Агентно-ориентированная модель распространения инфекции

Рассматриваемая агентная модель состоит из двумерной среды (пространство X-Y), в которой находятся несколько агентов (люди, популяция). Каждый агент имеет определенное положение и скорость. Столкновения с границами среды или другими людьми приводят к изменению направления движения агента, они отскакивают от стен и друг от друга («броуновское движение»). Все люди подвержены вирусной инфекции (здоровые, инфицированные, с иммунитетом). Некоторые люди по своей природе здоровее других, это отражается на показателях их здоровья. Инфекция снижает коэффициент здоровья, причем если этот коэффициент падает ниже 0, то считается, что человек умер от вируса.

Для этого моделирования размеры среды составляют 200 × 250. Местоположение агентов в ней выражается вектором положения X = (x, y), где x и y - горизонтальное и вертикальное положение агента, представленное числами с плавающей точкой. У каждого агента есть скорость V = (vx, vy).

В исходной модели движение агента происходит без ускорения. Однако на перспективу предусмотрим ускорение, поэтому двумерное уравнение для движения определяется как

$$\vec{x}_2 = \vec{x}_1 + \vec{v}_1 t + \frac{1}{2}\vec{a}t^2$$

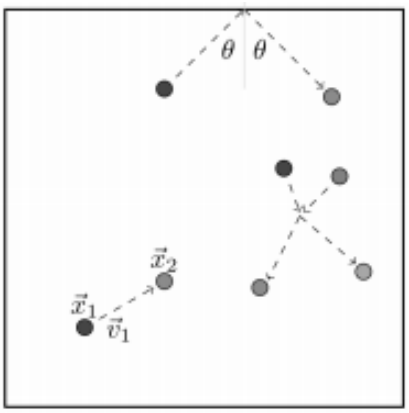
Столкновение агента со стеной приводит к изменению направления движения агента в соответствии с простыми правилами отражения - скорость агента не меняется, но меняется направление. Угол падения ϑ равен углу отражения, и вычисление представляет собой просто изменение знака одной из составляющих скорости. Если агент отскакивает от вертикальной стены, то не изменяется его горизонтальная скорость. Вертикальная скорость остается неизменной, но изменяется с положительного значения на отрицательное. Если бы стены имели какую-либо конфигурацию, то это вычисление было бы несколько сложнее.

Столкновение с другим агентом требует больше математических вычислений. Используется та же концепция, согласно которой агент отскакивает от столкновения, меняя направление, но не скорость. Однако изменение направления - это нечто большее, чем изменение знака составляющей скорости.

В модели агенты не бесконечно малые точки, но имеют радиус ***r***. Все агенты имеют одинаковый радиус. Таким образом, столкновение происходит, когда центры агентов находятся на расстоянии 2***r***. Следует отметить, что это не кинетическое столкновение, импульс и энергия не сохраняются. При кинетическом столкновении скорость агентов изменилась бы.

Агенты здесь представляют людей, каждый из которых обладает своей индивидуальностью. Некоторые люди двигаются быстро в мире, а другие перемещаются медленно. Эти свойства не должны меняться из-за встречи с другим агентом. Таким образом, агент покидает зону столкновения с той же скоростью, что и раньше. Меняется только направление движения.

Столкновение агентов использует идею отражения от теоретической стены, но необходимо определить ориентацию этой стены. Рассмотрим столкновение, на котором две сферы представляют двух агентов. Центры этих сфер соединены стрелкой. Теоретическая стенка перпендикулярна этой стрелке и проходит через точку соприкосновения двух сфер. Угол от горизонтали описывается как α. Нижний шар сталкивается с верхним шаром. Угол отражения от стены ϑ совпадает с углом падения на стену.



Способ вычисления отражения от стены под углом заключается в повороте системы в горизонтальной плоскости, вычислении отражения и последующем повороте системы обратно в исходное положение. Повернутая система поворачивается на угол α, так что входящий агент приближается к стене сверху. Расчет отражения заключается в простом изменении знака ***y***-компонента. Затем система поворачивается на угол α, чтобы вернуться к своей первоначальной ориентации.

Вектор скорости равен V = (vx, vy), содержащий значения горизонтальной и вертикальной составляющих скорости. Если один из агентов заражен, то при столкновении вирус может передаваться другому агенту. Параметры моделирования позволяют управлять вероятностью передачи и иммунитетом.

При кодировании на **Python** можно записать несколько способов реализации агента в скрипте - в этом примере рассмотрим два способа. Первый - определить каждый агент как объект класса. Это упрощает чтение кода, но программа будет работать медленно. Второй подход заключается в использовании массивов **numpy** и их методов обработки, что позволяет создавать быстрый код, но человеку сложнее его читать.

Вариант 1 - Агенты как объекты

Объект класса может содержать связанные переменные и функции, т.е. хорошо подходит для агентно-ориентированной модели. В нашем примере класс содержит информацию для одного агента. Совокупность - это несколько экземпляров этого класса. У одного агента есть свои переменные, каждый агент в этой модели владеет несколькими параметрами:

1) x: двумерный массив для позиции, которая изначально находится в случайном месте;
2) v: двумерный массив для скорости, которая изначально имеет случайные значения от 5 до 10;
3) a: двумерный массив для ускорения, которое изначально равно (0,0);
4) alive: True/False: флажок указывает на то, что агент еще жив;
5) infected: True/False флажок указывает, что агент заражен;
6) immune: True/False флажок указывает, что агент иммунизирован;
7) health: параметр запаса здоровья.
8) bottom: случайное значение от -50 до 100, значение для переносчика болезни, при котором он достигает конца болезни. Если это число положительное, то, когда здоровье агента снижается до этого значения, болезнь заканчивается, и агент становится иммунизирован. Если число отрицательное, то агент обречен на печальный конец - умирает.

У каждого агента также есть несколько функций:

a) move - перемещение агента;
b) LowerHealth - снижает уровень здоровья инфицированного человека;
c) Cured - устанавливает параметры больного, когда он излечивается от болезни;
d) Distance - возвращает расстояние между двумя агентами;
e) Collide - проверяет столкновение двух агентов.

Другие функции модели необходимы для управления популяцией агентов и итерациями моделирования.

Создание класса начинается с конструктора. Функция инициализации автоматически запускается при вызове создания экземпляра класса. В строках задаются рандомизированные переменные для одного пользователя.

```
class Person:
    def __init__(self):
        self.x = np.random.rand(2)*(SPACES-1)  # координаты x-y
        self.v = np.random.rand(2)*VELOCOEF +1  # скорость x-y
        self.a = np.zeros(2)+ASCOEF  # ускорение x-y
        self.alive = True
        self.immune = False
        self.infected = False
        self.health = np.random.randint(HEALTH_MID-30,HEALTH_MID+30) # текущий уровень здоровья
        self.bottom = np.random.randint(-50,self.health-30) # критический уровень здоровья
```

Вывод всех переменных на печать будет неудобным, поскольку для этого требуется выполнение нескольких команд. Python допускает определение функции str, которая создаёт строку, возвращающую пользовательскую строку описания объекта класса. Код str добавлен к классу Person для создания строки, которая будет распечатываться с помощью функции print().

```
def __str__( self ):
        st = '';  temps = vars(self)
        for item in temps:  st += f'{item}: {temps[item]}\n'
        return st
```

Каждому агенту необходимо будет изменить значения, такие как перемещение или взаимодействие с вирусом. Движение определяется уравнением ускоренного движения, а добавление к классу показано в коде. Эта функция будет корректировать положение на основе текущего положения, скорости и ускорения.

Функция перемещения может переместить агента за пределы окружающей среды. В таких случаях агент должен отразиться от стены, изменив знак одной из скоростей. Кроме того, перемещения происходят с дискретными временными интервалами, поэтому при вычислении можно поместить агента за периметр. Но когда агент проходит расстояние d за пределы границы, он помещается на расстояние d внутри границы.

```
    def Move(self, V, H, dt = 0.1):
        self.x[0] = self.x[0] + self.v[0]*dt*np.random.choice([-1,1]) + 0.5*self.a[0]*dt*dt
        self.x[1] = self.x[1] + self.v[1]*dt + self.a[1]*dt*dt
        if self.x[0] < 0:
           self.x[0] *= -1;  self.v[0] *= -1
        if self.x[0] > H:
           self.x[0] = H - 1
           self.v[0] *= -1
        if self.x[1] < 0:
           self.x[1] *= -1;  self.v[1] *= -1
        if self.x[1] > V:
           self.x[1] = V - 1
           self.v[1] *= -1
```

Часть границ показана в коде. Предполагается, что один угол в среде равен (0,0), а другой определен как (V,H). Эти два значения являются входными данными для функции. В строках показаны отражения для переменной x[0] и аналогичный код для переменной x[1]. В строке 3 рассматривается случай, когда x[0] стало меньше 0, то в строке 4 указывается местоположение агента, и изменяется скорость.

Если агент заражен, то его уровень здоровья снижается с каждой итерацией. Код На рис. 17.6 показано меньшее количество здоровья, которое снижает здоровье агента, если агент заражен, а уровень здоровья по-прежнему выше нижнего (bottom).

```
    def LowerHealth(self):
        # снижение здоровья от болезни
        if self.infected and self.health > self.bottom:
            self.health -= 1
        # если здоровье закончилось то вычеркиваем
        self.alive = (self.health > 0)
```

Удачливые агенты, у которых положительное bottom значение, могут быть излечены от болезни. Когда уровень здоровья падает ниже нижнего, флаг заражения становится False, указывая на то, что агент больше не заражен. Кроме того, иммунитет установлен на значение True, предотвращающее повторное заражение у агента.

Некоторые агенты не излечиваются. Если их нижнее значение отрицательное, то в конечном итоге их здоровье упадет ниже 0. Когда это происходит, их флагу жизни присваивается значение False.

```
    def Cured(self):
        global Inf_count
```

```
        # можем вылечиться
        if self.health <= self.bottom:
           self.infected = False
           Inf_count -=1
           self.immune = True
           self.bottom -= 5
        # можем поднять здоровье если есть иммунитет
        if self.immune:
           self.health += np.random.choice([1,0])
```

После перемещения агента ему необходимо определить, не столкнулся ли он с другим агентом. Это происходит, когда расстояние между центрами двух агентов меньше, чем 1. Функция Distance вычисляет расстояние от рассматриваемого агента до второго агента, определенного в качестве входного параметра. Эта функция просто вычисляет расстояние, но не принимает никаких решений.

```
    def Distance(self, you):
        return np.sqrt(((self.x - you.x)**2).sum())
```

Функция Collide преобразует теорию столкновения в сценарий на Python. Эта функция вызывается дважды, чтобы изменить поведение обоих участников столкновения. Входными данными для функции Collide являются данные другого участника столкновения. В строках 3 и 4 вычисляются расстояния по вертикали и горизонтали между двумя агентами, а в строке 5 вычисляется угол α, определенный в теории столкновений. В строке 6 создается матрица поворота, и строка 7 применяет ее. Строка 8 вычисляет отражение, а строки 10 и 11 возвращают систему в исходное пространство. Строки 15 и 16 переносят инфекцию, если условия подходящие. Если при моделировании нужно, чтобы скорость передачи инфекции была другой, то надо изменить строку `self.infected = … .`

```
   def Collide(self,you):
        # проверка столкновения персон
        ypart = you.x[1] - self.x[1]
        xpart = you.x[0] - self.x[0]
        alpha = np.arctan2(ypart, xpart)
        R = np.array(((np.cos(-alpha),-np.sin(-alpha)),
                      (np.sin(-alpha),np.cos(-alpha))))
        vrot = R.dot(self.v)
        wrot = vrot * np.array((1,-1))
        R = np.array(((np.cos(alpha),-np.sin(alpha)),
                      (np.sin(alpha),np.cos(alpha))))
        w = R.dot(wrot)
        self.v = w + 0.05
        # передача болезни
        if you.infected:
           if self.immune:
              self.infected = (np.random.ranf() > IMUNOPOWER)
              self.immune = not self.infected
           else:
              self.infected = True
```

За одну итерацию каждый агент перемещается, и вычисляются события, связанные с любым столкновением. Шаги следующие:

1. Переместим агента,

2. Отразим агента от стен, если необходимо,

3. Понизим уровень здоровья зараженных агентов,

4. Определим, удалось ли каким-либо агентам победить вирус,

5. Определим, погибли ли какие-либо агенты,

6. Вычислим последствия столкновения.

Функция Iterate выполняет одну итерацию проверок агентов при моделировании. Функция является внешней по отношению к классу Person. Входными данными являются агенты. `XLim` и `YLim` - это пространственные границы окружающей среды. Переменная people представляет собой список агентов. Каждый агент рассматривается в цикле, начиная со строки 3. Если этот агент все еще жив (строка 4), то вычисляется его перемещение (строка 5). В строке 6 - проверка изменения состояния здоровья. В строке 7 проверяются условия иммунитета и болезни.

```
def Iterate():
    global R_COMM, CollideCount, XLim, YLim, people, Inf_count
    #
    for me in people:
        if me.alive:
           me.Move(YLim, XLim, dt=0.11)
           me.LowerHealth()
           me.Cured()
           # проверка встречи!
           for you in people:
              if you.alive and me != you:
                  if me.Distance(you) < R_COMM:
                     CollideCount +=1
                     me.Collide(you)
                     you.Collide(me)
        if me.infected: Inf_count +=1
```

Далее идёт проверка столкновений агента. В строке 9 начинается процесс рассмотрения агентов при столкновении. Текущий агент (me) сравнивается с каждым из других агентов. Если другой агент активен и расстояние между двумя агентами меньше `R_COMM`, то агенты сталкиваются. Обоим агентам необходимо изменить свои скорости, и это обрабатывается в строках Collide().

Функция CreateWorld создаёт популяцию агентов. Входными данными являются количество агентов и протяженность окружающей среды. Эта функция создает случайное распределение агентов. В строке 2 создаётся `NPeople` агентов, а в строке 3 заражаются первые `INI_ILL агентов`.

```
def CreateWorld(NPeople=100):
    people = [Person() for _ in range(NPeople)]
    # задаем несколько больных
    for p in people[:INI_ILL]:  p.infected=True
    return people
```

Моделирование выполняется в функции main. Итерации выполняются в цикле for. Функция Iterate перемещает агентов и выполняет действия, связанные с конфликтами. В массиве `LivInfIm` "живые" и "зараженные" будут отображаться количества живых и инфицированных людей после каждой итерации. Списки `pos_liv` будут отображены в конце моделирования.

```
    np.random.seed(SEED)
    # длительность наблюдения мира
    N_ITERS=250
    # создаем мир персон
    people = CreateWorld(POPULATION)
    # массив счетчиков = здоровые-больные-имунные
    LivInfIm = np.zeros((3,N_ITERS), dtype=int)
    # массив популяций для графика
    pos_liv=np.ones((3,N_ITERS), dtype=list)
    # запускаем модель,сбор данных,график
    for step in range(N_ITERS):
        Iterate()
        CollectData(step)
    plotgrafiks()
```

При анализе и моделировании второй волны инфекций необходимо будет внести некоторые изменения. Не все агенты могут быть инфицированы в ходе первой волны, или иммунитет не сохраняется навечно. Это условие потребует к классу Person добавить ещё одну переменную для контроля продолжительности действия иммунитета.

В этой программе есть несколько переменных, которые можно изменять для получения различных результатов моделирования.

- NPeople - количество агентов в начале моделирования, увеличение количества агентов также увеличивает их плотность в окружающей среде, что может существенно повлиять на уровень заражения.

- NIters - количество итераций моделирования.

- X и Y - это границы пространства, при увеличении плотность агентов уменьшается, а также создается прямоугольное пространство, ограничивающее перемещения в одном измерении.

- Скорость агента - это случайное значение в диапазоне от 5 до 10. При увеличении диапазона агенты будут чаще сталкиваться. Если потребуется учитывать социальное дистанцирование, то скорость распространения между агентами снизится.

- Ускорение в данном примере не использовалось, его можно использовать для замедления заражения возбудителей или ускорения тех, кто игнорирует социальное дистанцирование.

- Коэффициент здоровья составляет от 50 до 150. Если этот диапазон увеличить, то для восстановления зараженного агента потребуется больше времени.

- коэффициент bottom - это значение от -50 до 100, что означает, что около 1/3 заражённых агентов погибнет. При увеличении этого диапазона увеличивается количество агентов, которые могут выжить после заражения вирусом.

- Распределение агентов является однородным, что означает, что все имеют примерно одинаковое количество соседей. На этапе инициализации могут быть созданы популяции с другой плотностью.

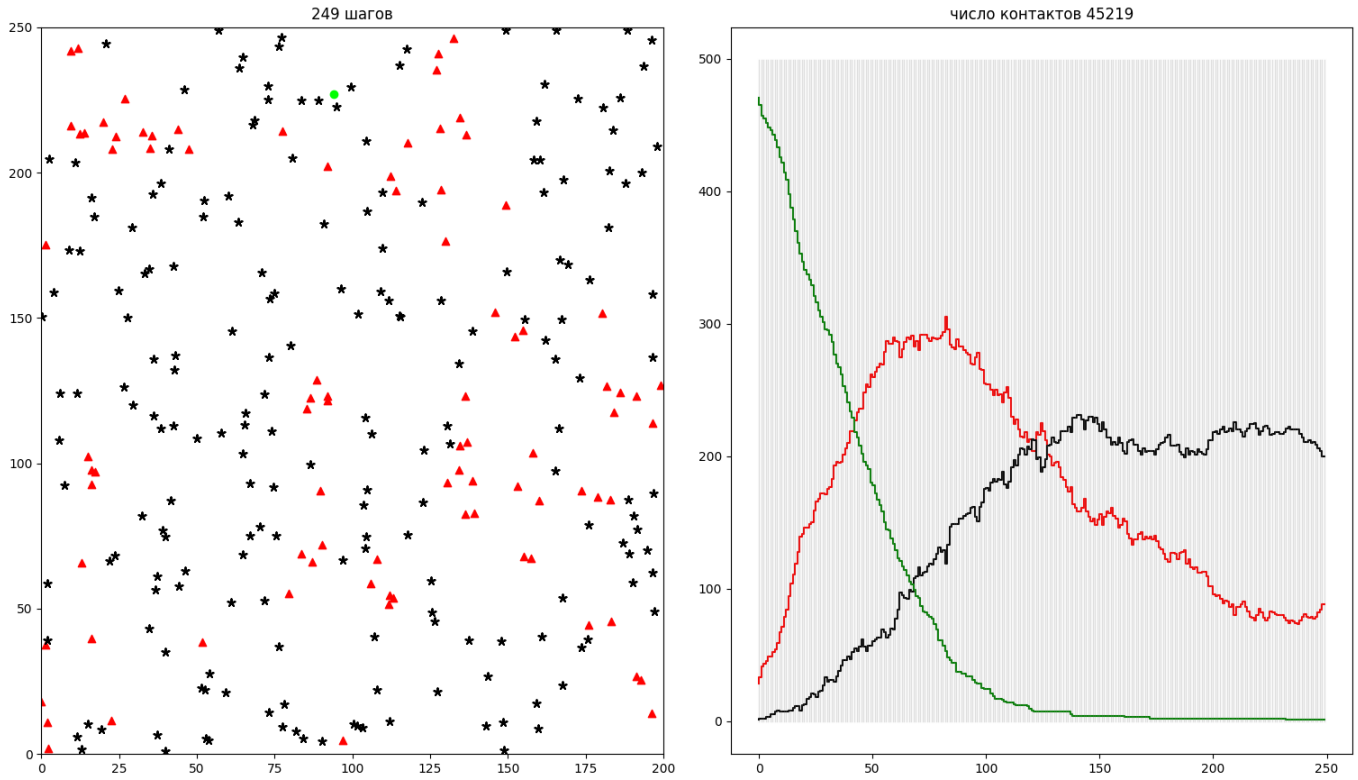


Рисунок 1 – Геометрическое положение и графики количества инфицированных и живых агентов.

Рассмотрим случай, когда при моделировании необходимо использовать влияние вместимости больницы. Если одновременно заражено слишком много пациентов, больницы перегружены и не могут оказать помощь

всем инфицированным. Это может быть смоделировано путем снижения минимального коэффициента для инфицированных агентов, количество которых превышает установленное пользователем значение для пропускной способности больницы.

В другом случае можно смоделировать, что в популяции имеется большое количество агентов, подверженных заражению. Это моделируется путем уменьшения диапазона значения bottom, поскольку это приведет к увеличению числа погибших инфицированных агентов.

В другом случае можно смоделировать эффект высокой плотности населения. Это зависит от начальных положений и скоростей некоторых агентов. Для области с высокой плотностью размещения установите расположение нескольких агентов в пределах небольшой области и уменьшите их скорости, чтобы они не могли легко покинуть эту область.

Чтобы имитировать наличие вакцины средства, в определенный момент времени для случайных агентов увеличивают минимальное значение health. С течением времени все больше агентов получают лекарство, и их минимальные значения health увеличиваются.

В модели считается, что выздоровевшие агенты обладают иммунитетом. В одном из сценариев по истечении определенного периода времени флага иммунитета может быть установлен в значение False. В другой симуляции флаг иммунитета может быть установлен не для всех агентов. Возможно, у вылеченного агента вероятность развития иммунитета составляет 75%. Это позволит агентам многократно заражаться этой болезнью.

Вариант 2 - Ускорение вычислений

Одна из проблем заключается в том, что некоторые подходы к кодированию могут привести к замедлению вычислений. В предыдущем варианте было смоделировано 500 агентов на 250 итераций. Для стандартного вычисления потребовалось около 100 секунд вычислений. Хотя 100 секунд - это не так уж плохо, с увеличением числа агентов время будет расти в геометрической прогрессии, поскольку каждый агент должен взаимодействовать с другими агентами для определения столкновений. Это двойной вложенный цикл, который значительно сокращает время вычислений по мере увеличения числа агентов. Такой код хорошо работает для небольших групп, но будет плохо работать для больших групп.

Для увеличения скорости вычислений используем новую структуру данных. Каждый агент содержит три типа данных - плавающие значения для местоположения, скорости и ускорения. У агентов есть бинарные значения для флагов "жив", "заражен" и "иммунен", и есть целочисленные значения их здоровья и значения bottom.

Эта структура данных будет зависеть от скорости работы функций **numpy**, что означает, что данные должны содержаться в массивах. Создаются три матрицы. Каждая строка в матрице связана с одним агентом. Таким образом, 10-й агент будет связан со значениями в 10-й строке каждой матрицы.

Первая матрица - это M, которая содержит значения с плавающей запятой. Количество строк - это количество агентов, а столбцов шесть: по два в каждом для определения положения, скорости и ускорения.

Матрица B содержит двоичный по два для определения местоположения, скорости и ускорения. Матрица B содержит двоичные данные, которые являются флагами "жив", "заражен" и "иммунен".

Матрица I содержит целые значения для здоровья и уровни bottom.

Эти матрицы создаются с помощью функции InitVirusMx. В строках с 3 по 5 создаются три матрицы, но в них пока нет данных. Линия 6 устанавливает в первых двух столбцах матрицы M случайные значения в диапазоне от 0 до 100. Это вертикальное и горизонтальное расположение агентов. Строка 7 устанавливает начальная скорость всех агентов. В строках 8 и 9 задаются случайные числа для здоровья агента и его минимальных значений. В строках с 10 по 12 устанавливаются флажки, указывающие на то, что каждый агент жив, здоров, но не обладает иммунитетом.

```
def InitVirusMx(NPeople = 100, space=99):
    M = np.zeros((NPeople, 6)) # x1,x2, v1,v2, a1,a2
```

```
    # расставим людей в мире
    M[:,:2] = np.random.ranf((NPeople, 2)) * space
    M[:,2:4] = np.random.ranf((NPeople, 2)) * 5 + 5
    #
    I = np.zeros((NPeople, 2), int) # запас здоровья
    I[:,0] = np.random.randint(75, 150, NPeople)
    I[:,1] = np.random.randint(-50, 100, NPeople)
    #
    B = np.zeros((NPeople,3), bool) # alive, infected, immune
    B[:,0] = True # все живы
    B[:,1] = False # не заражен
    B[:,2] = False # без иммунитета
    return M, B, I
```

Далее показана функция CollideM, которая контролирует столкновение агентов. Математика для этой функции представлена ранее.

```
def CollideM(M,B,me, him):
    #print(me,him)
    alpha = np.arctan2(M[him,1] - M[me,1], M[him,0] - M[me,0])
    R = np.array(((np.cos(-alpha),-np.sin(-alpha)),(np.sin(-alpha),np.cos(-alpha))))
    vrot = R.dot(M[me,2:4])
    wrot = vrot * np.array((1,-1))
    R = np.array(((np.cos(alpha),-np.sin(alpha)),(np.sin(alpha),np.cos(alpha))))
    w = R.dot(wrot)
    M[me,2:4] = w + 0
    # перенос болезни
    if B[him,1] and not B[me,2]:
       B[me,1] = True # (if np.random.random()>0.25 else False)
```

Функция IterateM вычисляет одну итерацию и следует той же логике, что и ранее. Строка 5 вычисляет новое местоположение. Строки с 6 по 11 удерживают агента в пределах одного направления. Тот же код для других направлений не используется. показано (строка 12) для экономии места. В следующих строках вычисляется расстояние от агента до других агентов и определяется, сталкиваются ли они.

```
def IterateM(M,B,I,space = 300):
    dt = 0.1
    X = Y = space
    alive = B[:,0].nonzero()[0] # счет живых
    for me in alive:
        # перемещение в мире
        M[me,:2] = M[me,:2] + M[me,2:4] *dt + 0.5*M[me,4:6] *dt**2
        # проверка границ мира
        if M[me,0] < 0:
            M[me,0] *= -1;  M[me,2] *= -1
        if M[me,0] > X:
            M[me,0] = X - (M[me,0] -X)
            M[me,2] *= -1
        if M[me,1] < 0:
            M[me,1] *= -1;  M[me,3] *= -1
        if M[me,1] > Y:
            M[me,1] = Y - (M[me,1] -Y)
            M[me,3] *= -1
        # проверка уровня здоровья
        if B[me,1] and I[me,0] > I[me,1]:
           I[me,0] -= 1
        # излечение
        if I[me,0] <= I[me,1]:
           B[me,1:3] = False, True
        # вычисление расстояния до других
        them = list(alive)
        them.remove(me)
        them = np.array(them)
```

```
        dist = np.sqrt(((M[me,:2] - M[:,:2])**2).sum(1))
        # дистанция заражения < 4
        closendx = (dist < 4).nonzero()[0]
        # проверка столкновений людей
        for i in closendx:
            if i in them:
                CollideM(M,B,me,i)
                CollideM(M,B,i,me)
        # проверка смерти
        ndx = (I[:,0] <= 0).nonzero()[0]
        B[ndx] = False, True, False
```

Ниже показано код запуска моделирования. На этот раз число агентов равно 1000, и количество итераций равно 250. В строке 3 создаются матрицы. Цикл **for** выполняет итерации и сохраняет количество инфицированных и живых агентов в их массивах.

```
def RunMatrixVirus(NPeople=1000,N_ITERS=250):
    space = 250
    M, B, I = InitVirusMx(NPeople,(space-1))
    B[:10,1] = True # 10 персон больны
    inf = np.zeros(N_ITERS)
    aliv = np.zeros(N_ITERS)
    immun = np.zeros(N_ITERS)
    clean = np.zeros(N_ITERS)
    for step in range(N_ITERS):
        IterateM(M, B, I, space)
        inf[step] = int(B[:,1].sum())
        aliv[step] = int(B[:,0].sum())
        immun[step] = int(B[:,2].sum())
        clean[step] = aliv[step]-inf[step]-immun[step]
    #
    return inf, clean, immun
```

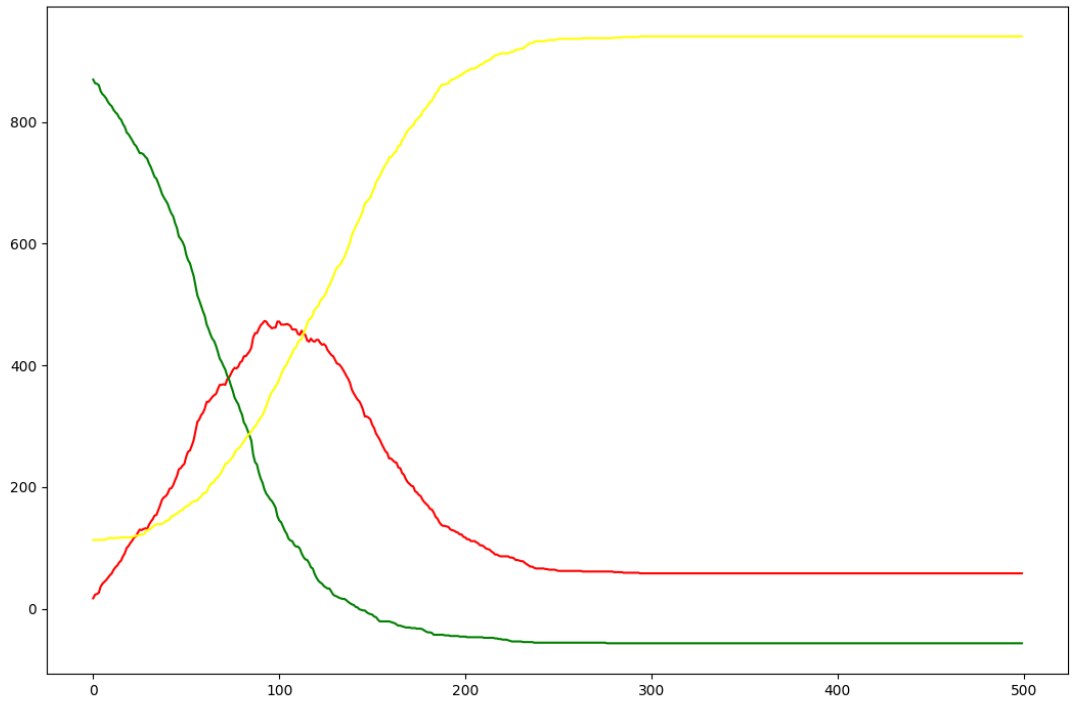


Рисунок 2 - Графики количества инфицированных и живых агентов.

Результаты первых 200 итераций в обеих вариантах модели одинаковы. Основное различие заключается в том, что в первой модели для выполнения 200 итераций потребовалось 100 секунд, а во второй версии для

выполнения 500 итераций потребовалось 15 секунд. Причина в строке цикла, в которой вычисляется расстояние от одного агента до всех агентов за одну строку Python. Это все еще процесс с вложенным циклом, но один из циклов вложен в функции **numpy**, и, следовательно, он значительно быстрее.

Однако из-за этого сценарий сложнее читать. Аналитику нужно будет запомнить, что представляет собой каждый из столбцов в матрицах. Кроме того, значения агентов более подвержены путанице. Строка ***n*** в каждой матрице соответствует агенту ***n***. Однако нет механизма, который бы на самом деле связывал их воедино. Можно сдвинуть данные в одной матрице на одну строку и полностью испортить все моделирование. Таким образом, нужно проявить больше осторожности и убедиться, что данные для каждого агента находятся в нужном месте. Увеличение скорости почти в 20 раз стоит затраченных усилий.

Задания

В примере представлена простая модель передачи данных в среде, использующей моделирование на основе агентов (Agent-Based Model). Каждый агент представляет человека и обладает свойствами передвижения, здоровья и текущего статуса. Эти агенты перемещаются в замкнутом пространстве, и при их столкновении существует вероятность того, что инфицированный человек может передать инфекцию неинфицированному человеку. К каждому возбудителю можно приобрести иммунитет. Болезнь отрицательно сказывается на здоровье агента, и если оно падает ниже порогового значения, то агент считается мертвым.

Такое моделирование не учитывает многие важные особенности реалистичного моделирования, например как экономические последствия, воздействие на психическое здоровье, неоднородная среда, неоднородное поведение агентов и т.д. Однако оно демонстрирует основные принципы построения моделирования.

Представлены два варианта реализации Python. В первом варианте в качестве структуры данных используется класс, который обеспечивает простое чтение кода, но работает медленно. Вторая реализация использует инструментарий **numpy**, что намного быстрее, однако вторую реализацию сложнее читать и поддерживать.

Эти модели предназначены исключительно для образовательных целей. В них не учитываются важные факторы, которые необходимо учитывать при моделировании реальности. Не используйте результаты, полученные здесь, в качестве фактора, определяющего ваши личные решения!

Упражнения основаны на модификациях модели.

1. Проведите 3 моделирования, в каждом из которых задействовано 500 агентов и 250 итераций. Различия обусловлены состоянием потока случайных чисел. Покажите графики популяции для трех случаев. Прокомментируйте сходства и различия в этих трех тестах.

2. Смоделируйте режим "не выходить из дома". Запустите моделирование для 300 агентов и 300 итераций. На 50-й итерации вводится режим "не выходить из дома". Сохраните все текущие скорости для последующего использования. Установите для всех скоростей значение 0. Выполните 100 итераций, а затем с 150-й итерации отмените режим изоляции, восстановив сохранённые значения скоростей. Постройте график результатов.

3. Запустите 3 моделирования, в каждом из которых задействовано 500 агентов:

- исходное моделирование с 200 итерациями;

- моделирование, в которой на 50й итерации был введен порядок "не выходить из дома" и отменен на 150-й итерации;

- моделирование режима "не выходить из дома", в котором приказ "не выходить из дома" был введен на 35-й итерации и отменен на 135-й;

Эффективен ли в этом моделировании порядок "не выходить из дома"? Объясните свой ответ.

4. В исходном моделировании переменная bottom была рассчитана таким образом, что около 1/3 агентов погибнет от вируса. Измените это значение примерно на 2/3. Запустите моделирование и прокомментируйте, чем результаты отличаются от исходного моделирования.

5. Смоделируйте распространение вируса, если после 30 итераций была создана вакцина, обеспечивающая мгновенный иммунитет для 90% населения (выбранного случайным образом). Возможно, у некоторых людей уже была прививка (иммунитет), но они все равно получили прививку. Запустите такую модель и покажите графики. Прокомментируйте эффективность вакцины на основе данных моделирования.

6. Повторите предыдущий эксперимент с 50%-ной эффективностью вакцины. Сравните эти результаты с предыдущей задачей.

7. Оцените эффективность при наличии вакцины. Проведите моделирование, в котором вакцина эффективна на 90%, но не применяется до 100-й итерации. Сравните это со случаем, в котором вакцина была эффективна на 90% и доступна на 100-й итерации.

8. Создайте модель, в которой 300 агентов и 200 итераций. Однако все агенты запускаются в ограниченной области размером 150 × 150. Запустите симуляцию и постройте график результатов.

9. Рассмотрите эффект от ношения маски. В этой симуляции маски не предохраняют человека от заболевания, поскольку вирус может проникнуть в организм другими путями, кроме дыхательных. Маски предотвращают распространение вируса. Проще говоря, любой агент, носящий маску в этой модели, не может передавать вирус. Запустите модель с 400 агентами с масками и 400 итерациями. В исходной модели первые 10 агентов в популяции были изначально инфицированы. Выведите результаты на график.

10. В этой модификации предыдущего задания первые 50 человек решили не надевать маски. В эту группу из 50 человек входят изначально инфицированные люди. Запустите модифицированную модель и выведите результаты на график.

В исходном примере ускорение не использовалось. Его можно использовать для замедления заражения от переносчиков или ускорения тех, кто игнорирует социальное дистанцирование.

Имитация открытия общественных заведений означает привлечение некоторых агентов в определенное место. Для этого можно изменять ускорение, и, возможно, направление, но не обязательно скорость, движения к кинотеатру или пляжу.

Комментарий к реализации модели

Entity Component System (ECS) - это архитектурный паттерн, набирающий все большую популярность у разработчиков игр и тренажеров, он отлично подходит для описания динамического виртуального мира. Из-за его особенностей, некоторые считают его чуть ли не новой парадигмой программирования, это скорее не так, но мозг программиста перестраивать потребуется.

ECS базируется на следующих 3 определениях: entity (сущность), component (компонент) и system (система). Собственно, акроним ECS и есть сокращение этих 3 слов по их первым буквам.

ECS возводит в абсолют принцип Composition Over Inheritance (композиция важнее наследования) и может являться частным примером Data Oriented Design (ориентированного на данные дизайна, DOD), однако это уже зависит от интерпретации паттерна конкретной реализацией.

Сущность - сущность, максимально абстрактный объект. Условный контейнер для свойств, определяющих чем будет являться эта сущность. Зачастую представляется в виде идентификатора для доступа к данным. Это может быть любой объект в игровом мире. Например, это может быть юнит игрока, кнопка в интерфейсе, событие с данными от одной системы к другой и т.п. Сущности сами по себе не имеют свойств и выступают контейнерами для компонентов.

Основная идея ECS - переход от стандартного ООП подхода Inheritance (Наследование) к Composition over Inheritance. Это позволяет смешивать различные поведения без необходимости ломания родительского класса - вместо иерархии наследования мы разделяем данные на изолированные блоки (Компоненты) и набираем из них как из кубиков нужную нам информацию об игровом объекте, добавляя компоненты на сущности.

Компоненты не должны иметь какой-то бизнес-логики в своей обработке и содержат только данные - это очень важно и является ключевой особенностью по расширению функционала в дальнейшем. Это не означает, что ссылочные типы запрещены (которые тоже могут быть данными, сервисами или кешами других данных), или компоненты не могут содержать вспомогательной обслуживающей логики (это часто используется для корректной очистки/реинициализации полей, пулинга и т.п.) - это базовая рекомендация по отсутствию основной бизнес-логики в типах данных компонентов.

В паттерне ECS роль обработчиков берут на себя Системы - блоки кода, которые каким-либо образом обрабатывают требуемый набор компонентов на сущностях. Системы не содержат никаких локальных данных или ссылок на сущности или компоненты (за исключением кеширования для повышения производительности), они служат только для обработки потока отфильтрованных по их условиям сущностей и компонентов, привязанных к сущностям. Системы в ECS - это только логика обработки данных.

Такой подход - разделение логики и данных - позволяет комбинировать любое сложное поведение набором простых систем, каждая из которых будет обрабатывать только нужные этой конкретной системе данные на сущностях. В результате мы сможем добавлять и удалять системы с минимальными изменениями в других системах, меняя поведение всего игрового мира.

Поведение объекта в ECS определяется не интерфейсами/контрактами/публичным API, как в классическом объектно-ориентированном программировании (ООП), а присвоенными объекту свойствами с данными + существующей отдельно логикой обработки. В ECS данные определяют всё - это и есть главное свойство, которое выделяет его на фоне других подходов к разработке: всё есть данные. И свойства объекта, и его характеристики, и даже события - всё это просто данные существующие в ECS-мире. Логика же является просто конвейерной обработкой всех этих данных. В некотором приближении, ECS можно сравнить с базой данных, которая обрабатывается каждый кадр потоком обработчиков, написанных в стиле процедурного программирования.