机器人导论期末作业报告

学号	姓名
19335109	李雪堃
19335110	李宜谦

(一) 实验任务

- 实验要求：本次期末作业参考 MotionPlanning.ppt 中 52页的视频设计。需要完成感知、规划、控制算法，在未知环境中控制小车从起点运动到终点。起点为右上角，终点为左下角。
- 实验场景：作业给出了三个场景 world1、world2、world3，其中 1 和 2 为静态场景，3 为动态场景，有两扇可开关的门。

(二) 实验环境

环境配置请查看 README。

- 操作系统：Ubuntu 20.04.3 LTS x86_64
- 仿真软件：Webots R2021a
- 编译器：gcc 9.3.0
- 构建系统：GNU Make 4.2.1
- 第三方库：OpenCV

(三) 实验内容与步骤

(1) 实验思路

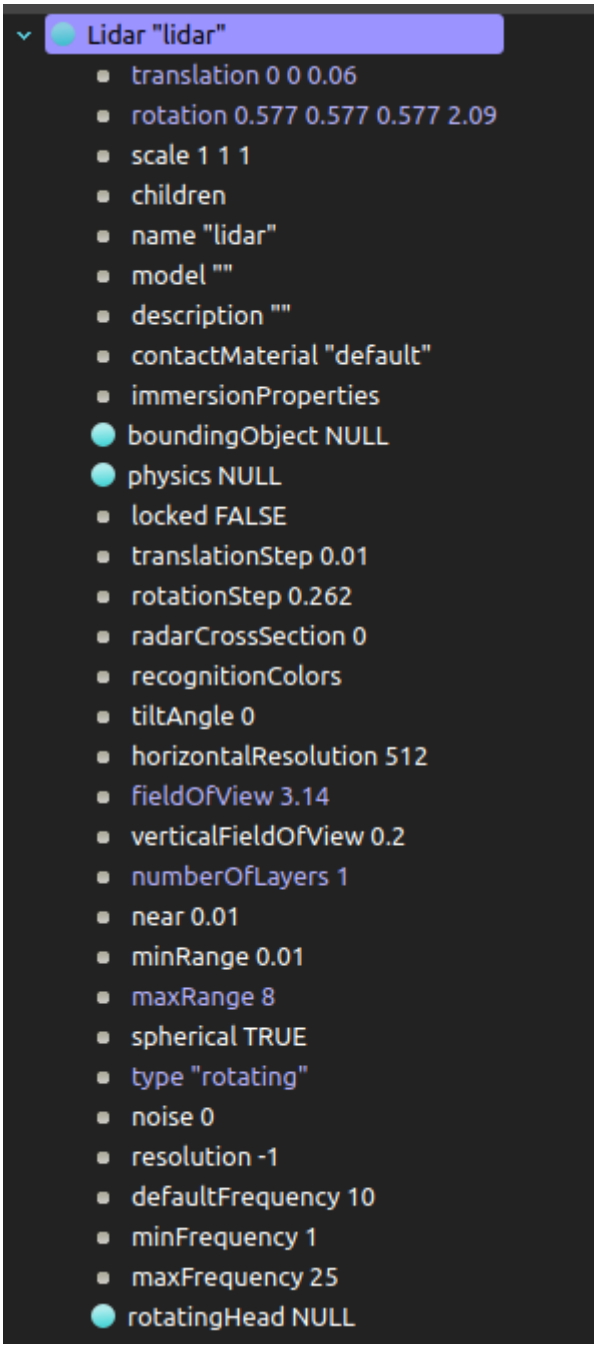
实验实现分为感知、规划、控制三部分。其中感知部分使用激光雷达、GPS 和 InertialUnit，规划部分使用 PRM 和 A* 算法，控制部分依据小车的位置和朝向角度控制小车的速度。

(2) 使用雷达感知障碍物

lidar 参数如下，关键参数是 `fieldOfView`、`maxRange` 和 `type`：

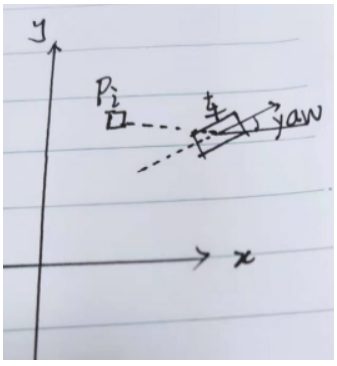
- `fieldOfView`：是雷达视野的弧度，设置为 3.14 则可以获取周围一圈的信息
- `maxRange`：是雷达扫描的最大范围
- `numberOfLayers`：是雷达的层数，我们建的是 2D 图，所以设置为 1，不需要深度信息

- `type`：是雷达的类型，设置为 `rotating` 则雷达每次旋转一周



下面是由助教提供的感知部分的关键代码，我们采用的也是这个感知代码。

- 处理运动畸变：每 60 帧一个循环，每当第 4 帧时 (`cnt` 等于 4) ，使用 `getRangeImage` 获得雷达图像，因此在前 3 帧 (`cnt` < 3)，让小车减速
- 判断离群点：若某点与其他点的距离差的平均在阈值外，则认为它是离群点，进行舍弃。
- 计算点云的坐标：首先获取某个点与小车之间的角度 $\alpha = yaw + \pi - \frac{i}{numPts} \times 2\pi$，然后再将小车的坐标加上 `lidarImage[i]` (即小车到点的距离) 乘以该角度的余弦/正弦，得到该点在地图中的坐标



```
pos = gps->getValues();
double carX = pos[0];
double carY = pos[1];
int carPixelX = int((carX + worldWidth / 2.0) * world2pixel);
int carPixelY = mapHeight - int((carY + worldHeight / 2.0) * world2pixel);

rpy = imu->getRollPitchYaw();
double carAngle = rpy[0]; // rpy[0] = roll, 但旋转小车时 yaw 不发生变化。
// cout << "rpy = " << carAngle << endl;

// step1 建图
if (cnt < 4) // 减速过程
  keyValue = 0;

if (cnt == 4)
{
  lidarImage = lidar->getRangeImage();
  double mapPointAngle, mapPointX, mapPointY;
  for (size_t i = outlierCnt; i < lidarRes - outlierCnt; ++i)
  {
    if (isfinite(lidarImage[i]) && lidarImage[i] != 0)
    {
      double outlierCheck = 0.0;
      for (int k = i - outlierCnt; k < i + outlierCnt; ++k)
      {
        outlierCheck += abs(lidarImage[i] - lidarImage[k]);
      }
      outlierCheck /= (2 * outlierCnt);
      if (outlierCheck > 0.1 * lidarImage[i])
        continue;

      mapPointAngle = carAngle + M_PI - double(i) / double(lidarRes) * 2.0 * M_PI;
      mapPointX = lidarImage[i] * cos(mapPointAngle);
      mapPointY = lidarImage[i] * sin(mapPointAngle);
      mapPointX += carX;
      mapPointY += carY;

      int imgX = int((mapPointX + worldWidth / 2.0) * world2pixel);
      int imgY = mapHeight - int((mapPointY + worldHeight / 2.0) * world2pixel);
      if (imgX >= 0 && imgX < map.cols && imgY >= 0 && imgY < map.rows)
      {
        circle(map, Point(imgX, imgY), 18, 0, -1);
      }
    }
  }

  // imshow("map", map);
  // waitKey(1);
}
```

(3) 使用 PRM + A* 规划路线

两点之间距离采用曼哈顿距离。两点之间碰撞检测的思想是，对两点之间的连线进行采样，判断采样点是否是障碍点，如果是则认为两点之间连线存在障碍。

```cpp
double distance(point p1, point p2)
{
  return abs(p1.x - p2.x) + abs(p1.y - p2.y);
  // return sqrt(pow((double)p1.x - (double)p2.x, 2.0) + pow((double)p1.y - (double)p2.y, 2.0));
}

vector<double> linspace(int a, int b, int n)
{
  vector<double> res;
  double d = (double)(max(a, b) - min(a, b)) / (double)n;
  for (int i = 0; i < n; i++)
    res.push_back((double)min(a, b) + d * i);
  return res;
}

bool is_collision(Mat map, point p1, point p2)
{
  int step_len = max(abs(p1.x - p2.x), abs(p1.y - p2.y));
  vector<double> path_x = linspace(min(p1.x, p2.x), max(p1.x, p2.x), step_len + 1);
  vector<double> path_y = linspace(min(p1.y, p2.y), max(p1.y, p2.y), step_len + 1);
  for (int i = 1; i < step_len + 1; i++)
  {
    if (map.data[(int)round(path_x[i]) * mapWidth + (int)round(path_y[i])] == 0)
    {
      return true;
    }
  }
  return false;
}
```

下面是 A* 算法和构建路径的 C++ 代码。

```cpp
vector<point> rebuild_path(vector<point> vertex2coord, map<int, int> cameFrom, int current)
{
  vector<point> path;
  path.push_back(vertex2coord[current]);

  vector<int> keys;
  for (auto it = cameFrom.begin(); it != cameFrom.end(); it++)
  {
    keys.push_back(it->first);
  }

  while (find(keys.begin(), keys.end(), current) != keys.end())
  {
    current = cameFrom[current];
    path.push_back(vertex2coord[current]);
  }
  return path;
}

vector<point> astar(int graph[][sampleNum + 2], vector<point> vertex2coord, point src, point dst)
{
  vector<int> openSet;
  vector<int> closeSet;
  map<int, int> cameFrom;
  openSet.push_back(0); // 加入起点到开集

  map<int, int> gScore;
  gScore[0] = 0; // 起点的 g 值为 0
  for (int i = 1; i < sampleNum + 2; i++)
    gScore[i] = INF;

  map<int, int> fScore;
  fScore[0] = distance(vertex2coord[0], vertex2coord[1]); // 起点的 f 值为起点和终点的距离
  for (int i = 1; i < sampleNum + 2; i++)
    fScore[i] = INF;

  while (openSet.size() > 0)
  {
    map<int, int> fScoreOpen;
    for (int &i : openSet)
    {
      fScoreOpen.insert(pair<int, int>(i, fScore[i]));
    }
    pair<int, int> min_pair = *min_element(fScoreOpen.begin(), fScoreOpen.end(), compare_fscore);
    int current = min_pair.first;

    if (current == 1) // 如果到达终点
      return rebuild_path(vertex2coord, cameFrom, current);

    remove(openSet.begin(), openSet.end(), current);
    closeSet.push_back(current);

    vector<int> neighbors;
    for (int i = 0; i < sampleNum + 2; i++) // 获取当前节点的所有邻居节点的索引
    {
      if (graph[current][i] > 0)
      {
        neighbors.push_back(i);
      }
    }

    for (int &neighbor : neighbors) // 遍历所有邻居节点
    {
      if (find(closeSet.begin(), closeSet.end(), neighbor) != closeSet.end())
        continue;

      int tentative_gscore = gScore[current] + distance(vertex2coord[current], vertex2coord[neighbor]);
      if (find(openSet.begin(), openSet.end(), neighbor) == openSet.end())
        openSet.push_back(neighbor);
      else if (tentative_gscore >= gScore[neighbor])
        continue;

      cameFrom[neighbor] = current;
      gScore[neighbor] = tentative_gscore;
      fScore[neighbor] = gScore[neighbor] + distance(vertex2coord[neighbor], vertex2coord[1]);
    }
  }
}
```

下面是 PRM 算法的 C++ 代码。

`vertex2coord` 是顶点编号到其在 opencv 矩阵中坐标的映射。

图的存储采用一个全局的邻接矩阵 `graph`， `graph[i][j]` 为 0 表示 i 号点和 j 号点之间没有边。

```
vector<point> prm(Mat map, point src, point dst)
{
  vector<point> vertex2coord;
  vertex2coord.push_back(src); // 索引 0 对应起点（即小车位置）
  vertex2coord.push_back(dst); // 索引 1 对应终点

  // memset(graph, 0, sizeof(int) * (sampleNum + 2) * (sampleNum + 2));

  while (vertex2coord.size() < sampleNum + 2)
  {
    int x = rand() % mapHeight; // 随机选取采样点
    int y = rand() % mapWidth;
    point samplePoint(x, y);
    if (map.data[x * mapWidth + y] != 0)
    {
      vertex2coord.push_back(samplePoint);
    }
  }

  for (int i = 0; i < sampleNum + 2; i++)
  {
    for (int j = i; j < sampleNum + 2; j++)
    {
      if (i != j && distance(vertex2coord[i], vertex2coord[j]) <= 100 && is_collision(map, vertex2coord[i], vertex2coord[j]) == false)
      {
        graph[i][j] = graph[j][i] = distance(vertex2coord[i], vertex2coord[j]);
      }
      else
      {
        graph[i][j] = graph[j][i] = 0;
      }
    }
  }

  vector<point> path = astar(graph, vertex2coord, src, dst);
  reverse(path.begin(), path.end());

  return path;
}
```

在模拟的每一帧中，判断 `path` 是否为空或者是否需要重新规划。

重新规划的条件是，如果上一次规划的路径上有新检测到的障碍，或者当前小车位置到路径上的下一个点有障碍，那么就进行重新规划。

```
bool needPlan(Mat map, vector<point> path, point car_pos)
{
  for (int i = 0; i < path.size() - 1; i++)
  {
    if (is_collision(map, path[i], path[i+1]))
      return true;
  }

  return is_collision(map, car_pos, path[1]);
}
```

```
point car_pos(carPixelY, carPixelX);
if (path.empty() || needPlan(map, path, car_pos))
{
  do
  {
    path = prm(map, car_pos, dst_pos);
  }
  while (path.empty() || (path[path.size() - 1].x != dst_pos.x && path[path.size() - 1].y != dst_pos.y));
}
```

(4) 控制小车按规划的路线运动

主要逻辑：判断小车的朝向角度Yaw与路线设置的前进的方向对应的角度之差是否在一个较小阈值范围内。若在该范围内，则小车直行，否则小车将进行左转或者右转来调整朝向角度。这里前进的方向通过路径上的点 `path[0]` 到 `path[1]` 的角度来计算。通过函数 `convertDiffToDirection` 计算前进方向。

```
double convertDiffToDirection(point start, point end)
{
  return -atan2(end.x - start.x, end.y - start.y);
}
```

前进方向的计算需要维护路径 `path`。维护 `path[1]` 为下一个目标点，设置当小车与 `path[1]` 的距离小于一定值时，认为到达目标点，将 `path[0]` 从路径中删除。

```
// step4
// 控制
if (cnt > 4)
{
  if (distance(car_pos, path[1]) < 5)
  {
    path.erase(path.begin());
  }
  if (path.size() > 1)
  {
    double direction = convertDiffToDirection(path[0], path[1]);
    keyValue = control(carAngle, direction);
  }
}
```

当小车朝向角度 `currentYaw` 与路径方向 `pathDirection` 之差不在阈值内时，小车进行转向。当路径方向在小车中轴线的左侧时，小车左转，否则路径方向在小车中轴线的右侧，小车右转。

具体判断逻辑如下：当 `currentYaw < 0` 时，小车运动的反方向为 `currentYaw + M_PI` ，若 `pathDirection` 在 `(currentYaw, currentYaw + M_PI)` 之间，说明路径方向在小车中轴线的左侧，小车左转；否则小车右转。

当 `currentYaw >= 0` 时，小车运动的反方向为 `currentYaw - M_PI`，若 `pathDirection在(currentYaw, M_PI)` 之间或者 `(-M_PI, currentYaw - M_PI)` 之间时，说明路径方向在小车中轴线的左侧，小车左转；否则小车右转。

```
char control(double currentYaw, double pathDirection)
{
  char keyValue = 'Z';
  if (abs(currentYaw - pathDirection) < thershold)
  {
    //printf("%lf\n",currentYaw-pathDirection);
    keyValue = 'W'; //forward
  }
  else
  { //find path
    // double oppositeYaw = currentYaw > 0 ? currentYaw - M_PI : currentYaw + M_PI;
    // printf("yaw:%lf dire:%lf opposite yaw:%lf\n", currentYaw, pathDirection,oppositeYaw);
    if (currentYaw < 0)
    {
      if (pathDirection > currentYaw && pathDirection < currentYaw + M_PI)
      {
        keyValue = 'Q'; //turn left
      }
      else
      {
        keyValue = 'E'; //turn right
      }
    }
    else
    {
      if (pathDirection > currentYaw || pathDirection < currentYaw - M_PI)
      {
        keyValue = 'Q'; //turn left
      }
      else
      {
        keyValue = 'E'; //turn right
      }
    }
  }
  return keyValue;
}
```

当小车到达终点 `dst_pos` 时，将速度设置为0。

```
if (distance(car_pos, dst_pos) < 5)
{
  keyValue = 'Z';
}
```

(三) 实验结果与分析

我们的控制器代码在大部分测试中可以让小车跑到终点，但是有时候小车顶到墙上时，控制器会 crash。我们认为是代码中存在数组越界的问题，但是 debug 了很久也没有很好地解决。另一种可能是，每 60 帧的第 4 帧才会获取雷达点云，有可能后面控制小车运动的过程中，小车撞到了墙，这之间存在一定的延迟。还有可能是 A* 算法死循环了，有的时候雷达扫描的图像会在莫名其妙的地方出现黑色障碍，而这些地方原本不是障碍物，导致切断了到终点之间的可行路径，A* 算法死循环。

world1 和 world2 可以顺利跑到终点，world3 有时候不能跑到终点（当小车在门周围时控制器容易 crash）。

demo 请查看 `world1_demo.mp4`、`world2_demo.mp4` 和 `world3_demo.mp4`。