第1章　计算几何：导言

正漫步于校园的你，突然需要打一个紧急电话. 在遍布于校园之中的各个公用电话中，你当然想找到离自己最近的那部. 然而，哪部电话才是最近的呢？一张校园地图能够提供帮助，无论你身处何处，都可以在地图上找到最近的公用电话. 这张地图可能会将整个校园划分成不同区域，每个区域都对应着一部最近的公用电话（如图 1-1 所示）. 这些区域会是什么样子呢？又该如何计算出它们呢？

尽管这还算不上是一个至关重要的问题，但它还是简要描述了一个主要的几何概念，而这一概念在众多应用中都扮演着重要的角色.

对校园如此划分之后，就得到了所谓的 Voronoi 图（Voronoi diagram），我们将在第 7 章对这一结构详细探讨. 我们可以用它来为由多个城市组成的商业区域建立模型，指挥机器人，甚至描述和模拟晶体的生长过程. 为了构造像 Voronoi 图这样的几何结构，需要一些几何算法. 这些算法就是本书的主题.

第二个例子. 假设你已经找到了最近的公用电话. 只要手中有一份地图，你就能很容易沿着一条很短的路径到达电话的位置，而且中途不会撞上墙或者其他障碍物（如图 1-2 所示）. 然而，想要通过程序让机器人自己来完成这一任务，却要困难得多.

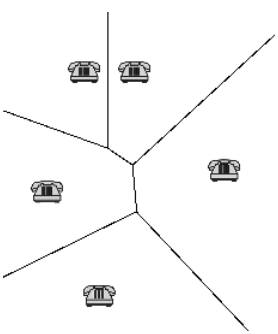

图 1-1　按照公用电话的分布，可以将校园划分为若干区域

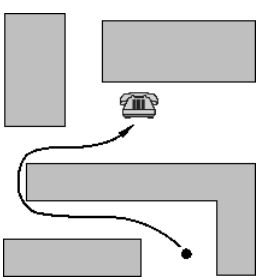

图 1-2　从当前位置通往某一公用电话的最短路径

与上例相同，这一问题的实质也是几何的：给定一组几何形状不同的障碍物，需要在不与任何障碍物发生碰撞的前提下，找出连接于任意两点之间的最短路径. 这就是所谓的运动规划（motion planning），在机器人学中，这类问题的求解是至关重要的. 第 13 章和第 15 章，将针对运动规划所需的几何算法进行讨论.

第三个例子.假设你可以利用不止一张地图,而是两张,一张描述了各个建筑物,包括公用电话；另一张则画出了校园内的道路.为了规划出通往公用电话的运动路径,我们需要将这两张地图叠合(overlay)起来,也就是说,需要将这两张地图所提供的信息合并起来.在地理信息系统(geographic information system,GIS)中,地图的叠合是基本的操作之一.这种操作涉及某张地图中的对象在另一张地图中的定位、不同特征物之间的求交计算等问题.第2章将讨论这一问题.

许多几何问题的解决,都要依靠设计精巧的几何算法,上面只是其中的三个例子.计算几何这一研究领域出现于20世纪70年代,它旨在解决的就是这类几何问题.这一学科可以被定义为"针对处理几何对象的算法及数据结构的系统化研究",其重点在于"渐进快速的精确算法".由几何问题而带来的挑战,吸引了众多的研究人员.从对问题的明确表述到得出高效而优雅的解决方法,往往需要经历漫长的过程,其间既要克服很多困难,也要积累一些次优的中间结果.今天,我们已经掌握了功能广泛的一整套几何算法,它们不仅高效,而且相对更加易于理解和实现.

本书将介绍计算几何中最重要的概念、方法、算法以及数据结构,但愿我们的介绍方式能够吸引有志于将计算几何的研究成果付诸实际应用的读者.每一章都从某一实际问题入手,而这种问题的求解,都需要借助于几何算法.为了说明计算几何应用范围的广泛性,这些问题分别选自机器人学、计算机图形学、CAD/CAM以及地理信息系统等不同的应用领域.

然而你并不能指望在解决应用领域中的主要问题时,总是有现成的软件可以直接利用.这里的每一章,只是孤立地对计算几何中的某一特定概念进行讨论；其中所涉及的应用问题,只是作为一个例子,用以导出有关的概念,继而展开介绍.这些例子可以使我们体会到,如何才能针对工程性问题建立(数学)模型,进而得出严谨的解答.

1.1 凸包的例子

面对具有几何本质的算法问题,我们所采用的解决方法,大多具备两方面要素：一是对该问题几何特性的深刻理解,二是算法和数据结构的合理应用.如果对某个问题的几何性质尚不甚了解,那么纵然精通世界上所有的算法,也依然不能高效地解决它.反过来,如果不知道有哪些算法技术适用于这个问题,那么即使你已经对问题的几何特性烂熟于胸,也是枉然.通过本书,你将对最为重要的若干几何概念以及算法技术有个透彻的理解.

为了说明在几何算法的建立过程中所出现的问题,本节将讨论曾在计算几何中首先研究的问题之一——平面凸包的计算.这里忽略该问题的来由,对此有兴趣的读者,可以阅读第11章(该章讨论的是三维凸包问题)的导言部分.

平面的一个子集 S 被称为是“凸”的，当且仅当对于任意两点 $p,q\in S$，线段 $\overline{pq}$ 都完全属于 S（如图 1-3 所示）. 集合 S 的凸包 $\mathcal{CH}(S)$，就是包含 S 的最小凸集，更准确地说，它是包含 S 的所有凸集的交.

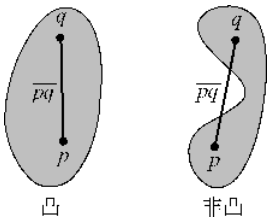

图 1-3 凸集与非凸集

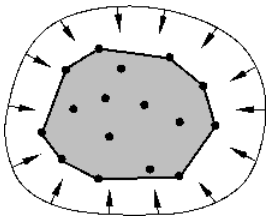

图 1-4 凸包的直观理解

这里所要讨论的是如何计算平面上由 n 个点组成的有限集合 P 的凸包. 我们可以借助一个虚构式实验，来想象这种凸包的模样：如图 1-4 所示，将这里的点想象成钉在平面上的钉子；取来一根橡皮绳，将它撑开围住所有的钉子，然后松开手，啪的一声，橡皮绳将紧绷到钉子上，它的总长度也将达到最小. 此时，由橡皮绳围住的区域就是 P 的凸包. 因此，我们也可以将平面有限点集 P 的凸包定义为，顶点取自于 P、包含 P 中所有点的惟一凸多边形. 当然，这一定义是否没有歧义（也就是说，此多边形是否惟一），以及这一定义是否等同于前面所给出的定义，都需要严格地予以证明，然而鉴于这是本章的导言，我们将跳过这一环节.

如何来计算凸包呢？在回答这一问题之前，必须首先回答另一个问题：所谓“计算凸包”，到底是什么含义？正如我们已经看到的，P 的凸包是一个凸多边形. 表示多边形的一种自然的方法，就是从任一顶点开始，沿顺时针方向依次列出所有顶点. 因此，我们所要求解的问题就转化为：

> 给定平面点集[①] $P=\{p_1,\cdots,p_n\}$，通过计算从 P 中选出若干点，它们沿顺时针方向依次对应于 $\mathcal{CH}(P)$ 的各个顶点（参见图 1-5）.
>
> input＝平面上一组点：$p_1,p_2,p_3,p_4,p_5,p_6,p_7,p_8,p_9$.
>
> output＝凸包的表示：p_4,p_5,p_8,p_2,p_9.

当着手设计一个计算凸包的算法时，凸包的前一个定义对我们没有多少帮助. 按照此定义，需要计算出“包含 P 的所有凸集的交”，可是这种集合有无限多个. 而我们所观察到的“$\mathcal{CH}(P)$ 是一个凸多边形”这一事实，则更有帮助. 下面，我们就来看看 $\mathcal{CH}(P)$ 是由哪些边构成的.

这些边的端点 p 和 q 都来自于 P；另外，只要适当地定义由 p 和 q 所确定的直线的

① 译者注：更准确地，应该是“平面有限点集”.

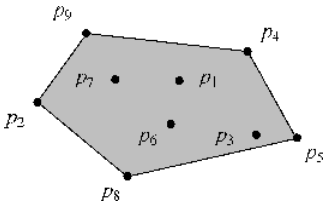

图 1-5 计算凸包

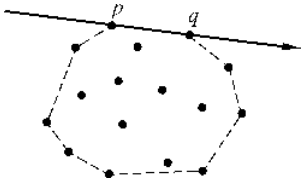

图 1-6 相对于 $\mathcal{CH}(P)$ 边界上任一边所在的直线，P 中所有点均居于同侧

方向，使得 $\mathcal{CH}(P)$ 总是位于其右侧，那么 P 中的所有点也都落在该直线的右侧（如图 1-6 所示）. 反之亦然. 如果相对于由 p 和 q 所确定的直线，$P\setminus\{p,q\}$ 中的所有点都位于右侧，那么 $\overline{pq}$ 就构成了 $\mathcal{CH}(P)$ 的一条边.

在对该问题的几何特性有了更深的理解之后，我们就可以构造一个算法了. 我们通过伪代码来描述该算法，本书将统一采用这种伪代码的形式.

Algorithm SLOWCONVEXHULL(P)
Input. 平面点集 P.
Output. 由 $\mathcal{CH}(P)$ 的顶点沿顺时针方向排成的队列 $\mathcal{L}$.
1. $E \leftarrow \varnothing$.
2. **for** (每一有序对 $(p,q) \in P \times P, p \neq q$)
3. **do** $valid \leftarrow$ **true.**
4. **for** (除 p 和 q 之外的所有点 $r \in P$).
5. **do if** (r 位于 p 和 q 所确定有向直线的左侧).
6. **then** $valid \leftarrow$ **false.**
7. **if** ($valid$) **then** 将有向边 $\overrightarrow{pq}$ 加入到 E.
8. 根据集合 E 中的各边，找出 $\mathcal{CH}(P)$ 的所有顶点，并按照顺时针方向将它们组织为列表 $\mathcal{L}$.

也许，你对该算法中的两个步骤还不甚清楚.

第一处出现在第 5 行：应该如何进行比较，才能判断某个点到底是位于一条有向直线的左侧，还是右侧？对于大多数几何算法而言，这都是必需的基本操作之一. 本书假定，这些操作都是现成的. 显然，（从理论上分析）它们都可以在常数时间内完成，因此从渐进复杂度的角度看，算法的具体实现方法不会对其运行时间的数量级有任何影响. 但这并不是说这些基本操作不甚重要，或者不值一提. 实际上，要想正确地实现这些操作并非易事，

而且它们对算法实际的运行时间的确会有影响. 幸运的是,包括这些基本操作的软件包现已随处可得. 因此,我们不必担心如何实现第 5 行中的测试; 可以假定我们已经拥有一个子函数,(通过调用该函数)可以在常数时间内完成这类测试.

第二处出现在最后一行. 通过第 2～7 行的循环,可以构造出凸包的边集 E. 根据 E,可以按照如下方法构造出列表 $\mathscr{L}$. E 中各边都是有向的,因此可以定义它们的起点与终点. 在指定每条边的方向时,我们使得其他的所有点都位于它的右侧. 这样,如果按照顺时针方向遍历所有顶点,那么每条边的起点都会先于其终点被枚举出来.

现在,如图 1-7 所示,在 E 中任意删除一条边$\overrightarrow{e_1}$,将$\overrightarrow{e_1}$的起点、终点分别作为第一、第二个点,放入 $\mathscr{L}$; 从 E 中找出以$\overrightarrow{e_1}$的终点为起点的边$\overrightarrow{e_2}$,将$\overrightarrow{e_2}$从 E 中删去,并将其终点插入到当前 $\mathscr{L}$ 的末尾; 接下来,再找出以$\overrightarrow{e_2}$的终点为起点的边$\overrightarrow{e_3}$,将$\overrightarrow{e_3}$从 E 中删去,也将其终点插入到当前 $\mathscr{L}$ 的末尾……不断重复上述过程,直到 E 中只剩下最后一条边. 到这时,就已经大功告成了. 因为,最后这条边的终点必然就是$\overrightarrow{e_1}$的起点,而这个点已经加入到 $\mathscr{L}$ 中了. 如果是直截了当地实现,这一过程需要 $O(n^2)$ 的时间. 虽然将这一复杂度改进至 $O(n\log n)$ 并不困难,但是毕竟算法的整体复杂度已经由其他部分决定了.

SLOWCONVEXHULL 算法的复杂度并不难分析. 总共要检查 n^2-n 对点. 对每一对点,要检查其他的 $n-2$ 个点,看看它们是否都位于该点对所确定的有向线段的右侧. 这总共需要运行 $O(n^3)$ 的时间. 最后一步需要 $O(n^2)$ 的时间,故总体的时间复杂度为 $O(n^3)$. 在实际应用中,这样一个需要运行三次方时间的算法,除非是处理小规模的输入集,否则都会由于太慢而毫无用处. 之所以会出现这种问题,是因为我们没有采用任何精巧的算法设计技术,而只是以一种蛮力(brute-force)的方式,将对算法的几何理解直接转换为算法. 实际上,只要对该算法作进一步的审视,就不难找出改进的方法.

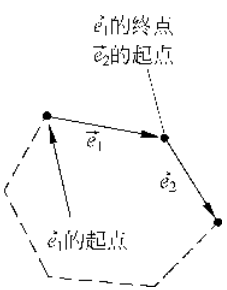

图 1-7 确定 E 中各边的次序

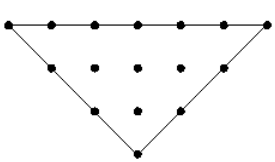

图 1-8 多点共线的退化情况

前面介绍了一个准则,借以判定点对 p 和 q 是否定义了 $\mathscr{CH}(P)$ 的一条边. 然而,在推导这个准则时,我们做得还不够细致. 如图 1-8 所示,相对于由 p 和 q 所确定的直线,一个点 r 的位置并不是非左即右. 有时,可能正好落在这条直线上,这就是所谓的“退化情况”

(degenerate case)，或者简称为“退化”(degeneracy). 对于这种情况，应该如何处理呢？在刚开始思考某个问题的时候，我们更愿意(暂时地)忽略这些情况，这样，在从问题中抽取出其几何性质的过程中，才不至于把思路搞乱. 然而在实践中，这些情况都有可能发生. 例如，当借助鼠标在屏幕上标定点的位置时，每个点的坐标都将局限在很窄的一段整数区间之内，因而很有可能会定义出三个共线的点.

在可能出现退化情况时，为了保证算法始终运行正确，就必须这样来重新表述上述准则：某条有向边$\overrightarrow{pq}$是$\mathcal{CH}(P)$的一条边，当且仅当相对于由 p 和 q 所确定的有向直线，所有的其他点 $r\in P$ 或者严格地位于其右侧，或者落在开线段$\overline{pq}$上(我们假定，P 中没有相互重合的点). 这样，此算法第 5 行所涉及的测试，将被替换为一个更为复杂的版本.

还有一个重要的方面也被忽视了，而它却会影响到算法所得出结果的正确性. 不知不觉中，我们已经做了这样一个假定：只要给定一条(有向)直线，以及另外一个点，那么无论这个点是位于该直线的左侧还是右侧，我们总是能够准确地作出判断. 然而，这个假设并不见得一定成立. 如果各点的坐标都表示为浮点数，而且计算过程中所采用的也是浮点运算，那么就必然存在舍入误差(rounding error)，从而影响到测试的精度.

如图 1-9 所示，试想有三个点 p、q 和 r 几乎共线，而其他各点与它们都相距很远. 按照上面的算法，要分别对点对(p,q)、(r,q)和(p,r)进行测试. 既然这三个点几乎共线，由于舍入误差的存在，判断的结果很有可能是：r 位于直线$\overline{pq}$的右侧，p 位于直线$\overline{rq}$的右侧，而 q 位于直线$\overline{pr}$的右侧. 显然，这种几何位置关系是不可能的. 然而浮点运算可不管这些！在这种情况下，算法将会把这三条边全都挑选出来.

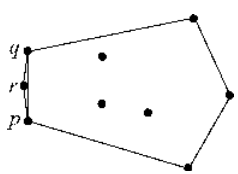

图 1-9 三点几乎共线，且与其他诸点相距足够远时，可能选出多余的边

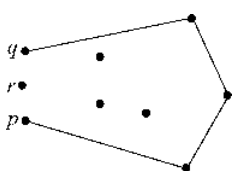

图 1-10 三点几乎共线，且与其他诸点相距足够远时，可能会遗漏边

更糟糕的情况是，这三次测试的结果也可能正好与上面相反，这样，如图 1-10 所示，算法就会将这三条边都排除掉，于是在所生成的“凸包”边界上将会出现一个缺口.

等到算法的最后一步，将凸包的顶点组织为有序表时，会导致严重的错误. 实际上，这一步有个假定：在凸包的每一顶点处，出边和入边都正好只有一条. 然而受到舍入误差的

影响，某个顶点 p 可能会有两条出边，也可能根本就没有出边. 在上面那个简单的算法中，最后一行并没有考虑到对不一致数据的处理，因此，若直接按照该算法编程实现，程序就可能会崩溃.

即使我们已经证明该算法是正确的，而且也能够处理各种特殊情况，它可能依然称不上稳健(robust)，也就是说，计算过程中出现的某个微小误差，可能会导致运行失败，而且失败的形式难以预料. 问题的症结在于，在证明算法正确性的时候，我们(想当然地)做了一个假设：可以精确地使用实数进行计算.

我们设计出了自己的第一个几何算法. 它可以计算平面点集的凸包. 然而，它的时间复杂度为 $O(n^3)$，故运行速度相当慢；它处理退化情况的能力也很差；此外，也不够稳健. 因此，我们应该尽力去做得更好.

为此，我们将采用一种标准的算法设计模式——递增式策略，来设计一个递增式算法(incremental algorithm). 顾名思义，我们将逐一引入 P 中各点，每增加一个点，都要相应地更新目前的解. 这个递增式方法将沿用几何上的习惯，按照由左到右的次序加入各点. 于是，首先需要根据 x 坐标对所有点进行排序，产生一个有序的序列 $p_1, p_2, \cdots, p_n$. 接下来，按照这一顺序，将它们逐一引入. 本来，既然是从左到右地进行处理，所以要是凸包上的顶点也能按照它们在边界上出现的次序自左向右地排列，将会更加方便. 然而，情况并没有这样好. 因此，我们将首先计算出构成上凸包(upper hull)的那些顶点.

如图 1-11 所示，所谓的上凸包，就是从最左端顶点 p_1 出发，沿着凸包顺时针行进到最右端顶点 p_n 之间的那段. 换言之，组成上凸包的，就是从上方界定凸包的那些边. 此后，再自右向左进行一次扫描，计算出凸包的剩余部分——下凸包(lower hull).

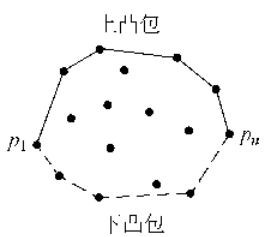

图 1-11 分别构造上凸包和下凸包

该递增式算法的基本步骤，就是在每次新引入一个点 p_i 之后，对上凸包进行相应的更新. 也就是说，已知点 p_1，p_2，…，p_{i-1} 所对应的上凸包，计算出 $p_1, p_2, \cdots, p_i$ 所对应的上凸包. 可以按照如下方法进行. 若按照顺时针方向沿着多边形的边界行进，则在每个顶点处都要改变方向. 若是任意的多边形，则每次的转向既可能是向左，也可能向右. 然而若是凸多边形，则必然每次都是向右转. 根据这一点，在新引入 p_i 之后，可以进行如下处理. 令 $\mathscr{L}_{\text{upper}}$ 为从左向右存放上凸包各顶点的一个列表. 首先，我们将 p_i 接在 $\mathscr{L}_{\text{upper}}$ 的最后. 既然在目前已经加入的所有点中，p_i 是最靠右的，则它必然是(当前)上凸包的一个顶点，所以这样做无可厚非. 然后，我们要检查 $\mathscr{L}_{\text{upper}}$ 中最末尾的三个点，看看它们是否构成一个右拐. 若构成右拐，则大功告成，此时(更新后的) $\mathscr{L}_{\text{upper}}$ 记录了组成上凸包的各个顶点 $p_1, p_2, \cdots, p_i$，接下来，就可以继续处理下一个点 p_{i+1}. 然而，若最后的三个点构成一个左拐，就必须将中间的(即倒数第二个)顶点从上