西南石油大学

# 本科毕业设计（论文）


## 三维空间最短路径算法设计与实现

院 (系) 名 称：　计算机科学学院

专 业 名 称：　软件工程

学 生 姓 名：　陈东

学　　号：　201731062232

指 导 教 师：　刘小玲　讲师

二〇二一年五月

**Southwest Petroleum University**

**Graduation Thesis**

# Design and Implementation of Shortest Path Algorithm in Three-Dimensional Space


Grade: 2017

Name: Chen dong

Specialty: Software Engineering

Instructor: Prof. Liu Xiao Ling




SouthWest Petroleum University


May, 2021

# 郑 重 声 明

本人呈交的学位论文, 是在导师的指导下, 独立进行研究工作所取得的成果, 所有数据、图片资料真实可靠. 尽我所知, 除文中已经注明引用的内容外, 本学位论文的研究成果不包含他人享有著作权的内容. 对本论文所涉及的研究工作做出贡献的其他个人和集体, 均已在文中以明确的方式标明. 本学位论文的知识产权归属于培养单位.

本人签名: _____ 日期: _____

# 摘　　要

在现实生活中的许多场景中，关于给定的某一起点出发到一终点的最短路径问题是极其普遍的，可以将问题抽象为二维空间或三维空间的最短路径问题，其中二维空间的最短路径问题已经拥有了成熟的适用于不同情况的最短路径算法，如单源最短路径算法中：Dijiktra 算法拥有优越的时间和空间复杂度但适用于带正权图；Bellman-Ford 算法不仅适用于正权图，也适用于负权图；基于 Bellman-Ford 算法改进的 SPFA 算法不仅适用于正负权值图，也拥有优越的平均时间复杂度；多源最短路径：Floyd 算法可以计算出顶点到顶点之间的最短路径；

三维空间最短路径算法，基于离散化近似计算原理，将障碍物离散到指定精度下的格点网络中，再利用 A*、SPFA 算法计算出离散格点下的给定起点到终点的最短路径。由于计算的格点路径还可以进一步优化路径，因此再不断拟合格点路径为连续线段，最终算法将计算出起点到终点经过的线段路径和最短路径长度。基于给定精度计算最短路径长度，因此可以根据实际情况调整计算精度，在结果精度和计算时间和空间消耗之间取的最好的平衡。

通过分析可行算法和粗略解决思路，提出了将连续空间离散化成为格点的思路，再利用最短路径算法计算最短路径，最后拟合成为连续线段。在算法设计完成后进行了样例测试并针对测试结果进行进一步完善。最后，本算法成功解决了三维空间最短路径求解问题，至此，本算法的设计与实现工作顺利结束。

关键词: 毕业论文; 最短路径算法; 三维空间;

# ABSTRACT

In many real-life scenarios, the shortest path problem from a given origin to an end point is extremely common and can be abstracted as a two-dimensional or three-dimensional shortest path problem, where the two-dimensional shortest path problem has mature shortest path algorithms for different situations, such as the single-source shortest path algorithm: Dijiktra algorithm has The Bellman-Ford algorithm is applicable not only to positive-weighted graphs but also to negative-weighted graphs; the improved SPFA algorithm based on the Bellman-Ford algorithm is not only applicable to positive- and negative-weighted graphs but also has superior average time complexity; the multi-source shortest path: Floyd's algorithm can calculate the shortest path between The Floyd algorithm can calculate the shortest path from vertex to vertex;

The shortest path algorithm in 3D space is based on the principle of discretization approximation, which discretizes the obstacles into a network of lattice points with specified precision, and then uses A* and SPFA algorithms to calculate the shortest path from a given starting point to the end point under the discrete lattice points. Since the calculated grid point path can be further optimized, the grid point path is continuously fitted as a continuous line segment, and the final algorithm will calculate the line segment path and the shortest path length from the starting point to the end point. The shortest path length is calculated based on the given accuracy, so the calculation accuracy can be adjusted according to the actual situation to get the best balance between the result accuracy and the

calculation time and space consumption.

By analyzing the feasible algorithms and approximation solutions, we propose the idea of discretizing the continuous space into lattice points, and then use the shortest path algorithm to calculate the shortest path and finally fit it into continuous line segments. After the algorithm was designed, a sample test was conducted and further improved based on the test results. Finally, the algorithm successfully solves the shortest path problem in three-dimensional space, and thus the design and implementation of the algorithm are successfully completed.

**Key words:** Graduation Thesis; Shortest path algorithm;Three-dimensional space;

# 目　　录

# 1 绪论

## 1.1 论文研究背景与意义

### 1.1.1 选题的背景

对于许多实际生活中需要求得两个地方之间的路径和距离，并且考虑障碍物的影响，以提供后续决策等的帮助，例如房间当中两个窗户之间风的流向轨迹，在山体之间修路等问题的求解时需要计算出起点到终点之间的最短路径。对于小数据范围如房屋空间，不需要复杂的数学模型建模转换为二维模型以求解最短路径，可以在考虑一定误差范围内将障碍物转换为 AABB 包围盒，利用最短路径算法直接在三维空间中求解起点到终点的近似最短路径。

### 1.1.2 选题的技术现状

本算法通过将空间离散为若干个格子点，将障碍物近似为每一边都平行于一个坐标平面的简单六面体，通过常规最短路径算法：BFS 广度优先搜索算法和 A* 寻路算法求解给定起点到终点不经过障碍物的的可行路径，再拟合成若干线段，通过检查线段是否穿过障碍物实现路径拟合，最后通过动态规划计算最优路径长度。

### 1.1.3 选题的意义

完成本选题，是为了设计并实现算法以解决实际项目中遇到的路径长度求解问题，提供近似最短路径和欧式距离以提供路径选择参考。

## 1.2 国内外研究现状

国内外对于三维最短路径算法的研究一般基于数字高程模型（Digital Elevation Model，简称 DEM），是通过用一组有序数值阵列形式表示地面高程的数据集。对于 DEM 给出的地面高程信息，在测绘、水文、气象等应用地面模型的领域都取得了突破性进展。本文不利用 DEM 模型，而是将三维空间离散为若干个格点，再应用最短路径算法在格点间，求得起点格点到终点格点的最短路径后再将格点之间的路线拟合为线段的实现方式。

## 1.3 论文组织结构

本文针对三维空间路径规划问题开展研究，首先将障碍物的 aabb 包围盒以某个尺度离散为具体的格点，对格点进行不可访问标记处理；随后基于所给的起点，采用最短路径算法求得到终点所经过的格点序列，提出一种基 SPFA 和 A* 的最短路径算法实现，同时对比了基于不同最短路径算法之间的空间-时间开销。本文的组织结构如下：

第一章：本章主要介绍研究课题的背景和意义，三维空间最短路径研究现状及发展的趋势，并讨论了国内外在相关方面的研究状况及其应用前景，给出本文的主要贡献、创新性和组织结构。

第二章：本章主要介绍了最短路径模型、二维和三维空间最短路径的区别和联系、图的概念、存储方式并对比了不同存储方式的优缺点。最后，总结和阐述了三维空间最短路径的规划方法。

第三章：本章主要对三维空间最短路径求解算法进行分类总结，提出将三维空间离散为格点后，在格点图上进行 BFS、A*、SPFA 等算法求解最短路径的效率和开销。

第四章：对全文进行了总结，对未来基于三维空间的最短路径算法深入研究作出了展望。

# 2 最短路径优化算法研究综述

## 2.1 图

### 2.1.1 图的基本概念

图论，是组合数学分支，和其他数学分支也有密切关系，如群论、拓扑学。图是图论的主要研究对象，图是给点若干顶点以及两顶点所构成的图形，这种图形可以用来描述事物之间的某种特定关系。顶点可以用来描述某种事物，连接两顶点可以代表事物之间存在的某种关系，如顶点代表学习任务，连接顶点代表学习该任务之前需要学习的其他任务。图论起源于著名的柯尼斯堡七桥问题即著名的欧拉图一笔画问题。该问题于1736年被欧拉解决，因此普遍认为欧拉是图论的创始人[1]。通常在图论中，以有序对$G=(V,E)$，其中V是点集；$E\subset\{\{x,y\}:(x,y)\in V^2,x\neq y\}$是边集，由所有顶点序列构成，其中一条边$\{x,y\}$中的x，y被称作边的端点；图的分类分为有向图和无向图，其中有向图是指给图的每条边都规定一个方向，其边称为有向边，以$<x,y>$；相反，边没有方向的图称为无向图，无向边以$(x,y)$表示，则在边集会同时存在$(x,y),(y,x)$。并且V，E的元素个数通常都是有限的，图的阶数是其顶点个数$|V|$，图的边数是$|E|$。每条边都连接两个不同的顶点且没有两条不同的边连接一对相同顶点的图称为**简单图**，可能有**多重边**连接同一对顶点的图成为**多重图**。其中最短路径问题、最小生成树问题、关键路径问题都属于是基于图上的问题，并提出了相应的解决方案。如图2.1分别是无向图和有向图的一个实例。其中，对于实际问题中每条边代表了从顶点$x$到$y$所需要的权值的图叫做带权图，其中边上的权值也叫边的长度。

**三维空间**是类似我们生存的空间的数学模型，由长、宽、高三个维度，也即是三维欧几里得空间，定义欧几里得平面为装备了内积的二维实数的向量空间。满足

. 在这个向量空间中的向量对应欧几里得平面中的点，

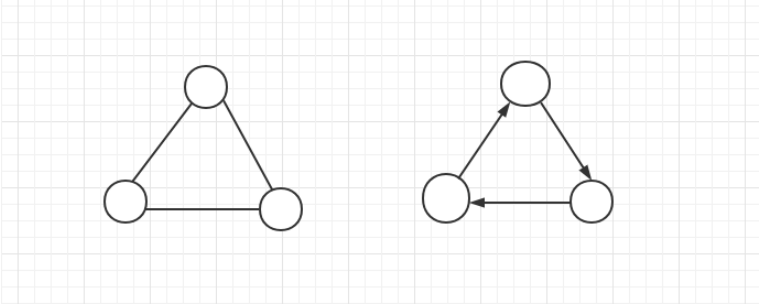


图 2.1　无向图和有向图实例

. 在向量空间中的加法运算对应于平移，

. 内积蕴含了角和距离的概念，它可被用来定义旋转。

如图2.2就是三维欧几里得空间的示意图，三个轴相互垂直，点的坐标为其到三个轴的投影。

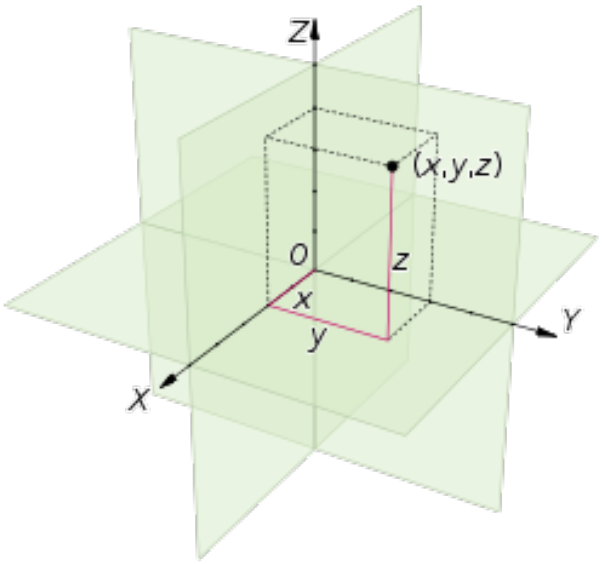



图 2.2　三维欧几里得空间

### 2.1.2　图的存储表示

一般在计算机中存储图信息，有以下几种方式

1. 邻接矩阵，使用多维数组记录点与点之间的边权信息；
2. 邻接表，使用链表等数据结构维护该点到其他点的边信息；

3. 十字链表，将有向图的邻接表和逆邻接表的结合；
4. 邻接多重表，将无向图的邻接表和逆邻接表的结合；

邻接矩阵是应用多维数组保存的点和点之间的关系，因此可以很容易的判断两个点之间的关系，但缺点在于占用空间大，尤其对于多维空间时保存信息时需要占用大量的存储空间；邻接表使用链表保存点所连接的边信息，节省了空间，但不易得到两点之间的关系；十字链表和邻接多重表都能节省空间、快速判断点与点之间的关系，但缺点是实现起来比较复杂。

## 2.2 路径规划

首先,在图中的**路径**是指从顶点 $u$ 到顶点 $v$ 的一个序列 $v_0, e_1, v_1, e_2, v_2, \ldots, e_k, v_k$，有时简写为 $v_0 \to v_1 \to v_2 \to \cdots \to v_k$，其中 $e_i$ 表示起点终点为 $v_{i-1}$ 及 $v_i$；$k$ 称为路径的长度，即经过的边的数量；$v_0 = u$ 称为路径的起点；$v_k = v$ 称为路径的终点。当没有必要区分多重边时，就用顶点序列 $x_0, x_1, \ldots, x_n$ 表示通路 $e_1, e_2, \ldots, e_n$，其中对于 $i = 1, 2, \ldots, n, f(e_i) = x_{i-1}, x_i$，这种记法仅仅指出通路所经过的顶点。其中论文求解的最短路径是指：若从顶点 $u$ 到顶点 $v$ 之间长度最短的路径，即是加权图中一条路径的长度是这条路径上各条边的总和最小。

## 2.3 最短路径规划算法

最短路径问题是图论研究的一个经典算法问题，目的在于求出两顶点之间的最短路径。算法具体的形式有

- 确定起点的最短路径问题，也即已知起始起点求最短路径的单源最短路径问题；
- 确定终点的最短路径问题，与确定起点的问题相反，该问题是已知终结顶点，求最短路径的问题；
- 确定起点终点的最短路问题，即已知起点和终点，求两顶点之间的最短路径；
- 全局最短路径问题，也叫多源最短路问题，求出图中所有的最短路径。

用于解决最短路径问题的算法叫做“最短路径算法”，常用的路径算法有:

- BFS 算法；

- Dijkstra 算法；
- A* 算法；
- Bellman-Ford 算法；
- SPFA 算法（Bellman-Ford 算法的改进版本）；
- Floyd 算法；

广度优先搜索 (BFS) 和深度优先搜索 (DFS) 会从起点开始求解出可行路径，但由于 DFS 需要求解所有路径才知道最短路径，而 BFS 算法的特点是逐层扩展的，决定了其不需要遍历所有的路径就能找到最短路径。Dijkstra、A*、Bellman-Ford、SPFA 算法都是用于解决单源最短路径，其中 Dijskra 用于正权路径最短路径求解，Bellman-Ford、SPFA、A* 可用于正负权路径最短路径求解，Floyd 用于解决多源最短路径；其中不同算法在基于不同的图结构上拥有不同的时间、空间复杂度，其中 Dijkstra 和 A* 在寻路算法中应用广泛，由于 A* 使用了估值函数，会更倾向的选择路线，更加节省内存空间。

## 2.4 本章小结

本章主要介绍了图结构的定义以及常见的计算机存储图结构的数据结构，以及介绍了路径定义和最短路径的定义和求解算法。对于常见的图结构，使用邻接矩阵和邻接表在不同的应用领域都有很好的发挥，如邻接矩阵可以很快的得到顶点与顶点之间的关系，但需要记录每个顶点到其他所有顶点的关系，占用的内存空间较大；邻接表使用链表结构存储每个顶点所连接的边信息，可以节省大量的空间，但不容易得到顶点与顶点的关系。随后介绍了关于对图上最短路径求解的常用算法，如单源最短路径算法，有用于正权图最短路径求解的 Dijskra 算法，正负权都适用的 Bellman-Ford、SPFA 算法，基于估值函数优化的 A* 算法在寻路方面有广泛的应用；以及用于多源最短路径求解的 Floyd 算法。

# 3 最短路径算法实现

## 3.1 最短路径算法分析

在二维空间中，通常是基于二维垂直的欧几里得空间，其中 x 轴和 y 轴垂直，顶点通常以类似经纬度形式给出，如 (x,y) 代表该顶点分别该点坐标分别在 x 轴、y 轴上的长度分量，点 (a,b) 在 x 轴的分量为 a ，在 y 轴的分量为 b ；其中顶点之间会与若干条路径相连，其中路径不能经过相关障碍物，路径的权值可以是该条路径的长度或该条路径所需费用，记为 W(i,j) 表示第 i 个顶点到第 j 个顶点的边的权值。例如在运输网络中，顶点代表的是不同城市的集散地，顶点之间的路径就是城市之间的交通网，路径的权值代表该条交通路线所需要的时间，则所求的顶点到顶点的最短路径即是最短时间的路线。

### 3.1.1 基于 Dijkstra 算法的最短路径求解

Dijkstra 该算法基于贪心的思想，解决了图 G=<V,E> 上带权的单源最短路径问题，通过设置一顶点集合 S ，在集合 S 中所有的顶点与源点 s 之间的最终最短路径权值均已被计算出来。算法反复选择最短路径估计最小的点 $u \in V - S$ 并将 u 加入到 S 中，最终计算出源点到其他所有点的最短距离。举例来说，若图中的顶点代表城市，而边上的权值表示城市间开车行车的距离，该算法可以用来找到两个城市之间的最短路径。但是 Dijkstra 算法并不能有效处理带有负权边的图。

算法描述：Dijkstra 算法通过保留目前为止所找到的每个顶点 $v \in V$ 从 s 到 v 的最短路径来工作。初始时，原点 s 的路径权重被赋为 0。同时把所有顶点的路径长度设为无穷大，即表示我们不知道任何通向这些顶点的路径。当算法结束后，d[v] 中存储的便是从 s 到 v 的最短路径，或者如果路径不存在的话是无穷大。

松弛操作是 Dijkstra 的基础操作：如果存在一条从 u 到 v 的边，那么从 s 到 v 的一条新路径是将边 $w(u,v) \in E$ 添加到从 s 到 u 的路径尾部来拓展一条从 s 到 v

的路径。这条路径的长度是 $d[u]+w(u,v)$。若这个值比目前已知的 $d[v]$ 的值要小，那么可以用这个值来替代当前 $d[v]$ 中的值。松弛边的操作一直运行到所有的 $d[v]$ 都代表从 s 到 v 的最短路径的长度值。由于基于松弛操作，因此若存在负权值边中的负环时会重复松弛，导致无法正确求解出最短路径。

算法维护两个顶点集合 openList 和 clostList，集合 openList 保留所有已知实际最短路径值的顶点，而集合 clostList 则保留其他所有顶点。集合 S 初始状态为空，而后每一步都有一个顶点从 Q 移动到 S。这个被选择的顶点是 Q 中拥有最小的 $d[u]$ 值的顶点。当一个顶点 u 从 Q 中转移到了 S 中，算法对 u 的每条外接边 $w(u,v)$ 进行松弛。

算法的步骤如下

1. 输入边全为正权的图，G 中带有顶点 V=$v_0, v_1, v_2 \ldots$ 和若干边 $w(v_i, v_j)$
2. 设置一个待检测列表 clostList 和一个不需检测列表 clostList;
3. 将起始点 startPoint 加入 openList 中;
4. 初始化距离向量 d ，其中 startPoint 的距离为 0，其他全为正无穷;
5. 检测 openList，找出 d 值最小的一个点，将此节点作为当前节点 (curNode)，并加入 closeList 中不再检测;
6. 若该点等于终点 endPoint，此时终止算法，已找到最短路径，返回该点的距离向量;
7. 否则遍历 curNode 的相邻节点，若该节点是不可通过的、或在 closeList 中的、或超过边界的，则跳过;
8. 保存该节点的 parent 为 curNode，计算其距离向量值 d ，再保存到 openList 中;
9. 若此时 openList 为空，算法结束，未找到起点到终点的最短路径;
10. 若 openList 不为空，回到步骤 5;

算法中的步骤 8，计算相邻节点 (Node) 的距离向量 d 时，即是算法的松弛操作，具体计算方法是：$if(d[curNode]+w(curNode,Node)<d[Node])d[Node]=d[curNode]+w(curNode,Node)$，通过不断的将所有点加入 openList 来松弛到其他顶点的距离向量，最后就能计算出从起点到终点的最短路径了。

例如使用该算法来寻找两个城市之间的最短路径，整个图是城市之间的交通网，此时顶点是城市，顶点之间的路径是城市之间的交通路线，路径的权值是行

驶所需的时间，因此两个城市之间可能有多条权值不同的路径，此时使用 Dijkstra 算法求解得到的最短路径即是从起点出发到其他顶点的所需要的最短时间的路线。

### 3.1.2 基于 A* 算法的最短路径求解

A* 搜索算法综合了最良优先算法 (Best-first search) 和 Dijkstra 算法的优点：在进行启发式搜索提高算法效率的同时，可以保证找到一条最优路径（基于评估函数）。

该算法中，以 g(n) 表示从起点到任意顶点 n 的实际距离， h(n) 表示任意顶点 n 到目标顶点的估算距离，那么 A* 算法的估算函数为 f(n)=g(n)+h(n) ，这个公式遵循以下特性:

- 如果 g(n) 为 0，即只计算任意顶点 n 到目标的评估函数 h(n) ，而不计算起点到顶点 n 的距离，则算法转化为使用贪心策略的最良优先搜索，速度最快，但可能得不出最优解；
- 如果 h(n) 不大于顶点 n 到目标顶点的实际距离，则一定可以求出最优解，而且 h(n) 越小，需要计算的节点越多，算法效率越低，常见的评估函数有-欧几里得距离、曼哈顿距离、切比雪夫距离；
- 如果 h(n) 为 0，即只需求出起点到任意顶点 n 的最短路径 g(n) ，而不计算任何评估函数 h(n) ，则转化为单源最短路径问题，即 Dijkstra 算法，此时需要计算最多的顶点。

其步骤如下

1. 设置一个待检测列表 openList 和一个不需检测列表 clostList；
2. 把起始点 startPoint 加入 openList 中；
3. 检测 openList，找出其中 f 值最小的一个点，将此节点作为当前节点 (curNode)，并加入 closeList 中不再检测；
4. 若该点等于终点 endPoint，此时终止算法，已找到最短路径；
5. 否则遍历 curNode 周围的节点，若该节点是不可通过的、在 closeList 中的、超过边界的，则跳过；
6. 保存该节点的 parent，计算 f 值，再保存到 openList 中；
7. 若此时 openList 为空，算法结束，未找到目的点；

8. openList 不为空，回到步骤 3；

A* 算法通过启发式搜索，即评估函数 h(n)，进一步提高了寻找最短路径的效率，减小了无用点的遍历过程，可以大大的节省时间和空间的消耗。

## 3.2 三维最短路径

对于类似实际空间的三维空间，由于三维空间当中顶点的数量较多，且存在复杂的障碍物关系，使得在三维空间中求解最短路径算法对于空间复杂度和时间复杂度之间存在一定的受限，在三维空间当中求解最短路径更偏向于求解近似最短路径，以满足时间和空间的要求，因此算法主要解决在一定的精度误差内近似求解出给出起点到终点之间的最短路径长度及所经过的顶点路径。假设空间离散时精度为 precision ，即代表实际空间中的 1 单位长度对应离散化空间的 precision 单位长度。则在实际空间中的点坐标 (x,y,z) 可以离散化为格子点坐标 $(\lfloor\frac{x}{precision}\rfloor,\lfloor\frac{y}{precision}\rfloor,\lfloor\frac{z}{precision}\rfloor)$，则我们需要对起点、终点和障碍物都完成离散化的过程，然后在离散化空间中求解起点到终点的最短路径。

### 3.2.1 数据处理

首先，根据实际空间的障碍物，模型成若干个如图的长方体，其中长方体满足底面为一个矩形，但可能长宽不平行于 x、y 轴，高平行于三维空间的 z 轴。数据中会以实数给出底面矩形平行的两边的中点 $M_0(a,b), M_1(c,d)$，与之垂直的边的长度 D ，以及该立方体的 z 轴范围 $z_0, z_1$。通过给出的数据可以计算得出该长方体障碍物所对应的顶点。根据实际空间模型，需要首先按照指定的精度将实际空间的模型划分到指定精度的空间，将空间划分为若干个离散化的格点，通过计算该精度下障碍物所经过的格子点，这些障碍物的离散格点都是在计算最短路径时不可以通过的。

根据障碍物底面为一个矩形，其满足长与宽垂直，则设两中点 $M_0, M_1$ 所在直线为矩形的长，则如图3.1，设该矩形的四个顶点分别为 $A(x_0,y_0)$、$B(x_1,y_1)$、$C(x_2,y_2)$、$D(x_3,y_3)$ 且满足相邻的关系，根据 $M_0M_1$ 直线的表达式，和 $\vec{M_0A}\times$

$\vec{M_0M_1} = 0$ 可以计算出与之垂直的直线 $M_0A$ 的表达式为：

$$y - b = \frac{d-b}{c-a}(x-a)$$

设 $k = \frac{d-b}{c-a}$，则原式为 $y - b = k(x-a)$。且矩形的宽为 D，则 $d(M_0, A) = \frac{D}{2}$，故有 $\sqrt{(a-x_0)^2 + (b-y_0)^2} = \frac{D}{2}$。将 $y - b = k(x-a)$ 代入方程，可以得到与 $M_0$ 相邻的顶点 A、D 的 x 坐标分别为

$$x = a \pm \frac{\frac{D}{2}}{\sqrt{k^2+1}}$$

将计算出来的坐标代入 $y - b = k(x-a)$ 可得该点对应的 y 坐标。相似地，计算与 $M_1$ 相邻的两顶点时，将与 $M_0M_1$ 的直线方程所经过的点 $M_0$ 修改为 $M_1$，即该直线方程为 $y - c = k(x-d)$，类似的满足与 $M_1$ 相邻的顶点 B、C 满足的 x 坐标的关系为

$$x = c \pm \frac{\frac{D}{2}}{\sqrt{k^2+1}}$$

将计算的结果代入相应的直线方程当中即可。

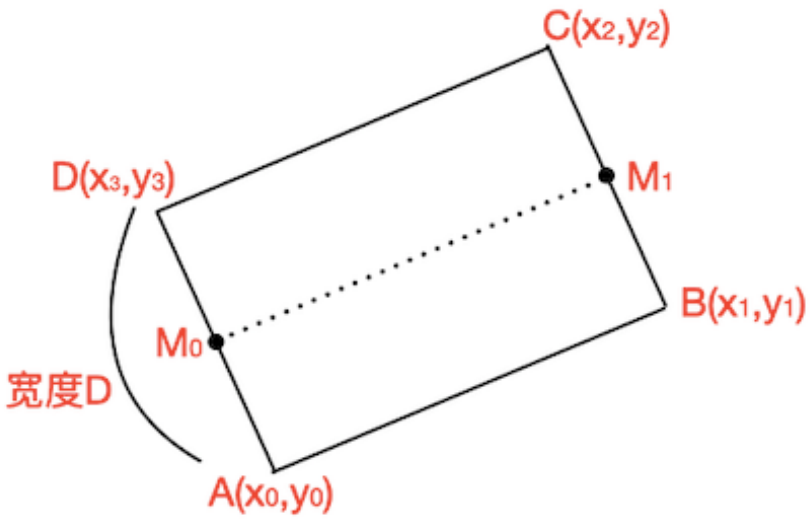



图 3.1　障碍物底面

特别注意特殊判断长宽平行与轴的情况，此时可以直接通过 $M_0, M_1$ 的坐标计算四个顶点的坐标，即四个顶点的坐标分别为 $(x_0, y_0 - \frac{D}{2}), (x_0, y_0 + \frac{D}{2}), (x_1, y_1 - \frac{D}{2}), (x_1, y_1 + \frac{D}{2})$。具体的代码如图3.1，通过 if-else 判断情况来具体计算，最后通

过重载结构体的读入来实现使用 c++ 的 cin 就可以读入障碍物数据时计算顶点并存储在结构体当中，方便后续使用。其中结构体当中的 conveyByPrec 函数是将原始数据下的障碍物离散到指定 precision 精度下的处理后的障碍物。

```
struct node {
    point2D vec[4];
    double z0, z1;
    /*friend ostream & operator <<(ostream & os,const node & p){
        return os << "[" << p.lb << ", " << p.ru << "]";
    }*/
    process_data_node conveyByPrec(double precision) {
        process_data_node node;
        node.vec[0] = vec[0] / precision;
        node.vec[1] = vec[1] / precision;
        node.vec[2] = vec[2] / precision;
        node.vec[3] = vec[3] / precision;
        node.z0 = z0 / precision;
        node.z1 = z1 / precision;
        return node;
    }
    friend istream & operator >>(istream &in, node& p){
        double x0, y0, x1, y1, d;
        in >> x0 >> y0 >> x1 >> y1 >> d >> p.z0 >> p.z1;
        //特判平行的状况
        if (y0 == y1 || x0 == x1) {
            p.vec[0] = {x0, y0 - d / 2};
            p.vec[1] = {x1, y1 - d / 2};
            p.vec[2] = {x1, y1 + d / 2};
            p.vec[3] = {x0, y0 + d / 2};
        }
        else {
            double k = (x1 - x0) / (y1 - y0);
```

```
29              double delta = (d / 2) / sqrt(1 + k * k);
30
31              p.vec[0] = {x0 - delta, y0 + k * delta};
32              p.vec[1] = {x1 - delta, y1 + k * delta};
33              p.vec[2] = {x1 + delta, y1 - k * delta};
34              p.vec[3] = {x0 + delta, y0 - k * delta};
35          }
36          return in;
37      }
38  };
```

**代码 3.1** 顶点处理

通过处理数据，使得可以将最短路径算法应用在三维空间当中，通过控制离散化的精度来实现控制最短路径的精度，离散后格点的标记来实现不可访问情况的判断，达到将实际空间离散化到指定精度的理想空间，从而计算出格点组成的最短路径。

### 3.2.2 相交判断

通过计算出格点组成的最短路径后，由于存在三角形定则，即三角形两边之和大于第三边，所以在计算的格点路径后，是可以通过运用三角形定则拟合计算出更短的路径，此时就需要判断所拟合的新路径是否经过了障碍物。此时问题可以抽象为线段与长方体是否相交的判定问题，根据线段的向量表示法，有线段 $\vec{PQ}$:$P+k(Q-P), k \in [0,1]$，若存在与某个障碍物相交的情况，即存在 k 属于 0～1 之间，使得该点在障碍物之内。考虑障碍物：底面 $A(x_0,y_0), B(x_1,y_1), C(x_2,y_2), D(x_3,y_3)$，高的范围 $[z_0,z_1]$。由于障碍物的高与 z 轴平行，故只需要考虑该障碍物由无数个矩形在 z 轴上堆叠而成。首先，我们讨论如何判断一个二维点坐标是否位于底面矩形当中，设该点坐标为 $E(x,y)$，若该点在矩形内，则满足 $\vec{AB}\times\vec{AE}*\vec{CD}\times\vec{CE} \geq 0$，若该式子成立则说明该点在边 AB 和边 CD 之间；同理，还需要成立 $\vec{DA}\times\vec{DE}*\vec{BC}\times\vec{BE} \geq 0$，即该点在边 DA 和 BC 之间。两个条件需要同时满足，则说明该点在四条边之内，即在该矩形范围内。

将问题放大到三维空间当中，该线段若与该障碍物有交点，则至少存在一个可行的 k 解满足 E 的坐标在障碍物内，则我们求出要与该障碍物相交时 k 值所取的范围，若存在有效范围，则表明该线段与该障碍物相交。首先，对于 z 轴相交，设 $\vec{R} = PQ = Q - P$ 为向量 $PQ$ 的方向向量，则 k 的范围应在 $k \in [0,1]$，对于障碍物的 z 轴所计算的 k 的取值为：$k_0 = \frac{z_0 - P.z}{R.z}, k_1 = \frac{z_1 - P.z}{R.z}$，若 $R.z < 0$ 小于 0 时交换 $k_0, k_1$ 的值。则 $k_0 \leq k_1$ 存在 z 轴可行解，否则无解。代码如下3.2，首先初始化 k 取值范围为 [0,1]，通过分别计算 $z_0, z_1$ 对应的 k 取值来更新。

```
1  bool check_insect(pointInt P, pointInt Q, process_data_node box) {
2      double t1 = 0.0, t2 = 1.0;
3      pointInt R = Q - P;
4      //check the k of z-axis
5      if (R.z == 0) {
6          if (!(box.z0 <= P.z && P.z <= box.z1))
7              return false;
8      }
9      else {
10         double tmp1 = 1.0 * (box.z0 - P.z) / R.z;
11         double tmp2 = 1.0 * (box.z1 - P.z) / R.z;
12         if (R.z < 0) swap(tmp1, tmp2);
13         t1 = max(t1, tmp1);
14         t2 = min(t2, tmp2);
15     }
16     if (t1 > t2)
17         return false;
```

**代码 3.2**　计算 z 轴 k 取值

对于 x、y 所决定的 k 值的范围，需要计算的是该点在矩形内部的情况，则可以用到之前我们所讨论到的公式

$$\vec{AB} \times \vec{AE} * \vec{CD} \times \vec{CE} \geq 0 \tag{3.1}$$

和

$$\vec{DA} \times \vec{DE} * \vec{BC} \times \vec{BE} \geq 0 \tag{3.2}$$

公式3.1和3.2两者需要同时成立。针对可能出现该点在 xy 平面内是一个点的情况，即一条平行与 z 轴的线段，此时利用公式3.1和3.2对该点做一个计算，若两公式都满足则说明该线段在 xy 平面的点在障碍物底面中之中，该部分代码如下3.3。

```
1 //special judge one point situation
2 if (R.x == 0 && R.y == 0) {
3     point2DInt E(P.x, P.y);
4     int tmp1 = ((box[1] - box[0]) * (E - box[0])) * ((box[3] - box[2]) * (E - box[2]));
5     int tmp2 = ((box[0] - box[3]) * (E - box[3])) * ((box[2] - box[1]) * (E - box[1]));
6     if (tmp1 >= 0 && tmp2 >= 0)
7         return true;
8     return false;
9 }
```

**代码 3.3** 相交时单点的情况

对于线段 $\vec{PQ}$ 能到达的点坐标 E，有 $E(P.x + k(Q.x - P.x), P.y + k(Q.y - P.y))$。将该点 E 坐标代入公式3.1中有

$$\begin{aligned}\vec{AB} \times \vec{AE} &= (x_1 - x_0, y_1 - y_0) \times (P.x + k(Q.x - P.x) - x_0, P.y + k(Q.y - P.y) - y_0) \\ &= [(x_1 - x_0)(Q.y - P.y) - (Q.x - P.x)(y_1 - y_0)]k \\ &\quad + (x_1 - x_0)(P.y - y_0) - (P.x - x_0)(y_1 - y_0)\end{aligned}$$

我们设 k 的系数 $a_1 = (x_1 - x_0)(Q.y - P.y) - (Q.x - P.x)(y_1 - y_0)$，后的常数为 $b_1 = (x_1 - x_0)(P.y - y_0) - (P.x - x_0)(y_1 - y_0)$，则公式可化简为 $\vec{AB} \times \vec{AE} = a_1 k + b_1$。

$$\begin{aligned}\vec{CD} \times \vec{CE} &= (x_3 - x_2, y_3 - y_2) \times (P.x + k(Q.x - P.x) - x_2, P.y + k(Q.y - P.y) - y_2) \\ &= [(x_3 - x_2)(Q.y - P.y) - (Q.x - P.x)(y_3 - y_2)]k \\ &\quad + (x_3 - x_2)(P.y - y_2) - (P.x - x_2)(y_3 - y_2)\end{aligned}$$

设 k 的系数为 $a_2 = (x_3 - x_2)(Q.y - P.y) - (Q.x - P.x)(y_3 - y_2)$，后的常数为 $b_2 = (x_3 - x_2)(P.y - y_2) - (P.x - x_2)(y_3 - y_2)$，则公式可化简为 $\vec{CD} \times \vec{CE} = a_2 k + b_2$。

则 $\vec{AB} \times \vec{AE} * \vec{CD} \times \vec{CE} \geq 0$ 等价于方程 $(a_1k + b_1) * (a_2k + b_2) \geq 0 = a_1a_2x^2 + (a_1b_2 + a_2b_1)x + b_1b_2 \geq 0$，即是二次函数求解的问题。关于二次函数根求解的问题，我们需要讨论系数是否为 0，以及特判二次函数出现非实数解的情况。若不存在这些情况，则对于二次函数 $y = ax^2 + bx + c$，二次函数根的计算公式:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{3.3}$$

故我们将公式3.3应用在方程 $a_1a_2x^2 + (a_1b_2 + a_2b_1)x + b_1b_2 \geq 0$ 上，可以得到该方程的解为

$$k = \frac{-(a_1b_2 + a_2b_1) \pm \sqrt{(a_1b_2 + a_2b_1)^2 - 4a_1a_2b_1b_2}}{2a_1a_2} \tag{3.4}$$

我们将之前假设的 $a_1, a_2, b_1, b_2$ 代入该公式既可以求出对于公式3.1限制下 k 的取值范围。使用该求解值更新基于 z 范围求出的 k 范围，若新的 $k_0 \leq k_1$ 则存在可行解，否则无解。代码如下3.4，代码首先判断了出现系数为 0 时二次函数退化为一次函数的情况，下文将进行分析，否则计算二次函数对应的 k 值用来更新并判断。

```
1  //judge ABxAE*CDxCE>=0
2  double a1 = 1.0 * (Q.y - P.y) * (box[1].x - box[0].x) - 1.0 * (Q.x - P.x) * (box[1].y - box
       [0].y);
3  double b1 = 1.0 * (P.y - box[0].y) * (box[1].x - box[0].x) - 1.0 * (P.x - box[0].x) * (box[1].
       y - box[0].y);
4  double a2 = 1.0 * (Q.y - P.y) * (box[3].x - box[2].x) - 1.0 * (Q.x - P.x) * (box[3].y - box
       [2].y);
5  double b2 = 1.0 * (P.y - box[2].y) * (box[3].x - box[2].x) - 1.0 * (P.x - box[2].x) * (box[3].
       y - box[2].y);
6  double delta = 1.0 * (a1 * b2 + a2 * b1) * (a1 * b2 + a2 * b1) - 4.0 * a1 * a2 * b1 * b2;
7  if (delta < 0) return false;
8  if (a1 == 0 && a2 == 0) {
9      if (b1 * b2 < 0)
10         return false;
11 }
12 else if (a1 == 0) {
13     double tmp = -1.0 * b2 / a2;
```

```
14      if (a2 > 0) t1 = max(t1, tmp);
15      else t2 = min(t2, tmp);
16  }
17  else if (a2 == 0) {
18      double tmp = -1.0 * b1 / a1;
19      if (a1 > 0) t1 = max(t1, tmp);
20      else t2 = min(t2, tmp);
21  }
22  else {
23      double x1 = (-1.0 * (a1 * b2 + a2 * b1) - sqrt(delta)) / (2.0 * a1 * a2);
24      double x2 = (-1.0 * (a1 * b2 + a2 * b1) + sqrt(delta)) / (2.0 * a1 * a2);
25      if (x1 > x2) swap(x1, x2);
26      t1 = max(t1, x1);
27      t2 = min(t2, x2);
28  }
29  if (t1 > t2)
30      return false;
```

**代码 3.4** 公式3.1求解 k 值

对于公式3.2，相似的有，我们将点 E 坐标代入有

$$\begin{aligned}\vec{DA} \times \vec{DE} &= (x_0 - x_3, y_0 - y_3) \times (P.x + k(Q.x - P.x) - x_3, P.y + k(Q.y - P.y) - y_3) \\ &= [(x_0 - x_3)(Q.y - P.y) - (Q.x - P.x)(y_0 - y_3)]k \\ &\quad + (x_0 - x_3)(P.y - y_3) - (P.x - x_3)(y_0 - y_3)\end{aligned}$$

同样的，设 k 的系数 $a_1 = (x_0 - x_3)(Q.y - P.y) - (Q.x - P.x)(y_0 - y_3)$，后的常数为 $b_1 = (x_0 - x_3)(P.y - y_3) - (P.x - x_3)(y_0 - y_3)$，则公式可化简为 $\vec{DA} \times \vec{DE} = a_1 k + b_1$。

$$\begin{aligned}\vec{BC} \times \vec{BE} &= (x_2 - x_1, y_2 - y_1) \times (P.x + k(Q.x - P.x) - x_1, P.y + k(Q.y - P.y) - y_1) \\ &= [(x_2 - x_1)(Q.y - P.y) - (Q.x - P.x)(y_2 - y_1)]k \\ &\quad + (x_2 - x_1)(P.y - y_1) - (P.x - x_1)(y_2 - y_1)\end{aligned}$$

设 k 的系数为 $a_2 = (x_2 - x_1)(Q.y - P.y) - (Q.x - P.x)(y_2 - y_1)$，后的常数为 $b_2 = (x_2 - x_1)(P.y - y_1) - (P.x - x_1)(y_2 - y_1)$，则公式可化简为 $\vec{CD} \times \vec{CE} = a_2 k + b_2$。故我们将新的 $a_1, a_2, b_1, b_2$ 的值代入公式3.4中计算出对于公式3.2限制下的 k 的取值范围。使用新求解的 k 值更新由公式3.1更新的 k 值，若最后求解的 k 值仍满足 $k_0 \leq k_1$，则说明线段 $PQ$ 存在与该障碍物相交的情况。具体代码如下3.5，代码同样先判断了退化的情况，再计算的二次函数对应 k 取值来更新并判断。若最后更新后的 k 取值仍满足 $k_0 \leq k_1$，则说明存在可行 k 解使得存在该线段 $PQ$ 与障碍物相交，此时代码 true 表示产生碰撞。

```
1  //judge DAxDE*BCxBE>=0
2  a1 = 1.0 * (Q.y - P.y) * (box[0].x - box[3].x) - 1.0 * (Q.x - P.x) * (box[0].y - box[3].y);
3  b1 = 1.0 * (P.y - box[3].y) * (box[0].x - box[3].x) - 1.0 * (P.x - box[3].x) * (box[0].y - box
       [3].y);
4  a2 = 1.0 * (Q.y - P.y) * (box[2].x - box[1].x) - 1.0 * (Q.x - P.x) * (box[2].y - box[1].y);
5  b2 = 1.0 * (P.y - box[1].y) * (box[2].x - box[1].x) - 1.0 * (P.x - box[1].x) * (box[2].y - box
       [1].y);
6  delta = 1.0 * (a1 * b2 + a2 * b1) * (a1 * b2 + a2 * b1) - 4.0 * a1 * a2 * b1 * b2;
7  if (delta < 0 || a1 * a2 == 0) return false;
8  if (a1 == 0 && a2 == 0) {
9      if (b1 * b2 < 0)
10         return false;
11 }
12 else if (a1 == 0) {
13     double tmp = -1.0 * b2 / a2;
14     if (a2 > 0) t1 = max(t1, tmp);
15     else t2 = min(t2, tmp);
16 }
17 else if (a2 == 0) {
18     double tmp = -1.0 * b1 / a1;
19     if (a1 > 0) t1 = max(t1, tmp);
20     else t2 = min(t2, tmp);
21 }
```

```
22  else {
23      double x1 = (-1.0 * (a1 * b2 + a2 * b1) - sqrt(delta)) / (2.0 * a1 * a2);
24      double x2 = (-1.0 * (a1 * b2 + a2 * b1) + sqrt(delta)) / (2.0 * a1 * a2);
25      if (x1 > x2) swap(x1, x2);
26      t1 = max(t1, x1);
27      t2 = min(t2, x2);
28  }
29  if (t1 <= t2) return true;
30  return false;
```

**代码 3.5** 公式3.2求解 k 值

针对可能出现的二次函数系数为 0 的情况，即公式退化为一次函数的正负问题，如 $b_1(a_2k+b_2) \geq 0$，此时特判系数为 0 的各种情况，计算一次函数下 k 的取值即可，代码如图3.4的 8 行至 20 行。对于出现非实数解的情况，即 $\Delta = b^2 - 4ac < 0$ 时，此时表明不存在有效 k 的取值的点使之在障碍物内，故该线段与障碍物不相交。

至此，已经完成了三维空间内线段与障碍物相交的讨论，利用通过离散化格点求解的格子点路径，不断检查格子点之间是否存在不与障碍物相交也能相连且距离更短的情况，来进一步优化路径。

### 3.2.3 最短路径算法

三维空间最短路径算法的实现，在指定精度后，将实际空间离散到指定精度的空间，因此算法需要求解出离散空间中从起点到终点的格点最短路径，再利用上述讨论的相交算法来拟合路径，最后给出该精度下近似的最短路径和所通过的格点线段。

#### 3.2.3.1 基于 BFS 的三维最短路径算法

广度优先搜索 (BFS) 算法，又叫做宽度优先搜索，是一种在图上的搜索算法。算法将从根节点开始，沿着树的宽度遍历树的节点。在图上时，可以视为根节点

为起点的一棵树。BFS 是一种暴力算法，目的是系统的展开并检查图中的所有节点，以找寻结果。算法的实现一般采用 open-closed 表，所有因为展开节点而得到都会被加进一个先进先出的队列中，其邻居节点尚未检验的节点会被存放在一个称为 open 的容器，而被检验过的节点则被存放在被称为 closed 的容器中。其实现方法为：

1. 首先将根节点放在队列中。
2. 从队列中取出现在的第一个节点，并检查。
   - 如果找到目标，则结束搜索并返回结果。
   - 否则将它所有还没有检查过的相邻子节点加入队列中。
3. 若队列为空，表示检查了所有点都没有找到目标。结束搜索并返回未找到目标。
4. 重复步骤 2.

在离散后的三维空间中，将把每一个整数格点视为顶点，即 $V = \{v_i(x,y,z)|x \in N, y \in N, z \in N\}$，则对于顶点 $v_1 = (x,y,z)$ 相邻的顶点有分别在 x 轴、y 轴、z 轴相邻的 6 个顶点：$(x-1,y,z),(x+1,y,z),(x,y-1,z),(x,y+1,z),(x,y,z-1),(x,y,z+1)$。则基于 BFS 的三维空间最短路径算法实现，是基于从起点格点逐渐向外发散并记录所经过的路径，直到找到终点。算法需要遍历大量的点，若起点 $st(x_0,y_0,z_0)$ 与终点 $ed(x_1,y_1,z_1)$ 之间三维坐标差的最大值为 $n = max\{x_1-x_0, y_1-y_0, z_1-z_0\}$，则算法至少需要遍历 $n^3$ 个点，则算法的时间复杂度为 $O(n^3)$，空间复杂度为 $O(n^3)$。搜索算法初始化部分如代码3.6所示，首先定义了距离向量 dist，用来存储起点距离该格点的最短路径的长度；方向向量 dir，用来找到现在顶点相邻的顶点；以及使用数组 q 和指针 front 和 rear 实现队列的功能，并能将顶点保存起来用来回溯路径；pre 数组存储该顶点 nowNode 的前驱顶点 preNode，即是 BFS 时 nowNode 是通过 preNode 搜索到的且经过格子点数量最少的情况时的前驱顶点。

```
1  namespace BFS {
2      //define
3      int dist[maxn][maxn][maxn];
4      int dir[6][3] = {{-1, 0, 0}, {1, 0, 0}, {0, -1, 0}, {0, 1, 0}, {0, 0, -1}, {0, 0, 1}};
5      bool set_dist(point p, point pre) {
6          if (dist[p.x][p.y][p.z] > dist[pre.x][pre.y][pre.z] + 1) {
```

```
            dist[p.x][p.y][p.z] = dist[pre.x][pre.y][pre.z] + 1;
            return true;
        }
        return false;
    }
    point q[maxm]; int front, rear;
    int pre[maxm], min_dist; bool min_dist_flg;
    vector<int> ed_idxs;
    //initial
    void init() {
        memset(dist, 0x3f, sizeof(dist));
        dist[st.x][st.y][st.z] = 0;
        vis[st.x][st.y][st.z] = 1;
        front = 0; rear = -1;
        q[++rear] = st; pre[0] = -1;
        min_dist_flg = false;
        ed_idxs.clear();
    }
```

**代码 3.6** BFS 搜索算法初始化代码

初始化完成后，算法使用数组实现的队列 q，front 指针表示当前节点在队列中的位置，rear 指针表示队列的最后一个节点的位置，当 $front \leq rear$ 时表示队列不为空。则算法通过不断取出队列的当前节点，并检查是否为终点，或者跳过路径长度以及大于了已经找到的最短路径长度的节点，对搜索做剪枝。随后遍历该节点的相邻节点，若相邻节点没有超出边界且可以访问，则更新其距离向量，并将其加入到队列中。

```
    bool bfs() {
        cerr << "正在搜索最短路径。。。" << endl;
        while (front <= rear) {
            point now = q[front++];
```

```
5             if (min_dist_flg && dist[now.x][now.y][now.z] > min_dist) continue;
6             if (now == ed) {
7                 if (!min_dist_flg || dist[now.x][now.y][now.z] <= min_dist) {
8                     min_dist_flg = true;
9                     min_dist = dist[now.x][now.y][now.z];
10                    ed_idxs.push_back(front - 1);
11                }
12                continue;
13            }
14
15            for (int i = 0; i < 6; ++i) {
16                point nx(now.x + dir[i][0], now.y + dir[i][1], now.z + dir[i][2]);
17                if (check_outofbound(nx)) continue;
18                if (check_insideofobstacles(nx)) continue;
19                if (!set_dist(nx, now)) continue;
20                q[++rear] = nx;
21                vis[nx.x][nx.y][nx.z] = 1;
22                pre[rear] = front - 1;
23            }
24        }
25        return ed_idxs.size() != 0;
26    }
```

**代码 3.7**　BFS 搜索算法主要函数

若算法返回 true 则代表至少存在一条从起点格点到终点格点的格点最短路径，此时需要我们对路径做一个拟合，进一步优化路径长度。根据上述障碍物相交判断中所实现的函数 check_insect 做该条路径上顶点到顶点的拟合操作，即若直接在这两个顶点之间连成线段不经过障碍物且距离更短则直接连接这两个顶点为一条线段而不经过中途格点。此时拟合算法所需要的时间与所找到的最短路径长度有关，对于 BFS 搜索算法，平均最短路径上的格子节点数量在 kn 个，即 n 的常数级别的个数。由于路径上的每个顶点都至多与路径上的所有顶点做一

次障碍物相交判断算法，且使用一个长度为路径上的顶点个数的数组作为距离向量，则算法的时间复杂度在 $O(n^2)$，空间复杂度为 $O(n^2)$。具体的代码如代码3.8，check_segement_cross_all_obstacles(P,Q) 是对所有的障碍物做相交判断，若出现与一个障碍物与该线段 $\vec{PQ}$ 相交则返回非-1，算法通过使用定义的 pre 数组逐渐找到整条路径并记录到 tmp 链表中，对每找出的一条路径做一次拟合，dp_dist 记录该条路径上顶点的距离向量，若此条路径长度更短，则记录在 path 链表中。

```
vector<point> path;
double optimal_dist; bool optimal_dist_flg;

vector<double> dp_dist;
vector<int> tmp, tmp_pre;
void getpath() {
    if (!bfs()) {
        cerr << "未找到路径！" << endl;
        return;
    }
    cerr << "共找到" << ed_idxs.size() << "条路径！开始拟合..." << endl;
    optimal_dist_flg = false;
    for (int k = 0; k < ed_idxs.size(); ++k) {
        int ed_idx = ed_idxs[k];
        cerr << "正在拟合第" << k + 1 << "条路径..." << endl;
        tmp.clear();
        for (int idx = ed_idx; idx != -1; idx = pre[idx]) {
            tmp.push_back(idx);
        }
        reverse(tmp.begin(), tmp.end());
        dp_dist.clear();
        dp_dist.resize(tmp.size());
        tmp_pre.clear();
        tmp_pre.resize(tmp.size());
        for (int i = 0; i < tmp.size(); ++i) {
```

```
26              if (!i) dp_dist[i] = 0;
27              else dp_dist[i] = dp_dist[i - 1] + get_dist(q[tmp[i]], q[tmp[i - 1]]);
28              tmp_pre[i] = i - 1;
29
30              for (int j = 0; j < i; ++j) {
31                  if (check_segement_cross_all_obstacles(q[tmp[i]], q[tmp[j]]) == -1) {
32                      if (dp_dist[j] + get_dist(q[tmp[i]], q[tmp[j]]) < dp_dist[i]) {
33                          dp_dist[i] = dp_dist[j] + get_dist(q[tmp[i]], q[tmp[j]]);
34                          tmp_pre[i] = j;
35                      }
36                  }
37              }
38          }
39          cerr << "拟合完毕，该条路径长度为：" << dp_dist[tmp.size() - 1] << endl;
40          if (!optimal_dist_flg || dp_dist[tmp.size() - 1] < optimal_dist) {
41              optimal_dist_flg = true;
42              optimal_dist = dp_dist[tmp.size() - 1];
43              path.clear();
44              for (int i = tmp.size() - 1; i != -1; i = tmp_pre[i]) {
45                  path.push_back(q[tmp[i]]);
46              }
47              reverse(path.begin(), path.end());
48          }
49      }
50      cerr << "找到经过" << path.size() << "个点的路径";
51      for (int i = 0; i < path.size(); ++i) {
52          if (!i) cerr << path[i];
53          else cerr << "->" << path[i];
54      }
55      cerr << endl;
56      cerr << "该条路径的长度为：" << optimal_dist << "，起点到终点的欧式距离为：" << get_dist(
```

```
        st, ed) << endl;
}
```

**代码 3.8** BFS 搜索算法拟合路径

至此已经完成了使用 BFS 算法对三维空间带有障碍物寻找最短路径的设计和实现，如分析中拟合算法的时间复杂度平均为 $O(n^2)$ 会小于寻路算法的时间复杂度 $O(n^3)$，拟合算法的空间复杂度 $O(n^2)$ 会小于寻路算法的空间复杂度 $O(n^3)$，因此整个算法的时间复杂度和空间复杂度都为 $O(n^3)$。当时间空间较大或离散精度较大时，整个算法运行时间和占用空间不太理想。

#### 3.2.3.2 基于 A* 的三维最短路径算法

正如章节3.1.2中对 A* 算法的分析，由于 A* 算法采用了估价函数 h(n) 进行启发式搜索，提高了找到路径的时间和空间的效率，并且可以保证找到基于估价函数的最优路径。对于估价函数 h(n)，我们采用欧几里得距离，即 $nowNode(x_0, y_0, z_0)$, $endNode(x_1, y_1, z_1)$ 之间的欧几里得距离为 $d(nowNode, endNode) = \sqrt{(x_0 - x_1)^2 + (y_0 - y_1)^2 + (z_0 - z_1)^2}$。则 g(n) 表示的是起点到任意顶点 n 的最短路径长度，估算函数即为 f(n)=g(n)+h(n)。代码如代码3.9所示，结构体 heap_node 用来存储函数 $f(n)$ 和当前节点在队列中的索引，并重载了小于和大于运算符配合优先队列 priority_queue 以实现大顶堆；链表 arr 记录的是队列遍历的点，dist 向量存储的是该点的最短路径长度，pre 向量存储的是该点在最短路径中前驱节点的编号，paths 链表存储的是最后计算出的最短路径所经过的格子节点，record 哈希表存储的是三维空间内的格子点对应在队列元素 arr 中的索引，__init__ 函数实现算法的初始化操作。

```
namespace Astar {
    //define
    struct heap_node {
        double h, g;
        int id;
        heap_node() {}
```

```
7         heap_node(double _h, double _g, int _id) : h(_h), g(_g), id(_id) {}
8         bool operator <(const heap_node& rhs) const {
9             return h + g < rhs.h + rhs.g;
10        }
11        bool operator >(const heap_node& rhs) const {
12            return h + g > rhs.h + rhs.g;
13        }
14    };
15    priority_queue<heap_node, vector<heap_node>, greater<heap_node> > Q;
16    vector<pointInt> arr;
17    vector<double> dist;
18    vector<int> pre;
19    vector<pointInt> paths;
20    double min_dist;
21    map<pair<int, pair<int, int> >, int> record;
22    pair<int, pair<int, int> > point2pair(pointInt p) {
23        return make_pair(p.x, make_pair(p.y, p.z));
24    }
25    void __init__() {
26        arr.clear();
27        dist.clear();
28        pre.clear();
29        record.clear();
30        paths.clear();
31        while (Q.size()) Q.pop();
32    }
```

**代码 3.9** A* 算法初始化

A* 算法每次从队列中找出 f(n) 值最小的顶点，将其作为当前顶点，去遍历其周围的顶点，由于顶点到顶点之间的边权是欧几里得距离，故可以应用 Dijkstra 算法的松弛操作，遍历顶点时计算出该点到已经计算过的点（即 arr 链表存储的点）之间

不经过障碍物的距离，即将原来的拟合算法与寻路算法合并在了一起，代码如代码3.10所示，__build__ 是寻路并拟合的函数，__process__ 是处理最短路径并输出的函数，__main__ 是算法入口。

```
bool __build__() {
    cerr << "A*算法搜索可达路径中。。。" << endl;
    arr.push_back(st); Q.push({0, get_dist(st, ed), 0});
    pre.push_back(-1); dist.push_back(0);
    record[point2pair(st)] = 0;
    while (!Q.empty()) {
        heap_node top = Q.top(); Q.pop();
        pointInt now = arr[top.id];
        //cerr << top.h << " " << top.g << ": " << now << endl;
        if (now == ed) {
            min_dist = top.h;
            return true;
        }
        for (int i = 0; i < 6; ++i) {
            pointInt nx(now.x + dir[i][0], now.y + dir[i][1], now.z + dir[i][2]);
            //check_is_out_of_bound
            if (nx.x < 0 || nx.y < 0 || nx.z < 0) continue;
            if (nx.x > range_x + 1 || nx.y > range_y + 1 || nx.z > range_z + 1) continue;
            //check_is_able_to_reach
            if (check_all_insect(now, nx) != -1) continue;
            if (record.count(point2pair(nx)) == 0) {
                arr.push_back(nx);
                int id = arr.size() - 1;
                pre.push_back(top.id);
                dist.push_back(dist[top.id] + get_dist(now, nx));
                record[point2pair(nx)] = id;
                for (int pre_id = 0; pre_id < id; ++pre_id) {
                    if (check_all_insect(arr[pre_id], nx) == -1) if (dist[pre_id] +
```

```
                        get_dist(arr[pre_id], nx) < dist[id]) {
29                      dist[id] = dist[pre_id] + get_dist(arr[pre_id], nx);
30                      pre[id] = pre_id;
31                  }
32              }
33              Q.push({dist[id], get_dist(nx, ed), id});
34          }
35          else {
36              int id = record[point2pair(nx)];
37              bool isup = false;
38              for (int preid = 0; preid < arr.size(); ++preid) if (preid != id &&
                    check_all_insect(arr[preid], arr[id]) == -1) {
39                  if (dist[preid] + get_dist(arr[preid], arr[id]) < dist[id]) {
40                      dist[id] = dist[preid] + get_dist(arr[preid], arr[id]);
41                      pre[id] = preid;
42                      isup = true;
43                  }
44              }
45              if (isup)
46                  Q.push({dist[id], get_dist(nx, ed), id});
47          }
48      }
49  }
50  return false;
51 }
52 bool __process__() {
53  int ed_idx = record[point2pair(ed)];
54  for (int i = ed_idx; i != -1; i = pre[i]) {
55      paths.push_back(arr[i]);
56  }
57  reverse(paths.begin(), paths.end());
```

```
58          cerr << "找到经过" << paths.size() << "个点的路径";
59          for (int i = 0; i < paths.size(); ++i) {
60              cout << paths[i] << endl;
61              if (!i) cerr << paths[i];
62              else cerr << "->" << paths[i];
63          }
64          cerr << endl;
65          cerr << "该条路径的长度为: " << min_dist << endl;
66          cerr << "起点到终点的欧式距离为: " << get_dist(st, ed) << endl;
67          return true;
68      }
69      int __main_() {
70          __init__();
71          bool build_flg = __build__();
72          if (!build_flg) {
73              cerr << "未找到从" << st << "到" << ed << "的有效路径! " << endl;
74              return 0;
75          }
76          __process__();
77          return 1;
78      }
79  }
```

**代码 3.10** A* 算法主要函数

由于 A* 算法属于启发式算法，对于其复杂度的分析不太准确，但已知的是由于估价函数的存在，可以使路径朝终点方向搜索，有效地减少遍历点的数量，从而在时间和空间复杂度上都有不小的优化。

## 3.3 本章总结

本章主要讨论了算法实现当中的重要部分的实现原理，首先分析了常用的最

短路径 Dijkstra 和 A* 算法，对其原理和实现方法进行了分析，以及介绍了对障碍物数据进行处理的算法和线段与障碍物相交算法，最后就是讨论关于基于 BFS 和 A* 最短路径算法的实现。在常见的正权图结构中，Dijkstra 算法拥有非常好的时间和空间复杂度，而引入了估价函数的 A* 算法，在进行启发式搜索提高搜索算法效率的同时，可以保证找到一条基于评估函数的最优路径。对于输入的数据，我们需要先对输入的障碍物底面的两中点和 z 轴范围做一个处理，计算出其的底面四个顶点。在起点、终点和障碍物离散到指定精度的空间后，提出了利用了向量叉积来检查线段是否与障碍物相交的算法。最后设计和实现了基于 BFS 和 A* 的三维空间最短路径算法，并分析了两者的特点，BFS 算法原理较为简单，是基于暴力的系统展开并检查图中的所有顶点；A* 算法采用了估价函数，可以大大提高算法使用的运行时间和减小空间占用，并且能找到一条基于估价函数的最优路径。

# 4 算法测试

为了测试算法的正确性，需要用尽可能全面的数据样例来测试。根据算法构成，设计了数据对数据处理算法模块、路径求解算法模块以及整体算法进行测试。

## 4.1 数据处理算法测试

对数据处理模块而言，算法代码主要是代码3.1，即将输出的数据信息格式化到可以供后续算法使用的格式，主要是关于计算障碍物底部四个顶点和将顶点离散到指定精度下。测试数据如表4.1。测试数据为起点终点经过了单个障碍物的情

**表 4.1 输入测试数据 1**

| | |
|---|---|
| 起点 | 0 0 0 |
| 终点 | 5 5 5 |
| 障碍物 1 | 0.5 0.5 3.7 3.7 1 0.2 4 |
| 精度 | 0.1 |

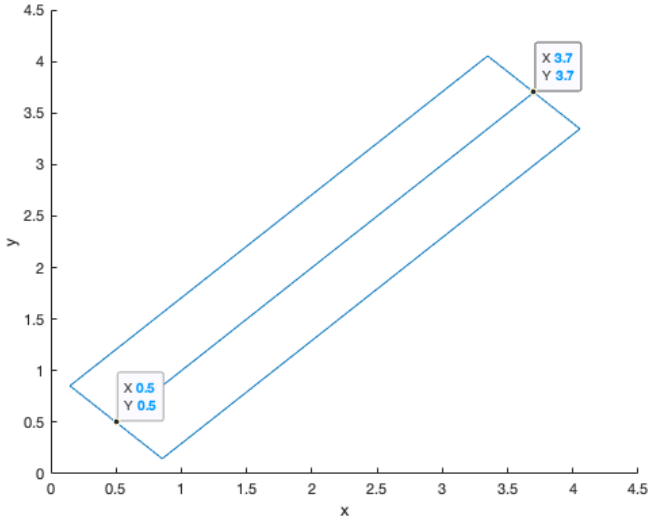
况，此时障碍物信息如表4.2。障碍物底面是由中点为 (0.5, 0.3)，(3.7, 3.7)，边长为 1 的矩形组成，高度 z 轴范围为 [0.2, 4]。根据算法3.1，计算出障碍物的数据为四顶点的坐标分别为 (0.146447, 0.853553), (3.34645, 4.05355), (4.05355, 3.34645), (0.853553, 0.146447), 此时, 使用 Matlab 绘制出关于此障碍物的图像, 图像如图4.1。

验证图示底面是否满足障碍物底面，底面边长为 $d = \sqrt{(0.146447-0.853553)^2+(0.853553-0.146447)^2} = \sqrt{0.999997790436} \approx 1$，在误差范围内满足相等；若两边垂直，满足 $L_1 \cdot L_2 = 0$，$\vec{AB} = (0.853553, 0.146447) - (0.146447, 0.853553) = (0.707106, -0.707106)$，$\vec{M_0M_1} = (3.7, 3.7) - (0.5, 0.5) =$

**表 4.2　障碍物 1 信息**

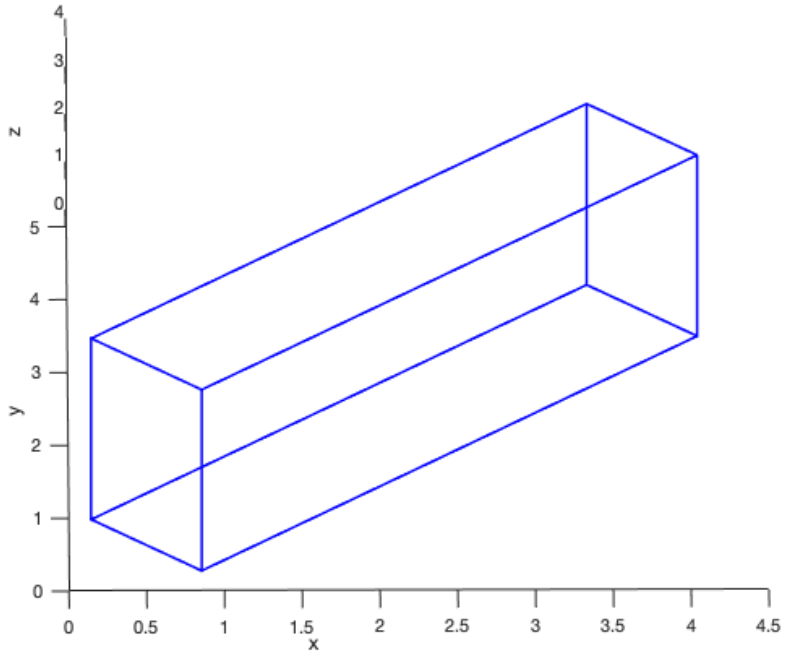
| | |
|---|---|
| 底面平行边的两中点 | (0.5, 0.5) |
| | (3.7, 3.7) |
| 底面边长 | 1 |
| z 轴范围 | [0.2, 4] |

图 4.1　数据处理测试障碍物底面




$(3.2, 3.2)$，则 $\vec{AB} \cdot \vec{M_0M_1} = 0.707106 \times 3.2 - 0.707106 \times 3.2 = 0$，说明边 $AB$ 与中点连线 $M_0M_1$ 相垂直；再验证相邻边是否垂直，$\vec{BC} = (4.05355, 3.34645) - (0.853553, 0.146447) = (3.199997, 3.200003)$，则此时 $\vec{AB} \cdot \vec{BC} = 0.707106 \times 3.199997 - 0.707106 \times 3.200003 \approx 0$，说明相邻边垂直。将计算顶点后的该障碍物在 Matlab 中绘制出来如图4.2。完成对障碍物顶点的计算之后，算法还将会对数据进行离散到指定精度下的操作，即 $(\lfloor \frac{x}{precision} \rfloor, \lfloor \frac{y}{precision} \rfloor, \lfloor \frac{z}{precision} \rfloor)$。此时精度指定为 0.1，故离散后的障碍物数据为底面矩形顶点为 [(1, 8), (33, 40), (40, 33), (8, 1)]，z 轴范围为 [2, 40]。出发点 (0,0,0) 离散后仍为 (0,0,0)，终点 (5,5,5) 离散后为 (50,50,50)。

图 4.2　测试障碍物

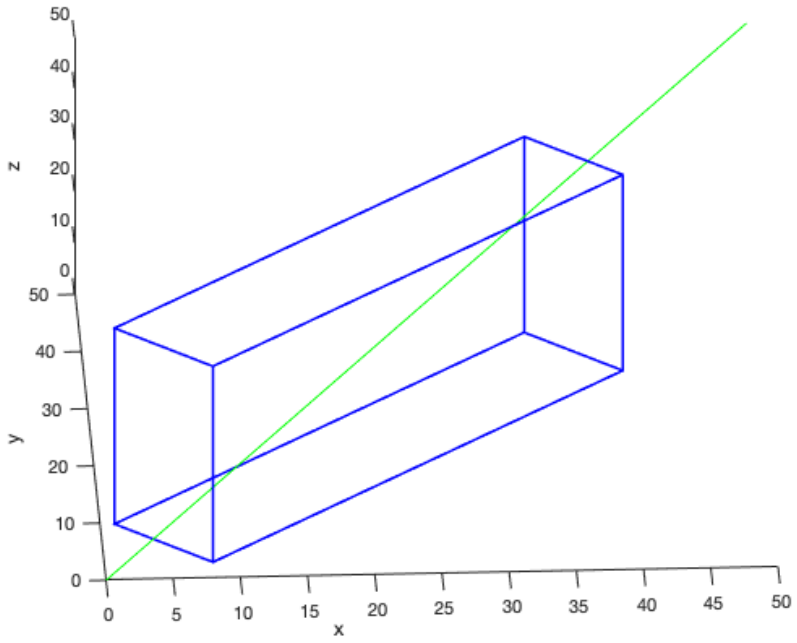



## 4.2　路径求解算法测试

关于路径求解算法的正确性测试，主要是障碍物碰撞检查算法和路径寻找算法正确性的测试，且数据为完成障碍物计算并离散到指定精度下的数据，障碍物碰撞检查算法主要是公式3.1和3.2的应用，路径寻找算法是应用 BFS 或 A* 算法在指定精度下寻找格点路径，路径拟合算法是通过结合路径寻找算法和障碍物碰撞检测算法从而找到一条最短的路径。数据采用上述表4.1中关于障碍物计算算法测试的数据，首先起点 P(0,0,0) 到终点 Q(50,50,50) 的线段与障碍物的关系如图4.3，此时线段是与障碍物相交的，即不可以直接到达的情况。将线段 $\vec{PQ} = P + k \cdot PQ = (0,0,0) + k(50-0, 50-0, 50-0) = (50k, 50k, 50k)$ 和障碍物代入障碍物碰撞检查算法进行测试，步骤如下：

1. 初始化可行解 k 范围为 [0,1]；
2. 对障碍物的 z 取值进行计算，由 $k_i = \dfrac{z_i}{50k}$ 解得新的可行解范围 [0.04,0.8]；
3. 对障碍物进行公式3.1检查，计算得到系数、常数分别为 $a_1 = 0, b_1 = -224, a_2 = 0, b_2 = -224$，此时退化为一次函数，由于 $b_1 b_2 > 0$ 可知满足公式，此时不对 k 范围做更新；

图 4.3　障碍物与起点到终点线段的关系




4. 对障碍物进行公式3.2检查，计算得到系数、常数分别为 $a_1 = -700, b_1 = 63, a_2 = 700, b_2 = -511$，此时 $\Delta = 98344960000 > 0$，解的新 k 的取值为 [0.09,0.73]；
5. 综上，存在 k 的有效范围为 [0.09,0.73]，故线段 PQ 与障碍物相交；

对寻路算法进行测试，使用算法对该起点、终点和障碍物进行寻路操作，此时找到的拟合格点路径所经过点如图4.4。此时，根据算法输出结果，将经过格点数据和障碍物在 Matlab 中绘制出来，如图4.5。此时可知该拟合路径满足不与障碍物相交的情况下，路径长度最短的条件。

图 4.4　寻路算法测试输出

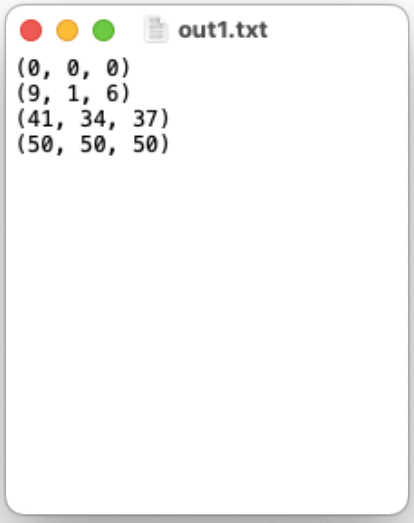
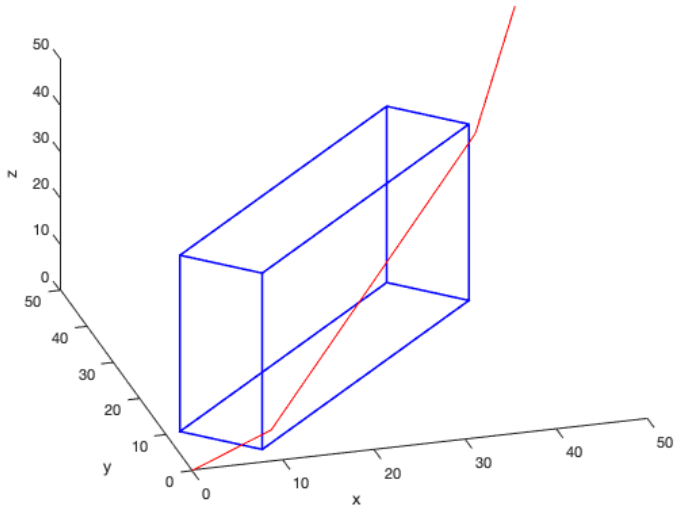



图 4.5　寻路算法测试路径与障碍物关系



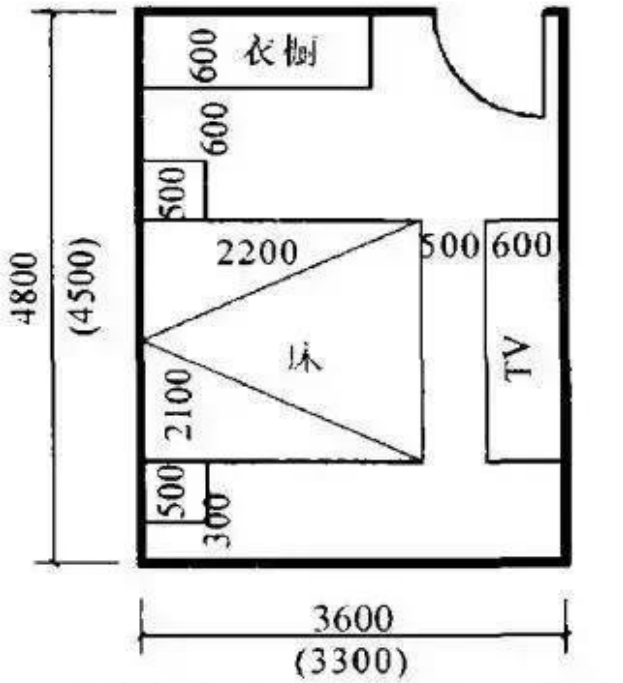
## 4.3 综合测试

完成对算法包含的主要模块：数据处理算法、路径求解算法的测试之后，还需要对整个算法进行全面的样例测试，需要包含多个场景，常见或稀有场景，以测试算法的稳定性。

### 4.3.1 常见室内场景

常见室内场景，其中包含沙发、板凳、电视和冰箱等物体，根据图4.6中所示实际空间，建模数据为表4.3，其中包含 5 个障碍物。 对数据完成预处理并离散到

图 4.6 常见室内空间图




注：括号内为房间内净尺寸，下同。

图1 适用型主卧室图
(17.28m²)

指定精度 0.1 后，得到离散后数据为表4.4，此时将建模后的数据在 Matlab 中绘制的模型俯视图和侧视图如图4.7和图4.8。 对数据进行寻路操作，设定起点为 (2, 0, 0)，终点为 (0.1, 5, 0.1)，离散在指定精度后起点为 (20, 0, 0)，终点为 (1, 50, 1)，算

**表 4.3　常见室内空间测试数据**

| | |
|---|---|
| 起点 | 2 0 0 |
| 终点 | 0.1 5 0.1 |
| 障碍物 1 | 0.25 0.3 0.25 0.8 0.5 0 0.5 |
| 障碍物 2 | 1.1 0.8 1.1 2.9 2.2 0 0.5 |
| 障碍物 3 | 0.25 2.9 0.25 3.4 0.5 0 0.5 |
| 障碍物 4 | 0.9 4 0.9 4.5 1.8 0 2 |
| 障碍物 5 | 3 0.8 3 2.9 0.6 0 1 |
| 精度 | 0.1 |

法得到的输出如图4.9，包含了起点、终点信息，障碍物信息以及最短路径通过的格点信息。将障碍物信息和路径信息使用 Matlab 绘制出来的模型的俯视图和侧视图分别如图4.10和图4.11，此时路径合法不经过障碍物并且满足路径长度最短。

### 4.3.2　复杂室内场景

考虑在简单室内场景的基础上，增加障碍物数量和路线的复杂度，因此建模为两个房屋相邻的情况，从一个房屋到另一个房屋内的最短路径求解情况。此时重复简单室内场景建模，原始数据为表4.5，对其完成离散到指定精度 0.1 后，得到离散后的数据如表4.6，此时将处理后数据使用 Matlab 绘制出来可得图像的俯视图和侧视图分别为图4.12和图4.13。由建模图像可知，封闭的房屋有门和窗两处开口，此时设计出发点在左侧房间内，为 (3, 0, 0)，离散后为 (30, 0, 0)；终点在右侧房间内，为 (7, 0, 0)，离散后为 (70, 0, 0)。所以若存在路线，应该是从左侧门进入左侧房间，通过左侧窗口离开左侧房间，再通过右侧窗口进入右侧房间，最后达到右侧门口的终点位置。使用算法在处理后的空间求出起点到终点的最短路径，此时算法输出的结果如图4.14，其中同样包括起点、终点、障碍物信息和路径经过的顶点信息。将输出信息使用 Matlab 绘制成图像，此时最短路径经过的路线图像的

**表 4.4　常见室内空间处理后数据**

| | |
|---|---|
| 起点 | 20 0 0 |
| 终点 | 1 50 1 |
| 障碍物 1 | [(0, 2), (5, 2), (5, 8), (0, 8)], (0, 5) |
| 障碍物 2 | [(0, 8), (22, 8), (22, 28), (0, 28)], (0, 5) |
| 障碍物 3 | [(0, 28), (5, 28), (5, 34), (0, 34)], (0, 5) |
| 障碍物 4 | [(0, 40), (18, 40), (18, 45), (0, 45)], (0, 20) |
| 障碍物 5 | [(27, 8), (32, 8), (32, 28), (27, 28)], (0, 10) |

俯视图和侧视图分别为图4.15和图4.16，可以看出此路径满足没有经过障碍物且路径最短的条件。

### 4.3.3　无法到达情况

除了考虑常见的室内场景和复杂的室内场景，还需要检查算法是否能在无法到达的情况下给出无法找到路径的提示，因此在简单室内场景中将起点和终点分别置于室内和室外，并且房屋密闭不留出门窗，此时是不存在有效路径可以完成起点到终点的。因此数据如表4.7，数据包含 10 个障碍物构成的房屋场景，以及指定精度为 0.1。对数据进行离散到精度 0.1 的操作，得到的数据为表4.8，此时将处理后的障碍物数据绘制到 Matlab 中，可以得到的建模信息的俯视图和侧视图分别为图4.17和图4.18。此时使用算法求解起点 (2, 1, 0)，离散后为 (20, 10, 0), 到终点 (0.1, 5, 0.1)，离散后为 (1, 50, 1) 的最短路径，此时算法输出为图4.19，无路径信息表示未找到有效路径，符合我们设计的条件。

图 4.7　常见室内空间处理后建模-俯视图

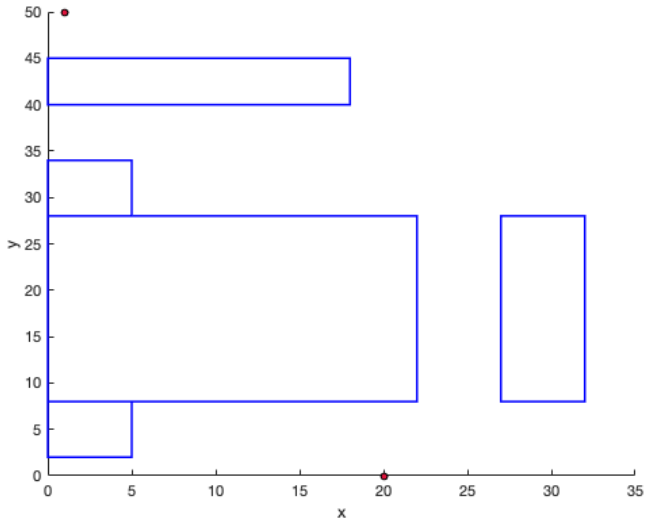



图 4.8　常见室内空间处理后建模-侧视图

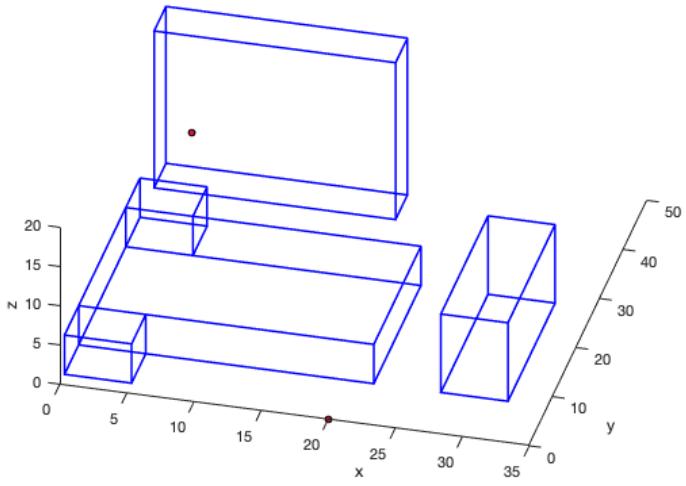


图 4.9　常见室内空间最短路径经过顶点

out1.txt

```
(20, 0, 0)
(1, 50, 1)
5
[(0, 2), (5, 2), (5, 8), (0, 8)], (0, 5)
[(0, 8), (22, 8), (22, 28), (0, 28)], (0, 5)
[(0, 28), (5, 28), (5, 34), (0, 34)], (0, 5)
[(0, 40), (18, 40), (18, 45), (0, 45)], (0, 20)
[(27, 8), (32, 8), (32, 28), (27, 28)], (0, 10)
6
(20, 0, 0)
(22, 7, 0)
(23, 18, 0)
(22, 29, 0)
(18, 46, 0)
(1, 50, 1)
```

图 4.10　常见室内空间最短路径结果-俯视图

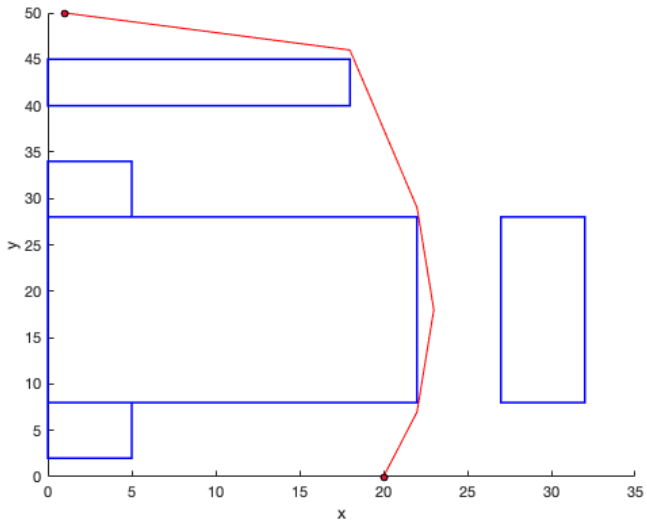


图 4.11　常见室内空间最短路径结果-侧视图

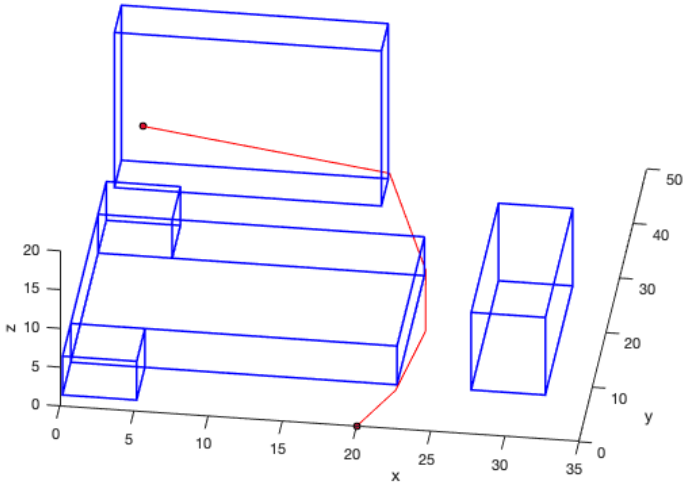



图 4.12　常见室内空间处理后建模-俯视图

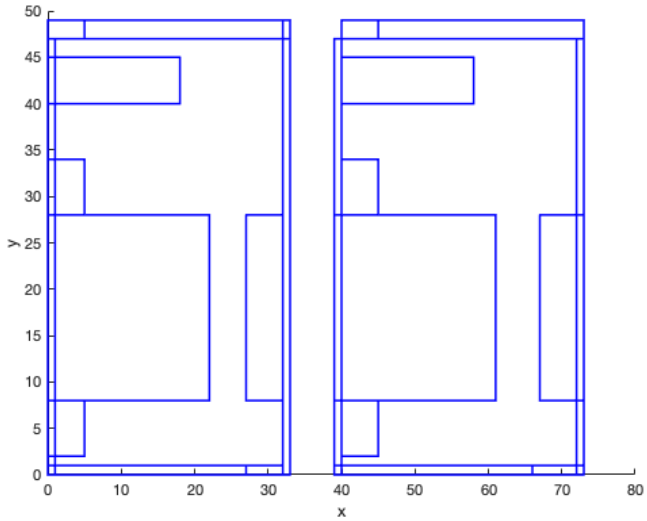


**表 4.5　复杂室内空间测试数据**

| | |
|---|---|
| 起点 | 2 0 0 |
| 终点 | 0.1 5 0.1 |
| 障碍物 | 1.35 0 1.35 0.1 2.7 0 2 |
| | 1.65 0 1.65 0.1 3.3 2 2.5 |
| | 0.25 0.3 0.25 0.8 0.5 0 0.5 |
| | 3.3 0 3.3 4.8 0.1 0 2.5 |
| | 0.05 0 0.05 4.8 0.1 0 2.5 |
| | 1.7 0 1.7 4.9 3.3 2.5 2.6 |
| | 1.7 4.8 1.7 4.9 3.3 0 2 |
| | 1.9 4.8 1.9 4.9 2.8 2 2.5 |
| | 1.1 0.8 1.1 2.9 2.2 0 0.5 |
| | 0.25 2.9 0.25 3.4 0.5 0 0.5 |
| | 0.9 4 0.9 4.5 1.8 0 2 |
| | 3 0.8 3 2.9 0.6 0 1 |
| | 5.35 0 5.35 0.1 2.7 0 2 |
| | 5.65 0 5.65 0.1 3.3 2 2.5 |
| | 4.25 0.3 4.25 0.8 0.5 0 0.5 |
| | 7.3 0 7.3 4.8 0.1 0 2.5 |
| | 4 0 4 4.8 0.1 0 2.5 |
| | . . . |
| 精度 | 0.1 |

**表 4.6　复杂室内空间处理后测试数据**

| | |
|---|---|
| 起点 | 2 0 0 |
| 终点 | 0.1 5 0.1 |
| 障碍物 | [(0, 0), (27, 0), (27, 1), (0, 1)], (0, 20) |
| | [(0, 0), (32, 0), (32, 1), (0, 1)], (20, 25) |
| | [(0, 2), (5, 2), (5, 8), (0, 8)], (0, 5) |
| | [(32, 0), (33, 0), (33, 47), (32, 47)], (0, 25) |
| | [(0, 0), (1, 0), (1, 47), (0, 47)], (0, 25) |
| | [(0, 0), (33, 0), (33, 49), (0, 49)], (25, 26) |
| | [(0, 47), (33, 47), (33, 49), (0, 49)], (0, 20) |
| | [(5, 47), (32, 47), (32, 49), (5, 49)], (20, 25) |
| | [(0, 8), (22, 8), (22, 28), (0, 28)], (0, 5) |
| | [(0, 28), (5, 28), (5, 34), (0, 34)], (0, 5) |
| | [(0, 40), (18, 40), (18, 45), (0, 45)], (0, 20) |
| | [(27, 8), (32, 8), (32, 28), (27, 28)], (0, 10) |
| | [(39, 0), (66, 0), (66, 1), (39, 1)], (0, 20) |
| | [(40, 0), (73, 0), (73, 1), (40, 1)], (20, 25) |
| | [(40, 2), (45, 2), (45, 8), (40, 8)], (0, 5) |
| | [(72, 0), (73, 0), (73, 47), (72, 47)], (0, 25) |
| | [(39, 0), (40, 0), (40, 47), (39, 47)], (0, 25) |
| | . . . |

图 4.13　常见室内空间处理后建模-侧视图

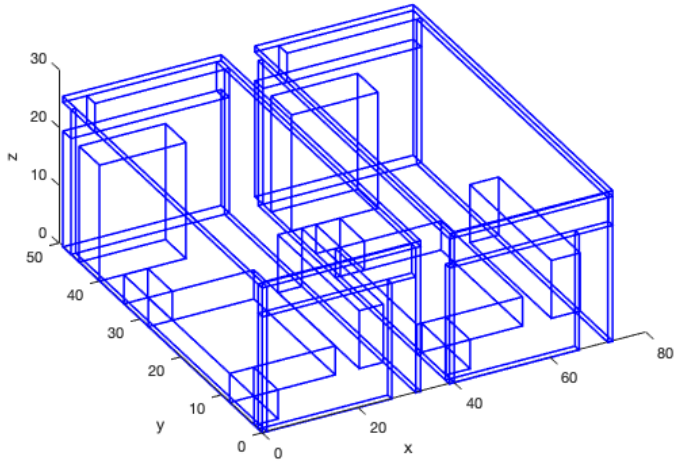



图 4.14　复杂室内空间最短路径结果

out2.txt

```
(30, 0, 0)
(70, 0, 0)
24
[(0, 0), (27, 0), (27, 1), (0, 1)], (0, 20)
[(0, 0), (32, 0), (32, 1), (0, 1)], (20, 25)
[(0, 2), (5, 2), (5, 8), (0, 8)], (0, 5)
[(32, 0), (33, 0), (33, 47), (32, 47)], (0, 25)
[(0, 0), (1, 0), (1, 47), (0, 47)], (0, 25)
[(0, 0), (33, 0), (33, 49), (0, 49)], (25, 26)
[(0, 47), (33, 47), (33, 49), (0, 49)], (0, 20)
[(5, 47), (32, 47), (32, 49), (5, 49)], (20, 25)
[(0, 8), (22, 8), (22, 28), (0, 28)], (0, 5)
[(0, 28), (5, 28), (5, 34), (0, 34)], (0, 5)
[(0, 40), (18, 40), (18, 45), (0, 45)], (0, 20)
[(27, 8), (32, 8), (32, 28), (27, 28)], (0, 10)
[(39, 0), (66, 0), (66, 1), (39, 1)], (0, 20)
[(40, 0), (73, 0), (73, 1), (40, 1)], (20, 25)
[(40, 2), (45, 2), (45, 8), (40, 8)], (0, 5)
[(72, 0), (73, 0), (73, 47), (72, 47)], (0, 25)
[(39, 0), (40, 0), (40, 47), (39, 47)], (0, 25)
[(40, 0), (73, 0), (73, 49), (40, 49)], (25, 26)
[(40, 47), (73, 47), (73, 49), (40, 49)], (0, 20)
[(45, 47), (73, 47), (73, 49), (45, 49)], (20, 25)
[(39, 8), (61, 8), (61, 28), (39, 28)], (0, 5)
[(40, 28), (45, 28), (45, 34), (40, 34)], (0, 5)
[(40, 40), (58, 40), (58, 45), (40, 45)], (0, 20)
[(67, 8), (73, 8), (73, 28), (67, 28)], (0, 10)
9
(30, 0, 0)
(8, 39, 20)
(4, 49, 21)
(19, 50, 21)
(33, 49, 21)
(37, 48, 21)
(41, 47, 21)
(46, 39, 20)
(70, 0, 0)
```

图 4.15　复杂室内空间最短路径结果-俯视图

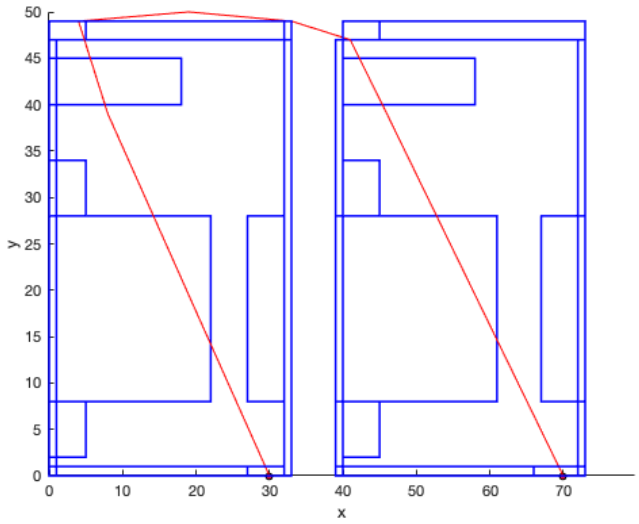



图 4.16　复杂室内空间最短路径结果-侧视图

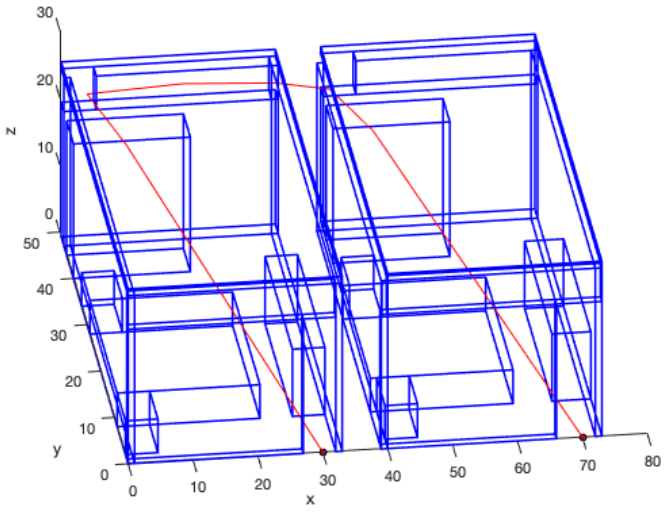


**表 4.7　无法到达情况空间测试数据**

| | |
|---|---|
| 起点 | 2 1 0 |
| 终点 | 0.1 5 0.1 |
| 障碍物 | 0.25 0.3 0.25 0.8 0.5 0 0.5 |
| | 1.1 0.8 1.1 2.9 2.2 0 0.5 |
| | 0.25 2.9 0.25 3.4 0.5 0 0.5 |
| | 0.9 4 0.9 4.5 1.8 0 2 |
| | 3 0.8 3 2.9 0.6 0 1 |
| | 1.7 0 1.7 0.1 3.3 0 2.5 |
| | 1.7 4.7 1.7 4.8 3.3 0 2.5 |
| | 0.05 0 0.05 4.8 0.1 0 2.5 |
| | 3.35 0 3.35 4.8 0.1 0 2.5 |
| | 1.7 0 1.7 4.8 3.3 2.5 2.6 |
| 精度 | 0.1 |

**表 4.8　无法到达情况空间处理后数据**

| | |
|---|---|
| 起点 | 20 0 0 |
| 终点 | 1 50 1 |
| 障碍物 | [(0, 2), (5, 2), (5, 8), (0, 8)], (0, 5) |
| | [(0, 8), (22, 8), (22, 28), (0, 28)], (0, 5) |
| | [(0, 28), (5, 28), (5, 34), (0, 34)], (0, 5) |
| | [(0, 40), (18, 40), (18, 45), (0, 45)], (0, 20) |
| | [(27, 8), (32, 8), (32, 28), (27, 28)], (0, 10) |
| | [(0, 0), (33, 0), (33, 1), (0, 1)], (0, 25) |
| | [(0, 47), (33, 47), (33, 47), (0, 47)], (0, 25) |
| | [(0, 0), (1, 0), (1, 47), (0, 47)], (0, 25) |
| | [(33, 0), (34, 0), (34, 47), (33, 47)], (0, 25) |
| | [(0, 0), (33, 0), (33, 47), (0, 47)], (25, 26) |

图 4.17　无法到达情况空间建模-俯视图

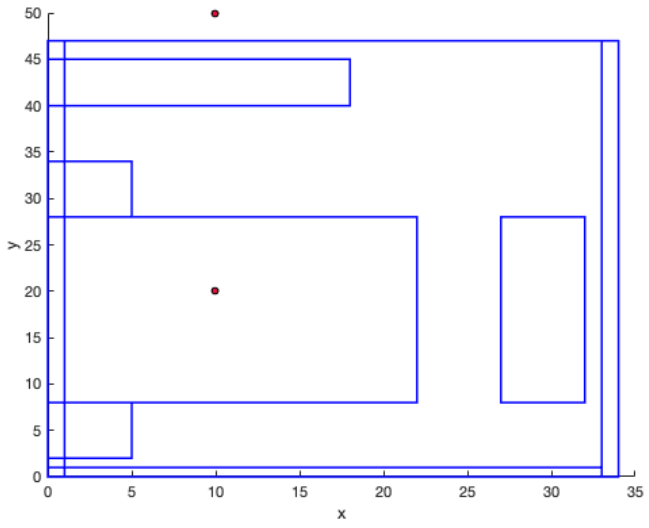


图 4.18　无法到达情况空间建模-侧视图

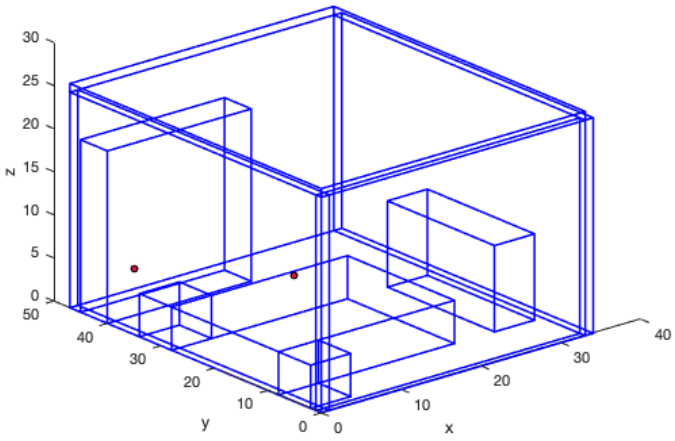



图 4.19　无法到达情况算法输出

out3.txt

```
(20, 10, 0)
(1, 50, 1)
10
[(0, 2), (5, 2), (5, 8), (0, 8)], (0, 5)
[(0, 8), (22, 8), (22, 28), (0, 28)], (0, 5)
[(0, 28), (5, 28), (5, 34), (0, 34)], (0, 5)
[(0, 40), (18, 40), (18, 45), (0, 45)], (0, 20)
[(27, 8), (32, 8), (32, 28), (27, 28)], (0, 10)
[(0, 0), (33, 0), (33, 1), (0, 1)], (0, 25)
[(0, 47), (33, 47), (33, 47), (0, 47)], (0, 25)
[(0, 0), (1, 0), (1, 47), (0, 47)], (0, 25)
[(33, 0), (34, 0), (34, 47), (33, 47)], (0, 25)
[(0, 0), (33, 0), (33, 47), (0, 47)], (25, 26)
```

## 4.4 本章总结

本章主要是对整个算法正确性进行了测试，使用了相关的测试样例和使用了Matlab 绘图辅助显示。对主要模块：数据处理算法、路径求解算法进行了测试并分析，数据处理算法主要是代码3.1的应用，即将输入只包含底面两中点、边长和z 轴范围的障碍物数据处理出底面顶点坐标信息，进而可以被后续算法使用；路径求解算法主要测试关于使用 BFS 或 A* 算法求解起点到终点的格点路径，以及使用公式3.1和3.2的应用以判断线段与障碍物的关系以判断是否路径可以拟合。

其次是对整个算法进行综合测试，构造了常见室内场景空间、复杂场景空间以及无法到达情况空间的数据以测试算法正确性，通过分析算法输出和 Matlab 辅助显示判断算法是否符合预期效果。

至此，整个算法已经完成构思、编码和测试的环节。

# 参考文献


[1] 卜月华; 吴建专; 顾国华; 殷翔, 《图论及其应用》第一版, 东南大学出版社: 1-2, 2007

[2] 作者. 书名 [M]. 版次. 出版地：出版单位，出版年份：起止页码.

[3] 邓建松等, 《LaTeX $2_\varepsilon$ 科技排版指南》, 科学出版社.

[4] 吴凌云, 《CTeX FAQ (常见问题集)》, *Version 0.4*, June 21, 2004.

[5] Herbert Voß, Mathmode, `http://www.tex.ac.uk/ctan/info/math/voss/mathmode/Mathmode.pdf`.

# 致　　谢

感谢你, 感谢他和她, 感谢大家.