

Assignment 1: Basic Controllers

You will implement a series of simple, basic controllers using an e-puck robot. The main goals of this assignment are:

- To get comfortable with connecting to and programming software to remotely control an e-puck robot.
- Understand what an open-loop controller is and what the key limitations are.
- To understand the forms and limitations of simple kinematics for differential drive robots.

You will develop a very high level of comfort with programming differential drive robots to perform a range of desired high level commands. You will create toolkits that enable you to think on the level of what you want the robot to do, not how to manage actuators directly.

There is a lot of overhead on this assignment before you can really get anything working. You will need to take the time to get a connection to the robot, a development environment working, and live code running and communicating with the robot. Check out the online **e-puck robot primer** document and be sure you can successfully run the included sample program before you start. I recommend connecting to the robot via Bluetooth for this assignment as it is simpler and more robust than the WiFi connection.

epuck library

I have (written and...) provided a simplistic library for connecting to and communicating with the epuck robot. See the sample code in the primer package and the epucksample.py program. This library has minimal error handling and has had minimal debugging so you may have to delve into the robot and communications protocol if something breaks.

However, it should provide sufficient functions to connect to the robot, send commands, and read its sensors. Take time to read it over and be familiar with the functions and data provided.

Hint: helper functions

I provide line counts throughout the assignment. This is not a target, but a hint: if your code is significantly longer, you may be doing something wrong.

Put each task into its own file. I use AS1.2.py, AS1.3.py, etc.

I found it useful to start by implementing and testing the following helper functions. You will need to take measurements of your robot to do these. There is sufficient variation that I would not rely on spec sheets but rather use high quality calipers to measure if you can. I put these into a file called As1lib so I could use them across my tasks, which I put into separate files.

- `steps_delta(last, current): int`, calculates the difference in robot steps from the last position to the current, accounting for counter wraparound, and returns it as a signed integer. This one is tricky. Mine is 5 lines including def and return.
- `steps_to_rad(steps): float`, converts signed motor steps to signed radians, and returns that value, using your knowledge of the motor construction. 2 lines
- `rad_to_steps(rad): float`, converts signed radians to signed motor steps, and returns that value, using your knowledge of the motor construction. 2 lines
- `rad_to_mm(rad): float`, converts the given signed radians of wheel rotation into expected signed ground distance, and returns that value. 2 lines

3000
4000
5000
6000
7000
8000
9000
10000
11000
12000
13000
14000
15000
16000
17000
18000
19000
20000
21000
22000
23000
24000
25000
26000
27000
28000
29000
30000
31000
32000
33000
34000
35000
36000
37000
38000
39000
40000
41000
42000
43000
44000
45000
46000
47000
48000
49000
50000
51000
52000
53000
54000
55000
56000
57000
58000
59000
60000
61000
62000
63000
64000
65000
66000
67000
68000
69000
70000
71000
72000
73000
74000
75000
76000
77000
78000
79000
80000
81000
82000
83000
84000
85000
86000
87000
88000
89000
90000
91000
92000
93000
94000
95000
96000
97000
98000
99000
100000

- `mm_to_rad(mm): float`, converts the given mm distance into expected radians of wheel rotation, and returns that value. 2 lines
- `steps_to_mm(steps): float`, converts motor steps into expected ground distance, and returns that value. 2 lines
- `mm_to_steps(mm): float`, converts expected ground distance into motor steps, and returns that value. 2 lines
- `print_pose ((x_mm, y_mm, theta_rad))`, prints x,y,theta while converting theta to degrees. 3 lines
-

Task 1: Simple open-loop¹ controller (20%)

You will first develop a simple control loop that uses the built-in robot odometry to hit a target movement distance. Start by implementing the following function:

```
move_steps(epuckcomm, l_speed_steps_s, r_speed_steps_s, l_target_steps,
r_target_steps, Hz=10): (left_steps_moved, right_steps_moved)
```

This function sets the robot's left and right wheel speed as given, and then starts a control loop that monitors the robot odometry readings to see how far (in motor steps) the robot has gone. The loop should stop after *both* the left and right targets were met (note that one may overshoot). Use `time.sleep()` to control the loop speed. After the targets are hit, make the robot stop moving and the function return. Return a tuple representing the actual left and right steps moved. Mine is 31 lines.

- You can test this with a few simple cases.
 - o `move_steps(epuckcomm, R_MAX_SPEED, R_MAX_SPEED, 1000, 1000)` should move the robot forward about 13cm
 - o `move_steps(epuckcomm, -R_MAX_SPEED, -R_MAX_SPEED, 1000, 1000)` should move the robot backward about 13cm
 - o `move_steps(epuckcomm, R_MAX_SPEED, -R_MAX_SPEED, 1290, 1290)` should move the robot on the spot, clockwise, about one rotation
 - o `move_steps(epuckcomm, R_MAX_SPEED, 0, 2580, 0)` should make the robot turn right, pivoting on the right wheel, about one rotation

Next, we will make a wrapper that lets you work in mm instead of steps.

```
move_straight (epuckcomm, distance_mm, Hz=10)
```

which commands the robot to move a given distance then stop. A negative distance indicates backward. The function returns when the motion is complete, returning the actual robot distance moved based on odometry readings. This should call `move_steps` to get the work done.

The following two lines should move the robot forward by 1m, then move back roughly to the same spot. Tune your math and measurements until you are reasonably accurate.

```
move_straight (epuckcomm, 1000)
move_straight (epuckcomm, -1000)
```

¹ Note that, in using the wheel's built in step counter, it appears as if you are in fact using a closed loop controller on the wheel motor. However, since the motor doesn't provide any actual feedback this is a guess. Further, it is open loop with respect to the robot's location within the environment as we are not having any feedback on our location, we can only guess based on what directions we gave.


Deliverables

- **Demonstration /5**

Task 2: Open-loop forward kinematics (15%)

In this task you will implement forward kinematics for the robot to convert a given robot movement into an expected real-world outcome.

Write the following function which takes in the current robot pose as a tuple (x, y, θ) and number of steps that each of the wheels have turned, and returns a tuple representing the robot's configuration after the given movement. This function does not move the robot at all but only does calculations.

 `diff_drive_forward_kin((r_x, r_y, r_theta), left_steps, right_steps): (newx, newy, newtheta)`, Takes current robot pose and returns the new pose as a tuple after the left and right wheel movement in steps

Note that θ of zero means the robot is looking straight down the x axis. Tips: Using numpy with vectors and matrices makes this much easier. Mine is 31 lines.

You can test this with a few simple cases. Note that due to floating point and math errors, and your particular robot measurements, you may not have exact answers to those shown.

```
o print_pose(diff_drive_forward_kin( (0, 0, 0), 0, 0)) # should give: (0, 0, 0)
o print_pose(diff_drive_forward_kin( (10, 20, 0), 1290, 1290)) # should give:
  (178, 20, 0)
o print_pose(diff_drive_forward_kin( (10, 20, np.pi/2), 1290, 1290)) # should
  give: (10, 188, 90)
o print_pose(diff_drive_forward_kin( (0, 0, 0), -1290, 1290)) #should give (0, 0,
  0)
o print_pose(diff_drive_forward_kin( (0, 0, np.pi/2), 1290, -1290)) #should give
  (0, 0, 90)
o print_pose(diff_drive_forward_kin( (0, 0, 0), 2580, 0)) #should give (0, 0, 0)
o print_pose(diff_drive_forward_kin( (1000, 1000, np.pi/2), 1290, -1290)) #should
  give (1000, 1000, 90)
o print_pose(diff_drive_forward_kin( (0, 0, np.pi/2), 1290, 100)) #should give
  (62, 7, 283)
o print_pose(diff_drive_forward_kin( (0, 0, 0), 1991, 2075)) #should give (263,
  27, 12)
o print_pose(diff_drive_forward_kin( (0, 0, 0), 189, 2422)) #should give (-23,
  10, 312)
o print_pose(diff_drive_forward_kin( (0, 0, 0), 1249, 2598)) #should give (-11,
  152, 188)
```

Deliverables

- **Writeup: Discuss the limitations with this approach and the accuracy of the function you made. Half page. /5 (30%)**
- **Demonstration /5 (70%)**

Task 3: Simple Robot Teleoperation with Odometry (20%)

Using your differential drive forward kinematic solver from Task 2, we can now implement odometry based on the commands you give the robot. What you will do is assume that the robot starts at coordinate $x=0, y=0, \theta=0$ (looking down the x axis), create a simple program to let the user drive the robot, and along the way calculate where in the real world the robot ends up. Here are some tips

- Getting keyboard requires hooks into your OS/windowing system. I used pynput but feel free to use whatever you know. I setup events that set global variable states for moving left, right, forward, backward, for WASD, where an on_press event set the global variable to true, while on_release set to false. You don't read key states in your control loop, just read those variables.
- I created a simple control loop (e.g., like in task1) that set the robot wheel speed based on keyboard input (reading your global state variables). Every loop tick (I run at 10Hz) read the motor steps from the robot and use the forward kinematic solver to estimate the new robot location.
- Make some reasonable mapping from keys/key combinations to robot wheel speeds, nothing is prescribed but you should have flexible control of the robot.
- Print the new robot pose (position in the world) every second (10 ticks)

Once this interactive controller works, set your robot at a clearly marked start location and drive forward and backward, turn, etc., to observe how accurate the odometry approach is. My entire As1.3.py file (excluding As1lib.py) is 94 lines.

Deliverables

- **Writeup: Discuss the limitations with using odometry, including the sources of errors. How would you improve it? Half page /5 (20%)**
- **Demonstration /5 (80%)**

Task 4: Inverse Kinematics (15%)

Inverse kinematics has a well-earned reputation of being a difficult problem. Especially for non-holonomic configurations like a differential drive, many configurations may not be solvable with a simple motion – you may need a planner. For example, in this case to move to a target configuration, you should probably first turn in the right direction, drive there, then turn toward the goal orientation. We'll do this later. Instead we'll implement a much simpler form.

Write a function to calculate what would make the robot move a given distance at a given speed with a given delta turn. This is a solvable problem. For example, you could tell the robot to go 500mm at 100mm/s while turning a total of $\pi/2$ radians.

`def diff_drive_inverse_kin(distance_mm, speed_mm_s, omega_rad):` (left_steps_s, right_steps_s, left_steps, right_steps). The `distance_mm` is the absolute distance to be travelled, with `speed_mm` being the signed speed (negative means moving backward). The first two returned items represent the left and right wheel speeds in steps/s. The second two represent just how many steps each wheel should move overall (positive, unsigned). This does not move the robot at all, just performs calculations. Note the special case of turning on the spot (`distance_mm == 0`). Mine is 25 lines.

Test cases, note that your results may be slightly different based on your robot measurements.

```
o print(diff_drive_inverse_kin(130, 10, 0)) # should give (75, 75, 978, 978)
o print(diff_drive_inverse_kin(130, -10, 0)) # should give (-75, -75, 978, 978)
o print(diff_drive_inverse_kin(300, 50, 0)) # should give (376, 376, 2257, 2257)
o print(diff_drive_inverse_kin(200, 70, np.pi/4)) # should give (472, 582, 1348, 1661)
o print(diff_drive_inverse_kin(-200, 70, np.pi/4)) # should give (472, 582, 1348, 1661)
o print(diff_drive_inverse_kin(300, -40, -np.pi*2)) # should give (-134, -468, 1005, 3510)
```

```
o print(diff_drive_inverse_kin(0, 100, -np.pi*2)) # should give (753, -753, 1253, 1253)
o print(diff_drive_inverse_kin(0, 50, np.pi/2)) # should give (-376, 376, 313, 313)
o print(diff_drive_inverse_kin(0, -50, np.pi/2)) # should give (-376, 376, 313, 313)
```

Now that this works, you can connect it to your `move_steps` function from Task 1. Simply call the inverse kinematics function and use `move_steps` to drive the robot. For example:

```
move_steps(epuckcomm, *diff_drive_inverse_kin(300, -40, -np.pi*2), Hz=30)
```

Deliverables

- **Writeup: When does this chain, (diff_drive_inverse_kin to move_steps) , break? Think about the extreme values. Where should this be checked and fixed? Half page. /5 (40%)**
- **Demonstration /5 (60%)**

Task 5: Measuring Open-loop trajectories (20%)

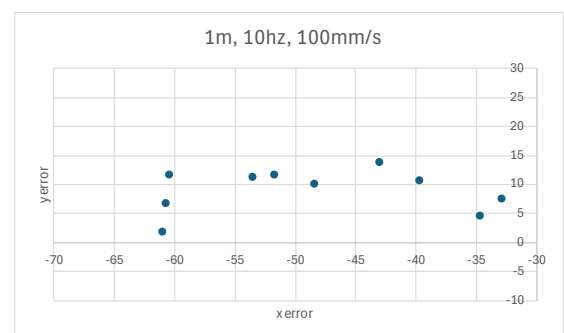
You will do a series of very simple experiments to learn about error in open-loop trajectories. For this part you will need to setup an experimental test bed that needs to be consistent throughout all your experiments.

- You need a clean, large hard surface like a table. Carpet or floors with bumps (e.g., tile) will not work. It needs to be at least 1.2m long and .6m wide, preferably bigger.
- You need to setup a mechanism for starting the robot from the exact same starting spot every time. I used electrical tape on my table, and placed a pencil mark on the robot and table.
- You need a mechanism for measuring coordinates on the table, the Δx , Δy from the robot. X is lateral and y is forward distance. Note that this is different from the robot's frame. I used a protractor and string to measure the polar coordinates which you can then convert into cartesian. Easier to do with 2 people.
- You should measure all distances from the same spot on the robot. In my case I measure the center of the front at start position vs. end position as shown.



Trials:

- Run your loop at 10Hz. Place your robot on the starting position and use `move_straight` to make the robot move forward 1000mm, at 100mm/s. Do this 10 times and record the resulting locations.
- Calculate the two components of the error, `errX` and `errY`, by subtracting your measured end point from the optimal end point. For each, calculate the average error (for X, and Y) and standard deviation (for X, and Y). Plot the error using a scatter plot. Hint: use Excel for all this. E.g., such as the plot on the right
- Repeat the above process, except process, except at 1Hz and 30Hz.



- Pick the best result (least error), and do another ten trials at 130mm/s, and take all measurements again.

Deliverables

- **Writeup:** A clear lab report showing your findings including your error statistics and graphs. No analysis just results. 1-2 pages. /5
- **Writeup:** What does controller Hz do to the accuracy of odometry? Why? Half page. /5
- **Writeup:** What does robot speed do to the accuracy of odometry? Why? Half page /5
- **Writeup:** In a point form fashion, two sentences each, list all significant sources of error that contribute to your final recorded robot position error, and explain why it happens and how it can be mitigated. /5

Task 6: More Complex Behaviors (5%)

This is an easy task, and more to highlight a problem with what we've been doing.

Code a robot behavior that (should) first move forward 0.5m, turns around, returns to its original location, then turns again to face forward – it ends in the same configuration where it started. Don't re-tune your calculations or measurements beyond what you did earlier, or you may have to re-do your experiments! Fix bugs of course, if you find them.

Next, imagine we want to make the robot to go forward for a meter, but stop immediately if it detects an object in front of it. How would you implement this using your current code structure? Write a paragraph explaining what you would do, and then discuss why this is a problem.

Deliverables

- **Writeup:** What does this highlight about using odometry for more complex behaviors? Half page /5
- **Writeup:** Imagine you wanted to do this behavior, but have the robot stop immediately if it detects an object in front of it. How would you implement this using your current code structure? Discuss why this is a problem. Half page /5

Code (5%)

Deliverables

Code: / 5