Hyun Kim
Daniel Palma
Akshay Paluri

Team 4

Robot design

The design of the robot was focused around making the robot as small and compact as possible. We created a base for the two motors to attach to and then used the metal ball attachment so that the robot was able to move forward, turn in place, and turn in arcs. This essentially created a robot that would act like a two wheeled unicycle.

The gyro sensor was also attached to the robot in a spot directly between the two motors that also allowed it to be level to the ground. The decision to use the gyro sensor was made because the motors of the robot were inconsistent in the power delivered moving forward and backward at the same time for turns. This difference in the motor's operation meant that turns would either be overshot or short of their desired 90° rotation in either direction.

The wheel choice for the robot was not one by design but by necessity. The chassis of the robot was too narrow to allow for wheels that were when the robot was attached to the top of it. The wheels, however, are secured to the robot in such a way that the wheels do not sag or droop in an angle when in use. This helps in aiding the robot move in a way that is predictable and does not shake or add movements that are hard to account for through observation alone.

Other design decisions made were to add pegs on to the robot to route cables so that they would stay closer to the body of the robot and pegs on the wheels so that testing could be conducted on the rotation of the wheels. We also ensured that the body of the robot was made as sturdy as possible by connecting as many parts of the robot to each other as we could.

The robot keeps track of its position within the course using multiple flags that represent the direction the robot is facing. The robot then performs the necessary movements to move to the desired cell as determined by the path finding algorithm.

Movement functions

- Constants:
  1. TURN_SPEED – Set the robot's turning speed.
  2. MOVE_SPEED - Set the robot's forward or backward movement speed.
  3. ANGLE - Define the degree of rotation for the left and right wheels when turning (used for left and right turns).
  4. TILE_DEGREE – Specify the number of degrees the robot will move in the x or y direction. This is used for forward and backward movements and depends on the length of one side of a tile.
  5. right_f, up_f, down_f, left_f – Set flags based on the robot's orientation.

- External inputs:
  1. ev3 – Our interface for passing input and receiving output. We programmed how much the moters will rotate and read output from gyroscope sensor.
  2. left_wheel – A variable used to control the left motor's movement. It uses Port.C
  3. right_wheel - A variable used to control the right motor's movement. It uses Port.B
  4. gyro – A sensor used to measure real-life degrees of rotation. It uses Port.S4

- Functions:
  1. BotForward(left_wheel, right_wheel):
     - This function enables the robot to move forward by spinning both wheels for an amount equal to TILE_DEGREE.
  2. BotBackward(left_wheel, right_wheel):
     - This function allows the robot to move backward by spinning both wheels for an amount equal to TILE_DEGREE.
  3. BotLeft(left_wheel, right_wheel):
     - This function enables the robot to turn left by specifying the degree of rotation for the wheels (right wheel: -ANGLE, left wheel: ANGLE).
  4. BotRight(left_wheel, right_wheel):
     - This function enables the robot to turn right by specifying the degree of rotation for the wheels (right wheel: ANGLE, left wheel: -ANGLE).
  5. BotCenter(left_wheel, right_wheel):
     - This function instructs the robot to move toward the center of a tile. It involves a right turn, moving forward, and a left turn and moving forward.
  6. BotLeft45(left_wheel, right_wheel):
     - This function instructs the robot to turn left by 45 degrees.
     - It is used when the robot goes to the intersection from the center of a tile.
  7. BotRight45(left_wheel, right_wheel):
     - This function instructs the robot to turn right by 45 degrees.
     - It is used when the robot goes to the intersection from the center of a tile.

Path finding

In order for the robot to find a valid path, we first translated the obstacle coordinates into a 2d array, which simulates a room that the robot has to follow. Since it was given that the obstacles are located on intersection of square tiles, and every tile and obstacle are squares of .305m, we filled in the room array, array we used to represent the area of the room. Every cell the obstacle covered was denoted with 1 and every free space cell was denoted with 0. Since the start and goal coordinates were also given, we denoted those points with 3 and 6 respectively.

Now that we had a room where every tile was labeled, we needed to implement a strategy for the starting point to be able to find a path to the goal. This meant that starting point, denoted by 3, needs to end up at the located initially labelled by 6. In order to implement this strategy, we used A star search, which would help us find the fewest number of cells the robot has to travel to get to the path. To implement this, we needed to come up with a node, which would be store an instance of the path in a queue, and a heuristic, which would find the best node to travel to. In order to implement the node for the queue, we included the room array, which would show the particular instance of the vehicle, and would contain the instance variable of cost, which tells the cost it took for the robot to travel to the current cell. We also included instance variable of parent, which would keep track of its parent node and the move it made.

Now that we had the sufficient information in the node, we now used the idea of breadth first search with it being sorted by cost it took travel to the current cell plus the heuristic, which would be the Manhattan distance of the current cell to the goal cell. The fringe would be sorted using this number and the loop would continue generating new path until the best path is found and meets the goal state criteria. Then the goal node is taken, and since we have the information of its parent node, we would store the direction of the current node and its parents in array. This array would contain the path required to start from the starting state to the goal state.

main.py

```python

from pybricks.hubs import EV3Brick
from pybricks.ev3devices import (Motor, TouchSensor, ColorSensor,
                                 InfraredSensor, UltrasonicSensor, GyroSensor)
from pybricks.parameters import Port, Stop, Direction, Button, Color
from pybricks.tools import wait, StopWatch, DataLog
from pybricks.media.ev3dev import SoundFile, ImageFile
import math
import path_functions as pf
import path_node as pn
import path as p
import time

TURN_SPEED = 40
MOVE_SPEED = 180
ANGLE = 120
TILE_DEGREE = 630

def BotForward(left_wheel, right_wheel, ev3):
    left_wheel.run_target(MOVE_SPEED, TILE_DEGREE, wait=False)
    right_wheel.run_target(MOVE_SPEED, TILE_DEGREE)
    left_wheel.reset_angle(0)
    right_wheel.reset_angle(0)

def BotBackward(left_wheel, right_wheel, ev3):
    left_wheel.run_target(MOVE_SPEED, -1*TILE_DEGREE, wait=False)
    right_wheel.run_target(MOVE_SPEED, -1*TILE_DEGREE)
    left_wheel.reset_angle(0)
    right_wheel.reset_angle(0)

def BotLeft(left_wheel, right_wheel, ev3):
    gyro.reset_angle(0)
    angle2 = ANGLE
    wait_f = False
    while(1):
        left_wheel.run_target(TURN_SPEED, -1*angle2, wait=False)
        right_wheel.run_target(TURN_SPEED, angle2)
        ang = gyro.angle()
        left_wheel.reset_angle(0)
        right_wheel.reset_angle(0)
        if wait_f:
            time.sleep(0.25)
        if(ang <= -50):
            angle2 = 1
        if(ang <= -92):
            ev3.screen.print(ang)
            left_wheel.run_target(TURN_SPEED, 5)
```

```
            break

def BotRight(left_wheel, right_wheel, ev3):
    gyro.reset_angle(0)
    angle2 = ANGLE
    wait_f = False
    while(1):
        left_wheel.run_target(TURN_SPEED, angle2, wait=False)
        right_wheel.run_target(TURN_SPEED, -1*angle2)
        ang = gyro.angle()
        left_wheel.reset_angle(0)
        right_wheel.reset_angle(0)
        if wait_f:
            time.sleep(0.25)
        if(ang >= 50):
            angle2 = 1
        if(ang >= 85):
            ev3.screen.print(ang)
            if ang > 90:
                left_wheel.run_target(TURN_SPEED, -1*angle2, wait=False)
                right_wheel.run_target(TURN_SPEED, angle2)

            right_wheel.run_target(TURN_SPEED, 5)
            break
def BotRight45(left_wheel, right_wheel, ev3):
    gyro.reset_angle(0)
    angle2 = ANGLE/2
    wait_f = False
    while(1):
        left_wheel.run_target(TURN_SPEED, angle2, wait=False)
        right_wheel.run_target(TURN_SPEED, -1*angle2)
        ang = gyro.angle()
        left_wheel.reset_angle(0)
        right_wheel.reset_angle(0)
        ev3.screen.print(ang)
        if wait_f:
            time.sleep(0.25)
        if(ang >= 30):
            angle2 = 1
            wait_f = True
        if(ang >= 44):
            ev3.screen.print(ang)
            break

def BotLeft45(left_wheel, right_wheel, ev3):
    angle2 = ANGLE/2
    gyro.reset_angle(0)
    wait_f = False
    while(1):
        left_wheel.run_target(TURN_SPEED, -angle2, wait=False)
```

```python
        right_wheel.run_target(TURN_SPEED, 1*angle2)
        ang = gyro.angle()
        left_wheel.reset_angle(0)
        right_wheel.reset_angle(0)
        if wait_f:
            time.sleep(0.25)
        if(ang <= -33):
            angle2 = 1
            wait_f = True
        if(ang <= -45):
            ev3.screen.print(ang)
            break
def BotDiagonal(left_wheel, right_wheel):
    left_wheel.run_target(MOVE_SPEED, TILE_DEGREE/2, wait=False)
    right_wheel.run_target(MOVE_SPEED, TILE_DEGREE/2)
    left_wheel.reset_angle(0)
    right_wheel.reset_angle(0)

def BotCenter(left_wheel, right_wheel, ev3):
    BotRight(left_wheel, right_wheel, ev3)
    BotDiagonal(left_wheel, right_wheel)
    BotLeft(left_wheel, right_wheel, ev3)
    BotDiagonal(left_wheel, right_wheel)

ev3 = EV3Brick()
left_wheel = Motor(Port.C)
right_wheel = Motor(Port.B)
gyro = GyroSensor(Port.S4)

path = p.findpath()
print(path)
BotCenter(left_wheel, right_wheel, ev3)
gyro.reset_angle(0)

right_f, up_f, down_f, left_f = True, False, False, False
for i in path:
    ev3.screen.print(i)
    if i == "right":
        if right_f == True:
            BotForward(left_wheel, right_wheel, ev3)
            right_f, up_f, down_f, left_f = True, False, False, False
        elif up_f == True:
            BotRight(left_wheel, right_wheel, ev3)
            BotForward(left_wheel, right_wheel,ev3)
            right_f, up_f, down_f, left_f = True, False, False, False
        elif down_f == True:
            BotLeft(left_wheel, right_wheel, ev3)
            BotForward(left_wheel, right_wheel, ev3)
            right_f, up_f, down_f, left_f = True, False, False, False
        elif left_f == True:
```

```
            BotRight(left_wheel, right_wheel, ev3)
            BotRight(left_wheel, right_wheel, ev3)
            BotForward(left_wheel, right_wheel, ev3)
            right_f, up_f, down_f, left_f = True, False, False, False
    elif i == "up":
        if right_f == True:
            BotLeft(left_wheel, right_wheel, ev3)
            BotForward(left_wheel, right_wheel, ev3)
            right_f, up_f, down_f, left_f = False, True, False, False
        elif up_f == True:
            BotForward(left_wheel, right_wheel, ev3)
            right_f, up_f, down_f, left_f = False, True, False, False
        elif down_f == True:
            BotLeft(left_wheel, right_wheel, ev3)
            BotLeft(left_wheel, right_wheel, ev3)
            BotForward(left_wheel, right_wheel)
            right_f, up_f, down_f, left_f = False, True, False, False
        elif left_f == True:
            BotRight(left_wheel, right_wheel, ev3)
            BotForward(left_wheel, right_wheel, ev3)
            right_f, up_f, down_f, left_f = False, True, False, False
    elif i == "down":
        if right_f == True:
            BotRight(left_wheel, right_wheel, ev3)
            BotForward(left_wheel, right_wheel, ev3)
            right_f, up_f, down_f, left_f = False, False, True, False
        elif up_f == True:
            BotRight(left_wheel, right_wheel, ev3)
            BotRight(left_wheel, right_wheel, ev3)
            BotForward(left_wheel, right_wheel, ev3)
            right_f, up_f, down_f, left_f = False, False, True, False
        elif down_f == True:
            BotForward(left_wheel, right_wheel, ev3)
            right_f, up_f, down_f, left_f = False, False, True, False
        elif left_f == True:
            BotLeft(left_wheel, right_wheel,ev3)
            right_f, up_f, down_f, left_f = False, False, True, False
    elif i == "left":
        if right_f == True:
            BotBackward(left_wheel, right_wheel, ev3)
            right_f, up_f, down_f, left_f = False, False, False, True
        elif up_f == True:
            BotLeft(left_wheel, right_wheel,ev3)
            BotForward(left_wheel, right_wheel)
            right_f, up_f, down_f, left_f = False, False, False, True
        elif down_f == True:
            BotRight(left_wheel, right_wheel, ev3)
            BotForward(left_wheel, right_wheel, ev3)
            right_f, up_f, down_f, left_f = False, False, False, True
        elif left_f == True:
```

```
            BotForward(left_wheel, right_wheel, ev3)
            right_f, up_f, down_f, left_f = False, False, False, True

if right_f == True:
    BotLeft(left_wheel, right_wheel, ev3)
    BotLeft45(left_wheel, right_wheel, ev3)
    BotDiagonal(left_wheel, right_wheel)
elif up_f == True:
    BotLeft45(left_wheel, right_wheel, ev3)
    BotDiagonal(left_wheel, right_wheel)
elif down_f == True:
    BotRight(left_wheel, right_wheel, ev3)
    BotRight45(left_wheel, right_wheel, ev3)
    BotDiagonal(left_wheel, right_wheel)
elif left_f == True:
    BotRight45(left_wheel, right_wheel, ev3)
    BotDiagonal(left_wheel, right_wheel)
```

path.py

```
import math
import path_functions as pf
import path_node as pn
def findpath():
    room = [[0 for i in range(16)] for j in range(10)]

    obstacle_coord = [[4, 1], [4, 2],[4, 3],[4,4],
                    [4, 5],[7, 4], [7, 5],[7, 6],
                    [7, 7],[7, 8],[7, 9],[7, 10],[10, 3], [10, 4], [10, 5], [10, 6], [10, 7],
                    [11, 3], [12, 3], [12, 4], [13, 3], [13, 4]]

    tile_obstacle = []
    for i in obstacle_coord:
         tile_obstacle.append([i[0],i[1]])

    for i in tile_obstacle:
        room[10 - i[1]][i[0]] = 1
        room[10 - i[1] - 1][i[0]] = 1
        room[10 - i[1]][i[0] - 1] = 1
        room[10 - i[1] - 1][i[0] - 1] = 1

    start = [.305,1.219]
    goal = [3.658,1.829]

    start_cellx = 2
    start_celly = 10 - 2

    goal_cellx = 13
    goal_celly = 10 - 7

    room[goal_celly][goal_cellx] = 6
    print(start_cellx)
    print(start_celly)
    room[start_celly][start_cellx] = 3

    goal = pf.gen_coord(room, 6)
    start = pf.gen_coord(room, 3)

    goal_room = pf.gen_goal_room(room, start, goal)

    start_node = pn.Node(list(room), 0, "Start", 0, 0,None)
    start_node.print_node()
    def manhattan(start,goal):
        return abs(start[0] - goal[0]) + abs(start[1] - goal[1])

    fringe = []
```

```python
    closed = []

    fringe.append([start_node,manhattan(pf.gen_coord(room,3),goal)])
    while fringe:
        node_puzzle = fringe.pop(0)
        node = node_puzzle[0]

        if node.grid[goal[0]][goal[1]] == 3:
            break

        if node.grid not in closed:
            closed.append(node.grid)
            if node.check("right"):
                right = node.gen_state("right",node,node.cost+1)
                fringe.append([right, right.cost + manhattan(pf.gen_coord(right.grid,3),goal)])
            if node.check("down"):
                down = node.gen_state("down",node,node.cost+1)
                fringe.append([down, down.cost + manhattan(pf.gen_coord(down.grid,3),goal)])
            if node.check("left"):
                left  = node.gen_state("left",node,node.cost+1)
                fringe.append([left, left.cost + manhattan(pf.gen_coord(left.grid,3),goal)])
            if node.check("up"):
                up = node.gen_state("up",node,node.cost+1)
                fringe.append([up, up.cost + manhattan(pf.gen_coord(up.grid,3),goal)])

        fringe.sort(key=lambda x:x[1])

    node.print_node()
    moveList = []
    moveArr = []
    while node is not None:
        moveList.append(node.move)
        moveArr.append(node.grid)
        node = node.parent

    moveArr.reverse()
    moveList.reverse()
    return moveList[1:]
```

path_node.py

```python
import path_functions as pf

class Node:
    def __init__(self, grid, cost, move, tileMoved, hval,parent):
        self.grid = grid
        self.cost = cost
        self.move = move
        self.tileMoved = tileMoved
        self.hval = hval
        self.parent = parent

    def gen_state(self, direction,parent,cost):
        deepGrid = [row[:] for row in self.grid]
        row = pf.locate_row(deepGrid, 3)
        col = pf.locate_col(deepGrid, 3)
        blank = 0
        tile = 0
        if direction == 'up':
            blank = deepGrid[row][col]
            tile = deepGrid[row-1][col]
            deepGrid[row][col] = tile
            deepGrid[row-1][col] = blank
        if direction == 'down':
            blank = deepGrid[row][col]
            tile = deepGrid[row+1][col]
            deepGrid[row][col] = tile
            deepGrid[row+1][col] = blank
        if direction == 'left':
            blank = deepGrid[row][col]
            tile = deepGrid[row][col-1]
            deepGrid[row][col] = tile
            deepGrid[row][col-1] = blank
        if direction == 'right':
            blank = deepGrid[row][col]
            tile = deepGrid[row][col+1]
            deepGrid[row][col] = tile
            deepGrid[row][col+1] = blank
        #newCost = self.cost+tile
        generatedState = Node(deepGrid, cost, direction, tile, self.hval,parent)
        return generatedState

    def check(self,direction):
        row = pf.locate_row(self.grid, 3)
        col = pf.locate_col(self.grid, 3)
        if direction == "up":
            if row > 0 and (self.grid[row-1][col] == 0 or self.grid[row-1][col] == 6)
:
```

```python
                return True
            else:
                return False
        if direction == "down":
            if row < 9 and (self.grid[row+1][col] == 0 or self.grid[row+1][col] == 6):
                return True
            else:
                return False
        if direction == "left":
            if col >= 1 and (self.grid[row][col-1] == 0 or self.grid[row][col-1] == 6):
                return True
            else:
                return False
        if direction == "right":
            if col < 15 and (self.grid[row][col+1] == 0 or self.grid[row][col+1] == 6):
                return True
            else:
                return False


    def print_node(self):
        for i in range(10):
            print(self.grid[i])
        print('\n')

```

path_functions.py

```python
def locate_row(grid, char):
    row = 0
    for i in grid:
        row+=1
        col = 0
        for k in i:
            col+=1
            if k == char:
                return row-1

def locate_col(grid, char):
    row = 0
    for i in grid:
        row+=1
        col = 0
        for k in i:
            col+=1
            if k == char:
                return col-1

def gen_coord(grid, char):
    coord = []
    coord.append(locate_row(grid, char))
    coord.append(locate_col(grid, char))
    return coord

def gen_goal_room(room, start, goal):
    goal_room = [row[:] for row in room]
    goal_room[start[0]][start[1]] = 0
    goal_room[goal[0]][goal[1]] = 3
    return goal_room
```