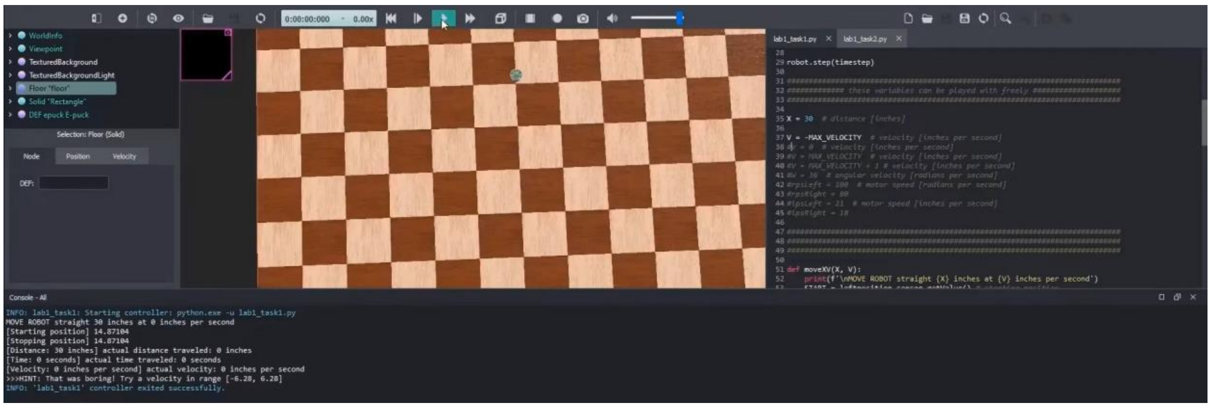
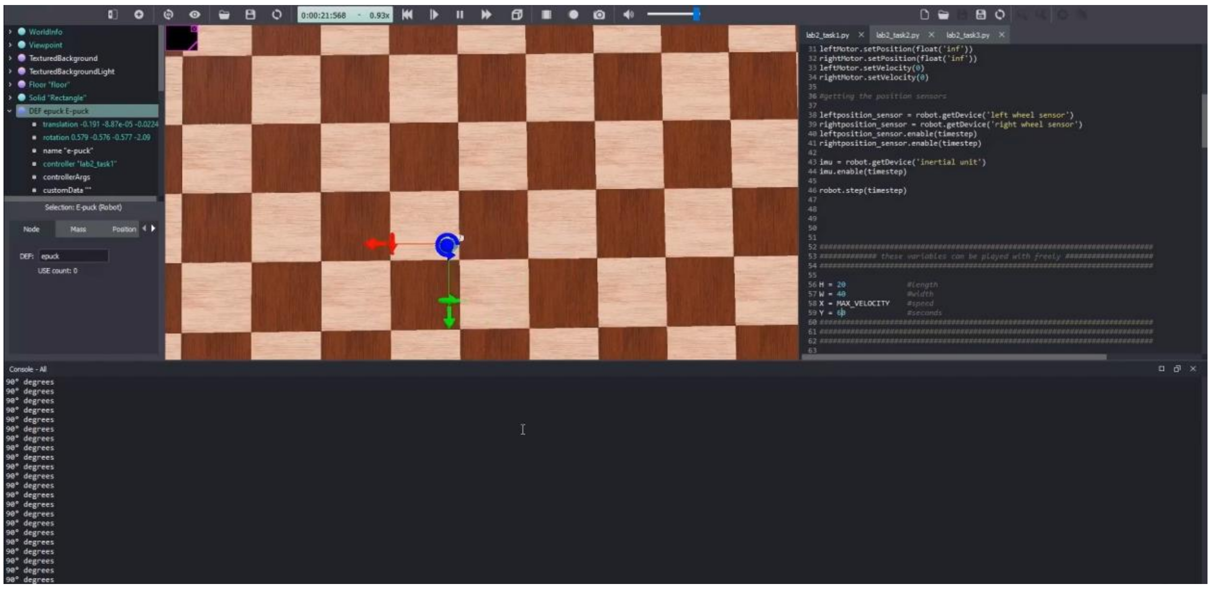


```
robot.step(timestep)

#############################################################################
############# these variables can be played with freely #####################
#############################################################################

X = 30  # distance [inches]

V = -MAX_VELOCITY  # velocity [inches per second]
#V = 0  # velocity [inches per second]
#V = MAX_VELOCITY  # velocity [inches per second]
#V = MAX_VELOCITY + 1 # velocity [inches per second]
#W = 36  # angular velocity [radians per second]
#rpsLeft = 100  # motor speed [radians per second]
#rpsRight = 80
#ipsLeft = 21  # motor speed [inches per second]
#ipsRight = 18

#############################################################################
#############################################################################
#############################################################################

def moveXV(X, V):
    print(f'\nMOVE ROBOT straight {X} inches at {V} inches per second')
```

```
INFO: lab1_task1: Starting controller: python.exe -u lab1_task1.py
MOVE ROBOT straight 30 inches at 0 inches per second
[Starting position] 14.87104
[Stopping position] 14.87104
[Distance: 30 inches] actual distance traveled: 0 inches
[Time: 0 seconds] actual time traveled: 0 seconds
[Velocity: 0 inches per second] actual velocity: 0 inches per second
>>>HINT: That was boring! Try a velocity in range [-6.28, 6.28]
INFO: 'lab1_task1' controller exited successfully.
```

- Lab 1 focuses on basic robot programming in Webots.
    1. Task 1 involves driving a robot in a straight line for a specified distance at a constant velocity.
       Four test cases are performed, including exceeding the motor speed, running at max velocity, zero velocity (with a print statement indicating no movement), and negative velocity for backward movement.
    2. Task 2 involves learning to read the encoder on the simulator, with the robot rotating in a circle of a specified radius at a given velocity.
       Five test cases are performed, covering scenarios such as exceeding max velocity, zero velocity (resulting in an error message about linear velocity), negative velocity (causing counterclockwise movement), positive velocity (causing clockwise movement), and a zero radius circle (resulting in the robot spinning in place).



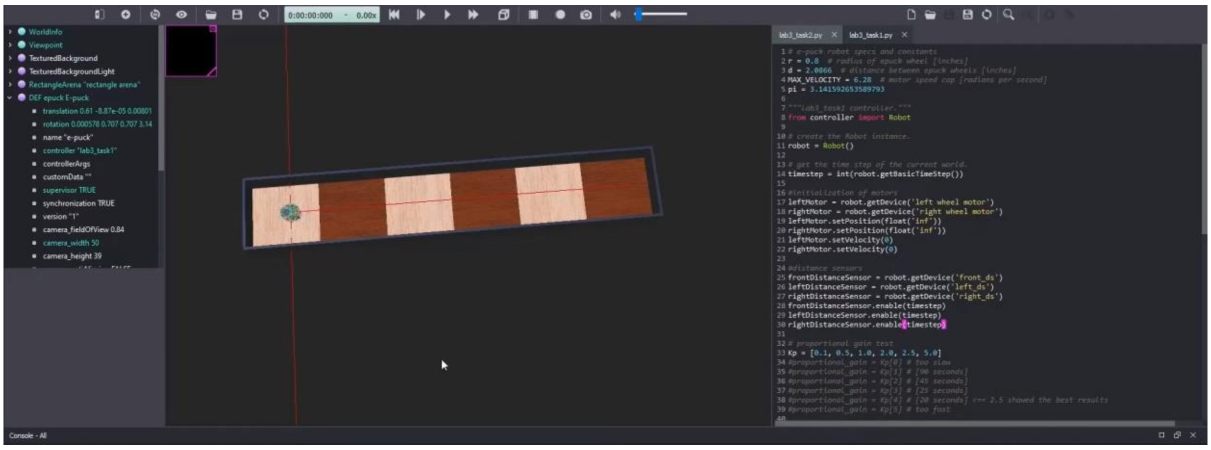
```
leftMotor.setPosition(float('inf'))
rightMotor.setPosition(float('inf'))
leftMotor.setVelocity(0)
rightMotor.setVelocity(0)

#getting the position sensors

leftposition_sensor = robot.getDevice('left wheel sensor')
rightposition_sensor = robot.getDevice('right wheel sensor')
leftposition_sensor.enable(timestep)
rightposition_sensor.enable(timestep)

imu = robot.getDevice('inertial unit')
imu.enable(timestep)

robot.step(timestep)





#############################################################################
############# these variables can be played with freely #####################
#############################################################################

H = 20              #length
W = 40              #width
X = MAX_VELOCITY    #speed
Y = 60              #seconds
#############################################################################
#############################################################################
#############################################################################
```

```
90° degrees
90° degrees
90° degrees
90° degrees
90° degrees
90° degrees
90° degrees
90° degrees
90° degrees
90° degrees
90° degrees
90° degrees
90° degrees
90° degrees
90° degrees
90° degrees
90° degrees
90° degrees
90° degrees
90° degrees
90° degrees
90° degrees
```

- Lab 2 involves three tasks focused on implementing controllers for a differential drive robot using Python.
  Controllers are designed to move the robot in various shapes, dictated by given parameters such as distances, widths, heights, speeds, or radii.
  Each controller throws error warnings in two scenarios: when the robot can't complete the path within the specified time and when the velocity exceeds the maximum capacity.

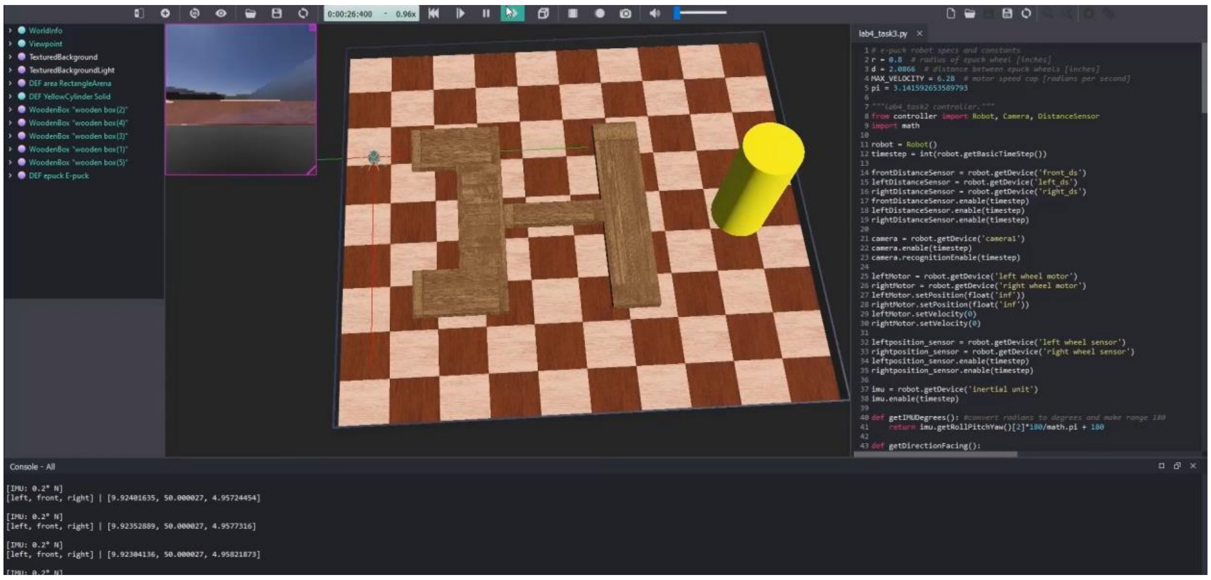
During execution, each controller prints the direction the robot is moving in degrees rather than radians.

- Task 1 involves moving the robot in a clockwise rectangular path, Task 2 requires driving in circles with specified radii, and Task 3 involves navigating waypoints to create an oval shape, with error handling for exceeding maximum velocity.



```python
# e-puck robot specs and constants
r = 0.8  # radius of epuck wheel [inches]
d = 2.0866  # distance between epuck wheels [inches]
MAX_VELOCITY = 6.28  # motor speed cap [radians per second]
pi = 3.141592653589793

"""lab3_task1 controller."""
from controller import Robot

# create the Robot instance.
robot = Robot()

# get the time step of the current world.
timestep = int(robot.getBasicTimeStep())

#initialization of motors
leftMotor = robot.getDevice('left wheel motor')
rightMotor = robot.getDevice('right wheel motor')
leftMotor.setPosition(float('inf'))
rightMotor.setPosition(float('inf'))
leftMotor.setVelocity(0)
rightMotor.setVelocity(0)

#distance sensors
frontDistanceSensor = robot.getDevice('front_ds')
leftDistanceSensor = robot.getDevice('left_ds')
rightDistanceSensor = robot.getDevice('right_ds')
frontDistanceSensor.enable(timestep)
leftDistanceSensor.enable(timestep)
rightDistanceSensor.enable(timestep)

# proportional gain test
Kp = [0.1, 0.5, 1.0, 2.0, 2.5, 5.0]
#proportional_gain = Kp[0] # too slow
#proportional_gain = Kp[1] # [90 seconds]
#proportional_gain = Kp[2] # [45 seconds]
#proportional_gain = Kp[3] # [25 seconds]
#proportional_gain = Kp[4] # [20 seconds] <== 2.5 showed the best results
#proportional_gain = Kp[5] # too fast
```

- Lab 3 focuses on navigation using distance sensors on a robot.
  Tasks involve continuous monitoring of left, right, and front sensors, with motor velocity controlled by a proportional control function. Proportional control is explained using a PID controller with only the proportional component considered for this lab.
- Task 1 involves a straight-line drive using PID to slow down as it approaches 10 inches from a wall, showcasing the saturation function to ensure the robot can handle the velocity.
- Tasks 2, 3, and 4 progressively introduce challenges such as maintaining distance from side walls, navigating a corridor with turns, and solving a maze using the same proportional control functions.

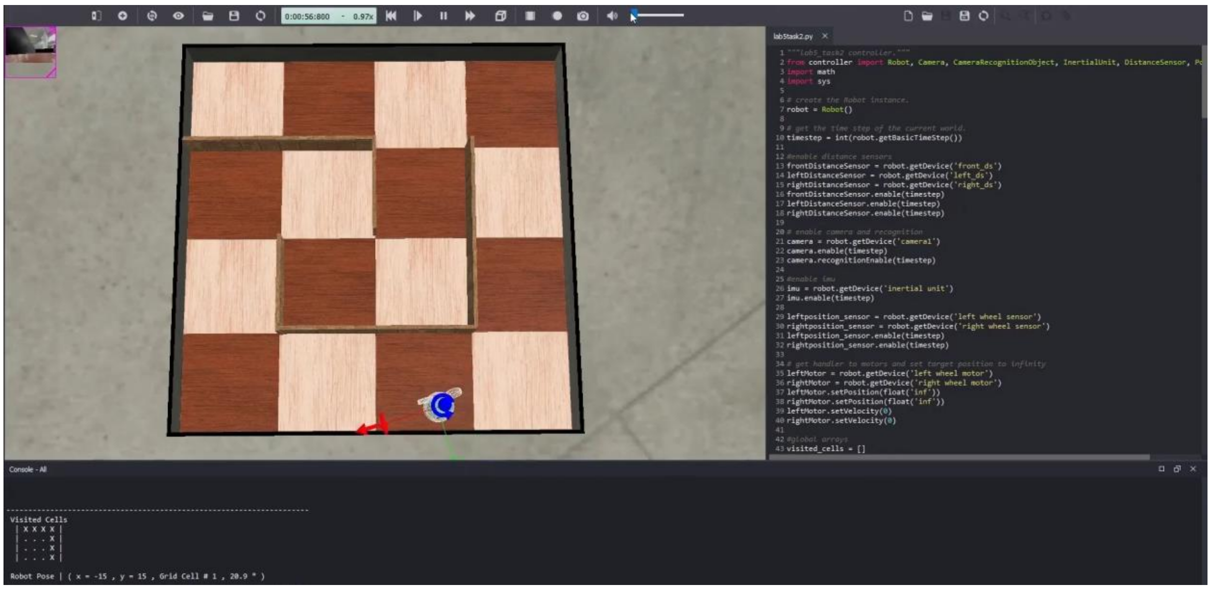


```python
# e-puck robot specs and constants
r = 0.8  # radius of epuck wheel [inches]
d = 2.0866  # distance between epuck wheels [inches]
MAX_VELOCITY = 6.28  # motor speed cap [radians per second]
pi = 3.141592653589793

"""lab4_task2 controller."""
from controller import Robot, Camera, DistanceSensor
import math

robot = Robot()
timestep = int(robot.getBasicTimeStep())

frontDistanceSensor = robot.getDevice('front_ds')
leftDistanceSensor = robot.getDevice('left_ds')
rightDistanceSensor = robot.getDevice('right_ds')
frontDistanceSensor.enable(timestep)
leftDistanceSensor.enable(timestep)
rightDistanceSensor.enable(timestep)

camera = robot.getDevice('camera1')
camera.enable(timestep)
camera.recognitionEnable(timestep)

leftMotor = robot.getDevice('left wheel motor')
rightMotor = robot.getDevice('right wheel motor')
leftMotor.setPosition(float('inf'))
rightMotor.setPosition(float('inf'))
leftMotor.setVelocity(0)
rightMotor.setVelocity(0)

leftposition_sensor = robot.getDevice('left wheel sensor')
rightposition_sensor = robot.getDevice('right wheel sensor')
leftposition_sensor.enable(timestep)
rightposition_sensor.enable(timestep)

imu = robot.getDevice('inertial unit')
imu.enable(timestep)

def getIMUDegrees(): #convert radians to degrees and make range 180
    return imu.getRollPitchYaw()[2]*180/math.pi + 180

def getDirectionFacing():
```

```
[IMU: 0.2° N]
[left, front, right] | [9.92401635, 50.000027, 4.95724454]

[IMU: 0.2° N]
[left, front, right] | [9.92352889, 50.000027, 4.9577316]

[IMU: 0.2° N]
[left, front, right] | [9.92304136, 50.000027, 4.95821873]

[IMU: 0.2° N]
```

- Lab focus: Webots Lab 4 - Navigation with Camera.
  Introduction of a camera tool in the lab, providing a first-person perspective for the robot.
  Tasks involve using PID control logic with the camera to maintain desired distance, orientation, and speed relative to a yellow cylinder, the goal object.

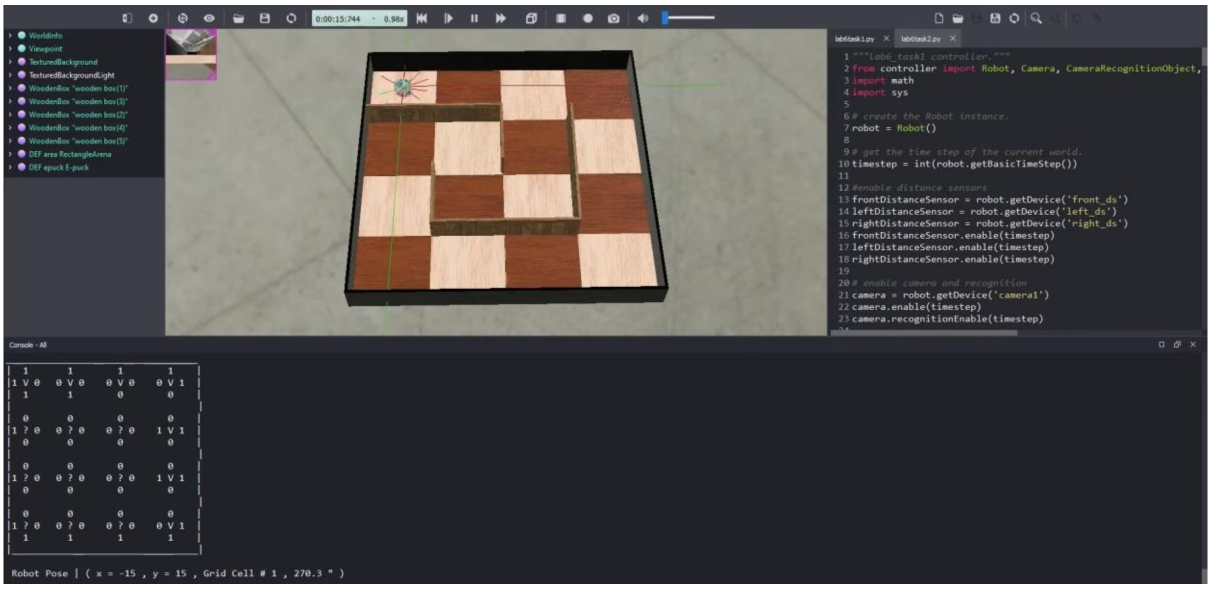
1. Task 1: Face the goal object at all times using the camera's ability to detect the goal's position on the x-axis.
2. Task 2: Drive towards the goal, stopping approximately five inches away, employing PID control for precise movement.
3. Task 3 (partially completed): Implementation of the Bug Zero algorithm using the camera, where the robot beelines toward the goal or wall follows around obstacles, with a demonstration of success and failure scenarios.



```
"""lab5_task2 controller."""
from controller import Robot, Camera, CameraRecognitionObject, InertialUnit, DistanceSensor, Po
import math
import sys

# create the Robot instance.
robot = Robot()

# get the time step of the current world.
timestep = int(robot.getBasicTimeStep())

#enable distance sensors
frontDistanceSensor = robot.getDevice('front_ds')
leftDistanceSensor = robot.getDevice('left_ds')
rightDistanceSensor = robot.getDevice('right_ds')
frontDistanceSensor.enable(timestep)
leftDistanceSensor.enable(timestep)
rightDistanceSensor.enable(timestep)

# enable camera and recognition
camera = robot.getDevice('camera1')
camera.enable(timestep)
camera.recognitionEnable(timestep)

#enable imu
imu = robot.getDevice('inertial unit')
imu.enable(timestep)

leftposition_sensor = robot.getDevice('left wheel sensor')
rightposition_sensor = robot.getDevice('right wheel sensor')
leftposition_sensor.enable(timestep)
rightposition_sensor.enable(timestep)

# get handler to motors and set target position to infinity
leftMotor = robot.getDevice('left wheel motor')
rightMotor = robot.getDevice('right wheel motor')
leftMotor.setPosition(float('inf'))
rightMotor.setPosition(float('inf'))
leftMotor.setVelocity(0)
rightMotor.setVelocity(0)

#global arrays
visited_cells = []
```

```
-------------------------------------------------------------------
Visited Cells
 | X X X X |
 | . . . X |
 | . . . X |
 | . . . X |

Robot Pose | ( x = -15 , y = 15 , Grid Cell # 1 , 20.9 ° )
```

- Lab Objective - Localization:
  Lab 5 focuses on introducing the concept of localization in the Webots environment. The goal is to determine the robot's position on a grid map at any given time.
- Trilateration Algorithm:
  Task 1 involves using the trilateration algorithm for localization. Trilateration is explained as a method similar to GPS systems, using distances from known points to pinpoint the robot's location.
- Algorithm Implementation:
  The lab uses four pillars in the corners of the grid, each with known coordinates, to perform trilateration. The distance formulas are employed to calculate distances from the robot to the pillars, and a system of equations is solved to find the robot's exact x-y coordinates.
- Grid Exploration and Troubleshooting:
  A beta version of the code is demonstrated, where the robot scans the environment, gathers pillar distances, performs trilateration, and prints the grid cell number. Challenges are mentioned, particularly in scanning cells two and three, which the presenter aims to address before the final build.
- Task 2 - Wall-Based Localization:
  Task 2 involves determining the robot's location based on distance sensor readings for walls. The robot explores the grid, avoiding walls, and uses sensor data to eliminate possible cell locations, achieving localization without trilateration.



```python
"""lab6_task1 controller."""
from controller import Robot, Camera, CameraRecognitionObject,
import math
import sys

# create the Robot instance.
robot = Robot()

# get the time step of the current world.
timestep = int(robot.getBasicTimeStep())

#enable distance sensors
frontDistanceSensor = robot.getDevice('front_ds')
leftDistanceSensor = robot.getDevice('left_ds')
rightDistanceSensor = robot.getDevice('right_ds')
frontDistanceSensor.enable(timestep)
leftDistanceSensor.enable(timestep)
rightDistanceSensor.enable(timestep)

# enable camera and recognition
camera = robot.getDevice('camera1')
camera.enable(timestep)
camera.recognitionEnable(timestep)
```

```
 ________________________________
|  1     1      1      1     |
|1 V 0  0 V 0  0 V 0  0 V 1  |
|  1     1      0      0     |
|                            |
|  0     0      0      0     |
|1 ? 0  0 ? 0  0 ? 0  1 V 1  |
|  0     0      0      0     |
|                            |
|  0     0      0      0     |
|1 ? 0  0 ? 0  0 ? 0  1 V 1  |
|  0     0      0      0     |
|                            |
|  0     0      0      0     |
|1 ? 0  0 ? 0  0 ? 0  0 V 1  |
|  1     1      1      1     |
|____________________________|

Robot Pose | ( x = -15 , y = 15 , Grid Cell # 1 , 270.3 ° )
```

- Lab 6 focuses on mapping and path planning in Webots.
  The mapping section uses sensor readings and Monte Carlo particle filter localization to detect nearby walls, map out the walls in the maze, and store the internal wall configuration for printing. In the path planning section, a wavefront planner algorithm is used to find the shortest path from a start cell to a goal cell. The known grid layout is used for path planning.
- Two types of wavefront algorithms, eight-point connectivity and four-point connectivity, are explained. The latter is used in the algorithm, considering only adjacent neighbor cells for movement.
- The path planning algorithm is demonstrated with known and random start/goal cells, showing how the robot follows the generated path using the wavefront planner.