北京航空航天大學

BEIHANG UNIVERSITY

数学模型

社会力模型的传统实现与神经网络实现

实验报告

北京航空航天大学

计算机学院

陈麒先，卓培锦，牛雅哲，韩笑冰

指导教师：宋晓

二〇一八年十二月

郑重声明

本实验报告由陈麒先完成第一至六章、第八第九章传统社会力模型的编写；由韩笑冰、卓培锦、牛雅哲同学共同完成第七章用神经网络实现社会力模型的编写；最后由陈麒先完成了对全文的审校。本实验报告中参考了文献或互联网资料的部分均有引用标注，另有与老师或同学研究后的成果引用将在特别致谢中说明。抄袭行为在任何情况下都是不能容忍的(COPY is strictly prohibited under any circumstances)！转载或引用须征得作者本人同意，并注明出处！勿谓言之不预！另，实验报告撰写仓促，如有错误，在所难免，欢迎批评指正！特此声明。

陈麒先，卓培锦，牛雅哲，韩笑冰

目录

一、实验目的

近年来发生的多起突发灾害给人民群众带来了巨大的生命和财产损失，如地震、雪灾等自然灾害，又如 2018 年 12 月 16 日北航合一食堂发生的火灾等突发事故灾害。一次又一次的灾难警示我们必须对突发性灾害进行研究，找出其一般规律，尽可能在灾害来临时将损失减少到最小。

这些突发灾害，都有一个共同特点，即受影响人数巨大，都会形成大规模人群行为，所以有必要对大规模人群的行为进行研究。

因此，本次实验基于此目的，实现了一种基于德国交通科学家 HelBing 提出的基本社会力模型，对人群的恐慌行为进行分析，给出在给定场景下的人群疏散仿真模拟结果。

二、数学模型

1、模型研究现状

对于人群行为的模拟，国内外学者已经做了很多研究，德国的 Helbing 等对人群恐慌进行了详细分析[0]，并提出了社会力模型(Social Force Model，SF)，模型考虑了行人流的离散特征，假设行人流的动态特征是在个体相互之间的作用力下产生的;日本的 Teknomo 对社会力模型中的排斥力构建了相关模拟模型;Cremer 和 Ludwig 将元胞自动机模型(Cellular Automata，CA)应用到车辆交通的研究中;Blue 和 Adler 提出了一种用于大型露天场所的行人运动模型，并设计了双向行人行动的元胞自动机模型。[6]国内浙江大学张晋在博士论文中[8]，结合混合交通流的特点，提出一种二维行人过街元胞自动机仿真模型，该模型引入了“停车点”的概念，不仅能够处理人行横道上行人与行人之间的相互冲突问题，而且能够成功处理行人与其他车辆的冲突和避让，从而模拟了行人各种类型的道路穿越行为;而陈涛等则引入相对速度对社会心理力的影响，对社会力模型进行了修正。[6]

国内外对人群行为的模拟研究大致可以分为宏观模型和微观模型两大类。宏观模型把人看作连续流动介质，利用流体力学的成果，但是忽略了个体的作用和个体间的差异，Fruin 于 1971 年首先提出了宏观行人仿真模型，该模型主要研究了行人的一些集聚性特点。宏观模型利用的是流体力学的成果，只能定性描述行人行为，而不能定量描述局部的细节信息。微观模型是基于个体特性的建模，个体行为随着环境发生动态变化，主要有社会力模型、元胞自动机模型、磁场力学模型以及排队网络模型等。

2、实现模型概述

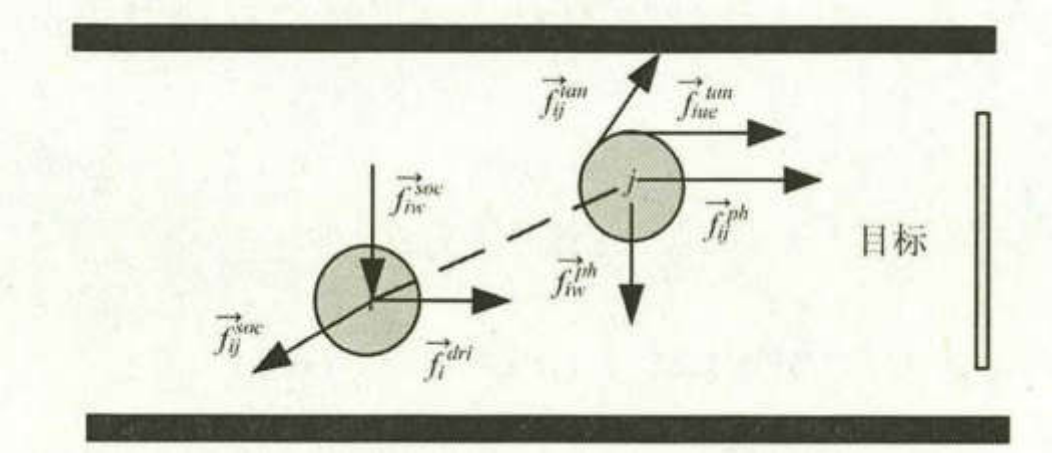
在本实验中，我们选择了微观模型中的社会力模型对大规模人群运动进行分析。

在 Helbing 的初始社会力模型的基础上，对其中提到的部分参数进行修正，从而使最终形成的模型可以更真实地模拟突发事件中大规模人群的运动行为，同时引入了 A* 算法，用于计算最短路径，从而更好地规划了行人的逃离路线。

3、模型实现原理概述

社会力是指一个人受到周围环境的影响，从而引起自身行为的某些改变。社会力模型认为行人的运动是在社会力的作用下发生的，其中包括自驱力、行人之间的

作用力、行人与周边环境之间的作用力。其模型示意图如下图所示。[5]



其中$\vec{f}_i^{dri}$表示行人 i 的驱动力$\vec{f}_{ij}^{soc}$为行人 i 受到前方行人 j 的心理排斥力；$\vec{f}_{ij}^{ph}$为行人 i 与行人 j 之间接触时才会产生的物理排斥力；$\vec{f}_{iw}^{soc}$为行人受到周围障碍物的排斥作用而远离障碍物的心理作用；$\vec{f}_{iw}^{tan}$与$\vec{f}_{iw}^{ph}$分别为行人 i 与障碍物接触时产生的法向与切向的物理作用力。

4、模型实现细节

（1）自驱动力

将行人去往目的地的主观愿望用驱动力来表示，在没有受到环境中其他阻碍的情况下，行人会主观地选择路径最短的方向与最舒适的速度。

公式如下：

$$\vec{f}_i^{dri} = m_i \frac{v_e \vec{e}_i - \vec{v}_i}{T_a}$$

其中 ei 表示行人期望速度方向。其计算公式如下：

$$\vec{e}_i(t) = \frac{\vec{r}_i^k - \vec{r}_i(t)}{\| \vec{r}_i^k - \vec{r}_i(t) \|}$$

（2）行人之间的相互作用力

行人在运动过程中总会尽可能与他人保持一定的距离，若距离过小，则会引起行人心理上的排斥感，这就是常说的“领域效应”，正是因为这种“领域效应”的作用，人与人之间尤其是陌生人之间总会存在无形的排斥力。

行人间相互作用力的计算公式如下：

$$\vec{f}_{ij}^{soc} = A\exp(\frac{r_{ij} - d_{ij}}{B})\left[\lambda_i + (1-\lambda_i)\frac{1+\cos(\varphi_{ij})}{2}\right]\vec{n}_{ij}$$

心理排斥力是当行人 i 进入行人 j 的作用区域后会产生的，远离区域时认为作用为 0，当行人距离靠近时，排斥力的大小是指数形式的增长，当行人发生接触时，除了心理排斥作用，还有物理作用力，包括沿着垂直行人接触面的方向作用，使行人抵抗接触，沿着平行于行人接触面的方向使行人快速分离。法向力和切向力的计算公式分别为：

$$\vec{f}_{ij}^{ph} = (K\Theta(r_{ij} - d_{ij}))\vec{n}_{ij}$$

$$\vec{f}_{ij}^{tan} = k\Theta(r_{ij} - d_{ij})\Delta\vec{v}_{ij}\vec{t}_{ij}$$

其中，

$$\Theta = \begin{cases} r_{ij} - d_{ij}, & r_{ij} - d_{ij} \leqslant 0 \\ 0, & r_{ij} - d_{ij} > 0 \end{cases}$$

在本实验中，我们为了简化模型的实现难度，对上述公式进行了改写，但是该过程并不影响模型实现的正确性。正确性已在数学模型课上由指导教师宋晓老师论证。改写后的公式如下：

$$f_{ij} = \{A_i \exp[(r_{ij} - d_{ij})/B_i] + kg(r_{ij} - d_{ij})\}\boldsymbol{n}_{ij} + \kappa g(r_{ij} - d_{ij})\Delta v_{ji}^{t}\boldsymbol{t}_{ij}$$

其中各项参数初步以论文中所述为准，后续优化过程中我们将对参数进行细微调整以优化模型仿真效果。

（3）行人受障碍物的作用

当行人作用区域范围内有其他障碍物时，行人会自主选择路径来避免与障碍物发生接触或碰撞，行人与障碍物作用原理与行人之间相近。由于障碍物的不可感知，行人的作用具有双向性，而与障碍物的作用是单向的。作用力仍然包括心理排斥力与接触时的物理作用力，但其中参数取值有所变化。

计算公式如下：

$$\vec{f}_{iw}^{soc} = A_w\exp(\frac{r_i - d_{iw}}{B_w})\vec{n}_{iw}$$

当行人速度过大，导致无法闪避障碍物时，则会对障碍物产生挤压，从而产生一个挤压力，计算公式如下：

$$\vec{f}_{ij}^{ph} = (K\Theta(r_i - d_{iw}))\vec{n}_{iw}$$

$$\vec{f}_{iw}^{tan} = k\Theta(r_{ij} - d_{ij})\langle \vec{v}_i \vec{t}_{iw} \rangle \vec{n}_{iw}$$

其中，

$$\Theta = \begin{cases} r_i - d_{iw}, & r_i - d_{iw} \leqslant 0 \\ 0, & r_i - d_{iw} > 0 \end{cases}$$

在本实验中，我们为了简化模型的实现难度，对上述公式进行了改写，但是该过程并不影响模型实现的正确性。正确性已在数学模型课上由指导教师宋晓老师论证。改写后的公式如下：

$$\boldsymbol{f}_{iw} = \{ A_i \exp [(r_i - d_{iw}) / B_i] + kg(r_i - d_{iw}) \} \boldsymbol{n}_{iw} - \kappa g(r_i - d_{iw}) (\boldsymbol{v}_i \cdot \boldsymbol{t}_{iw}) \boldsymbol{t}_{iw}$$

其中各项参数初步以论文中所述为准，后续优化过程中我们将对参数进行细微调整以优化模型仿真效果。

（4）行人合力的计算与运动状态的改变

行人运动合力计算公式如下：

$$m_i \frac{\mathrm{d}\boldsymbol{v}_i}{\mathrm{d}t} = m_i \frac{v_i^0(t)\, \boldsymbol{e}_i^0(t) - \boldsymbol{v}_i(t)}{\tau_i} + \sum_{j(\neq i)} \boldsymbol{f}_{ij} + \sum_{w} \boldsymbol{f}_{iw}$$

行人运动状态的改变可以由物理学基本运动关系表示：

V = v0 + a * t

X = x0 + v0 * t + 0.5 * a * t ^ 2

（5）A*算法

① 算法简介

路径规划是指的是机器人的最优路径规划问题，即依据某个或某些优化准则（如工作代价最小、行走路径最短、行走时间最短等），在工作空间中找到一个从起始状态到目标状态能避开障碍物的最优路径。机器人的路径规划应用场景极丰富，最常见如游

戏中 NPC 及控制角色的位置移动，百度地图等导航问题，小到家庭扫地机器人、无人机大到各公司正争相开拓的无人驾驶汽车等。[9]

② 算法原理

A*（A-Star)算法是一种静态路网中求解最短路径最有效的直接搜索方法，也是许多其他问题的常用启发式算法。注意——是最有效的直接搜索算法，之后涌现了很多预处理算法（如 ALT，CH，HL 等等），在线查询效率是 A*算法的数千甚至上万倍。

公式表示为：f(n)=g(n)+h(n),

其中，f(n)是从初始状态经由状态 n 到目标状态的代价估计，

g(n)是在状态空间中从初始状态到状态 n 的实际代价，

h(n)是从状态 n 到目标状态的最佳路径的估计代价。

（对于路径搜索问题，状态就是图中的节点，代价就是距离）

关于 h(n)的选取：

保证找到最短路径（最优解的）条件，关键在于估价函数 f(n)的选取（或者说 h(n)的选取）。

我们以 d(n)表达状态 n 到目标状态的距离，那么 h(n)的选取大致有如下三种情况：

如果 h(n)< d(n)到目标状态的实际距离，这种情况下，搜索的点数多，搜索范围大，效率低。但能得到最优解。

如果 h(n)=d(n)，即距离估计 h(n)等于最短距离，那么搜索将严格沿着最短路径进行， 此时的搜索效率是最高的。

如果 h(n)>d(n)，搜索的点数少，搜索范围小，效率高，但不能保证得到最优解。[9]

③ 算法描述

1.把起点加入 open list 。

2. 重复如下过程：

a. 遍历 open list ，查找 F 值最小的节点，把它作为当前要处理的节点，然后移到 close list 中

b. 对当前方格的 8 个相邻方格一一进行检查，如果它是不可抵达的或者它在 close list 中，忽略它。否则，做如下操作：

- 如果它不在 open list 中，把它加入 open list，并且把当前方格设置为它的父节点。
- 如果它已经在 open list 中，检查这条路径（即经由当前方格到达它那里）是否更近。如果更近，把它的父亲设置为当前方格，并重新计算它的 G 和 F 值。如果你的 open list 是按 F 值排序的话，改变后你可能需要重新排序。

c. 遇到下面情况停止搜索：

- 把终点加入到了 open list 中，此时路径已经找到了，或者
- 查找终点失败，并且 open list 是空的，此时没有路径。

3. 从终点开始，每个方格沿着父节点移动直至起点，形成路径。[10]

三、实验开发环境与相关技术说明

(1)【操作系统】

Windows

Linux Ubuntu

(2)【集成开发环境】

Anaconda 3 (64 bit)

(3)【系统】

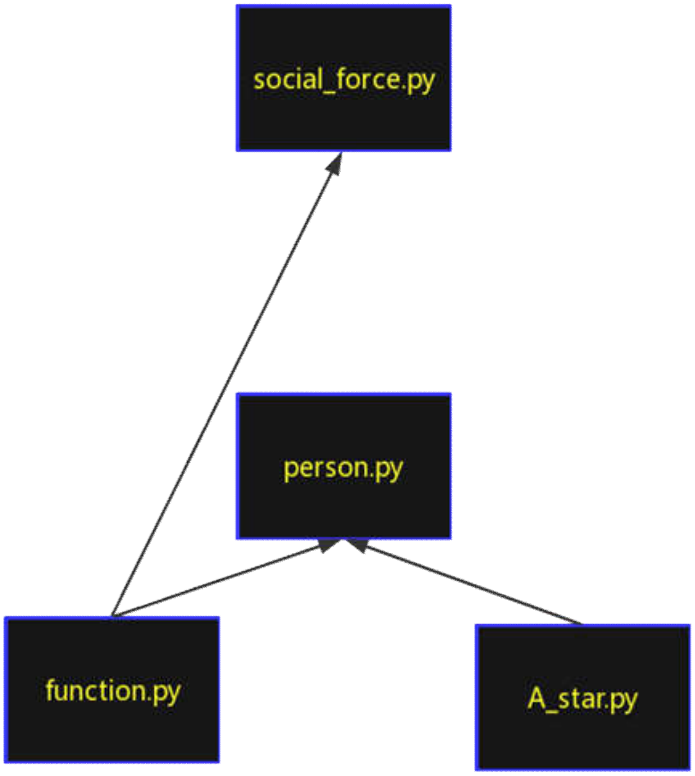
处理器：Intel® Core™ i5-3239M CPU @ 2.60GHz

RAM ：16.00 GB

系统类型：64 位操作系统

四、编程实现与调试过程

1、程序结构



2、参数设置

（1）参数列表[5]

模型参数名称	取值	单位
行人间相互作用强（A）	2000	N/m
行人间相互作用范围（B）	0.08	m
松弛时间	0.5	s
行人与障碍物间的作用力（A_w）	10	m/s^2
行人与障碍物间的作用范围（B_w）	0.3	m
人体正压力弹性系数（K）	12000	kg/s^2
人体滑动摩擦力系数（k）	24000	kg/s^2
行人作用区域矩形框 x 取值	1.4	m
行人作用区域矩形框 y 取值	1.2	m
行人作用需求空间参数 a	0.3	m
行人作用需求空间参数 b	1.06	m
仿真步长	0.01	s

（2）代码实现

```
def __init__(self , position):
    # random initialize a agent
    self.mass = 60.0              # 人的质量
    self.radius = 0.3             # 人的半径
    self.desiredV = 0.8           # 人的期望速度
    self.direction = np.array([0.0, 0.0])  # 人的期望速度方向 （应修改为用A*算法更新）
    self.actualV = np.array([0.0,0.0])  # 人的实际速度（向量：大小+方向）用 v = v0 + at更新
    self.tau = 0.01   # 即dt仿真时间间隔
    self.pos = position           # 人的位置
    self.dest = np.array([0,5])   # 目的地 （出口位置）
    self.bodyFactor = 120000       # 公式中第一项的 K
    self.slideFricFactor = 240000  # 公式中第二项的 k
    self.A = 2000                  # Ai
    self.B = 0.08                  # Bi
```

3、模型实现细节

（1）行人仿真

① 自驱动力

$$m_i \frac{v_i^0(t)\, e_i^0(t) - v_i(t)}{\tau_i}$$

```
# 自驱动力
def adaptVel(self):
    deltaV = self.desiredV * self.direction - self.actualV
    if np.allclose(deltaV, np.zeros(2)):    # 若deltaV 接近0则置0
        deltaV = np.zeros(2)
    return deltaV * self.mass / self.tau
```

② 行人之间的相互作用力

$$f_{ij} = \{ A_i \exp[(r_{ij} - d_{ij}) / B_i] + kg(r_{ij} - d_{ij}) \} n_{ij} + \kappa g(r_{ij} - d_{ij})\, \Delta v_{ji}^t t_{ij}$$

```
# 人与人之间的力
def peopleInteraction(self, other):
    rij = self.radius + other.radius
    dij = np.linalg.norm(self.pos - other.pos)
    nij = (self.pos - other.pos) / dij
    first = (self.A * np.exp((rij - dij) / self.B) + self.bodyFactor * ReLU(rij - dij)) * nij
    tij = np.array([-nij[1] , nij[0]])
    deltaVij = (self.actualV - other.actualV) * tij
    second = self.slideFricFactor * ReLU(rij-dij) * deltaVij * tij
    return first + second
```

③ 障碍物对行人的作用力

$$f_{iw} = \{ A_i \exp[(r_i - d_{iw}) / B_i] + kg(r_i - d_{iw}) \} n_{iw} - \kappa g(r_i - d_{iw}) (v_i \cdot t_{iw}) t_{iw}$$

```
# 人与墙之间的力
def wallInteraction(self, wall):
    ri = self.radius
    diw,niw = distanceP2W(self.pos,wall)  # d 为距离 , n为方向向量
    first = (self.A * np.exp((ri - diw) / self.B) + self.bodyFactor * ReLU(ri-diw)) * niw
    tiw = np.array([-niw[1],niw[0]])
    second = self.slideFricFactor * ReLU(ri - diw) * (self.actualV * tiw) * tiw
    return first - second
```

（2）A*算法实现

① 附属功能函数

- H 函数，计算点 neighbor 到 goal 的 manhattan 距离
- 计算 a ， b 两点的欧氏距离
- 更新路径函数

```
# H函数，计算点neighbor 到 goal的manhattan距离
def heuristic_cost_estimate(neighbor, goal):
    x = neighbor[0] - goal[0]
    y = neighbor[1] - goal[1]
    return abs(x) + abs(y)

# 计算a ，b 两点的欧氏距离
def dist_between(a, b):
    return (b[0] - a[0]) ** 2 + (b[1] - a[1]) ** 2

# 更新路径
def reconstruct_path(came_from, current):
    path = [current]            # current即为goal
    while current in came_from:
        current = came_from[current]    # 按照came_from中的节点信息复原出路径
        path.append(current)
    return path
```

② A* 算法函数

```
def astar(array, start, goal):
    #print(array)
    directions = [(0,1),(0,-1),(1,0),(-1,0),(1,1),(1,-1),(-1,1),(-1,-1)]
    # 8个方向
    close_set = set()
    # close list
    came_from = {}
    # 路径集（记录最优路径节点）
    gscore = {start:0}
    # g函数字典，值为到起点的距离 ，key为一个点坐标
    fscore = {start:heuristic_cost_estimate(start, goal)}
    # f函数字典，f = g + h,初始化只有起点的h值
    openSet = []     # open list 建立一个常见的堆结构
    heappush(openSet, (fscore[start], start))
    # 往堆中插入一条新的值 ，内部存按堆排序的f升序堆
```

```
# while openSet 非空
while openSet:
    current = heappop(openSet)[1]
    # 从堆中弹出fscore最小的节点 （heappop函数实现）
    # 循环终止条件
    if current == goal:
    # 当openlist包含目的地节点时，返回path
        path = reconstruct_path(came_from, current)
        # 根据came_from字典（k-v对的v指向父节点）生成path并返回
        length = len(path)
```

```
    direct = np.array([path[length-2][0] - path[length-1][0] , path[length-2][1] - path[length-1][1]])
    return normalize(direct)
    # 返回的是当前的从当前位置到目的地的速度方向向量
close_set.add(current)  # 把当前节点移入close list中
```

```
    for i, j in directions:       # 对当前节点的 8 个相邻节点一一进行检查
        neighbor = current[0] + i, current[1] + j    # 相邻节点的计算
        # 判断节点是否在地图范围内，并判断是否为障碍物
        if 0 <= neighbor[0] < array.shape[0]:      # 地图范围内判断
            if 0 <= neighbor[1] < array.shape[1]: # 地图范围内判断
                if array[neighbor[0]][neighbor[1]] == 1:    # 1为障碍物，有障碍物判断
                    continue  # 跳过障碍物
            else:
                continue  # 跳过超出地图范围
        else:
            continue  # 跳过超出地图范围
        # Ignore the neighbor which is already evaluated.
        if neighbor in close_set:
            continue  # 跳过已进入 Closelist 的节点
        #  计算经过当前节点到达相邻节点的g值，用于比较是否更新
        tentative_gScore = gscore[current] + dist_between(current, neighbor)
        # 如果当前节点的相邻节点不在open list中，将其加入到open list当中
        if  neighbor not in [i[1] for i in openSet]:                # Discover a new node
            heappush(openSet, (fscore.get(neighbor, numpy.inf), neighbor))
        # 若不是更优的解（g不具有更小值）则跳过该节点
        elif tentative_gScore >= gscore.get(neighbor, numpy.inf):   # This is not a better path.
            continue
        # 若未跳过，则该节点为经过的路径，修改came_from列表
        # This path is the best until now. Record it!
        came_from[neighbor] = current  # 相邻节点父节点指向当前节点
        gscore[neighbor] = tentative_gScore
        fscore[neighbor] = tentative_gScore + heuristic_cost_estimate(neighbor, goal)
return False
```

（3）其他功能函数支持

① g 函数

```
# ReLU 即函数g
def ReLU(x):
    if x > 0:
        return x
    return 0
```

② 向量标准化

```
# 向量v的标准化，化成单位向量
def normalize(v):
    norm=np.linalg.norm(v)
    if norm==0:
        return v
    return v/norm
```

③ 点到线段距离计算

```python
# person to wall距离（点到线段距离）
def distanceP2W(point, wall):
    p0 = np.array([wall[0],wall[1]])
    p1 = np.array([wall[2],wall[3]])
    d = p1-p0
    ymp0 = point-p0
    t = np.dot(d,ymp0) / np.dot(d,d)
    if t <= 0.0:
        dist = np.sqrt(np.dot(ymp0,ymp0))
        cross = p0 + t * d
    elif t >= 1.0:
        ymp1 = point - p1
        dist = np.sqrt(np.dot(ymp1,ymp1))
        cross = p0 + t * d
    else:
        cross = p0 + t * d
        dist = np.linalg.norm(cross - point)
    npw = normalize(cross - point)
    return dist,npw
```

（4）模型仿真

- 全局变量定义

```python
# 定义一些全局变量
AGENTNUM = 5          # 人数设置
ROOMSIZE = 10         # 房间大小设置（假设正方形房间）
ITERNUM  = 2          # 迭代次数
agents = []
agent_pos = []
obstacles = []
exits = []
```

- 初始化地图、障碍物与出口

```
# 初始化地图
MAP = np.zeros((ROOMSIZE , ROOMSIZE))
for i in range(ROOMSIZE):
    MAP[0][i] = 1
    MAP[i][0] = 1
    MAP[ROOMSIZE - 1][i] = 1
    MAP[i][ROOMSIZE - 1] = 1

# 初始化障碍物
for i in range(ROOMSIZE):
    for j in range(ROOMSIZE):
        if MAP[i][j] == 1:
            obstacles.append([i,j])

# 为地图添加出口
exit = [0,5]
exits.append(exit)
```

- 初始化人群（随机位置）

```
# 初始化人群（随机位置）
for i in range(AGENTNUM):
    point = [random.random() * ROOMSIZE , random.random() * ROOMSIZE]
    pp =[int(point[0]) , int(point[1])]
    while (pp in exits) or (pp in obstacles) or (pp in agent_pos):
    #防止随机生成的行人位置相互重叠，和与障碍物重叠
        point = [random.random() * ROOMSIZE , random.random() * ROOMSIZE]
        pp = [int(point[0]) , int(point[1])]
    agent_pos.append(point)
    position = np.array(point)
    agent = Agent(position)
    agents.append(agent)
```

- 行人状态更新

```
# 循环ITERNUM次更新人的状态
for i in range(ITERNUM):
    # 更新每一个人的状态
    for idx , ai in enumerate(agents):
        # 获得人的初始速度，位置
        v0 = ai.actualV
        p0 = ai.pos
        # 调用A*算法获取人在当前位置的预期方向
        start = (int(p0[0]) , int(p0[1]))
        ai.direction = astar(MAP , start , tuple(exit))
        adaptForce = ai.adaptVel() # 计算自驱动力
        p2pForce = 0
        w2pForce = 0
```

```
# 计算人与人之间作用力 (sigma(fij))
for idx_other , a_other in enumerate(agents):
    if idx == idx_other:
        continue
    p2pForce += ai.peopleInteraction(a_other)

# 计算人与墙之间作用力 (sigma(fiw))
for wall in walls:
    w2pForce += ai.wallInteraction(wall)

# 计算合力
sumForce = adaptForce + p2pForce + w2pForce
# 加速度
accumu = sumForce / ai.mass
# 更新速度
ai.actualV = v0 + accumu * ai.tau
# 更新位移
ai.pos = p0 + v0 * ai.tau + 0.5 * accumu * (ai.tau ** 2)
```

五、模型实现效果与分析

1、简单场景

(1) 场景概述

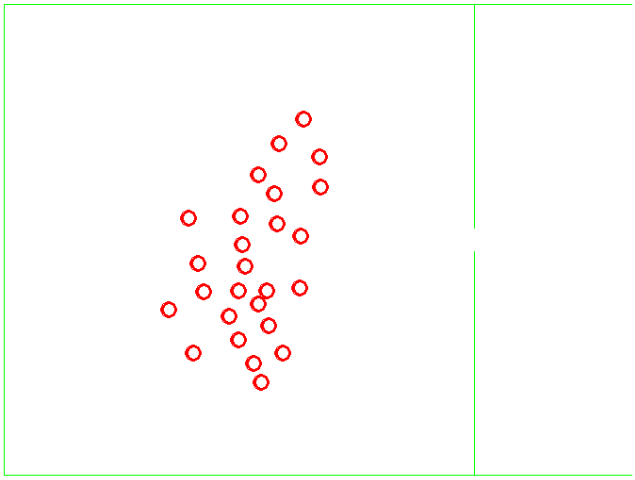
- 构造了一个简单场景，构造了一个密闭房间，仅有一个出口的环境。
- 人群行为模拟为密闭环境疏散行为

(2) 实现方案

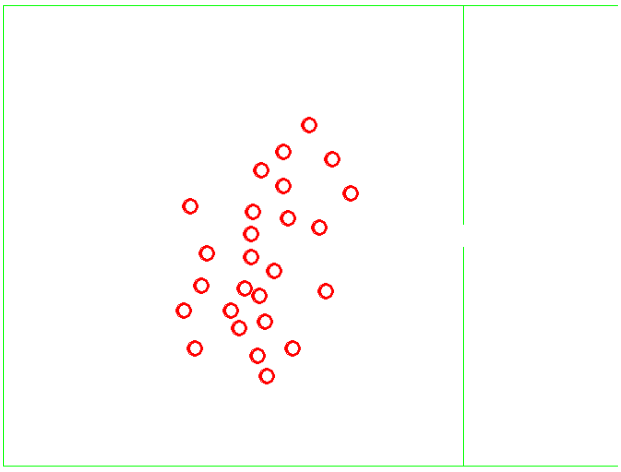
- 采用 pygame 提供的绘图工具，导入社会力模型得到的行人路径轨迹模拟模型实现效果，实现可视化模型评估方案。

(3) 执行效果

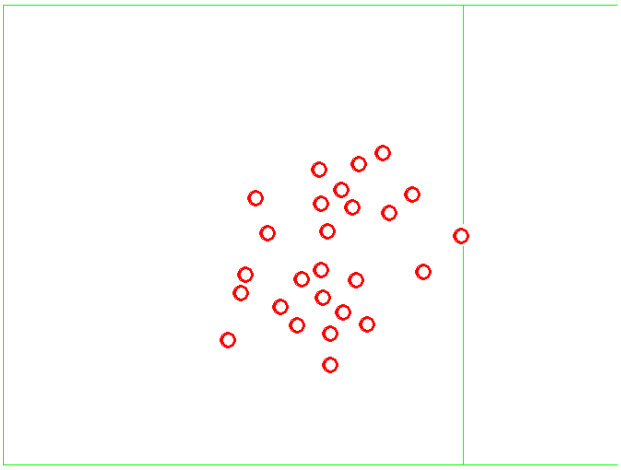
- 初始状态



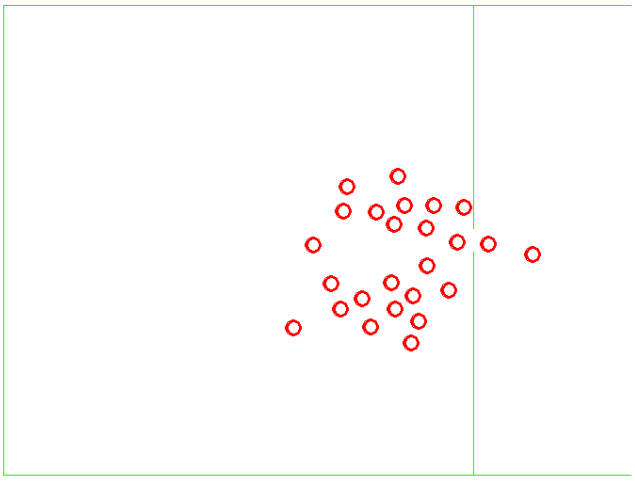
- 人群开始运动



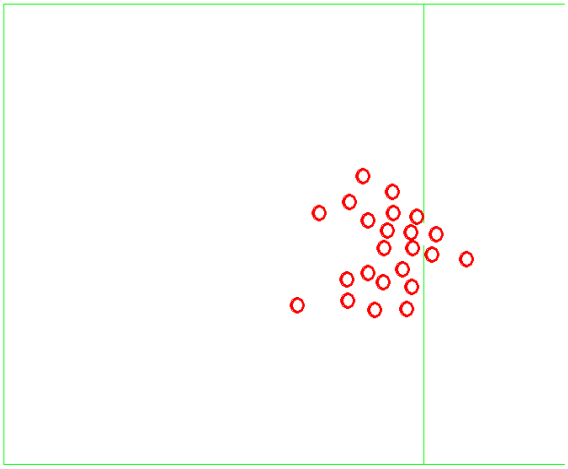
- 第一个人已经抵达出口



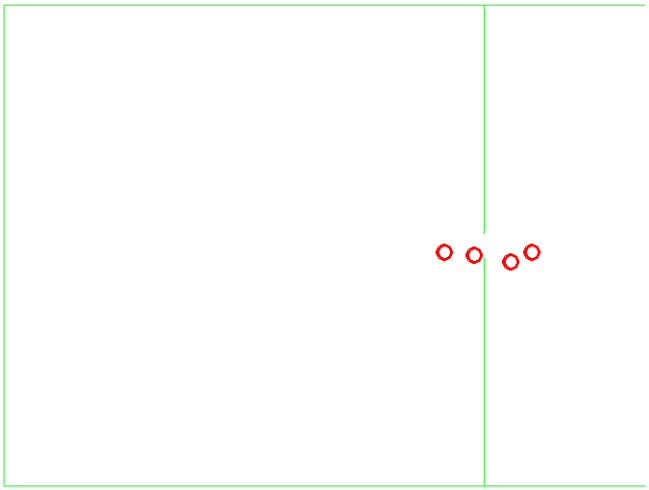
- 人群逐渐逃逸



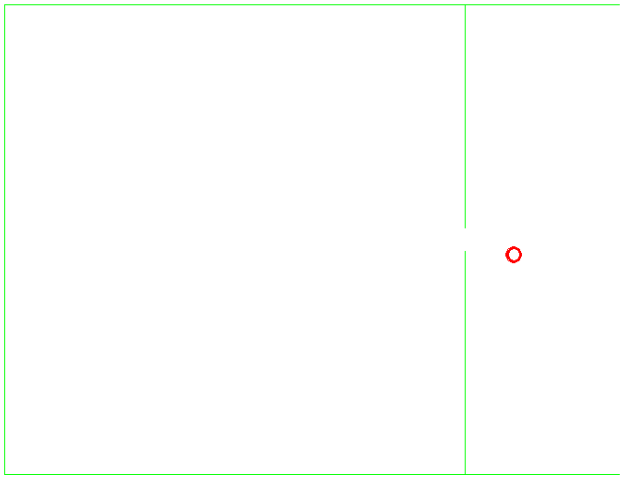
- 窄门口出现人群密集拥挤状况



- 行人几乎完全逃逸（逃逸后的行人即从仿真界面上消除）



- 最终状态（即行人完全从密闭房间中逃逸）



2、带有障碍物的逃逸场景

(1) 场景概述

- 构造了一个带有障碍物的场景，构造了一个密闭房间，仅有一个出口，在出口前放置一个障碍物。

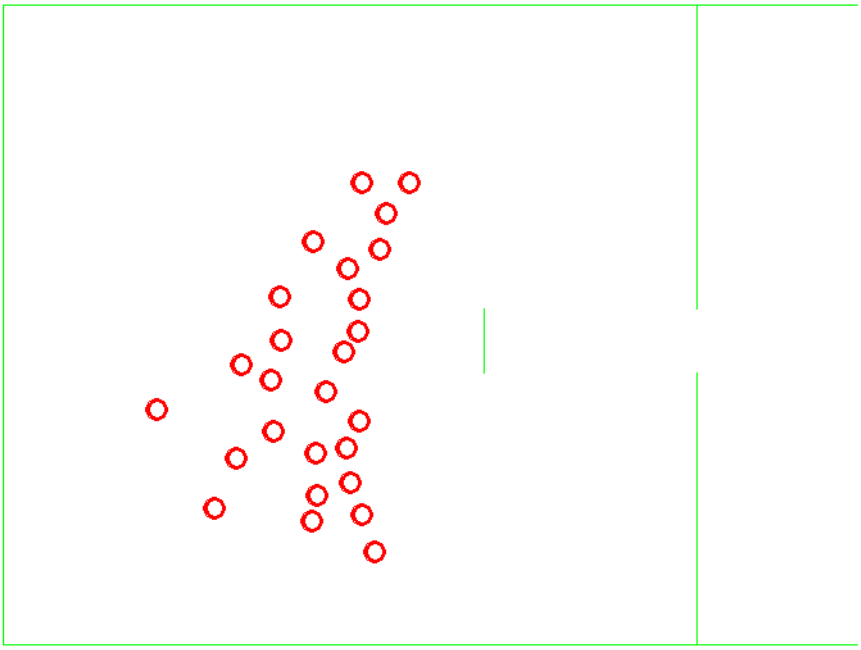
- 人群行为模拟为密闭环境疏散行为。

(2) 实现方案

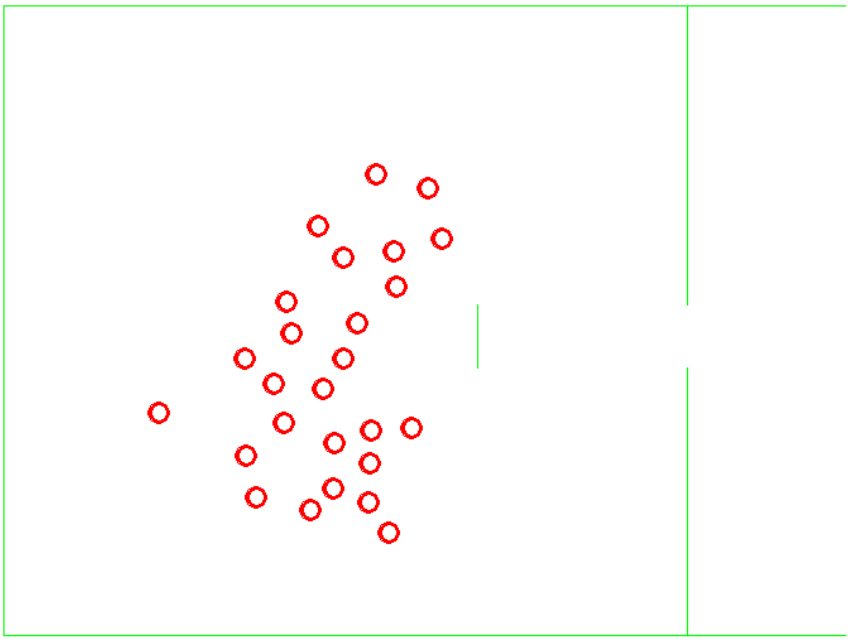
- 采用 pygame 提供的绘图工具，导入社会力模型得到的行人路径轨迹模拟模型实现效果，实现可视化模型评估方案。

(3) 执行效果

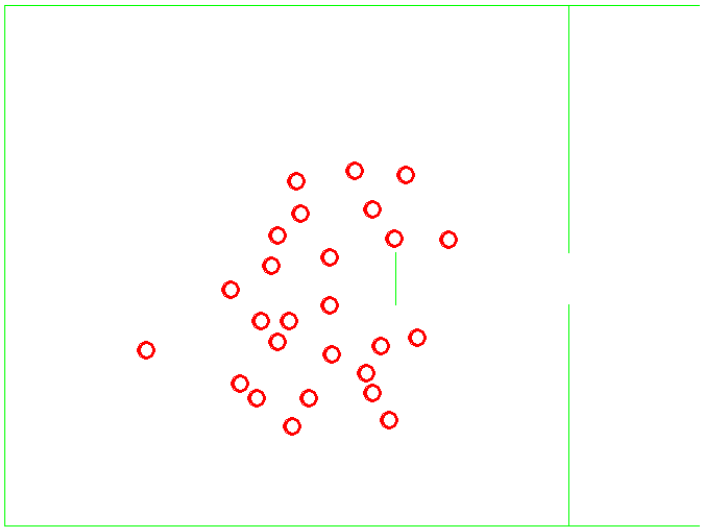
- 初始状态



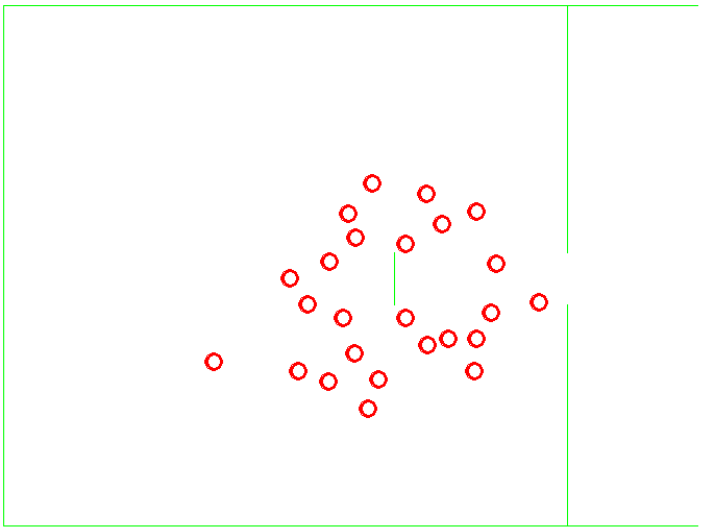
- 人群开始运动



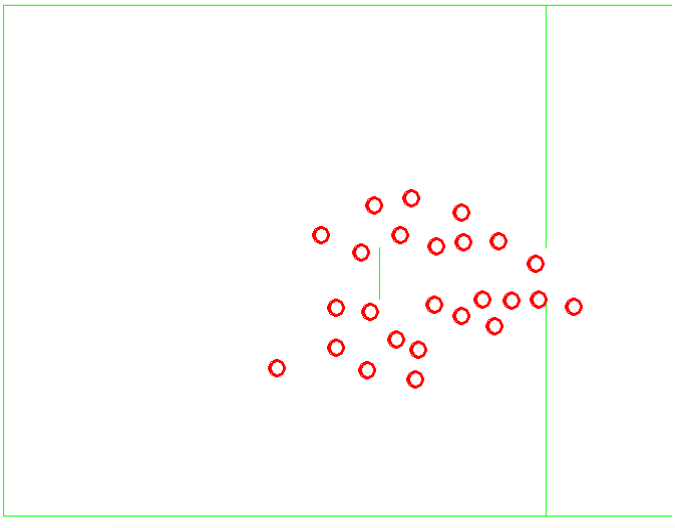
- 人群开始遇到障碍物



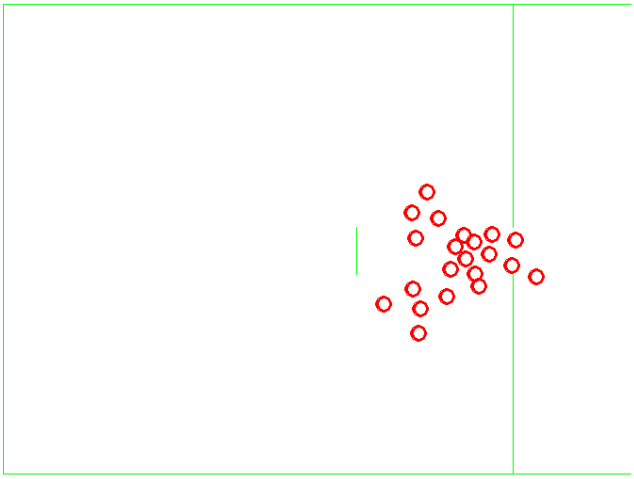
- 人群绕过障碍物通过



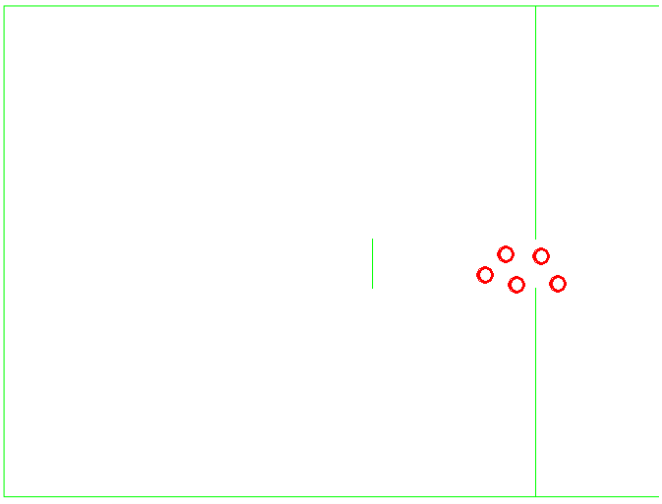
- 已有人抵达出口逃逸，其余人群继续绕过障碍物



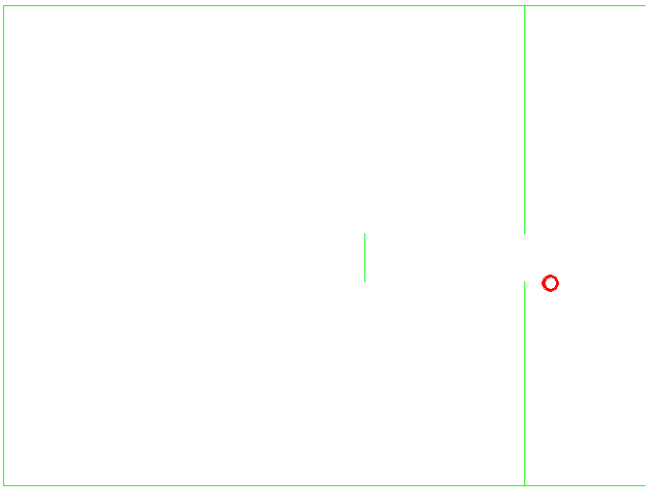
- 大部分人群绕过障碍物，出口处出现拥挤



- 逃逸人群从地图中移除，人群几乎已完全逃逸



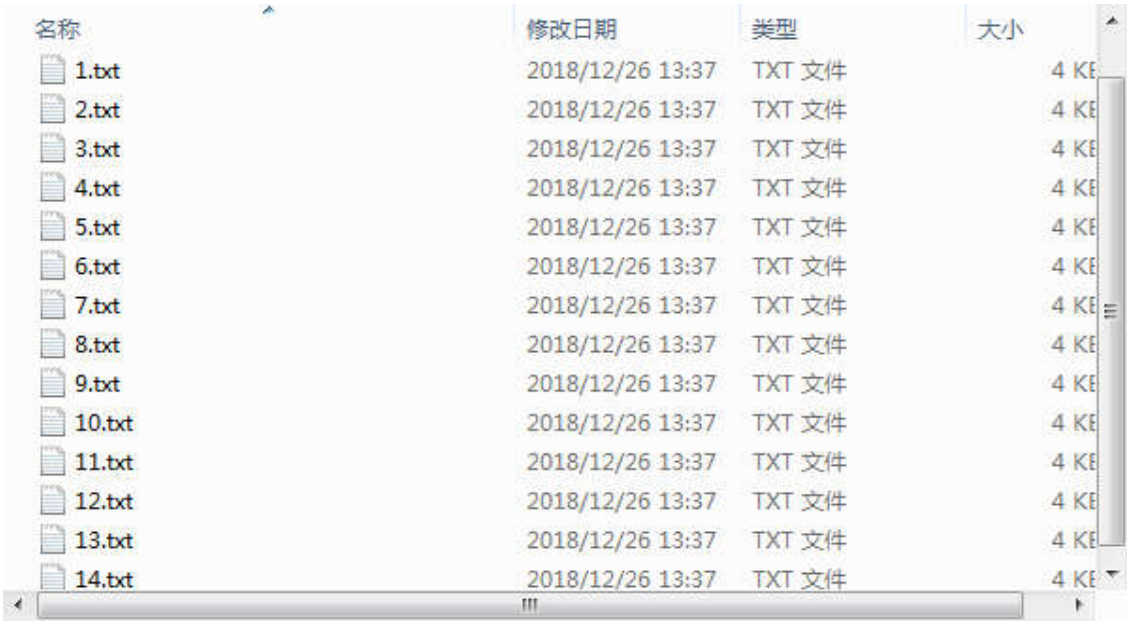
- 终止状态，全部人群均安全逃逸



3、生成数据解读

- 由于模型的计算过程的高度复杂性，计算耗时较长，因此仿真采取先计算出行人路线数据，后将数据传入 GUI 仿真的方式，以加快仿真速度，提高效率。

- 为每个行人单独生成一个路径文件，表示这个行人的行进路线，文件列表如下



名称	修改日期	类型	大小
1.txt	2018/12/26 13:37	TXT 文件	4 KB
2.txt	2018/12/26 13:37	TXT 文件	4 KB
3.txt	2018/12/26 13:37	TXT 文件	4 KB
4.txt	2018/12/26 13:37	TXT 文件	4 KB
5.txt	2018/12/26 13:37	TXT 文件	4 KB
6.txt	2018/12/26 13:37	TXT 文件	4 KB
7.txt	2018/12/26 13:37	TXT 文件	4 KB
8.txt	2018/12/26 13:37	TXT 文件	4 KB
9.txt	2018/12/26 13:37	TXT 文件	4 KB
10.txt	2018/12/26 13:37	TXT 文件	4 KB
11.txt	2018/12/26 13:37	TXT 文件	4 KB
12.txt	2018/12/26 13:37	TXT 文件	4 KB
13.txt	2018/12/26 13:37	TXT 文件	4 KB
14.txt	2018/12/26 13:37	TXT 文件	4 KB

- 每个文件的格式如下（部分）

- 数据表示行人每个时间单位推进后所处的位置（x,y）坐标表示。

1.txt

```
1  21.739246891264717   26.201603286956235
2  21.6825119226747     26.145364303551336
3  21.62577821064935    26.089122822419178
4  21.56904620091138    26.03287795265604
5  21.512315888989587   25.97662970047326
6  21.45558723981591    25.920378136006768
7  21.398860219236816   25.864123327555532
8  21.342152618481013   25.807825180993063
9  21.28548899165917    25.751432303770788
10 21.228865797406993   25.694957630894987
11 21.172266418327183   25.63843871482603
12 21.11568075516198    25.581896156550176
13 21.059102843868786   25.525341097234797
14 21.0025292393305     25.46877949520987
15 20.94595799262448    25.412214499331917
```

六、模型修正与改进方向

在本实验中我们所验证与仿真的 Helbing 的初始社会力模型可以模拟一些实际现象。虽然初始社会力模型能够在一定程度上对大规模人群行为进行模拟，但是该模型还是存在明显的不足。比如，在对人与人之间的作用力进行描述时，模型将所有行人看作是一样的，即所有人对行人 i 的作用力变化规律相同，而在实际情况中，往往会发现，不同的人对同一行人 i 的影响是不同的，好朋友之间会更乐于待在一起，陌生人之间所需要的安全距离会更大; 同时，现有社会力模型认为社会排斥力仅与两个行人之间的距离和两行人的半径和有关，并没有考虑两个行人的站位，但在现实中，当两个行人背靠背而行，谁也看不见谁时，彼此间显然不存在社会排斥力。此外，初始社会力模型中的期望速度是一个定值，只能在给定的期望速度下进行模拟，但实际上行人的期望速度是时刻在改变的。基于上述分析，我们认为初始社会力模型对大规模人群行为的模拟还不够真实，将对模型进行一些修正，使得修正后的模型能够更真实地对大规模人群行为进行模拟。

若能在模型中引入朋友间吸引力，并通过分析相对速度对社会心理力的影响，修正期望速度，最终形成改进的社会力模型，则会大大提高社会力模型的合理性与真实性。

七、神经网络实现社会力模型

1、数据预处理

（1）我们得到的数据集中只有某个点以 1/30 秒间隔呈现的坐标序列，所以我们首先处理坐标数据，得到某点的速度序列。

例如 a 在t_1时间的坐标是(x_1, y_1)，在相邻的t_2时间的坐标是(x_2, y_2)，那么他在t_1时间的速度是$\left(\frac{x_2-x_1}{1/30}, \frac{y_2-y_1}{1/30}\right)$，这样我们就得到了每个人的坐标和速度数据。

（2）我们将传统的 A-star 算法计算出的方向作为评估和修正模型的参考，以计算我们预测值的偏差。这一部分的实现我们使用了之前实现的传统社会力模型的有关内容。

（3）接下来，我们需要计算一个场景中一个人受到的其他人的影响。为了在计算复杂度和模型的准确程度之间进行权衡，考虑五个最近邻居来计算周围人对其的相互作用力。

我们截取同一场景下，所有人在t_1时间的状态，然后计算其他人对 a 的距离，取距离最近的五个人作为对其产生影响的因素，分别计算五个人对 a 的水平相对距离，垂直相对距离，水平相对速度，垂直相对速度。这样，我们得到一个含有 20 个维数的邻居数组 W。

$$W = \left\{ \begin{matrix} \Delta x_i,\ \Delta y_j,\ \Delta V_{xk},\ \Delta V_{yl}, \\ 1 \le i \le 5,\ 6 \le j \le 10,\ 11 \le k \le 15,\ 16 \le l \le 20 \end{matrix} \right\}$$

考虑到神经网络模型需要固定维数的输入，而在我们的数据中，一个人从图中出去以后就不再记录，那么留在图中的人的邻居数很可能不到 5 个，那么我们将不足 5 个人的邻居数组的相应维度置为 0，以示相应的影响为 0。

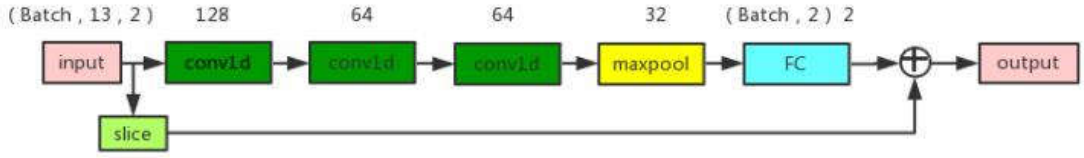
另外我们将相对距离做归一化，这样可以更好的适应神经网络模型。我们将 3 米作为对一个人的移动产生影响的最长距离，所以按照相对距离与 3 的比例做归一化。

$$x_i \ge x, w_i = \frac{x_i - x}{3} = \frac{\Delta x_i}{3}; x_i < x, w_i = -(1 + \frac{\Delta x_i}{3}), 1 \le i \le 5$$

$$y_i \geq y, w_j = 1 - \frac{\Delta y_i}{3}; y_i < y, w_j = -(1 + \frac{\Delta y_i}{3}), 6 \leq j \leq 10$$
$$w_k = \Delta Vx_i = Vx_i - Vx, 11 \leq k \leq 15$$
$$w_l = \Delta Vy_i = Vy_i - Vy, 16 \leq l \leq 20$$

2、神经网络结构

针对有障碍房间这个场景下的行人运动规律仿真，我们使用了如下的神经网络模型来进行建模仿真。



首先是网络的输入，这里我们借鉴了老师论文中的设计思路，并作出了一些自己的改动，输入的具体内容如下：

1	x， y	当前的水平坐标和垂直坐标
2	v_x, v_y	当前的水平速度和垂直速度
3	A_x^*, A_y^*	根据 A*算法结合障碍物和出口计算出的期望方向
4	$\Delta x_1, \Delta y_1$	5 个最近邻的水平相对距离和垂直相对位移
…		
8	$\Delta x_5, \Delta y_5$	
9	$\Delta v_{x1}, \Delta v_{y1}$	5 个最近邻的水平相对速度和垂直相对速度
…		
13	$\Delta v_{x5} \Delta v_{y5}$	

首先，我们先考虑被预测者当前的位置信息和速度信息，然后，使用 Astar 算法的预测方向作为考虑周边环境（障碍我，出口）的期望方向，最后我们使用与被预测者最近的五个人的相对位移和相对速度来衡量被预测者周围人的影响。

在这里，我们并没有将 13 * 2 一共 26 个信息平铺展开，这 13 组信息是一个维度，然后 x，y 又是一个维度，平铺展开不利于保持数据内部的拓扑关系，所以我们是将 13 *

2 的数据直接送进网络并使用 conv1d(1*1 卷积核的 1 维的卷积层)来进行处理。

对于实际数据中可能会有不足 5 个最近邻的情况，我们将相对位移置为 INT_MIN（便于之后归一化），相对速度置为 0。然后对于所有的相对位移，根据一个合理影响半径 r 进行归一化，将相对位移归一化到(-1, 1)之间再输入数据。

然后是网络结构设计，这里我们认为行人的运动是一个马尔科夫决策过程（MDP），即当前下一秒的运动情况只跟当前这一秒的状态有关，跟之前的历史无关，所以我们没有考虑使用 LSTM 等循环神经网络结构来设计网络。其次，由于我们的任务是预测下一秒的运动情况，而直接去回归得到一个下一秒的位置，会受到上一秒位置和位移两个因素的影响，所以我们使用了残差网络设计，即让网络去预测一个位移值，之后把这个位移值加到原位置上得到下一秒的位置，在网络中实际的体现就是这个跳远连接。还有，由于上文所述我们使用 13 * 2 的数据输入，所以简单的全连接层不适合我们的建模，因此我们使用了 1*1 卷积核的一维卷积，这样可以适应于输入，又没有减少连接权重的数量。

我们的神经网络使用 L2 损失函数，即衡量预测值和实际值之间的 L2 范数，使用 Adam 优化算法进行优化，batch_size 取 64，初始学习率为 1e-3，在训练过程中根据损失函数的情况适当调节学习率，一共训练了 100 个 epoch。

3、地图构建

数据集中缺乏的对应地图可以在论文中找到对应的坐标标注，因此需要将对应的墙体以及障碍物坐标记录，并且由于 A*算法需要将地图转换成 01 矩阵，所以针对地图的构建我们实现了一部分代码来支持模型的需求。具体的做法是，取合适的距离步长，持续对于二维坐标从 0 开始累加步长，产生坐标，判断坐标是在障碍物、墙体上，还是在可移动的区域中，可移动则设定当前位置为 0，否则设置为 1.

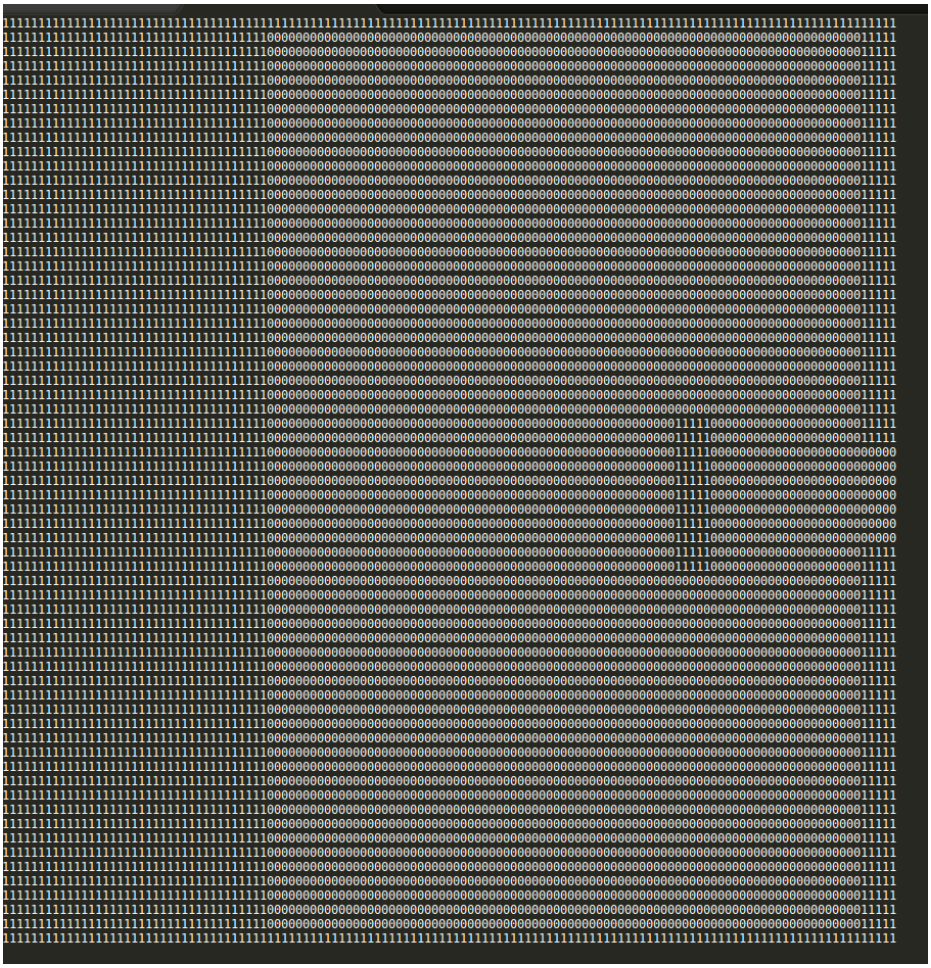
实现代码如下

```
# -*- coding:utf-8 -*-
# 横向障碍物 窄门
import numpy  as np
point00=np.array([3.66,3.66])
point01=np.array([3.66,9.977])
```

```
point10=np.array([11.96,3.689])
point11=np.array([11.96,9.977])
print(point00);
gate00 = np.array([11.96,6.534])
gate01 =np.array([11.96,7.234])
gate10 = np.array([12.45,6.534])
gate11= np.array([12.45,7.234])
#gateDown=np.array([11.96,6.534])
ob00=np.array([9.308,6.311])
ob01=np.array([9.308,7.481])
ob10=np.array([9.88,6.311])
ob11=np.array([9.88,7.481])
dstep = 0.1
row = 0
col = 0
len_wall=gate11[0]
hgt_wall=point11[1]
def inWall(x,y):
    if(x> point00[0] and x < point11[0] and y> point00[1] and y < point11[1] ): return
True
    return False
def inOb(x,y):
    if(x>= ob00[0] and x <= ob11[0] and y >= ob00[1] and y <= ob11[1]): return True
    return False
def inGate(x,y):
    if(x >= gate00[0] and x<= gate11[0]    and y>= gate00[1] and y<=gate11[1]): return
True
    return False
def islegal(x,y):
    if inGate(x,y) or (inWall(x,y) and not inOb(x,y)): return True
```

```
        return False
while(row * dstep <= hgt_wall+dstep):
        with open("map.csv","a") as f:
                if(col * dstep > len_wall):
                        col=0
                        row+=1
                        f.write("\n")
                if(row*dstep > hgt_wall+ dstep):break
                if islegal(col*dstep,row*dstep) and ((not (col==0 or row==0) )):
                        f.write("0")
                else:
                        f.write("1")
col+=1
```

创建出的地图如下：



八、结论

本实验的主要工作是基于 HelBing 提出的传统社会力模型模拟人群运动，同时结合了神经网络对模型进行了实现与验证，主要结论如下

(1) 社会力模型能够形象地用力的形式表达周围环境（如其他行人、边界或障碍物）对所描述的行人行为的影响，且能够较真实地模拟与实际行人运动相符合的运动现象。

(2) 引入求解行人自驱动力、行人之间相互作用力以及行人与障碍物之间相互作用力的合力的方法和一些规则可以较好地解决在社会力模型中出现的碰撞问题。

(3) 采用 A*算法，在每次推进时间步长之后规划行人的预期速度方向，使社会力模型更加具有合理性。

(4) 模拟了给定场景下的行人疏散行为，验证了社会力模型的正确性，并分析了传统的社会力模型的不足，提出了一些可能的改进方向。

九、参考文献

[0] Helbing D, Farkas I J, Vicsek T. Simulating Dynamical Features of Escape Panic[J]. Nature, 2000, 407(6803):487-90.

[1] 单庆超, 张秀媛, 张朝峰. 社会力模型在行人运动建模中的应用综述[J]. 城市交通, 2011, 09(6):71-77.

[2] 万灏, 肖泽南. 利用社会力模型模拟分析人群的自组织现象[J]. 消防科学与技术, 2010, 29(9):745-748.

[3] 陈涛, 应振根, 申世飞,等. 相对速度影响下社会力模型的疏散模拟与分析[J]. 自然科学进展, 2006, 16(12):1606-1612.

[4] 黄鹏, 刘箴. 一种面向人群仿真的改进型社会力模型研究[J]. 系统仿真学报, 2012, 24(9):1916-1919.

[5] 李连天. 基于社会力模型的拥挤人群状态研究与仿真[J].软件导刊,2013(8):36-40.

[6] 汪蕾, 蔡云, 徐青. 社会力模型的改进研究[J]. 南京理工大学学报, 2011, 35(1):144-149.

[7] OpenPTDS Dataset: Pedestrian Trajectories in Crowded Scenarios. Xiao Song, Jinghan Sun, Jing Liu, Kai Chen, Hongnan Xie

[8] 张晋．基于元胞自动机的城域混和交通流建模方法研究[D].杭州:浙江大学信息科学与工程学院，2004．

[9] 百度百科. A*算法

[10] 宋晓老师提供的各项 PPT、讲义以及论文资源

附录 工程文件目录及源代码

1、social_force.py

```
# 社会力仿真主函数
# author : Chen Qixian
# Date : 2018-12-13

import numpy as np
import random
np.seterr(divide='ignore',invalid='ignore')
from person import *
from functions import *
from A_star import *

# 定义一些全局变量
AGENTNUM = 1              # 人数设置
ROOMSIZE = 10             # 房间大小设置（假设正方形房间）
ITERNUM    = 200          # 迭代次数
agents = []
agent_pos = []
obstacles = []
exits = []
exit_agents = []          # 已逃逸行人

# 初始化地图
MAP = np.zeros((ROOMSIZE , ROOMSIZE))
for i in range(ROOMSIZE):
    MAP[0][i] = 1
    MAP[i][0] = 1
    MAP[ROOMSIZE - 1][i] = 1
```

```
    MAP[i][ROOMSIZE - 1] = 1

# 初始化障碍物
for i in range(ROOMSIZE):
    for j in range(ROOMSIZE):
        if MAP[i][j] == 1:
            obstacles.append([i,j])

# 为地图添加出口
exit = [0,5]
exits.append(exit)
MAP[exit[0]][exit[1]] = 0

# 初始化墙面（构造场景）
walls = []
wall1 = [0,0,10,0]
wall2 = [0,0,0,10]
wall3 = [0,10,10,10]
wall4 = [10,0,10,10]
walls.append(wall1)
walls.append(wall2)
walls.append(wall3)
walls.append(wall4)

# 初始化人群（随机位置）
for i in range(AGENTNUM):
    point = [random.random() * ROOMSIZE , random.random() * ROOMSIZE]
    pp =[int(point[0]) , int(point[1])]
    while (pp in exits) or (pp in obstacles) or (pp in agent_pos):  #防止随机生成的行人位置相互重叠，和与障碍物重叠
```

```
            point = [random.random() * ROOMSIZE , random.random() * ROOMSIZE]
            pp = [int(point[0]) , int(point[1])]
        agent_pos.append(point)
        position = np.array(point)
        agent = Agent(position)
        agents.append(agent)

# 循环 ITERNUM 次更新人的状态
for i in range(ITERNUM):
    # 更新每一个人的状态
    for idx , ai in enumerate(agents):
        if ai in exit_agents:
            continue
        # 获得人的初始速度，位置
        v0 = ai.actualV
        p0 = ai.pos

        # 调用 A*算法获取人在当前位置的预期方向
        start = (int(p0[0]) , int(p0[1]))
        ai.direction = astar(MAP , start , tuple(exit))
        adaptForce = ai.adaptVel() # 计算自驱动力
        p2pForce = 0
        w2pForce = 0

        # 计算人与人之间作用力（sigma(fij))
        for idx_other , a_other in enumerate(agents):
            if idx == idx_other:
                continue
            p2pForce += ai.peopleInteraction(a_other)
```

```
            # 计算人与墙之间作用力（sigma(fiw))
            for wall in walls:
                w2pForce += ai.wallInteraction(wall)

            # 计算合力
            sumForce = adaptForce + p2pForce + w2pForce
            # 加速度
            accumu = sumForce / ai.mass
            # 更新速度
            ai.actualV = v0 + accumu * ai.tau
            # 更新位移
            ai.pos = p0 + v0 * ai.tau + 0.5 * accumu * (ai.tau ** 2)
            if ai.pos[0] > ROOMSIZE or ai.pos[1] > ROOMSIZE or ai.pos[0] < 0 or ai.pos[1] <
0:
                exit_agents.append(ai)
                continue
            ai.path.append(ai.pos)

#行人路径信息输出到文件
for idx , ai in enumerate(agents):
    filename = "./#1/" + str(idx) + ".txt"
    doc = open(filename , 'w')
    for posi in ai.path:
        print(posi[0], "       " , posi[1] , file = doc)
    doc.close()
```

2、A_star.py

```
# 实现 A*算法函数
# author : 陈麒先
# date : 2018-12-14
```

```
import numpy
from heapq import heappush,heappop
from functions import *

# H 函数，计算点 neighbor  到  goal 的 manhattan 距离
def heuristic_cost_estimate(neighbor, goal):
    x = neighbor[0] - goal[0]
    y = neighbor[1] - goal[1]
    return abs(x) + abs(y)

#  计算 a , b  两点的欧氏距离
def dist_between(a, b):
    return (b[0] - a[0]) ** 2 + (b[1] - a[1]) ** 2

#  更新路径
def reconstruct_path(came_from, current):
    path = [current]                    # current 即为 goal
    while current in came_from:
        current = came_from[current]     #  按照 came_from 中的节点信息复原出路径
        path.append(current)
    return path



# astar function returns a list of points (shortest path)
def astar(array, start, goal):
    #print(array)
    directions = [(0,1),(0,-1),(1,0),(-1,0),(1,1),(1,-1),(-1,1),(-1,-1)]
   # 8 个方向
```

```
close_set = set()
# close list
came_from = {}
# 路径集（记录最优路径节点）
gscore = {start:0}
# g 函数字典，值为到起点的距离 ，key 为一个点坐标
fscore = {start:heuristic_cost_estimate(start, goal)}
# f 函数字典，f = g + h,初始化只有起点的 h 值
openSet = []        # open list 建立一个常见的堆结构
heappush(openSet, (fscore[start], start))
# 往堆中插入一条新的值 ，内部存按堆排序的 f 升序堆
# while openSet 非空
while openSet:
    current = heappop(openSet)[1]
   # 从堆中弹出 fscore 最小的节点 （heappop 函数实现）
    # 循环终止条件
    if current == goal:
    # 当 openlist 包含目的地节点时，返回 path
        path = reconstruct_path(came_from, current)
        # 根据 came_from 字典（k-v 对的 v 指向父节点）生成 path 并返回
        length = len(path)
        direct = np.array([path[length-2][0] - path[length-1][0] , path[length-2][1] -
path[length-1][1]])
        return normalize(direct)
        # 返回的是当前的从当前位置到目的地的速度方向向量
    close_set.add(current)  # 把当前节点移入 close list 中

    for i, j in directions:         # 对当前节点的 8 个相邻节点一一进行检查
        neighbor = current[0] + i, current[1] + j    # 相邻节点的计算
        # 判断节点是否在地图范围内，并判断是否为障碍物
```

```
                if 0 <= neighbor[0] < array.shape[0]:          # 地图范围内判断
                    if 0 <= neighbor[1] < array.shape[1]: # 地图范围内判断
                        if array[neighbor[0]][neighbor[1]] == 1:     # 1 为障碍物，有障碍物
判断
                            continue   # 跳过障碍物
                    else:
                        continue   # 跳过超出地图范围
                else:
                    continue   # 跳过超出地图范围
                # Ignore the neighbor which is already evaluated.
                if neighbor in close_set:
                    continue   # 跳过已进入 Closelist 的节点
                #   计算经过当前节点到达相邻节点的 g 值，用于比较是否更新
                tentative_gScore = gscore[current] + dist_between(current, neighbor)
                # 如果当前节点的相邻节点不在 open list 中，将其加入到 open list 当中
                if   neighbor not in [i[1] for i in openSet]:                    # Discover a new
node
                    heappush(openSet, (fscore.get(neighbor, numpy.inf), neighbor))
                # 若不是更优的解（g 不具有更小值）则跳过该节点
                elif tentative_gScore >= gscore.get(neighbor, numpy.inf):     # This is not a
better path.
                    continue
                # 若未跳过，则该节点为经过的路径，修改 came_from 列表
                # This path is the best until now. Record it!
                came_from[neighbor] = current   # 相邻节点父节点指向当前节点
                gscore[neighbor] = tentative_gScore
                fscore[neighbor] = tentative_gScore + heuristic_cost_estimate(neighbor, goal)
        return False
```

```
if __name__ == "__main__":
    nmap = numpy.array([
        [0,0,0,0,0,0,0,0,0,0,0,0,0,0],
        [1,0,1,1,1,1,1,1,1,1,1,1,0,1],
        [0,0,0,0,0,0,0,0,0,0,0,0,0,0],
        [1,0,1,1,1,1,1,1,1,1,1,1,1,1],
        [0,0,0,0,0,0,0,0,0,0,0,0,0,0],
        [1,1,1,1,1,1,0,1,1,1,1,1,0,1],
        [0,0,0,0,0,0,0,0,0,0,0,0,0,0],
        [1,0,1,1,1,1,1,1,1,1,1,1,1,1],
        [0,0,0,0,0,0,0,0,0,0,0,0,0,0],
        [1,1,1,1,1,1,1,1,1,1,1,1,0,1],
        [0,0,0,0,0,0,0,0,0,0,0,0,0,0]])

    path = astar(nmap, (0,1), (2,3))

    print(path)
```

3、Person.py

```
# 模拟人员类
# author : 陈麒先
# date : 2018-12-13

import numpy as np
import random
np.seterr(divide='ignore',invalid='ignore')
from functions import *

class Agent(object):
    def __init__(self , position):
```

```
        # random initialize a agent
        self.mass = 60.0                    # 人的质量
        self.radius = 0.3                   # 人的半径
        self.desiredV = 0.8                 # 人的期望速度
        self.direction = np.array([0.0, 0.0])  # 人的期望速度方向 （应修改为用 A*算法更新）
        self.actualV = np.array([0.0,0.0])  # 人的实际速度（向量：大小+方向）用 v = v0 + at 更新
        self.tau = 0.01     # 即 dt 仿真时间间隔
        self.pos = position                 # 人的位置
        self.dest = exit    # 目的地 （出口位置）
        self.bodyFactor = 120000            # 公式中第一项的 K
        self.slideFricFactor = 240000   # 公式中第二项的 k
        self.A = 2000                       # Ai
        self.B = 0.08                       # Bi

    # 以下计算受力带入公式计算

    # 自驱动力
    def adaptVel(self):
        deltaV = self.desiredV * self.direction - self.actualV
        if np.allclose(deltaV, np.zeros(2)):    # 若 deltaV 接近 0 则置 0
            deltaV = np.zeros(2)
        return deltaV * self.mass / self.tau

    # 人与人之间的力
    def peopleInteraction(self, other):
        rij = self.radius + other.radius
        dij = np.linalg.norm(self.pos - other.pos)
        nij = (self.pos - other.pos) / dij
```

```
        first = (self.A * np.exp((rij - dij) / self.B) + self.bodyFactor * ReLU(rij - dij)) * nij
        tij = np.array([-nij[1] , nij[0]])
        deltaVij = (self.actualV - other.actualV) * tij
        second = self.slideFricFactor * ReLU(rij-dij) * deltaVij * tij
        return first + second

    # 人与墙之间的力
    def wallInteraction(self, wall):
        ri = self.radius
        diw,niw = distanceP2W(self.pos,wall)   # d 为距离 ， n 为方向向量
        first = (self.A * np.exp((ri - diw) / self.B) + self.bodyFactor * ReLU(ri-diw)) * niw
        tiw = np.array([-niw[1],niw[0]])
        second = self.slideFricFactor * ReLU(ri - diw) * (self.actualV * tiw) * tiw
        return first – second
```

4、function.py

```
# 功能函数
# author : 陈麒先
# date : 2018-12-14

import numpy as np

# ReLU 即函数 g
def ReLU(x):
    if x > 0:
        return x
    return 0

# 向量 v 的标准化，化成单位向量
```

```
def normalize(v):
    norm=np.linalg.norm(v)
    if norm==0:
        return v
    return v/norm


# person to wall 距离（点到线段距离）
def distanceP2W(point, wall):
    p0 = np.array([wall[0],wall[1]])
    p1 = np.array([wall[2],wall[3]])
    d = p1-p0
    ymp0 = point-p0
    t = np.dot(d,ymp0) / np.dot(d,d)
    if t <= 0.0:
        dist = np.sqrt(np.dot(ymp0,ymp0))
        cross = p0 + t * d
    elif t >= 1.0:
        ymp1 = point - p1
        dist = np.sqrt(np.dot(ymp1,ymp1))
        cross = p0 + t * d
    else:
        cross = p0 + t * d
        dist = np.linalg.norm(cross - point)
    npw = normalize(cross - point)
    return dist,npw
```

5、gui.py

```
import numpy as np
import pygame
import os
```

```
data_dir="./dataset"
wall_file="./walls.csv"

walls = []
for line in open(wall_file):
    coords = line.split(',')
    wall = []
    wall.append(float(coords[0]))
    wall.append(float(coords[1]))
    wall.append(float(coords[2]))
    wall.append(float(coords[3]))
    walls.append(wall)



agents=[]

for filename in os.listdir(data_dir):
    with open(data_dir+'/'+filename,'r') as f:
        agent=[]
        content=f.read()
        for line in content.split('\n'):
            if len(line)==0:
                break
            sec=[x,y]=[int(float(z)*50) for z in line.strip().split('\t')]
            agent.append(sec)
        agents.append(agent)

screen = pygame.display.set_mode(([1000,800]))
pygame.display.set_caption('Social Force Model')
```

```
step=0
running=True
while running:
    event = pygame.event.wait()
    if event.type == pygame.QUIT:
        running=False
    screen.fill([255,255,255])
    for wall in walls:
        startPos = np.array([wall[0],wall[1]])*25
        endPos = np.array([wall[2],wall[3]])*25
        pygame.draw.line(screen, ([0,255,0]),startPos,endPos)
    for agent in agents:
        if(step < len(agent)):
            x=agent[step][0]
            y=agent[step][1]
            pygame.draw.circle(screen, ([255,0,0]),([x,y]),9,3)
    pygame.display.flip()
    pygame.time.wait(10)
    step+=1
```