# Lab 4

In this lab the goal was: implement goal to goal behavior in the e-puck using localization and specific positions by their coordinates. I did this using a P controller for the distance and manually setting the rotation duration per corner as the simulation steps made it hard to rotate accurately and using the phi variable was unreliable as there was no triangle, only a straight line, but any deviation from said line would make the program overcorrect. Because of the limitations I think I barely passed the bar and would give myself a 6. The go to goal behaviour is present, so is the P controller, but there is no line following behaviour, no PID and no saturation on the actuators.

## Code explanation

The robot has 6 goals. Top left, bottom left, bottom right, top right, top middle and middle middle. Each goal has an x and y coordinate, but as it is always a straight line, the distance to the goal is the goal – current position on said axis. So, for example: distance needed to travel to the goal along the x-axis is 1.6m but the distance on the y-axis = 0. The program takes the biggest value (1.6), sees the axis connected to said value and updates the error value using the robots value on said axis. It then drives the distance to the goal divided by the distance covered per simulation step plus the amount of taken simulation steps. Example: goal = 1.6m and the program has run for 100 steps, then the calculation would be: step distance = (1.6/0.004) + 100 = 500 steps. The program will now drive the step distance – the number of steps, this results in 400 steps taken. Every step is 0.004 meters so in total 400 \* 0.004 = 1.6m covered.

After driving to the first point using this algorithm it will turn 90 degrees and add 1 to the goal counter. It now targets the second goal, upon completion turns 90 degrees and adds 1 to the counter, and it does this until the last goal where it stops in the middle of the map.

## Code:

This is the code. This is not a screenshot; this is copy pasted from Visual Studio Code. The .py file is also available on blackboard. Set word view to web layout for the clearest picture of the code.

"""Lab4 controller."""

from controller import Robot, DistanceSensor, Motor

import numpy as np

#-------------------------------------------------------

# Initialize variables

MAX\_SPEED = 6.28

# create the Robot instance.

robot = Robot()

# get the time step of the current world.

timestep = int(robot.getBasicTimeStep())   # [ms]

delta\_t = robot.getBasicTimeStep()/1000.0    # [s]

# states

states = ['forward', 'turn\_right', 'turn\_left']

current\_state = states[0]

# counter: used to maintain an active state for a number of cycles

counter = 0

COUNTER\_MAX = 3

goalc = 0 #goal counter

xcom = 0 #xgoal completed

setgoal = 0

steps = 0

distancesteps = 0

rotatesteps = 0

srotate = 0

# Robot pose

# Adjust the initial values to match the initial robot pose in your simulation

x = 0    # position in x [m]

y = 0    # position in y [m]

phi = 0  # orientation [rad]

# Robot wheel speeds

wl = 0.0    # angular speed of the left wheel [rad/s]

wr = 0.0    # angular speed of the right wheel [rad/s]

# Robot linear and angular speeds

u = 0.0    # linear speed [m/s]

w = 0.0    # angular speed [rad/s]

# e-puck Physical parameters for the kinematics model (constants)

R = 0.0205    # radius of the wheels: 20.5mm [m]

D = 0.052    # distance between the wheels: 52mm [m]

A = 0.05    # distance from the center of the wheels to the point of interest [m]

#-------------------------------------------------------

# Initialize devices

# distance sensors

ps = []

psNames = ['ps0', 'ps1', 'ps2', 'ps3', 'ps4', 'ps5', 'ps6', 'ps7']

for i in range(8):

    ps.append(robot.getDevice(psNames[i]))

    ps[i].enable(timestep)

# ground sensors

gs = []

gsNames = ['gs0', 'gs1', 'gs2']

for i in range(3):

    gs.append(robot.getDevice(gsNames[i]))

    gs[i].enable(timestep)

# encoders

encoder = []

encoderNames = ['left wheel sensor', 'right wheel sensor']

for i in range(2):

    encoder.append(robot.getDevice(encoderNames[i]))

    encoder[i].enable(timestep)

oldEncoderValues = []

# motors

leftMotor = robot.getDevice('left wheel motor')

rightMotor = robot.getDevice('right wheel motor')

leftMotor.setPosition(float('inf'))

rightMotor.setPosition(float('inf'))

leftMotor.setVelocity(0.0)

rightMotor.setVelocity(0.0)

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

#----- functions ----------------------------------------------------------------

def PositionError(xgoal, ygoal, fx, fy, fphi):

    # Position error:

    x\_err = xgoal - fx

    y\_err = ygoal - fy

    dist\_err = np.sqrt(x\_err\*\*2 + y\_err\*\*2)

    # Orientation error

    phi\_d = np.arctan2(y\_err,x\_err)

    phi\_err = phi\_d - fphi

    # Limit the error to (-pi, pi):

    phi\_err\_correct = np.arctan2(np.sin(phi\_err),np.cos(phi\_err))

    return x\_err, y\_err, phi\_err\_correct, dist\_err

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

# Robot Localization functions

def get\_wheels\_speed(encoderValues, oldEncoderValues, delta\_t):

    """Computes speed of the wheels based on encoder readings"""

    #Encoder values indicate the angular position of the wheel in radians

    wl = (encoderValues[0] - oldEncoderValues[0])/delta\_t

    wr = (encoderValues[1] - oldEncoderValues[1])/delta\_t

    return wl, wr

def get\_robot\_speeds(wl, wr, r, d):

    """Computes robot linear and angular speeds"""

    u = r/2.0 \* (wr + wl)

    w = r/d \* (wr - wl)

    return u, w

def get\_robot\_pose(u, w, x\_old, y\_old, phi\_old, delta\_t):

    """Updates robot pose based on heading and linear and angular speeds"""

    delta\_phi = w \* delta\_t

    phi = phi\_old + delta\_phi

    phi\_avg = (phi\_old + phi)/2

    if phi >= np.pi:

        phi = phi - 2\*np.pi

    elif phi < -np.pi:

        phi = phi + 2\*np.pi

    delta\_x = u \* np.cos(phi\_avg) \* delta\_t

    delta\_y = u \* np.sin(phi\_avg) \* delta\_t

    x = x\_old + delta\_x

    y = y\_old + delta\_y

    return x, y, phi

#-------------------------------------------------------

# Main loop:

# - perform simulation steps until Webots is stopping the controller

while robot.step(timestep) != -1:

    # Update sensor readings

    steps += 1

    encoderValues = []

    for i in range(2):

        encoderValues.append(encoder[i].getValue())    # [rad]

    # Update old encoder values if not done before

    if len(oldEncoderValues) < 2:

        for i in range(2):

            oldEncoderValues.append(encoder[i].getValue())

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

    # Using the equations for the robot kinematics based on speed

    # Compute speed of the wheels

    [wl, wr] = get\_wheels\_speed(encoderValues, oldEncoderValues, delta\_t)

    # Compute robot linear and angular speeds

    [u, w] = get\_robot\_speeds(wl, wr, R, D)

    # Compute new robot pose

    [x, y, phi] = get\_robot\_pose(u, w, x, y, phi, delta\_t)

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

    # update old encoder values for the next cycle

    oldEncoderValues = encoderValues

    #set goals

    if goalc == 0:

        xg = 0.8

        yg = 0

    elif goalc == 1:

        xg = 0

        yg = 0.9

    elif goalc == 2:

        xg = 1.6

        yg = 0

    elif goalc == 3:

        xg = 0

        yg = 1

    elif goalc == 4:

        xg = 0.8

        yg = 0

    elif goalc == 5:

        xg = 0

        yg = 0.45

    xerr, yerr, phierr, disterr = PositionError(xg, yg, x, y, phi)

    if xg > yg:

        error = xerr

        current = x

        goal = xg

    else:

        error = yerr

        current = y

        goal = yg

    if goalc > 5:

        leftMotor.setVelocity(0)

        rightMotor.setVelocity(0)

        print('good job!')

        setgoal = 1

        srotate = 1

        break

    if setgoal == 0:

        distancesteps = (goal / 0.004) + steps

        setgoal = 1

    if distancesteps >= steps:

        leftSpeed = MAX\_SPEED

        rightSpeed = MAX\_SPEED

    elif steps > distancesteps:

        if srotate == 0:

            if goalc == 0:

                rotatesteps = steps + 44

            elif goalc == 1:

                rotatesteps = steps + 42

            elif goalc == 2:

                rotatesteps = steps + 44

            elif goalc == 3:

                rotatesteps = steps + 44

            elif goalc == 4:

                rotatesteps = steps + 43

            srotate = 1

        if rotatesteps > steps:

            leftSpeed = MAX\_SPEED/4

            rightSpeed = -MAX\_SPEED/4

        else:

            goalc += 1

            srotate = 0

            setgoal = 0

    # Set motor speeds with the values defined by the state-machine

    leftMotor.setVelocity(leftSpeed)

    rightMotor.setVelocity(rightSpeed)

    print(f'X ={x:.2f}m, Y ={y:.2f}m, phi ={phi:.2f}m, ')

    # Repeat all steps while the simulation is running.