**МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ**

**БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ**

**МЕХАНИКО-МАТЕМАТИЧЕСКИЙ ФАКУЛЬТЕТ**

**Кафедра дифференциальных уравнений и системного анализа**

# ОБЪЕКТНО-ОРИЕНТИРОВАННАЯ РЕАЛИЗАЦИЯ АГЕНТНЫХ МОДЕЛЕЙ

Курсовая работа

Старовойтовой Виктории Александровны

студентки 3 курса,
специальность 1-31 03 09
Компьютерная
математика
и системный анализ

Научный руководитель:
кандидат физ.-мат. наук,
доцент О.А.Лаврова

Минск, 2018

# ОГЛАВЛЕНИЕ

# ВВЕДЕНИЕ

Агентное моделирование и симуляция - это новый подход к системам моделирования, состоящим из автономных взаимодействующих агентов. Вычислительные достижения нашего времени позволяют увеличить количество моделей на основе агентов в самых разный областях применения. Эти области варьируются от поведения модельного агента на фондовом рынке, цепочек поставок и потребительских рынков до прогнозирования распространения эпидемий, смягчения угрозы биовойны и понимания факторов, которые могут быть причиной падения древних цивилизаций. Такой прогресс свидетельствует о том, что потенциал агентного моделирования оказывает далеко идущее влияние на то, как предприятия используют компьютеры для поддержки принятия решений, а исследователи используют модели на основе агентов в качестве электронных лабораторий. Существует мнение, что агентное моделирование «является третьим способом осуществления научных исследований» и может дополнять традиционные дедуктивные и индуктивные методы рассуждений.[2]

Данная курсовая работа носит программно-исследовательский характер. При этом целью работы является исследование принципов агентного моделирования на примере реализации полета стаи птиц. Ведь само агентное моделирование является относительно новым методом, получившим широкое практическое распространение только после 2000 года. Поэтому сейчас стоит задача не только разобраться с данным видом моделирования, а также реализовать собственную программу на основе данного вида.

# ГЛАВА 1
# ОБЩИЕ ПОЛОЖЕНИЯ

## 1.1 Введение в агентное моделирование

Моделирование на основе агентов  или агентное моделирование - это модельная и вычислительная основа для моделирования динамических процессов, которая включает автономных агентов. Автономный агент действует самостоятельно, без внешнего управления, в ответ на ситуации, с которыми агент сталкивается во время моделирования.[2]

Моделирование популяции автономных агентов, с присущими ей характеристиками и поведением, которые постоянно взаимодействуют, является особенностью агентного моделирования. Моделирование на основе агентов наиболее часто используется для моделирования индивидуального принятия решений и социального и организационного поведения.[7]

Агент является общим понятием, имеющим широкую применимость. Агенты часто представляют собой людей или группы людей, а агентные отношения - процессы социального взаимодействия. Например, модель распространения инфекционных заболеваний, иллюстрирующая как происходит передача инфекции в результате контакта с другими людьми. В агентной модели цепочки поставок, агенты - это фирмы с принятием решений о материальных источниках и заказах, запасах, доставке, расширении емкости и т. д. Разработка инструментов моделирования на основе агентов, наличие микроданных в транзакциях и взаимодействиях агентов и успехи в вычислении позволили увеличить число применений агентного моделирования в различных областях и дисциплинах.

## 1.2 Свойства агентов

Для практических целей моделирования мы считаем, что агенты обладают определенными свойствами и атрибутами:

• Агент является автономным и самонаправленным. Агент независимо действует в среде и независимо взаимодействует с другими агентами в определенном наборе случаев, которые представляют интерес в конкретной ситуации. Поведение агента - это процесс, который связывает восприятие агентом своей среды и его решения и действия.

• Агенты являются модульными или автономными. Агент - это идентифицируемый, дискретный индивидуум с набором характеристик или свойств, поведением и возможностью принятия решений. Требование дискретности подразумевает, что агент имеет в некотором смысле границу и можно легко определить, является ли что-то частью агента или нет.

• Агент является социальным, взаимодействует с другими агентами. У агентов есть протоколы или механизмы, которые описывают, как они взаимодействуют с другими агентами, также у агента есть свое поведение. Общие протоколы взаимодействия с агентом включают конкуренцию за пространство и предотвращение столкновений; распознавание агента; контактирование и обмен информацией; влияние; и другие механизмы, специфичные для домена или приложения.

Агенты часто имеют дополнительные свойства, которые могут считаться определяющими или необходимым для агентства.

• Агент может жить в среде. Агенты взаимодействуют со своей средой, а также с другими агентами. Агент локализован в том смысле, что его поведение является ситуационно зависимым, что означает, что его поведение основано на текущем состоянии его взаимодействия с другими агентами и с окружающей средой.

• У агента могут быть явные цели, которые управляют его поведением. Цели не обязательно должны быть достигнуты, это позволяет агенту постоянно сравнивать результаты своего поведения с его целями и дает ему ориентир для возможного изменения его поведения.

• У агента может быть возможность учиться и адаптировать свое поведение на основе его опыта. Индивидуальное обучение и адаптация требуют наличия у агента памяти, обычно в виде атрибута динамического агента. (Мы сравниваем индивидуальную адаптацию с адаптацией населения. При адаптации населения доля лиц в населении с определенными атрибутами, которые лучше подходят для их среды, со временем увеличивается. Люди не обязательно меняют свое поведение или адаптируются).

• Агенты часто имеют атрибуты ресурсов, которые указывают текущий запас одного или нескольких ресурсов, например, энергию, богатство, информацию и т. д.

Поведенческие правила агента могут варьироваться в зависимости от их сложности, от того, сколько информации рассматривается в решении агента (это называется когнитивной нагрузкой), от внутренней модели внешнего мира агента, включая возможные реакции или поведение других агентов и от памяти агента о прошлых событиях, которые тот сохраняет и использует в своих решениях. Часто в моделях отсутствует адаптация, потому что это не является важным для достижения намеченной цели модели. Например, в модели цепочки поставок может не потребоваться моделировать адаптацию агента, если целью модели является оценка набора конкретных правил управления запасами.

## 1.3 Взаимодействие агентов

Важной частью агентного моделирования является не только моделирование агентов и их поведения, но и моделирование их взаимоотношений и взаимодействий. Основные задачи моделирования взаимодействия агентов - это определение того, кто с кем связан или мог бы быть связанным и регулированием динамики механизмов взаимодействия. Например, основанная на агентах модель роста Интернета будет включать механизмы, которые определяют, кто с кем связывается, зачем и когда. [7]

Общие топологии для представления взаимодействия с социальными агентами включают:

- суп: не пространственная модель в которой агенты не имеют атрибута местоположения;
- сетки или решетки: сотовые автоматы представлящие собой шаблоны взаимодействия агента и доступную локальную информацию сеткой или решеткой; клетки, непосредственно окружающие агента, являются его окрестностями. Расположение агента - это индекс ячейки сетки;
- евклидово пространство: агенты перемещаются в 2D или 3D пространствах. Расположение агента - его относительные или геопространственные координаты;
- географическая информационная система (ГИС): Агенты перемещаются и взаимодействуют с реалистичными пятнами геопространственных ландшафтов. Расположение агента - это географическая единица (например, почтовый индекс) или геопространственные координаты;
- сети: сети могут быть статическими (ссылки предварительно заданы) или динамическими (связи, определяемые эндогенно механизмами

привязки отношений). Расположение агента - это относительное местоположение узла в сети.

Независимо от того, какая топология агент-взаимодействие используется в агентной модели для контактирования агентов, основная идея заключается в том, что агенты в любой момент времени взаимодействуют с ограниченным числом других агентов. Это реализуется путем определения локальной окрестности (возможно, сети) и ограничения взаимодействия с небольшим количеством агентов, которые находятся в этой окрестности. Это не означает, что агенты должны находиться в непосредственной близости друг к другу пространственно, чтобы иметь возможность взаимодействовать. Сетевая топология позволяет агентам связываться на основе отношений в дополнение к близости. Например, агент может быть членом многих сетей, например, близости, социальных, семейных отношений, идеологических и т. д.[7]

# ГЛАВА 2
# РЕАЛИЗАЦИЯ АГЕНТНОЙ МОДЕЛИ

## 2.1 Модель поведения стаи птиц

Данная модель имитирует поведение стаи птиц в нескольких измерениях. Эта модель широко использует операцию поиска ближайших соседей.

Поверхность, на которой происходит движение “агентов-птиц”, представляет из себя “закольцованный” квадрат (т.е. поверхность тора). Все “птицы” перемещаются с одной и той же постоянной по модулю скоростью. На каждом шаге модели происходит изменение направления вектора скорости каждой из “птиц” в соответствии со следующими тремя правилами:

1. **“Разделение”**: при наличии ближайшего соседа на расстоянии меньшем определенного “радиуса опасности” каждая “птица” отворачивает от направления движения этого соседа с ограничением по углу поворота;
2. **“Выстраивание”**: каждая “птица” поворачивает с ограничением по углу поворота к среднему направлению движения ее ближайших соседей, находящихся в определенном “радиусе зрения” от нее;
3. **“Сближение”**: каждая “птица” поворачивает с ограничением по углу поворота к направлению на геометрический центр всех ее соседей в пределах “радиуса зрения”.

**Поведение птиц в стае**

- Непрерывное пространство и движение
- Птицы адаптируют свой полет в соответствии с движением других птиц в стае
- Результат - сложные, на вид скоординированные образцы полета - стаи

## 2.2 Проблемы агентного моделирования

Открытые вопросы для агентного моделирования заключаются в проблемах масштабирования, возникающих при имитации большого количества агентов.

Как много агентов можно включить в среду моделирования?

Основной принцип моделирования на основе агентов заключается в том, что агенты взаимодействуют и обмениваются информацией, доступной локально, с другими агентами, расположенными только в их непосредственной близости.

Окружение агента быстро меняется с течением времени в симуляции, по мере того как агенты перемещаются в пространстве. Близость зависит от топологии, определенной для взаимодействия агентов. В нашем случае это двумерное непрерывное пространство $R^2$.

Общая стратегия для определения соседей агента:

1. Выбрать подмножество агентов в качестве кандидатов в соседи для вычисления таким образом, чтобы подмножество включало нынешних соседей.
2. Оценить кандидатов по критерию близости.

Особые характеристики проблемы нахождения соседей в агентном моделировании:

- Агенты мобильны и динамичны в своих положениях
  - для каждого агента необходимо определить его соседей в каждый момент времени симуляции
  - полная структура данных должна создаваться каждый раз, когда изменяются положения агентов по мере симуляции
- Все агенты хранятся в памяти

## 2.3 Алгоритмы поиска ближайших соседей

### Алгоритм 1: "All Neighbors"

- Наивная реализация алгоритма определения соседей состоит в перечислении всех агентов в качестве соседних кандидатов для каждого агента
    - Результат - N2сравнений, O(N2) алгоритм
    - O(N2) вычислительно сложен для больших N

- Из-за симметрии задачи определения соседей количество сравнений агентов может быть уменьшено:
    - (Симметрия) агент a является соседним агентом b тогда и только тогда, когда агент b является соседом агента a
    - Все еще O(N2)

### Алгоритм 2: Алгоритм решетчатой ячейки

- Все представленное пространство состоит только из пространства, охваченного агентами
- Пространство ячеек, которое не содержит хотя бы одного агента, не показывается

**Шаги алгоритма**

Вход: список агентов.

Создание области соседей:

1.1 Увеличение списка агентов с их положениями в ячейках

1.2. Собрать все положения ячеек с агентами (только клетки с агентами)

Нахождение агентов в областях соседства:

2.1. Сформулировать правила перевода ячеек в агентов, находящихся

там

2.2 Увеличение списка агентов с кандидатами на соседство

2.3 Увеличение списка агентов с текущими соседями.

Обновить агентов:

3.1 Обновить список агентов основанный на соседских атрибутах

Выход: обновленный список агентов

**Алгоритм 3: “Алгоритм дерева квадрантов” (KDTree algorithm)**

- Квадрантные деревья известны как способ обработки запросов в поиске баз данных
- Каждый узел квадрантного дерева представляет прямоугольную область в двумерном пространстве и содержит один из объектов, расположенных в этой области
- Прямоугольники перекрываются в отличие от сетки ячеек
- Обычно такое дерево предназначено для минимизации времени поиска объекта в данном дереве
  - Не нужно думать о времени для создания дерева, поскольку объекты в пространственных или графических базах данных являются статическими
  - При моделировании дерево должно создаваться на каждом временном шаге
  - Таким образом, существует рекуррентное взыскание за создание структуры данных при нахождении соседа агента

Квадратное дерево определяется пятью полями: агент в узле и четыре поля, обозначающие поддеревья агентов в четырех квадрантах, NE, NW, SW, SE или пустое поддерево

**Определение соседей с помощью дерева**

- сравнить координаты точки с расположение агента в вершине каждого дерева, начиная с корня

- применить рекурсивно для всех соответствующих поддеревьев
- продолжить на узлах всех соответствующих поддеревьев
- все упомянутые поддеревья дают набор кандидатов на соседство для данной точки

Общая идея состоит в том, что kd-дерево является двоичным деревом, каждый из узлов которого представляет собой гиперпрямоугольник, ориентированный по оси. Каждый узел определяет ось и разбивает множество точек на основании того, является ли координата вдоль этой оси больше или меньше определенного значения. Во время строительства ось и точка разделения выбираются по правилу «скользящей середины», что гарантирует, что не все клетки станут длинными и тонкими.

Дерево можно построить для поиска ближайших соседей любой точки (необязательно возвращая только те, которые находятся на некотором расстоянии от точки). Его также может построить с существенным повышением эффективности для поиска заданного количества ближайших соседей.

Для больших размеров (20 уже достаточно большое значение), алгоритм не будет работать намного быстрее, чем полный перебор. Высокоразмерные запросы ближайших соседей являются открытой проблемой.[7]

**Метод скользящей середины:** сначала выполняется деление ячейки пополам, с помощью гиперплоскости, проходящей через центр ячейки и делящую пополам самую длинную ее сторону. Если точки лежат по обе стороны от плоскости разделения, то плоскость разделения остается здесь. Однако, если все точки данных лежат по одну сторону плоскости разделения, то плоскость разделения «скользит» к данным точкам, пока не встретит первую такую точку. Один ребенок - это листовая ячейка, содержащая эту единственную точку, и далее алгоритм повторяется в остальных точках.

## 2.4 Описание классов и методов

Для агентного моделирования полета стаи птиц средствами Python был создан класс Boid, в котором каждая птица характеризуется следующим набором полей: положение в пространстве, скорость полета и направление. Все поля, кроме скорости полета, задаются случайным образом, скорость полета задается пользователем. В процессе работы создавалась популяция с 20 агентами и размерностью пространства 800×600 (рисунок 2.1). Шаг между эпохами принимался равным 1 секунде.

На первом шаге случайным образом происходит генерация агентов и отображение среды. За визуализацию среды отвечает библиотека Tkinter.

Моделирование поведения агента происходит в два этапа. На первом этапе агент корректирует свое направление согласно правилам, описанным в пункте 2.1, посредством функции adjust_heading(), а затем осуществляет непосредственное перемещение (функция move_agent()), изменяя значение своего положения в соответсвии с заданной пользователем скоростью – переменная speed.

На каждом последующем временном промежутке каждый агент определяет агентов в непосредственной близости – соседей (getConnections()). Для определения соседей используется алгоритм «Дерево квадрантов», который в Python реализован в библиотеке KDTree.

Подробнее об этапах моделирования поведения агента. Первый этап или определение направления агента состоит в следующем: если соседи агента слишком близко к нему расположены (расстояние близости пользователь задает через параметр crowded), агент корректирует свое направление, отварачиваясь дальше от направления соседей (реализовано функцией separate()). Угол отворота задает пользователь, определяя значение переменной max_turn_away. Далее агент корректирует направление своего полета в соответствии со средним направлением своих соседей посредством метода align() (максимальный угол разворота за одну секунду определяет

пользователь с помощью переменной max_turn), затем с помощью метода cohere() агент определяет среднее геометрическое положение своих соседей и корректирует направление так же, как и в предыдущем случае, направляя его в центр масс.

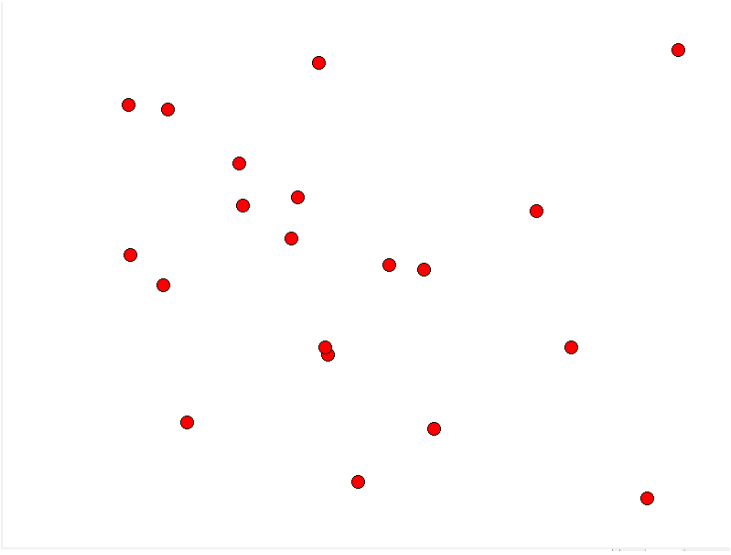

Рисунок 0.1 Задание агентов и среды

После того, как агент скорректировал свое направление согласно правилам модели, наступает второй этап – агент обновляет свое текущее положение в соответствии со своим направлением и заданной пользователем скоростью.

Далее более подробно о програмной реализации класса Boid.

Поля класса:

- **heading** – текущее направление в радианах (0…2π)

heading = random.uniform(0, 2 * math.pi)

- **x & y** – текущее положение (для задания положения используется вспомогательный класс TwoD, обладающий всеми свойствами точек в двухмерном пространстве)

x = random.uniform() * WIDTH

y = random.uniform() * HEIGHT

- **velocity** - скорость

**Параметры**

- Минимальная дистанция между птицами:

crowded = 5

- Максимальный возможный поворот:

max_turn = 0.05

max_turn_away = 0.02

**Методы:**

| **Название** | **Описание** | **Действия** |
|---|---|---|
| **void move_agent()** | Перемещение птицы в новое положение | • Вычисляет новое положение<br>• Превращает среду в тор<br>• Обновляет положение и направление агента |
| **void adjust_heading()** | Обновляет текущее направление в соответствии с правилами 1-3 | • Определяет ближайших агентов, если один из них слишком близок - разворачивает его<br>• Иначе - выстраивает и сближает с окружающими птицами из стаи |
| **void separate( double oth-ers_heading)** | Отворачивает от скопления птиц | • Определяет разницу в направлениях (direction & size) |

| | | |
|---|---|---|
| | | • Отворачивает дальше от направления “товарищей” |
| **double sub-tract_heading( double heading1, double heading2)** | Определяет разницу между двумя направлениями | • Определяет разницу<br>• Нормализует разницу между $-\pi$ и $\pi$ |
| **void align()** | Поворачивает к направлению окружающих птиц | • Определяет среднее направление окружающих птиц<br>• Выравнивает текущее положение в соответствии с полученным |
| **void cohere()** | Поворачивает к положению окружающих птиц | • Определяет среднее положение окружающих птиц<br>• Выравнивает направление для полета туда |
| **void turn_towards( double oth-ers_heading)** | Выравнивает направления с другими направлениями | • Определяет разницу в направлениях (direction & size)<br>• Поворачивает направление стаи |

Вложенность методов:

**adjust_heading()**

**separate(double others_heading)** *//dist < crowded*

**subtract_headings(double heading1,double heading2)**

**align()** //else

**turn_towards(avg_head)**

**subtract_heading(double heading1,double heading2)**

**cohere()**

**turn_towards(double others_heading)**

**subtract_heading(double heading1,double heading2)**

**void move()**

Перемещение птицы в новое положение: **move_agent(**self**)**

- Вычисляет новое положение
- Превращает среду в тор
- Обновляет положение и направление агента

```
def move_agent(self):
    self.position.x += self.velocity * math.cos(self.heading)
    self.position.y += self.velocity * math.sin(self.heading)
    if self.position.x > WIDTH:
       self.position.x -= WIDTH
    else:
       if self.position.x < 0:
          self.position.x += WIDTH
    if self.position.y > HEIGHT:
       self.position.y -= HEIGHT
    else:
       if self.position.y < 0:
          self.position.y += HEIGHT
```

---

Обновляет текущее направление в соответствии с правилами 1-3:

**adjust_heading**(self, boids):

- Определяет ближайших агентов, если один из них слишком близок - разворачивает его
- Иначе - выстраивает и сближает с окружающими птицами из стаи

```
def adjust_heading(self, boids):
    if self.getConnectionsNumber(boids) != 0:
        closest_head = self.heading
        dist = 1000
        for boid in self.getConnections(boids):
            new_dist = (((self.position.x - boid.position.x) ** 2) + (
                    (self.position.y - boid.position.y) ** 2)) ** 0.5
            if new_dist < dist:
                dist = new_dist
                closest_head = boid.heading
        if dist < crowded:
            head = separate(self.heading, closest_head)
        else:
            self.heading = self.align(boids)
            head = self.cohere(boids)
    else:
        head = self.heading
    return head
```

---

Отворачивает от скопления птиц: **separate(**heading, others_heading**)**

- Определяет разницу в направлениях (direction & size)
- Отворачивает дальше от направления “товарищей”

```
def separate(heading, others_heading):
    diff = subtract_headings(heading, others_heading)
    if diff > 0:
        heading -= max_turn_away
    else:
        heading += max_turn_away
    if heading < 0:
        heading += 2 * math.pi
    if heading > 2 * math.pi:
        heading -= 2 * math.pi
    return heading
```

---

Определяет разницу между двумя направлениями:

**subtract_heading(**self_heading, boid_heading**)**

- Определяет разницу
- Нормализует разницу между $-\pi$ и $\pi$

```
def subtract_headings(self_heading, boid_heading):
    diff = boid_heading - self_heading
    if diff <= -math.pi:
        diff += 2 * math.pi
    if diff > math.pi:
        diff -= 2 * math.pi
    return diff
```

---

Поворачивает к направлению окружающих птиц: **align(**self, boids**)**

- Определяет среднее направление окружающих птиц
- Выравнивает текущее положение в соответствии с полученным

```
def align(self, boids):
        avg_head = self.heading
        for boid in self.getConnections(boids):
            if boid is not self:
                avg_head += boid.heading
        avg_head /= self.getConnectionsNumber(boids)

        head = turn_towards(self.heading, avg_head)
        return head
```

---

Поворачивает к положению окружающих птиц: **cohere**(self, boids)

- Определяет среднее положение окружающих птиц
- Выравнивает направление для полета туда

```
def cohere(self, boids):
    # clumping
    avg_pos = TwoD(0, 0)
    for boid in self.getConnections(boids):
        if boid is not self:
            avg_pos += boid.position
    avg_pos /= self.getConnectionsNumber(boids)
    pos_head = math.atan2(self.position.y - avg_pos.y, self.position.x -
avg_pos.x)
    head = turn_towards(self.heading, pos_head)
    return head
```

---

Выравнивает направления с другими направлениями:

**turn_towards(others_heading)**

- Определяет разницу в направлениях (direction & size)

- Поворачивает направление стаи

```
def turn_towards(heading, boid_heading):
    diff = subtract_headings(heading, boid_heading)
    if diff > 0:
        heading += min(diff, max_turn)
    else:
        heading += max(diff, -max_turn)
    if heading < 0:
        heading += 2 * math.pi
    if heading >= 2 * math.pi:
        heading -= 2 * math.pi
    return heading
```

---

Определяет соседей агента:

**getConnections**(self, boids)

```
def getConnections(self, boids):
    arr = np.zeros(shape=(len(boids) - 1, 2))
    i = 0
    for boid in boids:
        if boid is not self:
            arr[i] = [boid.position.x, boid.position.y]
            i = i + 1
    T = KDTree(arr)
    idx = T.query_ball_point([self.position.x, self.position.y], r=closest)
    return np.array(boids)[idx]
```

# ЗАКЛЮЧЕНИЕ

На основании освоения полученной темы можно сделать вывод, что агентное моделирование на сегодняшний день является интересным, привлекательным и наиболее перспективным методом построения имитационных моделей сложных процессов. Были изучены возможные прикладные применения данного моделирования, варианты реализации, свойства и проблемы, а также была реализована агентная модель полета стаи птиц средствами Python.

# СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ


1. “Who's Your Neighbor? Neighbor Identification for Agent-Based Modeling using Mathematica”, Charles M. Macal.
2. C. Macal, M. North, Tutorial on Agend-based Modeling and Simulation, Journal of Simulation 4(3): 151 - 162, (2010)
3. Агентные модели: анализ подходов и возможности приложения к эпидемиологии., Авилов К.К., Соловей О.Ю., Институт вычислительной математики РАН, Россия, Москва, 2012. - 425-443 с.
4. Agent-Based Modelling and Simulation in AnyLogic [Электронный ресурс]. – Режим доступа: https://is.gd/CeKz1p. Дата доступа 25.03.2017.
5. It’s okay to be skinny if your friends are fat., S. Maneewongvatana and David M. Mount, in 4th Annual CGC Workshop on Computational Geometry, 1999.
6. R. Axtell, WHY AGENTS? ON THE VARIED MOTIVATIONS FOR AGENT COMPUTING IN THE SOCIAL SCIENCES, Center on Social and Economic Dynamics Working Paper No. 17 (2000)
7. S. Maneewongvatana, D. M. Mount, On the Efficiency of Nearest Neighbor Searching with Data Clustered in Lower Dimensions, Department of Computer Science, Univ. Maryland, 2001.

# Код программы

Ниже приведен код алгоритма на Python:

```
import random  # FOR RANDOM BEGINNINGS
from Tkinter import *  # ALL VISUAL EQUIPMENT
import math
import numpy as np
from scipy.spatial import KDTree

WIDTH = 800  # OF SCREEN IN PIXELS
HEIGHT = 600  # OF SCREEN IN PIXELS
BOIDS = 3  # IN SIMULATION
WALL = 100.0  # FROM SIDE IN PIXELS
BOID_RADIUS = 7  # FOR BOIDS IN PIXELS
wall = -200
speed = 2
crowded = 30
max_turn = 0.05
max_turn_away = 0.02
closest = 150


################################################################################
#

def main():
    # Start the program.
    initialise()
    mainloop()


def initialise():
    # Setup simulation variables.
    build_boids()
```

```
    build_graph()


def build_graph():
    # Build GUI environment.
    global graph
    root = Tk()
    root.overrideredirect(True)
    root.geometry('%dx%d+%d+%d' % (
        WIDTH, HEIGHT, (root.winfo_screenwidth() - WIDTH) / 2,
(root.winfo_screenheight() - HEIGHT) / 2))
    root.bind_all('<Escape>', lambda event: event.widget.quit())
    graph = Canvas(root, width=WIDTH, height=HEIGHT, background='white')
    graph.after(1, update)
    graph.pack()


def update():
    # Main simulation loop.
    draw()
    move()
    graph.after(1, update)


def simulate_wall(boid):
    # Create viewing boundaries.
    max_turn = 10
    xx = boid.position.x
    yy = boid.position.y
    if xx < WALL and yy <= HEIGHT / 2 or xx <= WIDTH / 2 and yy < WALL:
        boid.heading = math.atan2(HEIGHT / 2 - yy, WIDTH / 2 - xx)

    elif xx >= WIDTH / 2 and yy < WALL or xx > WIDTH - WALL and yy <= HEIGHT
/ 2:
        boid.heading = (math.pi - math.atan2(HEIGHT / 2 - yy, xx - WIDTH /
2))

    elif yy > HEIGHT - WALL and xx < WIDTH / 2 or xx < WALL and yy > HEIGHT
/ 2:
        boid.heading = -(math.atan2(yy - HEIGHT / 2, -xx + WIDTH / 2))
    elif yy > HEIGHT - WALL and xx >= WIDTH / 2 or xx > WIDTH - WALL and yy
> HEIGHT / 2:
```

```
        boid.heading = (- math.pi + math.atan2(yy - HEIGHT / 2, xx - WIDTH /
2))
    max_turn = 0.05


def draw():
    # Draw all boids.
    graph.delete(ALL)
    for boid in boids:
        x1 = boid.position.x - BOID_RADIUS
        y1 = boid.position.y - BOID_RADIUS
        x2 = boid.position.x + BOID_RADIUS
        y2 = boid.position.y + BOID_RADIUS

        graph.create_oval((x1, y1, x2, y2), fill='red')
        graph.create_oval(boid.position.x - closest,  boid.position.y -
closest, boid.position.x + closest, boid.position.y + closest)
        graph.create_line(boid.position.x,boid.position.y,
boid.position.x+(boid.velocity+20)*math.cos(boid.heading),boid.position.y+(b
oid.velocity+20)*math.sin(boid.heading))

    graph.create_rectangle(WALL,WALL, WIDTH-WALL,HEIGHT-WALL)
    graph.update()


def move():
    #for boid in boids:
    #    simulate_wall(boid)
    # Move all boids.
    for boid in boids:
        boid.heading = boid.adjust_heading(boids)

    #for boid in boids:
     #   simulate_wall(boid)
    for boid in boids:
        boid.move_agent()



def build_boids():
    # Create boids variable.
    global boids
```

```
    boids = tuple(Boid(WIDTH, HEIGHT, OFFSET_START) for boid in
xrange(BOIDS))


def subtract_headings(self_heading, boid_heading):
    diff = boid_heading - self_heading
    if diff <= -math.pi:
        diff += 2 * math.pi
    if diff > math.pi:
        diff -= 2 * math.pi
    return diff


def turn_towards(heading, boid_heading):
    diff = subtract_headings(heading, boid_heading)

    if diff > 0:
        heading += min(diff, max_turn)
    else:
        heading += max(diff, -max_turn)
    if heading < 0:
        heading += 2 * math.pi
    if heading >= 2 * math.pi:
        heading -= 2 * math.pi
    return heading


def separate(heading, others_heading):
    diff = subtract_headings(heading, others_heading)

    if diff > 0:
        heading -= max_turn_away
    else:
        heading += max_turn_away
    if heading < 0:
        heading += 2 * math.pi
    if heading > 2 * math.pi:
        heading -= 2 * math.pi
    return heading
```

```
# TWO DIMENTIONAL VECTOR CLASS

class TwoD:

    def __init__(self, x, y):
        self.x = float(x)
        self.y = float(y)

    def __repr__(self):
        return 'TwoD(%s, %s)' % (self.x, self.y)

    def __add__(self, other):
        return TwoD(self.x + other.x, self.y + other.y)

    def __sub__(self, other):
        return TwoD(self.x - other.x, self.y - other.y)

    def __mul__(self, other):
        return TwoD(self.x * other, self.y * other)

    def __div__(self, other):
        return TwoD(self.x / other, self.y / other)

    def __iadd__(self, other):
        self.x += other.x
        self.y += other.y
        return self

    def __isub__(self, other):
        self.x -= other.x
        self.y -= other.y
        return self

    def __idiv__(self, other):
        if isinstance(other, TwoD):
            self.x /= other.x if other.x else 1
            self.y /= other.y if other.y else 1
        else:
            self.x /= other
            self.y /= other
        return self
```

```
    def mag(self):
        return ((self.x ** 2) + (self.y ** 2)) ** 0.5


class Boid:
# TWO DIMENTIONAL VECTOR CLASS
# BOID RULE IMPLEMENTATION CLASS
    def __init__(self, width, height):
        self.velocity = speed
        self.heading = random.uniform(0, 2 * math.pi)
        self.position = TwoD(random.randint(WIDTH / 2 - 200, WIDTH / 2 +
200),
                             random.randint(HEIGHT / 2 - 200, HEIGHT / 2 +
200))

    def move_agent(self):
        self.position.x += self.velocity * math.cos(self.heading)
        self.position.y += self.velocity * math.sin(self.heading)
        if self.position.x > WIDTH:
            self.position.x -= WIDTH
        else:
            if self.position.x < 0:
                self.position.x += WIDTH
        if self.position.y > HEIGHT:
            self.position.y -= HEIGHT
        else:
            if self.position.y < 0:
                self.position.y += HEIGHT

    def cohere(self, boids):
        # clumping
        avg_pos = TwoD(0, 0)
        for boid in self.getConnections(boids):
            if boid is not self:
                avg_pos += boid.position
        avg_pos /= self.getConnectionsNumber(boids)
        pos_head = math.atan2(self.position.y - avg_pos.y, self.position.x -
avg_pos.x)

        head = turn_towards(self.heading, pos_head)
        return head
```

```
    def align(self, boids):
        avg_head = self.heading
        for boid in self.getConnections(boids):
            if boid is not self:
                avg_head += boid.heading
        avg_head /= self.getConnectionsNumber(boids)

        head = turn_towards(self.heading, avg_head)
        return head

    def adjust_heading(self, boids):
        if self.getConnectionsNumber(boids) != 0:
            closest_head = self.heading
            dist = 1000
            for boid in self.getConnections(boids):
                new_dist = (((self.position.x - boid.position.x) ** 2) + (
                        (self.position.y - boid.position.y) ** 2)) ** 0.5
                if new_dist < dist:
                    dist = new_dist
                    closest_head = boid.heading
            if dist < crowded:
                head = separate(self.heading, closest_head)
            else:
                self.heading = self.align(boids)
                head = self.cohere(boids)
        else:
            head = self.heading
        return head

         def getConnections(self, boids):
        arr = np.zeros(shape=(len(boids) - 1, 2))
        i = 0
        for boid in boids:
            if boid is not self:
                arr[i] = [boid.position.x, boid.position.y]
                i = i + 1

        T = KDTree(arr)
        idx = T.query_ball_point([self.position.x, self.position.y],
r=closest)
        return np.array(boids)[idx]
```

```
    def getConnectionsNumber(self, boids):
        arr = np.zeros(shape=(len(boids) - 1, 2))
        i = 0
        for boid in boids:
            if boid is not self:
                arr[i] = [boid.position.x, boid.position.y]
                i = i + 1

        T = KDTree(arr)
        idx = T.query_ball_point([self.position.x, self.position.y],
r=closest)
        return len(idx)


# Execute the simulation.
if __name__ == '__main__':
    main()
```