# Lab 5

The goal for this lab was:

Create functions to generate a trajectory of your choice inside the field. The functions must be properly defined to generate desired positions (xd, yd) and speeds with their corresponding time derivates (dxd, dyd).

You are free to choose the generated trajectory, as long as:

* It moves the robot along both x and y axes.
* It is possible for the robot to follow it (positions and speed).

Adjust the controller gains to make your robot follow the trajectory with as little error as possible.

In your demonstration, you must show:

* The value of distance error (Euclidean distance between the actual robot position and desired position) at every cycle.
* That the robot can follow the trajectory using the trajectory-tracking controller.
* That your robot can follow different trajectories (by changing the parameters of your equations (xd, yd, dxd, dyd)) without modifying its controller.

I think I deserve a 6 for this lab is didn’t code a whole lot myself and the error is usually bigger than 0.05m. It can however follow 2 trajectories and settle on an error beneath 0.1m and the same goes for a trajectory where the initial value is bigger than 1m. The only problem is that it would need a lot more tweaking of the gain values and very specific functions, and this would cost a lot of time. Seeing as I do not have a lot of time, I decided to opt for the realistic option.

## Code explanation

As I didn’t have a lot of time, I took the solution posted in the lab and adjusted the controller gains.

I managed to get the overall Euclidean error to stay under 0.1m. I also made 3 test functions which I used to get these gains. I found that the bigger the values in your functions the bigger the gain needed to keep the error small, but the bigger the gain the more inaccurate it gets with small values. So, I tried to keep the values within the functions in an equal range.

## 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.

"""Lab 5 controller."""

from controller import Robot, DistanceSensor, Motor

import numpy as np

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

# Initialize variables

TIME\_STEP = 64

MAX\_SPEED = 6.28

counter = 0

# create the Robot instance.

robot = Robot()

# get the time step of the current world.

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

delta\_t = timestep/1000.0    # [s]

# Robot pose

# \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* ADJUST VALUES OF ROBOT POSE \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

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

x = -0.06    # position in x [m]

y = 0.436    # position in y [m]

phi = 0.0531  # orientation [rad]

# \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# Robot initial velocity and acceleration in (x,z) coordinates

dx = 0.0   # speed in x [m/s]

dy = 0.0   # speed in y [m/s]

ddx = 0.0  # acceleration in x [m/s^2]

ddy = 0.0  # acceleration in y [m/s^2]

# 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]

is\_saturated = False    # Indicates when any of the wheels is saturated (max speed)

# Robot linear and angular speeds

u = 0.0    # linear speed [m/s]

w = 0.0    # angular speed [rad/s]

# Physical parameters of the robot for the kinematics model

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

D = 0.0565    # distance between the wheels: 52mm [m]

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

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

# Initialize devices

# 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 dist\_err

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\_cartesian\_speeds(u, w, phi, a):

    """Computes cartesian speeds of the robot"""

    dx = u \* np.cos(phi) + a \* w \* np.sin(phi)

    dy = u \* np.sin(phi) - a \* w \* np.cos(phi)

    dphi = w

    return dx, dy, dphi

def get\_robot\_pose(x\_old, y\_old, phi\_old, dx, dy, dphi, delta\_t):

    """Updates robot pose"""

    phi = phi\_old + dphi \* delta\_t

    if phi >= np.pi:

        phi = phi - 2\*np.pi

    elif phi < -np.pi:

        phi = phi + 2\*np.pi

    x = x\_old + dx \* delta\_t

    y = y\_old + dy \* delta\_t

    return x, y, phi

def traj\_tracking\_controller(dxd, dyd, xd, yd, x, y, phi, a):

    """Updates references speeds for the robot to follow a trajectory"""

    # Controller gains:

    KX = 4

    KY = 2

    # Position error:

    x\_err = xd - x

    y\_err = yd - y

    # If error is smaller than some value, make it null:

    if (abs(x\_err) < 0.001) and (abs(y\_err) < 0.001):

        x\_err = 0

        y\_err = 0

    # Controller equation - matrix format:

    #C = np.matrix([[np.cos(phi), np.sin(phi)],

    #               [-1/a\*np.sin(phi), 1/a\*np.cos(phi)]])

    #[u\_ref, w\_ref] = C \* np.matrix([[dxd + kx\*x\_err],[dyd + ky\*y\_err]])

    # Controller equations - non-matrix format:

    u\_ref = np.cos(phi)\*(dxd + KX\*x\_err) + np.sin(phi)\*(dyd + KY\*y\_err)

    w\_ref = -(1/a)\*np.sin(phi)\*(dxd + KX\*x\_err) + (1/a)\*np.cos(phi)\*(dyd + KY\*y\_err)

    return u\_ref, w\_ref

def wheel\_speed\_commands(u\_ref, w\_ref, d, r):

    """Converts reference speeds to wheel speed commands"""

    global is\_saturated

    leftSpeed = float((2 \* u\_ref - d \* w\_ref) / (2 \* r))

    rightSpeed = float((2 \* u\_ref + d \* w\_ref) / (2 \* r))

    # Limits the maximum speed of one wheel to MAX\_SPEED, if necessary.

    # Keeps the proportion between left and right wheel speeds

    if np.abs(leftSpeed) > MAX\_SPEED or np.abs(rightSpeed) > MAX\_SPEED:

        speed\_ratio = np.abs(rightSpeed)/np.abs(leftSpeed)

        is\_saturated = True

        if speed\_ratio > 1:

            rightSpeed = np.sign(rightSpeed)\*MAX\_SPEED

            leftSpeed = np.sign(leftSpeed)/speed\_ratio

        else:

            leftSpeed = np.sign(leftSpeed)\*MAX\_SPEED

            rightSpeed = np.sign(rightSpeed)\*speed\_ratio

    else:

        is\_saturated = False

    #leftSpeed = np.sign(leftSpeed) \* min(np.abs(leftSpeed), MAX\_SPEED)

    #rightSpeed = np.sign(rightSpeed) \* min(np.abs(rightSpeed), MAX\_SPEED)

    return leftSpeed, rightSpeed

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

# Main loop:

# - perform simulation steps until Webots is stopping the controller

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

    # Update sensor readings

    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())

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

    # Robot Localization

    x\_old = x

    y\_old = y

    phi\_old = phi

    # 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 cartesian speeds of the robot

    [dx, dy, dphi] = get\_cartesian\_speeds(u, w, phi, A)

    # Compute new robot pose

    [x, y, phi] = get\_robot\_pose(x\_old, y\_old, phi\_old, dx, dy, dphi, delta\_t)

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

    # Robot Controller

    # Desired trajectory (you can use equations to define the trajectory):

    xd = 0.0 + 0.3\*np.sin(0.005\*counter)

    yd = 0.436

    dxd = 0.3\*0.005\*np.cos(0.005\*counter) # This is the time derivative of yd

    dyd = 0.0

    # Desired trajectory 2

    #xd = 0.0 + 0.03\*np.sin(0.009\*counter)

    #yd = 0.436

    #dxd = 0.03\*0.009\*np.cos(0.009\*counter) # This is the time derivative of yd

    #dyd = 0

    # Desired trajectory 3

    #xd = 0.0 + 0.007\*np.sin(0.05\*counter)

    #yd = 0.436

    #dxd = 0.007\*0.05\*np.cos(0.05\*counter) # This is the time derivative of yd

    #dyd = 0

    Diste = PositionError(xd, yd, x, y, phi)

    # Trajectory tracking controller

    [u\_ref, w\_ref] = traj\_tracking\_controller(dxd, dyd, xd, yd, x, y, phi, A)

    # Convert reference speeds to wheel speed commands

    [leftSpeed, rightSpeed] = wheel\_speed\_commands(u\_ref, w\_ref, D, R)

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

    # update old encoder values and counter for the next cycle

    oldEncoderValues = encoderValues

    counter += 1

    # To help on debugging:

    print(f'Sim time: {robot.getTime():.3f}  Pose: x={x:.2f} m, y={y:.2f} m, phi={phi:.4f} Distance error={Diste:.3f}m.')# , w\_ref={w\_ref:.3f} rad/s, Saturation = {is\_saturated}.')

    # Update reference velocities for the motors

    leftMotor.setVelocity(leftSpeed)

    rightMotor.setVelocity(rightSpeed)