(ECE183DA/MAE162D)

Prof. Ankur Mehta, Prof. Jacob Rosen
mehtank@ucla.edu, jacobrosen@ucla.edu

Joint Lab Assignment 3
Due 3pm Friday, Feb. 26, 2021

1 Lab Overview

1.1 Objectives and Goals

In the previous lab you verified that the kinematics models you derived for your coded simulations match those of a CAD simulation environment to give confidence that they will work in reality. However, we cannot simulate and experiment with all the designs and algorithms we need to with just the tools we have learned so far. Here, we introduce a new tool, Webots, a physics simulator, that can bridge our gap between MAE and ECE, or CAD 3D models and codes.

You will setup Webots for both the Segway and Paperbot, and drive in a rectangle arena to measure sensor readings. Optionally, ECE students will investigate into software architecture that allows us to switch between Webots and our analytical simulator as necessary.

You will be working in your project teams, and will be responsible as a team for dividing the various tasks of this project between all members. Your grade will be based both on team and individual performance. All students are expected to individually learn the materials covered in sections 1 through 3.

1.2 General Aims

Prototyping and experimenting with hardware demands significant resources such as time, money and efforts. Simulating our robot dynamics and testing our algorithms are essential parts of the design process, which were done individually in Joint Lab 1 and 2. Here, finally, we integrate both design and algorithm simulations using a physics simulator, Webots.

1.3 Specific Aims

Through this lab,

- Both MAE and ECE students are to demonstrate understanding of Webots, a physics simulator
- Both MAE and ECE students are to demonstrate understanding of how to operate our robots in a simple environment
- Both MAE and ECE students are to demonstrate understanding of how to read and convert simulated sensor outputs
- Both MAE and ECE students are to demonstrate comprehensive quantitative analysis of simulation results
- Both MAE and ECE students are to understand software architecture of Webots and our analytical simulator controllers

1.4 Deliverables/Method of Reporting

For this joint lab, your team is required to submit:

- A write-up presenting your Webots simulation results by 3pm Friday, Feb. 26, 2021.

Both MAE and ECE students will also create a well documented git repository for your simulations containing all your code and data. Include in your write-up links to your code repository / documentation, as well as a complete list of references you've used and in what manner.

For all deliverables, you will be assessed on both the clarity and completeness of your content. Submit pdfs on CCLE. Submissions that are late will be accepted with a 50% grade penalty.

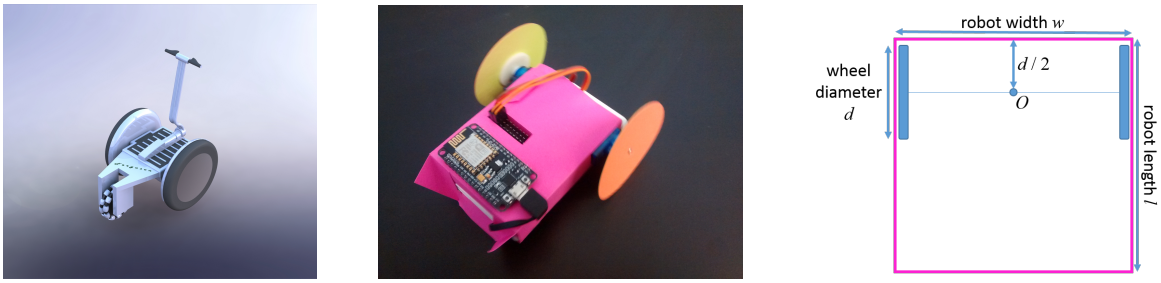

Figure 1: Two wheeled tank-drive robots with symbolic dimensions.

2 Joint Assignment Descriptions

2.1 System overview

Both MAE and ECE students modeled the same two-wheeled robots in Joint Lab 1, similar to the Segway and Paperbot shown in Fig. 1. These robots have two wheels of diameter $d \approx 502mm/50mm$ (Segway/Paperbot), separated by a distance $w \approx 530mm/90mm$. Each wheel is direct driven from a continuous rotation servo. They drag a tail (or castor wheel in the case of the Segway) for stability, that contacts the ground at a distance $l \approx 682mm/75mm$ behind their front edge. The position of each of these robots in the environment is defined relative to their centerpoints O.

More details about Paperbot is available at the git repository: `https://git.uclalemur.com/mehtank/paperbot`

2.2 Actuation model

Each wheel is powered independently by a continuous rotation servo —part number FS90R for the Paperbot, an unspecified industrial continuous rotation servo for the Segway— with the angular velocity of the wheel controlled by a PWM signal from the microcontroller. The control input to the robot hardware will be the PWM values you send to each wheel, for a total of 2 input variables. This allows the robot to drive forwards or backwards at variable speed, or turn with any turning radius.

2.3 Sensing model

Consider adding two laser range sensors —Lidar lite v3 (Segway) and part number GYVL53L0X (Paperbot)— and an inertial measurement unit (IMU) —part 3DM-GX5-10, LORD MicroStrain (Segway) or part MPU9250 IMU (Paperbot) onto your robot. The output of these sensors will be a function of the state of the robot within its environment.

The laser range sensors are mounted on the robot such that they measure 1) the distance to a wall in a straight line in front of the robot, and 2) the distance to a wall in a straight line to the right of the robot. The IMU will return 1) a measurement of the in-plane rotational speed from an angular velocity (gyro) sensor, and 2) the components of the measured magnetic field along each of the 2 in-plane coordinate axes, which can be used as a compass for absolute orientation relative to Earth's magnetic field. We will ignore the out-of-plane gyro and magnetometer axes, as well as the accelerometer on the IMU. Thus this robot will produce 5 output values.

3 Comprehensive Simulator Analysis

3.1 Statistical Analysis between Analytical and Physics Simulation

In this section, we aim to understand the capabilities and limitations of each simulator. Using the data obtained from coded and Webots simulations, verify if your analytical simulations match the result of Webots simulations. How similar are the robot trajectories of the two simulations? Do sensor outputs match each other? What is the range of errors? Are there any notable deviations in errors among different trajectories or initial conditions?

Your analysis should be quantitative, not merely qualitative. Be sure to use normalized quantities as opposed to absolute errors, meaning you compare results using unitless metrics or percentages. When comparing the same robot trajectories, outputs or inputs, they need to be in the same plot and they must be plotted in the same formats,

(scale, units, label, plotting software, Matlab/Python Matplotlib). In Webots, you can specify time step size, then both simulators should use the exactly same time steps.

3.2 System Errors and Potential Simulation Scenario

Come up with situations that may cause significant deviations between the results of the analytical and Webots simulations. Can you verify your Joint lab 2 answers to this question? Briefly, explain what scenario you tested, plot your results, then discuss why the results diverge.

3.3 Simulator Analysis

In Joint lab 2 and 3, we test 3 different simulators: analytical, SolidWorks and Webots. Discuss your procedures of how to evaluate/test your robot designs and algorithms based on your analysis done in Joint lab 2 and 3. What is the benefit of analytical/numerical simulators? When do you want to use your analytical/numerical simulators? Are the divergences between each simulator result significant? Are there any scenarios that require use of a particular simulator or where one does not work well? Through your procedures, can you convince evaluators/customers/-bosses/professors that your mechanism moves as designed/does not break and your algorithm works/can complete a task?

3.4 Design Process

By Joint lab 3, we have tested simulation tools you potentially need for your final projects. What could be potential issues/concerns if you're building hardware and run your algorithm on it? List them up in a order of the most concerns to less. Briefly explain your concerns.

4 Software Architecture

4.1 Software Structure

It is essential to spend time on software architecture before starting to work on code. Here, using Webots for all of your simulations may not be appropriate when you are developing your algorithm or debugging it since physics simulations are time consuming. Best practices are to construct your code such that you can easily switch between analytical and numerical simulations (your lab 1 simulator and Webots). Figure 2 shows our preferred software architecture. Here we define 3 layers:

- Simulation Layer
- Interface Layer

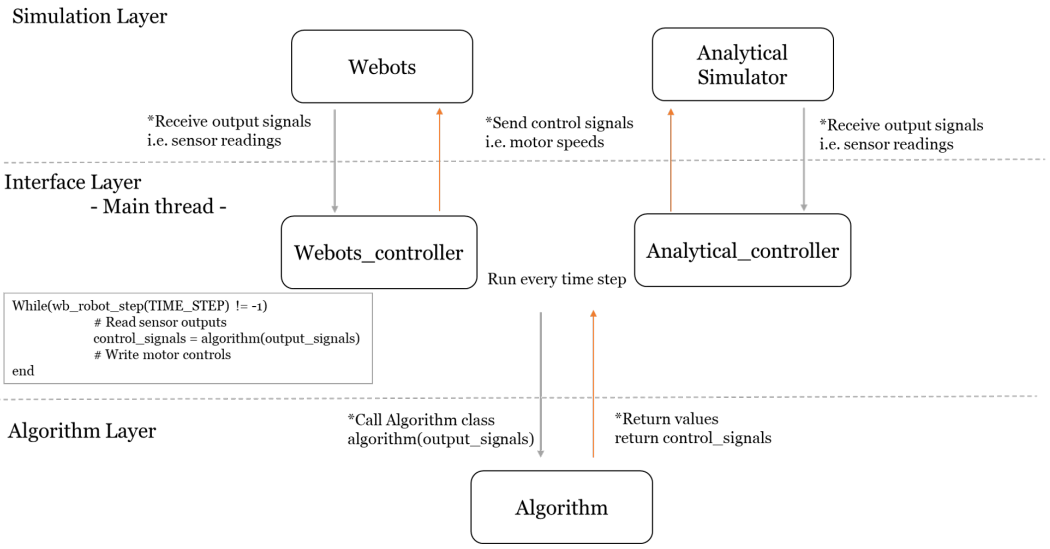

Figure 2: Software architecture chart

- Algorithm Layer

Webots or your analytical coded simulator belongs to the simulation layer. They receive your control signals, simulate and return output signals. The algorithm layer is where you implement your algorithm (MDP solver, LQR path planning, SLAM, etc). The Interface Layer (controller) is where communications between your algorithm and simulator happen and this layer is the main class. When switching simulators, you will simply run a controller for the simulator you want to use, and everything else will be unchanged.

4.1.1 Simulation Layer

You will setup the Webots simulator in this lab and you already coded your analytical simulator in Joint Lab 1 and 2. Thus, all of the tasks for this layer will be complete by the end of this lab.

4.1.2 Interface Layer

The pseudo code for your controllers is in Fig. 2. Your controller must be maintained as simple as possible and you should not have any algorithm parts here. i.e. you may call a MDP solver (algorithm), but you should not call a value_function class/function (computation/process). In the interface layer, at each time step you should:

- Read simulator output signals
- Call your algorithm
- Receive return control signals from algorithm
- Write control signals to a simulator

as well as initializing and closing your simulators if necessary.

Your algorithm should always return the same type and formatted control signals regardless of your controller. The interface layer will handle how to communicate with simulators and the difference between simulators (through wb_motor_set_velocity for Webots). Therefore, controllers should always provide the same type and formatted output signals to your algorithm.

Interface layer will be untouched as long as you don't change what information to read or write, or the way to communicate with your algorithm.

4.1.3 Algorithm Layer

Your algorithm layer is where you implement your algorithm. Here, you **must not** communicate with the simulation layer at all, meaning you will never use functions: wb_..., such as wb_motor_set_velocity and wb_gyro_get_values. Those will create dependencies in your algorithm codes and will make your code not interchangeable.

You may have one "Master" script/function in your algorithm if you have complex or many algorithms, which will handle communication between different algorithms (SLAM to planner, etc.) as shown in Fig. 3. The master will help make your program modular and easy to develop or debug algorithm by algorithm. However, for simple situations, this Master could be redundant, or even make structure more complicated. This is up to you, but remember that

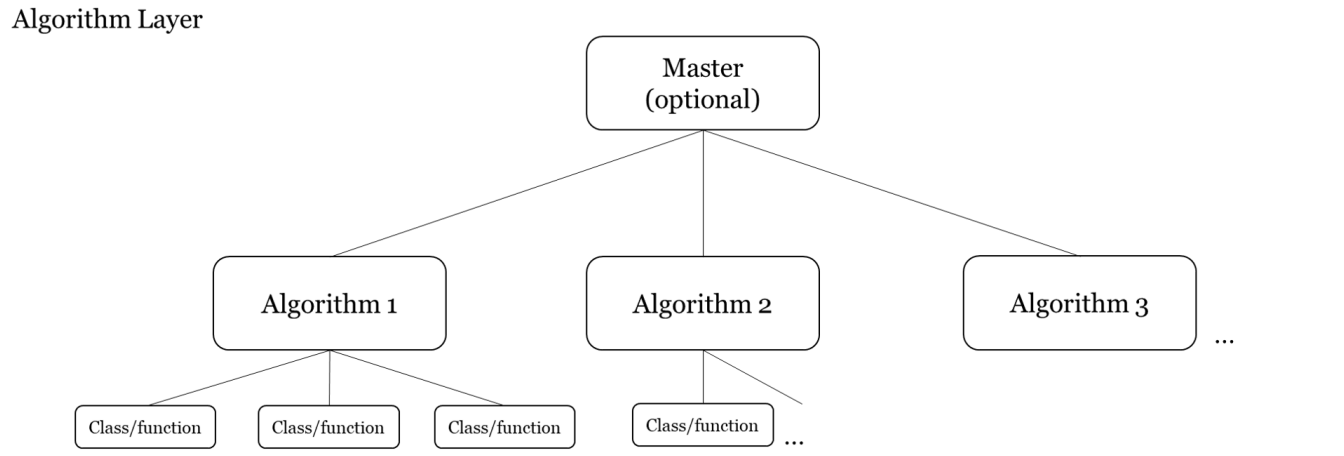

Figure 3: Algorithm layer details

your interface should only read, call, and write. Hence, you should not let your controller do the master role. Keep the interface layer as simple as possible.

Cross communications among classes, functions, or among algorithms are not recommended. They cause cross dependencies in your code, such that if you change one class/function, you will need to modify all of the classes/-functions that it communicates to. If you need cross communications among algorithms, consider making a master. For simple situations, cross communications can be acceptable, but not scalable or modular. You should draw your algorithm architecture before you start cording.

4.2 Implementing Our Software Architecture (Optional)

We will implement software architecture in Fig. 2. Our algorithm will take motor right and left angular velocities from a console, and receive sensor outputs, then print them in the console. You can use "input" function to ask for user inputs.

You write two independent controllers: one for Webots, one for your own simulator. Again, both should call the same function (algorithm) with the same values (sensor readings). Each controller should only do the tasks described in Sec. 4.1.2.

Then, run and test each controller to see if you have implemented your architecture properly. Note that your simulators run time step by time step. Hence, simulation will stop till user inputs.

If successful, you can implement planners in the next lab by replacing our algorithm with a new one.

A Physics Simulator

A.1 Introduction to Webots

In this lab, we use a physics simulator, Webots, to evaluate our robot motions and sensor models. Webots is an open source and free software to simulate dynamics and motions of robots. We will import Segway and Paperbot CAD from Solidworks, but both ECE and MAE students will work on Webots simulations.

A.1.1 Download and Install

You can download and install Webots from here. It is available for Windows 64bit, Mac, and Ubuntu. The latest version is Webots R2021. With the default installation, Webots comes with sample simulations and you can look into them to see what Webots is capable of.

A.1.2 Programming Language Setups

Webots has a built-in basic IDE for many different languages, including Matlab, Python, C++, etc. We recommend script languages. You need to install additional software depending on your choice of the programming language. For more details and procedures, please refer to their manuals.

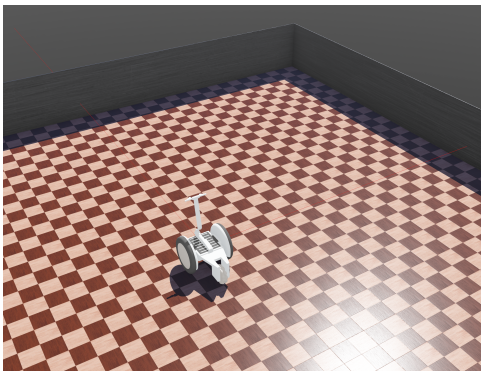

Figure 4: Webots Segway setup

A.2 Segway Webots Setups

Setting up Webots for our Segway robot includes following steps:

- Exporting 3D models from Solidworks and Importing them in Webots
- Configuring your robot boundary conditions and actuation
- Adding and defining sensors
- Setting up a controller to drive our robot and to read sensor outputs

A.2.1 Exporting Segway CAD files

Open the Segway CAD assembly in SolidWorks (from Joint Lab 2, not Joint Lab 1). Go to "File → Save as", and pick file type, VRML (.wrl), then click ***Options...***. Change output as version to "VRML 97" and make sure the unit is in meters. Click okay and save the file. Webots cannot import VRML 1.0 version. We will use Solidworks later. If you are an ECE student and are not familiar with Solidworks, you should get the VRML files from the MAE students on your team. When you are using other CAD software, you may export your design into a .obj file if .wrl is not in options.

A.2.2 Create a new Webots project and environment

Before importing our CAD models, we need to create a new project. Go to "Wizards → New project directory..." then pick a folder you want to place your Webots project in. Set a world name, be sure to enable the "Add a rectangle arena" option, and then finish the wizard. Now we have an environment for our robot. In the Scene tree, you can add or modify your environment, robots, and objects. If your world is missing a rectangle arena, click add a new object (or right click the tree then pick "Add New") and in search bar, type rectangle arena. You will find it under the base node section and click add.

A.2.3 Importing Segway CAD Files

Click "File → Import 3D model" then follow the wizard to locate your file. Click finish with all default settings. You should see several Transform nodes with shapes under their children nodes. We want to put them into one Transform node with all 3D shapes under its children node with appropriate names on them. Change one Transform node name, or DEF, to Segway. Then cut and paste all shapes into Segway Transform node children. When you cut one shape, you can identify which parts they are, and appropriately name them. Delete all empty Transforms.

Now be sure to ***Save your world***. You should save your world as you modify, otherwise all changes will be lost or you may be unable to undo your mistakes. Webots Undo does not keep more than one action, and it is not always available. You can reload your file with a refresh button next to the save button.

Let's set your arena size to 10 by 10 meter to fit our robot. This can be done by modifying the "floorSize" property in the "RectangleArena" drop-down. Additionally, increase the height of the walls to 1 meter by modifying "wallHeight". Your scene tree should appear as in Fig. 5 (your names for your Shapes may vary).

A.2.4 Defining Your Robot

Add a Robot node to your scene tree. Cut and paste your Segway Transform node under the Robot's children node. Now we need to define object boundaries for our robot so that Webots can compute contact with environments or objects. Select translation in the robot node (Not Transform node) and set $y = 0.288m$ to raise our robot above the floor. Our imported CAD models are Shape nodes, which only defines rendering (cosmetic) geometries. Within Segway Transform node's children, add a Solid node which will represents our CAD model shapes as a solid with physics applied. Cut and paste one of your shape into the Solid children node and **the boundingObject node**. The boundingObject nodes define your object boundary conditions and they are used to compute contacts between objects. Double click its physics node and add physics. Now your solid has mass and inertia based on its shape and density. Be sure to rename your solid appropriately and add more solid nodes for each object you want to have contacts or physics, except for the main body/base shape. You should only have one shape in the children node of each solid. Copy your body Shape node to "Robot" boundingObject. Your body shape node should stay under Segway Transform node's children. You should not have solids for your body as its boundary conditions are already defined in your robot node. You should have two cosmetic parts (foot anti-slip rubber), which may not need to be solids (although including their mass will result in a marginally more accurate simulation). Adding more boundingObjects will increase computation time. Under each physics node, set the density to "-1" and the mass (in kg) to reflect the appropriate mass of each part as found in Solidworks.

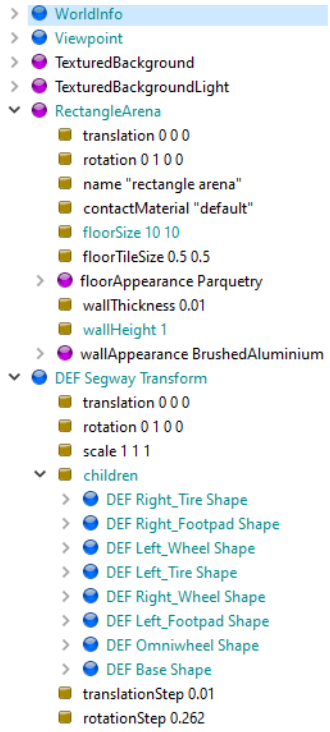

Figure 5: Scene tree after finishing "Importing Segway CAD Files". Your Shape names may vary.

One last step is to move your two wheel solids into your corresponding tire solid nodes' children. This will let us drive both wheel and tire together at the same time.

Save your progress occasionally.

A.2.5 Defining Joints and Actuators

We need to define all joints you want to simulate in your robot. SolidWorks joint definitions are not inherited because your file was exported as meshed objects. We have 3 joints: two actuated tires and an unactuated omni-wheel. Select Segway Transform node's children and add a HingeJoint node and then name it as joint_R. Double click jointParameters and add a jointParameter node. Double click device and add a rotational motor node.

We need to define the joint axis coordinate (the center of the rotation) relative to the robot body coordinate. You can find offsets by using measure tools in Solidworks. Note that the positive y axis is upwards (out of the floor) and the positive x axis points to the right of the robot. Segway wheel axis is at $y = -0.0383m$ and the omni-wheel was at $y = -0.162m$ and $z = 0.425m$. The option to fill for axis position is "jointParameters HingeJointParameters $\rightarrow$ anchor".

Now copy your HingeJoint node and paste at the same level for joint_L. Finally, cut and paste your right and left tires' solids into each endPoint node under respective joint nodes. Expand the rotationalMotor node and change its name to motor_R and motor_L. Be sure to change name, not only DEF. With these names we can control motors from our code.

Also add a HingeJoint for the omni wheel. Afterwards, your scene tree should appear like the one in Fig. 6 (your given names may differ).

Now, ***Save your world*** before you proceed.

A.2.6 Defining Sensors and Their Models

Here, we will add 4 sensors: Gyro, Compass, and two Distance sensors. Add Gyro, Compass and one Distance sensor nodes under the Segway Transform node's children. Change their names to gyro, compass, and lidar_F, but again not only DEF, also name. Expand Lidar_F node and change type to "laser". Open lookup Table node. This will define the relationship between the distance and sensor outputs. X column represents a distance [meter], Y column represents a corresponding sensor output [unitless], and Z column represents error standard deviation [percent]. You can add more control points, but for now let's set the first row to $[0, 0, 0.05]$ and the last row to $[10, 4096, 0.05]$. This

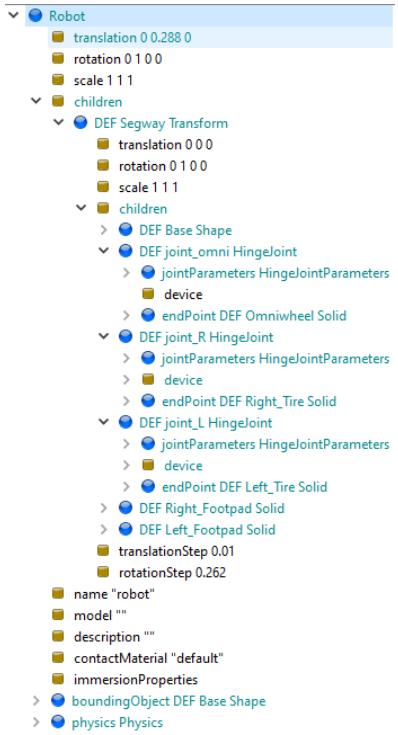

Figure 6: Scene tree after finishing "Defining Joints and Actuators". Your shape names may vary. Note, there are unshown details in the collapsed drop-down menus. Below "physics" is cropped out due to being unchanged.

means that this distance sensor returns 4096 (12 bits) at 10m, 2048 at 5m, and 0 at 0m with Gaussian noise range equal to 5% of the nominal value. Copy your Distance sensor node at the same level and rename it to lidar_R. In the gyro node, right click the lookup table node and select "add new" twice. In the first row, specify $[-100, -100, 0.05]$ and in the second $[100, 100, 0.05]$. This defines the range of your gyro sensor to be -100 to 100 radians per second with Gaussian noise with range of 5% of the nominal response value. For compass, fill out the lookup table according to Table 1.

Table 1: Our compass lookup table

	Actual	Sensor output	STD
1	-1	-1	0.05
2	-0.001	-0.001	0.05
3	0	0	0.05
4	0.001	0.001	0.05
5	1	1	0.05

Now, we need to define those sensor positions. For distance sensors, we need to make sure that the lasers are pointing at walls, not the robot itself. Click "View → Optional Rendering → Show DistanceSensor Rays". This will visualize your sensor ray and its intersection with the walls. Expand lidar_R and change translation to $x = 0.35m$ (moving the sensor to the right of the robot). Expand lidar_F and change translation to $z = -0.25$ (moving the sensors to the front) and the rotation angle to be 1.57rad (90deg). Now lidar_F should point at a wall in front and lidar_R at a wall on the right. IMU and Gyro can be kept at the origin $[0, 0, 0]$.

You can apply a shape or a solid with a bounding object on sensors to visualize or to enable contacts, but for this lab those are unnecessary. ***Save your world*** before you proceed.

A.2.7 Defining a Robot Controller

By now, we have setup robot boundary conditions, actuators, and sensors. The last thing to drive your robot is to setup your robot controller. Click "Wizards → New Robot Controller...". You need to setup your programming languages appropriately as stated in A.1.2. We recommend either Matlab or Python. ECE students may want to

use the same programming language as you used in Joint Lab 1 and 2. MAE students may opt for Matlab as we only try moving our robots and reading sensors. Set your controller name, "lab3_controller", then click "Finish". This should open IDE on your right with sample codes.

Expand your Robot node and click controller. Select your "lab3_controller". ***Save your world*** before you proceed.

Algorithm 1: Running your robot and reading sensors (Matlab)

```
Get simulation time step;
TIME_STEP = wb_robot_get_basic_time_step();
% Create instances of motors sensors, nodes;
motor_R = wb_robot_get_device('motor_R') % Device names have to be char, not string;
compass = wb_robot_get_device('compass');
robot = wb_supervisor_node_get_from_def('robot'); % For ground truth access;
% Enable sensors (instance, sampling period[ms]);
wb_compass_enable(compass, TIME_STEP);
% Make the motors non-position control mode;
wb_motor_set_position(motor_R, inf);
% Need to call wb_robot_step periodically to communicate to the simulator;
while wb_robot_step(TIME_STEP) do
    % Run a motor by velocity rad/sec;
    wb_motor_set_velocity(motor_R,1);
    % Or run a motor by torque N/m;
    wb_motor_set_torque(motor_R,1);
    % Read sensor data;
    compass_data = wb_compass_get_values(compass);
    % Get ground truth data;
    position = wb_supervisor_node_get_position(robot);
    orientation = wb_supervisor_node_get_orientation(robot);
    velocity = wb_supervisor_node_get_velocity(robot);
end
% Close your controller;
wb_robot_cleanup();
```

Before we try running our code, ***you must save both your code and world***. You should always start from simulation time 0 to reflect your code modification. You can click reset simulation. Sometimes, your code changes may not get reflected even if you reset the simulation, and then reload your world. Both reset and reload will delete all unsaved changes in simulations (codes may be safe).

Following the example code in Algorithm 1 (for MatLab) , your robot should run either by controlling angular velocity or torque of the wheels. You can make plots or display the compass reading realtime. Compass returns a vector of the -1 to 1 by default.

The function: wb_robot_step ensures that your while loop runs at every simulation time step and update inputs and outputs from Webots. This returns -1 when your Webots simulation is terminated. You should have only one call of wb_robot_step. For instance, sensor readings will not get updated unless you call wb_robot_step.

Note, the algorithms shown include functions to use a "supervisor". A supervisor allows the use of functions not normally available for nodes or fields. To enable the supervisor and use the commands above, make sure to set DEF for the Robot to be "robot" and set the "supervisor" field under the Robot in the scene tree to "True". It should also be noted that there is currently a bug in the MatLab function "wb_supervisor_node_get_velocity" that causes it to return only linear velocities instead of linear and angular. This bug fix will come out 2/19. You can apply their pre-release Webots (Nightly Build (18-2-2021)) here once available.

This "controller" code in Webots is the interface between the simulator and your algorithm. We discussed software architecture in 4.1.

A.2.8 WorldInfo and Contact definition

Expand WorldInfo tree. Here you can define physics and change simulation settings. Change your simulation simulation time step (basicTimeStep) to 20ms for now, which will slow down your simulation, but improve Segway

simulation accuracy. Later, you can increase time step to larger or reduce if you see some weird behaviors. Double click contactProperties to add a contactProperties node, which defines frictions between objects. Change material 1 to floor and material 2 to tire by typing it out in their properties. Set coulombFriction to 0.7 (rubber). Go to the RectangleArena and two tire solid nodes (recall that they are in your HingeJoints), then change contactMaterial to floor and tire, respectively from default. Copy and paste the contactProperties node at the same level. Change material 2 to omni_wheel and coulombFriction to 0.01. Go to the omni wheel solid node and change the contactMaterial to omni_wheel.

Save your world before you proceed. Now you have done all you need to control your robot. Congrats!

A.3 Paperbot Webots Setups

Procedures for Paperbot setups are the same as Segway, but with a different density and a 1 by 1m rectangle arena. Check to make sure that mass is about 70g in total. Make sure to place your robot on or very close to the floor. An initial drop could make your simulation unstable as discussed in A.5.2.4.

A.4 Running Simulations

Try the following tasks to make sure that you can move your robot and read sensor outputs.

- Make a large turn right (constant motor inputs) and plot the front and right distance sensor history (x=timestep, y=distance[m])
- Spin in place for about 10 seconds then go straight, while plotting gyro z and compass x and y axis outputs

For both, let Webots run for 20s simulation time. If your simulation is slow, you may do shorter duration.

In your write up, be sure to have those plots and screenshots when you stopped your simulation. Reset or reload your world as you change your code.

A.5 Helpful Webots Information

A.5.1 For Final Projects

We recommend to use Webots for your final project simulations, but you are free to use any others. You may not want to always develop or debug with Webots simulations because they are much slower than your analytical simulator. Thus, it is good coding practice to construct your software architecture to be interchangeable for both Webots and coded simulators. More details are in the optional Sec. 4.

A.5.1.1 CAD Modeling

Through this lab, you it is clear that setting up a robot is time consuming even for our simple 2DoF differential drive. If you have more complex kinematics or more DoFs, initial setup or replacing with new designs will take a significant amount of time. You may decide to simplify DoF to simulate in Webots or instead check kinematics in Solidworks. However, once you define your robot in Webots, simulations can be nearly as fast as real time or even faster.

A.5.1.2 Coding

You must present your design and algorithm for your capstone projects using simulations this year. You can use any physics simulator you like, such as CoppeliaSim/V-rep and Unity3D. However, we recommend using Webots as it is simpler, your entire team now has an introduction to it, and you don't need to use ROS or API. Communication between your code and a simulator can be annoying, but with Webots all you have to do is define your controller file. You can have other functions or code in the same directly, and you should still have scope to them from your controller. As a best practice, you should make your codes interchangeable between our coded simulator from Joint Lab 3 and the Webots controller. In this way, you can develop your code in your favorite IDE (Matlab or Pycharm) and test with your analytical simulation which is much faster than numerical simulations as we experienced in Joint Lab 2. Then, you can run Webots to test your robot and algorithm in more realistic setups.

A.5.2 Performance, Rendering, Multi-threading

Webots is many times faster than the Solidworks simulations in Lab 2, as it can run at realtime or faster. However, several factors can affect Webots performance. You may improve the speed by tweaking settings. Mostly, the speed of your simulation is determined by your single thread CPU and GPU performance.

A.5.2.1 Slow Down Due to Rendering

Webots renders its simulation results by default to visualize your simulation at runtime. However, rendering is a GPU intensive task and this could significantly slow down your simulation depending on your computer GPU. If this is your case here is the list of workaround you can do:

- Reduce FPS in Webots WorldInfo settings
- Reduce your screen resolutions
- Opt for Wireframe rendering in Webots
- Turn off rendering

A.5.2.2 Slow Down Due to Complex Geometry Definitions

If your one thread of CPU is running at 100% and it is slowing down your simulations, you may simplify your robots or objects in Webots. Webots uses meshed 3D models. Typically, CAD software exports fine meshed files even for simple geometries, i.e. cube, cylinder, sphere. For your final project, you may simplify your CAD design before exporting for simulation purposes (removing cosmetic designs or replacing some geometries with something simpler). Alternatively, you can define 3D shapes inside of Webots, which is computationally more efficient and optimized. You may consider to define simpler bounding objects using a Webots Cylinder node for Omni-wheels, for instance.

Another way to reduce CPU burden is increasing simulation time step. Increasing each timestep size (in milliseconds) will reduce the number of physics computations per second in the simulator. However, too large time step may generate inaccurate results and make physics unstable. More details are in Sec. A.5.2.4.

A.5.2.3 Slow Down Due to Complex Environment or Multi-agent Simulation

If you have multiple robots or objects in your environment, they will slow down your simulation because now Webots need to compute physics for all objects. Then, you can enable multi-threading, which will let Webots use other available CPU cores or threads. By default, the physics calculation is done by one thread. However, speed improvement due to multi-threading is dependent on your simulations, and there will be no benefit if we you have one object, such as in our Lab 3 simulations.

A.5.2.4 Instability Due to Complex Physics

Your simulation may behave weirdly or unrealistically (penetrating the floor, for instance). This could be because your simulated physics are too complicated, which also slows down your simulations. One common situation that causes instability is having an impact on your robot, i.e. dropping your robot from high or running full speed into a wall. If you need to simulate such conditions, then you can reduce timestep to improve or correct the results at the cost of simulation speed.