北京郵電大學

数据结构实验报告

题目： 加里森的任务

姓　　名：魏生辉

学　　院：计算机学院（国家示范性软件学院）

专　　业：计算机类

班　　级：2023211307

学　　号：2023211075

指导教师：杨震

2024年 10月

目 录

1 需求分析

1.1 题目描述

在首尾相接而组成的环（单循环链表）中有编号为 1 至 n 的 n 个元素依次首尾相接地排列，规定正方向为序号 1 开始依次指向下一个。初始时以第 x 个元素为起点，重复以下过程直到链表长度为 1：以起点为第 1 个元素，沿正方向找到第 y 个元素 deleteNode，从环中删除该元素，再将该元素下一个元素作为新起点。

求链表长度被删减到1时，环中仅剩的一个元素的序号是否是 1。

1.2 输入描述

输入中读入数据。输入一行三个整数，分别表示 n，x，y，这三个整数用空格分隔。
其中各个值的范围需要满足 $1 < n \leqslant 10^4$ $0 < x \leqslant n$ $0 < y \leqslant 3 \times 10^4$。

1.3 输出描述

输出结果划分为四种情况：

1. 输入合法且程序正常运行结束。此时输出两行，第一行一个字符”Y” 或者”N”,Y表示

   最后个元素是1，即加里森无需执行任务。N则反之，第二行一个数字，表示最后剩下的的序号。

2. 输入不合法。此时输出一行一个字符串”Wrong input.”。

3. 内存分配失败，此时程序直接结束程序，没有输出。

4. 链表被判空，此时程序直接结束程序，没有输出。

1.4 样例输入输出

1.4.1 样例输入输出 1

【输入1】7 -8 5

【输出1】Wrong input

1.4.2 样例输入输出 2

【输入2】 1 2 3

【输出2】Wrong input

1.4.3 样例输入输出 3

【输入3】9999 9991 9988

【输出3】N

9277

1.4.4 样例输入输出 4

【输入4】8888 7777 12098

【输出4】N

412

1.4.5 样例输入输出 5

【输入5】40 36 8

【输出5】Y

1.4.6 样例输入输出 6

【输入6】8 5 -1

【输出7】Wrong input

1.5 程序功能

程序通过标准输入进来的的 n, x, y 计算出最后循环链表中仅剩的节点序号，并且与 1 对比 ，输出结果：Y表示最后个元素是1 ，即加里森无需执行任务。N则反之，第二行一个数字，表示最后剩下的的序号。

2 概要设计

2.1 问题解决思路概述

预先准备：首先建立单循环链表，在链表中依次插入 n 个结点表示 n 名队员，新建计数指针模拟计数过程。
删除规则： 从元素 x 开始，每经过 y 步就删除一个元素，并将该元素的下一个作为新的起点。
最终目标： 最后剩下一个元素时，判断它是否为 1。

2.2 数据结构类型/单循环链表的定义

```
//数据对象    定义链表节点结构
typedef struct Node
{
      int date;       // 节点的值，表示元素的序号
       struct Node *next; // 指向下一个节点的指针
} Node;


/*
 * 操作：创建链表节点的函数
 * 前：提供节点的值
 * 后：返回一个指向新节点的指针，节点的值为传入的 date
 */
Node *createNode(int date)
{  //
}

/*
 * 操作：初始化循环链表并插入n个节点
 * 前件：提供节点总数 n
 * 后件：返回一个循环链表，头节点为 1，形成闭合的循环链表
 */
Node *initCircularList(int n)
{
  //
}


/*
 * 操作：删除链表中的某个节点
 * 前：提供当前节点 start 和要删除的节点间隔 y
 * 后：删除第 y 个节点，并返回新起点
 */
Node *deleteNode(Node *start, int y, int *length)
{//

}

/*
 * 操作：释放链表的所有节点
 * 前：链表的头节点 head
 * 后：释放链表中的所有节点的内存，链表不再有效
 */
void freeList(Node *head)
{   //
}
```

2.3 主程序的流程

1. • 输入:从用户处获取 n, x 和 y 的值, 再对输入值进行合法性检查。
2. • 建立链表,在链表中依次插入 n 个结点,并且赋给各节点序号。
3. • 找到第 x 个结点
5. • 循环至链表长度为1: 找到当前节点之后的第 y − 1 个结点, 删除这个结点
6. • 输出且释放空间

2.4 各程序模块之间的层次关系

函数模块层次关系图如下图。

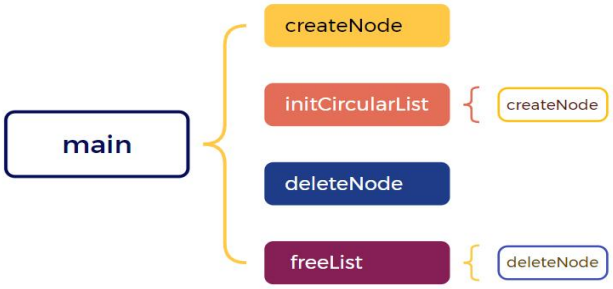


图 1: 函数的层次模块关系

3 详细设计

3.1 伪代码的设计及链表的实现

1. scanf("%d %d %d", &n, &x, &y);

   输入 n，x，y
   - 提示用户输入 n（总人数）、x（起始位置）和 y（步长）
   - 如果输入不合法（没有满足 $1 < n \leqslant 10^4$ $0 < x \leqslant n$ $0 < y \leqslant 3 \times 10^4$ 或 n <= 1 或者 x 不在 1 到 n 之间 或者 y <= 0），则输出“输入不合法”，结束程序

2. Node *initCircularList(int n)

   初始化循环链表
   - 调用函数 initCircularList 创建一个包含 n 个节点的循环链表
   - 调用函数：createNode使得每个节点的 `date` 属性表示节点的序号，依次为 1 到 n
   - 链表最后一个节点的 next 指向头节点，形成一个循环结构

3. for (int i = 1; i < x; i++)
   {
   start = start->next;
   }

   移动到第 x 个节点
   - 初始化指针 `start` 指向链表的头节点
   - 通过遍历链表，从头节点开始移动到第 `x` 个节点
   - 设置当前指针 `start` 指向这个节点，作为删除操作的起点

4. int length = n;

   初始化链表长度
   - 初始化变量 length = n，表示当前链表的长度

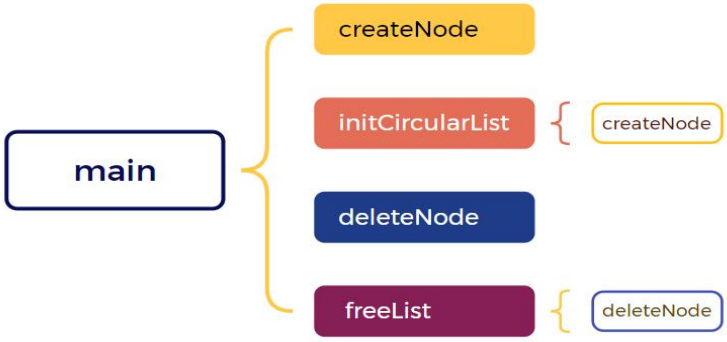
5. while (length > 1)
   {
   // 使用 y % length 来处理步长
   int step = y % length;
   step = step == 0 ? length : step; // 步长为0时，处理为length
   // 删除第 y 个元素，并更新起点
   start = deleteNode(start, y, &length);
   }

   循环 直到链表长度为1：找到当前节点之后的第 y - 1 个节点，删除这个节点
   - 循环执行 n - 1 次，逐步删除节点，直到链表中只剩下一个节点
   - 每次删除时：
   - 使用公式 step = y % length计算需要删除的节点位置（步长 y）
   - 调用函数 deleteNode(start, step, &length)` 删除节点，并将链表长度减 1
   - 更新 start 指向被删除节点的后继节点，继续下一轮循环

6. 输出剩余的节点

7. 释放空间
   - 调用函数 freeList(head) 释放链表中所有节点的内存，防止内存泄漏
   - 遍历链表，依次释放每个节点的内存
   - 程序结束

3.2 函数的调用关系图



4 调试分析报告

4.1 调试过程中遇到的问题和解决方法

初步完成本实验的基础代码后，通过了几组简单的样例，随后进行严格的非法输入、边界条件测试发现了些许问题。

1 • 边界条件错误：

解决方法：调整循环的边界条件，本实验中调整循环边界条件后顺利解决此问题。

2 • 不合法输入：

初始代码存在了n=1,无法判定的问题，加上限定条件后，顺利解决。

3 • 空指针问题

调试发现尾节点访问无效内存,发现为NULL,判断为没有在链表尾部添加指回头节点的指针,添加后正确执行。

4 • 第一个节点元素忘记赋值

大样本测试样例实验时，发现会输出错误值，调试后发现第一个节点元素存在忘记赋值的问题，添加后解决问题。

4.2 设计实现的回顾讨论

1 • 删除操作需要细心设计：
通过一个 for 循环，从当前节点 start 开始，遍历 y - 1 次。
每次循环更新 prev 为 current，并将 current 移动到下一个节点（current = current->next）。
当循环结束时，current 指向第 y 个节点，prev 指向第 y-1 个节点。接着进行删除和链表长度减少操作。

2 • 单循环链表中存在一个特殊的头结点，这个头节点需要另加特殊处理，为了方便问题解决，需跳过对头节点的处理。

4.3 算法复杂度分析

1•时间复杂度

createNode，freeList，函数的时间复杂度均为 O(1)。

initCircularList，deleteNode函数的时间复杂度为 O(n)，但是主程序调用deleteNode函数时，由于还要进行while（length > 1)的循环，最终使得主程序复杂度为 $O(n^2)$。

综上整体时间复杂度为 $O(n^2)$

2•空间间复杂度

由于不存在递归调用带来的栈空间增长,其他部分如临时变量的空间开销是常数级的,

所以，总的空间复杂度可以归结为存储链表节点的需求：O（n）。

4.4 改进设想的经验和体会

4.4.1 改进 1--使用%优化时间

当y 比n 大的时候，逐个寻找目标节点浪费大量时间，可以使用 y % length 来处理步长

代码如下：

```
// 使用 y % length 来处理步长
int step = y % length;
step = step == 0 ? length : step; // 步长为0时，处理为length
// 删除第 y 个元素，并更新起点
start = deleteNode(start, y, &length);
```

4.4.2 改进2 --增强程序健壮性

1.加入 isEmpty 函数 判断链表是否为空

2.将主函数流程中 寻找x位置的代码封装成新的函数findXthNode，用于循环的代封装成deleteUntilOneLeft，封装结果如下：

```
// 封装的函数：用于找到链表中的第 x 个节点
Node *findXthNode(Node *head, int x)
{
    Node *start = head;
    // 找到第 x 个节点
    for (int i = 1; i < x; i++)
    {
        start = start->next;
    }
    return start;
}
// 封装的函数：循环删除节点，直到链表中只剩下一个节点
Node *deleteUntilOneLeft(Node *start, int y, int *length)
{
    // 当链表中只剩下一个节点时，结束循环
    while (*length > 1)
    {
        // 使用 y % length 来处理步长
        int step = y % (*length);
        step = (step == 0) ? (*length) : step; // 步长为0时，处理为length
        // 删除第 y 个元素，并更新起点
        start = deleteNode(start, step, length);
    }
    return start;
}
```

4.4.3 改进 3--设计递归解法（时间复杂度O(n)）

设计josephus 函数（以方便后续测试）：

使用迭代方式（等价于递归解法的逆推思路）来解决约瑟夫问题。

从 n=2 开始，逐步计算删除元素后的剩余序号变化。由于每删除一个元素后，新的序号需要重新调整，因此通过 ans = (ans + y) % i 来更新序号。

最终，返回的 ans 是从 0 开始计算的最后剩下的元素的序号。

代码展示：

```
#include <stdio.h>
int josephus(int n, int y) {
    int ans = 0; // 初始化最后剩下的元素序号
    // 通过逆推计算约瑟夫问题的解
    for (int i = 2; i <= n; ++i) {
        ans = (ans + y) % i; // 每次计算删除后的新序号
    }
    return ans; // 返回最终的序号，注意此时序号是从0开始的
}

int main() {
    int n, x, y;
    // 输入
    scanf("%d %d %d", &n, &x, &y);
    // 计算约瑟夫环解法
    int ans = josephus(n, y); // 获取最后剩下的人的序号（从0开始计数）
    // 将计算结果映射到以 x 为起点的情况，并且转化为 1-based 索引
    printf("%d\n", (ans + x - 1) % n + 1); // 输出最后剩下的人，序号从1开始
    return 0;
}
```

4.4.4 改进4 --简化主函数循环条件

无需记录length以保证循环到最后一个元素，只需循环n-1次即可。

5 用户使用说明

1. 使用 gcc 编译生成可执行文件。

gcc -o main -std=c11 main.c

2. 执行可执行文件：

在在 Linux或macOS 环境下： ***./main***

- 在 Windows cmd 环境下： ***main***

3. 输入数据

执行程序后，用户需要通过标准输入提供三个整数，这三个整数用空格分隔。

示例：***5 2 3***

4. 输出参照1.3部分

6 测试结果

测试部分划分为如下环节。

6.1 测试1--基础样例测试

进行1.4 节的样例测试。

例如

【输入】

```
7 -8 5
```

【输出】

```
Wrong input.
```

6. 2测试2--非法输入和边界条件

测试非法输入和边界条件。

【输入】

```
7 -1 5
```

【输出】

```
Wrong input.
```

【输入】

```
44 2 -7
```

【输出】

```
Wrong input.
```

【输入】

```
1111119990 1 2
```

【输出】

```
Wrong input.
```

【输入】

```
-10000 5723 4627
```

【输出】

```
Wrong input.
```

6.2 测试3--利用输出语句与人力手动计算结果校对

利用输出语句加在删除后的操作,观察每次删除的元素，与人力手动计算结果校对

【输入1】 9 1 5

【输出1】 5
1
N

此样例中环中删除的元素依次为 5，1与自己手动模拟结果相符。

【输入2】 10 3 3

【输出2】 5
8
1
N

此样例中环中删除的元素依次为 5，8,1与自己手动模拟结果相符。

6.4 测试4--大样本数据测试

利用递归求法代码和原代码对比：

【输入】

```
9999 9991 9988
```

【输出】

```
N
9277
```

【输入】

```
8888 7777 12098
```

【输出】

```
N
4162
```

【输入】

```
7777 7774 30003
```

【输出】

```
N
1079
```

6.5 测试第五部分--python生成大量测试数据验证

为了方便第七部分的可视化模型建立，这里使用python生成数据，代码如下：

```
def josephus(n, x, y):
    people = list(range(1, n + 1))
    index = x - 1  # 将x 转换为0 索引

    while len(people) > 1:
        index = (index + y - 1) % len(people)
        people.pop(index)

    return people[0]
# 查找满足条件的 (x, y, n) 组合
def find_valid_combinations(max_n):
    valid_combinations = []
    for n in range(1, max_n + 1):
        for x in range(1, n + 1):
            for y in range(1, n + 1):
                if josephus(n, x, y) == 1:
                    valid_combinations.append((x, y, n))
    return valid_combinations
# 主函数
if __name__ == "__main__":
    max_n = int(input("请输入最大的 n 值："))
    # 获取满足条件的组合
    combinations = find_valid_combinations(max_n)
    # 打印满足条件的组合
    if combinations:
        #print("能够使最后剩下的节点为 1 的 (n, x, y) 组合如下：")
        for combo in combinations:
            print(f"{combo[2]}, {combo[0]}, {combo[1]}")
```

说明：上述代码，通过三重循环和递归函数的思路可以生成所有满足输出Y的数据

在 n <= 10，n <= 50，n <= 100 的范围下分别随机生成 100 组测试数据

样本量过大 下面是部分数据示例

n=10 （x，y，n）	n=50 （x，y，n）	n=100 （x，y，n）
1, 1, 1 2, 1, 2 2, 2, 1 3, 2, 1 3, 2, 2 3, 3, 3 4, 1, 2 4, 1, 3 4, 2, 1 4, 4, 4 5, 1, 4 5, 2, 1 5, 3, 3 5, 4, 2 5, 5, 5 6, 1, 3 6, 1, 5 6, 2, 1 6, 3, 2 6, 3, 4 6, 4, 6 7, 2, 1 7, 2, 2 7, 3, 5 7, 4, 7 7, 5, 3 7, 6, 6 7, 7, 4 8, 1, 2 8, 1, 6 8, 2, 1 8, 3, 3 8, 4, 4 8, 6, 7 8, 6, 8 8, 7, 5 9, 1, 3 9, 1, 4 9, 2, 1 9, 3, 5 9, 3, 9 9, 4, 6 9, 8, 2 9, 8, 8 9, 9, 7 10, 1, 8 10, 2, 1 10, 3, 7 10, 4, 10 10, 5, 9 10, 7, 2 10, 7, 4 10, 8, 3 10, 9, 5 10, 9, 6	47, 26, 20 47, 27, 38 47, 28, 9 47, 29, 29 47, 30, 27 47, 30, 40 47, 30, 45 47, 31, 26 47, 31, 47 47, 32, 8 47, 32, 22 47, 33, 13 47, 34, 43 47, 39, 12 47, 39, 28 47, 41, 44 47, 42, 17 47, 42, 42 47, 43, 10 47, 43, 16 47, 44, 19 47, 44, 33 47, 45, 5 47, 45, 31 47, 45, 37 47, 47, 3 48, 1, 29 48, 1, 42 48, 2, 1 48, 4, 27 48, 4, 48 48, 6, 26 48, 7, 20 48, 9, 15 48, 9, 37 48, 11, 4 48, 11, 22 48, 12, 11 48, 12, 28 48, 12, 33 48, 13, 7 48, 15, 31 48, 16, 46 48, 17, 2 48, 17, 6 48, 18, 35 48, 19, 30 48, 19, 36 48, 19, 39 48, 19, 41 48, 20, 9 48, 20, 34 48, 21, 13 48, 25, 8 48, 26, 17	99, 58, 7 99, 58, 46 99, 58, 51 99, 59, 5 99, 59, 57 99, 60, 69 99, 60, 72 99, 64, 23 99, 65, 17 99, 65, 91 99, 67, 44 99, 68, 45 99, 69, 39 99, 69, 54 99, 71, 4 99, 71, 11 99, 71, 55 99, 74, 15 99, 77, 22 99, 78, 85 99, 79, 95 99, 80, 26 99, 80, 67 99, 81, 89 99, 82, 38 99, 83, 76 99, 84, 65 99, 84, 94 99, 85, 10 99, 85, 56 99, 86, 33 99, 87, 36 99, 87, 73 99, 89, 66 99, 93, 37 99, 93, 83 99, 94, 48 99, 95, 77 99, 97, 21 99, 97, 24 99, 97, 32 100, 2, 1 100, 2, 40 100, 3, 57 100, 5, 8 100, 6, 19 100, 7, 50

利用循环输入，原代码输出为Y,均正确符合预期

7 拓展延申-- 三维模型建立

7.1 探求（x，y，n）组合之间的本质关系

为了展示（x，y，n）组合之间的本质关系，三维图形可视化模型的建立时很直观的方法，通过绘制三维散点图，我们能够更直观地看到哪些（x，y，n）组合能够满足约瑟夫问题的条件（即最后剩下的节点为1）

7.2 实现三维可视化的步骤

7.2.1导入需要的库：

用 matplotlib 进行三维绘图。

从 mpl_toolkits.mplot3d 中导入 Axes3D 以支持三维图形的绘制。

7.2.2. 数据准备： 利用6.5部分代码生成大量点集。

7.2.3. 定义绘图函数 ： 并且创建 3D 坐标系。

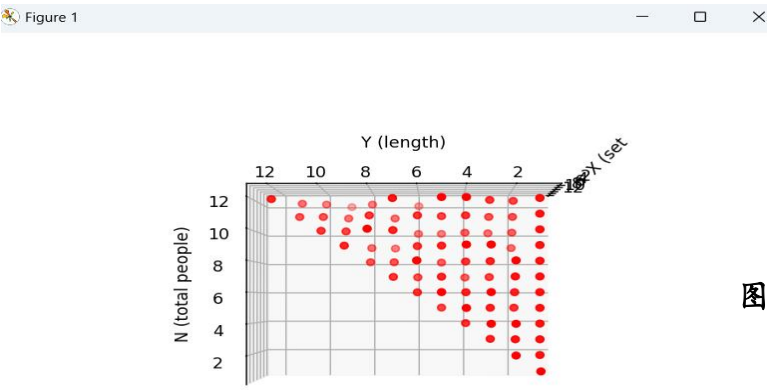
7.2.4. 绘制三维散点图->设置坐标轴标签->显示效果

7.2.5. 代码如下：（省略主函数）

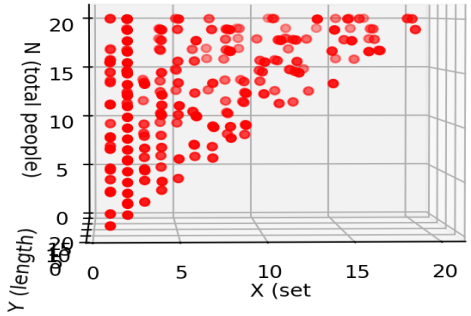
```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D


# 约瑟夫问题的模拟函数
def josephus(n, x, y):
    people = list(range(1, n + 1))
    index = x - 1  # 将 x 转换为 0 索引

    while len(people) > 1:
        index = (index + y - 1) % len(people)
        people.pop(index)

    return people[0]


# 查找满足条件的 (x, y, n) 组合
def find_valid_combinations(max_n):
    valid_combinations = []
    for n in range(1, max_n + 1):
        for x in range(1, n + 1):
            for y in range(1, n + 1):
                if josephus(n, x, y) == 1:
                    valid_combinations.append((x, y, n))
    return valid_combinations


# 绘制 3D 图形
def plot_combinations(combinations):
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')

    # 分别提取 x, y, z 坐标
    xs = [combo[0] for combo in combinations]
    ys = [combo[1] for combo in combinations]
    zs = [combo[2] for combo in combinations]

    # 绘制散点图
    ax.scatter(xs, ys, zs, c='r', marker='o')

    # 设置坐标轴标签
    ax.set_xlabel('X (set')
    ax.set_ylabel('Y (length)')
    ax.set_zlabel('N (total people)')

    plt.show(block=True)
```

7.3 实现三维模型展示

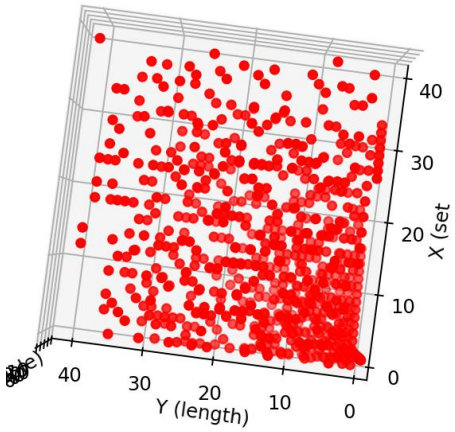
多角度，多参数设定观察图如下：



图一：n=10时，yOn面展示（x轴朝向读者）



图二：n=20时，xOn面展示（z轴朝向读者）



图三：n=40时，xOy面展示（z轴朝向读者）

7.4 本质理解和结论总结

7.4.1理解一

1.如图一，固定 x，变化 y 和 n：线性关系

我们可以这样理解：对于每个固定的 x，不同的步长 y 和人数 n 会影响删除顺序，但随着 y 增加，删除的位置会按一个相对线性的规律增长。

当 n 和 y 线性变化时，最终的删除结果也会呈现类似线性分布。

2如图2，固定 y，变化 x 和 n：非线性关系（接近类线性）

我们可以这样理解：在这种情况下，步长 y 固定，而起始位置 x 影响的是删除的第一轮顺序，之后每轮删除都会引入复杂的序号调整。因此，随着 n 变化，删除顺序也会变得复杂，呈现出一种非线性关系。

3.如图3，固定 n，变化 x 和 y：线性增长和非线性增长的混合

我们可以这样理解：在这种情况下，x 和 y 的变化共同影响每一轮删除的顺序。虽然 y（步长）的变化可能带来线性增长的趋势，但 x 作为起始位置的不同，影响了删除开始的位置，因此也会导致删除顺序的非线性调整

7.4.2另一种理解

1.固定 x 和n，变化 y：周期性变化
2固定 y和 n，变化 x：直线分布
3.固定 n，变化 x 和 y：非线性增长

7.4.3结论总结

三者关系本质：

n **代表规模**：n 的大小决定了问题的复杂度，即环的规模，影响每轮删除的序号调整。
x **代表初始位置**：x 的变化在固定 n 和 y 时影响简单，结果线性增加。
y **代表删除步长**：y 的变化会对每轮删除的位置产生复杂影响，使结果呈现周期性或非线性变化。

8 个人总结

忙忙碌碌好几天过去了，完成本次作业的成就感油然而生，但是我页也遇到了些许困难。
第一关--编程关：平日老是手写代码的我突然上机编程，最开始很不适应，以后一定要多写多练。
第二关--文档关：这是我第一次自己写测试样例，第一次完整写文档，这次经历让我成长许多。
第三关--建模关：想使用3D效果图，但是最后我恼火一整天才搞明白我的pycharm 中的plt.show(block=True) 括号里的东西必不可少，修改后顺利解决所有问题。

9致谢

最后，感谢我的老师和助教，你们辛苦啦！