approaching a wall, turn
    if (min(self.prox_sensors.get_value(1),
        self.prox_sensors.get_value(2),
        self.prox_sensors.get_value(3),
        self.prox_sensors.get_value(4),
        self.prox_sensors.get_value(5),
        self.prox_sensors.get_value(6)) < se
t_point):
        self.set_velocity(robot_linearvelocity/
3, -0.2*direction_coeff)
    else:
        if not right:
            wall_dist = min(self.prox_sensors.g
et_value(1),
                self.prox_sensors.g
et_value(0))
            else:
                wall_dist = min(self.prox_sensors.g
et_value(7),
                    self.prox_sensors.g
et_value(8))

        # Running approximately parallel to the
wall
        if (wall_dist < self.prox_sensors.max_r
ange):
            error = wall_dist - set_point
            control = self.pid(error)
            # adjust for right wall
            self.set_velocity(robot_linearveloc
ity, control*direction_coeff)
        else:
            # No wall, so turn
            self.set_velocity(robot_linearveloc
ity, 0.08*direction_coeff)
        self.state = MoveState.FOLLOW_WALL

```

The last thing we need to do is to add the case for **FOLLOW\_WALL** in the control loop. Before we do this, modify the schedule, which is currently set to the action arc, and replace this with **FOLLOW\_WALL**. Then add the following action to the if-then-else construct in the control loop:

<pre> if (state == PioneerNav2.MoveState.FOLLOW_ WALL) {     target_time = 0; // always refresh!     nav.follow_wall(robot_velocity, 0.2, fals e); } else </pre>	<pre> elif (nav.state == MoveState.FOLLOW_WAL L):     target_time = 0 # always refresh!     nav.follow_wall(robot_velocity, 0. 2, False) </pre>
--	---

If you compile and run the code, the robot should start following the wall. Try changing the boolean value for right to see it follow the wall on the right.

## Optional Tasks

If you try out this solution and run it for a while, you will notice that there are many cases where it fails. The approach taken here is simple, but highly susceptible to:

- The layout of the arena
- The PID controller settings
- The rate at which the robot turns when approaching corners
- The set point (i.e. distance) used by the PID controllers
- The choice of sensors
- The way in which the sensor readings are combined
- The refresh rate used when updating the wheels (currently done during every control loop update)

Even when a set of parameters have been selected, there is still the possibility of failure, and therefore failure recovery is an important aspect of robot control not handled here. Despite these challenges, there is still scope for improving the basic algorithm discussed here, as well as using it for problems such as obstacle avoidance (e.g. using the Bug1 or Bug2 algorithms). Explore how you could improve on the basic wall following algorithm, for example, making better use of turning when approaching corners, possibly by calculating the rate of rotation given the set\_point and the forward sensor readings!