Student Name: Shukai Liu UoB Student Number: 1909499

1. **Briefly describe the architecture of your system for the line-following task. Identify which elements of the system have the greatest influence on the level of performance of your Romi to complete the line following task.**

1.1 Provide a flow chart, diagram or pseudo-code to help describe your robotic system in overview.

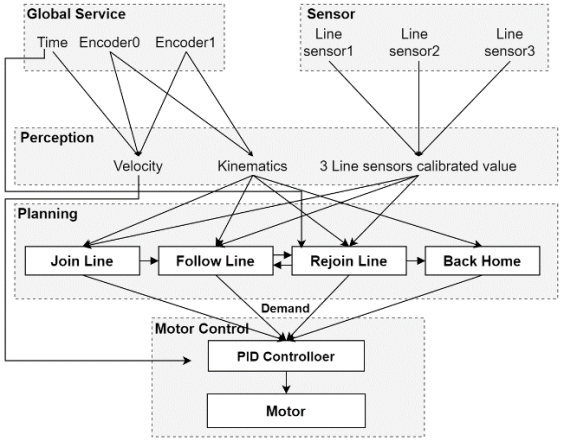

Figure 1. Architecture of following-line task system

Figure 1 shows the overall architecture of my system of line-following task. The system can be broken down into five modules: Global service, sensor, perception, planning, and motor control. The global service includes two basic services: time and encoder. Time service uses function *millis()* to keep tracking time and allows the system to do multi-tasks at once. The encoders on left and right wheels monitor the rotation of two wheels respectively. The sensor of the system is a QTR-3A reflectance sensor, which include three pairs of infra-red LED and phototransistor. In the perception module, the system calls the encoders' data to record kinematics information, call the encoders' data and time service to calculate the velocity of Romi, and read three sensors' values once calibration. The next module is planning, this module uses the information of perception module to generate the demands to control the rotation speed of Romi's two wheels. All movements of Romi are controlled by PID, the demand will be passed to the PID controller of motor control module, the PID controller can calculate the actual power required by the two wheels through the demand and the velocity of Romi, and uses this to control the motors.

```
void loop() {
  velocity_measure(20000);
  kine.update(count_e0, count_e1);

  switch (state) {
      case 0:    //join line
        pid_join_line();
        break;
      case 1:    //follow line (PID with probability)
        pid_follow_line();
        break;
      case 2:    //  determine if Romi is out of line
        //recheck_determine();
        rejoin();
        break;
      case 3:   //face to x direction
        face_x();
        break;
      case 4:    //straight line in -x direction
        straight_x();
        break;
      case 5:   //face to y direction
        face_y();
        break;
      case 6:    //straight line in y direction
        straight_y();
        break;
  }
```

Figure 2(a). Main loop of program

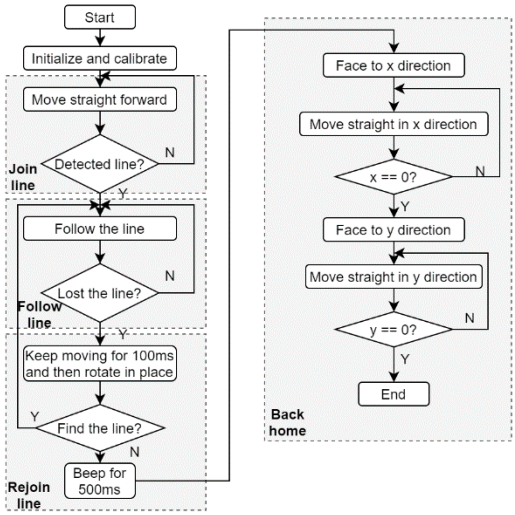

Figure 2(b). Structure of planning module

The planning module is the core of the system, it can be seen as composed of four functional modules: Join line, follow line, rejoin line, and back home. The program will switch between these function modules according to the task. The main structure of the program is shown in Figure 2(a), these functional modules are controlled by a *switch ()* function, where case 0,1,2 corresponding to join line, follow line, rejoin line respectively, and case 3,4,5,6 corresponding to back home. The details of these functional modules are shown in Figure 2(b):

- Join line: Romi will move straight forward after initializing and calibrating the sensors, and Romi will switch to *follow_line* module when the line is detected. Whether the line is detected is determined based on the calibrated sensors value. If the sum of the three values is larger than 400, the program will determine that Romi is on the line.

- Follow line: Romi will follow the line until the sum of the three sensor values is less than 100, which will make the program determine that Romi is off the line and switch to *Rejoin* module. How Romi follows the line will be described in 1.3.

- Rejoin: In this module, Romi will actively look for the line and determine whether the line is lost or the end is reached. If Romi finds the line again during this process, the program will return to the *Follow_line* module. Otherwise it will determine that Romi has reached the end and switch to the *Back_home* module after beeping for 500ms. The details of this module will be introduced in 1.2.

- Back home: This module controls Romi to return to the starting point. Romi's return path is to go back to x = 0 first, and then back to y = 0. The details of how Romi back home will be discussed in Q2.

1.2 Describe why you decomposed the line-following task into your solution the way you did

The task can be decomposed into three main part initially: join line, follow line, and back home. Between the *Join line* module and *Follow line* module, I didn't design transitional procedure of program. This can be achieved because I placed the sensor as close to the center of Romi as possible, so Romi can smoothly switch from moving forward to follow line motion.

In earlier versions of the program, Romi will start to go home directly when the line is not detected (sum of line sensors value < 100). Therefore, during the test, due to changes in light intensity, unevenness of the map, and stains on the map, Romi may start to go home before reaching the end point. To make Romi more robust and complete the task in poor test conditions, I design an interval module named *rejoin* between *Follow line* and *Back home.* In this module, Romi will rotate like a radar and scan the range of 80 degrees in front of it to find the line. Once the line is detected during this process, the program will switch back to the *Follow line* module. If no line is detected, Romi will rotate back to the original direction and enter the *Back home* module after beeping for 500ms. However, if Romi starts to rotate immediately after going out, it is very likely that he cannot stop at the end point, and may even rotate 180 degrees and return along the line. So before rotating, I make Romi continue to move for 100ms. So that after reaching the end point, it can continue to move forward a short distance and will not scan the line when rotating.

The details of how Romi back home will be discussed in Q2.

1.3 Which element of your system was the most successful, and why?

When I tested my Romi normally, I have 100% confidence that Romi can reach the green circle (the second smallest circle), and 70% are sure to reach the blue circle (the smallest circle). **And I tested Romi in assessment opportunity 2, Romi reached blue circle in all 3 chances**, but for better confidence, I chose

green circle in the test and scored 90, so I think all my system is successful. But the most successful part I think is *Follow line* part because when the light source conditions are good, my Romi will hardly get offline when following the line.

```
l_prob = (float)l_value / sum;
c_prob = (float)c_value / sum;
r_prob = (float)r_value / sum;
M = r_prob - l_prob;
if ( c_prob >= 0.7 || abs(M) <= 0.3) {    ////straight line
  demandL = -7;
  demandR = -7;
}
else if ( M > 0.3 && c_prob < 0.7) {      //rotation clock
  demandL = -20 * M;
  demandR = 20 * M;
}
else if ( M < -0.3 && c_prob < 0.7) {     //rotation anti-clock
  demandL = -20 * M;
  demandR = 20 * M;
}
```

Figure 3. Core of *Follow line* module

The core code that controls Romi to follow the line is shown in the Figure 3. It's simple but very effective. After calculating each sensor reading as proportional to the total and the output measurement ($M = P_{right} - P_{left}$), unlike the labsheet, to move more effectively, if the proportion of central sensor is larger than 0.7 (e.g. 0.1, 0.8, 0.1 for left, central and right sensor) or the absolute value of measurement is less than 0.3 (e.g. 0.33, 0.33, 0.33 for three sensors), the program will control Romi to move forward. Only when the proportion of central sensor is less than 0.7, Romi will rotate according to the measurement value. To complete the movement of following the line, program only generates two kinds of motion: move forward and rotate around center. Because of the characteristics of well-set PID controller, Romi can smoothly switch between the two movement modes. Zone3 is composed of polylines, which is more difficult to pass than zone2. To pass zone 3 perfectly, the output demand of rotation is set to be $demand = c * M$, so that Romi can control the output power of rotation according to the different angles on the map.

1.4 Which element received the most development time, and why?

The part I spent the longest time on was how to make Romi go home precisely, this part will be discussed in details in Q2.

Except implementing a specific function, the element received the most development time is rebuilding the code. In the early stages of development, I paid more attention to how to achieve a certain function or how to solve a specific problem, but neglected the task as a whole. This caused my code to be very cluttered, different function modules intermingled together, with a lot of repeated code and variables, and even many layers of *if()* functions nested together. It makes debugging to be extremely difficult and complicated. So I spent lots of time to reconstruct my code and change *if()* functions with many conditions to *switch()* functions. This made me more efficient when I optimized the system later.

1.5 What working practices would you recommend to others engineering a robotic solution for the same task, and why?

The point that I recommend the most is, before we start writing code, we should carefully study and plan the task first, decompose the task into some expected modules and build an overall structure for the program. This can make debugging and optimizing much easier later and make the program more readable. Another suggestion is to use serial plotters and serial monitors as much as possible during testing and debugging. When I was testing Romi on the map, I also use computer to monitor Romi's velocity, encoders'

count, kinematics information and other parameters. This allows me to more intuitively feel the influence of different parameters on the overall performance when actually performing the task, making parameter adjustment more efficient. And we should also aware that the battery power will affect Romi's performance to some extent.

2. Describe in detail a specific challenge or success relating to a sensor, motor control or a sub-system (e.g. behaviour) for your Romi robot.

The biggest challenge I met was how to make Romi go home precisely, how to make Romi get to the blue circle as much as possible. To achieve this, this challenge can be decomposed into some sub-challenges. I will discuss some of them and introduce my solutions as follow:

- **What back-home routine can make the algorithm the simplest**

In the original design, I planned to let Romi rotate directly to the direction of the starting point after reaching the end point and then return in straight line. However, because there are many situations for the direction of zone 4, I found it was difficult to calculate the angle required to turn directly to the starting point. Therefore, I design another back-home routine that can make the algorithm much simpler.

Solution: Assume Romi 's kinematics information is (x, y, θ) after reaching the end point, my basic idea is to clear θ first, then clear x, and finally clear y as Figure 4 shows.

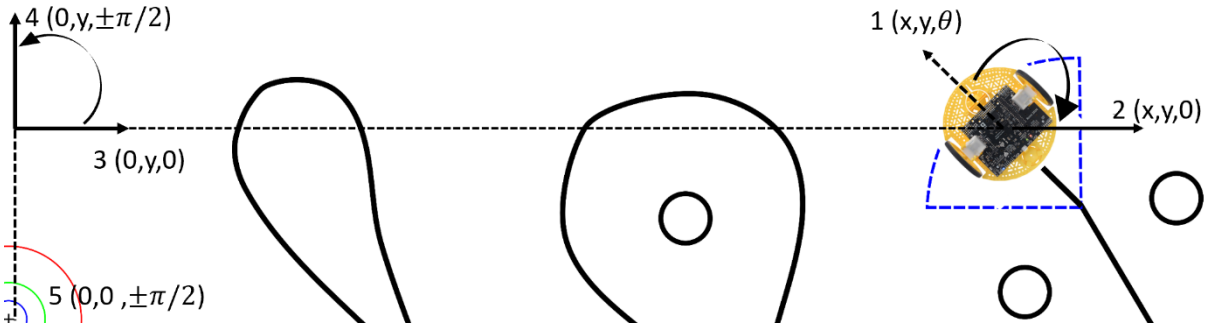

Figure 4. Back-home routine

Step 1: After the program determines that Romi has reached the end, Romi will rotate $-\theta$ angle to align the initial orientation (x direction), after this step the kinematics info of Romi is $(x, y, 0)$.

Step 2: Romi moves backward along the x direction until x=0. After this Romi's position becomes $(0, y, 0)$

Step 3: Romi rotates $\pm$90 degrees to align its back with the origin point, the direction of rotation is determined according to the sign of y. After this Romi's position becomes $(0, y, \pm\frac{\pi}{2})$.

Step 4: Romi moves backward along the y axis until y=0. After this Romi's position becomes $(0,0, \pm\frac{\pi}{2})$.

The only disadvantage is this routine requires more time than back home directly, but it is extremely simple and robust. Its two main advantages are: 1. It can share a same *kinematics* class, which means I only need to create one *kinematics* class in my code rather than create *kinematics.followline* and *kinematics.backhome*. 2. It doesn't require any calculation, and it is easy to determine whether Romi has reached the expected position or angle in each step. For example, to determine whether Romi has aligned with x direction, I only need to check whether $\theta > 0$ or $\theta < 0$ according to its rotation direction, it is shown as Figure 5.

```
face_x_theta = kine.get_theta();
if ((face_x_theta > 0 && rotate_dir == 1) || (face_x_theta < 0 && rotate_dir == 0))
```

Figure 5. The condition for determining Romi's alignment in the x direction

- **How to make Romi's trajectory completely straight**

To return to the initial position from the end point, Romi needs to move a long distance of straight line, if the line is not straight enough, it will seriously affect the precision of back home. However, if only input the same velocity demands to the PID controller, the speed of the two wheels cannot be guaranteed to be the same, which may cause the change of Romi's movement direction.

```
straight_x_theta = kine.get_theta();
if (straight_x_theta < 0) {
  demandL = 10.2;
  demandR = 10;
}
else if (straight_x_theta > 0) {
  demandR = 10.2;
  demandL = 10;
}
```

Figure 6. Straight-line code

Solution: To make Romi be able to create a completely straight trajectory when it moves along x direction, I wrote the code as Figure 6 shows. The program will detect the direction of Romi's movement in every loop. If Romi moves to the left, the program will make the left wheel rotate faster; if the direction of movement is to the right, the right wheel will rotate slightly faster. In this case, the rotation speed of Romi's two wheels will reach dynamic balance as Figure 7 shows.

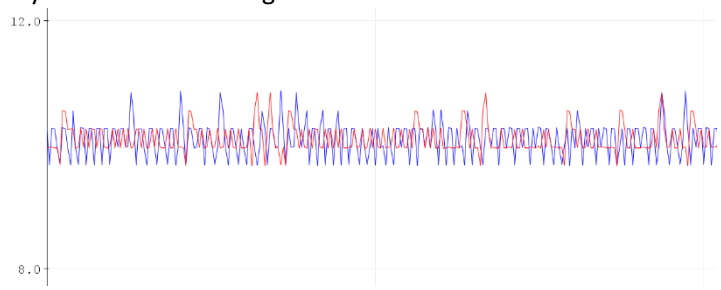

Figure7. Serial plot of the velocities of two wheels

- **How to minimize the error of kinematics**

The accuracy of kinematics has a great influence on whether Romi can return to the blue circle. I spent on a lot of time on minimizing the error of kinematics. Except the kinematics algorithm itself, according to my experience, I think there are two points that are easily overlooked that will affect the accuracy of kinematics. The first point is that we can't let Romi have a sudden change in speed, this can be solved by adjusting the parameter c in $demand = c * M$ in **1.3**. Another important point is that the time required for each loop of the program should be minimized as much as possible. For this reason, I have simplified the program as much as possible, and removed all *delay()* functions in the program.

- **Other consideration: PID**

PID controller is also another essential element of line-following task, its quality directly relates to the task performance. Figure 8 shows an example of my PID controller's performance.

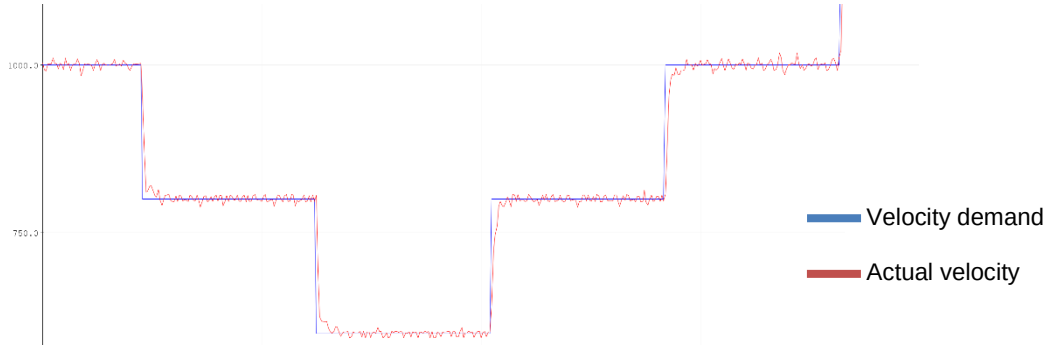

Figure 8. The serial plot of PID demand and actual velocity

3. Describe a challenge in robotics you would like to further investigate. Relate your ideas to your experience of working with the Romi for the line-following task.

The line-following task with Romi is the first time for me to work with a microcontroller like Arduino. This experience made me very interested in doing further development with Arduino. In fact, I have conceived some projects recently, and the project I most want to achieve is to make a writing robot based on Arduino. This is because my final-year project is about skill-transfer, and it will be very interesting if I can achieve writing skill-transfer on the writing robot made by myself in the future.

- *Task: Make a robot arm that can write or draw with a pen.*
  The primary goal is to allow the robot to draw according to pictures or planned trajectories, and the further goal is to enable the robotic arm to imitate human handwriting through skill transfer.

- *Software: Trajectory generation, trajectory analysis and motor path planning, motor control.*
  Whether it is to let the robot arm draw according to the picture or to imitate the handwriting, just using the Arduino IDE is not enough. In my conception, I will use Arduino to control the robotic arm, and use the host computer to generate the trajectory and commands. I can perform inverse kinematics analysis on the trajectory through Matlab to get the joint rotation angle and then pass the motor command to the Arduino; Or I can use G-code to describe the trajectory and run it on Arduino by Grbl.

- *Hardware: Robot arm, motor, microcontroller, sensors (encoders)*
  I have done some investigation on the structure of robot arm, and there is a mechanical structure named CoreXY is perfectly suitable for making a writing robot. And I can use RC servo or stepper motor to actuate the robot arm. I also need sensors to monitor the rotation of motors.

- *Environment: Writing on a 2D board or paper.*

Study design:
I plan to build the robotic arm model first, then study how to use Arduino to achieve precise control on the robotic arm, then explore how to generate trajectory with a computer, and finally convert the planned trajectory into motor command and control the robot arm to draw the trajectory.

Performance evaluation:
There is no doubt that the most important evaluation for writing robot is whether the output trajectory can consistent with the planned path. So, I will use the sensors to detect the rotation of motors and calculate the position of the end effector, and use this to analyse and optimize the accuracy of the robot arm.

Inspiration from the experience with Romi:
In line-following task, I gained the experience using motors and encoders, which will help me to use the motor to control the robot arm and record the position of the motors. But unlike the previous task, in this task I need to precisely control the angle and speed of the motor rotation, I will spend time working on this and study the hidden complexity behind the motor control and encoders. I also learned the basic idea of PID control in line-following task, this can help me control the robotic arm better and make the robotic arm move more smoothly.