Lab 3 Practical Activity

Introduction

The aim of Week 3's Lab is to explore how to get a robot to move around the ICR and to understand the difference between the angular velocity around the ICR, the angular velocity of the wheels, and their relationships with linear velocity. We will also make use of the Supervisor mode to get details of the true robot pose, as well as demonstrating the possibility of adding additional devices to the robot. Furthermore, we will see how we can use the robot to infer its properties - specifically to find out the actual length of the wheel axis as well as the actual size of the wheels. In this lab, we will be reusing using Adept's Pioneer 3-DXLinks to an external site. robot in addition to the environment created in Lab 2.

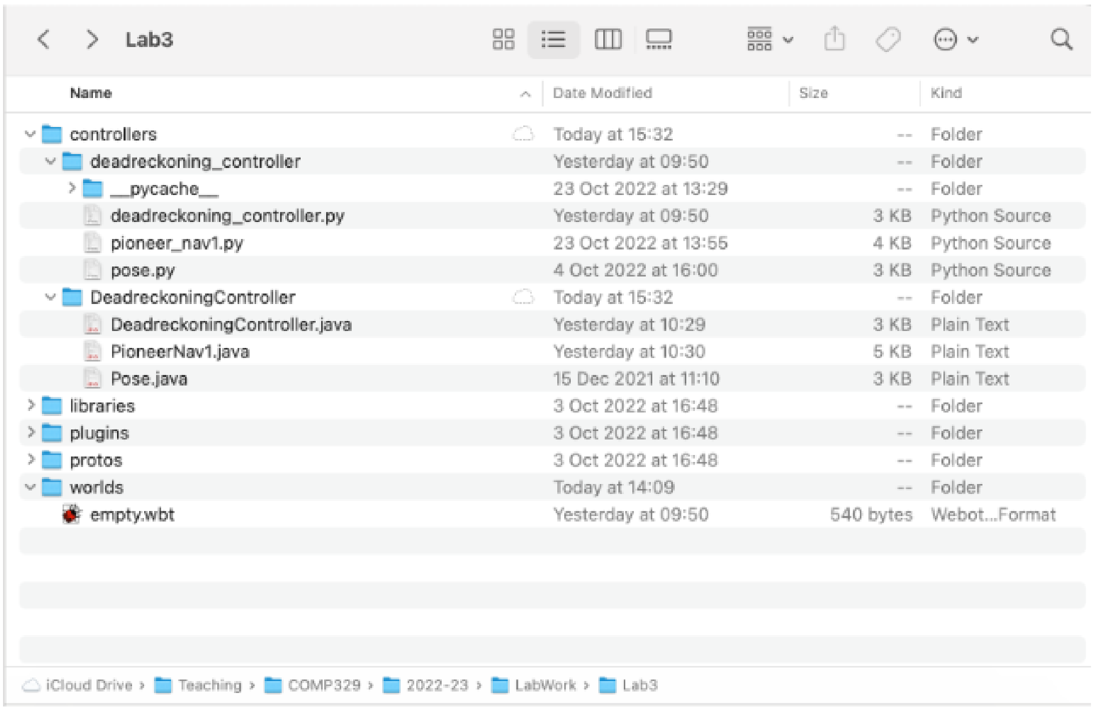
Step 1 - getting started

This lab builds upon the world created in Lab 2. You can either take the world resulting from that lab or rebuild it, using the following (condensed instructions):

Creating the World: Create a new world (by making use of the **New Project Directory** Wizard in Webots), and change the size of the floor to 4m by 4m, with a wall height of 0.4m. Add to your world a Pioneer 3dx robot, and position it so that its translation is as follows – **`x:-0.5m, y: -1.0m z:0.0975m`**. Note that you can do this by moving the robot using shift and left clicking the robot, or by first selecting the robot from the Scene Tree (i.e. the panel on the left of the screen by default) and then selecting the **translation** parameters.

Remember to save the world as you change the parameters and add the robot.

Download the zip file containing the controller and accompanying files Download Download the zip file containing the controller and accompanying files, and add the relevant directory (for the language of your choice) to the controllers directory. The webots platform manages each of the controller implementations in different directories, where multiple files may be included and compiled to form your final controller code. The diagram below illustrates the file tree of this Webots project captured using a Mac OSX Finder window, showing both Java and Python versions



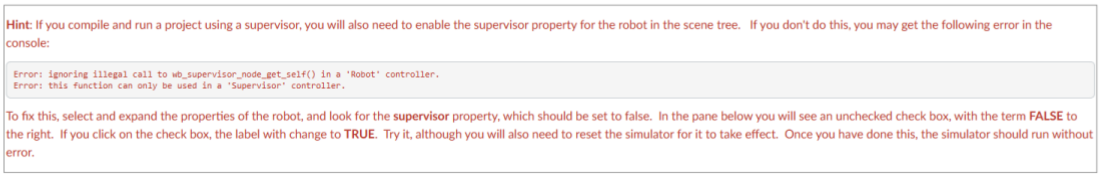
Name	Date Modified	Size	Kind
controllers	Today at 15:32	--	Folder
deadreckoning_controller	Yesterday at 09:50	--	Folder
__pycache__	23 Oct 2022 at 13:29	--	Folder
deadreckoning_controller.py	Yesterday at 09:50	3 KB	Python Source
pioneer_nav1.py	23 Oct 2022 at 13:55	4 KB	Python Source
pose.py	4 Oct 2022 at 16:00	3 KB	Python Source
DeadreckoningController	Today at 15:32	--	Folder
DeadreckoningController.java	Yesterday at 10:29	3 KB	Plain Text
PioneerNav1.java	Yesterday at 10:30	5 KB	Plain Text
Pose.java	15 Dec 2021 at 11:10	3 KB	Plain Text
libraries	3 Oct 2022 at 16:48	--	Folder
plugins	3 Oct 2022 at 16:48	--	Folder
protos	3 Oct 2022 at 16:48	--	Folder
worlds	Today at 14:09	--	Folder
empty.wbt	Yesterday at 09:50	540 bytes	Webot...Format

Lab3
iCloud Drive › Teaching › COMP329 › 2022-23 › LabWork › Lab3

Step 2 - asking Webots for the robot position

One of the main aims of using the Bayes Filter (introduced in Week 2) is to allow the robot to know where in the world it is, by estimating its new position based on Odometry, and then confirming this through sensor data. When using simulators, in can be useful to obtain the **actual** position of the robot, to compare with the robot's perceived position. This is possible through the *supervisor*.

The supervisor is not a specific entity within Webots, but rather a special type of privileged objectLinks to an external site. that is only available if the **Supervisor Mode** of a robot is set to true. This object that can be used to query the state of the various assets regarding the robot itself that would normally not be available to a robot in the real world. In this step, we will look at how to use the Supervisor mode to query the robot state and in particular, obtain the real position of the robot within the world coordinate system. Take a look at the controller - you will notice that instead of creating an instance of the class `Robot()`, we have created an instance of the class **`Supervisor()`**. This is a special form of the Robot class that permits manipulation of the world, as well as gaining parameters of the environment (if you want to explore this more - take a look at the Webots Tutorial 8 on the SupervisorLinks to an external site.).



Hint: If you compile and run a project using a supervisor, you will also need to enable the supervisor property for the robot in the scene tree. If you don't do this, you may get the following error in the console:

```
Error: ignoring illegal call to wb_supervisor_node_get_self() in a 'Robot' controller.
Error: this function can only be used in a 'Supervisor' controller.
```

To fix this, select and expand the properties of the robot, and look for the **supervisor** property, which should be set to false. In the pane below you will see an unchecked check box, with the term **FALSE** to the right. If you click on the check box, the label with change to **TRUE**. Try it, although you will also need to reset the simulator for it to take effect. Once you have done this, the simulator should run without error.

Take a look at the code for the class PioneerNav1 in Java (or pioneer_nav1 class in python). In this class, we include code to handle navigation for the robot (we'll be adding code to this class later in the tutorial). Note that the class definitions include various instance variables (at least in Java) prior to the definition of the constructor, which takes a reference to the robot, and initialises the motors. Note that it also creates a reference to a **`robot_node`**; this will be used to obtain details of the robot position. The method **`get_real_pose()`** queries the Scene Tree to obtain the current location of the robot in the world coordinate system, as well as a 3D rotation matrix which is used to obtain the orientation. The method returns an instance of a pose object (see below) with respect to the world coordinate system.

The other file that you downloaded was for a class called Pose (or pose). This class allows us to maintain a representation of the location of the robot (in x, y coordinates) and an orientation (as theta) in radians. Furthermore, different accessor methods and constructors have been defined to simply its use. For example, the **`setTheta()`** method ensures that the stored value lies within the range −pai,...,pai. Note that the Java implementation provides a set of getters and setters, whereas the python follows the convention of providing direct access to the values themselves. The class also provides a method for pretty-printing the pose value; we'll see how this can be exploited later in the tutorial.

Run the code (remember to compile the Java) and look at the console. You may have to also set the supervisor parameter on the robot. Note that you will see the output looking something like:

```
Action: Configuring... True Pose: <-0.503, -1.000, -0.000>
Action: Configuring... True Pose: <-0.503, -1.000, 0.000>
Action: Configuring... True Pose: <-0.503, -1.000, 0.000>
Action: Configuring... True Pose: <-0.503, -1.000, 0.000>
```

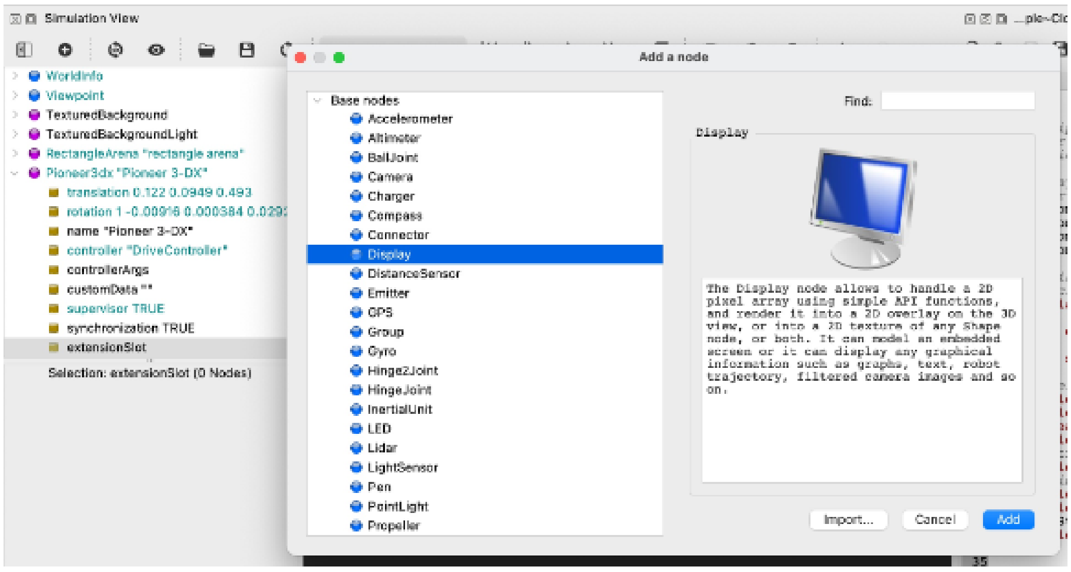
Return to the controller class to see what is happening. You will notice that there is an array called **`schedule`** with two elements. For this tutorial we make use of an enumerated type with different instructions (a similar thing was used in the previous tutorial), but in this case we will have a schedule that can be repeated (unless the **`STOP`** action is used). The current code starts with the action CONFIGURE, and runs indefinitely; we will use this to calculate the wheel radius and axel length later in the tutorial. For now, it does nothing other than updates the console with the action and the pose.

If you move the robot using the mouse cursor (remember to SHIFT-click on the robot, or you could modify its position using the translation property of the robot in the Scene Tree), you will see the values of the translation property reflected in the Real Position printed in the console. In the next step we will add a display device to the robot, which we can use to display data without sending it to the console.

Step 3 - adding different devices to a robot

It is often desirable to add additional devices to one of the proto-defined robots (such as the [Pioneer 3-DXLinks to an external site.]() robot that we are using in this lab). For example, one may want to add additional sensors to the robot, or actuators such as a robot arm onto a mobile platform, or even an end effector to the end joint of an arm. One of the most useful devices is a display device, which is then simulated as a window within the simulator. In this step we will look at how to add a screen to the robot, set its dimensions, and then write to the display.

Go to the Scene Tree and open up the properties of the robot. Select the extensionSlot, and then add a new object to this node (remember you can do this with the black circle icon with the add symbol above the Scene Tree - we also use this to add a robot to the world). When adding a new node, open up the Basic Nodes list, and look for the display device, as illustrated by the diagram below:



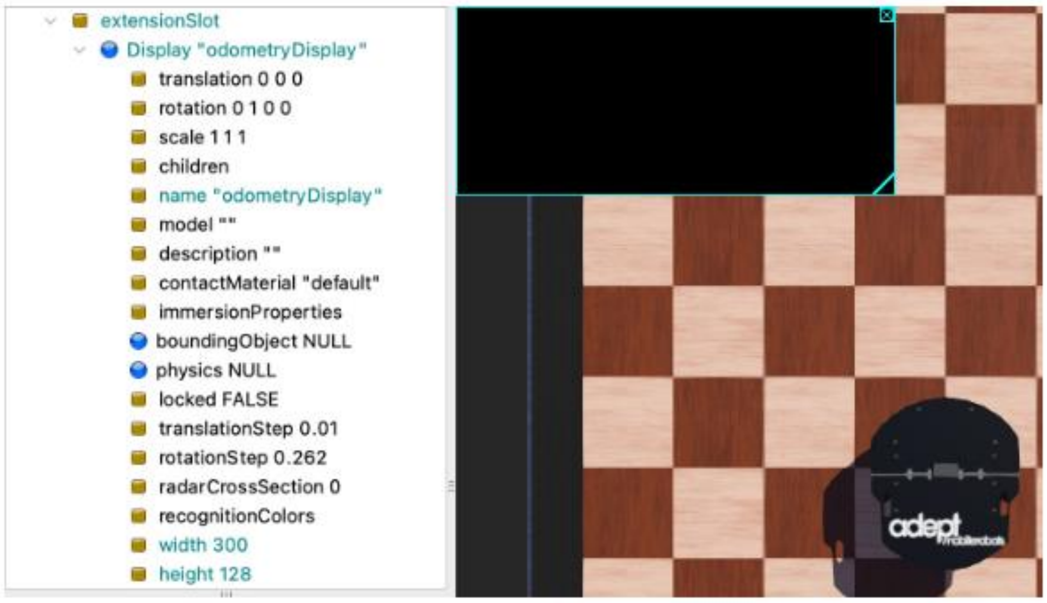
```
The Display node allows to handle a 2D
pixel array using simple API functions,
and render it into a 2D overlay on the 3D
view, or into a 2D texture of any Shape
node, or both. It can model an embedded
screen or it can display any graphical
information such as graphs, text, robot
trajectory, filtered camera images and so
on.
```

You should now see a new node in the scene tree under the extensionSlot of type **Display** with the name **"display"**. Its is possible to add multiple displays to a robot, provided that each has a unique name by which we obtain the devices reference. In addition, a small window should appear in the 3D view where the robot normally appears. If you resize this window, it will simply change the size that the window is rendered in the display, however, it doesn't change the resolution of the display device itself.

Start by changing the objects name - this is how we will reference the object. In the display properties, select the node name, and change the string associated with it. The default value should be **"display"** so lets change this to **"odometryDisplay"**. Once this is done you should also notice that the display node will also change its name to **Display "odometryDisplay"**.

Remember to save the world so that we don't loose these changes.

The next thing we will do is to change the size of the display, by updating the **width** and **height** properties of the display, which define the pixel size of these two dimensions. Good values for now would be **300** and **128** respectively. Note that if you change the size parameters of the display, you will also need to reload the world (also don't forget to save the changes first).



```
extensionSlot
  Display "odometryDisplay"
    translation 0 0 0
    rotation 0 1 0 0
    scale 1 1 1
    children
    name "odometryDisplay"
    model ""
    description ""
    contactMaterial "default"
    immersionProperties
    boundingObject NULL
    physics NULL
    locked FALSE
    translationStep 0.01
    rotationStep 0.262
    radarCrossSection 0
    recognitionColors
    width 300
    height 128
```

We can now write to this display, rather than writing to the console. The relevant classes have already been imported where necessary, so all we need to do is to set up the device. To do this, we use the same approach as setting up the motors or sensors, i.e. we get an object representing the display from the robot, and then we can write to it (to find out more about the methods one can use, look at the Webot Display API in JavaLinks to an external site. or PythonLinks to an external site. and is worth checking out if you want to do interesting things with the display, such as generating graphics, or modifying font parameters). In your code, after setting the schedule, add the following lines to obtain the display object. Note that we identify the display added by its name - in this case, "*odometryDisplay*". This allows for multiple displays, each being indexed using different names.

```java
    // set up display
    Display odometry_display = robot.getDi
splay("odometryDisplay");
```

```python
    # set up the display
    odometry_display = robot.getDevice('od
ometryDisplay')
```

We can now write to the display. For this example, we will reset the display on each frame reset, and then use two different font sizes in the displayed data.

In the code below, we check if we have a device defined (this is always good practice), and then we draw a rectangle whose dimensions are the same as the device itself . Once the display has been cleared, we can then write to the display, by setting the colour of the next set of draw actions (to Black), setting the font type and size, and then using the method **drawText** to position a string on the display. Note that the last two parameters of the **drawText** method refer to the **x** and **y** position of the start of the string.

```java
    if (odometry_display != null) {
      odometry_display.setColor(0xFFFFFF);      // White
      odometry_display.fillRectangle(0, 0,
              odometry_display.getWidth(),
              odometry_display.getHeight());

      odometry_display.setColor(0x000000);      // Black
      odometry_display.setFont("Arial", 18, true);  // font size = 18, with antialiasing
      odometry_display.drawText("Robot State", 1, 1);

      odometry_display.setFont("Arial", 12, true);
      if (display_action != "")
        odometry_display.drawText(display_action, 1, 30);

      Pose true_pose = nav.get_real_pose();

      odometry_display.drawText("True Pose: "+true_pose, 1, 50);
    }

    //Pose true_pose = nav.get_real_pose();
    //System.out.println("Action: " + display_action + " \t" + "True Pose: "+true_pose);
```

```python
        if odometry_display is not None:
            odometry_display.setColor(0xFFFFFF)      # White
            odometry_display.fillRectangle(0,0,
                    odometry_display.getWidth(),
                    odometry_display.getHeight())

            odometry_display.setColor(0x000000)      # Black
            odometry_display.setFont("Arial", 18, True)      # font size = 18, with antialiasing
            odometry_display.drawText("Robot State", 1, 1)

            odometry_display.setFont("Arial", 12, True)
            if (display_action != ""):
                odometry_display.drawText(display_action, 1, 30)

            true_pose = nav.get_real_pose()

            odometry_display.drawText(f"True Pose: {true_pose}", 1, 50)

        #true_pose = nav.get_real_pose()
        #print(f"Action: {display_action} \tTrue Pose: {true_pose}")
```

Note that in the above code, we have commented out the original lines which obtained the real pose, and printed this in the console. This is no longer needed as this information is presented in the display. If we now compile the code, we should see the real position of the robot appear in the display. If you move the robot around (by SHIFT-click dragging the robot around the environment, or modifying the translation properties of the robot in the Scene Tree), then this will update the pose information.

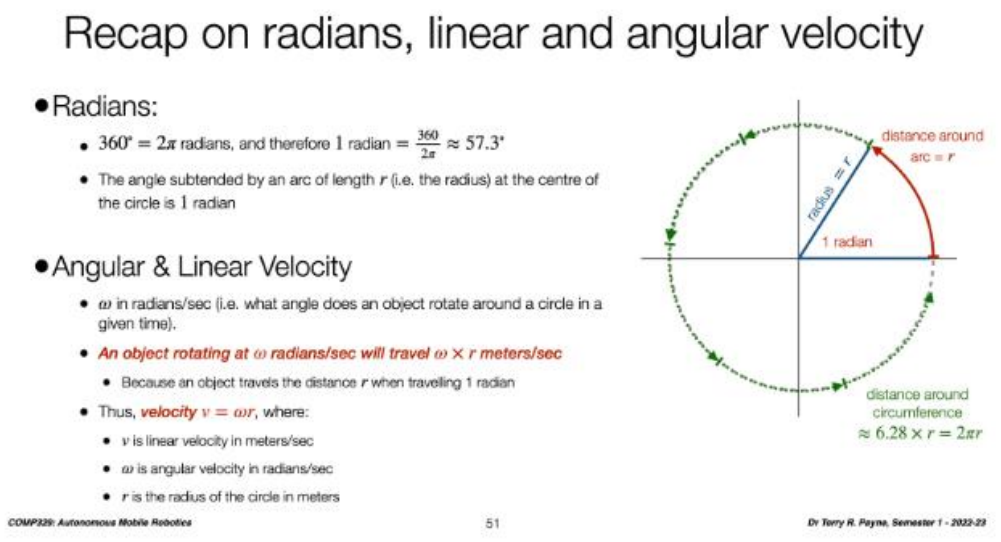
We now need to write some navigation code!

Step 4 - moving the robot forward

We have so far seen in previous labs how to move the robot forward or to rotate. In this lab, we look at this in a more principled way by adding code to a navigator class. We will start by defining a method to move the robot forward for a given distance, travelling at a given linear velocity. The method will set the velocity of both wheels so that the linear velocity obtained is that given by the arguments, and the return value of the method is the time required for the robot to travel the desired distance. Again, we will look at these calculations from first principles:

1. **How long should the robot travel forward?** Recall that d=v×t, and thus t=d/v, where t corresponds to time, d is distance, and v is linear velocity (note that here, we can assume the units to be seconds, meters, and meters/sec respectively). We know that the velocity is given by the argument robot_linearvelocity. The time is therefore simply **`target_time = abs(target_dist/robot_linearvelocity)`**. However, as the robot monitors time in milliseconds, we will return the time as milliseconds, and thus the return time is **`1000.0*target_time`**. Note also that we need to ensure that the time is positive, as we could also pass a negative velocity to drive the robot backwards.
2. **What is the angular velocity of the wheels?** For this we need to remember the formula for converting linear to angular velocity, v=w×r, and thus w=v/r, where v is linear velocity, w is the angular velocity, and r corresponds to a radius. In this case, we know the forward linear velocity of the robot, so we can calculate the angular velocity of the wheels, given their radius.

At this point - it may be worth reflecting on the relationship between angular momentum, radians and the radius. The definition of a radian is generally given as *"... the angle subtended from the centre of a circle which intercepts an arc equal in length to the radius of the circle ..."Links to an external site.*, or another way to think about it, if something has an angular velocity of 1 **radian** per second around an arc, then it will move the distance of 1 **radius** in that second around an arc. See the slide below:



Recap on radians, linear and angular velocity

- Radians:
  - $360^\circ = 2\pi$ radians, and therefore 1 radian $= \frac{360}{2\pi} \approx 57.3^\circ$
  - The angle subtended by an arc of length *r* (i.e. the radius) at the centre of the circle is 1 radian
- Angular & Linear Velocity
  - ω in radians/sec (i.e. what angle does an object rotate around a circle in a given time).
  - ***An object rotating at ω radians/sec will travel $\omega \times r$ meters/sec***
    - Because an object travels the distance *r* when travelling 1 radian
  - Thus, ***velocity*** $v = \omega r$, where:
    - v is linear velocity in meters/sec
    - ω is angular velocity in radians/sec
    - r is the radius of the circle in meters

distance around arc = r
radius = r
1 radian
distance around circumference $\approx 6.28 \times r = 2\pi r$

COMP329: Autonomous Mobile Robotics 51 Dr Terry R. Payne, Semester 1 - 2022-23

In this case, as we know the radius of the wheel, and the desired linear velocity, we can calculate the angular velocity as **`wheel_av = (robot_linearvelocity/this.WHEEL_RADIUS)`**, and set the velocities of both wheels to the resulting value.

Finally, the navigation code maintains the current action, or state, of the robot navigation, using the variable state. Therefore, we will need to set this to the state MoveState.FORWARD. Add the following code to the navigator class (**PioneerNav1.java** or **pioneer_nav1.py**):

```java
public int forward(double target_dist, dou
ble robot_linearvelocity) {
    double wheel_av = (robot_linearvelocit
y/this.WHEEL_RADIUS);
    double target_time = Math.abs(target_d
ist/robot_linearvelocity);

    this.left_motor.setVelocity(wheel_av);
    this.right_motor.setVelocity(wheel_a
v);
    this.state = MoveState.FORWARD;

    // return target_time as millisecs

    return (int) (1000.0*target_time);
  }
```

```python
    def forward(self, target_dist, robot_l
inearvelocity):
        wheel_av = (robot_linearvelocity/s
elf.WHEEL_RADIUS)
        target_time = abs(target_dist/robo
t_linearvelocity)

        self.left_motor.setVelocity(wheel_
av)
        self.right_motor.setVelocity(wheel
_av)
        self.state = MoveState.FORWARD

        # return target_time as millisecs

        return 1000.0*target_time
```

In the controller, we need to make a couple of additional changes. First, we will need to change the agenda in the main controller. Comment out the original schedule and add the following:

```java
PioneerNav1.MoveState[] schedule = { Pionee
rNav1.MoveState.FORWARD, PioneerNav1.MoveSt
ate.STOP };
```

```python
    schedule = [ MoveState.FORWARD, MoveSta
te.STOP ]
```

We also need to extend the state machine, to call the new method, and pass parameters. This approach is quite simple, and doesn't consider the parameters in the schedule, but could be extended by creating an action class, which includes not just the type of move, but parameters, and then a schedule could be generated (by hand or with a path planner) consisting of these items. The current code in the main loop of the controller handles just the states **CONFIGURE** and **STOP**. We need to extend this to handle the **FORWARD** state. In the code fragment below, we will also move the robot forward by 0.5m at 0.3m/s (i.e. the value of the variable `robot_velocity`). We also need to track the time elapsed. Replace the current test for the state (for **CONFIGURE** or **STOP**) with the following code fragments:

```java
      if (state == PioneerNav1.MoveState.CO
NFIGURE) {
        // Special case for checking robot
parameters
       display_action = nav.configure_check
_parameters(timeStep);
      } else if (time_elapsed > target_tim
e) {
        time_elapsed = 0;

        // select next action in schedule i
f not stopped
        schedule_index = (schedule_index +
1) % schedule.length;
        state = schedule[schedule_index];

        if (state == PioneerNav1.MoveState.
FORWARD) {
          target_time = nav.forward(0.5, ro
bot_velocity);
          display_action = "Forward Action:
0.5m";
        } else
        if (state == PioneerNav1.MoveState.
CONFIGURE) {
          display_action = "Determine Wheel
/ Axel Parameters";
        } else
        if (state == PioneerNav1.MoveState.
STOP) {
          nav.stop();
          display_action = "Stop for 1 minu
te";
          target_time = 60 * 1000; // This
doesn't really stop, but pauses for 1 minut
e
        }
      } else
        time_elapsed += timeStep;    // Inc
rement by the time state
```

```python
        if (nav.state == MoveState.CONFIGUR
E):
            # Special case for checking rob
ot parameters
            display_action = nav.configure_
check_parameters(timestep)

        elif (time_elapsed > target_time):
            time_elapsed = 0

            # select next action in schedul
e if not stopped
            schedule_index = (schedule_inde
x +1) % len(schedule)
            nav.state = schedule[schedule_i
ndex]

            if (nav.state == MoveState.FORW
ARD):
                target_time = nav.forward
(0.5, robot_velocity)
                display_action = "Forward A
ction: 0.5m"
            elif (nav.state == MoveState.CO
NFIGURE):
                display_action = "Determine
Wheel / Axel Parameters"
            elif (nav.state == MoveState.ST
OP):
                nav.stop()
                display_action = "Stop for
1 minute"
                target_time = 60 * 1000 # T
his doesn't really stop, but pauses for 1 m
inute
        else:
            time_elapsed += timestep    # I
ncrement by the time state
```

Note that we have added some code to change the current action in the schedule when the target time is reached. We have also added an extra state for **CONFIGURE**. We will discuss this later. Compile the code and see what happens.

Step 5 - moving the robot around an arc

Unlike the code developed in the previous lab that rotates the robot around its centre, in this method we want the robot to rotate around some Instantaneous Centre of Rotation (ICR), given the following parameters:

- `icr_r` - the distance (in meters) from the centre of the arc to the centre of the robot;
- `icr_omega` - the velocity (in radians/sec) at which the robot travels around the arc;
- `icr_angle` - the angle (in radians) through which the robot should travel to its destination.

It is also important that we can specify `icr_r=0` to allow the robot to rotate around its centre! The challenge here is to calculate the angular velocity of each of the wheels to be able to achieve this. In the **forward()** method, we used the equation $d = v \times t$ to define the relationship between distance (in meters), linear velocity (in meters/sec) and time (in sec). An analogous equation $\theta = \omega \times t$, where θ corresponds to the **distance around the circle** (in radians), ω corresponds to the **angular velocity** (i.e. distance in radians per sec), and again, t gives us the time in seconds. Therefore, we can re-arrange this to get $t = \frac{\theta}{\omega}$ and thus `target_time = abs(icr_angle / icr_omega)`. Note that we remove the sign of any calculation, as we always want the time to be positive. This is because that if the robot is turning clockwise, then the angular velocity will be negative, around an ICR whose radius is also negative, Therefore, we need to ensure that the target time is always positive.

From the notes on locomotion, we can now calculate the rotation of the robots left and right wheels around the ICR, based on the distance `icr_r` from the robot centre to the ICR. Both equations can be derived from the equation $v = \omega \times r$, which allows us to calculate the linear velocities of each wheel. Given that r (i.e. the parameter `icr_r`) is the radius of the circle that the robot should follow, the radius that each wheel follows will be offset by their distance to the centre of the robot (i.e. half the length of the axel). Hence we have the following equations:

• $v_l = \omega(R - \frac{l}{2})$ • $v_r = \omega(R + \frac{l}{2})$	• vl = icr_omega * (icr_r - (this.AXEL_LENGTH / 2)) • vr = icr_omega * (icr_r + (this.AXEL_LENGTH / 2))

Now that we have the linear velocities of each wheel, we can use the same approach used in the **forward()** method to determine the angular velocity of the wheels, using the equation $\omega_{wheel} = \frac{v}{r_{wheel}}$. Note that this time we specify the angular momentum of a wheel as ω_{wheel} to avoid confusion with the angular momentum of the robot around the ICR; likewise the radius r_{wheel} is specified to differentiate between the radius of the arc around the ICR and the wheel radius.

This gives us the final method for **arc()**:

```java
public int arc(double icr_angle, double icr_r, double icr_omega) {
    double target_time = Math.abs(icr_angle / icr_omega);

    // Calculate each wheel velocity around ICR
    double vl = icr_omega * (icr_r - (this.AXEL_LENGTH / 2));
    double vr = icr_omega * (icr_r + (this.AXEL_LENGTH / 2));

    double leftwheel_av = (vl/this.WHEEL_RADIUS);
    double rightwheel_av = (vr/this.WHEEL_RADIUS);

    this.left_motor.setVelocity(leftwheel_av);
    this.right_motor.setVelocity(rightwheel_av);
    this.state = MoveState.ARC;

    // return target_time as millisecs

    return (int) (1000.0*target_time);
  }
```

```python
    def arc(self, icr_angle, icr_r, icr_omega):
        target_time = abs(icr_angle / icr_omega)

        # Calculate each wheel velocity around ICR
        vl = icr_omega * (icr_r - (self.AXEL_LENGTH / 2))
        vr = icr_omega * (icr_r + (self.AXEL_LENGTH / 2))

        leftwheel_av = (vl/self.WHEEL_RADIUS)

        rightwheel_av = (vr/self.WHEEL_RADIUS)

        self.left_motor.setVelocity(leftwheel_av)
        self.right_motor.setVelocity(rightwheel_av)
        self.state = MoveState.ARC

        # return target_time as millisecs

        return 1000.0*target_time
```

Again, we need to make a couple of minor changes to the controller. This time, replace the action **STOP** with the action **ARC** in the schedule, so that the the robot moves forward and then follows an arc. The code for the scheduler has been designed to repeat the schedule when all of the actions have been repeated. Thus, with this schedule, the robot will move forward and then follow an arc, and then repeat these two actions.

We also need to add an extra condition to the the state machine in the main loop of the controller. The following two code fragments state that during the state **ARC**, the robot should follow an arc that is 0.7m away from the robot centre, travelling around an arc of $\frac{\pi}{2}$ radians (i.e. 90°) at the angular velocity of 0.3 rads/sec (i.e. `robot_velocity`).

```java
        if (state == PioneerNav1.MoveState.ARC) {
          target_time = nav.arc(Math.PI/2.0, 0.7, robot_velocity);
          display_action = "Arc Action around an ICR 0.7 away";
        } else
```

```python
        elif (nav.state == MoveState.ARC):
            target_time = nav.arc(math.pi/2.0, 0.7, robot_velocity)
            display_action = "Arc Action around an ICR 0.7 away"
```

Compile the code and test out your new controller. As the robot moves around the path, you will notice that the robot rapidly accumulates errors. However, could this be due to having the correct parameters for the robots wheel size and axel length? Is there a way of determining these experimentally?

Step 6 - determining the wheel size and axel length (optional)

In the previous lab, we were given parameters for the wheel radius (**`WHEEL_RADIUS = 0.0975`**) and axel length (**`AXEL_LENGTH =`**

`0.31`). However it would be good if we could determine these experimentally. In this optional step, we will look at one approach whereby we could measure these parameters.

Calculating the Axel length: The idea is that if the robot moves around an arc that is half the axel away from the robot's centre (i.e. one wheel is kept stationary), then the other wheel will rotate around a circle who's radius is the length of the distance between the two wheels (i.e. the axel length). Conversely, as the centre of the robot is half way along the axel, then the centre of the robot will rotate around a circle who's radius is half the axel length.

As we can monitor the real position of the robot, can we can use one axis (e.g the x axis of the robots pose) to monitor the maximum and minimum location of the robot along that dimension, to determine the width (I.e. diameter) of that circle? The answer is yes!

Calculating the Wheel radius: If we know the axel length, then as the robot rotates around the inner wheel on the ICR, then the outer wheel will travel around a circle, who's radius is the length of the axel. From this, we can calculate the circumference of the circle. If we calculate ***the time for the wheel to travel around a full circle***, and ***we have the angular velocity of the wheel itself***, then we can calculate the ***wheel radius***.

- Set the angular velocity of the ***inner*** wheel as 0 rad/sec - i.e. the wheel is stationary.
- Set the angular velocity of the ***outer*** wheel as 2π rad/sec - i.e. the wheel will do a full rotation every second.
  - This value ω_{wheel} is held by the variable `velocity` (***note this is short for angular velocity here***!)
- The outer wheel follows a path around a circle of radius r (the axel length) around the ICR. As the circumference of a circle is given as $2\pi r$, we know:
  - the full distance travelled by the right (outer) wheel is $d = 2\pi r$ (held by the variable `distance`)
- By tracking when the pose of the robot periodically passes through a specific orientation (i.e. after the robot has travelled all of the way around the circle centred at the ICR), we will know that the robot has travelled all the way around a circle (i.e. it has travelled the distance d).
  - The time taken (in milliseconds) to travel d is held by the variable `config_timer`. Thus the time in seconds t is `config_timer / 1000`.
- From this we know the outer wheel's forward velocity is given by $v_{wheel} = \frac{d}{t}$, or `distance / (config_timer / 1000)`.
- We also know that the we know the outer wheel's forward velocity is a function of the angular velocity and the wheel radius; i.e. $v_{wheel} = \omega_{wheel} \times r_{wheel}$, which can be re-arranged as $r_{wheel} = \frac{v_{wheel}}{\omega_{wheel}}$
  - Combining these two equations gives us the wheel radius $r_{wheel} = \frac{\frac{d}{t}}{\omega_{wheel}} = \frac{d}{\omega_{wheel} \times t}$, or `distance / (velocity * (config_timer / 1000))`.

```
wheel_radius = distance / (this.velocity *
(this.config_timer / 1000));
```

```
wheel_radius = distance / (self.velocity *
(self.config_timer / 1000))
```

We can put this together through the creation of two new methods; one to initialise the parameters, and one to make the final calculations. These will be used by the action state **CONFIGURE**.

The following code tracks the maximum and minimum positions along the x axis of the robot, to determine the diameter of the circle that the centre of the robot passes through as it rotates with the right wheel rotating at some angular velocity. It also sets a timer to determine the time to travel the full circumference (monitored using the robot's pose). Therefore, the method `configure_initialise_parameters` initialises the relevant parameters (taking as an argument the wheel angular velocity:

```
  public void configure_initialise_paramet
ers(double wheel_omega) {
    // This rotates the robot about one wh
eel, and monitors the time and distance
    // taken to rotate around a circle
    Pose pose = this.get_real_pose();
    this.velocity = wheel_omega;
    this.config_max_x = pose.getX();
    this.config_min_x = pose.getX();
    this.config_timer = 0;
    this.config_prev_theta = pose.getTheta
();

    this.left_motor.setVelocity(0.0);
    this.right_motor.setVelocity(this.velo
city);
    this.state = MoveState.CONFIGURE;
  }
```

```
    def configure_initialise_parameters(se
lf, wheel_omega):
        # This rotates the robot about one
wheel, and monitors the time and distance
        # taken to rotate around a circle
        pose = self.get_real_pose()
        self.velocity = wheel_omega
        self.config_max_x = pose.x
        self.config_min_x = pose.x
        self.config_timer = 0
        self.config_prev_theta = pose.thet
a

        self.left_motor.setVelocity(0)
        self.right_motor.setVelocity(self.
velocity)
        self.state = MoveState.CONFIGURE
```

Add this to the navigation class (note that a stub method already exists, and can be replaced).

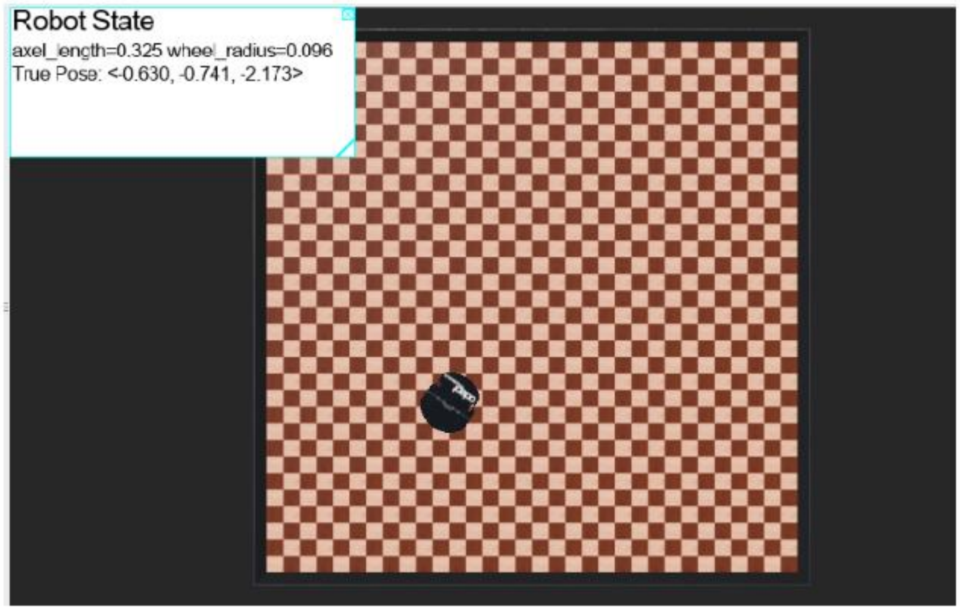
The next fragment defines the method that tracks the progress of the robot, and calculates both the axel length and the wheel radius once it complete a full circuit of the circumference. It takes as an argument the current tilmestep (i.e. number of milliseconds between each update) and returns a string that can display the final values of the axel length and wheel radius.

```java
public String configure_check_parameters(d
ouble timestep) {
    Pose pose = this.get_real_pose();
    double axel_length = this.config_max_x
- this.config_min_x; // represents a circl
e travelled by center of robot

    if ((this.config_prev_theta < 0) && (p
ose.getTheta() >= 0.0)) {
      // We have just passed a heading of
0
      double distance = axel_length * 2 *
Math.PI; // i.e. circumference around icr
      double wheel_radius = distance / (th
is.velocity * (this.config_timer / 1000));
      this.configure_str = String.format
("axel_length=%.3f wheel_radius=%.3f", axe
l_length, wheel_radius);
      this.configure_initialise_parameters
(this.velocity);
      this.config_timer = 0;
    } else {
      this.config_max_x = Math.max(this.co
nfig_max_x, pose.getX());
      this.config_min_x = Math.min(this.co
nfig_min_x, pose.getX());
    }
    this.config_timer += timestep;
    this.config_prev_theta = pose.getTheta
();
    return this.configure_str;
  }
```

```python
    def configure_check_parameters(self, t
imestep):
        pose = self.get_real_pose()
        axel_length = self.config_max_x -
self.config_min_x # represents a circle tr
avelled by center of robot

        if (self.config_prev_theta < 0) an
d (pose.theta >= 0.0):
            # We have just passed a headin
g of 0
            distance = axel_length * 2 * m
ath.pi # i.e. circumference around icr
            wheel_radius = distance / (sel
f.velocity * (self.config_timer / 1000))
            self.configure_str = f"axel_le
ngth={axel_length:.3f} wheel_radius={wheel
_radius:.3f}"
            self.configure_initialise_para
meters(self.velocity)
            self.config_timer = 0
        else:
            self.config_max_x = max(self.c
onfig_max_x, pose.x)
            self.config_min_x = min(self.c
onfig_min_x, pose.x)

        self.config_timer += timestep

        self.config_prev_theta = pose.thet
a
        return self.configure_str
```

This should also be added to the navigation class (again note that a stub method already exists, and can be replaced).

The final thing required is to update the robot schedule to the single action **CONFIGURE**, and test out the code. If everything works the robot will spin around the left wheel, and periodically the results of the experiment will be displayed:



```
Robot State
axel_length=0.325 wheel_radius=0.096
True Pose: <-0.630, -0.741, -2.173>
```

Note that these values are similar to the ones defined in the class you originally downloaded, but not exactly the same. Try noting these new values and updating the constants specified in the navigator class. Revert the schedule to the one used with the actions **FORWARD** and **ARC**. See if they have improved the resulting paths.

A full version of the files generated from this lab are available from a zip file for both python and Java Download are available from a zip file for both python and Java.

Optional Tasks

This lab used a simple mechanism for creating a schedule, based purely on the actions themselves. Explore the possibility of creating an action class where each instance consists of the action and its own parameters. Use this to create more sophisticated paths that the robot could follow.