北京航空航天大學
BEIHANG UNIVERSITY

实验报告

内容：模拟人群的对流和疏散

院（系）名称	计算机学院
专业名称	计算机科学与技术
指导教师	宋晓
学号姓名	16061028 刘乔杨
	16061040 王科翔
	16061037 刘赫铭
	16061125 周雨飞

2018 年 12 月

目录

第一部分 社会力模型实验报告

一、绪言

人群疏散是安全领域的一项重要内容，合理有效的逃生策略及疏散措施可以减少伤亡、挽救生命。与车辆交通相比，行人交通的动力学特性更加复杂。至今为止，学者们从不同角度出发，提出了多种行人运动模型，总体上可以分为宏观模型和微观模型。宏观模型，主要研究人群流量等宏观特性，而忽略单个个体的运动状态，比如利用流体力学模拟人流。近年来，随着计算机技术以及相关理论的发展，研究热点逐渐转向微观模型。与宏观模型不同，微观模型着眼于个体行为的建模，宏观群体特征视为个体间相互作用的涌现。常用的微观模型有元胞自动机模型、格子气模型、社会力模型、基于多智能体的模型等。

社会力模型是常用的连续微观模型，它由 Helbing 等学者在社会力概念的基础上结合牛顿经典力学而提出。在模型中，社会心理等各种影响行人运动的因素被量化成具体的力，行人的运动由经典牛顿力学规律决定。例如，一个质量 70kg 的行人想要以 $1\mathrm{m}/s^2$的加速度追赶正前方的同伴，这种心理作用可以等效为行人受到指向正前方的、大小为 70N 的作用力。作为连续模型，社会力模型具有元胞自动机等离散模型不具有的一些优点，比如行人可以朝任意方向移动、紧急情况下行人之间可以产生不同程度的重叠等，这使得社会力模型仿真出来的行人更接近现实。另外，社会力模型可以模拟仿真出自动渠化、瓶颈摆动、欲速则不达等多种行人自组织现象。

虽然社会力模型具有众多优点，但仍存在一些局限性，尤其在仿真人群恐慌、行人密度过大或过小等情况时经常产生一些不合实际的现象。例如，原始社会力模型中，行人的弹性系数取得非常大，这意味着两个行人之间若有 5cm 的重叠，将会产生 6000N 的压力。这么大的力，即使不考虑人的承受能力，也会使人产生巨大的加速度以及随后产生的速度跳变、位置跳变等一系列不符合实际的现象。

在上文所述的社会力模型的优点和缺点的基础之上，我们进行了本次实验。

二、实验目的

编程实现社会力模型，并对其中的细节进行改进，模拟人群的对流和疏散场

景。通过 GUI，将人群的运动过程直观地展现出来，验证社会力模型仿真人群运动的准确性。

三、数学模型

1. 社会力模型

在社会力模型中，行人用一个个具有一定质量和半径的圆来表示。行人i的运动，按照其所受各种力的合力，依据牛顿第二定律来决定

$$m_i \frac{\mathrm{d}\vec{v}_i(t)}{\mathrm{d}t} = \vec{F}_i^{\mathrm{drv}}(t) + \sum_{j \neq i} \vec{F}_{ij}(t) + \sum_W \vec{F}_{iW}(t),$$

其中，m_i、$\vec{v_i}(t)$分别为行人i的质量和速度，t 为时间。模型中，行人i总共受到三种力的作用，分别为行人i的自驱动力$\vec{F}_i^{drv}$、其他行人j对行人i的作用力$\vec{F}_{ij}$以及墙或障碍物W对行人i的作用力$\vec{F}_{iW}(t)$.

2. 自驱动力

对于行人i，其期望在时间τ_i内由目前的实际速度$\vec{v}_i(t)$到达期望速度$v_i^0(t)\vec{e}_i^0(t)$. 其中$v_i^0(t)$为期望速度大小，单位向量$\vec{e}_i^0(t)$为期望速度方向。这种作用可以用以下自驱动力的形式来表示

$$\vec{F}_i^{\mathrm{drv}}(t) = m_i \frac{v_i^0(t)\vec{e}_i^0(t) - \vec{v}_i(t)}{\tau_i}.$$

在本次实验中，$\vec{e}_i^0(t)$即期望速度方向是由A^*算法给出的。

3. 行人之间相互作用力

行人不仅受到自身驱动力$\vec{F}_i^{drv}$的作用，还受到其他行人的影响。相关研究显示，行人期望与其他行人之间保持一定距离，这种心理作用可以通过排斥力$\vec{F}_i^{soc}$来实现。在社会力模型中，$\vec{F}_i^{soc}$采用以下形式

$$\vec{F}_{ij}^{\mathrm{soc}} = A_i \exp\left[\frac{r_{ij} - d_{ij}}{B_i}\right] \vec{n}_{ij},$$

其中，$r_{ij}=r_i+r_j$为行人i的半径r_i与行人j的半径r_j的和，$d_{ij}=\left\|\vec{l_i}-\vec{l_j}\right\|$为行人$i$的圆心$\vec{l_i}$与行人j的圆心$\vec{l_j}$之间的距离，单位向量$\vec{n}_{ij}=\left(n_{ij}^1,n_{ij}^2\right)=\frac{\vec{l_i}-\vec{l_j}}{d_{ij}}$由j指向$i$. $A_i=2000\mathrm{N},B_i=0.08m.$

当行人i与行人j发生身体接触时，会产生物理作用力。j对i的物理作用力记为$\vec{F}_{ij}^{phy}$，由两部分组成，一是身体挤压造成的弹力$\vec{F}_{ij}^{phy1}=k(r_{ij}-d_{ij})\vec{n}_{ij}$，一是相对运动产生的滑动摩擦力$\vec{F}_{ij}^{phy2}=\kappa(r_{ij}-d_{ij})\Delta v_{ji}^t\vec{t}_{ij}$，其中$\vec{t}_{ij}=(-n_{ij}^2,n_{ij}^1)$为切方向，$\Delta v_{ji}^t=(\vec{v}_j-\vec{v}_i)\cdot\vec{t}_{ij}$为相对速度在切方向上的投影。参数$k=1.2\times10^5kg\cdot m^{-1}\cdot s^{-1}$,$\kappa=2.4\times10^5kg\cdot m^{-1}\cdot s^{-1}$.

综上，行人j对行人i的作用力可表示为

$$\begin{aligned}\vec{F}_{ij}(t)&=\vec{F}_{ij}^{\mathrm{soc}}+\vec{F}_{ij}^{\mathrm{phy}}\\&=\left\{A_i\exp\left[\frac{r_{ij}-d_{ij}}{B_i}\right]+kg(r_{ij}-d_{ij})\right\}\vec{n}_{ij}+\kappa g(r_{ij}-d_{iW})\Delta v_{ji}^t\vec{t}_{ij},\end{aligned}$$

其中

$$g(x)=\begin{cases}x, & x>0,\\0, & x\leq0.\end{cases}$$

4. 行人与障碍物之间的作用力

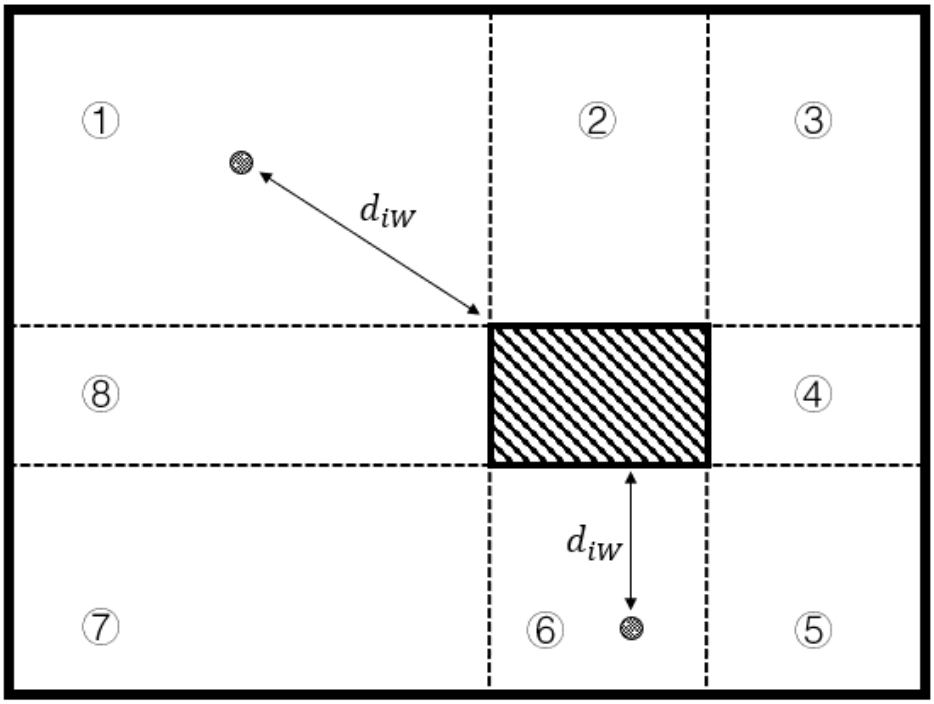
与行人之间的作用力类似，墙或障碍物W对行人i的作用力表示成以下形式

$$\vec{F}_{iW}(t)=\left\{A_i\exp\left[\frac{r_i-d_{iW}}{B_i}\right]+kg(r_i-d_{iW})\right\}\vec{n}_{iW}-\kappa g(r_i-d_{iW})(\vec{v}_i\cdot\vec{t}_{iW})\vec{t}_{iW},$$

其中d_{iW}为行人i圆心到W的距离，$\vec{n}_{iW}$和$\vec{t}_{iW}$为障碍物W相应的法向量和切向量。

在原始的社会力模型中，行人i圆心到W的距离这一概念并没有被准确定义。尤其是当障碍物在场景地图中占据一定面积的时候，对于行人而言，能够感知到的只有障碍物与自己最接近的那一部分，而无法感知到障碍物的全貌。因此，如果对障碍物W做分割逐块求作用力，或取障碍物W的质心算距离，在社会力模型中都是不符合实际的。因此，我们在本实验中把墙或障碍物W与行人i之间的距离定义为行人i与障碍物W的边界的最短距离。下图显示了这种距离计算方式的

具体细节



不失一般性，我们假设墙和障碍物都是矩形。四周的围墙到行人i距离的计算十分简单，取垂直距离即可，上图中暂且略去。重点在于行人i到障碍物W距离的计算。以障碍物W的四条边为界线，对地图区域进行划分，将矩形地图分为8个小矩形。当行人i处于①③⑤⑦这四个区域中时，按照其与障碍物W最近的一个角计算距离d_{iW}，如上图区域①所示；当行人处于②④⑥⑧这四个区域中时，按照其与障碍物最近的一条边的垂直距离计算距离d_{iW}.这种计算距离的方式很好地模拟了行人对场景中障碍物的感知，相比于逐点或逐边计算，减轻了模型给计算机带来的计算压力。

5. A^*算法

A^*搜索算法是一种在静态路网中求解最短路径最有效的直接搜索方法，也是许多其它问题的常用启发式算法。启发式搜索算法的关键，在于从当前搜索结点往下选择下一步结点时，可以通过一个启发函数来进行选择，选择代价最少的结点作为下一步搜索的结点。常见的用于求解最短路径的DFS和BFS算法在展开

子结点时，均属于盲目型搜索，也就是说它们不会选择哪个结点在下一次搜索中是更有解，而是直接跳转到某结点进行下一步的搜索。在最坏情形下，需要试探完整的解集空间才可以获得最短路径解。因此，这两种算法只适用于问题规模不大的搜索问题。

与传统的 DFS 和 BFS 不同的是，A^*搜索算法可以通过一个经过仔细设计的启发函数，在很短的时间内得到一个搜索问题的最优解。在社会力模型中，有大量的行人需要计算目标方向，因此采用A^*搜索算法是一个合适的选择。

A^*搜索算法的核心在于如何设计一个好的启发函数。启发函数的表达形式一般如$f(n) = g(n) + h(n)$.其中$g(n)$表示从起点到搜索点的代价，$h(n)$表示从搜索点到目标点的代价。$h(n)$设计的好坏，直接影响到该A^*搜索算法的效率。

A^*搜索算法的流程如下：

1. 为算法设置两个额外的存储空间：OPEN 表和 CLOSE 表。
2. 初始化 OPEN 和 CLOSE 表，将开始结点（开始结点的 H 和 G 值都视为 0）放入 OPEN 表中。
3. 重复下面的步骤：
   - 3.1 在开始列表中查找具有最小 F 值的结点，并把查找到的结点作为当前结点
   - 3.2 把当前结点从 OPEN 列表中删除，并加入到 CLOSE 列表中
   - 3.3 对当前结点相邻的每一个结点依次执行以下步骤：
     - 3.3.1 如果相邻结点不可通行，或者该结点已在 CLOSE 列表中，则什么操作也不执行
     - 3.3.2 如果该相邻结点不在 OPEN 列表中，则将该结点添加到 OPEN 列表中，并将该相邻结点的父结点设置为当前结点，同时计算保存相邻结点的 F 值
     - 3.3.3 如果该相邻结点在 OPEN 表中，则判断若经由当前结点到达该相邻结点的 F 值是否小于原来保存的 F 值，若小于，则将该相邻结点的父结点设为当前结点，并重新设置该相邻结点的 F 值
   - 3.4 若当终点结点被加入到 OPEN 列表作为待检验结点时，表示路径已

找到，此时应终止循环；若当前 OPEN 列表为空，则表示没有从开始结点到终点结点的路径，此时循环结束

4. 循环终止后，从终点结点开始沿着父结点向前遍历，从后向前输出搜索的路径

四、编程实现与调试过程

1. 编程实现

1、 社会力计算模块的接口：

```
class Model:
    def __init__(self, wall_describe, model_map, exit_list, people_list, wall_list, a_star_map_name, thickness,
                 encounter_mode=False, group_bound=0, exit_point=None):
```

wall_describe：障碍物位置的描述

model_map：地图，带有障碍物和墙体

exit_list：出口的点集

people_list：人的位置坐标集

wall_list：墙体的点集

a_star_map_name：调用的提前储存的 a 星地图的名字

thickness：墙体的厚度

encounter_mode=False：是否为人员对流模式，默认为逃生模式

group_bound=0：人员的分界线，讲人群分为对流的两组，只有对流模式下才有意义，默认为以第 0 人为界

exit_point=None：出口的方向，对流模式下才有意义，默认为空

2、 使用 python 的 numpy.array 作为保存墙、障碍物及行人所在坐标的容器，设置三者的坐标，对场景地图进行初始化。

```
def getDes():
    mapX = 50
    mapY = 140
    exitDes = []
    for i in range(5, 45):
        exitDes.append([i, 0])
    for i in range(5, 45):
        exitDes.append([i, 139])
    exitPoint = [[25, 139], [25, 0]]
    peopleDes = []
    lList = []
    rList = []
    for i in range(1, 27):
        with open("data/#10011对流有奖励1/" + str(i) + ".txt") as f:
            fcsv = csv.reader(f, delimiter = ' ')
            r = next(fcsv)
            px, py = changeXY(float(r[0]), float(r[4]))
            if px < 6 or px > 44:
                continue
            if py < 70:
                lList.append([px, py])
            else:
                rList.append([px, py])
    peopleDes = lList + rList
    groupBound = len(lList)

    return mapX, mapY, [], exitDes, peopleDes, True, groupBound, exitPoint
```

使用 map.py、convectionMap.py 文件分别读取疏散场景及对流场景地图信息、测试者初始位置信息。以 convectionMap 文件为例，由于提供的坐标系与本社会力模型模拟的坐标系并不相同，因此设计 changeXY 及 changeToCenter 两个函数，进行坐标转换。主程序通过调用 getDes 函数，可以获得与地图有关的所有信息。该函数返回值意义如下：

- mapX
  - 类型：int
  - 作用：地图 X 方向（第一个坐标分量）最大坐标范围，其中，X 方向为地图的纵向方向。
- mapY
  - 类型：int
  - 作用：地图 Y 方向（第二个坐标分量）最大坐标范围，其中，Y 方向为地图的横向方向。
- deskDes

- 类型：list
- 形状：[n, 4]
- 作用：障碍物列表，其中，n 表示地图中的障碍物总数。对每一个障碍物，有 4 个参数对其进行描述，依次为障碍物 X 方向起始、结束坐标，Y 方向起始、结束坐标。该坐标仅可为整数，且该位置以方格为单位计算，并非以距离为单位（即坐标为 n 的物体占据 n~(n+1)所有空间）。该量输出后，可通过 social_force 中函数转换为可供计算的 np.array 形式，并添加至地图中。

- exitDes
- 类型：list
- 形状：[x, 2]
- 作用：出口列表，里面有 x 个出口的二维坐标，多个坐标组成一个块状出口。主要用于 A 星算法确定方向。该量输出后，可通过 social_force 中函数转换为可供计算的 np.array 形式。

- peopleDes
- 类型：list
- 形状：[x, 2]
- 作用：人的坐标的列表，x 表示人的总数。每个人的位置用两个量描述，分别为人的 X 坐标、Y 坐标。人可以出现在任意位置。该量输出后，可通过 social_force 中函数转换为可供计算的 np.array 形式。

- mode
- 类型：boolean
- 作用：模式表征变量，其中 False 表示该模式为疏散模式，True 表示该模式为对流模式。

- bound
- 类型：int
- 作用：人类型分界线，当 mode 为 True 时，该量为地图两侧的人的分界处。

- exitPoint
- 类型：list
- 形状：[x, 2]

- 作用：出口点列表，返回所有出口的中心点列表，用于辅助 A*计算。

3、 社会力计算模块重要变量定义：

```
self.wall_describe = wall_describe
self.model_map = model_map
self.exit_list = exit_list
self.people_list = people_list
self.wall_list = wall_list
self.thickness = thickness
self.encounter_mode = encounter_mode
self.velocity_list = np.zeros(shape=(len(people_list), 2))
```

- self.wall_describe = wall_describe
- 类型：np.array
- 形状：[x, 4]
- 作用：列表记录障碍物的位置，里面有 x 个 wall，每个 wall，是个四维数组，分别代表障碍物的起始纵坐标，结束纵坐标，起始横坐标，结束横坐标。

- self.model_map = model_map
- 类型：np.array
- 形状：[a,b]
- 作用：地图，a*b 的矩阵地图，单位为分米，对应位置的点的数字代表可通过性。例如，model_map[1][1]==1，代表这个点为墙或者障碍物，不可通过，反之代表可以通过的自由区域。

- self.exit_list = exit_list
- 类型：np.array
- 形状：[x, 2]
- 作用：出口列表，里面有 x 个出口的二维坐标，多个坐标组成一个块状出口。主要用于 A 星算法确定方向。

- self.people_list = people_list
- 类型：np.array
- 形状：[x, 2]
- 类型：dtype=np.double
- 作用：人的坐标的列表，可以定位在非整数的坐标。随着时间增长，如果有人到达出口，队列会减短。用于定位人的坐标。

- self.wall_list = wall_list

- 类型：np.array
- 形状：[x, 2]
- 作用：墙和障碍物列表，里面有 x 个障碍和墙的二维坐标，储存了所有的 model_map 上的值为 1 的坐标，用于 A 星算法确定方向，以及人墙力量的计算。

- self.thickness = thickness
- 类型：int
- 作用：定义墙的厚度，用于计算人和墙的位置关系从而计算人和墙的作用力。

- self.encounter_mode = encounter_mode
- 类型：boolean
- 作用：用于决定所进行的过程是对流情况还是逃生情况，当值为 True 时，为对流，反之为逃生。

- self.velocity_list = np.zeros(shape=(len(people_list), 2))
- 类型：np.array
- 形状：[x, 2]
- 类型：dtype=np.double
- 作用：速度向量列表，与 people_list 一一对应，代表对应位置的人的速度。随着时间增长，如果有人到达出口，队列会减短。同时会根据加速度进行调整。

- self.map_height, self.map_width = self.model_map.shape
- 类型：int, int
- 作用：定义 map 的长和宽，分别是位置坐标第一个值得上限和第二个值得坐标上限。

4、 社会力计算模块重要常量定义：

```
self.const_number = 10
self.velocity_i_0 = 0.8 * self.const_number  # units of measurement: dm/s
self.A_i = 2000  # units of measurement: N
self.B_i = 0.08  # units of measurement: m
self.k = 1.2 * 10 ** 5  # units of measurement: kg(s**-2)
self.k_body_effect_coefficient = 2.4 * 10 ** 5 / self.const_number  # units of measurement: kg(dm**-1)(s**-1)
self.radius = 0.2 * self.const_number  # units of measurement: dm
self.radius_wall = 0.05 * self.const_number  # units of measurement: dm
self.t_gap = 0.05  # units of measurement: s
self.mass = 80  # units of measurement: kg
```

- self.const_number = 10

- 含义：定义倍率，我们的模型使用的是分米，为以正常量纲进行计算对一下常量和变量需要进行转换。
- 量纲：无

- self.velocity_i_0 = 0.8 * self.const_number
- 含义：每个人期望速度的模。
- 量纲：dm/s

- self.A_i = 2000
- 量纲：N

- self.B_i = 0.08
- 量纲：m

- self.k = 1.2 * 10 ** 5
- 量纲：kg(s**-2)

- self.k_body_effect_coefficient = 2.4 * 10 ** 5 / self.const_number
- 含义：人体影响系数。
- 量纲：kg(dm**-1)(s**-1)

- self.radius = 0.2 * self.const_number
- 量纲：dm

- self.radius_wall = 0.05 * self.const_number
- 含义：墙的半径。
- 量纲：dm

- self.t_gap = 0.05
- 含义：时间间隔。
- 量纲：s

- self.mass = 80
- 含义：人的质量。
- 量纲：kg

5、 定义各种辅助函数，如计算两个行人间距离的函数、计算人和墙间距离的函数，以及用于计算合力的函数。

计算两个行人间以及人和墙间距离的函数：

所使用的距离是欧拉距离。

```
def distance(people1, people2):
    return math.sqrt((people1[0] - people2[0]) ** 2 + (people1[1] - people2[1]) ** 2)
```

获得当前位置到四面墙的距离的函数：

```
current = self.people_list[location]
walls = []
for i in range(-1, 2):
    for j in range(-1, 2):
        if abs(i + j) == 1:
            target = np.ones(shape=2, dtype=np.int32)
            target[0] = math.floor(
                (current[0]) if i == 0 else self.thickness - 1 if i == -1 else self.map_height - self.thickness)
            target[1] = math.floor(
                (current[1]) if j == 0 else self.thickness - 1 if j == -1 else self.map_width - self.thickness)
            if self.model_map[target[0]][target[1]] == 1:
                walls.append(target)
```

- 注：location 是人索引，代表第几个人

获取每个障碍物距离人最近的点。

```
for w in self.wall_describe:
    choice = np.ones(shape=2, dtype=np.int32)
    choice[0] = math.floor(w[0] if current[0] <= w[0] else current[0] if current[0] < w[1] else w[1])
    choice[1] = math.floor(w[2] if current[1] <= w[2] else current[1] if current[1] < w[3] else w[3])
    walls.append(choice)
```

- 注：有多个障碍物，分别计算。

6、 三项力的合理的计算公式

```
def accelerate(self, i, e):
    ca1 = self.mass * (  # calculate naturally expected inertia force
            self.velocity_i_0 * e - self.velocity_list[i]) / self.t_gap / self.const_number
    ca2 = [0, 0]
    for j in range(len(self.people_list)):  # calculate the force between people and people
        if i != j:
            ca2 = ca2 + self.force_people_people(i, j)
    ca3 = [0, 0]
    current_wall_list = self.get_wall(i)  # get borders and obstacles
    for w in current_wall_list:  # calculate the force between people and borders and obstacles
        c = self.force_people_wall(i, w)
        ca3 = ca3 + c
    return (ca1 + ca2 + ca3) / self.mass
```

7、 编写行人之间相互作用力的函数

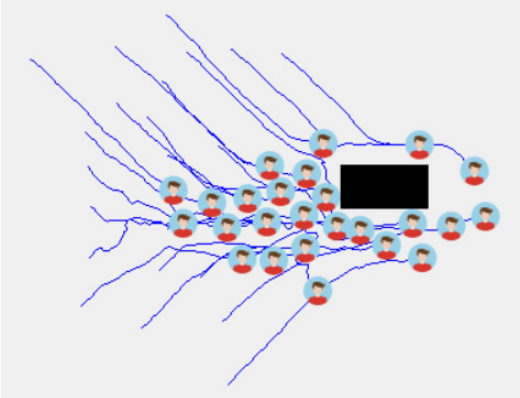
```
def force_people_people(self, i, j):
    r_ij = (self.radius + self.radius) / self.const_number
    d_ij = distance(self.people_list[i], self.people_list[j]) / self.const_number
    ca1 = self.A_i * math.exp((r_ij - d_ij) / self.B_i)
    g = 0
    ca2 = self.k * g
    n_ij = (self.people_list[i] - self.people_list[j]) / self.const_number / d_ij
    ca3 = (ca1 + ca2) * n_ij
    t_ij = [-n_ij[1], n_ij[0]]
    delta_v_ji = (self.velocity_list[j] - self.velocity_list[i]) * t_ij
    ca4 = self.k_body_effect_coefficient * g * delta_v_ji * t_ij
    return ca3 + ca4
```

8、 编写行人与墙之间相互作用力的函数

```
def force_people_wall(self, i, w):
    r_iw = (self.radius + self.radius_wall) / self.const_number
    d_iw = distance(self.people_list[i], w) / self.const_number
    ca1 = self.A_i * math.exp((r_iw - d_iw) / self.B_i)
    g = 0
    ca2 = self.k * g
    n_iw = (self.people_list[i] - w) / self.const_number / d_iw
    ca3 = (ca1 + ca2) * n_iw
    t_iw = [-n_iw[1], n_iw[0]]
    delta_v_wi = (0 - self.velocity_list[i]) * t_iw
    ca4 = self.k_body_effect_coefficient * g * delta_v_wi * (
            self.velocity_list[i][0] * t_iw[0] + self.velocity_list[i][1] * t_iw[1]) * t_iw
    return ca3 + ca4
```

9、 编写A^*搜索算法

a) 我们使用的算法考虑了走斜线的情况，如图，计算了斜线情况下的 H 和 G。



b) A_star 公式为 F = G + H

```
self.G = father.G + 1 * math.sqrt((x - self.father.x) ** 2 + (y - self.father.y) ** 2)
self.H = distance(self, end)
self.F = self.G + self.H
```

c) 使用曼哈顿距离作为 H(n) = D * (abs (cur.x - end.x) + abs (cur.x - end.x))

```
def distance(cur, end):
    return abs(cur.x - end.x) + abs(cur.y - end.y)
```

d) G = father.G + 1 * math.sqrt((x - self.father.x) ** 2 + (y - self.father.y) ** 2)

10、使用 PyQt5 编写 GUI.

利用 PyQt5 中 QLabel、QPushButton、QTimer、QPainter 等基本组件构建 GUI。GUI 核心功能为：显示当前时刻人的位置，显示当前模拟时间，并提供按钮控制单步或连续进行模拟。GUI 中还提供记录、显示人移动轨迹的功能。本 GUI 可以适应各种比例尺的地图，并根据地图大小对人、墙、桌子等物体进行自动缩放。核心代码实现如下：

```
def setMap():
    self.mapSize = self.modelMap.shape
    self.sizePerPoint = min(500 / self.mapSize[0], 500 / self.mapSize[1])
    self.paintX0 = (500 - self.sizePerPoint * self.mapSize[1]) / 2 + self.startX0
    self.paintY0 = (500 - self.sizePerPoint * self.mapSize[0]) / 2 + self.startY0
```

地图适应代码，计算地图的显示比例尺

```
def initPeople():
    self.peopleList = []
    image = QPixmap()
    image.load("1.png")
    image2 = QPixmap()
    image2.load("2.png")
    r = self.peopleRadius * self.sizePerPoint
    for i in range(len(people_list)):
        p = people_list[i]
        label = QLabel(self)
        px = (p[1] - self.peopleRadius) * self.sizePerPoint + self.paintX0
        py = (p[0] - self.peopleRadius) * self.sizePerPoint + self.paintY0
        label.setGeometry(px, py, self.sizePerPoint * self.peopleRadius * 2,
                          self.sizePerPoint * self.peopleRadius * 2)
        if i < bound:
            label.setPixmap(image2)
            pType = 0
        else:
            label.setPixmap(image)
            pType = 1
        label.setScaledContents(True)
        path = QPolygonF()
        path << QPointF(px + r, py + r)
        self.peopleList.append([label, path, pType])
    self.arriveList = []
```

初始化所有测试者代码

```
def drawWall(painter):
    def oneWall(x, y, painter):
        dx = x * self.sizePerPoint + self.paintX0
        dy = y * self.sizePerPoint + self.paintY0
        painter.setPen(Qt.black)
        painter.setBrush(Qt.black)
        painter.drawRect(dx, dy, self.sizePerPoint, self.sizePerPoint)

    for i in range(self.mapSize[0]):
        for j in range(self.mapSize[1]):
            if self.modelMap[i][j] == 1:
                oneWall(j, i, painter)
```

绘制障碍物代码

```
def updateModel(self):
    self.time += self.timeInterval
    self.timeLabel.setText('%.3f' % (self.time / 1000))
    pList, apList, aNum = self.model.update()
    r = self.peopleRadius * self.sizePerPoint
    for i in range(len(aNum)-1, -1, -1):
        ap = self.peopleList.pop(aNum[i])
        apx = (apList[i][1] - self.peopleRadius) * self.sizePerPoint + self.paintX0
        apy = (apList[i][0] - self.peopleRadius) * self.sizePerPoint + self.paintY0
        ap[0].move(apx, apy)
        ap[1] << QPointF(apx + r, apy + r)
        self.arriveList.append(ap)
    for i in range(len(pList)):
        px = (pList[i][1] - self.peopleRadius) * self.sizePerPoint + self.paintX0
        py = (pList[i][0] - self.peopleRadius) * self.sizePerPoint + self.paintY0
        self.peopleList[i][0].move(px, py)
        self.peopleList[i][1] << QPointF(px + r, py + r)
    if self.peopleList != [] and self.runState:
        self.timer.start(self.timerInterval)
    else:
        self.timer.stop()
```

位置更新代码，由 QTimer 调用

```
def drawPath(painter):
    painter.setBrush(False)
    for pInf in self.peopleList:
        path = QPainterPath()
        p = pInf[1]
        path.addPolygon(p)
        if pInf[2] == 1:
            painter.setPen(QPen(Qt.blue, 1))
        else:
            painter.setPen(QPen(Qt.red, 1))
        painter.drawPath(path)
    for pInf in self.arriveList:
        path = QPainterPath()
        p = pInf[1]
        path.addPolygon(p)
        if pInf[2] == 1:
            painter.setPen(QPen(Qt.blue, 1))
        else:
            painter.setPen(QPen(Qt.red, 1))
        painter.drawPath(path)
```

路径绘制代码

```
def step(self):
    if self.runState != 1:
        self.timer.start(self.timerInterval)

def run(self):
    sender = self.sender()
    if sender.text() == "RUN":
        sender.setText("PAUSE")
        self.runState = 1
        self.timer.start(self.timerInterval)
    else:
        sender.setText("RUN")
        self.runState = 0
        self.timer.stop()
```

单步调试、连续调试按钮设置代码

11、根据社会力模型的公式，将以上函数和算法计算出来的结果连接，GUI 不断调用社会里计算模块提供的接口函数算出行人在下一时刻所处的位置，在画面上渲染出来。

2. 调试过程

1）无 GUI 调试

调试过程中遇到的第一个问题是在 GUI 尚未编写完成的情况下该如何验证社会力计算的正确性。我们在实验中采用的方法是，先用单人进行测试，输出每一步社会里计算之后人的位置坐标、速度和加速度，观察有没有超出预计范围的情况。测试通过以后，设置两到三个人进行社会力的测试，通过同样的方法输出并检查每一个人在每一个仿真时刻的信息。

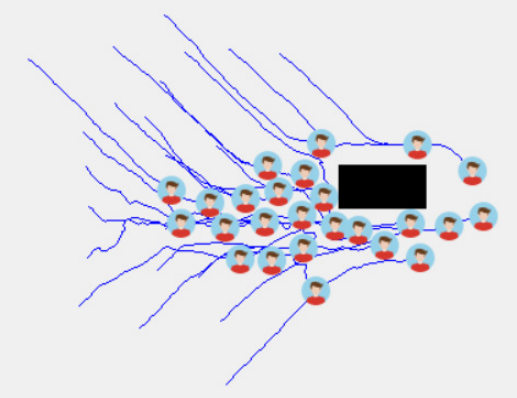
2）人与人间力过大问题

在一开始的版本中，我们遇到了行人在距离其他行人过近时会被弹开，以及某个行人在两个行人之间反复弹跳的问题。这种情况在行人初始位置过于密集的时候尤为明显。这是由社会力模型的缺陷导致的，如绪言中所述，两个行人之间若有 5cm 的重叠，将会产生 6000N 的压力。这么大的力，即使不考虑人的承受能力，也会使人产生巨大的加速度以及随后产生的速度跳变、位置跳变等一系列不符合实际的现象。为了解决这个问题，我们采取了两个措施：一是限制行人初始位置的密集程度，使仿真刚开始时行人之间不会发生重叠；二是限制行人的最大速度及最大加速度，使其不会受到过大的社会力。这两种调整都是符合实际的，

在模型仿真时取得了很好的效果，有效消除了原本存在的跳变问题。

3）在 a 星计算路径时

- 我们首先考虑的是曼哈顿距离，但是曼哈顿距离会使路径趋向于先走斜线，如图：



- 我们考虑了使用欧几里得距离 h(n) = D * sqrt((cur.x-end.x)^2 + (cur.y-end.y)^2)，效果并没有好转，路径计算速度也慢了。
- 我们还考虑了修改为对角线距离的统计和计算，如下图，但是实验效果表明这种情况虽然路径更合理，但是最终的逃生模拟中，会出现在墙角发生卡顿的情况，效果并不好，我们最终还是选择了曼哈顿距离。

```
def distance(cur, end):
    h_diagonal = min(abs(cur.x - end.x), abs(cur.y - end.y))
    h_straight = (abs(cur.x - end.x) + abs(cur.y - end.y))
    d = 1
    d_2 = math.sqrt(2)
    return d_2 * h_diagonal + d * (h_straight - 2 * h_diagonal)
```

4）障碍物力计算问题

社会力模型中，计算障碍物对人的作用力时，由于障碍物是以一块面积的形式存在的，因此存在多种计算障碍物对人合力的方式。

我们最初将障碍物视为一个实心物体，对障碍物内部的每一个网格点依次计算该格点所在位置对行人的作用力，最后将所有网格点的作用力按向量加法进行叠加，算出对行人的合力。然而这种问题存在一定的缺陷，在图形化仿真的过程中，我们发现人在和障碍物还有一段距离时就不再继续向障碍物靠近了，即使人群很密集时依然如此。经过我们的分析，我们认为这是由于如果将障碍物看作一个实心实体，依据原始社会力模型的公式，将会对行人产生一个巨大的合力，因

为障碍物中的每一个点都作为了合力的贡献者，即使行人其实并不能感知到障碍物内部的情况。因此我们接下来换了一种方式。

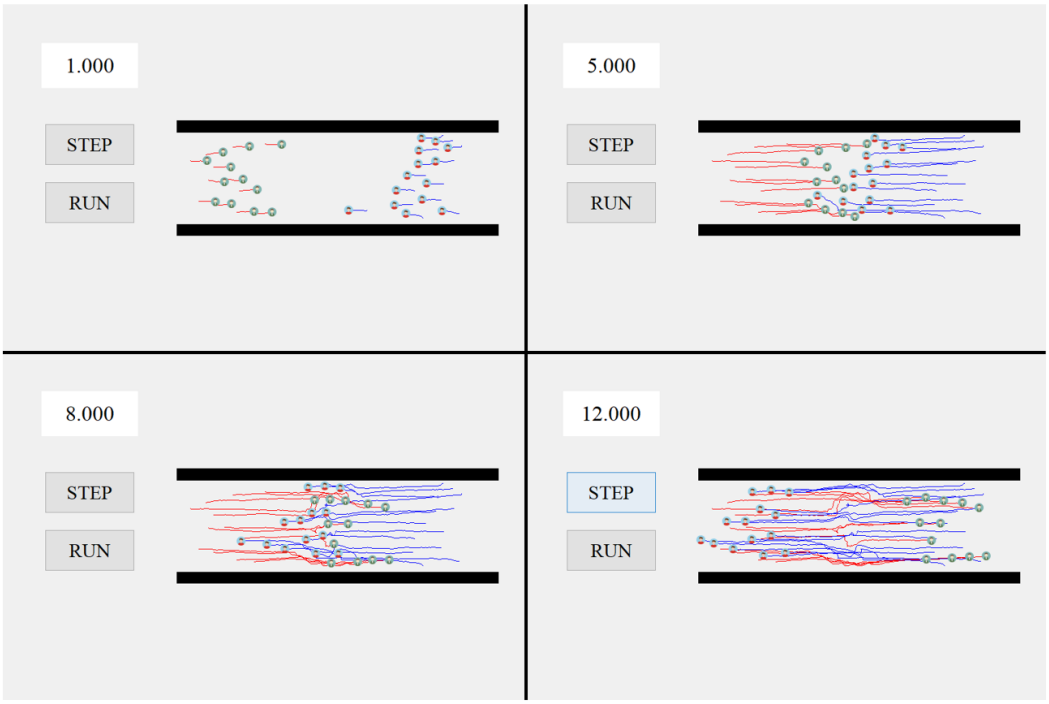
在第二次尝试中，我们不再将障碍物看作实心物体，而是看作空心物体，即只有边界会对行人产生作用力。我们对障碍物的四个边界上的所有格点依次按照社会力模型的公式计算其对人的作用力，并将其叠加成合力。这种方法运行结果较第一次尝试好了许多，但仍然存在行人在距墙还有一定距离时便不再前进的问题。究其原因，墙或障碍物（以下简称墙）对行人的力，归根结底只是一种心理力而非物理力，即行人在还没有触碰到墙的时候，就会受到墙对他的心理力。在此过程中，人是无法感知到墙的具体物理属性的，因此墙的实际厚度和行人看不到的部分对心理力并无影响。想明白这个问题之后，我们对计算墙和人之间距离的方式进行了第二次修改。

在现在的版本中，我们将墙和障碍物分开处理，对于墙而言，直接计算四个垂直距离，并只按四个垂点计算墙对行人的心理力。对与障碍物而言，我们将人到障碍物之间的距离视为人到障碍物的最近距离，并只按最近距离的那个点计算障碍物对人的心理力。具体的计算方式在上文第三节“数学模型”第四小节“行人与障碍物之间的距离”已经做了具体的阐释，此处不再赘述。通过使用这种方式，我们获得了很好的仿真效果，一个行人在受到其他行人足够大的力时，将有可能贴到墙或障碍物上，很好地模拟了真实情况。

五、程序运行结果分析

1. 对流场景模拟

本实验的第一个场景选择了人群的对流。如下图位于左上角的场景图所示，在对流场景中，地图两侧分别有目标处于相反位置的两群行人，他们的位置分布较为平均，在各自的前进方向上，都有对面的行人挡住去路。本场景的模拟目标为，使用社会力模型对两侧行人群的运动情况进行模拟。预期情形为，两侧的行人群分别到达对侧，且运动过程中会相互避让，体现出行人之间的心理力。



对流场景的四个典型时刻

上图显示了对流场景的四个典型时刻。本实验中，时间粒度为 0.05s，四个典型时刻从左至右、从上至下依次为 1.000s，5.000s，8.000s 和 12.000s.

在 1.000s 时，行人刚刚开始运动，此时运动的方向与目标出口方向一致，只是由于本侧其他行人的社会力影响而稍有垂直方向的偏移。

5.000s 时，两侧行人开始出现交会的情况，此时已有红线行人（即出发点在地图左侧的行人）和蓝线行人（即出发点在地图右侧的行人）距离很近的情形。距离相近的行人立刻会彼此产生排斥的作用力，使得其位移方向在竖直方向上产生一个很大的偏移。

8.000s 时，交会处于最激烈的时刻，由图上可见，许多行人的位移曲线已经发生了垂直方向长距离的偏移，这是为了避开与之相邻的行人。在此过程中，碰撞并没有发生，行人都按照社会力的模拟，在很好地避开本侧和对侧的行人的同

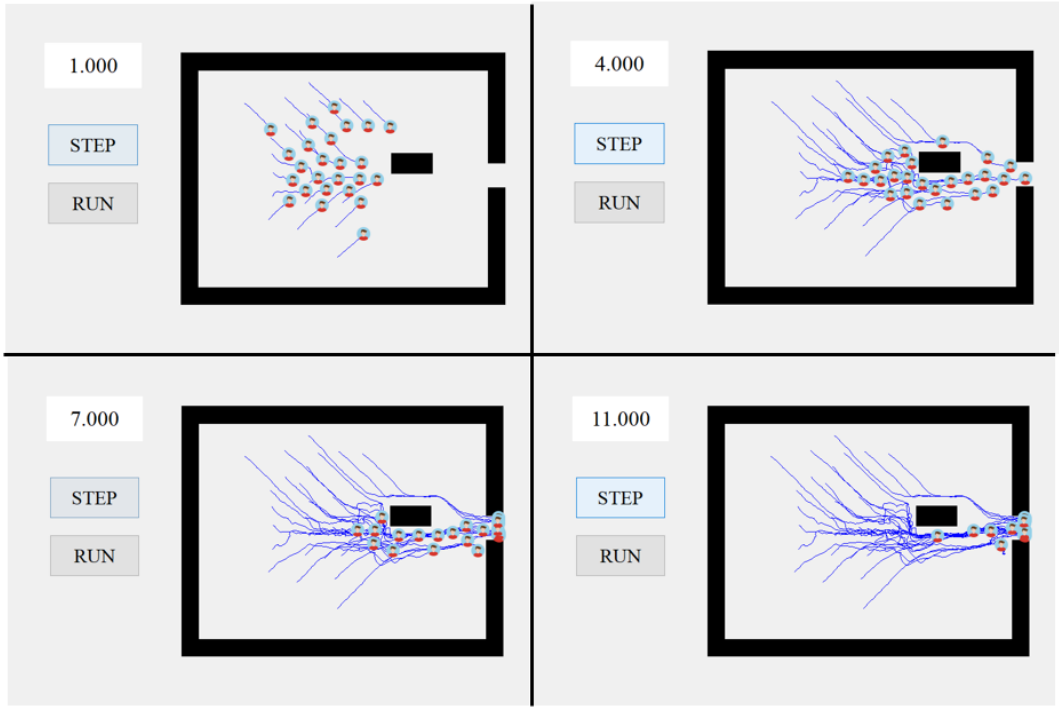
时，向自己的目标移动。

12.000s 时，交会已经基本完成，两侧的行人不再有交集，路线方向恢复为水平，各自向出口移去。在社会力模型中，当行人相距较远时，目标向量是位移方向的主导因素，行人之间的心理力不再起决定性作用。这个时刻的场景很好地模拟了这一点。

在对流场景中，本实验实现的社会力模型表现出了很优秀的运行结果，与我们预期的结果基本一致，最终两侧的行人都在没有接触和碰撞的情况下走到了对面。

2. 人群疏散场景模拟

实验的第二个场景选择了人群的疏散。如下图所示，地图中有一个长条状的出口，在出口的左侧有一个方形的障碍物阻挡行人的去路。行人需要避开障碍物、最终从出口离开房间。本场景的模拟目标为，使用社会力模型对处于一个单开口房间中的行人群进行模拟。预期现象为，行人先向出口方向移动，在离障碍物较近时会避开障碍物，同时不与其他行人发生碰撞，最终全部从出口离开。



疏散场景的四个典型时刻

上图显示了疏散场景的四个典型时刻。本实验中，时间粒度为 0.05s，四个

典型时刻从左至右、从上至下依次为 1.000s，4.000s，7.000s 和 11.000s.

在 1.000s 时，行人刚刚开始运动，由于大部分行人距离障碍物都还有一段距离，因此位移方向都朝向出口处，并没有因障碍物的存在而改变自己的方向。

4.000s 时，行人均已贴近障碍物，由于障碍物对人的作用力大于行人之间的作用力，因此即使行人之间互相排斥，障碍物依然将靠近的行人推向其他行人，使行人聚在一起。由于所有行人的目标都是地图中唯一的出口，因此障碍物下方的行人都在从左至右水平移动。障碍物上方的行人之所以没有选择从障碍物下方走，是因为障碍物下方的行人过多，对其造成的心理力合力大于目标出口对其的吸引力，使其无法继续向下前进。障碍物上方的行人先是向右走，离开障碍物范围以后再继续向下向出口移动。

7.000s 时，已有一部分行人从出口离开，剩下的行人依然有条不紊地从障碍物下方依次通过。由于目标出口的吸引力，行人不会离开地图中部太远，活动范围都仅限于障碍物下方的一小段距离。

11.000s 时，仿真基本完成，大部分行人已从出口离开。分析行人的轨迹可以看出，行人全都避开了障碍物，并始终以出口为移动的最终方向。障碍物左下角的一处杂乱的痕迹说明在那个地方行人较为密集，行人之间的排斥力较强，使得行人的加速度方向不断改变。

疏散场景中，本实验实现的模型也达成了预期目标。

六、结论

本实验通过编程实现社会力模型，实现了对人群对流和疏散场景的图形化仿真，并得到了很好的仿真结果。实验结果验证了社会力模型的诸多优点，也表明社会力模型在模拟人群运动时确实能起到很好的效果。对于社会力模型中存在的一些不足，我们对其进行了补足和修正，其结果在仿真模拟中有了正面的体现。

第二部分 神经网络实验报告

一、绪言

在第一部分绪言中提到，有多种常用的模型可以解决人群疏散问题，然而无论是宏观模型还是微观模型，都需要先建立好一个完善的模型，对系统的宏观状态和微观状态有足够详细的了解后，才可以应用于实际模拟当中。也即，模型中的每一个参数都需要被设置得足够准确，而这很大程度上有赖于模型设计者和模型实现者对待模拟系统的考察。这种考察并非无论何时都能进行得很全面，当系统规模变得很大，例如需要模拟大规模人群疏散时，由于人的性格具有多样性，做统一处理有失准确性，然而又很难为每一个行人都单独设置其自己的特征和属性。这最终导致宏观模型和微观模型被应用于大规模模拟时会出现很大的偏差，且模拟所需要的计算量和时间都非常巨大。

为了解决传统模型中存在的这些问题，需要引入新的技术手段。近年来人工神经网络（ANN）的兴起为我们提出了解决问题的思路。当我们拥有的数据集足够大时，我们就可以通过这些数据训练一个神经网络去学习真实行人的行为，而不是人为地为每一个行人构造参数。对于每一个行人而言，影响其运动的主要有三个因素：其自身向目标处移动的意愿、周围行人对他的影响、以及障碍物对他的影响。通过从高分辨率影像视频中提取行人在各种场景下的实际行为，我们就可以构造一个基于真实数据的行人数据集，用于搭建一个可以模拟行人在各种场景下的运动情况的神经网络。

使用神经网络解决行人运动的模拟问题，不仅解决了现有的模型模拟精度不高的问题，还使得模拟不再需要模拟者手工设置参数，仅从现有的数据集中就可以导出行人的运动方式。基于这样的理论指导，我们搭建了一个四层的人工神经网络，使用已有的数据对神经网络进行了训练，最终得到了一个可以很好模拟人群疏散的模型。

二、实验目的

搭建人工神经网络（ANN），通过处理给定数据集并输入给神经网络，训练出一个可以模拟人群行为的模型；通过 GUI，将人群的运动过程直观地展现出来，

验证神经网络模拟人群运动的准确性。

三、神经网络模型

1. 神经网络的设计

我们搭建的神经网络具有四层的架构，输入层有 22 个神经元，输出层有 2 个神经元，中间还有两个隐层，每个隐层各有 110 个神经元。增加隐层数可以降低神经网络的误差、提高精度，但也会使网络复杂化，有可能出现过拟合倾向。由于本实验的数据集并不是很大，且对模拟精度有一定要求，因此我们选择了四层神经网络，以取网络复杂程度和精度的折中。

2. 神经网络输入设计

为了能让神经网络得到合理有效的输入，我们对原始数据进行了预处理。处理之后的数据每一行代表一个行人的状态信息，一行含有 22 个域，对应了神经网络输入层的 22 个神经元。这些域的含义如下：

编号	含义	单位
1	当前速度	m/s
2	A^*与单位向量(1,0)之间的角度	rad
3-7	W[1-5]，与最近的五个行人之间的水平相对距离	m
8-12	W[6-10]，与最近的五个行人之间的垂直相对距离	m
13-17	W[11-15]，与最近的五个行人之间的水平相对速度	m/s
18-22	W[16-20]，与最近的五个行人之间的垂直相对速度	m/s

其中 W[1-10]是将该行人与最近的五个行人之间的距离归一化到了[0,1]之后的结果，具体的归一化方法如下面两个公式所述：

$$x_i \geq x, w_i = \frac{x_i - x}{3} = \frac{\Delta x_i}{3};\ x_i < x, w_i = -\left(1 + \frac{\Delta x_i}{3}\right), 1 \leq \mathrm{i} \leq 5$$

$$y_i \geq y, w_j = 1 - \frac{\Delta y_i}{3};\ y_i < y, w_j = -\left(1 + \frac{\Delta y_i}{3}\right), 6 \leq \mathrm{j} \leq 10$$

W[11-20]的计算方法如下面两个公式所述：

$$w_k = \Delta V_{x_i} = V_{x_i} - V_x, 11 \leq k \leq 15$$

$$w_l = \Delta V_{y_i} = V_{y_i} - V_y, 16 \leq l \leq 20$$

3. 神经网络输出设计

神经网络的训练目标是，在使用预处理好的数据集训练过后，能够从当前时刻的速度和状态，导出下一时刻的速度。因此我们的神经网络的输出具有两个域，这些域的含义如下：

编号	含义	单位
1	行人在下一时刻水平方向的速度	m/s
2	行人在下一时刻垂直方向的速度	m/s

4. A^*算法

在数据的预处理部分，我们需要将神经网络输入部分的第二个域设置为A^*与单位向量(1,0)之间的角度，这里需要用到A^*搜索算法，用以求得行人在某一时刻的目标方向。

该A^*算法与社会力模型中的算法相同，因此不再赘述，详见本文第 4-6 页 A*算法章节相关内容。

四、神经网络搭建

我们使用了 python 的深度学习库 pytorch 进行神经网络的搭建

构建四层 ANN

```
def __init__(self, in_dim, n_hidden_1, n_hidden_2, out_dim):
    super(simpleNet, self).__init__()
    self.in_dim = in_dim
    self.layer1 = nn.Sequential(nn.Linear(in_dim, n_hidden_1), nn.BatchNorm1d(n_hidden_1), nn.ReLU(True))
    self.layer2 = nn.Sequential(nn.Linear(n_hidden_1, n_hidden_2), nn.BatchNorm1d(n_hidden_2), nn.ReLU(True))
    self.layer3 = nn.Linear(n_hidden_2, out_dim)
```

向前传播函数

```
def forward(self, x):
    x = self.layer1(x)
    x = self.layer2(x)
    x = self.layer3(x)
    return x
```

将处理好的数据转换为张量

```
Totel_list_train_data = np.loadtxt("dataset/train_data.csv", delimiter=",")

train_x = np.array(Totel_list_train_data, dtype=np.float32)[:, 0:22]
train_y = np.array(Totel_list_train_data, dtype=np.float32)[:, 22:24]

x_train = torch.from_numpy(train_x)
y_train = torch.from_numpy(train_y)
```

网络结构同设计部分所述，在此使用均方误差作为误差评判标准，使用随机最速下降法作为优化方法

```
model = ANNNet(22, 110, 110, 2)
criterion = nn.MSELoss()
optimizer = optim.SGD(module.parameters(), lr=1e-3)
```

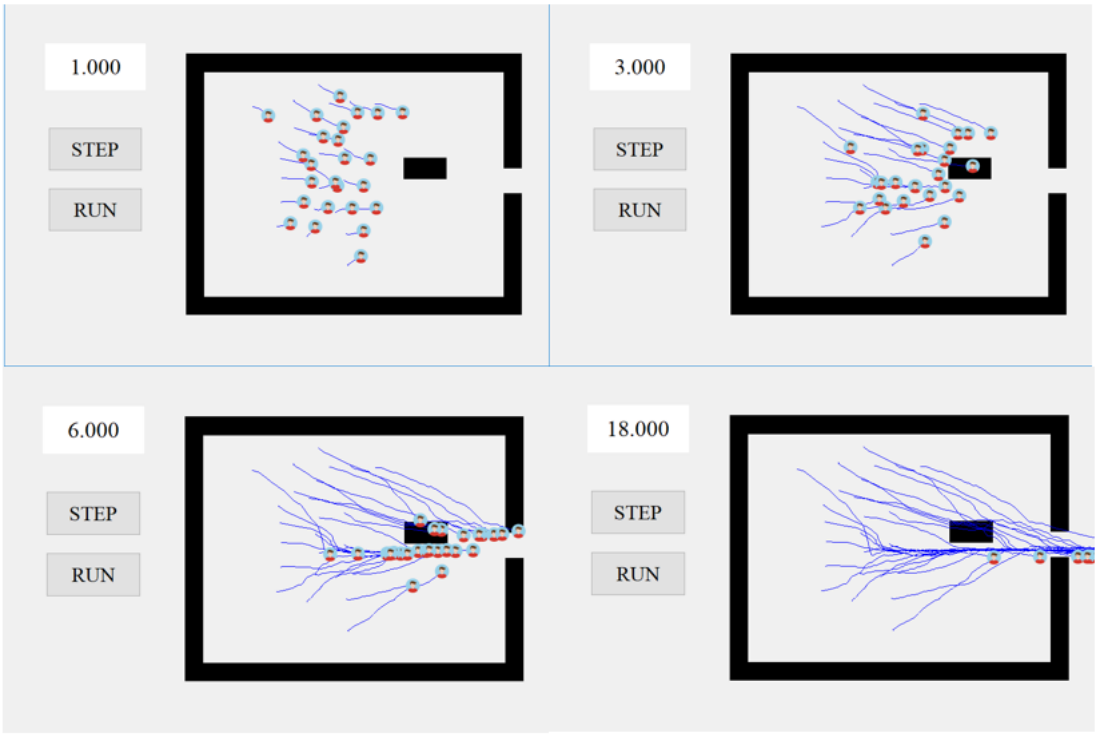
对所有数据循环学习五万次，在结束后保存模型

```
num_epoch = 50000
for epoch in tqdm(range(num_epoch)):
    input1 = Variable(x_train)
    target = Variable(y_train)
    out = module(input1)
    loss = criterion(out, target)

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

五、神经网络实验

我们用搭建好的神经网络，对人群疏散的场景进行了模拟。如下图所示，我们选取的场景在右侧的墙上有一个宽度可容纳两名行人的出口，以及一个在出口左侧、阻碍行人顺利疏散的障碍物。本场景的模拟目标为，利用训练好的神经网络，模拟人群从出口处疏散的场景。预期现象为：行人先向出口方向移动，尽量避开障碍物，最终全部从出口离开。



由以上疏散过程的典型场景可以看出，从已有数据集训练的神经网络可以使行人顺利向出口方向移动。大部分行人都有避开障碍物的趋向，且一部分行人成功避开了障碍物。全部行人的最终目标都位于出口处，因此没有行人的运动轨迹在竖直方向上产生过大的偏移，都逐步向地图中间靠近，符合实验的预期。最后全部行人都通过出口顺利离开，完成疏散过程。

六、神经网络相比社会力模型的优缺点

1. 神经网络的优势

神经网络与社会力模型都可用于模拟行人的运动场景，本实验中我们搭建的神经网络获得了与社会力模型模拟相近的效果。从模拟的前期准备上来讲，相比社会力模型，神经网络的方法具有诸多优点：

1）不需要构造严谨的数学模型，仅需要合理的数据集即可进行训练

在社会力模型的实验中，我们花费了大量时间研究社会力模型的数学公式，并用了很大篇幅的代码去实现该数学公式。由于社会力模型的公式环环相扣，任何一个计算函数出现问题，都会导致最终模拟出来的结果与预想值有很大偏差。社会力模型的正确运行很依赖于实现的正确性及各种概念定义的准确性，如一开

始我们没有定义好障碍物道人的距离，构造出的模型就不尽如人意。但神经网络不需要这些麻烦的操作，只要我们将原始数据处理成适合神经网络学习的数据，且数据集真实可靠，不需要进行数学分析即可开始训练。而且，随着数据集的增大，神经网络方法的可靠性也不断增强，但模拟的前期准备并不会变得更加复杂。

2）神经网络不需要专家设计参数

在社会力模型中，有许多参数是被模型的设计者预先设置好的，如行人之间的摩擦系数和弹力系数。这些参数需要对物理实体进行仔细的考察才能确定，而且随着环境的变化也会有所改变。当模拟复杂度增高或场景发生变化时，这些参数就不再适用，需要进行重新调整。对于模型的实现者而言，针对每一个场景都去实地考察参数的设置显然是不可能的，因此只能凭经验和直觉在原有的基础上做微调，这极大地降低了社会力模型的准确性和适用范围。神经网络则不需要实现者参与对物理世界的考察，因此神经网络和社会力模型相比，具有更广的适用范围。

3）在大规模模拟时，神经网络在模拟过程中消耗的计算量更少

社会力模型的工作原理意味着在模拟时必须对每一个个体都进行实时的计算。在人少的场景下，这种计算的成本是可以接受的，但随着模拟规模的增大，当待模拟的行人个数增加到万这个数量级时，社会力模型将消耗巨大的计算量，这对于计算机的资源和计算力都是一个极大的考验。在更大规模的模拟（如更大的地图和更多的行人）中，社会力模型会消耗更多的资源、计算量和时间，这种消耗发生在模拟的运行过程中，这使得实时模拟成为不可能。而神经网络则具有“一次训练，反复使用”的优点，可以在前期构建好数据集以后便开始训练。虽然训练会耗费很长时间，但训练出的模型在模拟运行期间不会占用更多的计算资源。因此，神经网络的方法可以用于大规模实时模拟，这是社会力模型无法做到的。

2. 神经网络的局限

然而，虽然有着以上诸多优点，但神经网络也有相比社会力模型不足的地方。这种不足主要体现在以下两点：

1）神经网络模拟的准确性严重依赖于数据集的准确性

人工神经网络依靠大量的数据进行训练，可以说数据集决定了训练效果的可靠性。如果数据集足够大且数据足够准确，神经网络便可以取得很好的效果。然而，对于行人运动模拟这个课题来说，如何提取到大量准确的运动数据是一个很大的问题。本实验采用的数据集规模并不大，而且是人为构造的一个特定的实验场景，因此用其训练出来的神经网络无法像社会力模型那样精确。只有当数据集足够准确，神经网络的方法才能发挥出很好的效果。

2）神经网络对于一些隐性作用（如人和墙之间的作用等）无法很好的模拟

社会力模型是一个严谨的数学模型，考虑了各种影响人的运动的因素，因此准确性较高，尤其是将一些隐形作用如人和墙之间的作用力囊括在模型之中，进一步提升了模型的准确性，也使得模拟更贴近真实情况。在神经网络的方法中，由于数据集中没有很好地体现行人与墙之间的作用力，因此在训练出来的模型中，这些作用力也没有得到体现。在上文中提到的部分行人无法避开障碍物，正是由于这个原因导致。当然，在数据集足够大且足够准确的情况下这个问题是可以被避免的，但相比社会力模型直接计算出每一个具体的分力，神经网络在这些基本情况的模拟上还存在着不足。

七、结论

本实验搭建的四层人工神经网络基本能够模拟人群的疏散场景，并取得与社会力模型相近的效果。神经网络方法还有一些提升空间，如通过选取带时序的神经网络如 RNN，对行人的运动演化进行时序上的刻画，以取得更好的模拟效果。实验所用的数据集还可以进一步加大，随着数据集的增大，神经网络的训练效果和训练出来的模型的准确也会进一步提升。总而言之，相对于社会力模型，神经网络是另一个方向上的尝试，这种尝试带给了我们新的思路。我们相信，随着计算机技术的进步与基础科学的发展，神经网络的方法将会是未来对大规模人群运动模拟的首选方法。