Assignment 2: Slightly Less-Basic Controllers

Assignment 1 was designed for you to get experience working with the robot, comfortable with kinematics, and starting to move from robot control to higher level planning. Unfortunately, the monolithic loop approach of AS1 quickly becomes too difficult to manage as we start to consider more complex behaviors.

In this assignment we will build a modular controller for our robot, building on AS1. If you did not finish AS1, now is the time to make sure everything works since you will need some of your code here.

We will start by focusing on structuring our code and rebuilding our simple controller in a modular fashion. Then we will build a robot that follows a line, but has to go around obstacles that it finds. Then we will finish the assignment by adding a design, human-robot interaction element to the controller.

The main goals of this assignment are:

- To be able to build and work with a tick-based robot controller model.
- To be able to build a closed-loop robot controller.
- To be able to design and implement slightly more complex behaviors using state machines.
- To be able to think about simple human-focused design elements of robot control.

There is a lot of initial coding for this assignment before things will work, even to reproduce your AS1 output. You will need to take time to get familiar with the paradigm; if you haven't done tick-based or update-loop design before, this may be a conceptual leap.

epuck library

Continue to use the epuck library from AS1.

Hints and helper functions

I created a new library, called epuck_lib, where I stored the robot parameters and helper functions that I wanted to pull from AS1, including unit conversions and kinematics.

Consider building a "debug" trace into your code, e.g., to print when entering or exiting a portion of your program, that can be turned on or off using a global or a passed parameter with default.

Task 1: Tick-based Robot Controller Skeleton (30%)

You will create a simple tick-based robot controller using OO. Your program will have the following modules:

Controller: manages the main loop and holds information to let other modules communicate with each other. Each module should have a way of accessing this information, e.g., I use `self.controller.robot`, etc.

Planner: high level planning of what the robot should do next, sets target movements for the navigator to use and manages state machines.

Navigator: takes a target movement and converts this into robot motion commands.

Robot: manages robot connection and communication, sends commands and gets sensor data.

The general flow is that the controller manages time, while the planner sets targets for the navigator, the navigator directs the robot, and the robot sends commands and updates state for others to use.

Abstract Base Classes

Start by coding up a skeleton of base classes for each module. I used the python abstract base class library to help implement these. This is not required but makes things simpler if you want to try different versions of your modules, and so I used base classes for the planner and navigator where things may change[1].

Following the tick-based design pattern, most modules have:

- `setup()`, called once after creation to do initial setup
- `update()`, called once per tick to reassess a situation and do needed updates
- `terminate()`, called to cleanup at the end of the program.

Create the following abstract base classes, each with their own `setup()`, `update()`, and `terminate()`:

- `planner.py`: `update()` returns true if it is still running, and false if it is done planning and the program should quit. Mine is 18 lines.
- `navigator.py`: Has `set_target()` to enable a planner to set navigation target and a `has_hit_target` boolean so others can see if the target has been reached. Mine is 25 lines.
- `robot.py`: Provides odometry functions `odom_reset()`, to re-center the robot, and `odom_update()`, which uses robot measurements to update odometry (both abstract and not implemented). Provides a public state variable (from `epuck_state`), and `robot_pose` result of odometry. Mine is 28 lines.

Initial Implementation

Use this structure to re-implement essentially what you did in AS1.

`controller.py`: manages your control loop, as in AS1, but without any planning, navigation, or robot features.

- `setup()`: takes a planner, navigator, and robot instance, and optionally Hz. Saves local pointers to these as public variables, and runs setup on each submodule. Further registers a pointer to itself in each submodule so they can find the controller.
- `start()`: runs the main program loop. In each loop iteration, calls the update functions of the other modules and sleeps to maintain the target Hz. The loop quits when the planner update returns false.[2]
- `terminate()`: cleans up and calls terminate on the sub modules.
- my implementation is 34 lines

`planner_move_once.py`: implements the `Planner` ABC. This takes a simple target as a tuple `(distance_mm, speed_percent, delta_theta_rad)` in its constructor. `update()` gives this to the navigator, which it monitors, and then returns false when the target is reached. 23 lines.

`navigator_diff_simple.py`: implements `Navigator` ABC. `set_target()` uses a tuple `(distance_mm, speed_percent, delta_theta_rad)`, using AS1 kinematics to solve how the robot should move. `update()` asks the robot to update odometry, tells the robot to move, monitors how much the robot

---

[1] There is opportunity for better OO design, e.g., using a standardized interface and list of executors. However, that may be a little too much overhead for this small assignment, and, it may add complexity that some find confusing when we're learning how to make robot controllers.

[2] Optional: my code monitors time passed using time.time_ns(), and auto tunes an error offset on the sleep calculation to maintain the target Hz over a 2 second window. This is not needed but cool, my controller.py with autotuning is 66 lines

has moved, and manages its `has_hit_target` variable. It should stop the robot motion when the target is reached. `terminate()` stops the robot movement. 43 lines.

`robot_epuck.py`: implements `Robot` ABC: The module should receive robot connection parameters, either in the constructor or setup. `setup()` creates the `EPuckCom` object, establishes the connection, enables sensors, etc. `update()` sends robot commands and updates its `state` variable. `terminate()` closes the connection and stops the robot. Be sure to implement the `odom_reset()` and `odom_update()` based on the e-puck's specifics, as in AS1. Note that the robot `state` contains both the sensor data *and* the motor commands. Convenient. 66 lines

This should now work and you can implement AS1's movement based on a kinematics target.

Hints:

Here is my file structure

```
├──── AS2
│   ├──── AS2.1.py
│   ├──── controller.py
│   ├──── epuck_lib.py  generic helper and kinematics functions from AS1
│   ├──── navigator.py  abstract base class for Navigator module
│   ├──── navigator_diff_simple.py  a simple differential drive navigator solver, as in AS1
│   ├──── planner.py  abstract base class for planner module
│   ├──── planner_move_once.py  a simple planner that sets a single target (distance, speed, theta)
│                                and configures the planner to do it, as per AS1
│   ├──── robot.py  an abstract base class for the Robot module
│   └──── robot_epuck.py  an implementation of robot class to manage the epuck robot connection
```

My AS2.1.py, excluding imports, is:

```
robot = robot_epuck.RobotEPuck("COM9")
planner = planner_move_once.PlannerMoveOnce( (300, 1, math.pi/2) )
navigator = navigator_diff_simple.NavigatorDiffSimple()

controller = controller.Controller()
controller.setup(planner, navigator, robot)
controller.start()  #blocking until done
controller.terminate()
print("Controller exited")
```

Deliverables

Demonstration /5

Task 2: Your First Planner (5%)

This should be easy if Task 1 is properly implemented. Quickly implement another planner, `PlannerMoveTargets`. Make it in its own file, and use your `Planner` ABC. Instead of taking a simple target, take a tuple of targets. Further, take a number of times that the targets should be cycled through. For example:

```
targets = ((200, 1, 0),
           (0, .2, math.pi/2)
           )
planner = planner_move_targets.PlannerMoveTargets(targets, 4)
```

This configuration will make the robot move forward, turn left, 4 times, to move in a square. As2.2.py should be similar to AS2.1 but with a different planner and target, the rest of your code should not change.

Deliverables

Demonstration /5

Task 3: Line-Following Robot (25%)

You will use your controller structure to now implement your first closed-loop controller- make a robot follow a line. You first need to make a new navigator that just moves at the given speed (instead of doing kinematics) to give full control to the planner, and then make a PID line following planner. Make the new planner and navigator as separate files `planner_linefollow.py` and `navigator_diff_direct_percent.py`

Make a simple navigator, `NavigatorDiffDirectPercent` that implements Navigator. It takes a target as a tuple representing the left and right wheel speeds (as a percentage of max), where negative means the wheel moves backward. It manages the robot odometry and robot movement speed. 29 lines.

Make a planner, `PlannerLineFollow`, that implements a PID line follower. The logic here is that the robot monitors the difference between the left and right ground sensor, trying to turn left or right to keep them relatively the same. E.g., if the right is darker than the left, the robot should turn right to get back on the line. In update, read the robot's ground proximity sensors, calculate the P, I, and D, and use these to generate an offset. Add this offset to one wheel speed and subtract from another to tune the turning. 38 lines.

Hint:

- PIDs are notoriously difficult to tune[3]. Start by setting KI and KD to 0, and tune KP until your robot turns fast enough to keep on the line. It may oscillate though around the line. Then, tune KD to reduce the oscillations. Finally, tune KI to smooth out the overall movements.

Download the provided sample robot course and print it over 6 sheets of paper. Ask the prof if you don't have access to a printer.

As part of this assignment, you need to devise some method to evaluate the quality of your line following. There are many things you could try, there is no single solution. Use this quality measure in your tuning.

Deliverables

Demonstration /5 * (60%)

Writeup: In your own words explain the impact of KP, KI, KD with regards to the line following behavior. Further, what limitations to PID controllers is imposed by the nature of the sensor we are using? What improvements or better sensor could you imagine using for this task? One page /5 (25%)

[3] In fact, if you do it in real work, there are advanced tuning methods and calculations you can do. This exercise, however, helps you really understand what is going on.

Writeup: Provide a description and explanation of the quality metric that you devised and implemented. Present the results, and compare your final result to an early result (e.g., your first line following before tuning) /5 (15%)

Task 4: Robot Behavior with states (15%)

Expand on your task 3 robot by making a new planner that uses a state machine to enable more complex behaviors. We have provided a very simple State machine ABC and example in `state.py`, which should simplify your implementation[4]. The provided file also includes documentation on how to use the class. My entire planner with all states is 137 lines.

Make a new planner, `planner_find_follow_line.py`, that implements a state machine for finding and following a line. Make the following states:

- `s_find_line`: Start in this state. Do some interesting movement to try and find a line, e.g., move in a wide arc, some random movement, etc. If a line is found, transition to `s_get_on_line`
- `s_get_on_line`: Assume that a line was found, try to do some movement to orient or move the robot to get back on the line. If the line was lost, or if you fail to get on the line after some time, transition back to `s_find_line`. Transition to `s_follow_line` when you determine you are on the line.
- `s_follow_line`: Implement your PID controller from Task 3. Check if we are on a line, if not, transition to `s_find_line`.

For each state, take the time to think through and plan out what should happen before deciding what to code for the robot. Try to be clever, and imagine what the robot will see in each state or situation. Build in special cases. Consider edge cases and try to implement them as best you can.

Deliverables

Demonstration /5 (75%)

Writeup: Generate a visual of your state machine. Try to include any states not reflected in the three above, for example, states within those (e.g., using if statements to determing the condition). Discuss any challenges or limitations you found with the state machine. /5 (10%)

Writeup: Explain the strategy you used for finding and getting on the line. Half page. /5 (15%)

Task 5: Robot obstacles (10%)

Add obstacle-avoiding features to your state machine. Leave Task 4 as-is in its own file, I suggest making `AS2.5.py` for this task and a new planner, `planner_line_with_obstacles.py`, and bring most of your work from Task 4 to start it.

Create a new state, `s_avoid_obstacle`, which avoids hitting an obstacle and turns away from it. Check for obstacles by monitoring the proximity sensors (e.g., make a helper function), and transition into this state where needed. When an obstacle is not found, transition back to finding a line.

---

[4] There are many excellent state machine libraries in python that one could use. However, they are often intended for more complex cases and so the overhead can be significant. Also, integration with a tick-based design pattern can cause a lot of overhead. Our provided solution should be simpler and serves our purposes of you understanding how to use state machines to make complex behaviors.

- You can implement this with a Braitenberg Vehicle approach. You can simply weight the values of the proximity sensors to increase or decrease wheel speed. That is, calculate some aggregate balance of the 10°, 45°, 90° sensors on the given side, and then add and subtract from wheel speeds. E.g., an obstacle on the right would increase right wheel and decrease left wheel.

While this works well, when a robot is following a line and finds an obstacle, it just goes the other way. Instead, it should go around the obstacle. Create a new state, `s_go_around_obstacle`, which attempts to go all the way around an obstacle, to get back on the line where it continues behind the obstacle.

- There are many ways to do this. One solution is to move forward while turning away from the obstacle, but turn back when you can't see it any more, thus skirting the obstacle. When you find a line again, then you are around the obstacle. Be careful to not decide that you are back on the line right away when it's the same spot – you can see how close you are to the original spot (e.g., using odometry) to make this call.

This task turns out to be really hard as the complexity of all the cases start to intertwine, and you start thinking of many exception cases. Try your best to tune out all the bugs. For example, states can share information with each other, or hints – perhaps if you lose the line you can keep track of which side you left on, and tell `s_find_line` which side to move, etc.

Deliverables

Demonstration /5 (80%)

Writeup: Generate a visual of your state machine, updated from the previous task. Discuss any challenges or limitations you found with the state machine approach. /5 (10%)

Writeup: Explain the strategy you used for going around obstacles. Half page. /5 (10%)

Task 6: Social Robot Behavior (10%)

For the last part of this assignment, you need to program some kind of personality or attitude into your robot. This is a design task which probably should take a lot more work and involve user-centered design. However, for this class you will just develop something ad-hoc to explore the ideas.

Assume that any obstacle the robot meets is a person. Before avoiding or going around the person, you should try to communicate something to the person using social robotics principles. For example, your robot can try to act annoyed, excited to see them, be aggressive to scare them away, etc. Use any of the robot modalities (motion, lights, sounds, etc.) that you like; this may take many states to accomplish.

Deliverables

Demonstration /5 (60%)

Writeup: Generate a visual of your state machine, including only the social robot behavior and not the rest. /5 (10%)

Writeup: Explain the strategy behind your behavior. Explain the personality and how you attempted to achieve it using your techniques. Explain how you developed your design. /5 (30%)

Code (5%)

Deliverables

Code: / 5