电子设计“基于路径规划的智能导航车”总结报告

曹展翔 付欣旺 明陈林 徐昕翊

目录:

一、引言

近年来，随着物联网技术和人工智能的飞快发展，越来越多的智能技术融入到生产和生活当中，推动了各行各业的发展。智能小车作为智能机器人的典型载体代表，可以通过控制器编程实现自动控制或者远程控制，是智能技术推进社会发展的重要助力发明。本次电子设计拟通过智能路径规划，使得小车具备一定的智能引导作用，帮助或者代替人类实现在复杂环境中的行动与探索。恰好本次电子设计课程提供了小车的模型以及开发平台，所以本组选择从智能小车的角度出发，更进一步地思考相关方向的细节。

二、选题背景

作为一款能够智能路径规划的导航车，本组依据社会背景，以盲人为对象进行开发。中国拥有最大的盲人群体，在 2017 年已经达到了惊人的 1700 多万人次；但现实生活中占用盲道、导盲犬失效等事件层出不穷；而且盲道上容易出现堆积共享单车、盲道穿过墙壁、树木甚至窨井的现象，对盲人正常出行造成了很大的压力。在市面上的盲人导航往往是由语音播报来告知盲人前进方向，而这样只预先收集数据做出的大致的方向指示，不仅精度不高，而且由于前进风险的不可预知，很难避免碰撞或受伤；而导航车正是起到了探路以及规避风险的作用。

以上是本组构想中的导航车的作用；在本次课程实践的过程中，我们希望小车能够实现按照 *PC* 端规定的路线前进，避障，提示等功能。

三、设计思路

对于小车本身，本组预期实现底层基本控制电路和程序，完成对于小车控制的底层接口。然后实现通过移动端对于小车进行控制的功能，以及小车通过接收蓝牙数据进行相应的移动，并按照指定路径前进的功能。同时小车能够探测周围环境的障碍，并能够通过蓝牙将相应的信息传回给电脑端，即能够完成实时交互。同时 *PC* 端能够处理障碍信息并对路径进行重新规划。

对于实验环境的设置，考虑到需要对实际情况进行拟合，所以设定一个理想的道路（$maze$），为小车的运行提供理想的环境。在这样的条件下， 本组想要实现的最终功能为模拟智能交通系统的流程：通过外部的摄像机等设备收集道路数据，在电脑端（对应于云端）对数据进行处理，以获得前进的整体路线；将路线通过蓝牙发送给小车，小车依据路线进行规划，之后进行智能行驶，实现直行、拐弯等基本功能。除此之外，额外的功能还有：面对障碍等紧急情况，能够进行局部的路线规划，自动规避障碍；在运行过程中能够对于特定情况发出声音提示；向后距离检测功能：后方有一个距离传感器，可以检测后向物体的距离（这里对应于盲人）；当电脑端有指令输入时，小车能够依据指令完成运动状态的修改并执行。

整体上本组希望能够实现一个小车-电脑端的互联系统，通过电脑端能够控制小车，同时传输或者接收相关信息；在小车端，其能够按照规划好的路径移动，并且能够实现语音提示，自动避障、后向距离检测等功能；同时能够和电脑端交互，接受电脑端的提示和控制。

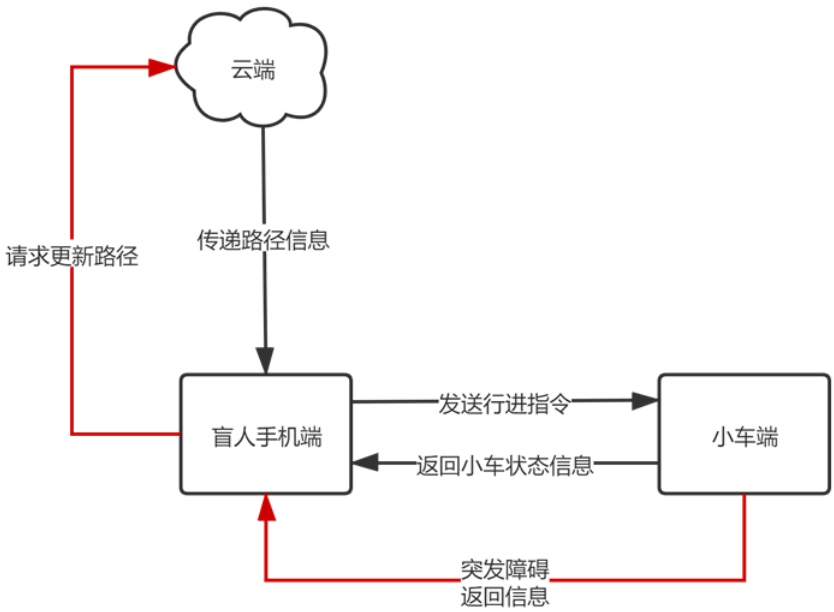
本组希望小车能够在追求足够智能的同时兼顾实用性和鲁棒性，使之能满足于更广泛的应用场景。

三、方案比较与选择

目前，基于 *Arduino* 开发板的智能小车配合以超声波模块，已经可以实现较为成熟的超声波避障功能，相对成熟的技术和较低廉的成本让其具有极大的市场和竞争力。相较于传统的导盲方式，虽然导盲犬也能做到带路以及探路的作用，但导盲犬也只能在其熟悉的地方正常工作；而导航车却能够带领盲人去往未知或不熟悉的地方，并且导航车在熟悉位置的循迹功能会更加强大精确，可以避免许多随机偶然事件。而目前最为前端的四足机械狗，其高昂的价格就可以说被排除在量化生产之外。此外，对于一些更加方便盲人使用的智能手杖和智能眼镜，虽然便捷性方面智能小车无法媲美穿戴型装备，但是智能小车可以根据先验信息，提前规划合理路线，完全比只是针对于避障和处理实时信息的穿戴型装备更加智能。另外，智能小车以其独立性和更加灵活的特点，势必能适用于更加广泛的场景，其在无人探索领域拥有比其他辅助型器械更加强大的功能。

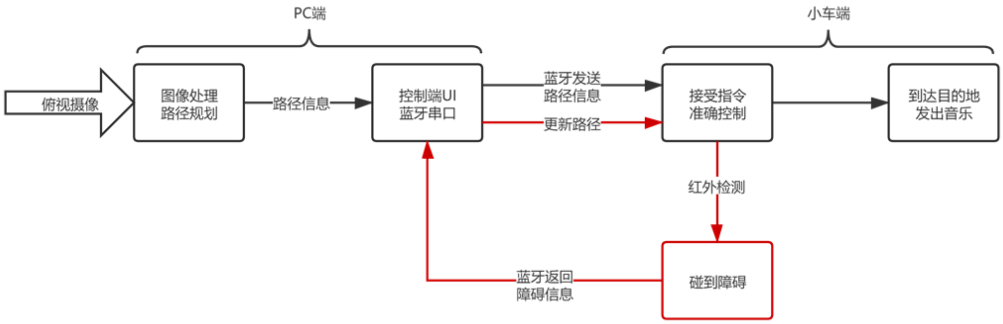
四、课题简介

本组想要模拟的整个流程如下图所示：改图的主要节点分为三个部分：云端，手机端和小车端。云端表示数据中心对于网络数据流的处理以及地理路径的定位，手机端的盲人控制小车，和小车交互的媒介，而小车端是具体的执行者，也就是为盲人服务的主要载体。



模型的主要逻辑是：在盲人输入想要前往的位置时，云端会进行路径归回，为小车提供路径信息；在盲人手机端获得信息后，再和盲人引导车进行交互，向小车发送路径信息和相关指令；同时小车不断向盲人的手机端返回状态信息，附带语音提醒，为盲人提供指引。如果遇到突发的障碍，小车端将会向盲人端发送信息，然后手段会和云端进行信息交互，请求新的路径规划，由此进行循环。

在本次电子设计过程中，考虑到需要充分利用时间以及场地的资源，尽可能降低成本，同时最大化小车的执行功效。所以本组构建了一个简化的模型，具体结构如下图所示：



这里本组首先在实验室中搭建了一个黑框地图（具体如下图所示），设置起点和重点，同时通过俯视拍照显示，其作为路径规划的地图，用来模拟真实的地图，俯视拍摄相当于卫星对于地图的直观处理。然后通过计算机对图像进行处理，得到最终路径。

得到小车前行的路径后，利用制作好的 UI 界面，通过蓝牙串口向小车发送指令，使小车按照规划好的路径进行行驶。如果路途通畅，小车顺利到达终点，则小车会在重点播放音乐，通知已经到达的消息。如果小车遇到障碍，则会通过蜂鸣器和 LED 信号对盲人进行提示，然后向 PC 端发送新的路径规划请求，这是 PC 端将会以小车的当前位置作为新的起点，重新进行路径规划，然后发出对应的指示使小车完成整个路径的行进。

五，细节功能介绍

1，小车的组装和基本编译平台 CCS 的应用：

在实验开始的前期，本组首先进行了小车的组装和 CCS 的学习。在最开始的学习过程中，本组发现对于小车硬件部分（$launchpad$）的学习，需要花费一定的时间。所以选择首先利用 TI 官方准备好的 $motor, tachometer, UART$ 等库完成对于小车的基本控制，然后在完善小车硬件控制的同时，继续进一步学习底层原理。

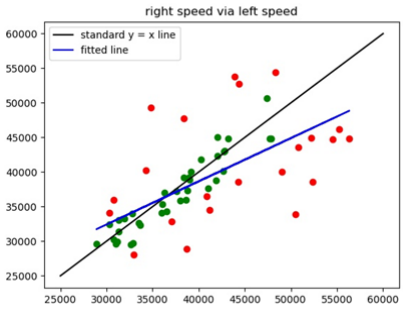
2，数据处理：

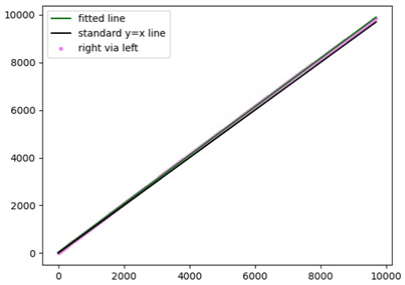
当本组完成装车以及对 CCS 编译器有了基础的了解之后，通过调用 TI 官方的库，已经可以实现对小车的基本控制了。这时本组使用了库中 $tachometer$ 库函数对小车行进的数据进行判断，通过让小车前行一段时间，判断，得到返回的左右轮转速和旋转圈数的数据。通过直接对数据进行观察，可以发现：即使在设定转速相同的情况下， 左右轮转速还是出现了很多奇异的差值，而且极不

稳定。在此基础上，本组使用 *python* 对点进行拟合处理，通过 *txt* 文档进行数据的保存和输入，实验代码如下图所示：

```
x = np.linspace(min(l_s), max(l_s), 300)
plt.scatter(l_s, -r_s, label='right via left', c='violet', s=10)
output = np.polyfit(l_s, -r_s, deg=1)
y = x * output[0] + output[1]
plt.plot(x, y, label='fitted line', c='green')
plt.plot(x, x, label='standard y=x line', c='black')
plt.legend()
plt.savefig('./l_r_step.jpg')
plt.show()
```

得到如下所示的图像：（这里左图为左右轮速度的拟合图像，右图为左右轮旋转角度的拟合值，也即绝对距离的拟合值）



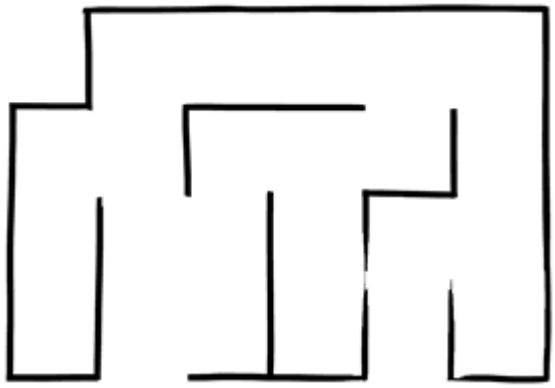


其中左图为转速计测量的左右两轮的转速点图，其中横轴表示左轮转速，纵轴表示右轮转速，红色的点表示在设定速度相同的情况下，左右两轮的转速误差比例超过 10% 的点（即误差很大的点），黑线表示理论曲线（即左轮转速等于右轮转速：y=x），深蓝色表示拟合的曲线。可以发现，使用测速计返回的左右轮速度是十分不可靠的，误差非常大。这里认为可能的原因是：由于电机是由电平脉冲控制的步进电机，可能会出现不稳定的情况。

右图为左右两轮旋转的角度进行拟合的结果，图中深蓝色的线为拟合曲线，黑色的线为标准曲线，具体代表左右轮旋转圈数的点被蓝线覆盖了，可见左右旋转圈数的线性性较好，之后可能可以作为调节车身直行姿态的判据。

3，迷宫骨架提取

为了模拟具体的导航，本组设计了一个较为简易的地图，用于路劲规划。为了方便图像处理，在地图的正上方进行拍照，然后对得到的图像进行简单的处理和去噪：首先通过 *python* 的 *cv* 库读取灰度图像，然后利用 *halcon* 库完成骨架的提取，消除多余像素点的干扰（例如地图中的地砖之间的缝隙），完成 0（黑） 和 255（白）的二值化。最终得到的效果是将上述的实际图片转换为黑白骨架，如下图所示：



4，路径规划算法

在获取骨架之后，可以通过路径规划算法得到最终的可行最短路径。原理如下：

（1）对图像骨架进行预处理：

标记起点，终点以及小车的位置和小车的占用范围。这里小车在图像上的大小是有所考量的，要尽可能保证小车的大小和路径的宽度相近，这样可以减少对于路径搜索的困难；或者换一种说法，没有人希望得到的路径在白色的部分进行来回折叠。所以这里最直接的想法就是通过模拟一个“小车”在路径中进行试错，最终在其到达终点时，只需要保留从起点到终点的那条路径就可以了，其余的试错路径可以放弃。同时这也在侧面说明了，在最终路径出现之前，需要标记所有小车移动过的点。在使用 $python$ 处理该算法时，为了减少时间复杂度，考虑增加空间复杂度，所以重新开辟了一个 $(image.shape[0], image.shape[1], 3)$ 的张量 $mark\_map$，用于保存标记 （$mark$， 服务于广度优先搜索算法），以及某点的前一个点坐标。这里的 map 使用一个类完成，方便对于整个算法进行统筹兼顾。

```
class car_map:
    image: np.array
    car_center: list
    car_length: int
    mark_map: np.array
    bfs_queue: list
    route: list
```

（2）路径规划算法：

对占有面积的“小车区域”使用广度优先搜索的方法，从起点开始上下左右四方向搜索（就如同小车在图像中运动一样），搜索步长设置为车身的像素长度；即只移动小车的中心点，然后通过检查小车面积占据的方位内，是否有像素点为 0 来判断小车是否碰到障碍，将没有障碍位置的可行路径进行标记，同时记录到达该点的前一个点的坐标。如果判断小车行驶到终点则退出搜索，然后通过回溯得到从起点至终点的最短路径。这里起点的灰度像素值设置为 （255 + 127） / 2 = 191，相对的，终点像素设置为 （255 - 127） / 2 = 64（这里没有额外的含义，只是用来表示区分）。这里通过 BFS 算法得到的路径，就是整个地图的最短路。最开始时曾考虑使用 DFS 进行遍历，速度可能会比 BFS 更快，但是得到的结果未必是最短路。具体的函数代码如下所示：

```
    # using BFS algorithm to get the best way to the end line
    def bfs_route(self, rate):
```

```
        self.bfs_queue.append(self.car_center)
        self.mark_map_assignment(1, self.car_center[0], self.car_center[1])
        while len(self.bfs_queue) > 0:
            temp_x, temp_y = self.bfs_queue[0][0], self.bfs_queue[0][1]
            self.car_center = [int(temp_x), int(temp_y)]
            # print(self.bfs_queue)
            # print(self.car_center)
            # self.init_car_in_map()
            # self.show_map(5)
            # self.destroy_car_in_map()
            mark = self.mark_map[temp_x, temp_y][0]
            del self.bfs_queue[0]
            if self.move_vertical(rate):  # go upward, car center modified
                if self.mark_map[self.car_center[0], self.car_center[1]][0] == 0:
                    self.mark_map_assignment(mark + 1, temp_x, temp_y)
                    self.bfs_queue.append(deepcopy(self.car_center))
                    if self.judge(64):
                        break
            self.car_center = [temp_x, temp_y]
            if self.move_vertical(-rate):  # go downward
                if self.mark_map[self.car_center[0], self.car_center[1]][0] == 0:
                    self.mark_map_assignment(mark + 1, temp_x, temp_y)
                    self.bfs_queue.append(deepcopy(self.car_center))
                    if self.judge(64):
                        break
            self.car_center = [temp_x, temp_y]
            if self.move_horizontal(rate):  # turn left
                if self.mark_map[self.car_center[0], self.car_center[1]][0] == 0:
                    self.mark_map_assignment(mark + 1, temp_x, temp_y)
                    self.bfs_queue.append(deepcopy(self.car_center))
                    if self.judge(64):
                        break
            self.car_center = [temp_x, temp_y]
            if self.move_horizontal(-rate):  # turn right
                if self.mark_map[self.car_center[0], self.car_center[1]][0] == 0:
                    self.mark_map_assignment(mark + 1, temp_x, temp_y)
                    self.bfs_queue.append(deepcopy(self.car_center))
                    if self.judge(64):
                        break
            self.car_center = [temp_x, temp_y]
```

同理这里的函数需要进行判断，在遍历车身的像素点确认小车是否碰到障碍时（或者是否触碰到起点和终点），可以选择只检测小车的四周边沿；因为这种检测方式在当前环境下和遍历整个正方形像素点的效果是等价的，即不可能出现只有黑色像素点在小车内部而不触及小车边界的情况，所以选择只遍历边界的效果更好，能够加快算法的执行速度，降低了时间复杂度。

（3）寻找最终路径：

在算法到达终点退出之后，可以利用终点的坐标，通过寻找 *mark_map* 上标记的路径上每一个点对应的前一个点，完成对于整个路径的提取，实验代码如下所示：

```
    # after bfs, get the shortest route
    def get_route(self):
        self.route = []
        while True:
            temp = deepcopy(self.car_center)
            temp[0], temp[1] = int(temp[0]), int(temp[1])
            self.route.append(temp)
            if self.mark_map[temp[0]][temp[1]][0] == 1:
                break
            self.car_center[0] = int(self.mark_map[temp[0], temp[1]][1])
            self.car_center[1] = int(self.mark_map[temp[0], temp[1]][2])
        self.route.reverse()  # from the beginning to the end
```

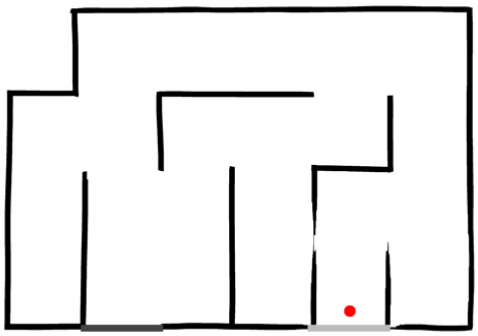
(4) 简化路径:

最终在得到路径（*route*），可以发现在同一条直线上可能出现很多的点，所以这里可以选择将每一条直线上的冗余的点去掉，只保留转角位置的点，使路径更加简化。实验代码如下所示:

```
    # cut out the points in one line, only keep the points on the turning
    def simplify_route(self):
        original_route = deepcopy(self.route)
        self.route = []
        for i in range(len(original_route)):
            if i == 0:
                self.route.append(original_route[i])
            elif i == len(original_route) - 1:
                self.route.append(original_route[i])
            else:
                if original_route[i][0] == original_route[i - 1][0] \
                        and original_route[i][0] == original_route[i + 1][0]:
                    continue
                elif original_route[i][1] == original_route[i - 1][1] \
                        and original_route[i][1] == original_route[i + 1][1]:
                    continue
                else:
                    self.route.append(original_route[i])
```

(5) 绘制路径图像和动态图:

至此已经获得了路径图，然后可以利用代码完成运动路径图像或者动图的绘制。在动图绘制的过程中可以添加采样，使直线上的点更加密集，动画效果更好。程序得到的图像如下图所示:



(6) 导出指令文档:

在得到了简化的路径之后，为了服务 PC 对于小车的控制，采用相对角度进行判别（由于陀螺仪是在每次上电后进行角度的初始化，所以绝对角度指每次执行或者转向依据最开始的角度进行修偏；而相对角度是指依靠每次转向后的角度进行修偏，而非初始的角度）。依据简化的路径，可以得到每次小车转向的方向（左右）以及每次前进的距离，然后通过："角度"+"速度"+"距离"的方式将指令导入 txt 文档中。

5，蓝牙通信和陀螺仪模块

在介绍蓝牙通信和陀螺仪模块之前，首先介绍一下$MSP432P401R$板子的端口配置原理。第一次接触$MSP432P401R$板子，我们组就在谷歌、$CSDN$、国外论坛等等平台上面搜集了较为全面的资料。其中如何设置端口的$GPIO$部分是大部分资料重叠的部分，也是我们想要使用$MSP432$的必要的预备知识。$MSP432$数字I/O模块拥有较多的I/O模块，并且每个都可独立编程。用程序的语言来说就是，在每个独立的I/O接口部分都有相应的寄存器，当用户需要此端口实现相应的功能时，我们只需要在相应的寄存器中设置一定的值，而板子会根据这些寄存器的值配置端口功能。 相对比较常用的设置为：通用$GPIO$的I/O功能、上拉电阻或下拉电阻的设置，初始状态的寄存器。相对特殊的是前六组端口的$X.2$，$X.3$端口，其可作为通信端口，实现像遵循$UART$或$I2C$的数据通信。

先以简单的 $GPIO$ 模块举例，首先我们需要确认 $PXSEL0$ 和 $PXSEL1$ 寄存器，将想要使用的寄存器对应位置 1。然后就需要设置 $PXDIR$ 方向寄存器，确认是输入还是输出方向（位=0是输入方向，位=1是输出方向）。若没有其他的特殊要求，普通的 $GPIO$ 接口的设置就已经完成。

对于像蓝牙串口和陀螺仪这样需要 $UART$ 通信的端口，首先我们需要将端口选择位 $PX.2$，$PX.3$，将端口的寄存器值配置为通信状态。我们根据所给的 $UART1$ 实例完成了对于蓝牙串口的调试（接线在 $P2.2$，$P2.3$），但是陀螺仪（接线在 $P3.2$，$P3.3$），却完全收不到信息。本组在仿照 $UART1$ 编写的 $UART2$ 文件中将所有引脚都修改后，还是不起作用。后来，我们仔细地参照用户手册研究了底层原理，发现是由外部设备引起的外部中断的程序是不一样的，在修改后陀螺仪终于将实时的车身姿态数据返还给了 PC 端。以上是 $UART2$ 中断入口编号的关键代码。

```
  EUSCI_A1->CTLW0 = 0x00C1;
                            // set the baud rate
                            // N = clock/baud rate = 12,000,000/115,200 = 104.1667
  EUSCI_A1->BRW = 104;      // UCBR = baud rate = int(N) = 104

  EUSCI_A1->MCTLW = 0x0000;   // clear first and second modulation stage bit fields
// since TxFifo is empty, we initially disarm interrupts on UCTXIFG, but arm it on OutChar
  P2->SEL0 |= 0x0C;
  P2->SEL1 &= ~0x0C;          // configure P2.3 and P2.2 as primary module function
  NVIC->IP[4] = (NVIC->IP[4]&0xFF00FFFF)|0x00400000; // priority 2
  NVIC->ISER[0] = 0x00020000; // enable interrupt 18 in NVIC
  EUSCI_A1->CTLW0 &= ~0x0001; // enable the USCI module
                            // enable interrupts on receive full
  EUSCI_A1->IE = 0x0001;      // disable interrupts on transmit empty, start, complete
```

（1）蓝牙模块：

本组很早就意识到，利用l蓝牙实现小车和 PC 端以及手机的通信对于小车的远程控制和方便的调试有着重要的意义，所以很早就开始进行蓝牙的调试。对于蓝牙模块的选择，我们没有选择 TI 公司内置昂贵的 $CC2650$ 模块，而是选择了更加便宜的 $HC-05$ 模块。虽然性能可能不若 $CC2650$ 模块那样优秀，但对于数据传输是足够的。最终我们也发现，使用 $HC-05$ 模块作为通信的基础模块没有任何问题，和 $python$ 编写的串口进行数据通信没有出现任何数据问题，表现得很稳定。

这里本组通过 TI 官方提供的 $UART$ 库完成蓝牙和电脑的调试。蓝牙 $HC-05$ 的接口主要有 TXD（用于发送），RXD（用于接收）已经高低电平，对应于 $launchpad$ 的连接。对于 $HC-05$ 模块，我们了解到它是使用 $UART$ 协议通信的，基于具备 $UART$ 相关知识的基础上，本组再次回顾了其具体的传输协议，并且我们发现官方对于 $UART$ 串口已经有了 $UART1$ 库，于是我们尝试调用该库对蓝牙模块进行调试，首先利用手机进行调试。将蓝牙模块连接至对应的引脚后，即 VCC 接 $5V$，GND 接 GND，RXD 接 TXD，TXD 接 RXD 后，使用手机上的蓝牙串口 APP 进行连接发送后，发现其未正常接收到信息。于是我们怀疑是蓝牙的波特率、主从模式等可能设置有误，于是我们找到 USB 转 TTL 的接线，使用电脑端的串口调试助手进入 AT 模式，将蓝牙模式调为从模式，将波特率设置为相匹配的 38400。在完成这些调整之后，再次测试，发现串口终于能够正常接

收到手机端发送的数据。接下来我们又对蓝牙模块进行了发送数据的测试，经过一段时间的调试，我们成功实现了蓝牙模块的收发功能。在此之后，我们立即将其与小车的运动功能相结合，实现了蓝牙遥控小车的功能。

在电脑端也需要对应的调试。最终能够完成手机上和电脑的串口的接收和发送。但是这样还是不足够的，为了进一步完成更加轻松和高效地完成控制和交互，还需要利用更加精细的程序完成，这也是之后本组利用 $python$ 完成串口的编写以及 UI 界面封装的主要原因。

（2）陀螺仪模块：

在本组的实验模拟中，PC 端引导的路径规划的实现需要小车在不依赖其他传感器的情况下独立完成精准的直行和转弯，需要严格控制小车转角和距离的偏移量。而实验室的设备无法提供一个上位机来实时地显示小车当前的姿态和位置。智能盲人导航车的精准性是保障盲人可以安全出行的必要条件，所以本组的小车急需要一种可以精确返回角度的方法，能够减小直行的角度偏差，同时完成精度高的转向。在实验进行的初始阶段，本组已经利用$python$ 程序拟合了左右轮转速和旋转圈数的曲线，证明了无法利用测速计返回的左右轮差速完成小车的调节，故本组准备采用陀螺仪测量小车的姿态。在小车上加装了 $JY-61$ 陀螺仪后，可以得到小车实时返回的当前角度和角速度，考虑到角速度为一个动态量，很难测准，会存在一定的误差，故只采用其传回来的角度作为我们角度调节负反馈的输入量。

$JY-61$陀螺仪采用$UART$串口进行数据的传输，发送给PC端的为三个包，分别为加速度包、角速度包和角度包。由于只是需要角度包，故这里仅介绍角度包的读取过程。角度包的包头为$0x55$，标识为$0x53$，即当PC机收到$0x55$时，表示开始接受一条新的数据，紧接着读到$0x53$，标志着这个包为角度包。由于UART串口传输数据每次为8位数据，而六姿态陀螺仪$JY-61$的三轴角度每个角度包都是16位的数据，故没个方向轴的角度数据都分割为高8位，低8位，分别发送。由于我们小车运动为一个二维平面，所以我们仅读取其Z轴的角度即可，但其他位的数据我们也需要存储起来。数据的具体传输顺序和编号如下图所示：

数据编号	数据内容	含义
0	0x55	包头
1	0x53	标识这个包是角度包
2	RollL	X 轴角度低字节
3	RollH	X 轴角度高字节
4	PitchL	Y 轴角度低字节
5	PitchH	Y 轴角度高字节
6	YawL	Z 轴角度低字节
7	YawH	Z 轴角度高字节
8	TL	温度低字节
9	TH	温度高字节
10	Sum	校验和

角速度计算公式：
滚转角（x 轴）Roll=((RollH<<8)|RollL)/32768*180(°)
俯仰角（y 轴）Pitch=((PitchH<<8)|PitchL)/32768*180(°)
偏航角（z 轴）Yaw=((YawH<<8)|YawL)/32768*180(°)
温度计算公式：
T=((TH<<8)|TL) /340+36.53 ℃
校验和：
Sum=0x55+0x53+RollH+RollL+PitchH+PitchL+YawH+YawL+TH+TL

为了确保数据的可靠性，我们先将数据存储在一个缓存区：$buffer$ 数组中，在将所有的数据读取完全后，我们将数据和和校验位相对比，若一致则将缓存区中的角度数据覆盖 Z 轴角度数组 $AngleZ$，否则，则认为数据传输失真，不予采用。在 $JY-61$ 模块中，我们还编写了返回角度的函数，可以将高 8 位和低 8 位的数据通过计算直接返回车身当前角度，以方便后续程序的运行。下面为 $JY-61$ 部分的关键代码：

$JY-61$陀螺仪采用$UART$串口进行数据的传输，发送给PC端的为三个包，分别为加速度包、角速度包和角度包。由于只是需要角度包，故这里仅介绍角度包的读取过程。角度包的包头为$0x55$，标识为$0x53$，即当PC机收到$0x55$时，表示开始接受一条新的数据，紧接着读到$0x53$，标志着这个包为角度包。由于UART串口传输数据每次为8位数据，而六姿态陀螺仪$JY-61$的三轴角度每个角度包都是16位的数据，故没个方向轴的角度数据都分割为高8位，低8位，分别发送。由于我们小车运动

为一个二维平面，所以我们仅读取其Z轴的角度即可，但其他位的数据我们也需要存储起来。在$JY-61$模块中，我们还编写了返回角度的函数，可以将高8位和低8位的数据通过计算直接返回车身当前角度，以方便后续程序的运行。下面为$JY-61$部分的关键代码：

```
//获取角度，AngleZ[2]
bool JY61_GetAngle(uint8_t* AngleZ)
{
    uint8_t Buffer[8];  //缓存
    uint8_t Sum = 0;    //总和校验
    uint8_t Receive;    //接受数据
    uint8_t i;

    RxFIFO2_Init();
    while (1)
    {
        Receive = UART2_InChar();
        if (Receive == DATAHEAD)
        {
            Sum = Receive;  //初始化校验位
            Receive = UART2_InChar();
            if (Receive == ISANGLESPEED)
            {
                Sum += Receive;
                for (i = 0; i < 8; i++)
                {
                    Buffer[i] = UART2_InChar();
                    Sum += Buffer[i];
                }
                Receive = UART2_InChar();
                if (Receive == Sum)
                {
//                  for(i = 0; i < 8; i++)
//                  {
//                      angular_speed[i] = Buffer[i];
//                  }
                }
            }
            else if (Receive == ISANGLE)
            {
                Sum += Receive;
                for (i = 0; i < 8; i++)
                {
                    Buffer[i] = UART2_InChar();
                    Sum += Buffer[i];
                }
                Receive = UART2_InChar();
                if (Receive == Sum)
                {
                    //printf("%d %d\n", Buffer[5], Buffer[4]);
                    AngleZ[1] = Buffer[4];  //低位
                    AngleZ[0] = Buffer[5];  //高位
                    return 1;
                }
            }
        }
    }

}
```

```
float JY61_ReturnAngle()
{
    uint8_t AngleZHL[2];
    float AngleZ;

    if (JY61_GetAngle(AngleZHL))
    {
        AngleZ = (AngleZHL[0] * 256 + AngleZHL[1]) * 1800 / 32768;
        AngleZ /= 10;
        return AngleZ;
    }

    return 0;
}
```

6，红外传感装置

为了完成小车的障碍规避，本组需要能够识别障碍的装置。经过一段时间的思考，红外测距仪成了我们最终的选择。红外测距仪的原理是发射一束红外光线，当收到红外光线时，可以通过时间差完成距离的测量。依据距离和标准距离的大小关系返回一个二值逻辑（在该距离内或在该距离外）。

下面就以我们红外测距模块的初始化端口函数作为例子。这里需要再提一下的是，一开始我们编写代码时，往往习惯于在初始化时，直接将对应值赋给寄存器，但是当多个寄存器在同一个串口组的时候，这样的幅值就会影响到其他端口的正常工作。经过查阅资料和研究所给的示例，我们调整代码，利用逻辑运算与和或来进行只对对应位的更改，而不影响到其他位。但这就需要我们在初始化前好好思考一下其中的逻辑关系。

```
void Infrared_Init(void){
    P2 -> SEL0 &= ~0x20;
    P2 -> SEL1 &= ~0x20;   //configure P2.5 as GPIO
    P2 -> DIR  &= ~0x20;   //make P2.5 in
}

// if an obstacle is detected, then return a low-level voltage
int Infrared_Get(void){
    return P2 -> IN & 0x20;
}
```

7，*PC* 端控制程序

上文提到为了方便数据的处理和简化控制流程，这里选择利用 *python* 中的 *serial* 库完成串口的编写。当我们获得稳定的交互串口时，交互变得更加轻松有效。在此调试时，主要的问题在于读取串口内容对应的接口函数的选择，因为函数有很多的种类，所以。这里串口的编写代码（这里展示部分代码，主要是整体的串口类和发送和接收的接口）如下图所示：

```
class BTSerial:
    ReceiveData = []
    Reception = None
    TransData = None
    serial: serial.Serial

    def __init__(self, port, bps, time_out):
        self.Port = port            # port
        self.Bps = bps              # baud rate
        self.TimeOut = time_out    # time out
        self.IsConnected = False
        self.serial = serial.Serial(port, bps, timeout=time_out)
```

```
        if self.serial.is_open:
            print('Initialize the port successfully')
        # try:
        #     self.serial = serial.Serial(port, bps, timeout=time_out)
        #     if self.serial.is_open:
        #         print('Initialize the port successfully')
        # except Exception as e:
        #     print('Failed to initialize the port')

    def BasicInfo(self):
        print('Serial name: ', self.serial.name)
        print('Serial port: ', self.serial.port)
        print('Serial baudrate: ', self.serial.baudrate)
        print('Serial bytesize: ', self.serial.bytesize)
        print('Serial stopbits: ', self.serial.stopbits)
        print('serial parity: ', self.serial.parity)

    def OpenSerial(self):
        self.serial.open()

    def CloseSerial(self):
        self.serial.close()
        if not self.serial.is_open:
            print('The serial is closed')
        else:
            print('Fail to close')

    def Start(self):
        self.OpenSerial()
        if self.serial.is_open:
            print('Open successfully')
            return True
        else:
            print('Fail to open')
            return False

    def Receive(self, ReceiveFinished):
        while self.serial.is_open:
            try:
                if self.serial.in_waiting:
                    self.Reception = self.serial.read_all().hex()

                    print(self.Reception)
                    ReceiveFinished.emit(self.Reception)    # 发送信号
                    self.ReceiveData.append(self.Reception)
                    sleep(0.1)
            except Exception as e:
                pass

    def Transmit(self):
        # print(self.TransData)
        self.serial.write(self.TransData.encode('utf-8'))
        # sleep(0.2)
```

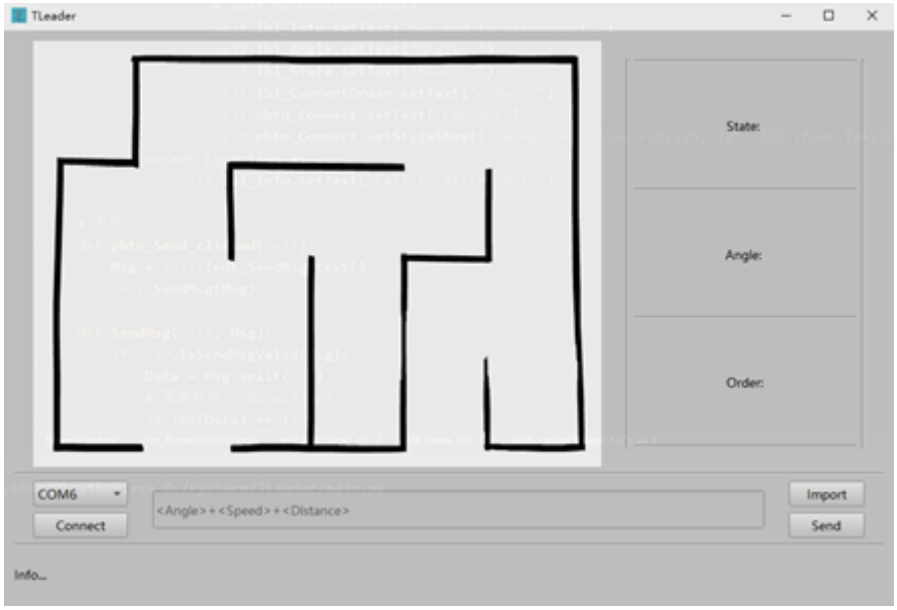
虽然上述串口具备串口调试助手的所有功能，但是它也就仅仅具备串口调试助手的功能：控制指令的发送还需要在控制台上完成，接收到的数据也不能直接利用，还是在控制台上打印。所以到目前为止，串口的功能还是不足够的。

所以为了完善整体的控制过程，方便调试，更好地利用多个线程完成完成更立体的操作。本组还通过 $PyQt$ 完成了 PC 界面的设计，通过多线程的方式同时完成指令的发送和接收。发送指令的方式设置成两种：即直接在命令行中输入对应指令的格式，或者导入指令的 txt 文档，这样小车会按照 txt 中指令的顺序进行执行。由于界面的代码相对冗杂，这里不再粘贴，详情请参见具体的代码附件。下面展示设计出的界面的具体样式。我们同时还为界面设计了 $LOGO$：TL，意味着 $T\ Leader$ 之意。

指令的格式为 $< angle > + < speed\ level > + < distance >$，指令含义为：小车先按照当前的正对方向作为基准，旋转对应的角度，然后以一定的速度（这里我们设定了三个 $speed\ level$），前进一定的距离。这里的 $distance$ 表示小车的车轮需要实际旋转的距离。其主体显示部分的代码如下所示：

```
    def retranslateUi(self, TI_RSLK):
        _translate = QtCore.QCoreApplication.translate
        TI_RSLK.setWindowTitle(_translate("TI_RSLK", "TI-RSLK小车助手"))
        self.lbl_Map.setText(_translate("TI_RSLK", "Map"))
        self.lbl_State.setText(_translate("TI_RSLK", "State: "))
        self.lbl_Angle.setText(_translate("TI_RSLK", "Angle: "))
        self.lbl_CurrentOrder.setText(_translate("TI_RSLK", "Order:"))
        self.cbx_ComSelect.setWhatsThis(_translate("TI_RSLK", "<html><head/><body><p>端口
号</p></body></html>"))
        self.cbx_ComSelect.setItemText(0, _translate("TI_RSLK", "COM6"))
        self.cbx_ComSelect.setItemText(1, _translate("TI_RSLK", "COM7"))
        self.cbx_ComSelect.setItemText(2, _translate("TI_RSLK", "COM3"))
        self.cbx_ComSelect.setItemText(3, _translate("TI_RSLK", "COM4"))
        self.cbx_ComSelect.setItemText(4, _translate("TI_RSLK", "COM5"))
        self.pbtn_Connect.setWhatsThis(_translate("TI_RSLK", "<html><head/><body><p>连接
</p></body></html>"))
        self.pbtn_Connect.setText(_translate("TI_RSLK", "Connect"))
        self.ledt_SendMsg.setPlaceholderText(_translate("TI_RSLK", "<Angle>+<Speed>+
<Distance>"))
        self.pbtn_Import.setText(_translate("TI_RSLK", "Import"))
        self.pbtn_Send.setWhatsThis(_translate("TI_RSLK", "<html><head/><body><p>发送</p>
</body></html>"))
        self.pbtn_Send.setText(_translate("TI_RSLK", "Send"))
        self.lbl_Info.setText(_translate("TI_RSLK", "Info..."))
```

最终得到的界面效果如下图所示。其中界面的左上方用于显示地图；当 PC 和小车交互时，图像会转换为具体的路径；界面的右上方显示了小车的状态（$state$），角度（$angle$）以及当前具体的一条指令（$order$）。界面的下方显示了连接串口（COM），连接或断开（$connect$），输入的某条指令（命令行），发送（$send$）以及导入（$import$，这里指导入一个由路径算法程序规划得出的 txt 文档，内部是一系列指令集）。最下方的是 $info$，显示当前的状态，如是否成功连接，是否成功断开，指令是否发送，信息是否接收等等内容。



但是仅有指令还是不够的，因为在像素点和实际距离之间还存在一定的换算关系。为了解决这个问题，本组测量了迷宫和像素换算的比例尺，然后通过这种比例撰写了 *pixel_point_convert* 函数，完成了像素点的距离和实际距离之间的转换；然后测量小车左右轮的周长取平均值，再将实际距离换算成轮子需要旋转的圈数，这样便于输出。实验代码如下图所示：

```python
# convert a certain point from one picture to another with different scale
def pixel_point_convert(firsthand_image, target_image, pixel_posi):
    new_x = int(pixel_posi[0] / firsthand_image.shape[1] * target_image.shape[1])
    new_y = int(pixel_posi[1] / firsthand_image.shape[0] * target_image.shape[0])
    return [new_x, new_y]
```

8，硬软件控制逻辑

在将软件和算法部分编写完毕，硬件设备都分别调试完成之后，我们需要再将其进行整合，设计成一个整体的逻辑控制单元和硬软件交互系统。这里软件的控制逻辑由 *PC* 界面发送的指令为基准；而硬件整体逻辑分为蓝牙控制模块，陀螺仪模块和用户交互模块，硬件部分的控制指令逻辑是根据软件发送的指令进行选择执行。在收到软件发送的信息之后，硬件部分会按照角度，速度等级和距离（旋转圈数）来进行读取，然后执行先旋转一定角度，然后以三种速度等级（L, M, H）行驶指定的距离（即完成对应的圈数）。同时如果红外部分读取到前方有障碍，则会通过蓝牙向 *PC* 发送一个指令，告知 *PC* 障碍的出现并提醒电脑使用路径算法重新更换路径，硬件部分的主要代码如下所示：

```c
//初始化
void HC05_Init()
{
    UART1_Init();
    Motor_Init();
}

//蓝牙发送uint8_t
void HC05_SendChar(uint8_t Data)
{
    UART1_OutChar(Data);
}

//蓝牙发送数组
void HC05_SendString(uint8_t* Data)
{
    UART1_OutString(Data);
}

//电脑给出角度、速度等级、运动距离
```

```
void HC05_Motor()
{
    uint8_t AngleH = UART1_InChar();
    //printf("%d", AngleH);
    uint8_t AngleL = UART1_InChar();
    //printf("%d", AngleL);
    uint8_t SpeedLevel = UART1_InChar();
    //printf("%c", SpeedLevel);
    uint8_t Distance = UART1_InChar();   //以0.1圈为单位，范围0~255
    //printf("%d", Distance);
    int Speed;
    int Angle = (AngleH * 256 + AngleL) * 180 / 32768;
    uint8_t AngleZHL[2];
    float DeltaAngle = 0;

    if (Angle > 44 && Angle < 46)
    {
        Angle = 270;
    }

    //选择速度
    switch(SpeedLevel)
    {
    case HIGHSPEEDLEVEL:
        Speed = HIGHSPEED;
        break;
    case MIDSPEEDLEVEL:
        Speed = MIDSPEED;
        break;
    case LOWSPEEDLEVEL:
        Speed = LOWSPEED;
        break;
    default:
        Speed = MIDSPEED;
        break;
    }
    //发送当前绝对角度
    DeltaAngle = JY61_ReturnAngle();
    DeltaAngle =
            (DeltaAngle - INIT_ANGLE) >= 0 ?
                    (DeltaAngle - INIT_ANGLE) : (DeltaAngle - INIT_ANGLE + 360);
    HC05_AngleToBit8(DeltaAngle, AngleZHL);
    HC05_SendChar (AngleZHL[0]);     //高位
    HC05_SendChar(AngleZHL[1]);     //低位

    TARGET_ANGLE = Ang1AndAng2(TARGET_ANGLE, Angle);
    PID_Turn(LOWSPEED, Angle);
    if (Infrared_Get())
    {
        PID_Forward(Speed, Distance * 0.1);
        HC05_SendChar('E');
    }
    else
    {
        HC05_SendChar('C');
    }
}
```

9，用户交互电路

为了模拟小车对于盲人行进的提醒，需要一些外部的指示电路进行处理提示，所以这里我们选择了蜂鸣器电路和 LED 电路，用于对某些特定情景进行提示。

（1）蜂鸣器模块：

由于 $launchpad$ 开发板的高电平不能够提供足够的电流用于带动蜂鸣器，所以这里采取外接模拟电路的方式，使蜂鸣器能够正常工作，并且能够通过程序的设定播放特定的音乐。蜂鸣器的逻辑为：当遇到障碍时，蜂鸣器直接鸣叫，进行提示；当到达重点时，蜂鸣器会播放指定的音乐。

蜂鸣器模拟电路部分电路图如下图所示：可见其是一个模拟电流放大电路，其基本原理如下所示：

（2）LED 电路：

LED 电路依靠车身的电路完成，通过不停高电平切换选择不同的发光二极管，然后完成闪烁。每次在遇到障碍时，LED 灯就会进行闪烁，以起到提示的作用。在按照 TI 官方的教程编写对应的代码之后，三色 LED 模块也工作得非常好，能够实现按照一定的频率进行换色闪烁。

六，遇到的困难和失败的尝试

部分失败的案例在细节部分的介绍已经完成，这里介绍一些本组在实践过程中遇到的比较突出的问题。

1，陀螺仪协议的选择：

上文已经提及，为实现小车角度的准确控制，我们必须使用陀螺仪模块。

最初本组设想的是直接使用现成的 $GY-521$ 模块。在进一步查阅资料之后，我们了解到该模块是使用 $I2C$ 协议进行串口通信的，于是我们在网上查找了 $I2C$ 协议的具体实现，并尝试对陀螺仪的进行具体的调试。后来我们发现：在官方给出的 SDK 中是有 $I2C$ 通信的例程的，该模块使用引脚的专用的 SCL, SDA 线进行通信，后又在网页看到说官方的例程通信总是出问题，建议自己使用 $GPIO$ 软串口模拟实现 $I2C$ 通信，于是我们又尝试从头构造一个 $I2C$ 的库，在写完一部分之后，一天的时间也结束了，在最后我们想起在搜索 $MPU6050$ 陀螺仪时，看到了一个既可以使用 $I2C$ 协议又可以使用 $UART$ 协议的 $JY-61$ 模块，于是我们在淘宝上了解到，该模块的确能够使用更为简便的 $UART$ 协议，并且精度比 $GY-521$ 更高，于是我们果断地转而选择更为方便、精度更高的 $JY-61$ 模块。

2，路径算法的实现

这里的困难主要在于，最初时面对一张图像很难想到如何处理：仅仅针对像素直接获得路径几乎是不现实的。后来考虑到，如果假设小车在这样的图像迷宫中真实行驶，那就可以利用 BFS 算法完成对应的设计。最初我们还怀疑是不是这样处理会很慢，但是后来我们简单分析了复杂度：如果使用小车边框判断的方法，复杂度大约为 $[L(image\_height)+L(image\_width)]\cdot[L(car\_length)+L(car\_height)]\cdot n=L(image)\cdot L(car\_length)\cdot n$，由于图像和车的长度都是定值，所以复杂度几乎是线性的（而且这里开辟了新的三维张量减少时间复杂度），最终算法也确实能够很快实现。这样路径规划的主题就完成了。

3，PID的选择和调整：

根据陀螺仪返回的参数，理论上我们可以按照其只要当角度为我们所需的转弯读数时，就停止电机的转动，但显然这是做不到的。由于程序运行需要一定的时间，每次从陀螺仪中读取数据实则为一个间隔时间不确定的离散采样过程，而要让其正好取到360度中的某一个值的概率为零。故我们采用一个判断语句，考虑到角速度，我们将阈值设置为比预期角度小某一角度的角，当当前角度小于阈值时，继续转弯，若大于阈值时，则停止电机的工作。阈值的确定则由实际场地的情况确定。但是我们发现无论我们如何调节参数，我们只能做到转弯的角度的期望等于预期角度，但是仍然会存在不可忽略的方差。其原因就是因为角速度的存在，其在上一个循环中读取陀螺仪的值还未超过阈值，但是当下一次读取陀螺仪的值时，其增加量为角速度和间隔时间的积分，累加值可能大于预

期角度，也可能小于预期角度，当相差量达到一定范围的时候，小车的姿态已经出现了明显的偏移，对行进精准度有着很大的影响。并且我们采用的是相对角度的调节方式，误差会以累加的方式进行堆叠，这样的精准度显然无法撑起我们小车走完一个复杂的行进路线。

因为相对角度的误差会有累计的特点，所以我们一开始的调整为采用绝对角度来代替相对角度。以陀螺仪上电后的角度为初始角度，以后转弯的所有角度都在此基础上进行计算，理论上可以消除相对角度带来的误差累积的问题，但是实际调试的过程中，小车却没能按照我们的预期运动，在第一个转弯处，小车就出现了不停旋转的问题。我们猜测为若采用绝对角度，小车当前角度即使超过了阈值但是旋转300多度之后仍可以小于阈值，并且小车最终角度的随机落点还是没能解决，只能期望于直行的PID进行调节。

我们的第二种方案受到绝对角度的启发，既然都在直行阶段调节姿态，我们就可以把直行阶段的初始角度设置为转弯之前的角度+转弯的角度，这样理论上可以在每一次直行阶段调节小车姿态。并且小车姿态可以进行递推，只要初始角度的准确性得到保证，则后续的角度就一定是正确。但在赛道上实现时我们发现，直行赛道有时过短并不能修正角度的偏差，如果将直行的PID比例系数调大，整个车会走S型路线，以致到达终点时还不能达到一个收敛，结果甚至会放大角度的偏差，最终我们放弃了这个方案。

最后，我们提出了自创的“**小角度精度调节**”算法，让小车每次转弯动作结束后，对自身姿态进行调节，而每次的调节都为小角度的调整，直到当前角度与目标角度达到0.5度范围之内。虽然这样会每一次转弯都需要调整姿态（抖动车身），但这样的算法，使得我们的小车在每一次转弯后都可以精确地开始下一步的行进，实现了小车的精准控制。

关于直行PID调节，我们经过讨论先从简单的P（比例系数调节的方法）调节，若效果较好，积分微分则不需要再拖慢整体程序的运算速度。幸运的是，比例运算方法在实地实验时效果非常好，甚至在某些场合纠偏了偏差角，故我们的直行PID最终采用了简单的P进行反馈。值得一提的是，由于地面地缝和胶带打滑的影响，在直行刹车阶段，小车往往会因为地面的原因车身旋转一定的角度。在我们开发出“小角度精度调节”算法后，我们将它移植到直行结束阶段，做最后的判断是否当前角度和直行初始角度一致。取得了非常好的效果。

根据陀螺仪返回的参数，理论上我们可以按照其只要当角度为我们所需的转弯读数时，就停止电机的转动，但显然这是做不到的。由于程序运行需要一定的时间，每次从陀螺仪中读取数据实则为一个间隔时间不确定的离散采样过程，而要让其正好取到360度中的某一个值的概率为零。故我们采用一个判断语句，考虑到角速度，我们将阈值设置为比预期角度小某一角度的角，当当前角度小于阈值时，继续转弯，若大于阈值时，则停止电机的工作。阈值的确定则由实际场地的情况确定。但是我们发现无论我们如何调节参数，我们只能做到转弯的角度的期望等于预期角度，但是仍然会存在不可忽略的方差。其原因就是因为角速度的存在，其在上一个循环中读取陀螺仪的值还未超过阈值，但是当下一次读取陀螺仪的值时，其增加量为角速度和间隔时间的积分，累加值可能大于预期角度，也可能小于预期角度，当相差量达到一定范围的时候，小车的姿态已经出现了明显的偏移，对行进精准度有着很大的影响。并且我们采用的是相对角度的调节方式，误差会以累加的方式进行堆叠，这样的精准度显然无法撑起我们小车走完一个复杂的行进路线。

因为相对角度的误差会有累计的特点，所以我们一开始的调整为采用绝对角度来代替相对角度。以陀螺仪上电后的角度为初始角度，以后转弯的所有角度都在此基础上进行计算，理论上可以消除相对角度带来的误差累积的问题，但是实际调试的过程中，小车却没能按照我们的预期运动，在第一个转弯处，小车就出现了不停旋转的问题。我们猜测为若采用绝对角度，小车当前角度即使超过了阈值但是旋转300多度之后仍可以小于阈值，并且小车最终角度的随机落点还是没能解决，只能期望于直行的PID进行调节。

我们的第二种方案受到绝对角度的启发，既然都在直行阶段调节姿态，我们就可以把直行阶段的初始角度设置为转弯之前的角度+转弯的角度，这样理论上可以在每一次直行阶段调节小车姿态。并且小车姿态可以进行递推，只要初始角度的准确性得到保证，则后续的角度就一定是正确。但在赛道上实现时我们发现，直行赛道有时过短并不能修正角度的偏差，如果将直行的PID比例系数调大，整个车会走S型路线，以致到达终点时还不能达到一个收敛，结果甚至会放大角度的偏差，最终我们放弃了这个方案。

最后，我们提出了自创的“小角度精度调节”算法，让小车每次转弯动作结束后，对自身姿态进行调节，而每次的调节都为小角度的调整，直到当前角度与目标角度达到 0.5° 范围之内。虽然这样会每一次转弯都需要调整姿态（抖动车身），但这样的算法，使得我们的小车在每一次转弯后都可以精确地开始下一步的行进，实现了小车的精准控制。

关于直行PID调节，我们经过讨论先从简单的P（比例系数调节的方法）调节，若效果较好，积分微分则不需要再拖慢整体程序的运算速度。幸运的是，比例运算方法在实地实验时效果非常好，甚至在某些场合纠偏了偏差角，故我们的直行PID最终采用了简单的P进行反馈。值得一提的是，由于地面地缝和胶带打滑的影响，在直行刹车阶段，小车往往会因为地面的原因车身旋转一定的角度。在我们开发出“小角度精度调节”算法后，我们将它移植到直行结束阶段，做最后的判断是否当前角度和直行初始角度一致。取得了非常好的效果。

下面是我们的PID模块直行部分的代码：

```
void PID_Forward(int Speed, float Distance)
{
    int leftDuty, rightDuty;
    leftDuty  = Speed;
    rightDuty = Speed - 100;
    int InitStep = Tachometer_GetStep();    //初始步数
    float InitAngle = JY61_ReturnAngle();   //初始角度
    float AngleT;
    float CurAngle = 0;
    Distance -= CIRCLEREPAIR;

    while(Tachometer_GetStep() - InitStep < Distance * 360)
    {
        CurAngle = Ang1SubAng2(JY61_ReturnAngle(), InitAngle);
        if(CurAngle > 1.0 && CurAngle < 90.0)        //右偏
        {
            rightDuty +=  FB * CurAngle;
        }
        else if(CurAngle < 359.0 && CurAngle > 270.0)  //左偏
        {
            rightDuty  -= FB * (360 - CurAngle);
        }
        Motor_Forward(leftDuty, rightDuty);
    }
    Motor_Stop();
    AngleT = Ang1SubAng2(JY61_ReturnAngle(), InitAngle);
    while ((AngleT > 0.5 && AngleT < 30) || (AngleT < 359.5 && AngleT > 330))
    {
        AngleT = Ang1SubAng2(JY61_ReturnAngle(), InitAngle);
        if (AngleT > 0.5 && AngleT < 30)
        {
            Motor_Left(5000, 5000);
            Clock_Delay1ms(40);
            Motor_Stop();
        }
        if (AngleT < 359.5 && AngleT > 330)
        {
            Motor_Right(5000, 5000);
            Clock_Delay1ms(40);
            Motor_Stop();
        }
    }
}
```

以下为转弯模块代码：

```
void PID_Turn(int Speed, float Angle)
{
    float InitAngle = JY61_ReturnAngle();
    int leftDuty, rightDuty;
```

```
    float AngleS;

    leftDuty  = Speed;
    rightDuty = Speed - 100;

    //右转，顺时针转
    if (Angle > 0 && Angle <= 180)
    {
        Angle -= TURNREPAIR;
        Motor_Right(leftDuty, rightDuty);
        while(Ang1SubAng2(JY61_ReturnAngle(), InitAngle) < Angle)
        {
            Motor_Right(leftDuty, rightDuty);
        }
        Motor_Stop();
        Angle += TURNREPAIR;
        AngleS = Ang1SubAng2(JY61_ReturnAngle(), InitAngle) - Angle;
        while(AngleS > 0.5 || AngleS < -0.5){
            AngleS = Ang1SubAng2(JY61_ReturnAngle(), InitAngle) - Angle;
            if(AngleS > 0.5)
            {
                Motor_Left(5000, 5000);
                Clock_Delay1ms(40);
                Motor_Stop();
            }
            if(AngleS < -0.5)
            {
                Motor_Right(5000, 5000);
                Clock_Delay1ms(40);
                Motor_Stop();
            }
        }
    }
    else if (Angle > 180 && Angle < 360)
    {
        Angle += TURNREPAIR;
        Motor_Left(leftDuty, rightDuty);
        while(Ang1SubAng2(JY61_ReturnAngle(), InitAngle) > Angle)
        {
            Motor_Left(leftDuty, rightDuty);
        }
        Angle -= TURNREPAIR;
        AngleS = Ang1SubAng2(JY61_ReturnAngle(), InitAngle) - Angle;
        while(AngleS > 0.5 || AngleS < -0.5){
            AngleS = Ang1SubAng2(JY61_ReturnAngle(), InitAngle) - Angle;
            if(AngleS > 0.5)
            {
                Motor_Left(5000, 5000);
                Clock_Delay1ms(40);
                Motor_Stop();
            }
            if(AngleS < -0.5)
            {
                Motor_Right(5000, 5000);
                Clock_Delay1ms(40);
                Motor_Stop();
            }
        }
```

```
    }
}

```

七，创新点

以下描述本次电子设计的主要创新点，部分调试细节不在此赘述。

1，路径规划：

为了模拟现实世界中导盲车的卫星定位，这里本组采用 PC 端作为指示器，小车作为行动端的模拟。卫星定位通过路径规划和小车移动完成。最初本组设想完成路径规划是完全依靠像素点来区分路径，但是这样操作会过于复杂，而且似乎也没有很好的算法。后来经过查阅资料之后，发现一种将像素迷宫分解成树然后进行运算的方法。由此想到，可以通过模拟小车在迷宫中进行碰撞，这样可以确定可行的点和墙，然后通过队列和广度算法，当找到终点时，退出循环。这是通过回溯得到的路径就是从起点到终点的最短路径。这种模拟算法还可以通过一系列细节的处理简化计算量，几乎能够在瞬间完成计算，并且等到很好的结果。

2，遇到障碍重新规划：

如果车头的红外部分识别到了障碍，则小车将会向 PC 端返回一个信号，表示遇到了障碍，“此路不通”。这是电脑会以小车所在的位置为起点进行重新规划，生成新的路径控制指令，然后将另一条路径转发给小车，小车将会掉头，沿着另一条可以同行的路径完成通行。这里在实际测试中也完成了，而且效果也很好。

3，PC 控制过程：

直接通过有限 USB 线和小车进行交互然后修改代码的方式进行测试是十分麻烦的，所以通过蓝牙交互控制的方法是必然不可缺少的。在本组完成小车的蓝牙控制之后，我们发现仅仅这样是不足够的，这样自主性不够高，只能完成最基本的控制，而且也不能实现自主的调控。所以最终本组制作了高效的界面和控制算法，可以直接通过指令完成小车的旋转、直行等操作，十分便于调试，也方便完成小车整体的控制。

4，小车控制算法：

如果直接利用 $motor$ 库完成小车的直行操作，发现小车直接会向右偏移，即左轮移速较快。在经过多种调试方案后（开环，闭环），我们发现可以利用小角度进行旋转的细节调节是最好的，虽然需要经过一定时间的调整，但是最终的精度很好，可以完成在光滑地面或有轻微高度差（但是会有很大的影响）的地面上完成精准的转弯。在这样完成后，我们想到，在小车的直行过程中，虽然整体上比较直，但是偶尔也会在最终完成直行的位置打滑，导致相对角度出现偏差，这是十分致命的。所以我们在每次直行结束时也添加了小角度的修正，这样能够保证直行和转弯的最终角度都是相对精准的。事实证明，这样操作得到的最终实现效果远好于之前的开环操作。

八，感想与总结

经过本次电子设计小学期，本组经历了小车装置设计的完整过程：从设计之前的思考，探讨，调研；至设计过程中的不太轻松的入门，认真的思考和纠正错误，整合整体的软硬件逻辑；到最终完成任务，拍摄视频，总结.....这是一个非常难忘的过程，我们在此期间完成了很多自己的设想，修改完善自己的逻辑，经历困难的 $debug$ 过程。也曾因为调试不出来而感到困惑和烦恼，但是最终我们还是同心同力坚持了下来，完成了自己的目标，没有浪费两周的时光。同时也万分感谢为我们提供很多帮助的老师和助教，帮助我们更加顺利地完成任务，达到最终的目标。我们会牢记本次小学期实践的经验和教训，未来更好地解决电子装置设计和调试的问题。