Правительство Российской Федерации
Федеральное государственное автономное образовательное учреждение
высшего профессионального образования
**"Национальный исследовательский университет
Высшая школа экономики"**
Департамент прикладной математики, бакалавр

Работу на тему:

Моделирование столкновения дисков внутри замкнутой поверхности, обадающей свойством замкнуости.

Выполнил:
Колодин Матвей Алексеевич

Академический руководитель:
Щур Лев Николаевич

Москва 2022

Содержание

Часть I
Постановка задачи и ее решение

1 Введение

1.1 Постановка задачи

Мы будем представлять тор, как его развертку на плоскость - в частном случае квадрат некоторых размеров. Условие задачи:

Пусть дано вещественное число L, которое определяет линейные размеры квадрата. Вещественное число R отвечает за радиус диска, а N - является количеством дисков, помещенных в данный квадрат. А также велична $\eta = \frac{N*\Pi*R^2}{L^2}$, которая находится в промежутке $0.5 < \eta < 1$.

Из коробки выбирается случайный диск и начинает свое движение, параллельно основанию коробки. Если на его пути встречается другой диск, то происходит столкновение, в результате чего движущийся диск останавливается, а тот диск, с которым он столкнулся продолжает движение. Если диск достигает границы квадрата, то он начинает «появляться» из начала коробки (в этом и заключается свойство тора, которое применяется в задаче).

Движение заканчивается в тот момент, когда суммарное расстояние, пройденное дисками становится равно l.

После чего данный алгоритм повторяется по оси, направленной параллельно боковой стороне коробки.

1.2 Уточнения по условию задачи

1. Случай столкновения с двумя дисками. Давайте условимся считать, что в случае, когда движущийся диск сталкивается с двумя дисками одновременно, движние продолжает любой из них.

2. Случай, если расстояние от центра диска до границы $< R$. В таком случае, будем визуализировать картинку, опираясь на свойства тора, о которых мы сказали выше, таким образом:

3. Поскольку вещественные числа представлены в компьютере с некоторой точностью, то необходимо аккуратно работать с математическими операциями. Для этого введем константу ε, применение которой будет показано далее.

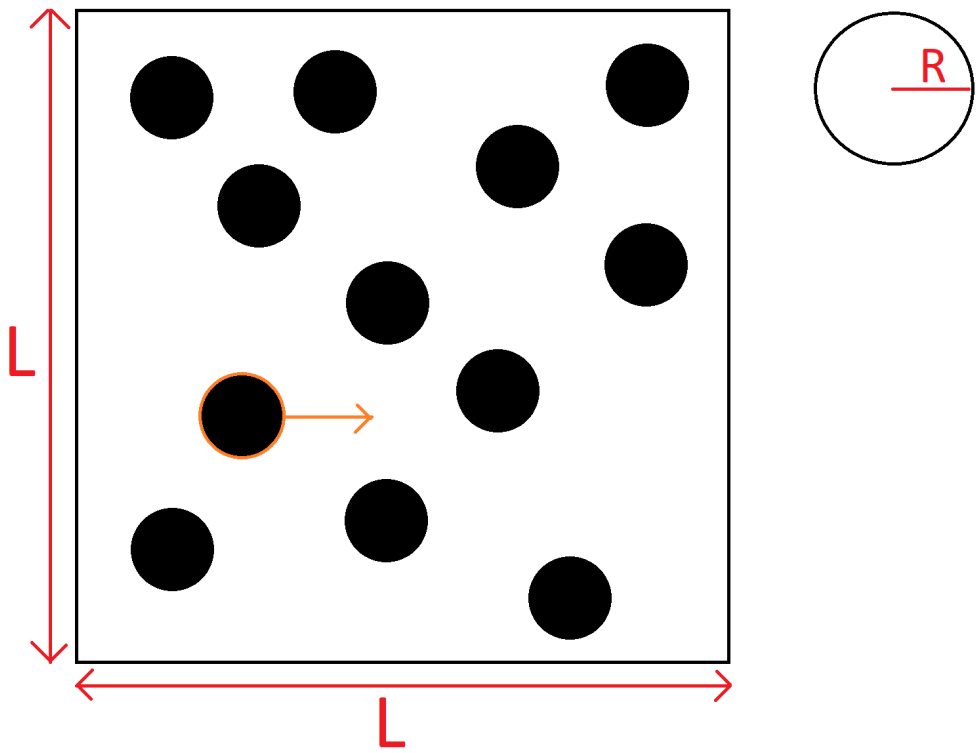

Рис. 1: Пояснительный рисунок к условию задачи.

2 Алгоритм решения задачи

2.1 Движение диска параллельно основанию квадрата (O_x). Поиск следующего диска в цепочке столкновений

Данную подзадачу разобьем на два пункта - поиск диска, с которым произойдет следующее столкновение, и вычисление промежуточных положений центров дисков.

Поиск следующего диска в цепочке столкновений. Рассмотрим, где может находится диск, с которым возможно столкнуться. Если разность координат центров произволльного диска и выбранного по O_y отличается не более, чем на $2R$, то такие диски столкнутся, в противном случае, при движении параллельно O_x диски столкнуться не могут впринципе.

Мы получили список дисков, с которыми мы можем столкнуться, теперь необходимо понять, что произойдет столкновение с тем диском, который ближе всего находится в синему, относительно O_x, и при этом, лежащему правее него:

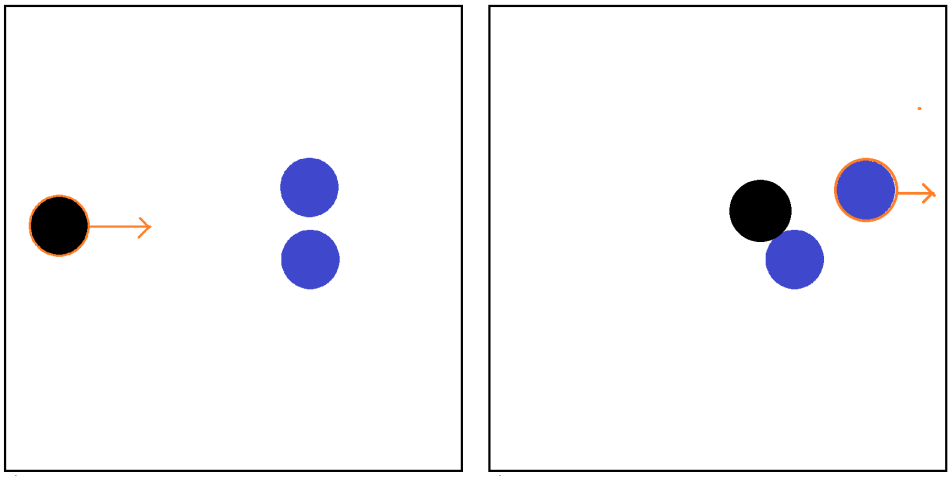

Рис. 2: Случай столкновения с двумя дисками.

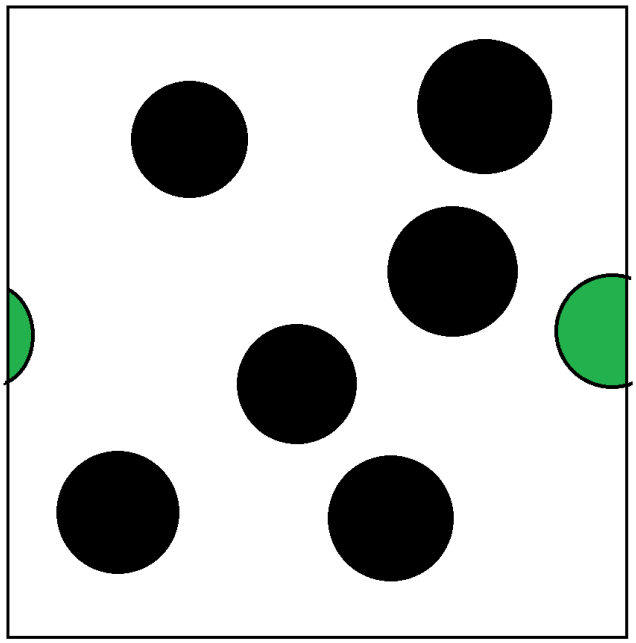

Рис. 3: Случай нахождения диска на границе.

Таким образом можем получить следующий код (язык - Python + библиотека NumPy):

```
def FNB(x, y, arr, r):
    eps = 1e-8
    minn = 1e+9
    pos = -1
    for i in range(len(arr)):
        if arr[i][1] > y + 2 * r - eps:
            break
        elif arr[i][1] > y - 2 * r + eps:
            if minn - eps > arr[i][0] > x + eps:
                minn = arr[i][0]
```

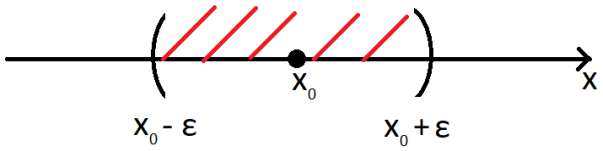

Рис. 4: Проблема с точностью при вычислении одних и тех же выражений. Полученное значение будет лежать в некоторой эпсилон-окрестности.

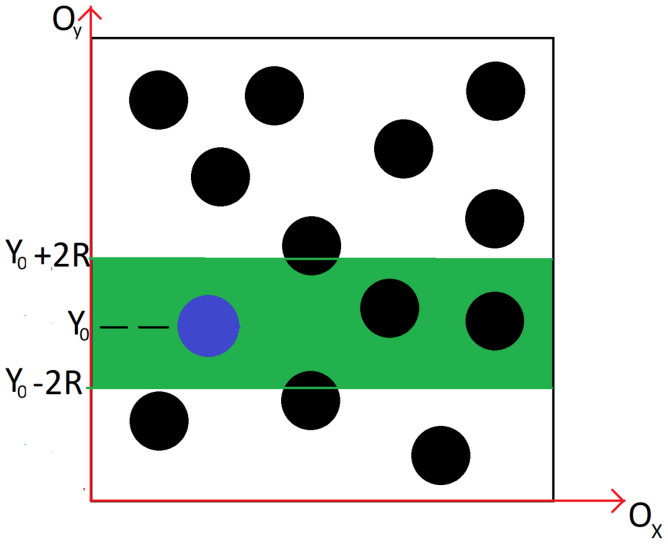

Рис. 5: диски, центр которых (относительного выбранного синего диска) находится на расстоянии меньше $2R$ подойдут. На данной картинке подойдет всего два диска.

```
                pos = i
    return pos
```

В алгоритме ищем диски, которые будут лежать в полосе (по O_y) от $y - 2 * r$ до $y + 2 * r$. Из полученных дисков мы выбираем тот, который будет находиться наиболее близко к выбранному дискуу, и при этом находиться правее него. Таким образом, будет получен следующий диск в цепочке столкновений.

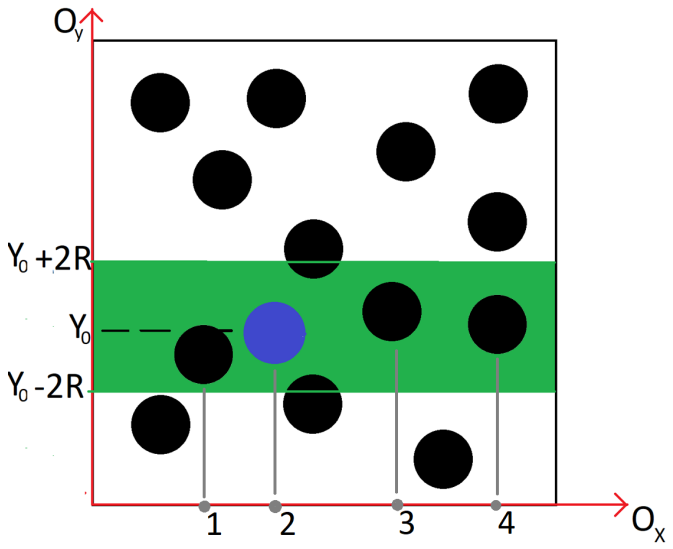

Рис. 6: Можно видеть, что первый диск лежит ближе ко второму, но он находится левее него, таким образом он не подойдет. Остается диск под номером 3.

2.2 Движение диска параллельно основанию квадрата. Моделирование движения

По большому счету, моделирование движения дисков сводится к разбору ряда случаев. Но перед тем, как перейти к разбору вариантов, уточним одну вещь: мы будем создавать некоторый буфер (ширины $2R$, а высотой будет высота квадрата). Этот буфер присоединим в конец квадрата. В нем будет отображено начало квадрата (прямоугольник размера $n * 2R$. Данная процедура будет необходима для удобной работы при моделировании движения дисков.

Для лучшего понимания, будем размечать еще одну зону, симметричную относительно стенки квадрата буферной зоне. Цветом выделены те диски, которые начали процесс перетекания, то есть коснулись стенки квадрата. Такое представление необходимо для того, чтобы точно понимать, когда диск закончил перемещение через границу и оказался в начале квадрата (такое визуальное представление облегчает восприятие. Также часть кода основана на принципе аккуратной работы с буфером.)

В результате чего остается аккуратно разобрать случаи, при движении диска - в частности можно выделить две большие группы ситуаций по столкновению с одним из объектов: с другим диском или стенкой квадрата.

Функция FNB позволяет находить ближайший диск. Теперь нужно научиться моделировать движение, до столкновения с другим диском. Таким образом стоит задача о перемещении диска, до момента касания с другим диском (диском столкновения).

Пусть есть два диска с центрами G (имеет координаты (x_1, y_1)) и E (имеет координаты (x_2, y_2)) соответственно. Очевидно, что центр диска G движется по прямой $y = y_1$. Столкновение произойдет когда расстояние между их центрами станет равно $2R$. Таким образом, чтобы диски столкнулись, необходимо, чтобы центр диска G находился на окружности с центром в точке E и радиусом $2R$.

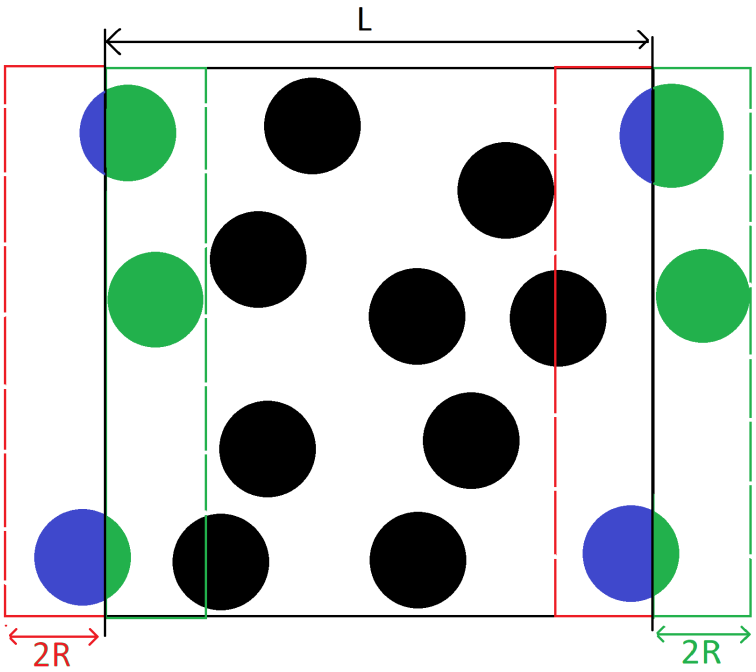

Рис. 7: Визуализация буфера для квадрата.

Пусть при движении в точку столкновения центр диска G изменил значения своих координат с (x_1, y_1) на (x_{new}, y_{new}). Тогда можно сказать следующее:

$$\begin{cases} y_{new} = y_1 \\ (x - x_2)^2 + (y - y_2)^2 = (2R)^2 \end{cases}$$

Поскольку значение y_{new} мы знаем, то чтобы получить x_{new} остается решить квадратное уравнение. Тогда получаем: $x = \pm((4R^2 - (y_1 - y_2)^2)^{\frac{1}{2}}) + x_2$. А по скольку движение идет слева от диска E, то из полученых корней необходимо брать меньшее значение. Таким образом: $x_{new} = -((4R^2 - (y_1 - y_2)^2)^{\frac{1}{2}}) + x_2$. А значит новые координаты центра диска G были получены.

Объвим функцию CCB, которая и будет обрабатывать процесс движения дисков:

```
def CCB(x1, y1, x2, y2, r):
    x_new = -(((2 * r) ** 2 - (y1 - y2) ** 2) ** (1 / 2)) + x2
    l_pr = x_new - x1
    return [x_new, l_pr]
```

Она будет возрващать новую координату по O_x и расстояние, которое диск пройдет до столкновения.

Теперь же перейдем к функции, которая и будет просчитывать движение всех дисков:

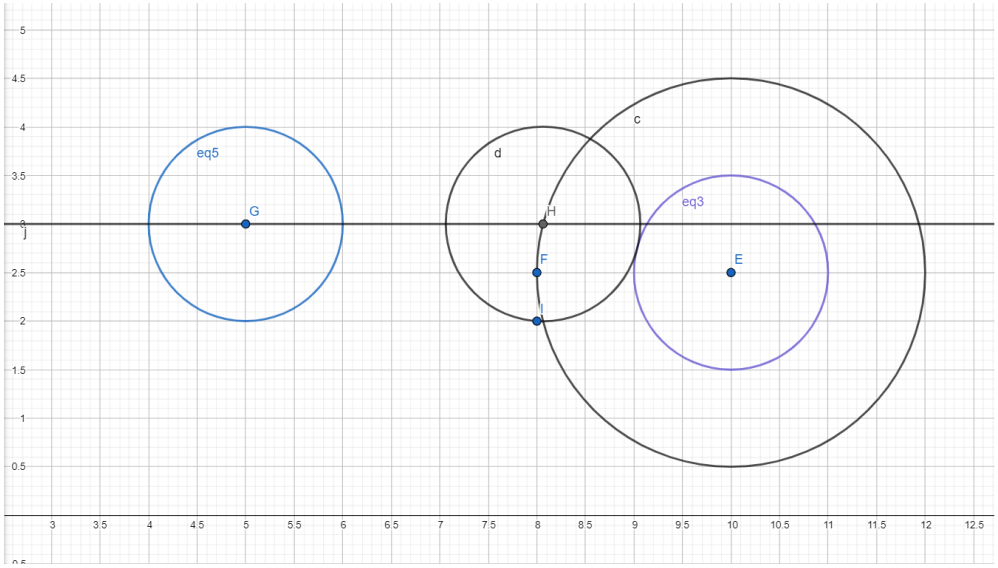

Рис. 8: Моделирование столкновения двух дисков.

```
def MOB_x(ind_sh, arr, r, l_ost, l_box):

# Прежде всего, необходимо понять, находится ли диск в буферной зоне
или нет.
eps = 1e-8
x = arr[ind_sh][0]
y = arr[ind_sh][1]

if x < l_box - r - eps:
    # Мы находимся внутри коробки.
    pos = FNB(x, y, arr, r)

    if pos == -1:
        # Значит дисков, с которыми можно столкнуться внутри
        коробки нет.
        if l_ost > (l_box - r) - x + eps:
            l_ost -= (l_box - r) - x
            arr[ind_sh][0] = l_box - r
            return MOB_x(ind_sh, arr, r, l_ost, l_box)
        else:
            arr[ind_sh][0] += l_ost
            return arr

    else:
        # Значит есть диск, с которым можно столкнуться.
        x_new, l_pr = CCB(x, y, arr[pos][0], arr[pos][1], r)
        if l_ost > l_pr + eps:
            l_ost -= l_pr
            arr[ind_sh][0] += l_pr
```

```
            return MOB_x(pos, arr, r, l_ost, l_box)
        else:
            arr[ind_sh][0] += l_ost
            return arr

else:
    # Мы находимся внутри буферной зоны или на ее границе.
    # Важным уточнением является то, что мы ввели буферную зону, но
    не заполняли ее. Поэтому важна отдельная обработка поиска дисков
    для начала коробки, так как в начальный момент, например,
    в буфере ничего нет, но вот в начале диск может быть, это
    необходимо учитывать.
    pos_buf = FNB(x, y, arr, r)
    pos_strt = FNB(x - l_box, y, arr, r)

    # Для начала проверим, есть ли диски, с которыми мы можем
    столкнуться в буферной зоне.
    if pos_buf != -1:
        # Таким образом, есть диск, с которым мы столкнемся в
        буферной зоне.
        pos = pos_buf
        x_new, l_pr = CCB(x, y, arr[pos][0], arr[pos][1], r)
        if l_ost > l_pr + eps:
            l_ost -= l_pr
            arr[ind_sh][0] += l_pr
            return MOB_x(pos, arr, r, l_ost, l_box)
        else:
            arr[ind_sh][0] += l_ost
            return arr

    else:
        # То есть буферная полоса пустая. По крайней мере, если
        "смотреть" с конца, давайте рассмотрим ситуацию
        от левого края коробки (левой буферной зоны).
        if pos_strt != -1 and
        np.array_equal(arr[pos_strt], np.array(x, y)) == False:
            # Значит мы нашли диск, с которым мы столкнемся, причем
            он лежит внутри коробки, иначе мы бы нашли его при
            просмотре буферной зоны.
            # Здесь необходимо работать аккуратно и, в случае
            если диск прошел буферную зону перевести его в
            обычную зону.
            pos = pos_strt
            x_new, l_pr = CCB(x - l_box, y, arr[pos][0], arr[pos][1], r)
```

```
            if l_ost > l_pr + eps:
                l_ost -= l_pr
                arr[ind_sh][0] += l_pr
                if arr[ind_sh][0] > (l_box + r) - eps:
                    arr[ind_sh][0] -= (l_box + r)
                return MOB_x(pos, arr, r, l_ost, l_box)
            else:
                arr[ind_sh][0] += l_ost
                if arr[ind_sh][0] > (l_box + r) - eps:
                    arr[ind_sh][0] -= (l_box + r)
                return arr

        else:
            # Значит дисков для столкновения, тогда двигаем настолько,
            насколько можем.
            arr[ind_sh][0] += l_ost
            if arr[ind_sh][0] >= (l_box + r) - eps:
                arr[ind_sh][0] -= (l_box + r)
            return arr
```

P.S. В конце отчета будет приведен итоговый код без комментариев, для удобства.

Таким образом, была получена функция, которая моделирует движение дисков, параллельно O_x.

2.3 Движение диска параллельно высоте квадрата (O_y)

В действительности же, моделирование, при движении параллельно высоте квадрата ничем не отличается от движения параллельно его основанию. Таким образом, код для данного случая уже получен (остается лишь в полученные функции передать "перевернутый массив"(и отсортированный), то есть заменить координаты: x на y. И после того, как мы прогоним массив, еще раз его перевернуть, тем самым мы вернемся в систему O_x O_y).

2.4 Нормализация координат и вывод результатов

Итоговый алгоритм будет выглядеть следующим образом:

```
a = np.array([[1,1], [4,1], [10, 2.5], [3,3], [5,3], [1,4], [3,5]])
a = a[np.argsort(a[:, 1])]
shx = 0
shy = 1

a_res_x = MOB_x(shx, a, 1, 7, 11)
copy_x = a_res_x.copy()
```

```
for i in range(len(a_res_x)):
    a_res_x[i] = np.array([a_res_x[i][1], a_res_x[i][0]])
a_res_x = a_res_x[np.argsort(a_res_x[:, 1])]


a_res_xy = MOB_x(shy, a_res_x, 1, 5, 11)
for i in range(len(a_res_xy)):
    a_res_xy[i] = np.array([a_res_xy[i][1], a_res_xy[i][0]])

a_res_x = copy_x
```

В итоге мы получим два массива: a_res_x - который содержит новые координаты центра дисков после движения параллельно O_x, а также a_res_xy - который содержит новые координаты центра дисков после движения параллельно O_x и O_y.

Для вывода начального, промежуточного и конечного положения дисков я буду пользоваться библиотекой Matplotlib:

```
import matplotlib.pyplot as plt
%matplotlib inline
l_box = 11
r = 1
a = np.array([[1,1], [4,1], [10, 2.5], [3,3], [5,3], [1,4], [3,5]])
a = a[np.argsort(a[:, 1])]

fig, (ax1, ax2, ax3) = plt.subplots(nrows=3, ncols=1, figsize=(15,15))

ax1.set(xlim=(0, l_box), ylim=(0, l_box))
for i in range(len(a)):
    circle = plt.Circle((a[i][0] % l_box, a[i][1] % l_box), r,
    color='b')
    ax1.add_patch(circle)
ax1.set_aspect('equal')

ax2.set(xlim=(0, l_box), ylim=(0, l_box))
for i in range(len(a_res_x)):
    circle = plt.Circle((a_res_x[i][0] % l_box, a_res_x[i][1] % l_box),
    r, color='b')
    ax2.add_patch(circle)
ax2.set_aspect('equal')

ax3.set(xlim=(0, l_box), ylim=(0, l_box))
for i in range(len(a_res_xy)):
    circle = plt.Circle((a_res_xy[i][0] % l_box, a_res_xy[i][1] %
    l_box), r, color='b')
```

```
        ax3.add_patch(circle)
    ax3.set_aspect('equal')
```

При отрисовке дисков координаты берутся по модулю l_{box}, поскольку при работе алгоритма мы позволяли центрам дисков будто оказываться за границей, но на деле же, их центр всегда пренадлежит квадрату.

3 Результаты

В качестве результатов рассмотрим один пример движения дисков:

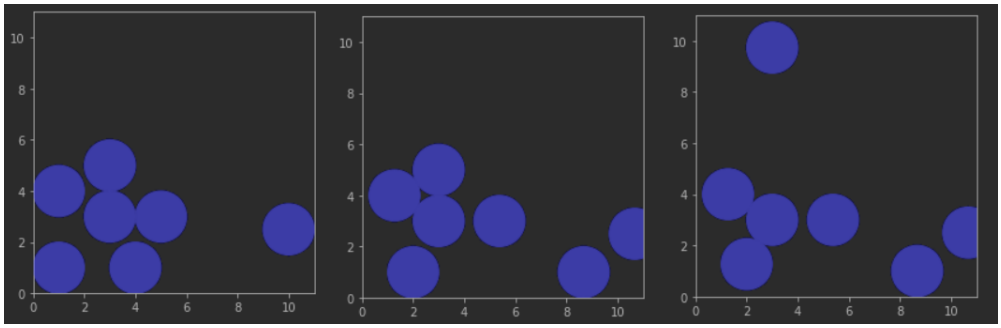

Рис. 9: Пример работы алгоритма

3.1 Приложение

Ссылка на проект: https://github.com/HpPpL/Collision-of-balls.

Часть II
Отчеты

4 Отчет №1

Выполненые задачи:

4.0.1 Уточнение отрисовки результатов

Было выполнено улучшение по отрисовке дисков.

Проблема с отображением частей дисков была вызвана не рассмотрением всех случаев, и попыткой упростить задачу, что оказалось фатальной ошибкой.

В последующем я разбил область квадрата на 9 частей и для каждого диск, находящегося в той или иной зоне, теперь существует определенная, корректная отрисовка.

Красным обозначены диски, которые находятся в стадии перетекания, синим же диски, которые этого действия не выполняют.

Пример сравнения эффективности работы отрисовки первой и второй версии:

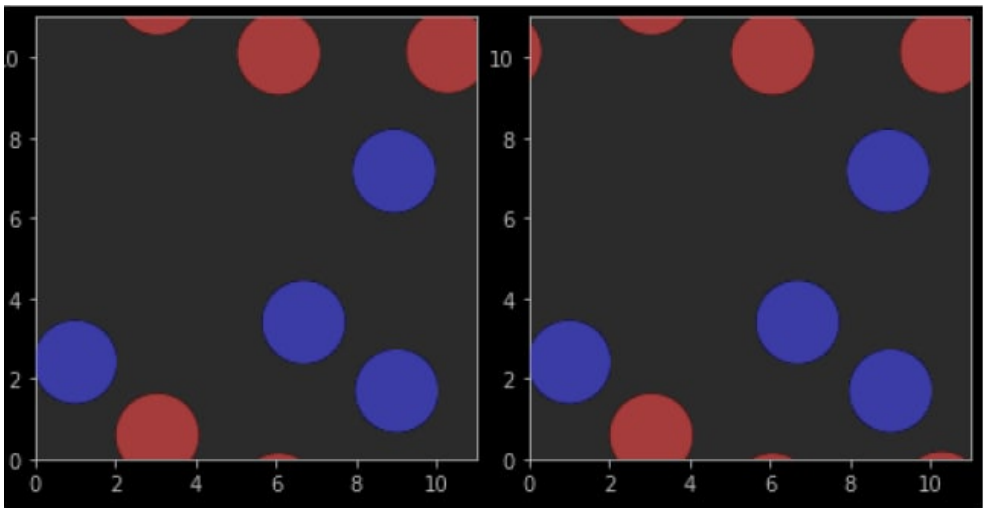

Рис. 10: Качественные отличия работы версий.

4.0.2 Сохранение картинок и результат в виде gif-анимации

Теперь промежуточные этапы сохраняются в формате png, и из них в последующем создается gif-анимация, отображающая все шаги:

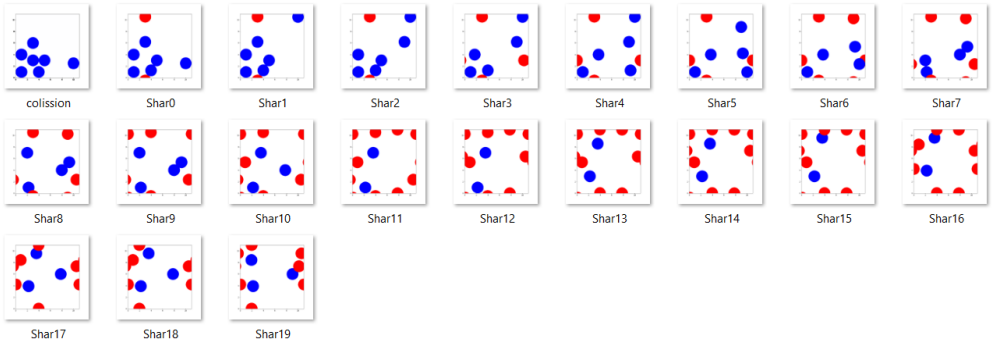

Рис. 11: Папка с результатами.

4.1 Выявленные ньюансы

4.1.1 Доработка функции FNB

Функция поиска следующего диска требует некоторой доработки, поскольку при поиске, который совершается достаточно близко к границе коробки (центр диска находится на расстоянии меньше, чем $L - 2R$ до верхней или нижней границы), в силу особенности хранения и работы функции следующие случаи обработаны правильно не будут:

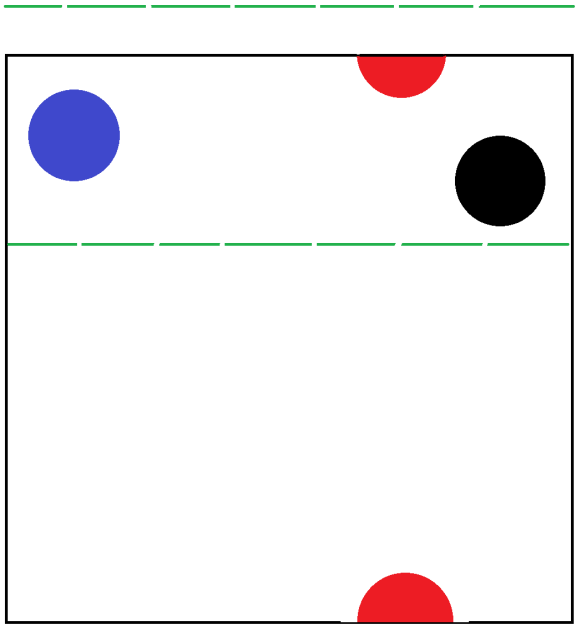

Рис. 12: При движении синего диска, центр красного диска находится снизу, а поэтому его часть, которая находится сверху, распознана при поиске следующего диска не будет. И будет столкновение с черным диском.

4.1.2 Некорректность рассуждений при составлении функции FNB

Было невереное предположение о том, что в полосе следующим диском в цепочке столкновений будет диск, у которого меньшая координата по O_x. Поэтому нужно пользоваться функцией CCB, при этом, я постараюсь оптимизировать

этот процесс, чтобы вычисляли расстояние не до всех дисков, в данный момент уже есть некоторые идеи.

Пример, когда рассуждение некорректно:

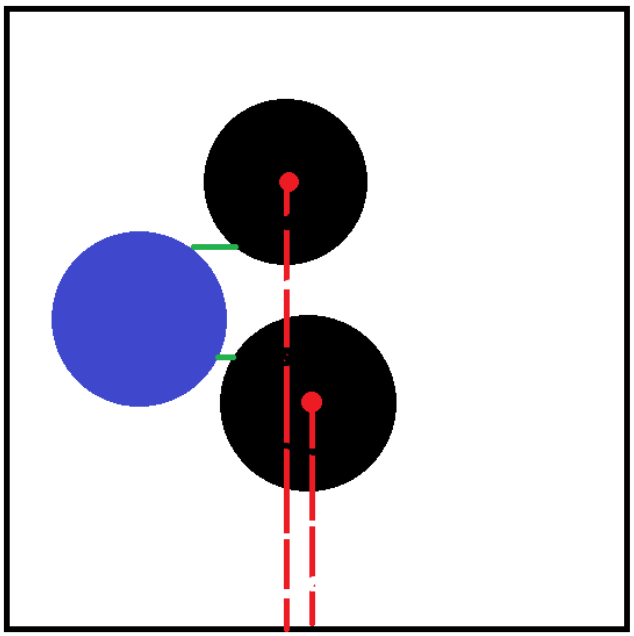

Рис. 13: Верхний черный диск находится ближе по O_x, но столкновение произойдет с нижним диском.

5 Отчет 2

5.1 Выполненые задачи

5.1.1 Уточнение функции FNB. Поиск ближайшего диска

В связи с некорректностью предположения о том, что ближайший диск в цепочке столкновений - диск с наименьшей координатой по 0_x, было пересмотрен алгоритм функции FNB.

В результате чего появилось две идеи о поиске ближайшего диска:

1. Брать в лоб диски в полосе и искать с минимальным расстоянием. (Что и было реализовано).

2. Есть интересная идея, которая поможет снизить вычислительную мощность, которая можно в последующем реализовать.

   Суть заключается в том, что если есть выбранный диск, и диск в полосе, то столкновение, если и может произойти, то оно обязанно быть в левой части предполагаемого диска, то есть: пусть движется синий диск, и он должен столкнуться с красно-зеленым, тогда столкновение должно произойти в его левой части (окрашена в салатовый цвет) и получается, что диски, с которыми мы можем потенциально находятся в части, которая окрашена в светло-зеленый цвет:

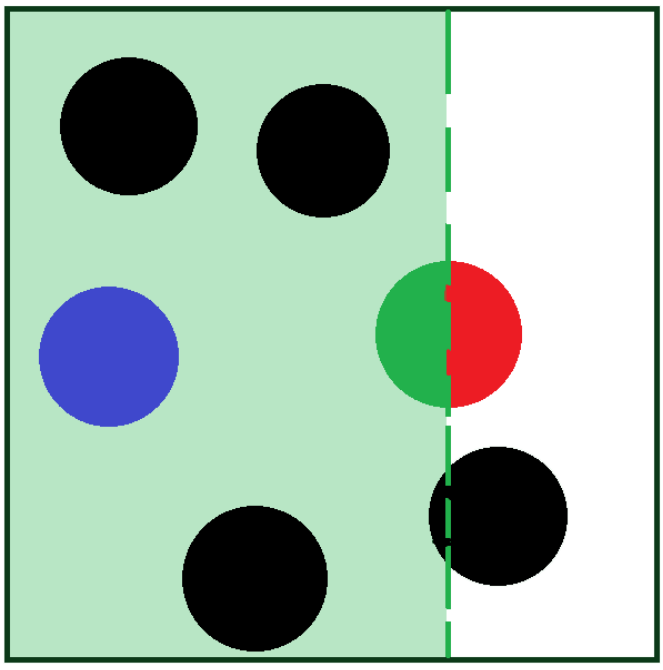

Рис. 14: Графическое пояснение к высказыванию

Таким образом видно два решения сложившийся проблемы, возможно, в будущем появятся и другие.

5.1.2 Уточнение функции FNB. Рамка поиска

Из-за особенностей храненения, в некоторых ситуациях, когда, например, центр диска находится на нижней границе коробки, не учитывалось то, что сверху выпирает ее часть, то есть:

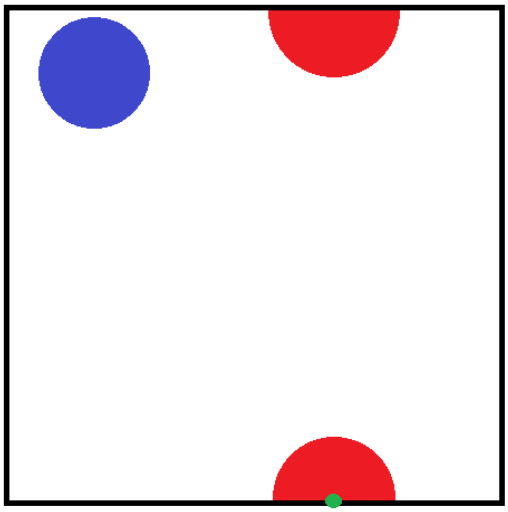

Рис. 15: Центр красного диска находится в зеленой точке (снизу). В таких случаях синий диск, ранее, не видел половины красного диска, которая находится снизу.

Проблема была решена рассмотрением трех областей:

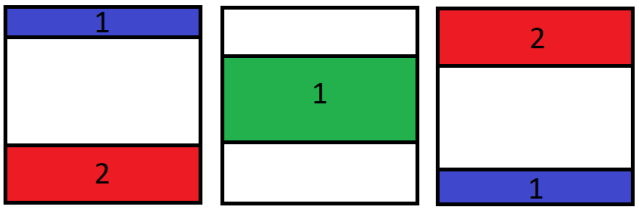

Рис. 16: Разбиение на области.

5.2 Выявленные ньюансы

5.2.1 Проблема при создании gif-картинок

Возникает проблема с созданием гифки из картинок (пропуск кадров), необходимо будет либо найти проблему, либо выбрать другое средство для построения анимированных изображений

6 Отчет 3

6.1 Выполненые задачи

Исправления будут примерно в порядке возрастания их значимости, поскольку их очень много, то трудно сказать будет точно, что важнее, а что нет.

Можно сказать, что по большей части, код был переосмыслен, переписан и исправлен. Оптимизация, исправления и улучшения - девиз этого отчета.

Думаю, что можно переходить к самим исправлениям.

6.1.1 Реоформление git'а проекта

Проект на Github'е был немного переконструирован, убраны ненужные папки, сам код был обновлен. Прочие изменения также можно посмотреть по ссылке проекта: https://github.com/HpPpL/Collision-of-balls.

6.1.2 Исправление оформления кода

Был исправлен code style, улучшено структурирование notebook - файла, в частности - увеличено количество коментариев, появились заголовки к разделам программы. Большая часть переменных вынесена в отдельную ячейку.

Логическая последовательность программы также была улучшена - теперь более точно и логически правильно изложен код. В частности, порядок теперь следующий: идет описание функций, инициализация переменных, а после ячейка, в которой происходит запуск всех необходимых функций, для вычисления столкновения дисков. В конце оставлено все дополнительное, не совсем относящиеся к теме выполнения задачи (перевод Python-кода на псевдоязык).

6.1.3 Исправление анимации столкновений дисков

Эта проблема тянулась довольно долго, и поначалу казалось вовсе странным ее существование - проблема заключалась в том, что картинка в один момент начинала строиться не так - где-то пропускала кадры, где-то перемешивала, хотя покадровое разбиение (из которого строится сама gif-картинка) было описано и отрисовано правильно, была последовательность в названии файлов - "Shar0 "Shar1"...

В один момент произошло предположение, что кадры берутся по какой-то причине в неправильном порядке. В последствии так и оказалось - после вывода списка файлов в директории было замечено, что файлы уложены не совсем в алфавитном порядке. Выглядело это примерно так: "Shar0 "Shar1 ... ,"Shar9 "Shar10 "Shar20 "Shar30"... Встроенная сортировка Python допускала такой момент, поэтому я начал пользоваться библиотекой $Natsort$. Это помогло решить проблему с анимацией.

Также была проблема с индексацией - начальное положение не было отражено в анимации столкновений, это также было исправлено.

6.1.4 Изменение выбора диска

Ранее выбор диска с последующим движением был не верным. Приведу поясняющую картинку как было раньше, и как есть сейчас:

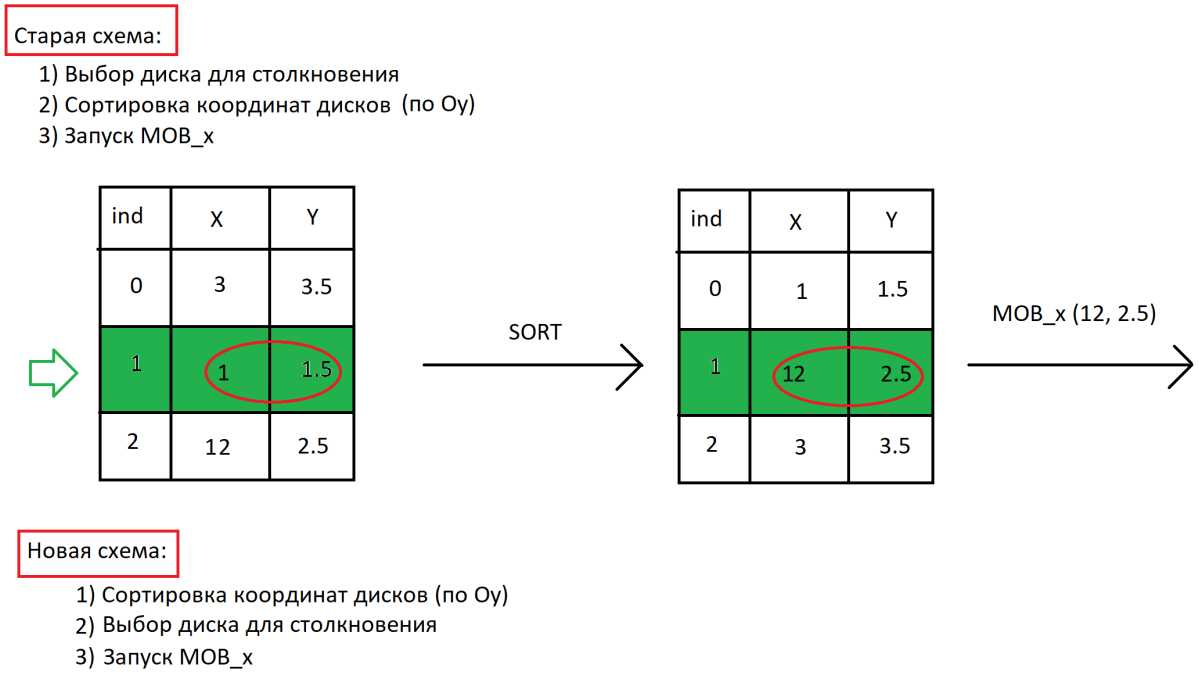

Рис. 17: Сравнение новой и старой версии работы алгоритма после выбора диска.

Ошибка заключалась в том, что в старой схеме столкновения происходили не для того диска. К тому же, происходила сортировка каждый раз при запуске MOB_x, что в самом деле является избыточным, поскольку существует инвариант - диски не меняют своей координаты по O_y, а именно по ней и происходит сортировка, поэтому ее достаточно запустить только перед запуском MOB_x.

Далее будут идти более глобальные изменения, которые касаются **трех основных функций** - MOB_x (MOB_y), FNB, CCB.

6.1.5 Изменение функции CCB

По ходу отладки программы выяснялось, что не смотря на страховку с погрешностью, со временем ошибка может все равно накопиться. В частности, из-за этой проблемы не работал CCB, поскольку вычислял корень из отрицательного числа (но по какой-то причине это ранее не показывалось, и пришлось хорошо подумать, прежде чем я понял это).

Возникала эта проблема в двух случаях - первый был завязан на работе старой функции FNB, поэтому отдельно правилась она, но об этом подробнее далее, и также была ошибка связанная с погрешностью, например, два диска, у которых координаты были равны: (x, 2), (y, 4.0000000000001), с радиусом r = 1, скорее всего, должны были коснуться друг друга, но при вычислении

выражения $\sqrt[2]{(2r)^2 - (y_1 - y_2)^2}$ под корнем получался не ноль, а какое-то очень маленькое (порядка 10^{-10}, например) отрицательное число.

Чтобы решить данную проблему я сделал следующее: если получилась так, что разность отрицательная, тогда запускаем проверку - из модуля разности координат по O_y (для двух дисков) вычесть $2r$. Если выражение будет меньше, чем некоторая величина eps, то считаем, что в таком случае происходит касание дисков, иначе выдаем исключение и выводим координаты дисков (это радикальный случай, который не должен происходить, но, тем не менее, будет возможность отследить проблему).

6.1.6 Изменение функции FNB

Не буду проводить сравнение со старой версией в деталях, лишь скажу что функция была переписана полностью, поскольку начальный вариант не выдерживал никакой критики, и по-хорошему говоря, основную работу выполняла функция MOB_x - разбор случаев, зон, вычисления, замены и т.д.

Теперь же функция FNB занимается корректным вычислением данных следующего диска в цепочке столкновений, в частности - его координат и позиции в массиве. Нет никаких ограничений, на то что мы сталкиваемся с частью диска или с целым диском, нужно ли совершить переход через стенку или нет, все это учтено.

Был долгий путь отладки, разбора вариантов, поиска ошибок, но финальный вариант кажется наиболее оптимальным из возможных. Почему нужно возвращать координаты и отдельно позицию в массиве, а также как это работает будет разобрано в основной части отчета - разборе кода.

6.1.7 Изменение функции MOB_x (MOB_y)

Данная функция была также сильно изменена и рационализирована. Вместо громоздко кода теперь реализовано три интуитивных ситуации, но про это все опять же более подробно будет сказано в разборе кода!

7 Пояснения к коду

В этой части отчета будет описан код в ячейках, с входными и выхолдными данными, а также, естественно, алгоритмическим смыслом написанных строк:

7.1 Импорт функций

В первой ячейке происходит подгрузка необходимых библиотек:

```
# Импортируем библиотеки для работы.
import numpy as np
import matplotlib.pyplot as plt
import os
from PIL import Image
from natsort import natsorted
import python2pseudocode as p2p
```

Рис. 18: Список необходимых библиотек.

Поясню кратко за предтсавленные библиотеки:

1. **numpy** - отвечает за работу с массивами, в частности благодаря нему удобна работа с координатами дисков.
2. **matplotlib** - графический модуль проекта.
3. **os** - позволяет работать с директориями и файлами компьютера более удобно, чем просто через встроенные возможности языка.
4. **PIL** - благодаря данной библиотеке можно строить gif-изображения из отдельных кадров.
5. **natsort** - сортирует контейнеры также, как делает Windows, потому что встроенная сортировка Python не подходит для выполнения задачи.
6. **python2pseudocode** - переводит Python-код в псевдокод.

7.2 Объявление функции LowerBound

Сразу оговорюсь, что на будущее было бы неплохо написать UpperBound, но вспомнил об этом только сейчас.

Для начала рассмотрим реализацию функции:

```
# LowerBound отвечает за поиск элемента, котор
def LowerBound(arr, key):
    left = -1
    right = len(arr)

    while right > left + 1:
        middle = (left + right) // 2
        if arr[middle][1] >= key:
            right = middle
        else:
            left = middle

    return right
```

Рис. 19: Функция LowerBound.

Задача функции состоит в том, чтобы найти значение в массиве (отсортированном) найти наименьшую возможную позицию, значение массива в которой будет $\geq$ **key**.

Поэтому на вход принимается отсортированный массив **arr** и значение для поиска - **key**. Внутри реализован классический бинарный поиск, поэтому в деталях не вижу смысла описывать суть происходящего.

На выходе мы получаем позицию в массиве, которая и будет указывать на наименьший элемент в массиве, который $\geq$ **key**.

7.3 Объявление функции FNB

Данная функция является одной из наиболее важных функций всего проекта - она позволяет найти следующий диск в цепочке столкновений.

Прежде чем переходить к ее детальному описанию, скажу небольшое предисловие! Говоря в целом, ранее работа в процессе движения дисков между функциями FNB и MOB_x (непосредственно функция сдвига дисков) распределялась как 20/80, сейчас же, я бы мог сказать, что это 70/30.

Так получилось, потому что был принципиально пересмотрен подход к решениию задачи и распределению ресурсов, в результате чего основные функции изменились координально.

Не вижу смысла построчно сравнивать с тем, что было, прошлый код можно перечитать в отчетах выше, поэтому сосредоточимся на том, что делает функция сейчас, это, как я считаю, является наиболее важным среди прочего.

Прежде всего определю еще раз, что делает данная функция - для выбраного диска из массива она находит следующий диск в цепочке столкновений.

Перейдем на более детальный уровень и рассмотрим код (он будет разбит на несколько частей + из 3 случаев будет показано лишь два, поскольку 3 идейно будет совпадать со вторым):

```
def FNB(x, y, arr, l_box, r):
    eps = 1e-8
    minn = 1e+12
    pos = -1
    x_res, y_res = 0, 0

    # Будет три случая - зона поиска в рамках коробки, зона поиска выходит снизу, зона поиска выходит с
    # Можно выделить особый случай - когда рамки поиска будут выходить за границы, но засчет разделения
      проблема невелируется.
    # Пока предположим, что такой ситуации быть не может.

    # 1 случай. Зона поиска внутри коробки.
    if y - 2 * r + eps >= 0 and y + 2 * r - eps <= l_box:

        for i in range(LowerBound(arr, y - 2 * r + eps), len(arr)):

            if arr[i][1] > y + 2 * r + eps:
                break

            elif x + eps <= arr[i][0] and CCB(x, y, arr[i][0], arr[i][1], r)[1] <= minn - eps:
                minn = CCB(x, y, arr[i][0], arr[i][1], r)[1]
                x_res, y_res = arr[i][0], arr[i][1]
                pos = i
            elif arr[i][0] <= x + eps and CCB(x, y, arr[i][0] + l_box, arr[i][1], r)[1] <= minn - eps:
                minn = CCB(x, y, arr[i][0] + l_box, arr[i][1], r)[1]
                x_res, y_res = arr[i][0] + l_box, arr[i][1]
                pos = i
```

Рис. 20: Функция FNB. 1 случай.

При реализации финального варианта FNB был использован подход, который был упомянут в предыдущих отчетах - разбиение поиска на 3 глобальных ситуации:

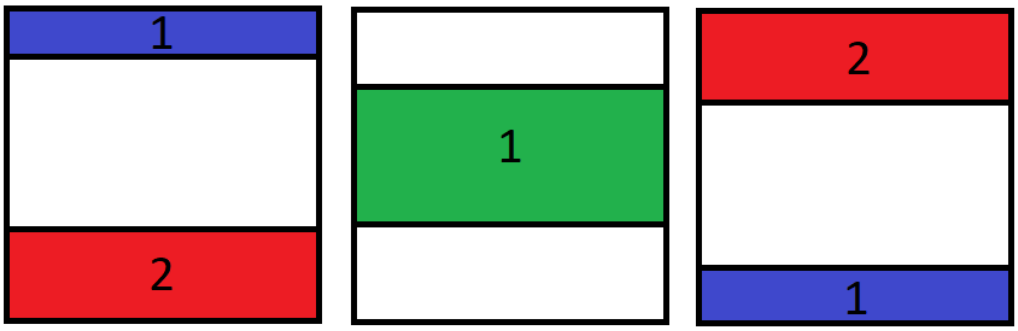

Рис. 21: Функция FNB. Графическое пояснение.

7.3.1 Первый случай

Первый случай отвечает рисунку в центре - когда диск, с которым может пройзойти столкновение "не перетекает"через **верхнюю или нижнюю** грань.

Определение случая происходит в строке 15, там буквально и описано, что верхняя грань поиска не превышает верхнюю грань коробки, а нижняя грань поиска не ниже нижней грани коробки.

Внутри нам впервые понадобится функция LowerBound - для того, чтобы найти диск, координата которого по O_y будет наименьшая из возможных, и при этом $\geq y - 2 * r$, где y - координата по O_y для диска, который был передан в функцию для поиска следующего диска в цепочке столкновений, необходимо воспользоваться LowerBound. Таким образом, засчет того, что массив отсортирован по O_y нам достаточно подряд перебирать диски до тех пор, пока координата по O_y не станет $\geq y + 2 * r$, тем самым выйдя за рамки поиска.

Внутри цикла сначла идет проверка на $\geq y + 2 * r$, а потом два варианта развития событий - elif в 22 второй строчке отвечает за поиск диска справа (проверяется два условия - перебираемый диск из массива лежит справа и меньше ли до него расстояние, чем текущий минимум, в результате чего, если это так, то отдельно складываются координаты предпологаемого следующего диска и его позиция в массиве, для чего это нужно - станет понятно дальше), и поиска диска слева, про второй случай поговорим чуть подробнее.

Имеется ввиду следующая ситуация:

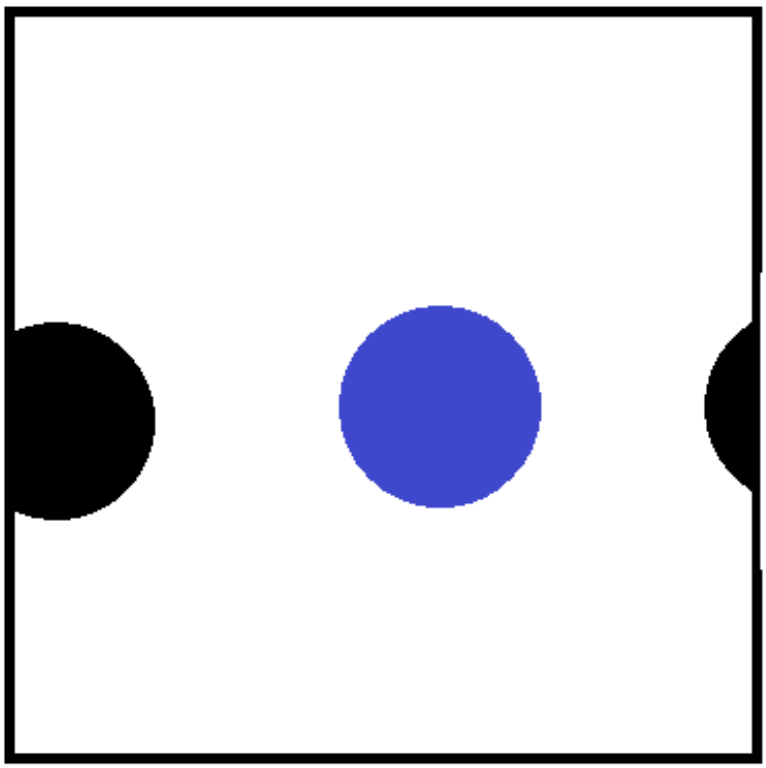

Рис. 22: Функция FNB. Поиск слева.

В такой ситуации ранее была проблема, потому что центр черного диска находился слева, а его часть, которая находилась на правой границе коробки никак не проглядывалась, теперь эта проблема была решена поиском слева. Буквально делается следующее - если диск находится в полосе поиска, при этом левее выбранного, синего диска то мы "перекидываем"такой черный диск на l_box вправо по O_x и запускаем ССВ (функция вычисления расстояния). Мы

не перещаем фактически черный диск, лишь передаем измененные координаты в ССВ.

Таким образом, в отличие от прошлых вариантов FNB просматривается вся полоса, а не идет муторный разбор вариантов и в случае нужды повторный запуск FNB.

И момент, про который я хотел сказать отдельно - **почему возвращаются отдельно координаты диска и его позицию в массиве?** Причина заключается в том, что если мы получаем диск из левого случая, то расстояние при столкновении (и соответственно перемещение) необходимо расчитывать для диска сдвинутого, а не в его исходном положении, но при этом, если он будет следующим в цепочке столкновений , то необходимо будет запускать движение из его исходных координат, то есть:

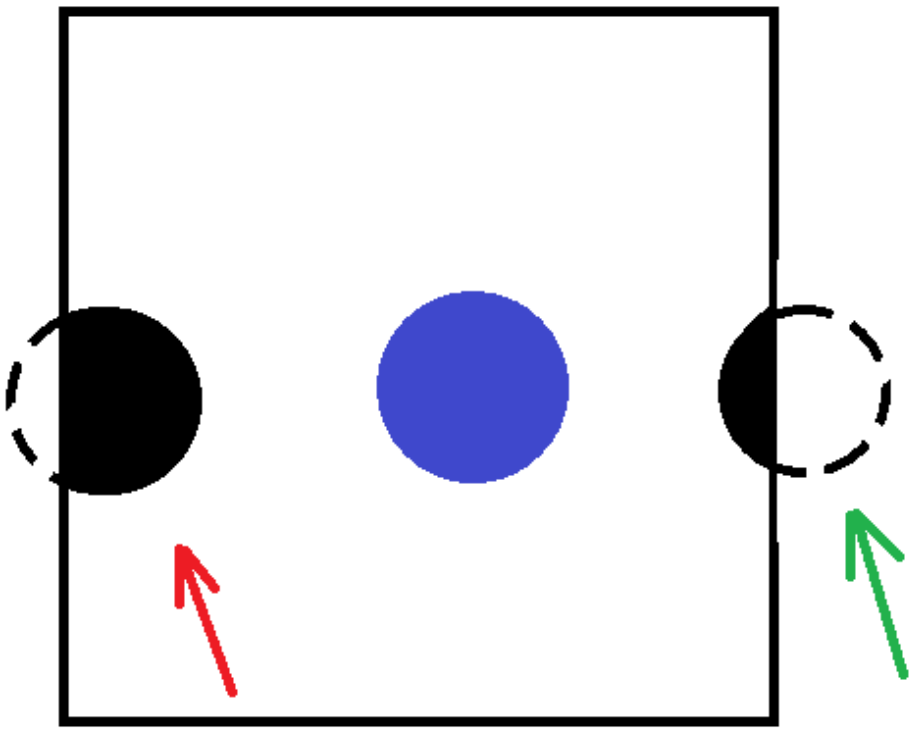

Рис. 23: Функция FNB. Пояснение к возвращаему результату.

Моделируем столкновение до диска, на который указывает зеленая стрелка (его, чисто фактически, не будет в массиве, его образ "получился"засчет функции ССВ и сдвига реального диска), а последующее движение будет запущено для диска, на который указывает красная стрелка (который, как раз таки, фактически есть в массиве).

7.3.2 Второй случай

Рассмотрим второй случай, который на рисунке 21 находится справа - выход зоны поиска снизу:

```
31      # 2 случай, зона поиска выходит снизу.
32      elif y - 2 * r + eps < 0:
33
34          # Поиск в стандартном пространстве.
35          for i in range(LowerBound(arr, y - 2 * r + eps), len(arr)):
36
37              if arr[i][1] > y + 2 * r + eps:
38                  break
39
40              elif x + eps < arr[i][0] and CCB(x, y, arr[i][0], arr[i][1], r)[1] < minn - eps:
41                  minn = CCB(x, y, arr[i][0], arr[i][1], r)[1]
42                  x_res, y_res = arr[i][0], arr[i][1]
43                  pos = i
44
45              elif arr[i][0] < x + eps and CCB(x, y, arr[i][0] + l_box, arr[i][1], r)[1] <= minn - eps:
46                  minn = CCB(x, y, arr[i][0] + l_box, arr[i][1], r)[1]
47                  x_res, y_res = arr[i][0] + l_box, arr[i][1]
48                  pos = i
49
50          # Поиск в сдвинутом пространстве.
51          x_tmp, y_tmp = x, y + l_box
52
53          for i in range(LowerBound(arr, y - 2 * r + l_box + eps), len(arr)):
54
55              # Тут мы просто пробегаем доверху, то есть до конца массива.
56              if x_tmp + eps < arr[i][0] and CCB(x_tmp, y_tmp, arr[i][0], arr[i][1], r)[1] < minn - eps:
57                  minn = CCB(x_tmp, y_tmp, arr[i][0], arr[i][1], r)[1]
58                  # Важный момент с тем, что возвращаемся к обычным координатам! Не сдвинутым!
65                  x_res, y_res = arr[i][0] + l_box, arr[i][1] - l_box
66                  pos = i
```

Рис. 24: Функция FNB. Второй случай.

elif в 32 строке определяет случай, после чего с 35 по 48 строку все по аналогии с 1 случаем (таким образом разобрали синюю область с цифрой 1, с рисунка 21).

Поиск в сдвинутом пространстве (красная область на рисунке 21, с цифрой 2) осуществляется почти аналогично, но есть два ключевых отличия:

1. Происходит замена координат диска, для которого производится поиск - увеличиваем координату по O_y на l_{box}, и будто бы перемещаем диск наверх для поиска.

2. Меняются рамки поиска - идем от сдвинутых координат по O_y, из которых вычли удвоенный радиус (все аналогично первому случаю) до самого верха - конца массива.

В поиске в сдвинутом пространстве также есть поиск слева и справа, все реализовано аналогично.

Остается аккуратно вернуть координаты - где-то вычесть l_{box}, а где-то добавить, в зависимости от случая.

7.3.3 Третий случай

Третий случай аналогичен второму, с единственным отличем - там происходит не поднятие, а спуск диска (соотвествует первому рисунку на рисунке 21)

Если не один из случаев не вернул результата, то делаем вывод, что в полосе дисков нет, и движение ничем не ограниченно - в таком случае будем возвращать $pos = -1, x_{res} = 0, y_{res} = 0$.

7.4 Объявление функции CCB

Говоря в целом, изменения функции были описаны выше, но давайте рассмотрим их еще раз. Для начала код:

```
2  def CCB(x1, y1, x2, y2, r):
3      eps = 1e-8
4      if ((2 * r) ** 2 - (y1 - y2) ** 2) < 0:
5          # Если написанная проверка на программную ошибку не сработала, то делаем собственную.
6          if abs(y1 - y2) - 2 * r < eps:
7              x_new = x2
8              l_pr = x_new - x1
9          else:
10             raise ValueError('A very specific bad thing happened for disks.', (x1, y1), " and ", (x2, y2))
11     else:
12         x_new = x2 - (((2 * r) ** 2 - (y1 - y2) ** 2) ** (1 / 2))
13         l_pr = x_new - x1
14
15     return [x_new, l_pr]
```

Рис. 25: Функция CCB.

В целом, функция делает все тоже, что и раньше - вычисляет расстояние между дисками и возвращает новую координату по O_x (по O_y она не меняется)

Но в последней версии появились проверки, в результате накопления ошибки и отрицательного значения варажения проверка - погрешность $<$ eps $\Longrightarrow$ касание, иначе выдаем исключение.

В результате возвращаем новую координату и пройденное расстояние.

7.5 Объявление функции MOB_x (MOB_y)

Финальный вариант данной функции гораздо компактнее, логичнее и проще своих предшественников. Рассмотрим его ближе:

```
def MOB_x(ind_sh, arr, l_ost, l_box, r):

    # Прежде всего, необходимо понять, находится ли диск в буферной з
    eps = 1e-8
    x = arr[ind_sh][0]
    y = arr[ind_sh][1]

    # Теперь находим следующий диск в цепочке столкновений:
    pos, x_cl, y_cl = FNB(x, y, arr, l_box, r)

    # Pos = -1 => дисков в полосе поиска вообще нет, поэтому двигаем
    if pos == -1:
        arr[ind_sh][0] = (x + l_ost) % l_box
        return arr

    # Pos != -1 => есть диск для столкновения.
    else:
        x_new, l_pr = CCB(x, y, x_cl, y_cl, r)

        # l_ost < l_pr => мы даже не дойдем до следующего диска.
        if l_ost < l_pr - eps:
            arr[ind_sh][0] = (x + l_ost) % l_box
            return arr

        # l_ost >= l_pr => мы даже не дойдем до следующего шара.
        else:
            arr[ind_sh][0] = (x + l_pr) % l_box
            return MOB_x(pos, arr, l_ost - l_pr, l_box, r)
```

Рис. 26: Функция MOB_x.

Теперь тут три простых случая:

1. Pos = -1 $\Longrightarrow$ дисков в полосе нет, поэтому тут двигаемся по-максимуму, и в конце берем координату по O_x по модулю l_{box}. Таким образом, в 1 случае мы сразу получаем результат, рекурсии нет.

2. Далее, Pos $\neq$ -1 $\Longrightarrow$ столкновение есть, причем расстояние, которое можно пройти - $l_{ost} < l_{pr}$ меньше расстояния, до следующего диска. Поступаем схоже с первым случаем - прибавляем к координате O_X выбранного диска l_{ost} и берем все по модулю l_{box}. Если диск по O_x "за пределами"коробки после движения, то он "нормализуется а если нет, то останется на месте.

3. В последнем случае Pos $\neq$ -1 и $l_{ost} \geq l_{pr}$ $\Longrightarrow$ будем двигать выбранный диск по O_x на l_{pr}, уменьшать l_{ost} на l_{pr} и рекурсивно запускать MOB_x

для следующего диска. Здесь нам и надобится возвращение из FNB позиции диска, а не просто его координат.

Если говорить про описание MOB_y - то нужно лишь заметить, что движение относительно O_y можно сделать поменяв в массиве местами координаты по O_x и O_y для каждого диска, после чего остается отсортировать массив по O_y (вне функции, при чем координаты по O_y в самом же деле являются координатами по O_x) и запустить MOB_x:

```
# Моделирование движения шаров параллельно оси y.
def MOB_y(ind_sh, arr, l_ost, l_box, r):

    # Разворачиваем массив.
    for i in range(len(arr)):
        arr[i] = np.array([arr[i][1], arr[i][0]])

    # Передаем в функцию движения.
    MOB_x(ind_sh, arr, l_ost, l_box, r)

    # Разворачиваем массив.
    for i in range(len(arr)):
        arr[i] = np.array([arr[i][1], arr[i][0]])

    # Возвращаем результат.
    return arr
```

Рис. 27: Функция MOB_y.

7.6 Объявление функции SP

Данная функция отвечает за построение графика и сохранения изображения столкновений по массиву. Рассмотрим часть кода:

```
def SP(arr, l_box, r, ind):

    a = arr
    name = "Shar" + str(ind) + ".png"
    fig, ax = plt.subplots(figsize=(7,7))
    ax.set(xlim=(0, l_box), ylim=(0, l_box))

    for i in range(len(a)):

        if a[i][0] < r and a[i][1] > l_box - r:
            circle1 = plt.Circle((a[i][0], a[i][1]), r, color='r')
            circle2 = plt.Circle((a[i][0] + l_box, a[i][1]), r, color='r')
            circle3 = plt.Circle((a[i][0], a[i][1] - l_box), r, color='r')
            circle4 = plt.Circle((a[i][0] + l_box, a[i][1] - l_box), r, color='r')
            ax.add_patch(circle1)
            ax.add_patch(circle2)
            ax.add_patch(circle3)
            ax.add_patch(circle4)

        elif r < a[i][0] < l_box - r and a[i][1] > l_box - r:
            circle1 = plt.Circle((a[i][0], a[i][1]), r, color='r')
            circle2 = plt.Circle((a[i][0], a[i][1] - l_box), r, color='r')
            ax.add_patch(circle1)
            ax.add_patch(circle2)
```

Рис. 28: Функция SP (ее часть).

На вход она принимает: массив дисков - arr, размер коробки - l_{box}, радиус дисков - r, и переменная ind, которая отвечает за название изображения.

Разберем подробнее, как она устроена:

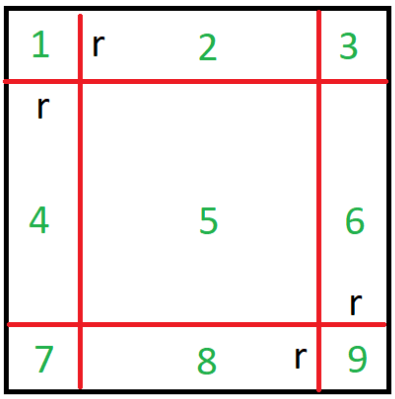

Рис. 29: Функция SP. Логика функции.

В зависимости от координат диска рассматривается один из 9 вариантов, за отвечают if-ы. На рисунке представлен код для случаев 1 и 2, разберем,

например, случай два - если диск находится там, то его верхняя часть будет снизу, в зоне 8, а также основная часть (где находится центр) в зоне 2.

Аналогичным образом рассмотрены все 9 зон. Каждый диск или его часть будет отображена на результирующем графике, который в последующем будет сохранен.

В начале функции отдельно копируем массив, дабы не испортить исходный, задаем настройки графика и начинаем в цикле перебор случаев для каждого из 9 случаев.

7.7 Объявление переменных

В данной ячейке выписаны ключевые значения, при генерации дисков:

```
# Переменные для генерации размещения дисков:
# l_box = 38
# mas = np.array([]).reshape(0,2)

# Параметры дисков и коробки:
r = 1
l_box = 11

a = np.array([[1,1], [4,1], [10, 2.5], [3,3], [5,3], [1,4], [3,5]])
a = a[np.argsort(a[:, 1])]
tmp = a
intrmd_res = []

# Переменные для графического отображения:
frames = []
```

Рис. 30: Объявление ключевых переменных.

В процессе написания кода отлаживал код на массиве a, в последующем - генерировал упаковку и складывал ее в mas.

7.8 Генерация упаковки дисков

Тут и происходит генерация расположения дисков по формуле (для r = 1 (!)), описанной в ячейке, в результате чего получается плотность размещения около 0.65:

```
for i in range(1, l_box, 3):
    for j in range(1, l_box, 3):
        mas = np.append(mas, [[j, i]], axis = 0)

for i in range(2, l_box, 3):
    for j in range(2, l_box, 3):
        mas = np.append(mas, [[j + 0.5, i + 0.5]], axis = 0)

# tmp = mas
```

Рис. 31: Генерация координат для размещения дисков.

7.9 Основная ячейка

Настало время для финальной ячейки:

```
1  %%time
2  # Переходим в папку с результатами.
3  os.chdir("Results")
4
5  # Сохраняем начальное положение дисков.
6  intrmd_res.append(tmp)
7  SP(tmp, l_box, r, 0)
```

Рис. 32: Основная ячейка. Подготовительная часть.

В начале ячейке прописана преписка "%%time которая позволяет узнать итоговое время работы ячейки после выполнения алгоритма. После чего происходит переход в другую директорию, где будут сохранены результаты.

"intrmd_res лист, который будет содержать в себе все промежуточные массивы с результатами. В 7 строке происходит сохранение первого кадра.

```
9  # После чего начинаем моделировать столкнов
10 for i in range(1, 20):
11     seed = np.random.randint(2)
12     l_ost = np.random.randint(1, l_box)
13
14     if seed == 0:
15         # Перед началом нужно не забыть отс
16         tmp = tmp[np.argsort(tmp[:, 1])]
17         shx = np.random.randint(len(tmp))
18
19         MOB_x(shx, tmp, l_ost, l_box, r)
20         intrmd_res.append(tmp)
21
22     if seed == 1:
23         tmp = tmp[np.argsort(tmp[:, 0])]
24         shy = np.random.randint(len(tmp))
25
26         MOB_y(shy, tmp, l_ost, l_box, r)
27         intrmd_res.append(tmp)
28
29     SP(tmp, l_box, r, i)
```

Рис. 33: Основная ячейка. Основная вычислительная часть.

После чего в цикле генерируется случайное число - seed, которое отвечает за

выбор движения - по O_x или O_y, а также случайное расстояние, которое нужно пройти.

Два случая отличаются лишь тем, что из-за устройства функций MOB_x и MOB_y в первом случае нужно сделать сортировку до запуска функции по O_y, а во втором по O_x. Нужно это для того, чтобы был одназначно выбран диск, потому что если выбрать сначала диск, потом поменять координаты и отсортировать, то произойдет то, что было описано на рисунке 17.

После каждого шага в intrmd_res складывается промежуточный массив, а через SP сохраняем результат.

```
31  # Список для хранения кадров.
32  frames = []
33  LoF = natsorted(os.listdir())
34
35  for frame_number in range(len(LoF)):
36      frame = Image.open(LoF[frame_number])
37      frames.append(frame)
38
39  frames[0].save(
40      'colission.gif',
41      save_all=True,
42      append_images=frames[1:],  # Срез котор
43      optimize=True,
44      duration=1500,
45      loop=0
46  )
47
48  # Не забываем вернуться в исходную папку.
49  os.chdir("..")
50
51  res = tmp
```

Рис. 34: Основная ячейка. Часть создания анимации.

В данной части ячейки происходит следующее: создается лист frames, в котором будут лежать кадры для последующего объеденения их в gif-картинку.

Для этого, благодаря библиотеке os, создается список всех файлов в директории "Results далее в цикле каждая картинка добавляется в frames.

После чего через библиотеку Image в строках 39 - 46 идет создание покадровой анимации. В конце ячейки возвращаемся в исходную директорию и отдельно складываем в list "res"финальное положение дисков.

7.10 Дополнительная ячейка. Перевод Python-кода в псевдокод.

Тут, думаю, будет достаточно просто самого кода, поскольку процесс, который происходит - довольно понятен: берется строчка и прогоняется через библиотечную функцию, после чего она складывается в текстовый файл:

```
txt = open('bebra.txt', 'r', encoding="utf-8")
lines = txt.readlines()
lines_r = []
for i in range(len(lines)):
    lines_r.append(p2p.python_to_pseudocode(lines[i].replace("\n","")))

txt.close()
```

```
with open ("Results_pseudo.txt", "w", encoding="utf-8") as output:
    for line in lines_r:
        output.write(line + '\n')
```

Рис. 35: Дополнительная ячейка. Перевод Python-кода в псевдокод.

На этом 3 отчет заканчивается. На всякий случай, прикладываю еще раз ссылку на git: https://github.com/HpPpL/Collision-of-balls.