



МИНОБРНАУКИ РОССИИ

**Федеральное государственное бюджетное образовательное
учреждение высшего образования
«САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭКОНОМИЧЕСКИЙ УНИВЕРСИТЕТ»**

Факультет информатики и прикладной математики
Кафедра прикладной математики и экономико-математических методов

**ОТЧЁТ
по дисциплине:
«Имитационное моделирование»**

Студента: Бронникова Егора Игоревича

Курс: 4

Группа: ПМ-1901

Форма обучения: очная

Форма представления на кафедру выполненных заданий:
отчёт в электронной форме

Оценка по результатам текущего
контроля (КТ№3)

Санкт-Петербург
2022

Содержание

Введение. Моделирование случайных величин	3
Треугольное распределение	3
Датчики - равномерное распределение	6
Датчик - экспоненциальное распределение	12
Датчик - нормальный закон	15
Датчик - треугольное распределение	19
Метод Монте-Карло	22
Дискретные СВ и случайные события	24
Проверка качества датчиков	26
Моделирование СМО	27
Модель СМО в AnyLogic	27
Одноканальная СМО	31
Hold	35
СМО с отказами	37
Проект СМО	40
Приёмы работы с элементами презентации	43
Движение между двумя ограниченными линиями	43
Движение по окружности и спирали	44
Движение по периметру	46
Реплицированные объекты. Заполнение областей	48
Программный доступ к элементам презентации	51
Библиотека моделирования потоков	54
Моделирование потоков	54
Пешеходная библиотека	58
Пешеходная библиотека. Первая модель	58
Проект «Станция метро»	60
Внешние данные. Табличные функции	63
Чтение из Excel – фигуры	63
Фигуры из презентации в файл MS Excel	65
Системная динамика	67
Модель развития социального стресса	67

Агентные модели. Дискретное пространство	70
«Жизнь» Конвея	70
Модель сегрегации Шеллинга в AnyLogic	72
Модель сегрегации Шеллинга в Python	75
Диаграммы состояний и события	79
Светофор	79
SIR – агентная модель	81
Ноутбук + зарядка	84
SIERD + вакцинация	86
Диаграмма действий	88
Сумма ряда – экспонента	88
Сумма ряда – синус	90
Агентные модели	92
Подготовка к зачёту	92
Связи агентов – добавление, удаление, список, количество	95
Работа с ГИС-картами	100
Доставка с покупками	100

Введение. Моделирование случайных величин.

Треугольное распределение

Задание:

Получить аналитическое выражение для функции распределения $F(x)$ треугольного закона с параметрами a (*min*), b (*max*), c (*мода*). Найти аналитическое выражение для обратной функции $F^{-1}(x)$.

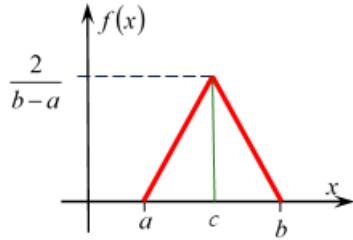


Рис. 1: График плотности треугольного закона

Решение:

Плотность распределения $f(x)$

Уравнение по двум точкам:

$$\frac{y - y_1}{y_2 - y_1} = \frac{x - x_1}{x_2 - x_1} \Rightarrow y = \frac{x - x_1}{x_2 - x_1} \cdot (y_2 - y_1) + y_1$$

1) Рассмотрим случай $x \in [a, c]$:

Точки: $(a, 0), \left(c, \frac{2}{b-a}\right)$

$$f_1(x) = \frac{x - a}{c - a} \cdot \left(\frac{2}{b-a} - 0\right) + 0 \Rightarrow f_1(x) = \frac{2(x - a)}{(b - a)(c - a)}$$

2) Рассмотрим случай $x \in [c, b]$:

Точки: $\left(c, \frac{2}{b-a}\right), (b, 0)$

$$f_2(x) = \frac{x - c}{b - c} \cdot \left(0 - \frac{2}{b-a}\right) + \frac{2}{b-a} \Rightarrow f_2(x) = \frac{2(b - x)}{(b - c)(b - a)}$$

Плотность треугольного распределения:

$$f(x) = \begin{cases} \frac{2(x-a)}{(b-a)(c-a)}, & x \in [a, c] \\ \frac{2(b-x)}{(b-c)(b-a)}, & x \in [c, b] \\ 0, & x \notin [a, b] \end{cases}$$

Функция распределения $F(x)$

По свойству функции распределения: $f'(x) = F(x)$.

1) Рассмотрим случай $x \in [a, c]$:

$$F_1(x) = \int_a^x \frac{2(t-a)}{(b-a)(c-a)} dt = \frac{t(t-2a)}{(b-a)(c-a)} \Big|_a^x = \frac{(x-a)^2}{(b-a)(c-a)}$$

2) Рассмотрим случай $x \in [c, b]$:

При вычислении функции распределения на этом участке следует помнить, что у нас был предшествующий отрезок от $[a, c]$, тогда нужно посчитать его площадь и прибавить к $F_2(x)$.

$$F_1(c) = \frac{c-a}{b-a}$$

$$\begin{aligned} F_2(x) &= \frac{c-a}{b-a} + \int_c^x \frac{2(b-t)}{(b-c)(b-a)} dt = \frac{c-a}{b-a} + \frac{t(2b-t)}{(b-c)(b-a)} \Big|_c^x = \\ &= \frac{c-a}{b-a} + \frac{(2b-c-x)(x-c)}{(b-a)(b-c)} = 1 - \frac{(x-b)^2}{(b-a)(b-c)} \end{aligned}$$

Функция распределения треугольного закона:

$$F(x) = \begin{cases} 0, & x < a \\ \frac{(x-a)^2}{(b-a)(c-a)}, & x \in [a, c] \\ 1 - \frac{(x-b)^2}{(b-a)(b-c)}, & x \in [c, b] \\ 1, & x > b \end{cases}$$

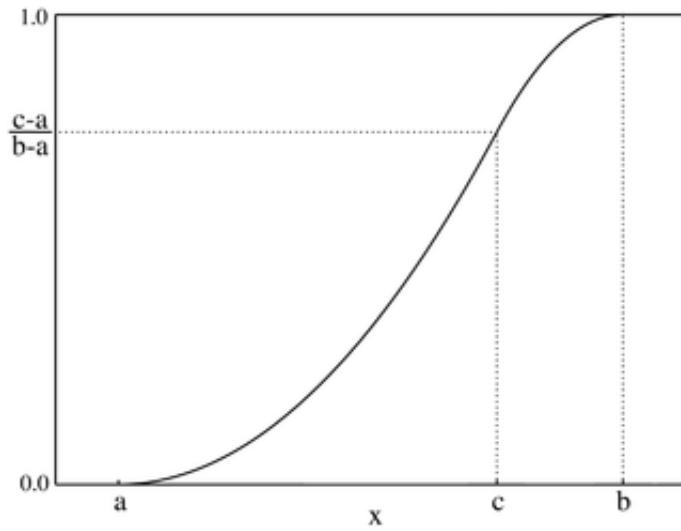


Рис. 2: График функции распределения треугольного закона

Обратная функция $F^{-1}(x)$

Пусть $y = F(x)$.

У функции распределения есть ограничение: $0 < y < 1$, которое стоит учитывать.

1) Рассмотрим случай $0 < y < \frac{c-a}{b-a}$:

$$F_1^{-1}(y) = x_1 = a + \sqrt{(b-a)(c-a)y}$$

2) Рассмотрим случай $\frac{c-a}{b-a} \leq y < 1$:

$$F_2^{-1}(y) = x_2 = b - \sqrt{(b-a)(b-c)(1-y)}$$

Обратная функция $F^{-1}(x)$:

$$F^{-1}(y) = x = \begin{cases} a + \sqrt{(b-a)(c-a)y} & , 0 < y < \frac{c-a}{b-a} \\ b - \sqrt{(b-a)(b-c)(1-y)} & , \frac{c-a}{b-a} \leq y < 1 \end{cases}$$

Датчики - равномерное распределение

Задание:

Реализовать генератор случайных чисел, используя метод серединных квадратов (фон Нейман). Проанализировать свойства полученной последовательности.

Решение:

В методе серединных квадратов изначально задаётся количество разрядов числа k и начальное значение R_0 . Далее число R_0 возводится в квадрат и из середины квадрата числа берётся k -значное число, которое снова возводится в квадрат, и так далее.

Обязательным условием является то, что количество разрядов k должно быть чётным числом.

Данный алгоритм был реализован на языке программирования Python.
(Рисунок 3)

```
def mid_square_method(init, digit, n = 10):
    if digit % 2 != 0: return

    r = init
    res = [r]
    mid_d = digit//2
    for i in range(n):
        r *= r
        digits = list(str(r))
        while len(digits) < digit:
            digits = ['0'] + digits
        r = int(''.join(digits[1:-1]))
        res.append(r)
    return res
```

Рис. 3: Реализация метода серединных квадратов

Алгоритм был запущен с параметрами $k = 10$, $R_0 = 31$, $n = 10$, где n – это количество сгенерированных случайных чисел. Можно проследить, что при достаточно малых разрядах и малом начальном значении быстро получается вырождение, что плохо. (Рисунок 4)

```
res = mid_square_method(31, 4, 10)
res
```

[31, 96, 21, 44, 93, 64, 9, 8, 6, 3, 0]

```
plt.hist(res)
plt.show()
```

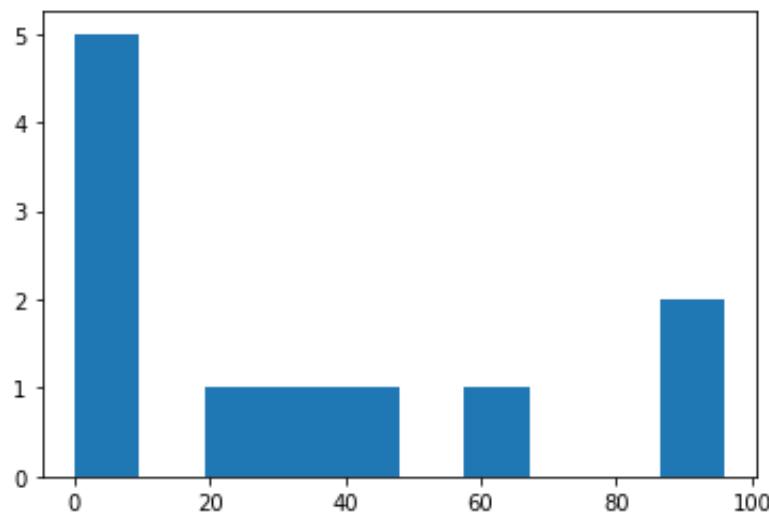


Рис. 4: Результаты генерации случайных чисел методом серединных квадратов

Также стоит проверить гипотезу о том, что получившиеся значения распределены равномерно. Для этого воспользуемся критерием Колмогорова-Смирнова. (Рисунок 5)

Проверка гипотезы

```
sp.stats.kstest(res, lambda x: sp.stats.uniform.cdf(x, loc=0, scale=100))
```

KstestResult(statistic=0.3645454545454545, pvalue=0.08123628783386883)

Рис. 5: Результаты теста Колмогорова-Смирнова

Значение $p\text{-value} > 0.05$, значит мы принимаем гипотезу о том, что данная выборка имеет равномерное распределение.

Задание:

Реализовать линейный конгруэнтный датчик случайных чисел. Сгенерировать последовательность вещественных чисел, распределённых равномерно: 1) на интервале $[0,1)$; 2) на интервале $[a,b)$. Проанализировать полученные последовательности. Определить период, построить гистограмму.

Решение:

Линейный конгруэнтный метод – это один из рекуррентных методов генерации случайных чисел. Следующий элемент последовательности может быть найден по следующей формуле:

$$r_{i+1} = (k \cdot r_i + b) \bmod M$$

Линейная конгруэнтная последовательность, определённая числами M, k, b, r_0 периодична с периодом, не превышающим M . При этом длина периода равна M тогда и только тогда, когда:

1. числа b и M взаимно простые;
2. $k - 1$ кратно p для каждого простого p , являющегося делителем M ;
3. $k - 1$ кратно 4, если M кратно 4.

Сначала был реализован алгоритм для интервала $[0;1)$ на языке программирования Python. (Рисунок 6)

```
def linear_congruent_gauge_0_1(init, k, b, M, n):
    r = init
    unique = 0
    res = []
    for i in range(n):
        r = (k*r + b) % M
        if r/M not in res:
            unique += 1
        res.append(r/M)
    return res, unique
```

Рис. 6: Реализация линейного конгруэнтного счётчика на интервале $[0,1)$

Для того чтобы получить случайные числа в интервале от $[0,1)$ нужно поделить каждый случайный сгенерированный элемент последовательности на M . Также данная функция выводит период сгенерированной последовательности.

```
res = linear_congruent_gauge_0_1(3, 2, 1, 10, 12)
res
([0.7, 0.5, 0.1, 0.3, 0.7, 0.5, 0.1, 0.3, 0.7, 0.5, 0.1, 0.3], 4)

plt.hist(res[0])
plt.show()
```

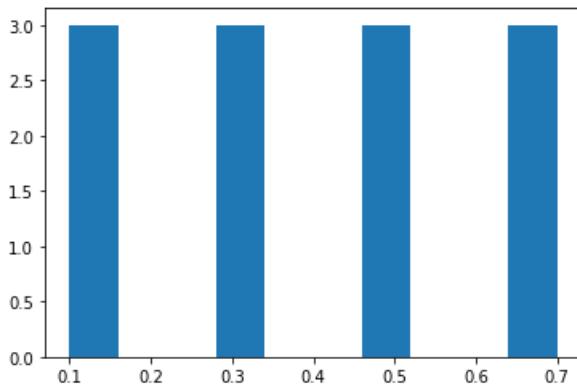


Рис. 7: Результаты генерации случайных чисел линейным конгруэнтным счётчиком на интервале $[0,1)$

В качестве аргументов были выбраны следующие значения: $r_0 = 3$, $k = 2$, $b = 1$, $M = 10$, $n = 12$, где n – это количество сгенерированных случайных чисел. Можно видеть, что при данном наборе аргументов длина периода составила 4. (Рисунок 7)

Также стоит проверить гипотезу о том, что получившиеся значения распределены равномерно. Для этого воспользуемся критерием Колмогорова-Смирнова. (Рисунок 8)

Проверка гипотезы

```
sp.stats.kstest(res[0], lambda x: sp.stats.uniform.cdf(x, loc=0, scale=1))

KstestResult(statistic=0.3000000000000004, pvalue=0.18735709092941655)
```

Рис. 8: Результаты теста Колмогорова-Смирнова

Значение $p\text{-value} > 0.05$, значит мы принимаем гипотезу о том, что данная выборка имеет равномерное распределение.

Далее был реализован алгоритм для интервала $[a;b)$. (Рисунок 9)

```
def linear_congruent_gauge_a_b(init, k, b, M, n, a_param, b_param):
    r = init
    unique = 0
    res = []
    for i in range(n):
        r = (k*r + b) % M
        val = (1-r/M)*a_param + (r/M)*b_param
        if val not in res:
            unique += 1
        res.append(val)
    return res, unique
```

Рис. 9: Реализация линейного конгруэнтного счётчика на интервале $[a,b)$

Для того чтобы получить случайные числа в интервале от $[a,b)$ нужно проделать следующее преобразование:

$$(1 - \frac{r_i}{M})/a + \frac{r_i}{M} \cdot b$$

То есть сначала мы генерируем числа в интервале от $[0,1)$, а дальше преобразуем их к интервалу от $[a,b)$.

```
res = linear_congruent_gauge_a_b(3, 2, 1, 10, 10, 100, 200)
res
([170.0, 150.0, 110.0, 130.0, 170.0, 150.0, 110.0, 130.0, 170.0, 150.0], 4)
```

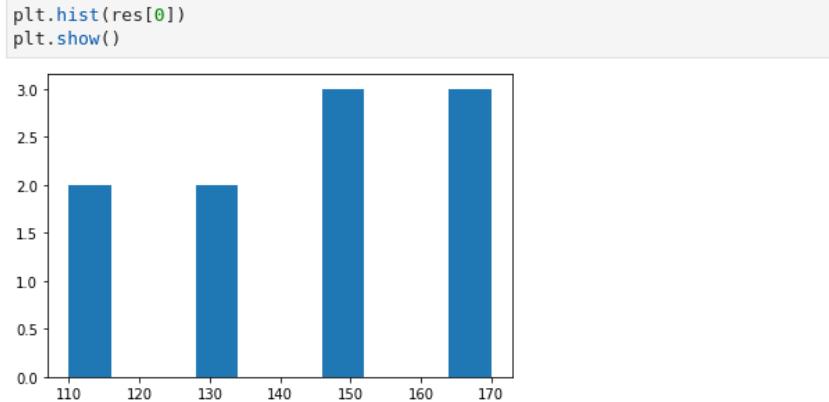


Рис. 10: Результаты генерации случайных чисел линейным конгруэнтным счётчиком на интервале $[a,b)$

В качестве аргументов были выбраны следующие значения: $r_0 = 3$, $k = 2$, $b = 1$, $M = 10$, $n = 10$, $a_{param} = 100$, $b_{param} = 200$, где n – это количество сгенерированных случайных чисел. Можно видеть, что при данном наборе аргументов длина периода составила 4. (Рисунок 10)

Также стоит проверить гипотезу о том, что получившиеся значения распределены равномерно. Для этого воспользуемся критерием Колмогорова-Смирнова. (Рисунок 11)

Проверка гипотезы

```
sp.stats.kstest(res[0], lambda x: sp.stats.uniform.cdf(x, loc=30, scale=200))  
KstestResult(statistic=0.4, pvalue=0.05898924519999926)
```

Рис. 11: Результаты теста Колмогорова-Смирнова

Значение $p-value > 0.05$, значит мы принимаем гипотезу о том, что данная выборка имеет равномерное распределение.

Датчик - экспоненциальное распределение

Задание:

Используя метод обратной функции, получить последовательность случайных чисел, распределённых экспоненциально с заданным параметром λ . Проанализировать полученную последовательность. Оценить математическое ожидание и дисперсию, построить гистограмму.

Решение:

Плотность распределения экспоненциального закона:

$$f(x) = \begin{cases} \lambda e^{-\lambda \cdot x} & x \geq 0 \\ 0 & x < 0 \end{cases}$$

Функция распределения экспоненциального закона:

$$F(x) = \begin{cases} 1 - e^{-\lambda \cdot x} & x \geq 0 \\ 0 & x < 0 \end{cases}$$

Получается, что обратная функция $F^{-1}(x)$ будет выглядеть следующим образом:

$$x = \frac{-\ln(1-y)}{\lambda} = -\frac{\ln(y)}{\lambda}$$

Если подставлять вместо y случайные равномерно распределённые значения, то можно получать требуемые числа.

Таким образом, была реализована функция на языке программирования Python. (Рисунок 12)

```
def exp_inverse_function_method(lambda_, n):
    return [-np.log(np.random.random()) / lambda_ for _ in range(n)]
```

Рис. 12: Реализация метода обратной функции для экспоненциального закона

При $\lambda = 5$ и $n = 1000$ получается следующий результат. (Рисунок 13)
Математическое ожидание экспоненциального распределения:

$$E = \frac{1}{\lambda}$$

```

x = np.linspace(sp.stats.expon.ppf(0.01), sp.stats.expon.ppf(0.99), 100)
plt.plot(0.3*x, 600*sp.stats.expon.pdf(x), c="r")
plt.hist(res)
plt.show()

```

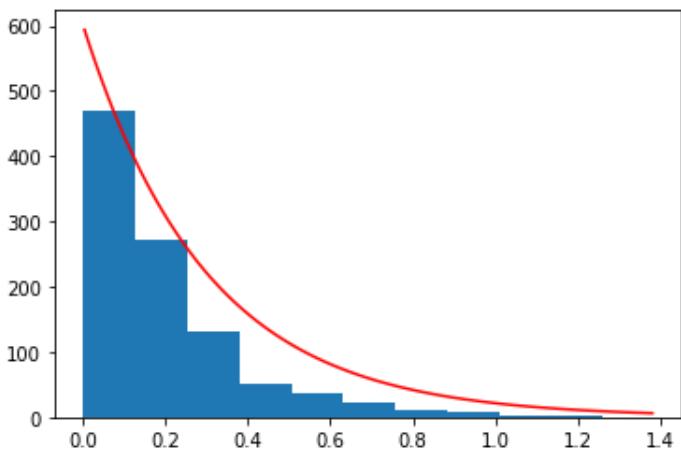


Рис. 13: Результаты генерации случайных чисел методом обратной функции для экспоненциального закона

Если рассчитывать математическое ожидание как среднее значение в выборке, то получается следующий результат. (Рисунок 14)

Математическое ожидание:

```
m = sum(res)/len(res)
m
```

0.331958674573819

```
1/lambda_
```

0.3333333333333333

Рис. 14: Теоретическое и расчётное значения математического ожидания ($\lambda = 3$)

Можно заметить, что при $n = 1000$ значения получились достаточно близкими.

Дисперсия экспоненциального распределения:

$$D = \frac{1}{\lambda^2}$$

Далее можно рассчитать дисперсию как среднее квадратное отклонение от среднего значения выборки. (Рисунок 15)

Дисперсия:

```
d = sum((x-m)**2 for x in res) / len(res)
d
```

```
0.1084274985543402
```

```
1/lambda_**2
```

```
0.1111111111111111
```

Рис. 15: Теоретическое и расчётное значения дисперсии ($\lambda = 3$)

Можно заметить, что при $n = 1000$ значения получились достаточно близкими.

Также стоит проверить гипотезу о том, что получившиеся значения распределены экспоненциально. Для этого воспользуемся критерием Колмогорова-Смирнова. (Рисунок 16)

Проверка гипотезы

```
sp.stats.kstest(res, lambda x: sp.stats.expon.cdf(x, loc=0, scale=0.2))
```

```
KstestResult(statistic=0.03048548325866085, pvalue=0.3043931949589904)
```

Рис. 16: Результаты теста Колмогорова-Смирнова

Значение $p-value > 0.05$, значит мы принимаем гипотезу о том, что данная выборка имеет экспоненциальное распределение.

Датчик - нормальный закон

Задание:

Получить нормально распределённую последовательность случайных чисел, используя:

1. преобразование Бокса-Мюллера

2. формулу $Z = \sqrt{\frac{12}{n}} \left(\sum_{i=1}^n x_i - \frac{n}{2} \right)$ (частный случай $Z = \sum_{i=1}^{12} x_i - 6$)

Решение:

1. Преобразование Бокса-Мюллера

Данное преобразование заключается в замене равномерно распределённых случайных величин на нормально распределённые случайные величины, которые можно найти по следующим формулам:

$$u_1 = \cos(2\pi \cdot v_1) \sqrt{-2 \cdot \ln(v_2)}$$
$$u_2 = \sin(2\pi \cdot v_1) \sqrt{-2 \cdot \ln(v_2)}$$

То есть на каждом шаге создания нового элемента последовательности нужно сначала сгенерировать два равномерно распределённых случайных числа v_1 и v_2 , а дальше случайно выбрать формулу u_1 или u_2 для расчёта результирующего элемента последовательности.

Данный алгоритм был реализован на языке программирования Python.
(Рисунок 17)

```
def box_muller_transform(n):
    res = []
    for _ in range(n):
        v1, v2 = np.random.uniform(size=2)
        res.append(np.random.choice([np.cos(2*np.pi*v1)*np.sqrt(-2*np.log(v2)),
                                     np.sin(2*np.pi*v1)*np.sqrt(-2*np.log(v2))]))
    return res
```

Рис. 17: Реализация преобразования Бокса-Мюллера

Была сгенерирована последовательность из 1000 элементов и на основании этого построена гистограмма. (Рисунок 18)

```
mu = 0
variance = 1
sigma = np.sqrt(variance)
x = np.linspace(mu - 3*sigma, mu + 3*sigma, 100)
plt.plot(x, 600*sp.stats.norm.pdf(x, mu, sigma), c="r")
plt.hist(res)
plt.show()
```

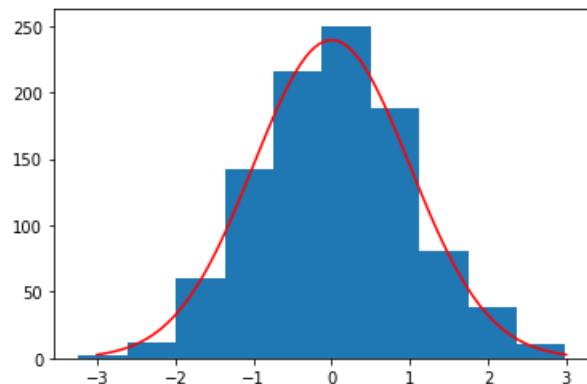


Рис. 18: Результаты применения преобразования Бокса-Мюллера

Как можно видеть на получившейся гистограмме, распределение действительно получилось нормальное, то есть алгоритм сработал корректно.

Также стоит проверить гипотезу о том, что получившиеся значения распределены нормально. Для этого воспользуемся критерием Колмогорова-Смирнова. (Рисунок 19)

Проверка гипотезы

```
sp.stats.kstest(res, lambda x: sp.stats.norm.cdf(x, loc=0, scale=1))

KstestResult(statistic=0.025903023429157512, pvalue=0.5050985031013947)
```

Рис. 19: Результаты теста Колмогорова-Смирнова

Значение $p\text{-value} > 0.05$, значит мы принимаем гипотезу о том, что данная выборка имеет нормальное распределение.

2. Применение центральной предельной теоремы

В данном алгоритме нужно изначально сгенерировать n штук случайных равномерных чисел и дальше воспользоваться формулой, которая вытекает из центральной предельной теоремы:

$$Z = \sqrt{\frac{12}{n}} \left(\sum_{i=1}^n x_i - \frac{n}{2} \right)$$

Данный алгоритм был реализован на языке программирования Python. (Рисунок 20)

```
def central_limit_theorem(n, m):
    return [np.sqrt(12/m)*np.sum(np.random.uniform(size=m))-m/2 for _ in range(n)]
```

Рис. 20: Реализация применения центральной предельной теоремы

Была сгенерирована последовательность из 1000 элементов и на основании этого построена гистограмма. (Рисунок 21)

```
mu = 0.5
variance = 1
sigma = np.sqrt(variance)
x = np.linspace(mu - 3.5*sigma, mu + 3.5*sigma, 100)
plt.plot(x, 650*sp.stats.norm.pdf(x, mu, sigma), c="r")
plt.hist(res)
plt.show()
```

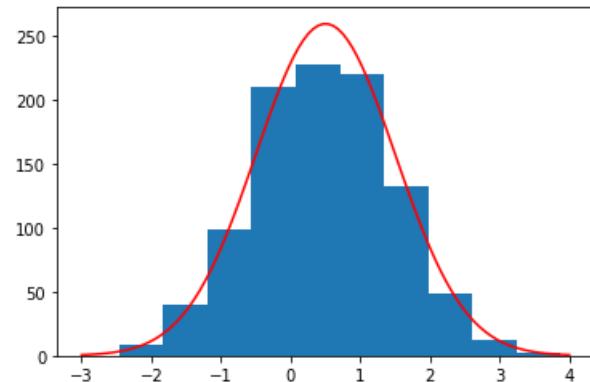


Рис. 21: Результаты применения центральной предельной теоремы

Как можно видеть на получившейся гистограмме, распределение действительно получилось нормальное, то есть алгоритм сработал корректно.

Также стоит проверить гипотезу о том, что получившиеся значения распределены нормально. Для этого воспользуемся критерием Колмогорова-Смирнова. (Рисунок 22)

Проверка гипотезы

```
sp.stats.kstest(res, lambda x: sp.stats.norm.cdf(x, loc=0.5, scale=1))

KstestResult(statistic=0.026468548604575204, pvalue=0.47720508196355993)
```

Рис. 22: Результаты теста Колмогорова-Смирнова

Значение $p\text{-value} > 0.05$, значит мы принимаем гипотезу о том, что данная выборка имеет нормальное распределение.

Датчик - треугольное распределение

Задание:

Для треугольного распределения получить функцию распределения. Используя метод обратной функции, получить последовательность случайных чисел с треугольным распределением. Оценить математическое ожидание и дисперсию, построить гистограмму.

Решение:

Пусть имеются следующие параметры: a (*min*), b (*max*), c (*мода*).

Плотность треугольного распределения:

$$f(x) = \begin{cases} \frac{2(x-a)}{(b-a)(c-a)}, & x \in [a, c] \\ \frac{2(b-x)}{(b-c)(b-a)}, & x \in [c, b] \\ 0, & x \notin [a, b] \end{cases}$$

Функция распределения треугольного закона:

$$F(x) = \begin{cases} 0, & x < a \\ \frac{(x-a)^2}{(b-a)(c-a)}, & x \in [a, c] \\ 1 - \frac{(x-b)^2}{(b-a)(b-c)}, & x \in [c, b] \\ 1, & x > b \end{cases}$$

Получается, что обратная функция $F^{-1}(x)$ будет выглядеть следующим образом:

$$F^{-1}(y) = x = \begin{cases} a + \sqrt{(b-a)(c-a)y}, & 0 < y < \frac{c-a}{b-a} \\ b - \sqrt{(b-a)(b-c)(1-y)}, & \frac{c-a}{b-a} \leq y < 1 \end{cases}$$

Если подставить вместо y случайные равномерно распределённые значения, то можно получить требуемые числа.

Таким образом, была реализована функция на языке программирования Python. (Рисунок 23)

При $a = 0$, $b = 1$, $c = 0.5$ и $n = 10$ получается следующий результат. (Рисунок 24)

Математическое ожидание треугольного распределения:

$$E = \frac{a+b+c}{3}$$

```

def triangle_inverse_function_method(a, b, c, n):
    res = []
    for _ in range(n):
        y = np.random.random()
        if (0 < y < (c-a)/(b-a)):
            res.append(a + np.sqrt((b-a)*(c-a)*y))
        else:
            res.append(b - np.sqrt((b-a)*(b-c)*(1-y)))
    return res

```

Рис. 23: Реализация метода обратной функции для треугольного закона

```

x = np.linspace(sp.stats.triang.ppf(0.01, c), sp.stats.triang.ppf(0.99, c), 100)
plt.plot(x, 100*sp.stats.triang.pdf(x, c), c="r")
plt.hist(res)
plt.show()

```

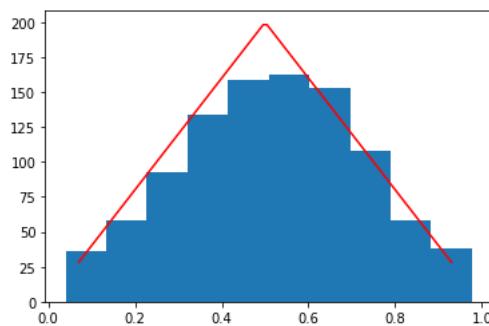


Рис. 24: Результаты генерации случайных чисел методом обратной функции для треугольного закона

Если рассчитывать математическое ожидание как среднее значение в выборке, то получается следующий результат. (Рисунок 25)

Можно заметить, что при $n = 1000$ значения получились достаточно близкими.

Дисперсия треугольного распределения:

$$D = \frac{a^2 + b^2 + c^2 - a \cdot b - a \cdot c - b \cdot c}{18}$$

Далее можно рассчитать дисперсию как среднее квадратное отклонение от среднего значения выборки. (Рисунок 26)

Можно заметить, что при $n = 1000$ значения получились достаточно близкими.

Математическое ожидание:

```
m = sum(res)/len(res)  
m
```

0.5064398956415695

```
(a+b+c)/3
```

0.5

Рис. 25: Теоретическое и расчётное значения математического ожидания ($a = 0, b = 1, c = 0.5$)

Дисперсия:

```
d = sum((x-m)**2 for x in res) / len(res)  
d
```

0.04249121135286575

```
(a**2+b**2+c**2-a*b-a*c-b*c)/18
```

0.04166666666666664

Рис. 26: Теоретическое и расчётное значения дисперсии ($a = 0, b = 1, c = 0.5$)

Также стоит проверить гипотезу о том, что получившиеся значения распределены по треугольному закону. Для этого воспользуемся критерием Колмогорова-Смирнова. (Рисунок 27)

Проверка гипотезы

```
sp.stats.kstest(res, lambda x: sp.stats.triang.cdf(x, c, loc=0, scale=1))  
KstestResult(statistic=0.05169087947748163, pvalue=0.009212252362537258)
```

Рис. 27: Результаты теста Колмогорова-Смирнова

Значение $p\text{-value} > 0.05$, значит мы принимаем гипотезу о том, что данная выборка имеет треугольное распределение.

Метод Монте-Карло

Задание:

Используя метод Монте-Карло, вычислить $\int_0^1 \sin(\pi x) dx$. Проанализируйте, как зависит точность полученного результата от количества точек, использованных для получения оценки. Постройте график этой зависимости.

Решение:

Для начала стоит нарисовать график данной функции $\sin(\pi x)$ и сгенерировать n штук случайных точек. (Рисунок 28)

```
n = 1000  
  
x = np.linspace(0,1,1000)  
y = np.sin(np.pi*x)  
  
x_rand = np.random.uniform(size=n)  
y_rand = np.random.uniform(size=n)  
  
plt.scatter(x_rand,y_rand, s=2, c="r")  
plt.plot(x, y)  
plt.show()
```

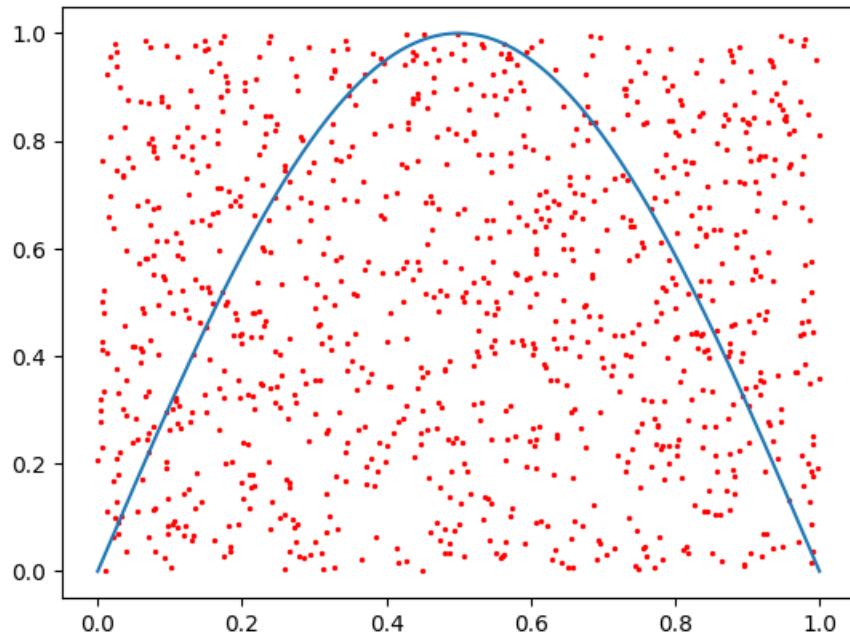


Рис. 28: Пример применения метода Монте-Карло

Далее остаётся просто посчитать то количество точек, которые попали под график и разделить их на общее количество сгенерированных точек.

Данный алгоритм был реализован на языке программирования Python.
(Рисунок 29)

```
def monte_carlo(n):
    x_rand = np.random.uniform(size=n)
    y_rand = np.random.uniform(size=n)
    count = 0
    for i in range(n):
        if y_rand[i] < np.sin(np.pi*x_rand[i]):
            count += 1
    return count/n
```

Рис. 29: Реализация метода Монте-Карло для подсчёта интеграла

Также была проанализирована зависимость между точностью результата и количеством сгенерированных точек. (Рисунок 30)

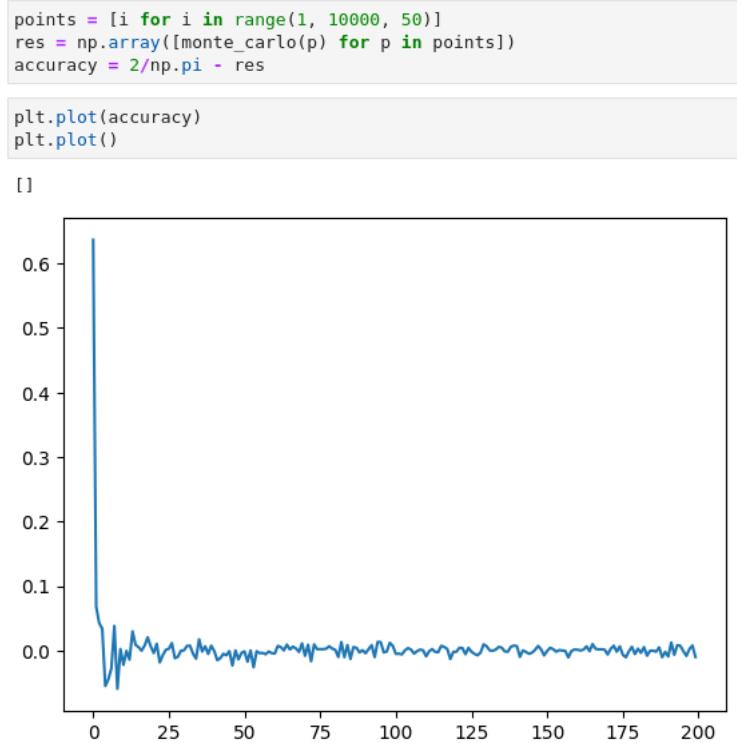


Рис. 30: Зависимость точности результатов от количества точек

Как можно видеть на графике, что чем больше количество сгенерированных точек, то тем точнее получается результат.

Дискретные СВ и случайные события

Задание:

Используя метод статистических испытаний, оценить вероятность выпадения герба при бросании правильной монеты. Проанализируйте, как зависит точность полученного результата от количества точек, использованных для получения оценки. Постройте график этой зависимости.

Решение:

Для решения данной задачи стоит сгенерировать n случайных равномерно распределённых чисел, если $x_i < 0.5$, то добавлять в результирующий список 0, в противном случае 1. Далее стоит просуммировать элементы получившегося списка и разделить на количество элементов в этом списке.

Данный алгоритм был реализован на языке программирования Python. (Рисунок 31)

```
def coin(n):
    xs = np.random.uniform(size=n)
    return sum([0 if x < 0.5 else 1 for x in xs])/n
```

Рис. 31: Реализация метода статистических испытаний

Также была проанализирована зависимость между точностью результата и количеством сгенерированных точек. (Рисунок 32)

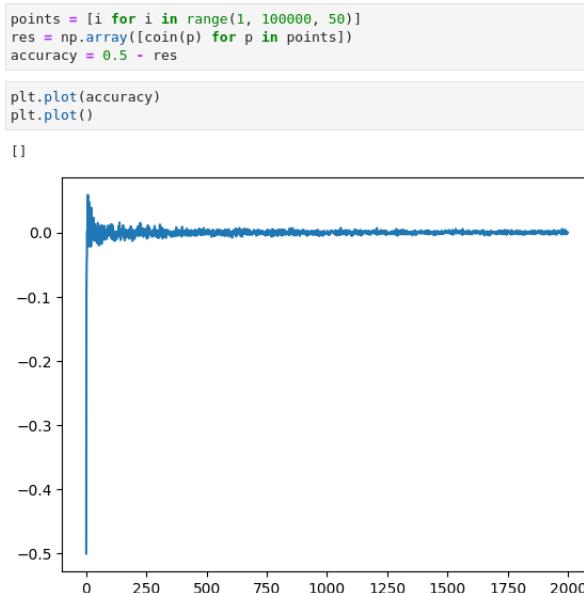


Рис. 32: Зависимость точности результатов от количества точек

Как можно видеть на графике, что чем больше количество сгенерированных точек, то тем точнее получается результат.

Проверка качества датчиков

По ходу выполнения заданий данного раздела, в конце каждого из датчиков, были сделаны определённые выводы о проверках гипотез распределений данных датчиков.

Моделирование СМО

Модель СМО в AnyLogic

Задание:

Создать и проанализировать модель системы массового обслуживания на примере банка в AnyLogic.

Решение:

Создадим простейшую модель обслуживания клиентов банка. Обслуживающими блоками будут банкомат и кассиры. Для создания модели будем использовать Библиотеку моделирования процессов. Каждый блок будет задавать определённый элемент СМО.

Базовая схема СМО представлена на рисунке 33. Данная схема моделирует простейшую СМО, состоящую из источника заявок, очереди, обслуживающего блока и блока финального уничтожения агентов.

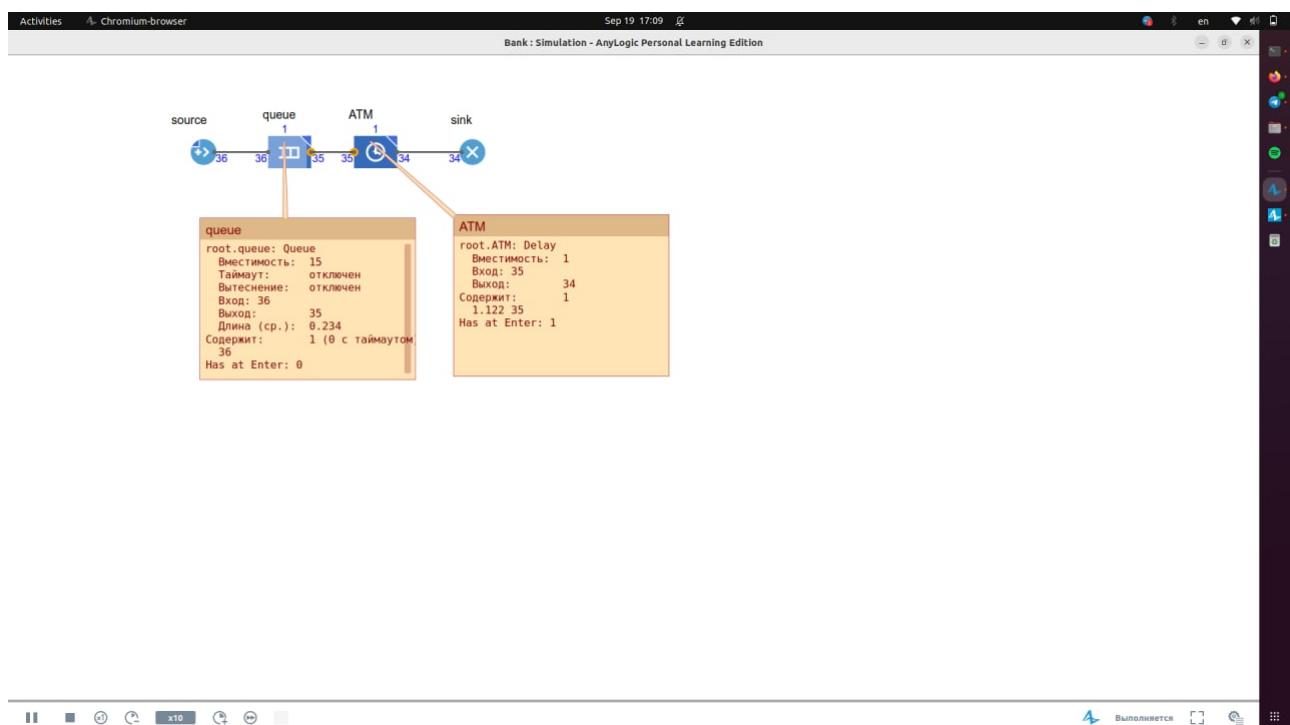


Рис. 33: Схема модели простейшей СМО

В модели задействованы следующие блоки:

1. *Source* генерирует агентов определенного типа. Обычно он используется в качестве начальной точки диаграммы процесса, формализующей поток агентов. В нашем примере агентами будут посетители

банка, а объект *Source* будет моделировать их приход в банковское отделение.

2. *Queue* моделирует очередь агентов, ожидающих приема объектами, следующими за данным в диаграмме процесса. В нашем случае он будет моделировать очередь клиентов, ждущих освобождения банкомата.
3. Объект *Delay* задерживает агентов на заданный период времени, представляя в модели банкомат, у которого посетитель банковского отделения тратит некоторое время для проведения необходимой ему операции.
4. Объект *Sink* утилизирует заявки, обработанные системой. Обычно он используется в качестве конечной точки потока агентов (и диаграммы процесса соответственно).

Далее в модель была добавлена 2D и 3D анимация того, как агенты (люди) заходят в модель и ожидают в очереди к банкомату. Получившаяся модель представлена на рисунке 34.

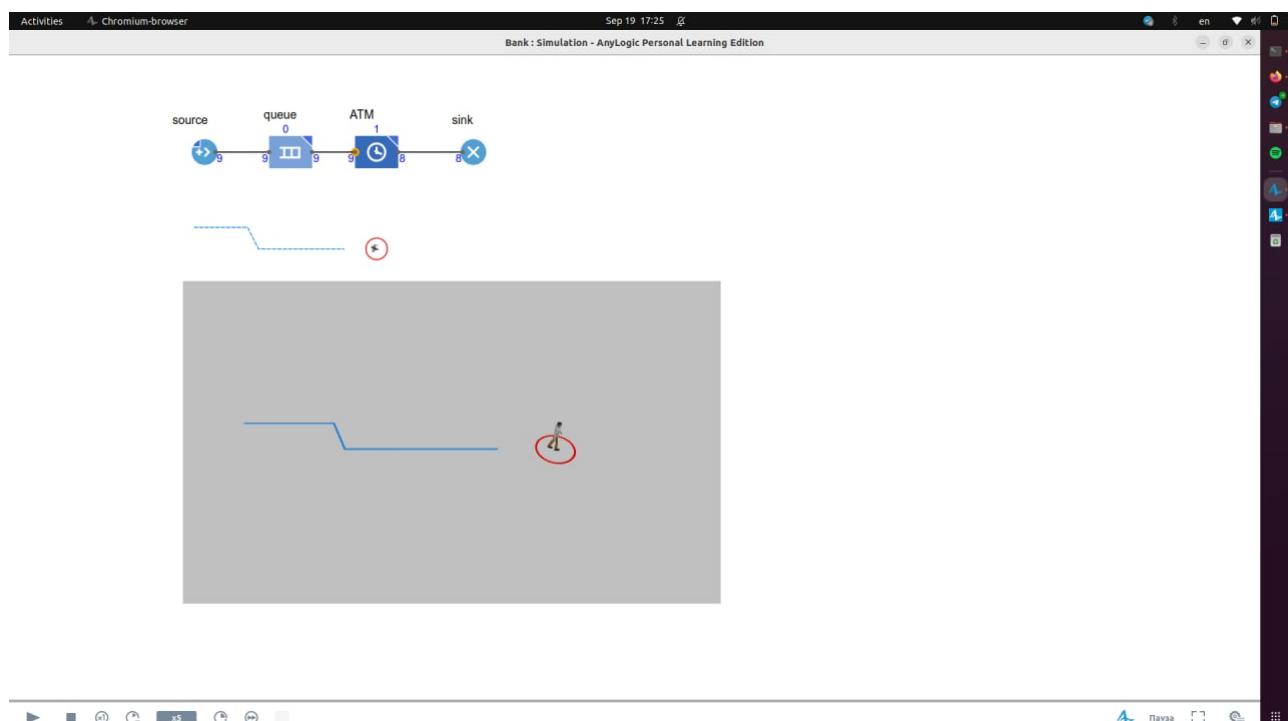


Рис. 34: Анимация модели в среде AnyLogic

Далее модель была усложнена. Были добавлены служащие – банковские кассиры. Можно промоделировать кассиров, как и банкомат, с помощью объекта *Delay*. Но удобнее моделировать кассиров с помощью ресурсов. Объект *Service* захватывает для агента заданное количество ресурсов, задерживает агента, а затем освобождает захваченные им ресурсы.

Следующим этапом был добавлен выбор клиентов, то есть агенты могут пойти либо к банкомату, либо к банковским кассирам. Данная логика была реализована с помощью блока *SelectOutput*, он является блоком принятия решения. В зависимости от заданного условия, агент, поступивший в объект, будет отправляться на один из двух выходных портов.

Также был добавлен блок *ResourcePool*, который необходим для хранения ресурсов модели. В данном случае в качестве ресурсов будут выступать кассиры.

Также в блоке *Source* установим интенсивность прибытия клиентов 0.3 в минуту, т.е. каждые 10 минут будет приходить по 3 человека. В свойствах блока *Queue* установим вместимость 15, то есть в очереди может находиться не более 15 человек. В свойствах блока *Delay* время задержки зададим как *triangular(0.8, 1.5, 3.5)*, т.е. в виде треугольного распределения с минимальным временем обслуживания 0.8 секунд, среднем временем обслуживания – 1.5 секунды и максимальным – 3.5 секунд. В блоке *Service* зададим вместимость очереди 20 человек, а время задержки как *triangular(2.5, 6, 11)*. Количество ресурсов, т.е. количество кассиров, установим равным 4.

Теперь модель будет выглядеть следующим образом, как показана на рисунке 35.

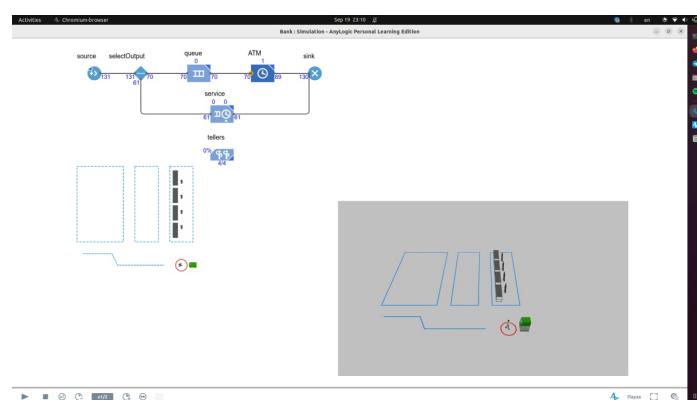


Рис. 35: Усложнение модели СМО в среде AnyLogic

Также была добавлена анимация для новой части модели.

Добавим в модель сбор статистики. С помощью столбиковой диаграммы из палитры Статистика отобразим среднее время занятости банкомата и среднюю длину очереди:

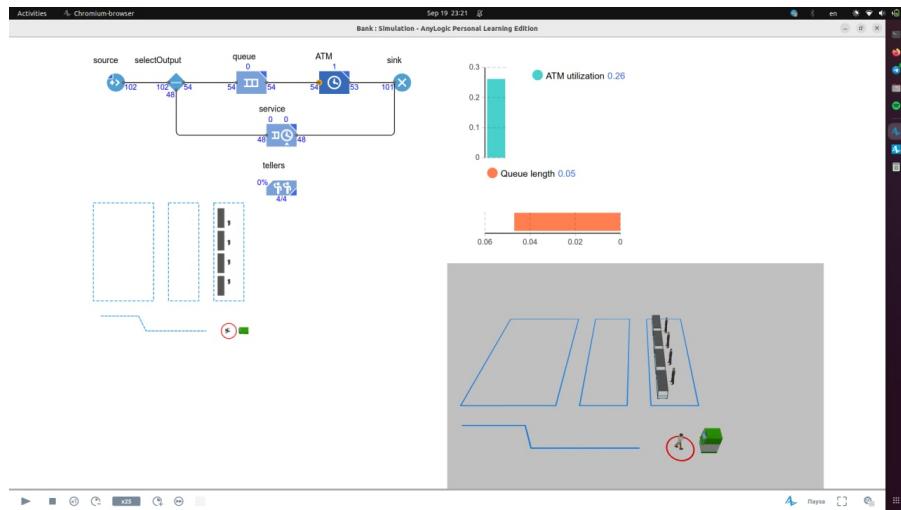


Рис. 36: Добавление статистики среднего времени занятости банкомата и средней длины очереди

С помощью гистограммы из палитры Статистика отобразим время ожидания и время обслуживания в системе.

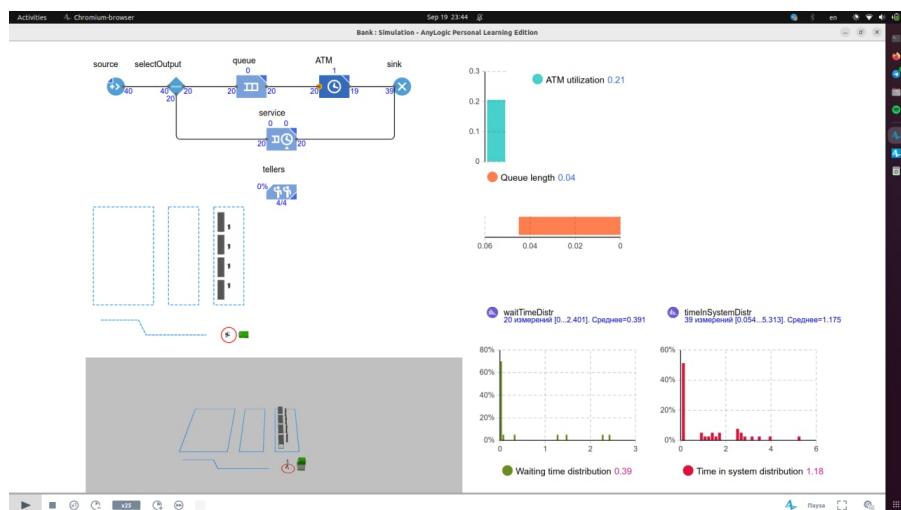


Рис. 37: Добавление статистики времени ожидания и времени обслуживания в системе

На гистограммах видно, что система работает хорошо, очереди маленькие, клиентов обслуживают быстро.

Одноканальная СМО

Задание:

Реализовать одноканальную СМО на языке программирования Python.

Решение:

СМО с бесконечной очередью – это СМО, в которой всегда есть места в очереди и если требование приходит, в момент, когда обслуживающее устройство занято, то оно не получает немедленного отказа, а может стать в очередь и ожидать освобождения обслуживающего устройства.

На вход одноканальной СМО с бесконечной очередью поступает пуассоновский поток требований с интенсивностью λ .

Интенсивность пуассоновского потока обслуживания – μ .

Дисциплина очереди естественная: кто раньше пришёл, тот раньше и обслуживается.

Число мест в очереди не ограничено.

Реализация модели

Данная модель была реализована на языке программирования Python. Для начала был создан класс *QueuingSystemWithInfinityQueue*, в котором реализованы все необходимые методы.

Данная функция принимает на вход параметр λ , параметр μ и время моделирования системы в условных единицах. (Рисунок 38)

```
class QueuingSystemWithInfinityQueue:
    def __init__(self, *, lambda_: float, mu: float, simulation_time: int):
        self.lambda_ = lambda_
        # интенсивность потока требований
        self.mu = mu
        # интенсивность обслуживания требований
        self.simulation_time = simulation_time
        # время моделирования системы

        self.requirements = [] # время поступления нового требования
        self.execute_services = [] # время обслуживания конкретного требования
        self.end_services = [] # время конца обслуживания конкретного требования
        self.queue = {} # очередь на момент подачи i-го требования
```

Рис. 38: Параметры модели

Был реализован метод *generate_requirements*, который генерирует поток требований в соответствии с пуассоновским законом распределения. (Рисунок 39)

```
def generate_requirements(self) -> List[float]:
    last_requirements_time = 0
    while last_requirements_time < self.simulation_time:
        arrival_time = np.random.exponential(self.lambda_)
        last_requirements_time += arrival_time
        self.requirements.append(last_requirements_time)
    return self.requirements
```

Рис. 39: Реализация метода генерации требований

Был реализован метод *get_service_times*, который в соответствии с пуассоновским законом задаёт каждому требованию его время обработки. (Рисунок 40)

```
def get_service_times(self) -> List[float]:
    for _ in range(len(self.requirements)):
        service_time = np.random.exponential(self.mu)
        self.execute_services.append(service_time)
    return self.execute_services
```

Рис. 40: Реализация метода задания обработки требований

Был реализован метод *get_service_end*, который считает сколько времени провело требование в системе. (Рисунок 41)

```
def get_service_end(self) -> List[float]:
    self.end_services.append(self.requirements[0] + self.execute_services[0])
    for requirement in range(len(self.requirements) - 1):
        value = self.requirements[requirement + 1] + self.execute_services[requirement + 1]
        if self.requirements[requirement + 1] < self.end_services[requirement]:
            value += self.end_services[requirement] - self.requirements[requirement + 1]
        self.end_services.append(value)
    return self.end_services
```

Рис. 41: Реализация метода подсчёта времени требования в системе

Был реализован метод `get_queue`, который для каждого требования сопоставляет количество требований, который находится перед ним в очереди. (Рисунок 42)

```
def get_queue(self) -> Dict[int, int]:
    self.queue = {}
    for requirement in range(1, len(self.requirements)):
        self.queue[requirement + 1] = (self.requirements[requirement] < np.array(self.end_services)[:requirement]).sum()
    return self.queue
```

Рис. 42: Реализация метода нахождения количества требований в очереди перед текущим требованием

Был реализован метод `get_features`, который рассчитывает основные характеристики модели. (Рисунок 43)

```
def get_features(self):
    number_of_requirements = len(self.requirements)          # количество требований
    average_service_time = np.mean(self.execute_services)    # среднее время обслуживания
    average_time_spent_in_system = (np.array(self.end_services) - np.array(self.requirements)).mean()      # среднее время пребывания в системе
    middle_length_queue = np.mean(list(self.queue.values()))   # средняя длина очереди
    requirements_per_unit_of_time = number_of_requirements / self.end_services[-1]                         # количество требований в единицу времени
    arriving_per_unit_of_time = number_of_requirements / self.requirements[-1]                          # количество требований поступающих в единицу времени

    return {"Number of requirements": number_of_requirements,
            "Average service time": average_service_time,
            "Average time spent in system": average_time_spent_in_system,
            "Middle length queue": middle_length_queue,
            "Requirements per unit of time": requirements_per_unit_of_time,
            "Arriving requirements per unit of time": arriving_per_unit_of_time}
```

Рис. 43: Реализация метода нахождения характеристик модели

Характеристики модели:

1. количество требований;
2. среднее время обслуживания;
3. среднее время пребывания в системе требования;
4. средняя длина очереди;
5. количество требований обслуженных в единицу времени;
6. количество требований поступающих в единицу времени;

Если задать параметры модели: $\lambda = 1$, $\mu = 2$, $simulation_time = 100$, то получаются следующие результаты. (Рисунок 44)

```
lambda_ = 1
mu = 2
simulation_time = 100

qs = QueuingSystemWithInfinityQueue(lambda_=lambda_,
                                      mu=mu,
                                      simulation_time=simulation_time)

requirements = qs.generate_requirements()
service_time = qs.get_service_times()
service_end = qs.get_service_end()
queue = qs.get_queue()

qs.get_features()

{'Number of requirements': 116,
 'Average service time': 2.2798619976439727,
 'Average time spent in system': 84.5850848144408,
 'Middle length queue': 37.904347826086955,
 'Requirements per unit of time': 0.43519544141659766,
 'Arriving requirements per unit of time': 1.1598994853157558}
```

Рис. 44: Результаты модели при $\lambda = 1$, $\mu = 2$, $simulation_time = 100$

Hold

Задание:

Реализовать модель СМО с применением блока Hold.

Решение:

При моделировании СМО можно столкнуться с ситуацией, когда количество агентов, поступающих в очередь превышает вместимость этой очереди. Для того, чтобы не потерять этих агентов, не создавая дополнительный блок *Delay*, можно осуществлять задержку агентов с помощью блока *Hold*.

Блок *Hold* блокирует/снимает блокировку с потока агентов на определенном участке блок-схемы. Он используется, например, когда блок может принимать агентов, но вы не хотите (временно) продолжать их обработку или когда нужно заблокировать поступление агентов только от какого-то определенного блока, в то же время принимая агентов, приходящих с выходных портов других блоков.

Рассмотрим СМО, состоящую из источника, очереди, блока задержки и стока, в которую добавим блок Hold. (Рисунок 45)

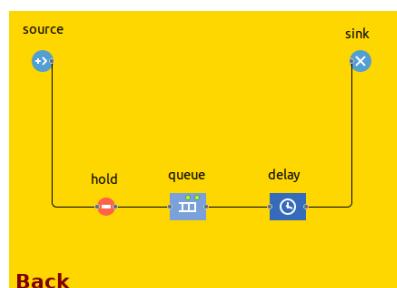


Рис. 45: Схема СМО с блоком *Hold*

Логика блокировки и снятия блокировки потока агентов реализована в блоке *Queue*. (Рисунок 46)

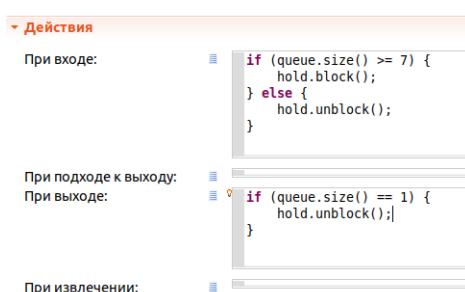


Рис. 46: Логика работы с блоком *Hold*

При входе проверяется достигнута ли максимально возможная вместимость, в данном случае 7, если да, то осуществляется блокировка. В случае, когда размер очереди становится равен 0, то блокировка с агентов снимается.

Таким образом, была рассмотрена СМО, в которой осуществляется блокировка поступления агентов в очередь.

СМО с отказами

Задание:

На многоканальный телефон (количество каналов N) звонят пользователи с заданной интенсивностью.

Если все каналы заняты, клиент получает отказ в обслуживании.

Если соединение установлено и есть свободный оператор, принимающий звонки, он общается с клиентом и оформляет заявку.

Если все операторы заняты, звонок пользователя отправляется в очередь (FIFO) и ожидает, пока не освободится один из операторов. Если при этом время ожидания превышает заданное значение (T), пользователь прекращает ожидание и кладет трубку.

Время общения с оператором случайно, распределено по треугольному закону (параметры задайте самостоятельно).

Количество операторов, отвечающих на звонки, определяется расписанием и зависит от дня недели и времени суток.

Создайте имитационную модель центра обработки звонков.

Определите процентную долю следующих групп звонков:

1. Упущеные
2. Прекратившие ожидание
3. Обслуженные

Проанализируйте распределение времени ожидания обслуженных звонков.

Проанализируйте, как интенсивность звонков, количество каналов и количество операторов влияют на качество обслуживания.

Решение:

Для моделирования данной СМО была построена модель в AnyLogic. (Рисунок 47)

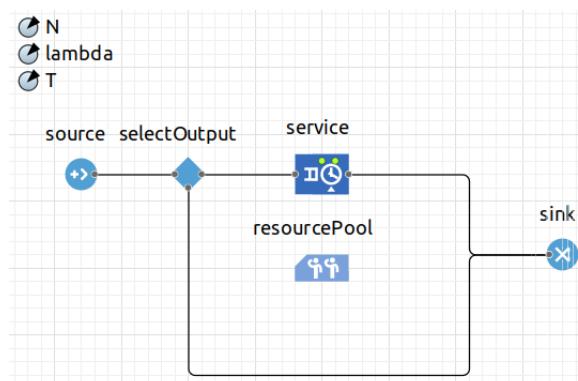


Рис. 47: Схема СМО с отказами

В качестве интенсивности потока была взята $\lambda = 10$, крайнее время ожидания $T = 5$ и количество каналов $N = 10$.

В данной схеме имеется блок *Source*, в котором с экспоненциальным законом распределения создаются агенты, *SelectOutput*, который отправляет агентов в *Sink*, если количество агентов в блоке *Service* больше 10. Также имеется блок *ResourcePool*, который хранит в себе каналы обслуживания, которые в данной модели выступают как ресурсы.

Также в данной модели предусмотрен сбор статистики. (Рисунок 48)

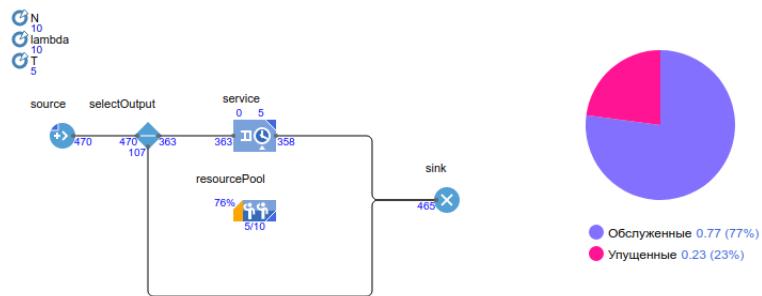


Рис. 48: Сбор статистики

Можно видеть, что при данных параметрах обслуженными остались 77% агентов, оставшиеся остались не обслуженными.

Если же варьировать параметры модели, а именно интенсивность входного потока, то можно будет видеть, что с увеличением интенсивности потока – качество обслуживания понижается, при уменьшении – наоборот. (Рисунок 49)

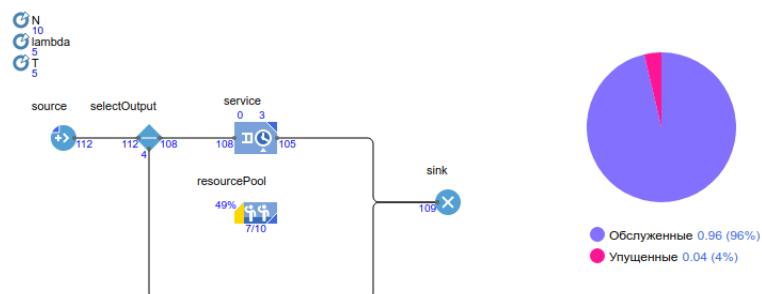


Рис. 49: Уменьшение интенсивности звонков

Если же варьировать количество каналов обслуживания, то с увеличением числа каналов – качество обслуживания повышается, а при уменьшении – наоборот. (Рисунок 50)

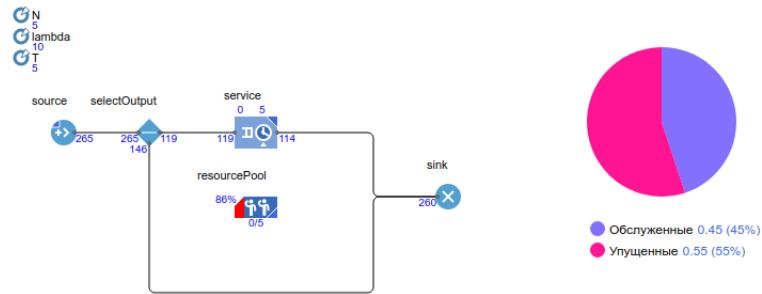


Рис. 50: Уменьшение количества каналов обслуживания

Таким образом, была реализована модель центра обработки звонков, было проанализировано влияние факторов на качество её обслуживания.

Проект СМО

Задание:

Придумать проект СМО и реализовать его в среде AnyLogic.

Решение:

Пусть имеется автомойка, которая работает ежедневно. У автомойки есть несколько 10 боксов: 5 боксов осуществляют помывку автомобилей, а другие 5 боксов осуществляют незначительный ремонт автомобилей (замена шин, ремонт фар, ...). Заявки – автомобили, которые поступают в соответствии с расписанием. Если все боксы заняты, то автомобиль может подождать некоторое время, а может сразу выйти. После обслуживания автомобиль покидает систему.

Автомобили могут выйти по времени ожидания в очереди или по количеству заявок, которые накопились в очереди.

В соответствии с описанием выше была построена модель в среде AnyLogic. (Рисунок 51)

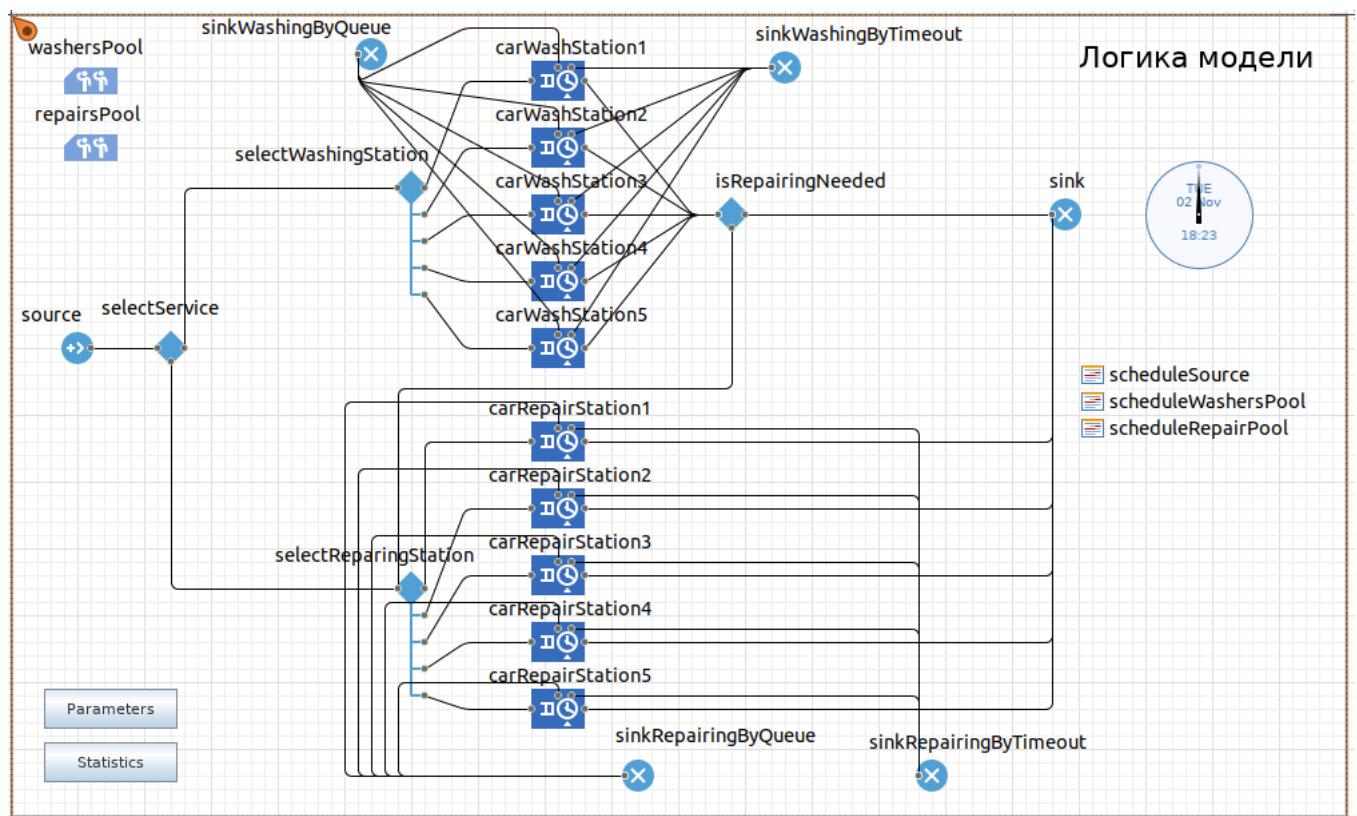


Рис. 51: Схема СМО станции технического обслуживания машин

В *Source* заявки поступают в соответствии с расписанием, далее агент с вероятностью 65% направляется на помывку машины, а может направиться на починку машины. Если агент направился на помывку, то он выбирает наименее загруженный бокс из всех и ждёт своей очереди обслуживания.

Предельное время ожидания клиента на помывку машины составляет 120 минут, а предельное количество агентов в очереди для данной услуги составляет 10 человек. Время обслуживания одного клиента было взято с помощью треугольного распределения с параметрами: минимальное время – 20 минут, мода – 30 минут, максимальное время – 50 минут.

После того как клиент помыл свою машину, то с вероятностью 25% он направляется на починку машины. Это стандартная практика для станций технического обслуживания, что перед починкой стоит помыть машину. Если агент направился на починку машины, то он выбирает наименее загруженный бокс из всех и ждёт своей очереди обслуживания.

Предельное время ожидания клиента на починку машины составляет 240 минут, а предельное количество агентов в очереди для данной услуги составляет 10 человек. Время обслуживания одного клиента было взято с помощью треугольного распределения с параметрами: минимальное время – 30 минут, мода – 40 минут, максимальное время – 120 минут.

Если агент полностью обслужен, то он покидает систему через *Sink*.

Из-за того, что на практике агенты поступают с некоторой периодичностью, то было задано расписание для агентов и расписание для сотрудников, которые осуществляют мойку и починку машин. (Рисунок 52)

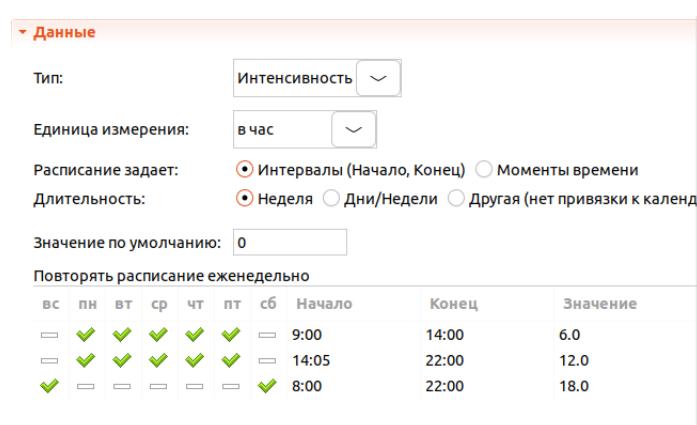


Рис. 52: Расписание поступление агентов в систему

Можно видеть, что в будние дни количество заявок меньше, а в выходные – больше, что соответствует реальности. В качестве ресурсов в данной модели выступают сотрудники, которые осуществляют мойку и починку автомобилей. Они тоже задавались в соответствии с расписанием.

Также в данной модели собиралась статистика о упущеных и обслуженных клиентах по каждому виду сервисов, статистика о загруженности сотрудников по каждому из сервисов. (Рисунок 53)

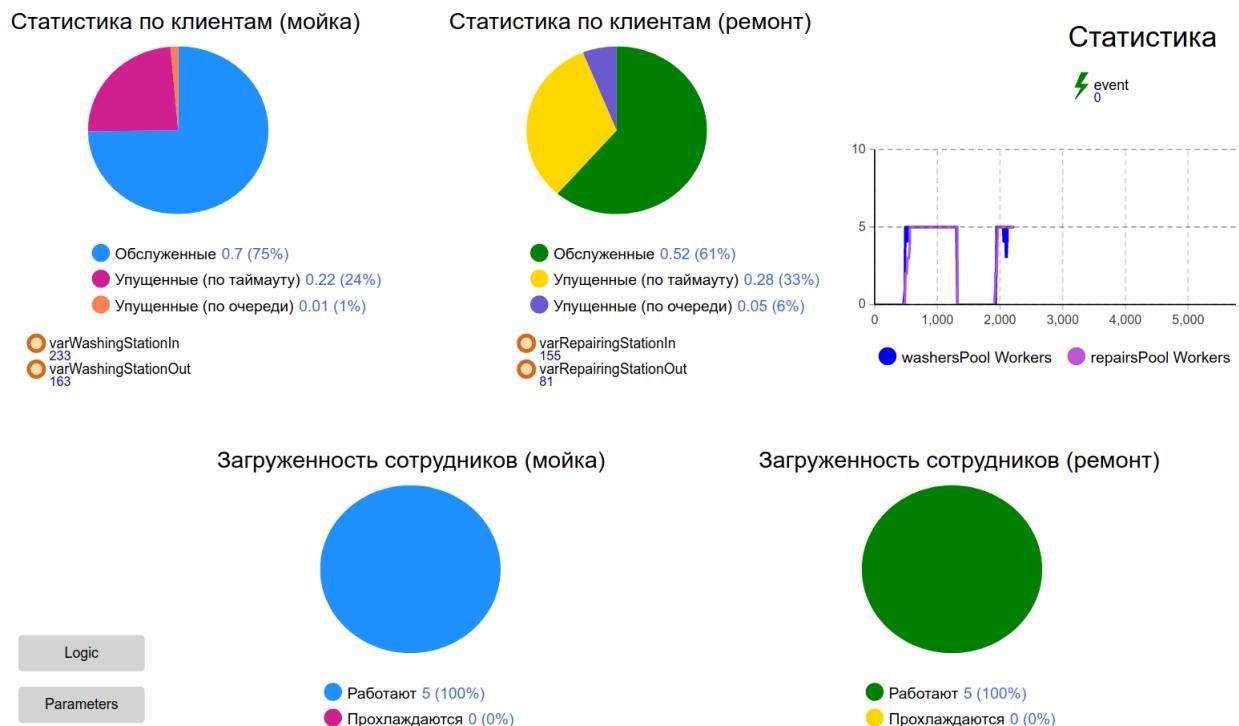


Рис. 53: Статистика по модели

Можно видеть, что при данных параметрах большинство клиентов, примерно 70%, обслуживаются, а работники полностью загружены. Для уменьшения числа упущеных клиентов стоит сделать больше боксов и соответственно нанять больше сотрудников для каждого из сервисов.

Ещё одним фактором влияющим на число упущеных клиентов является предельное время ожидания клиентов, если его увеличить, то соответственно должно уменьшиться число упущеных клиентов. Аналогичные действия можно предпринять и для увеличения величины очереди.

Таким образом, в данной работе удалось промоделировать работу станции технического обслуживания и проанализировать показатели модели.

Приёмы работы с элементами презентации

Движение между двумя ограниченными линиями

Задание:

Реализовать в среде AnyLogic движение между двумя ограниченными линиями.

Решение:

Построим два параллельных отрезка и квадрат между ними. (Рисунок 54)

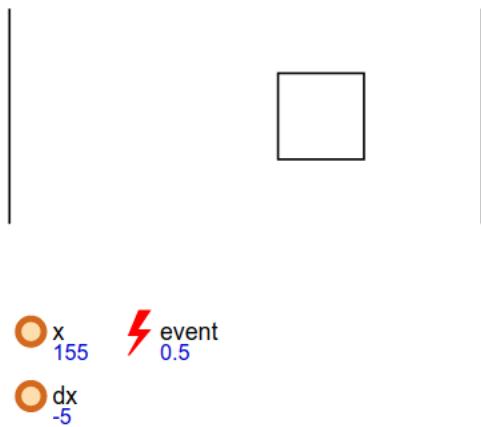


Рис. 54: Схема модели в AnyLogic

Для решения данной задачи нужно завести переменную x , которая будет отвечать за текущие координаты квадрата. Также потребуется переменная dx , которая отвечает за направление движения квадрата. В событии стоит прописать следующий код. (Рисунок 55)

```
▼ Действие
if (x + dx + rectangle.getWidth() > line1.getX() || x + dx < line2.getX()) {
    dx *= -1;
}
x += dx;
rectangle.setX(x);
```

Рис. 55: Алгоритм события движения прямоугольника

Получается, что если мы дошли до границы второй линии или наоборот дошли до границы первой линии, то квадрат меняет своё направление движения. Также он стабильно наращивает свою позицию в соответствии с направлением и обновляет свои координаты.

Движение по окружности и спирали

Задание:

Реализовать в среде AnyLogic движение по спирали Архимеда и прямоугольной спирали.

Решение:

Для решения данной задачи будет рассмотрено движение по полярным координатам. Будем на каждом шаге изменять координаты x , y и i .

Для визуализации зададим овал и будем изменять его координаты в соответствии с формулами. (Рисунок 56)



Рис. 56: Схема модели в AnyLogic

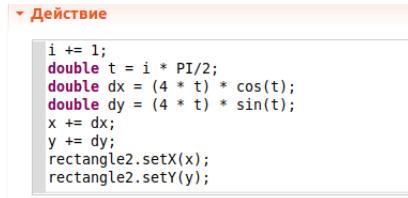
В действии события пропишем, что на каждом шаге у нас изменяется угол. В соответствии с этим мы изменяем приращение по оси x и по оси y . Далее добавляем получившиеся приращения к текущим координатам и обновляем координаты овала. (Рисунок 57)

The screenshot shows the 'Действие' (Action) tab in AnyLogic. The code is as follows:

```
i += 1;
double t = i / 10. * PI;
double dx = (1 + 3 * t) * cos(t);
double dy = (1 + 3 * t) * sin(t);
x += dx;
y += dy;
oval.setX(x);
oval.setY(y);
```

Рис. 57: Алгоритм события движения по спирали Архимеда

Алгоритм же для движения по прямоугольной спирали заключается в том, что угол отклонения должен быть кратен 90 градусам. (Рисунок 58)



```
▼ Действие
i += 1;
double t = i * PI/2;
double dx = (4 * t) * cos(t);
double dy = (4 * t) * sin(t);
x += dx;
y += dy;
rectangle2.setX(x);
rectangle2.setY(y);
```

Рис. 58: Алгоритм события движения по прямоугольной спирали

Подводя итог, нам удалось воспроизвести необходимые типы движения и познакомиться с приёмами работы с элементами презентации.

Движение по периметру

Задание:

Реализовать в среде AnyLogic движение по периметру.

Решение:

Построим квадрат в качестве поля и красный квадрат в качестве объекта, который будет двигаться по внешнему квадрату. (Рисунок 59)

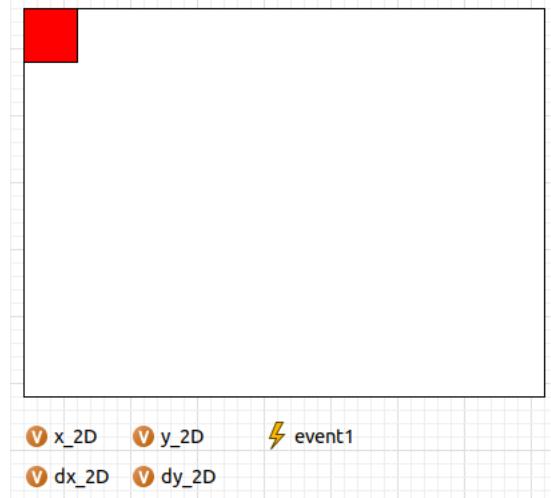


Рис. 59: Схема модели в AnyLogic

Для решения данной задачи нужно завести переменную x , которая будет отвечать за текущие координаты квадрата по оси X и переменную y , которая будет отвечать за текущие координаты квадрата по оси Y . Также потребуются переменные dx и dy , которые отвечают за направление движения квадрата. В событии стоит прописать следующий код. (Рисунок 60) (Рисунок 60)

```
▼ Действие
if (x_2D + rectangle2.getWidth() < rectangle1.getWidth() + rectangle1.getX() &&
    y_2D <= rectangle1.getY()) {
    x_2D += dx_2D;
    y_2D = rectangle1.getY();
} else if (x_2D + rectangle2.getWidth() >= rectangle1.getWidth() + rectangle1.getX() &&
          y_2D + rectangle2.getHeight() < rectangle1.getHeight() + rectangle1.getY()) {
    x_2D = rectangle1.getWidth() + rectangle1.getX() - rectangle2.getWidth();
    y_2D += dy_2D;
} else if (x_2D > rectangle1.getX() &&
          y_2D + rectangle2.getHeight() <= rectangle1.getHeight() + rectangle1.getY()) {
    x_2D -= dx_2D;
    y_2D = rectangle1.getHeight() + rectangle1.getY() - rectangle2.getHeight();
} else if (x_2D >= rectangle1.getX() &&
          y_2D > rectangle1.getY()) {
    x_2D = rectangle1.getX();
    y_2D -= dy_2D;
}

rectangle2.setX(x_2D);
rectangle2.setY(y_2D);
```

Рис. 60: Алгоритм события движения прямоугольника по периметру

Получается, что если мы дошли до какой-то из границ внешнего квадрата, то мы просто меняем направление движения и обновляем координаты красного квадрата.

Подводя итог, нам удалось воспроизвести необходимые типы движения и познакомится с примерами работы с элементами презентации.

Реплицированные объекты. Заполнение областей

Задание:

Научиться работать с объектами презентации и элементами управления.

Решение:

В среде AnyLogic существуют такие элементы презентации как области просмотра, они предназначены для упрощения демонстрации объектов, находящихся на достаточно больших расстояниях друг от друга. Для осуществления перехода между областями просмотра можно использовать как блоки элементов управления, такие как кнопки и обычные элементы презентации, например текст. (Рисунок 59)

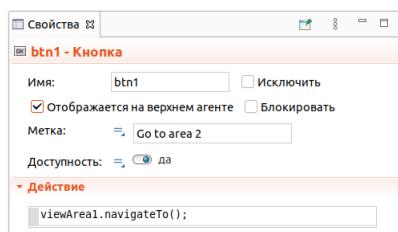


Рис. 61: Переход между областями просмотра

Иногда перед пользователем стоит задача создания некоторого количества равноудаленных друг от друга одинаковых элементов, в этом случае он может воспользоваться такими свойствами элементов презентации как указание количества элементов и их координат, зависящих от индекса. (Рисунок 62)

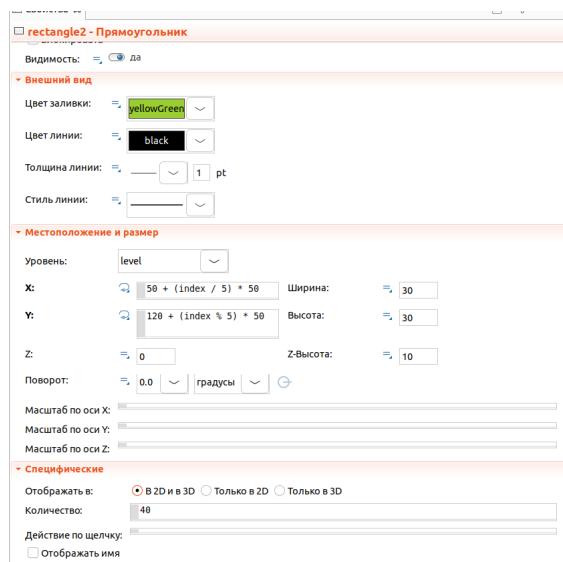


Рис. 62: Задание количества и местоположения элементов

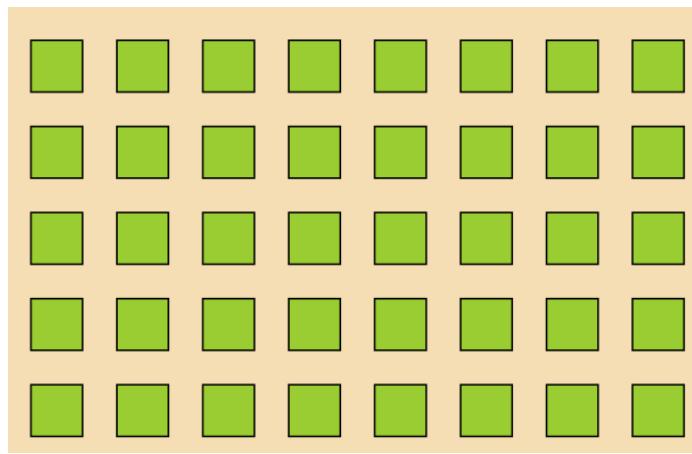


Рис. 63: Вид заданных элементов при демонстрации

Положение реплицированных элементов можно так же менять с течением времени. (Рисунок 64)

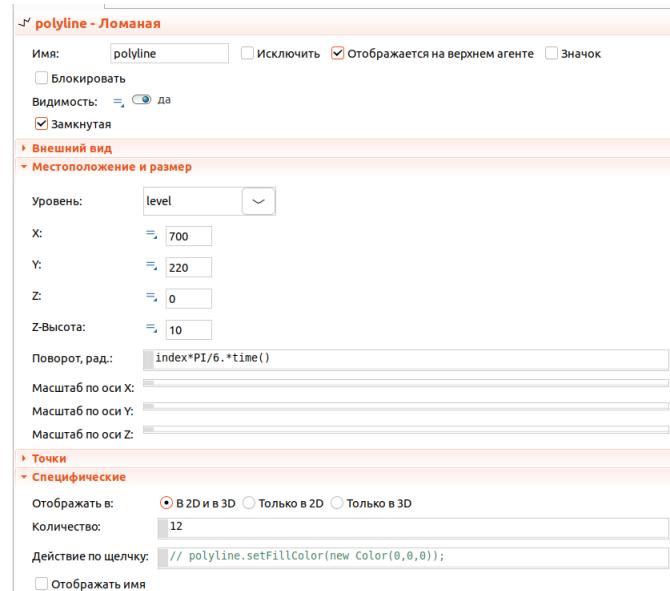


Рис. 64: Изменение положения элементов в зависимости от времени

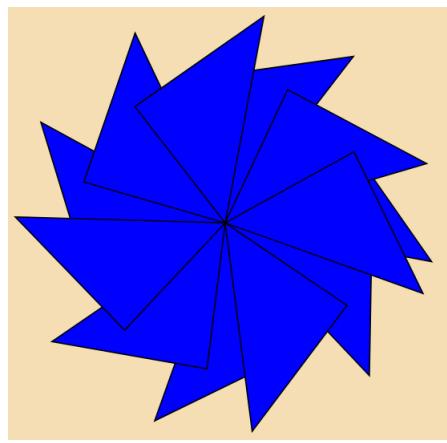


Рис. 65: Вид заданных элементов при демонстрации

Таким образом, нами были изучены основные механики работы с объектами презентации и элементами управления.

Программный доступ к элементам презентации

Задание:

По щелчку мыши строить в области просмотра фигуры различной формы. Тип фигуры (КВАДРАТ, КРУГ, ОТРЕЗОК) определять с помощью радиокнопок, размер – с помощью бегунка.

При нажатии кнопки 1 сформировать коллекцию кругов. Установить с помощью бегунка одинаковый радиус всех кругов.

По нажатию кнопки 2 разместить их в ряд (строку) в нижней части области просмотра на равном расстоянии друг от друга.

Решение:

В начале создадим радио переключатель, в котором укажем все возможные типы объектов. Далее создадим бегунок, который будет определять размер фигуры.

Для того, чтобы фигуры создавались по щелчку мыши, необходимо создать прямоугольник размером с область просмотра, в котором следует прописать какого типа и размера фигуру необходимо создать в месте клика мышки. (Рисунок 66)

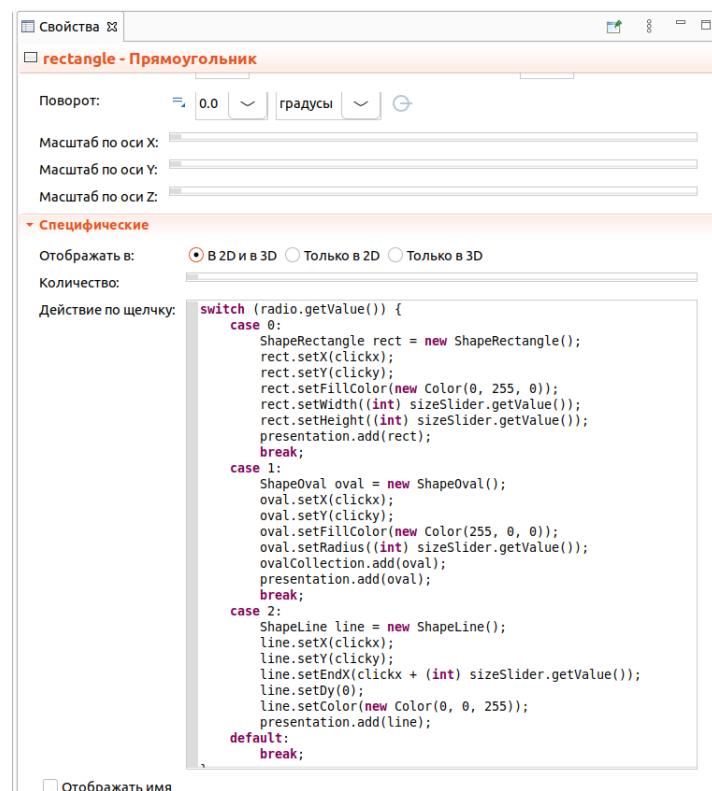


Рис. 66: Алгоритм создания фигуры

При создании объекта типа «круг», объект добавляется в коллекцию, в которой будут храниться все объекты этого типа. Теперь, когда все круги собраны в одной коллекции, необходимо создать кнопку, которая будет задавать кругам одинаковый размер. (Рисунок 67)

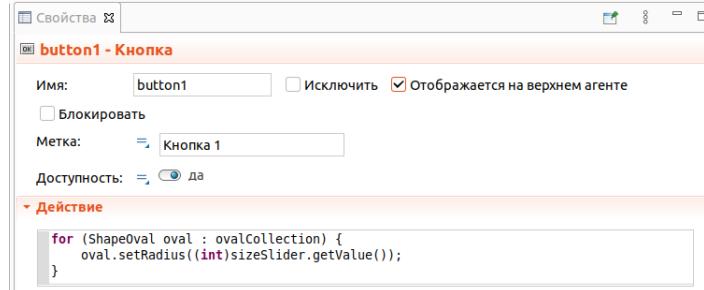


Рис. 67: Задание одинаковых размеров

Еще одним заданием было создание кнопки, которая размещает круги в ряд в нижней части области просмотра на равном расстоянии друг от друга относительно середины фигуры, расстояние задается при помощи бегунка. (Рисунок 68)

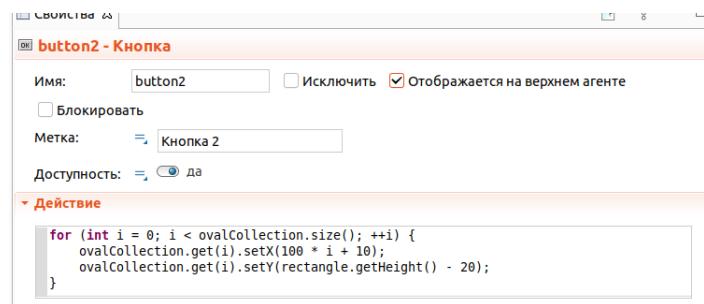


Рис. 68: Размещение кругов в ряд

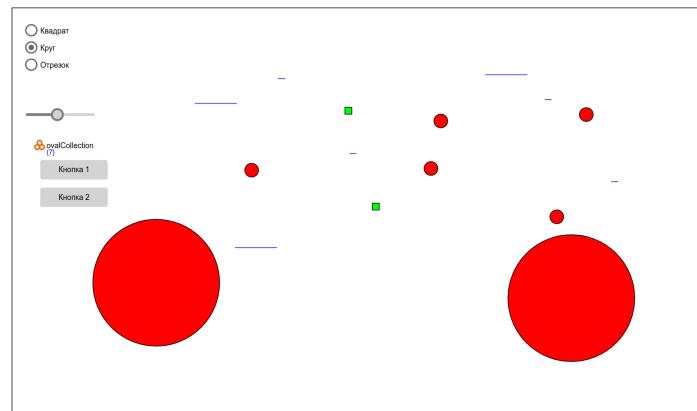


Рис. 69: Визуализация работы получившейся модели

Таким образом, нами был освоен механизм программного доступа к элементам презентации.

Библиотека моделирования потоков

Моделирование потоков

Задание:

Промоделировать процесс производства мороженого. Мороженое производится из молока, сахара и масла в пропорциях 60:10:30. Ингредиенты поступают в реактор-смеситель из резервуаров потрубопроводам – молоко и сахар, по контейнеру — масло. В смесителе составляющие смешиваются в заданных пропорциях и смесь настаивается 10 минут. Далее смесь по трубопроводу поступает в реактор заморозки. Процесс замораживания занимает 10 минут. Полученная смесь порциями по 100 граммов помещается в стаканчики. Стаканчики пакуются по 50 штук. Упаковки мороженого отправляются на склад.

Решение:

Для начала необходимо создать источники потоков. Источники потоков моделируются блоком *FluidSource*. Для трех источников ингредиентов потребуются 3 подобных блока. *FluidSource* может работать либо как источник с неограниченным объемом, либо как источник с ограниченным начальным объемом, который может наполняться заново вызовом функции *inject()*. Данный блок задает ограничение на скорость выходного потока, реальная скорость не может превосходить это значение. (Рисунок 70)

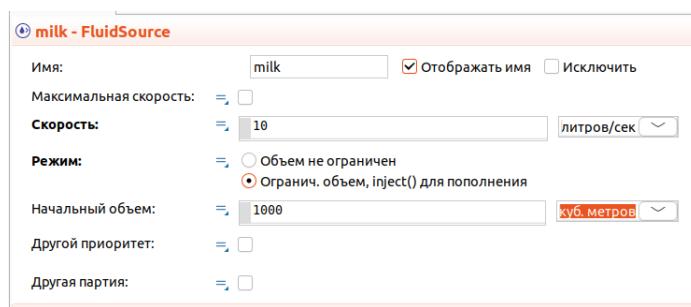


Рис. 70: Блок источник *FluidSource*

Скорости подачи ингредиентов разные, поэтому понадобятся резервуары для их хранения после того, как они потупили в модель. Резервуары моделируются блоком *Tank*, в котором задаются такие свойства как объем резервуаров и скорости потоков на их выходе. (Рисунок 71)

Доставка жидких ингредиентов в смеситель осуществляется по трубам. Для моделирования трубопроводов используется *Pipeline*. Также он моделирует трубу, по которой жидкость транспортируется из одной точки в

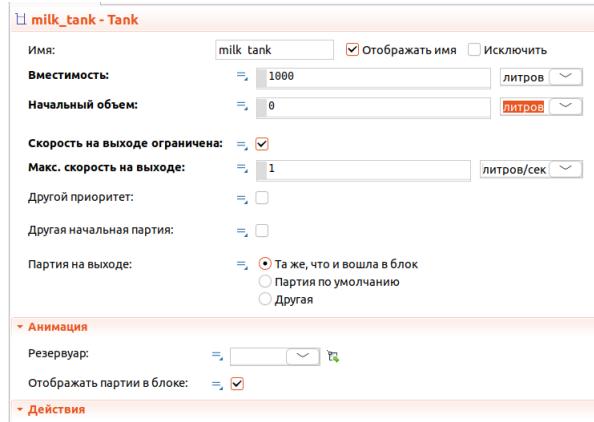


Рис. 71: Блок *Tank*

другую. Имеет ограниченный объем. Есть опция, позволяющая содержать некоторое начальное количество жидкости на входе. (Рисунок 72)

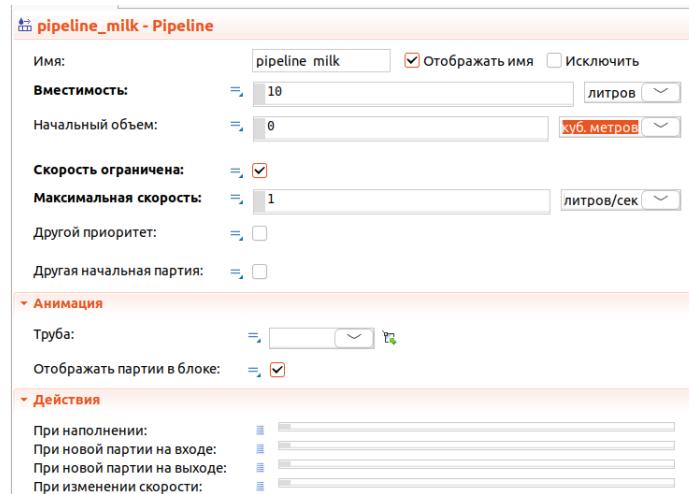


Рис. 72: Блок *Pipeline*

Доставка масла выполняется по конвейеру. Для моделирования этого конвейера будем используется объект *BulkConveyor*. Транспортирует объемные или конденсирующиеся летучие вещества из одной точки в другую. По сравнению с трубой *Pipeline*, допускает образование зазоров и участков с различной "плотностью". Скорость потока на входе конвейера не обязательно равна скорости потока на его выходе. (Рисунок 73)



Рис. 73: Блок *BulkConveyor*

Процесс смешивания моделируется блоком *MixTank*. У этого блока пять входов и один выход. На вход подаются ингредиенты в заданных пропорциях, на выходе – смесь в заданных пропорциях. (Рисунок 74)

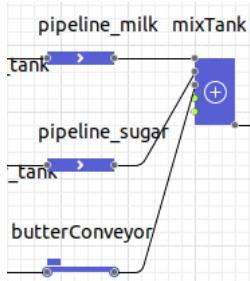


Рис. 74: Блок *MixTank*

Поскольку смесь заморожена, то доставка ее осуществляется конвейером для конденсированных веществ. Порция мороженого – это выделяемая из потока смеси заявка. Для процесса разделения на порции используем блок *FluidToAgent*. (Рисунок 75)

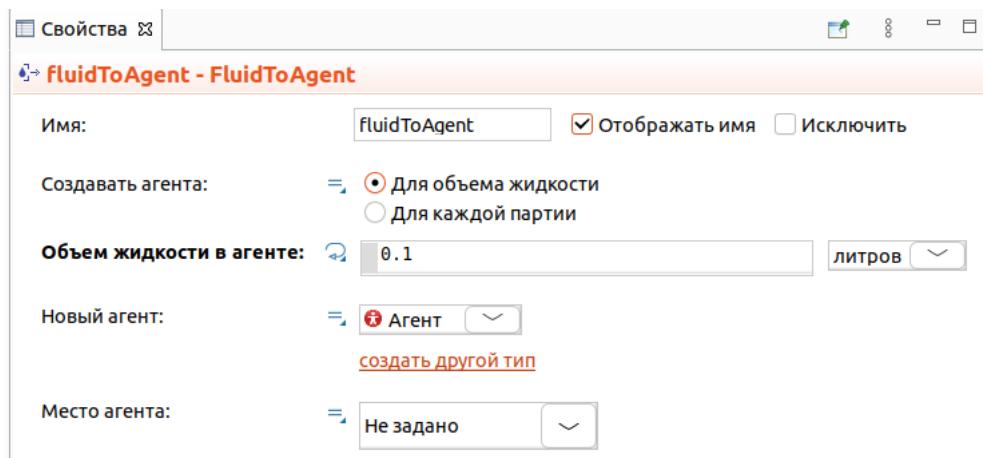


Рис. 75: Блок *FluidToAgent*

Порции смеси должны помещаться в стаканчики. Для моделирования процесса появления стаканчиков будем использовать библиотеку моделирования процессов. Источник стаканчиков – блок *Source*. Поскольку скорость производства мороженого и стаканчиков в модели разная, то необходимы их накопители. Накопители моделируем блоком *Queue*.

Сборка штучных заявок моделируется блоком *Assembler*. Этот блок имеет пять входов и один выход. Он может принимать до пяти агентов и собирать из них нового агента. Первый вход блока – выход очереди мороженого, второй – выход очереди стаканчиков. Моделирование доставки стаканчиков

мороженого до упаковщика моделируется конвейером. Упаковка моделируется объектом *Service*, в свойствах которого задается время процесса и его ресурсы. Ресурсы – упаковщики. Упаковку мороженого промоделируем объектом *Batch*, который собирает партии из входящих в него заявок. (Рисунок 76)

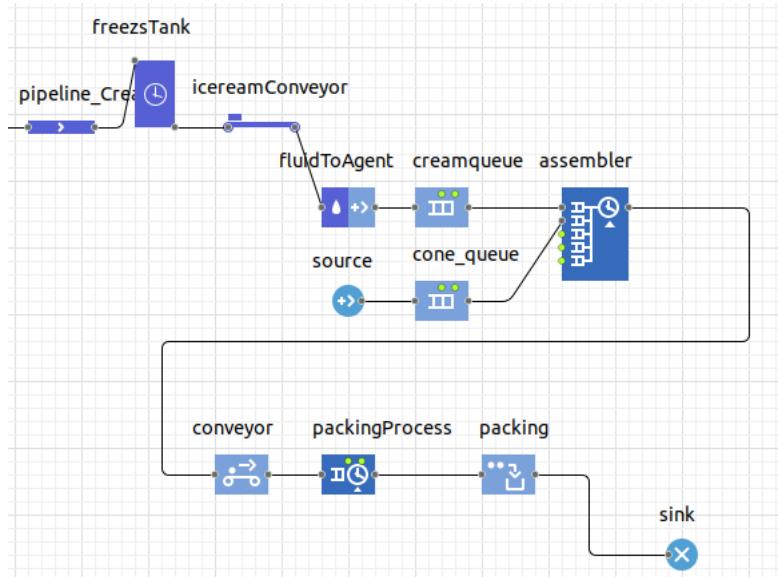


Рис. 76: Процесс сборки, упаковки и отправки на склад стаканчиков с мороженым

Узкими местами модели являются блоки *Tank*, однако, в связи с технологией производства мы не можем уменьшать время задержки в них, но можем увеличить скорость работы конвейеров и скорость потока в трубах и в начальных блоках *Tank*, тогда количество партий, отправленных на склад, увеличится. (Рисунок 77)

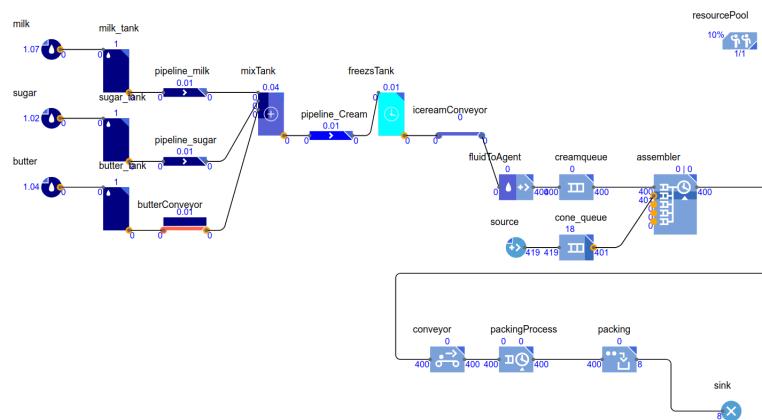


Рис. 77: Результат работы модели

Таким образом, был промоделирован процесс производства мороженого.

Пешеходная библиотека

Пешеходная библиотека. Первая модель

Задание:

Разработать модель движения пассажиров в наземном павильоне метро.

Решение:

Перед тем, как пройти к поездам метро, пассажиры проходят через турникеты, проверяющие наличие проездного документа. Некоторые пассажиры должны будут вначале приобрести жетоны или проездные билеты в кассе.

На первом шаге необходимо разметить пространство, а именно – отрисовать стены, целевые линии, которые указывают вход и выход из модели, разместить сервисы с очередями, чтобы обозначить турникеты и кассы. (Рисунок 78)

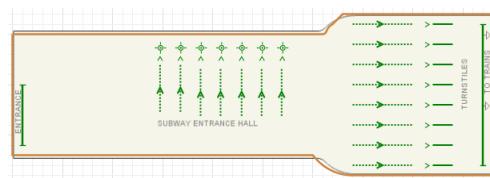


Рис. 78: Разметка пространства наземного павильона метро

Далее была разработана основная логика модели. (Рисунок 79)

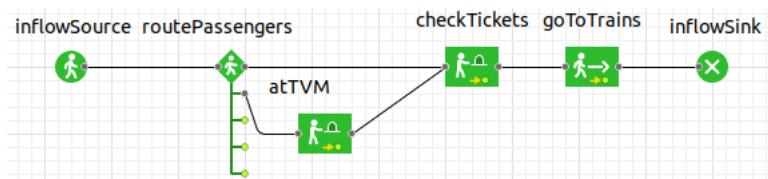


Рис. 79: Описание логики модели

Агенты поступают в модель согласно заданной интенсивности, затем у них есть выбор либо пойти в кассы и купить билет, либо сразу пойти к турникетам. Выбор осуществляется на основе заданной вероятности в 70%. (Рисунок 80)

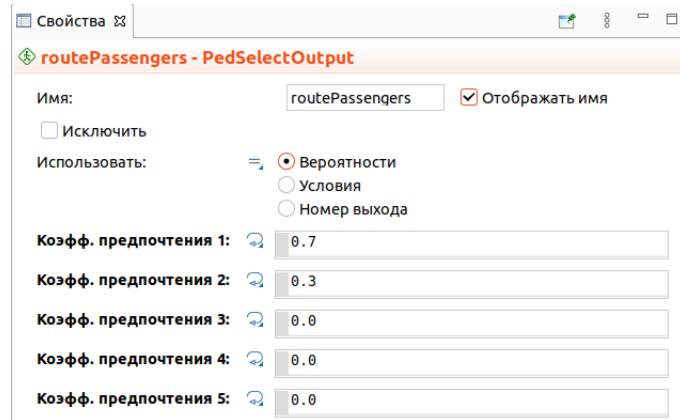


Рис. 80: Принятие решения в модели

Блоки обслуживания агентов задаются идентичным образом как это было в Библиотеке моделирования процессов. Выход из модели осуществляется путём достижения линии выхода.

Для отслеживания возникновения мест с большим скоплением агентов используем карту плотности, тогда модель будет выглядеть следующим образом. (Рисунок 81)

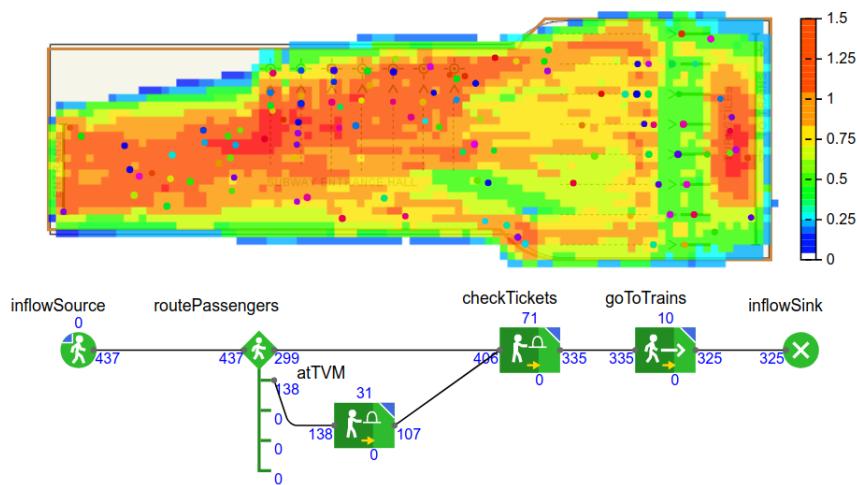


Рис. 81: Принятие решения в модели

Таким образом, на примере модели метро, нами был рассмотрен инструментарий пешеходной библиотеки.

Проект «Станция метро»

Задание:

Построить и проанализировать пешеходную имитационную модель станции метро Озерки.

Решение:

Имеется следующая схема станции метро Озерки. (Рисунок 82)

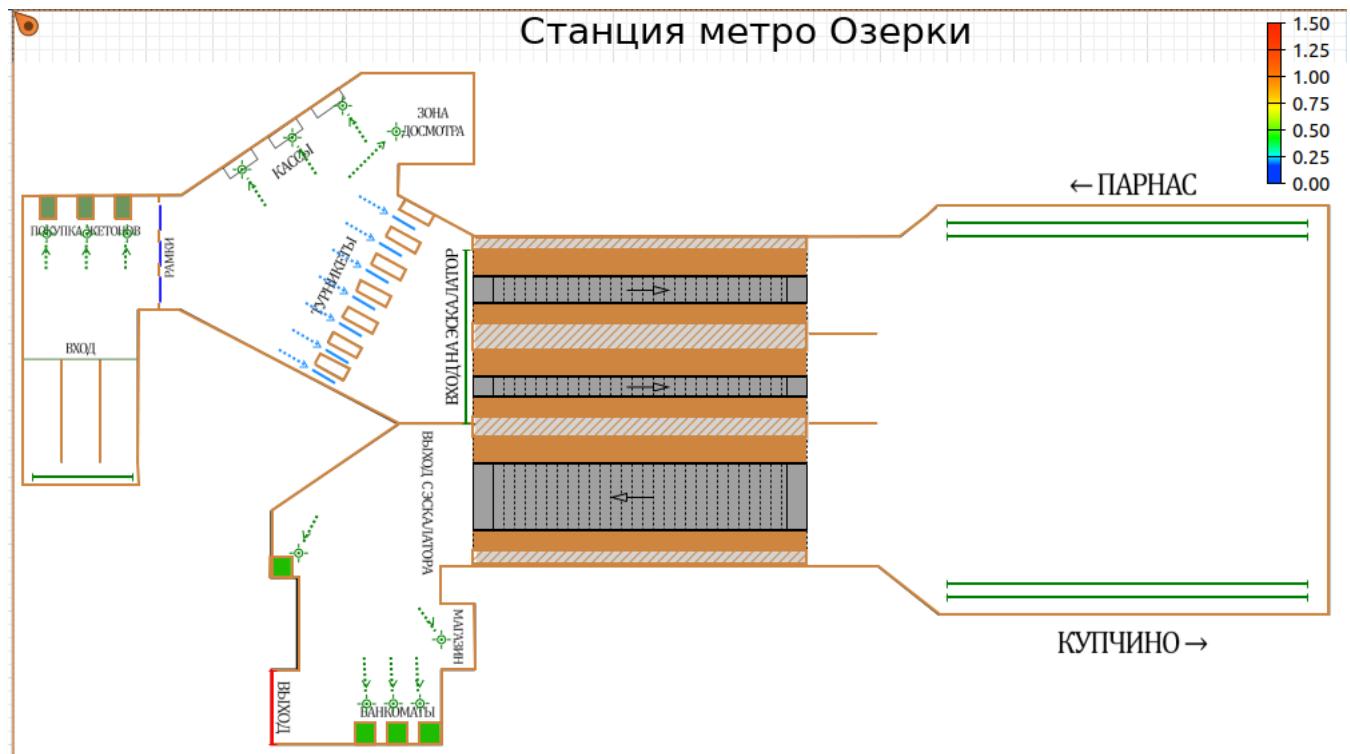


Рис. 82: Схема станции метро Озерки

При входе пассажиры могут купить жетоны либо в специальных автоматах, либо пойти на кассу, также их могут остановить на досмотр. После чего пассажиры проходят через турникеты и спускаются по эскалатору, далее они выбирают направление движения и садятся на поезд.

Соответственно, пассажиры, которые прибывают на станцию метро с других направлений, могут пройти по эскалатору на верх. После того как они поднялись, у них есть выбор пойти в группу банкоматов, пойти в «непопулярный» банкомат или зайти в магазин, также в магазине они смотрят на товар и в случае, если там нет нужного им продукта покинуть магазин или купить что-то. После всех данных альтернатив пассажиры покидают станцию метро.

В соответствии с описанием данная модель была реализована в среде моделирования *AnyLogic* (Рисунок 83).

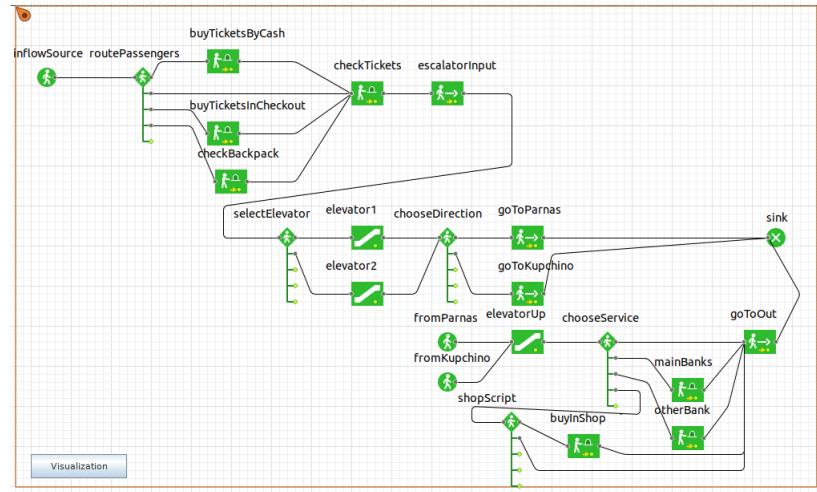


Рис. 83: Модель в среде *AnyLogic*

Данная модель не имеет модификаций с изменением направления эскалатора и имеет статический поток интенсивности пассажиров.

Также в соседнем окне была построена визуализация модели и тепловая карта, которая соответствует плотности различных участков станции. (Рисунок 84)

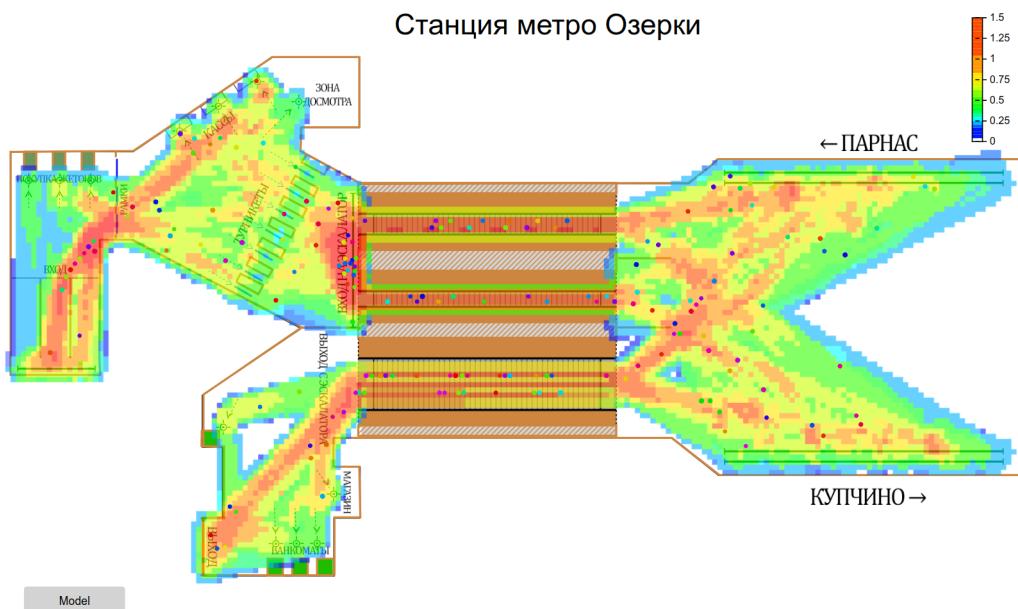


Рис. 84: Модель в среде *AnyLogic*

На данной тепловой карте можно видеть, что если средняя интенсивность пассажиропотока составляет 2000 человек в час, то «узким горлышком» на станции служит вход до рамок металлодетектора и входа на спуск по эскалатору.

Таким образом, нами была построена модель станции метро Озерки и была проанализирована зависимость плотности от различных факторов.

Внешние данные. Табличные функции

Чтение из Excel – фигуры

Задание:

Построить фигуры, считав данные о типах фигур и их параметрах из файла ФигурыExcel.xlsx (для отрезков указаны координаты концов, для прямоугольников – координаты левого верхнего угла).

Решение:

Для того, чтобы связать Excel файл с моделью, необходимо использовать блок – файл Excel. Для загрузки была создана кнопка *Load*. (Рисунок 85)



Рис. 85: Логика кнопки для загрузки данных из файла

При нажатии на данную кнопку происходит отрисовка фигур заданных типов с заданными параметрами. (Рисунок 86)

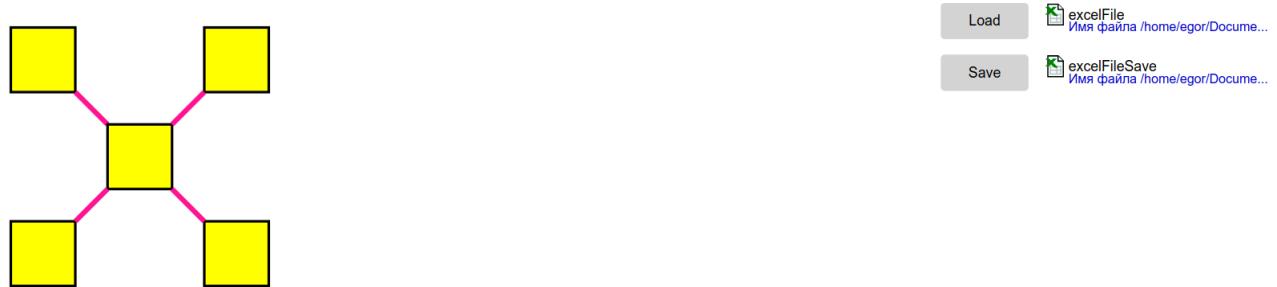


Рис. 86: Результат нажатия на кнопку *Load*

Таким образом, по данным из excel файла нами были построены необходимые фигуры, тем самым нами был рассмотрен процесс чтения данных из excel файла.

Фигуры из презентации в файл MS Excel

Задание:

Поместить в область просмотра несколько фигур из палитры Презентация. Запишите в файл Excel данные о размещённых фигурах – тип фигуры и параметры.

Решение:

Для того, чтобы связать Excel файл с моделью, необходимо использовать блок – файл Excel. Далее были расставлены элементы презентации. Для выгрузки была создана кнопка *Save*. (Рисунок 87)

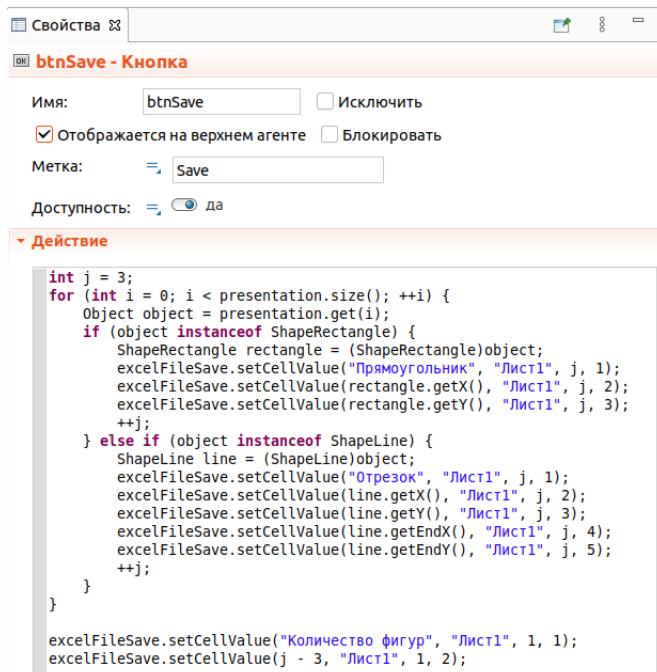


Рис. 87: Логика кнопки для выгрузки данных из файла

При нажатии на данную кнопку происходит сохранение фигур заданных типов с заданными параметрами. (Рисунок 88)

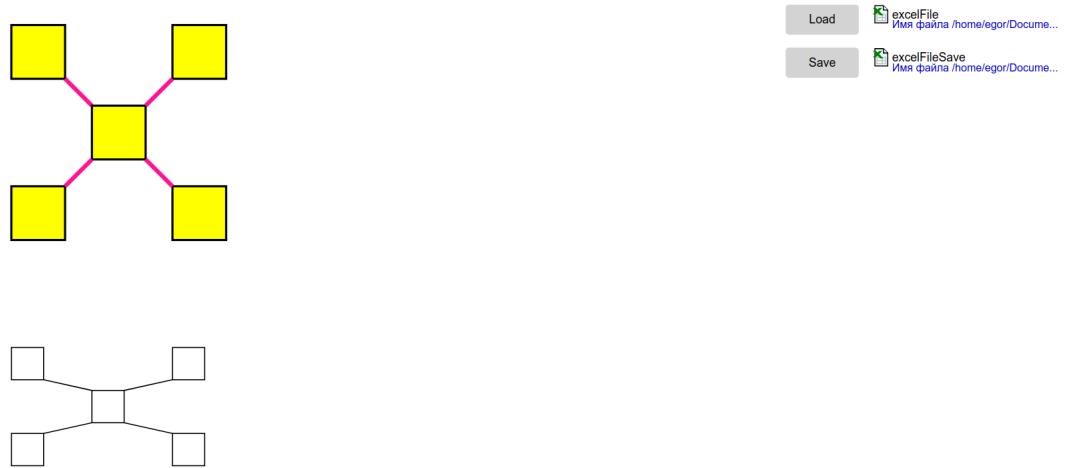


Рис. 88: Результат нажатия на кнопку *Save*

	A	B	C	D	E	F
1	Количество фигур	18				
2						
3	Прямоугольник	50	350			
4	Прямоугольник	200	350			
5	Прямоугольник	200	430			
6	Прямоугольник	50	430			
7	Прямоугольник	125	390			
8	Отрезок	80	380	125	390	
9	Отрезок	155	390	200	380	
10	Отрезок	80	430	125	420	
11	Отрезок	155	420	200	430	
12	Прямоугольник	50	50			
13	Отрезок	100	100	125	125	
14	Прямоугольник	125	125			
15	Прямоугольник	200	50			
16	Прямоугольник	50	200			
17	Прямоугольник	200	200			
18	Отрезок	200	100	175	125	
19	Отрезок	100	200	125	175	
20	Отрезок	200	200	175	175	
21						

Рис. 89: Получившийся набор данных

Таким образом, по данным из презентации нами были сформированы данные excel файла, тем самым нами был рассмотрен процесс экспорта данных в excel файл.

Системная динамика

Модель развития социального стресса

Задание:

Реализовать и проанализировать модель развития социального стресса.

Решение:

Имеется три фазы развития психологического стресса:

$$\begin{cases} \frac{dN_1}{dt} = -\alpha VN_1 N_2 - pVN_1 + qN_3 \\ \frac{dN_2}{dt} = \alpha VN_1 N_2 + pVN_1 - \beta N_2 \\ \frac{dN_3}{dt} = \beta N_2 - qN_3 \\ \frac{dV}{dt} = (cN_1 - r - m_0)V \end{cases}$$

V – степень «эмоциональной выраженности» информации, обусловленной стрессовым фактором.

Механизмы психологического давления:

1. $r = m \cdot N_3$
2. $r = m \cdot N_2$
3. $r = m \cdot (N_2 + N_3)$

Покажем поведение модели при начальных значениях $N_1(0) = 995$, $N_2(0) = 5$, $V(0) = 1$, $\alpha = 0.0005$, $q = 0.05$, $b = 1/15$, $c = 0.0001$, $m = 0.00005$, $m_0 = 0.05$, $H = 1000$, $p = 0.017$. (Рисунок 90)

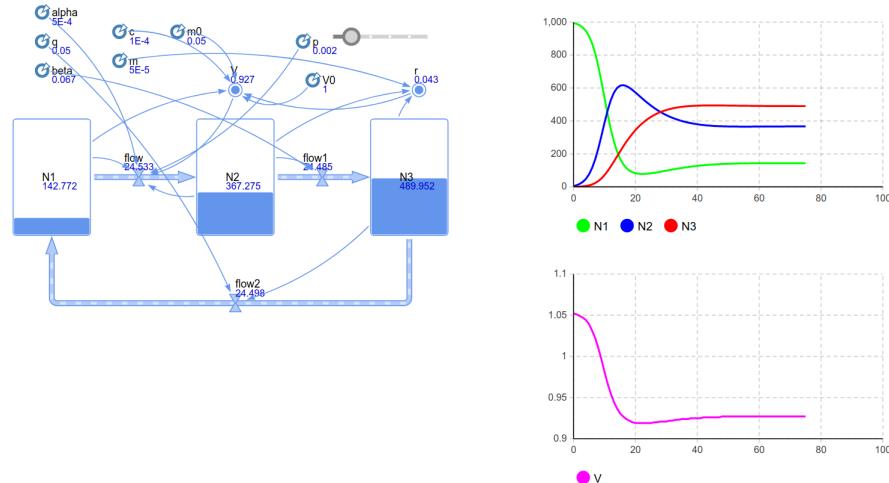


Рис. 90: Модель развития социального стресса при $p = 0.0017$

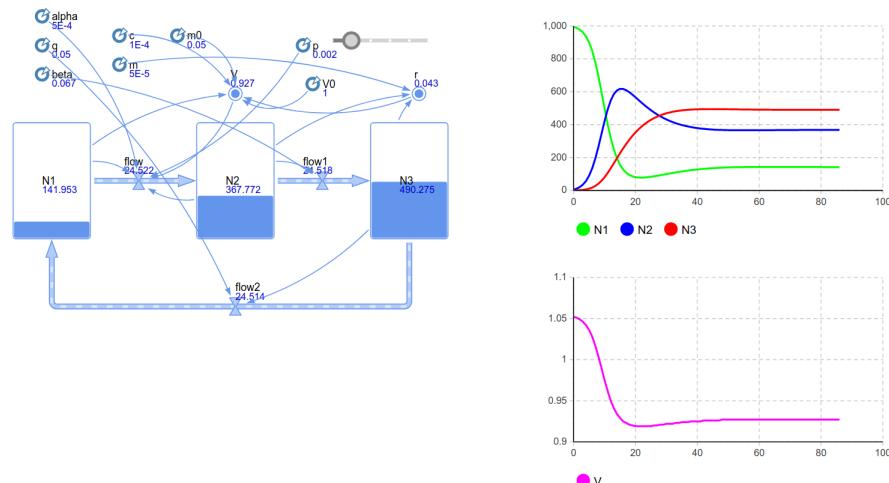


Рис. 91: Модель развития социального стресса при $p = 0.00246$

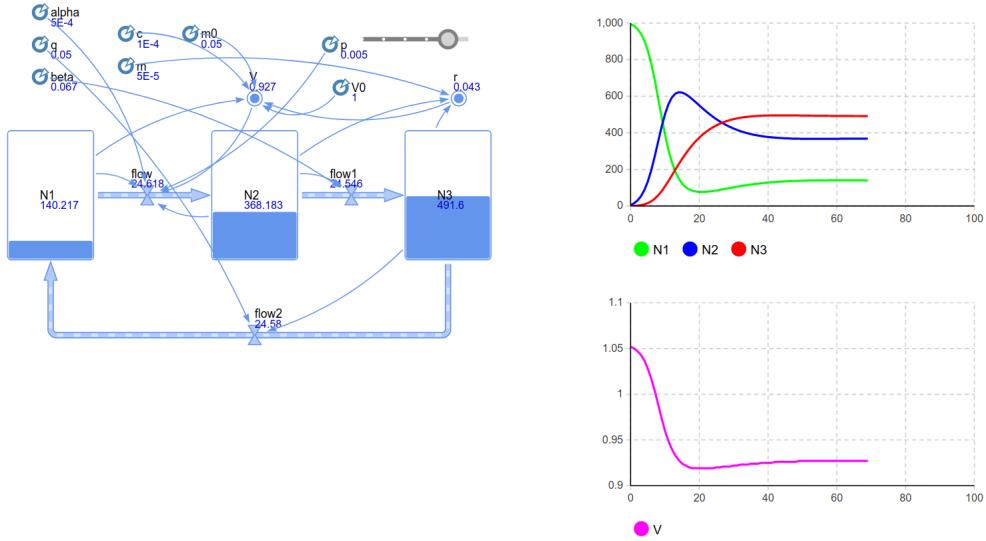


Рис. 92: Модель развития социального стресса при $p = 0.00534$

Из графиков можно видеть, что чем больше значение p , тем интенсивнее происходят изменения в составе групп. Также данная модель с течением времени приходит к стационарному состоянию из-за того, что V примет значение, при котором численность групп будет оставаться на том же уровне.

Таким образом, можно вывести закономерность, связанную с изменением численности групп подверженных стрессу. С ростом степени «эмоциональной выраженности» информации, обусловленной стрессовыми факторами, увеличивается численность людей в генерализированной стадии, с ростом численности данной группы увеличивается численность людей в восстановительной стадии, с ростом численности данной группы уменьшается степень «эмоциональной выраженности». С уменьшением степени «эмоциональной выраженности» возрастает численность людей в начальной стадии и уменьшается численность оставшихся групп.

Таким образом, была реализована модель развития социального стресса, был проведён численный анализ и были проанализированы взаимосвязи между переменными и параметрами модели.

Агентные модели. Дискретное пространство

«Жизнь» Конвея

Задание:

Реализовать модель «Game Of Life»

Решение:

Сначала была создана популяция размером – 2500 агентов, заданная в дискретном пространстве и имеющую тип соседства – Мурово. (Рисунок 93)

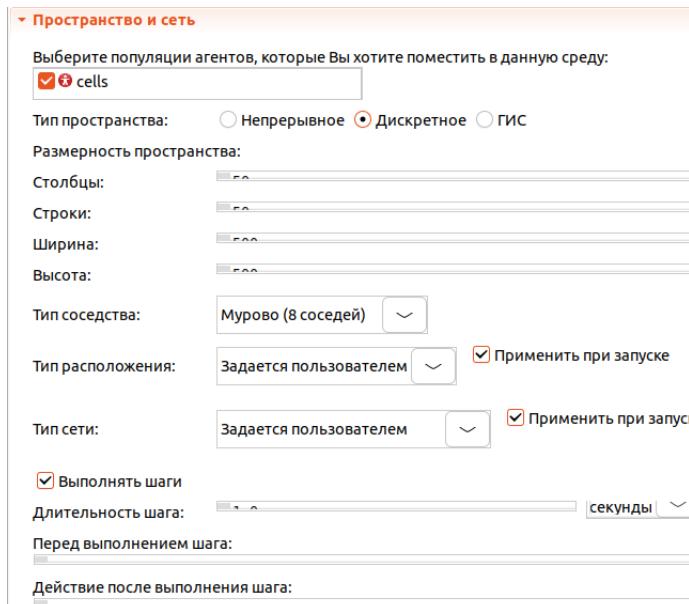


Рис. 93: Настройка популяции агентов

Цель данной модели заключается в том, что изначально каждой клетке присваивается состояние – жива или мёртва. Далее это состояние изменяется, если рядом с живой клеткой находится меньше двух или больше трёх живых клеток, то она умирает. Мёртвая клетка, рядом с которой находится ровно три живые клетки, оживает.

Для того чтобы реализовать данный алгоритм нужно перейти в популяцию агентов и задать действия перед выполнением шага и задать действие на каждом шаге. (Рисунок 94)

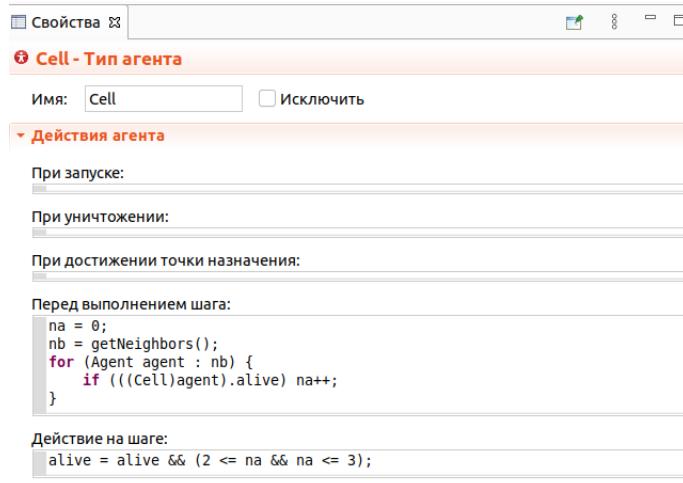


Рис. 94: Реализация алгоритма Игры в жизнь

В результате получается модель, в которой в соответствии с описанным алгоритмом меняются состояния клеток. Также можно поменять состояние клетки шёлкнув по ней. (Рисунок 95)

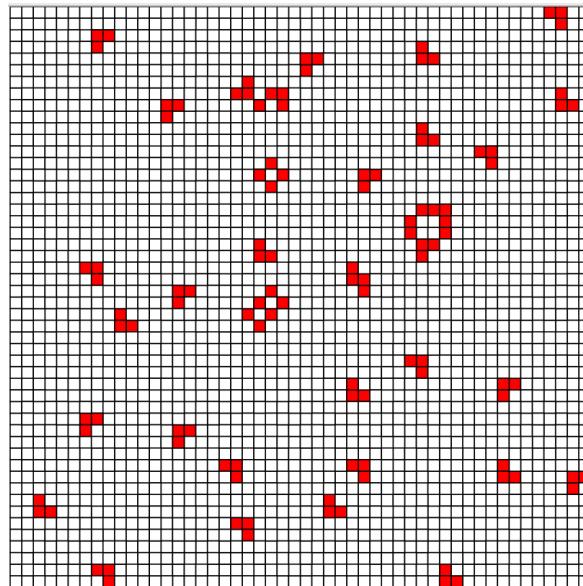


Рис. 95: Работа алгоритма Игры в жизнь

Таким образом, была реализована модель «жизнь» Конвея, на примере которой мы познакомились с агентными моделями в дискретном пространстве.

Модель сегрегации Шеллинга в AnyLogic

Задание:

Реализовать модель сегрегации Шеллинга в AnyLogic.

Решение:

Сначала была создана популяция размером – 9000 агентов, заданная в дискретном пространстве и имеющую тип соседства – Мурово. (Рисунок 96)

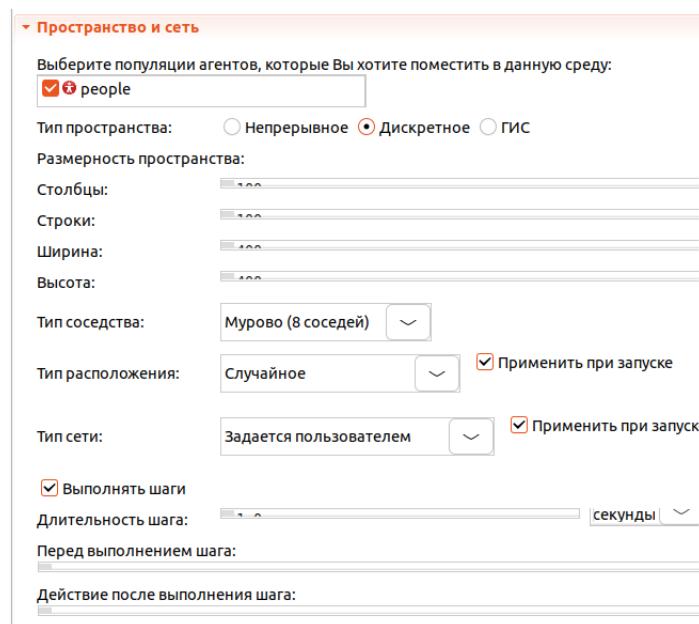


Рис. 96: Настройка популяции агентов

Цель данной модели заключается в том, чтобы промоделировать поведение людей. Существует несколько правил, по которым действуют люди в данной модели:

1. агент, который имеет только 1 агента соседа, переедет, если это сосед отличного цвета;
2. агент, имеющих 2 соседей, не будет переезжать, если хотя бы один из них имеет тот же цвет, что и он сам;
3. агент, проживающий по соседству от 3 до 5 человек, не будет переезжать, если хотя бы два из них будут цвета, что и он сам;
4. агент с соседями от 6 до 8 человек не будет переезжать, если хотя бы 3 из них будут одного цвета.

Также стоит сказать, что изначально агентам задаются случайные цвета с равной вероятностью. Таким образом, агент будет иметь два параметра: цвет и состояние счастья, которое принимает тип boolean. (Рисунок 97)



Рис. 97: Переменные и параметры агентов

Также в данной модификации модели рассматривается вариант при котором, агенты могут положительно взаимодействовать не только с соседями своего цвета, но и соседями других цветов. (Рисунок 98)

People - Тип агента

```
Перед выполнением шага:
int near = 0;
Agent[] neighbors = getNeighbors();

if (neighbors == null) {
    happiness = true;
    return;
}

for (Agent agent : neighbors) {
    if (((People)agent).color == color) {
        ++near;
    }
}

// Зелёные дружат с синими
if (get_Main().radio.getValue() == 1) {
    if ((color == Color.green && ((People)agent).color == Color.blue)
        || (color == Color.blue && ((People)agent).color == Color.green)) {
        ++near;
    }
}

// Красные дружат с зелёными
if (get_Main().radio.getValue() == 2) {
    if ((color == Color.red && ((People)agent).color == Color.blue)
        || (color == Color.blue && ((People)agent).color == Color.red)) {
        ++near;
    }
}

// Красные дружат с зелёными
if (get_Main().radio.getValue() == 3) {
    if ((color == Color.red && ((People)agent).color == Color.green)
        || (color == Color.green && ((People)agent).color == Color.red)) {
        ++near;
    }
}

if (color == Color.red) {
    happiness = (near >= get_Main().intoleranceLevelRed * neighbors.length);
} else if (color == Color.green) {
    happiness = (near >= get_Main().intoleranceLevelGreen * neighbors.length);
} else if (color == Color.blue) {
    happiness = (near >= get_Main().intoleranceLevelBlue * neighbors.length);
}
```

Рис. 98: Реализация алгоритма взаимодействия агентов

Для сбора статистики кто среди агентов счастлив, а кто нет, была создана круговая диаграмма, которая отражает данные показатели. (Рисунок 99)

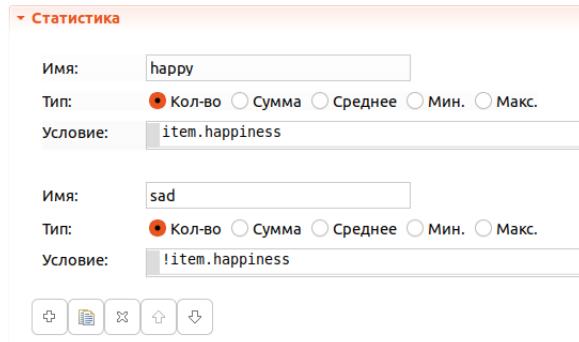


Рис. 99: Реализация алгоритма взаимодействия агентов

Также можно промоделировать различные сценарии взаимодействия агентов. (Рисунок 100)

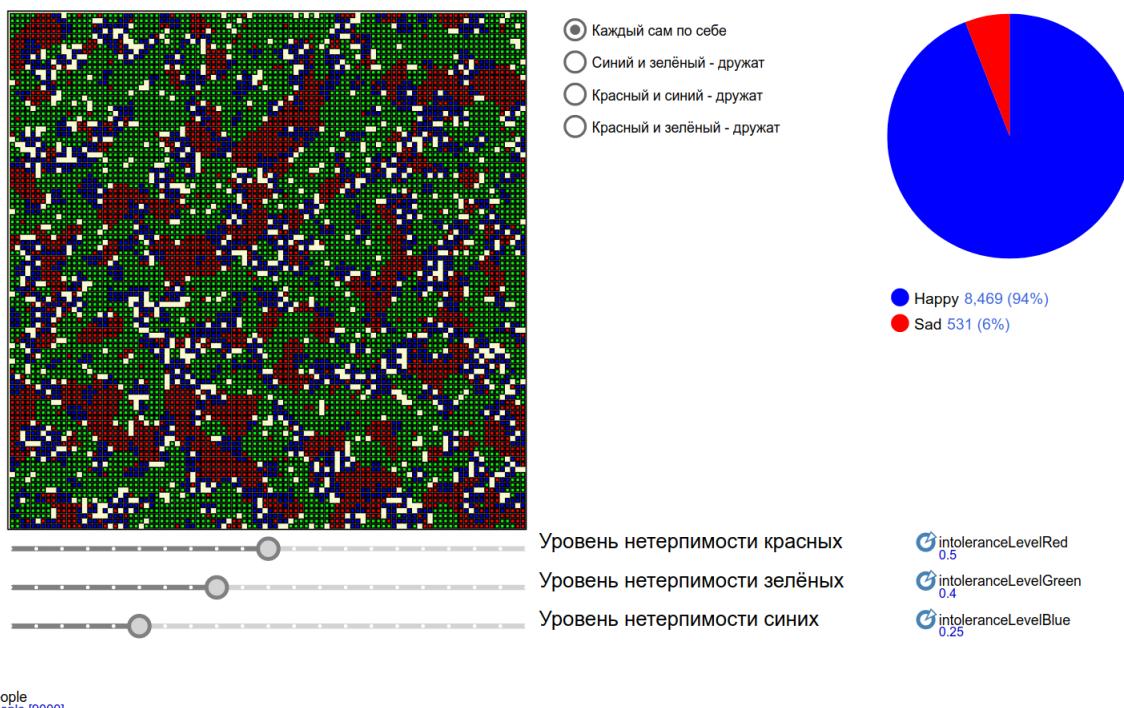


Рис. 100: Результат работы модели

Таким образом, была реализована модель сегрегации Шеллинга при различных условиях взаимодействия агентов.

Модель сегрегации Шеллинга в Python

Задание:

Реализовать модель сегрегации Шеллинга в Python.

Решение:

Суть данного алгоритма была описана в предыдущем разделе. Здесь же будет рассмотрена просто реализация предыдущей задачи на языке программирования Python.

Для начала был создан класс *Person*, который содержит информацию о члене популяции. У него имеется два поля: тип и счастлив ли данный агент или нет. (Рисунок 101)

```
class Person(object):
    def __init__(self, *args, types_count: int, **kwargs) -> None:
        self.type = random.choice([x for x in range(1, types_count + 1)])
        self.happiness = True

    def __repr__(self) -> None:
        return f"Person(type={self.type}, self.happiness={self.happiness})"
```

Рис. 101: Класс члена популяции *Person*

Далее был создан класс популяции – *Population*, в котором изначально генерировалась популяция агентов и задавалась начальная конфигурация системы. (Рисунок 102)

```
class Population(object):
    def __init__(self, *args,
                 dimention: int,
                 population_size: int,
                 types_count: int,
                 tolerance_levels: List[float],
                 **kwargs) -> None:
        self.dimention = dimention
        self._population_size = population_size
        self.types_count = types_count
        self._tolerance_levels = tolerance_levels

        self._population = [Person(types_count=types_count) for _ in range(population_size)]
        self.grid = np.zeros((dimention, dimention), dtype=Person)
        self._empty_cells = []

        self.simulation_results = []

    def _initial_empty_cells(self) -> None:
        for i in range(self.dimention):
            for j in range(self.dimention):
                if self.grid[i, j] == 0:
                    self._empty_cells.append((i, j))

    def _initial_configuration(self) -> None:
        grid_positions = [(i, j) for i in range(self.dimention) for j in range(self.dimention)]
        people_positions = random.sample(grid_positions, self._population_size)
        for i in range(len(self._population)):
            self.grid[people_positions[i]] = self._population[i]
        self._initial_empty_cells()
        self.simulation_results = [deepcopy(self.grid)]
```

Рис. 102: Класс популяции *Population*

Далее были созданы функции для перемещения агента в системе, изменения уровня счастья конкретного агента и собственно основной функции симуляции. (Рисунок 103)

```

def _get_neighbors_position(self, i: int, j: int) -> None:
    function = lambda element: not (element[0] < 0 or element[0] > self.dimention - 1 or \
                                    element[1] < 0 or element[1] > self.dimention - 1)
    return list(filter(function, [(i - 1, j + 1), (i, j + 1), (i + 1, j + 1), (i - 1, j), (i + 1, j), (i - 1, j - 1), (i, j - 1), (i + 1, j - 1)]))

def _agent_happy(self, agent: Person, i: int, j: int) -> None:
    near = 0
    neighbors_positions = self._get_neighbors_position(i, j)
    neighbors = [self.grid[neighbor] for neighbor in neighbors_positions]
    for neighbor in neighbors:
        if neighbor == 0 or neighbor.type == agent.type:
            near += 1
    agent.happiness = near >= self._tolerance_levels[agent.type - 1] * len(neighbors)

def _jump_to_random_empty_cell(self, i: int, j: int) -> None:
    agent = self.grid[i, j]
    new_position = random.choice(self._empty_cells)
    self._empty_cells.remove(new_position)
    self.grid[new_position] = Person(types_count=self.types_count)
    self.grid[new_position].type = agent.type
    self.grid[new_position].happiness = agent.happiness
    self.grid[i, j] = 0
    self._empty_cells.append((i, j))

def simulation(self, iterations_count: int) -> None:
    self._initial_configuration()
    for _ in range(iterations_count):
        for i in range(self.dimention):
            for j in range(self.dimention):
                if self.grid[i, j] != 0:
                    agent = self.grid[i, j]
                    self._agent_happy(agent, i, j)
                    if not agent.happiness and random.random() > 0.7:
                        self._jump_to_random_empty_cell(i, j)
    self.simulation_results.append(deepcopy(self.grid))

```

Рис. 103: Функции для перемещения агентов в системе и симуляции

Также были созданы вспомогательные функции для отрисовки текущего состояния системы и для анимации получившихся результатов. (Рисунок 104)

```

def plot_population(population: Population, *, colors: List[str]) -> None:
    paint_grid = [[population.grid[i, j].type if population.grid[i, j] != 0 else 0
                  for j in range(population.dimention)]
                  for i in range(population.dimention)]
    cmap = col.ListedColormap(colors)
    plt.figure(figsize=(10,10))
    plt.pcolormesh(paint_grid, edgecolor="black", cmap=cmap)
    plt.show()

```

Рис. 104: Функции для отрисовки текущего состояния системы

```

def draw_animation(population: Population, *, colors: List[str], step=1) -> HTML:
    fig, ax = plt.subplots(figsize=(10, 10))

    cmap = col.ListedColormap(colors)
    paint_grids = []

    for k in range(kIterations):
        paint_grids.append([[population.simulation_results[k][i, j].type if population.simulation_results[k][i, j] != 0 else 0
                            for j in range(population.dimention)]
                            for i in range(population.dimention)])

    def animate(i):
        return ax.pcolormesh(paint_grids[i], edgecolor="black", cmap=cmap)

    anim = animation.FuncAnimation(fig, animate, frames=range(0, kIterations, step), blit=False)
    plt.close()

    return HTML(anim.to_jshtml())

```

Рис. 105: Функции для анимации процесса симуляции

Ещё были реализованы функции для сбора статистики по популяции, а именно отслеживание количества счастливых агентов. (Рисунок 106)

```

def _stats(population: Population) -> Tuple[int, int, int]:
    happy, sad = 0, 0

    for i in range(population.dimention):
        for j in range(population.dimention):
            if population.grid[i, j] != 0:
                if population.grid[i, j].happiness:
                    happy += 1
                else:
                    sad += 1

    return (happy, sad, happy + sad)

def pie_population(population: Population) -> None:
    happy, sad, _ = _stats(population)
    plt.figure(figsize=(8, 8))
    plt.pie([happy, sad], autopct="%1.1f%%", labels=["Happy", "Sad"])
    plt.title("Соотношение счастья")
    plt.legend()
    plt.show()

```

Рис. 106: Функции сбора статистики по популяции

Визуализация различных уровней счастья агентов в модели, построенной при помощи Python, сходна с той, что была получена при использовании среды AnyLogic. (Рисунок 107)

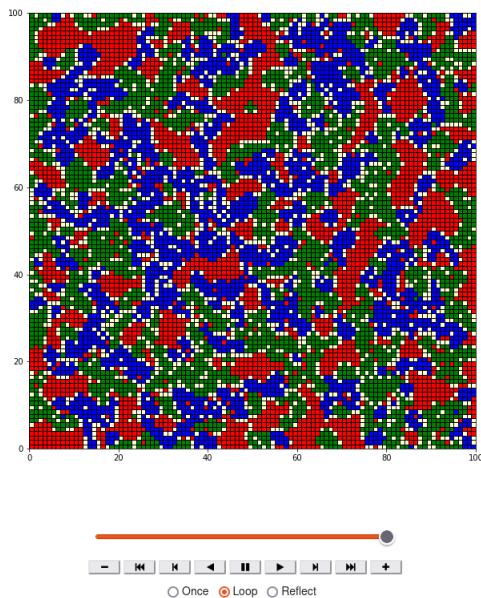


Рис. 107: Визуализация полученных результатов

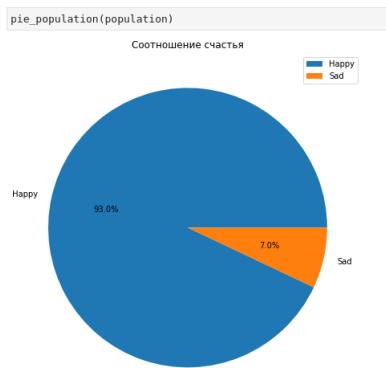


Рис. 108: Статистика по полученным результатам

Также были промоделированы различные другие ситуации, допустим когда рассматривается всего две типа людей или когда изменяется уровень толерантности групп.

Таким образом, была реализована модель сегрегации Шеллинга при различных условиях взаимодействия агентов.

Диаграммы состояний и события

Светофор

Задание:

Реализовать модели работы светофоров при помощи диаграммы состояний.

Решение:

Необходимо промоделировать работу двух светофоров – дорожного и пешеходного. Сначала стоит нарисовать данные объекты. (Рисунок 109)

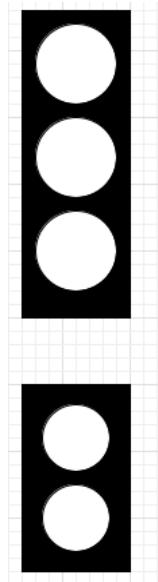


Рис. 109: Визуальное отображение светофоров

Далее была создана диаграмма действий, в которой реализована основная логика светофора. (Рисунок 110)

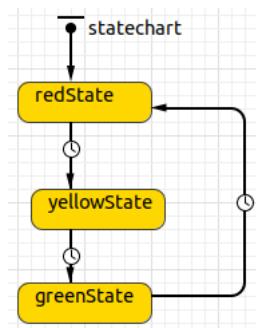


Рис. 110: Диаграмма состояний светофоров

Можно сказать что данная логика реализована относительно дорожного светофора.

Все переходы модели осуществляются по таймауту. Рассмотрим логику работы светофоров, когда на дорожном светофоре горит красный свет. Мы назначаем дорожному светофору красный свет, а пешеходному назначаем зелёный. (Рисунок 111)

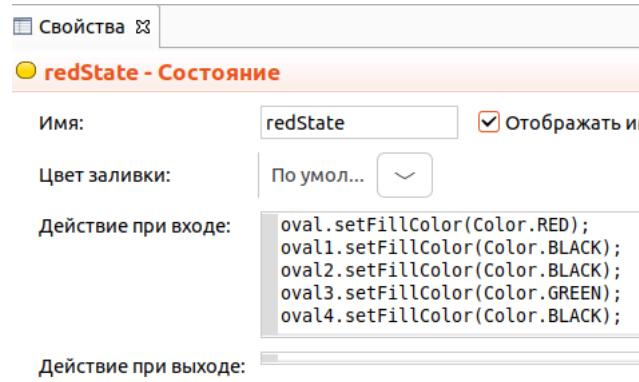


Рис. 111: Пример логики одного из состояний

Аналогично было проделано для жёлтого и зелёного состояний.

Таким образом, при помощи диаграммы состояний была промоделирована работа изменения сигналов дорожного и пешеходного светофоров.

SIR – агентная модель

Задание:

Реализовать и проанализировать модель распространения инфекционного заболевания посредством агентного моделирования.

Решение:

Изначально необходимо создать популяцию людей в размере – 1000 человек, в непрерывном пространстве, имеющую упорядоченный тип расположения и случайный тип сети с двумя связями. (Рисунок 112)

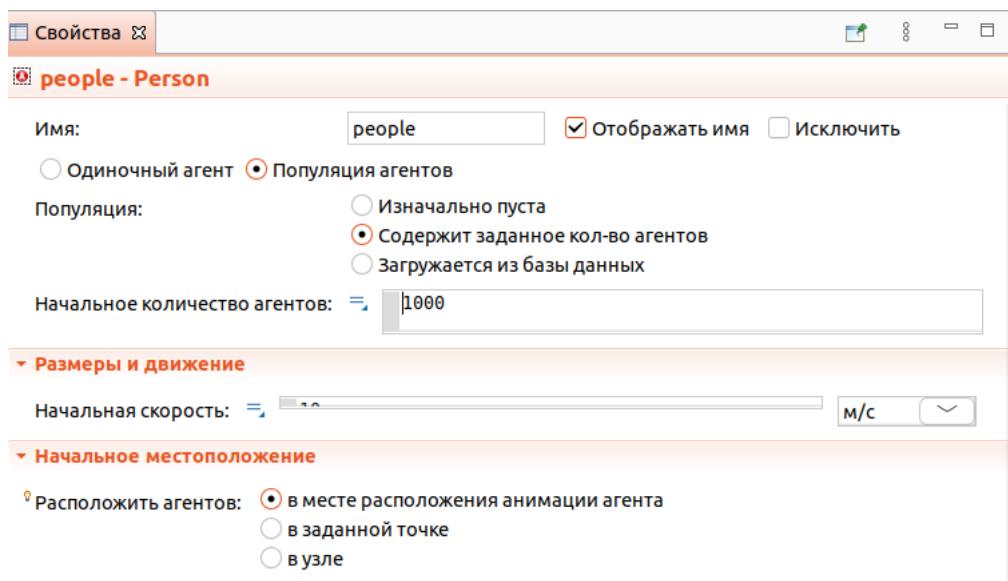


Рис. 112: Настройка агентов модели

Далее для всех агентов указываются количество дней, требуемое на восстановление, количество контактов и вероятность заразиться. После задания необходимых параметров модели было решено перейти описанию диаграммы состояний для конкретного агента в популяции. (Рисунок 113)

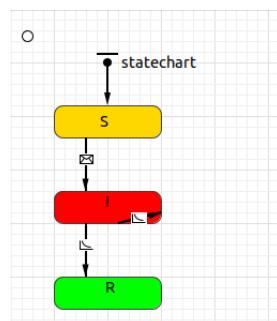


Рис. 113: Описание диаграммы состояний конкретного агента в популяции

В приведённой выше диаграмме человек может находиться в состояниях:

1. когда человек здоров и может заразиться;
2. когда человек заразился и болеет;
3. когда человек выздоровел и уже не может заразиться.

Переход из первого состояния во второе происходит на основе получения сообщения, отправка сообщения заражённым агентом осуществляется с интенсивностью, равной вероятности заразиться умноженной на количество контактов. Переход из состояния, когда человек болеет в состояние выздоровления происходит согласно интенсивности, равной единице делённой на количество дней, требуемых на восстановления после заболевания данной болезнью.

Для начала распространения инфекции необходимо инфицировать нескольких агентов в начале симуляции. (Рисунок 114)

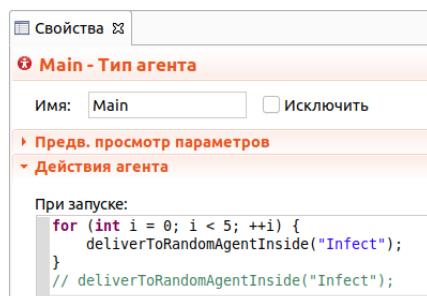


Рис. 114: Инфицирование нескольких агентов на начальной стадии

Также был проведёт сбор статистики по числу агентов, которые находятся в рассмотренных выше состояниях для того, чтобы на графиках проанализировать текущую ситуацию модели и как данная модель отличается от рассмотренной ранее. (Рисунок 115)

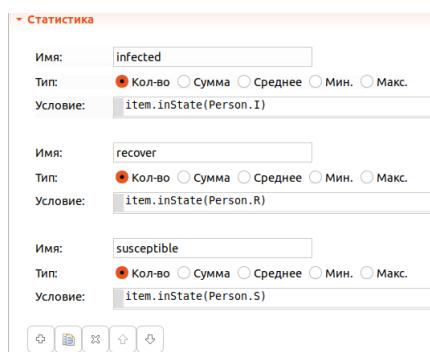


Рис. 115: Сбор статистики по каждому агенту

Если запустить данную модель, то получим схожую картину, которая была когда мы рассматривали системную-динамику. (Рисунок 116)

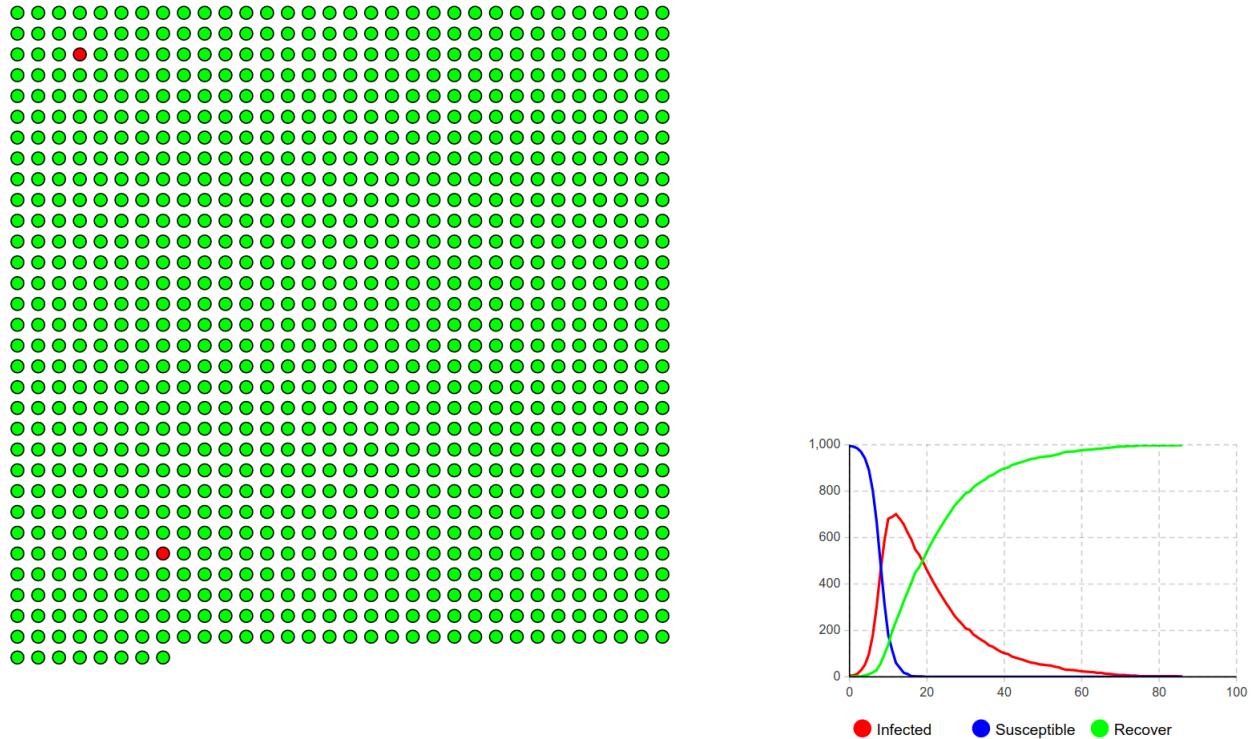


Рис. 116: Процесс моделирования

Таким образом, посредством агентного моделирования была реализована модель распространения инфекций.

Ноутбук + зарядка

Задание:

Реализовать модель работы ноутбука при условии, что присутствует возможность зарядки.

Решение:

Для начала стоит с помощью элементов презентации нарисовать мышку и сам ноутбук. (Рисунок 117)

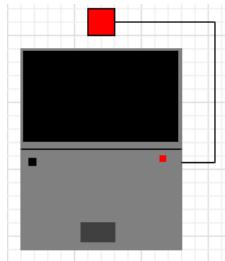


Рис. 117: Визуализация ноутбука и мышки

Далее была описана логика работы и построены диаграммы состояний. (Рисунок 118)

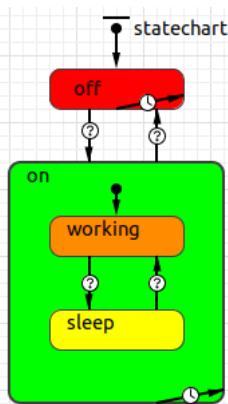


Рис. 118: Диаграммы состояний

Всего существуют два глобальных состояний: ноутбук включен и ноутбук выключен. Пока ноутбук включен, то у него постепенно расходуется заряд, если заряд ноутбука заканчивается, то он переходит в состояние выключения. Для отслеживания текущего состояния переменных у двух состояний есть внутренние переходы по таймауту, которые обновляются каждую секунду. Как только ноутбук подключается к зарядке, то им снова можно пользоваться.

Также в глобальном состоянии, когда ноутбук включен есть два внутренних: работает и в состоянии сна. Если не нажимать на кнопки на клавиатуре или на тачпад, то ноутбук перейдет в состояния сна и продолжит потреблять зарядку.

Ещё у ноутбука есть кнопка включения и выключения, чтобы пользователь мог сам переключаться между состояниями.

Реализация логики одного из внутренних переходов ноутбука представлена ниже. (Рисунок 119)

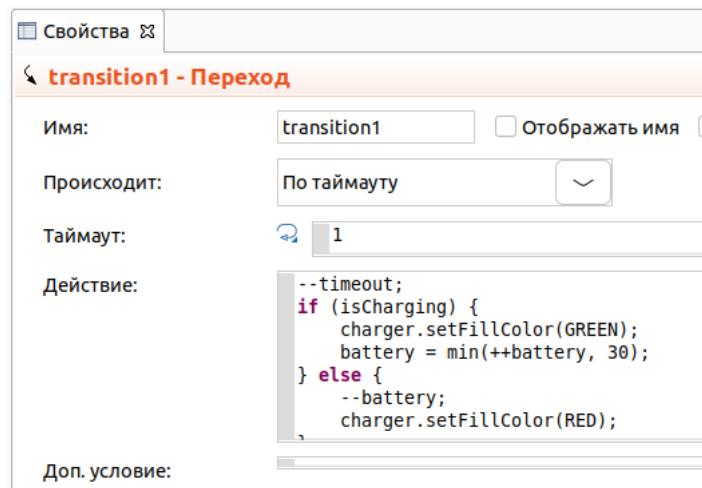


Рис. 119: Реализация логики внутреннего перехода в ноутбуке

Таким образом, была реализована модель, имитирующая работу ноутбука с зарядкой.

SIERD + вакцинация

Задание:

Реализовать модель распространения заболевания с добавлением новых состояний: носитель заболевания, умершие и вакцинированные.

Решение:

Данная модель реализуется похожую логику на ту которая была в модели SIR, поэтому в данном разделе будут рассмотрены только модификации.

В качестве новых параметров добавились вероятность вакцинации, вероятность выжить и время, которое должно пройти с момента вакцинации, чтобы вакцина успела действовать.

Диаграмма состояний в таком случае будет изменена и примет следующий вид. (Рисунок 120)

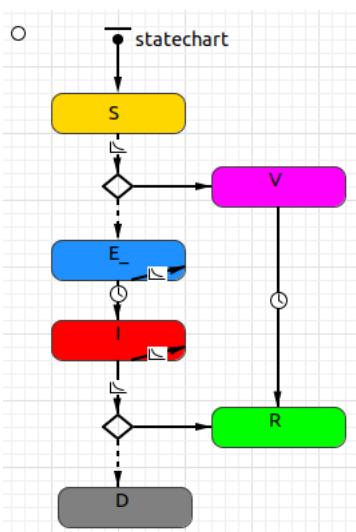


Рис. 120: Диаграмма состояний

Теперь из состояния когда человек полностью здоров можно перейти в состояние, когда человек вакцинирован или является носителем. Данный переход происходит с вероятностью, для которой был введён отдельный параметр. Если агент вакцинировался, то он сразу же попадает в состояние – выздоровевший.

Если же человек стал носителем, то дальше через некоторое время он заболевает, и в процессе пока он является носителем он также может заражать других здоровых людей. Далее из состояния больного агент может перейти в состояние умершего или выздоровевшего с заданной вероятностью. Для

этого был введён специальный параметр, который рассмотрен чуть выше.

В результате работы данной модели был получен график, который похож, на тот, что мы рассматривали на системной динамике. (Рисунок 121)

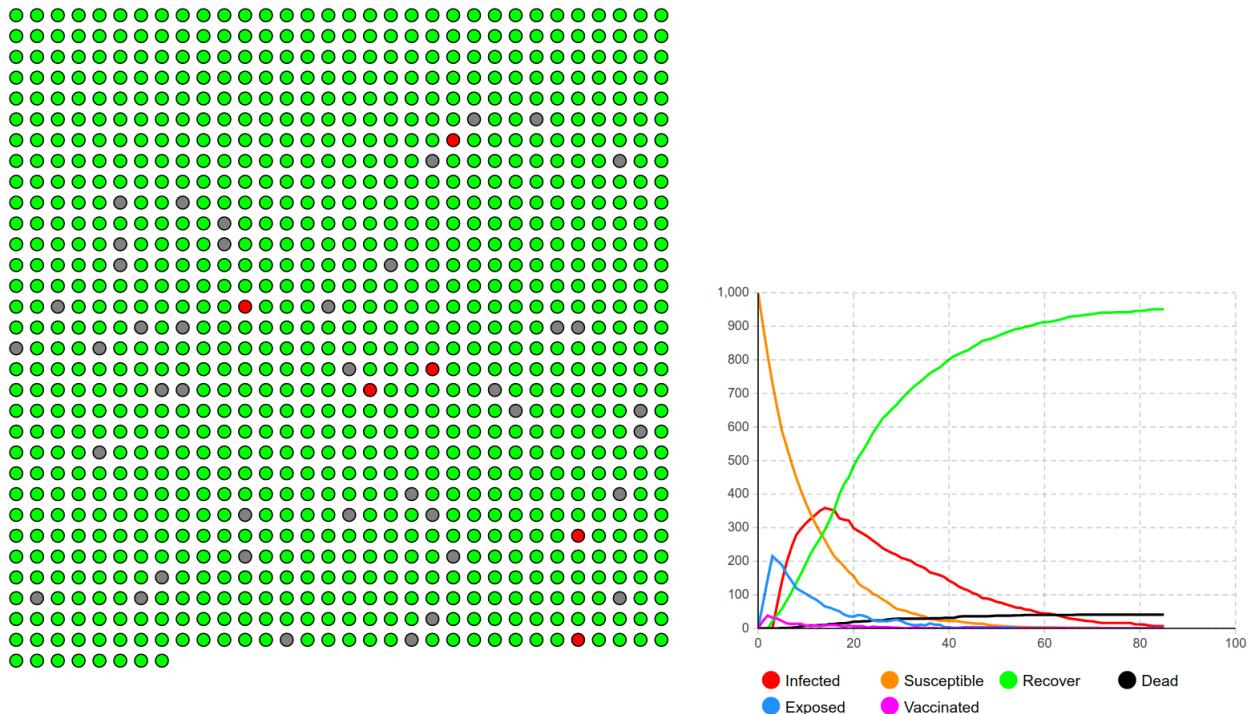


Рис. 121: Результат модели SIERD

Таким образом, посредством агентного моделирования была построена модель распространения заболевания с учётом вакцинации.

Диаграмма действий

Сумма ряда – экспонента

Задание:

Используя диаграмму действий, реализовать алгоритм вычисления суммы ряда экспоненты.

Решение:

Для вычисления экспоненты воспользуемся суммой ряда Тейлора:

$$e^x \approx \sum_{k=0}^N \frac{x^k}{k!}$$

В соответствии с данной формулой была построена рекуррентная формула для вычисления данной суммы. (Рисунок 122)

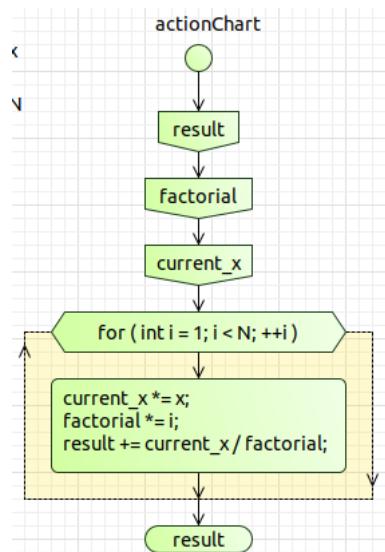


Рис. 122: Диаграмма действий для суммы ряда экспоненты

На вход данному алгоритму поступает количество итераций и степень x . (Рисунок 123)

1	G_x
10	G_{10}

Power: 1
Precision: 10
Result: 2.7182815255731922

Рис. 123: Результаты вычисления ряда экспоненты

Результаты расчёта совпали с результатами Wolfram Mathematica.

Таким образом, на примере создания функции для нахождения суммы ряда экспоненты был освоен процесс работы с диаграммами действий.

Сумма ряда – синус

Задание:

Используя диаграмму действий, реализовать алгоритм вычисления суммы ряда синуса.

Решение:

Для вычисления экспоненты воспользуемся суммой ряда Тейлора:

$$\sin x \approx \sum_{k=0}^N (-1)^k \frac{x^{2k+1}}{(2k+1)!}$$

В соответствии с данной формулой была построена рекуррентная формула для вычисления данной суммы. (Рисунок 124)

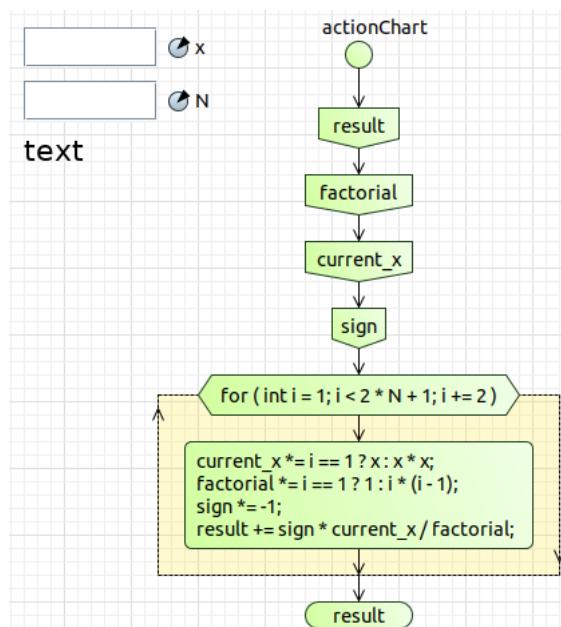


Рис. 124: Диаграмма действий для суммы ряда синуса

На вход данному алгоритму поступает количество итераций и степень x . (Рисунок 125)

3.14 $\odot_{3.14}$
200 \odot_{200}
Power: 3.14
Precision: 200
Result: 0.0015926529164867666

Рис. 125: Результаты вычисления ряда синуса

Результаты расчёта совпали с результатами Wolfram Mathematica.

Таким образом, на примере создания функции для нахождения суммы ряда синуса был освоен процесс работы с диаграммами действий.

Агентные модели

Подготовка к зачёту

Задание:

Реализовать модель подготовки к зачёту для одного студента.

Для подготовки к зачёту дано N вопросов.

Студент использует следующую схему подготовки.

1. Сначала он прорабатывает теоретический материал по очередному вопросу (и старается запомнить). Среднее время для этого по каждому вопросу одной темы колеблется в диапазоне от 30 до 40 минут (распределение равномерное).
2. Затем студент по памяти воспроизводит материал изученного вопроса.
3. Если материал не усвоен, то требуется повторение.
4. После однократной повторной проработки вопроса студент переходит к следующему вопросу.

Процесс продолжается до тех пор, пока не будут подготовлены все вопросы.

Степень усвоения изученного материала задается случайно (треугольное распределение от 5 до 100, мода - 60).

Решение:

Сначала необходимо создать популяцию вопросов. (Рисунок 126)

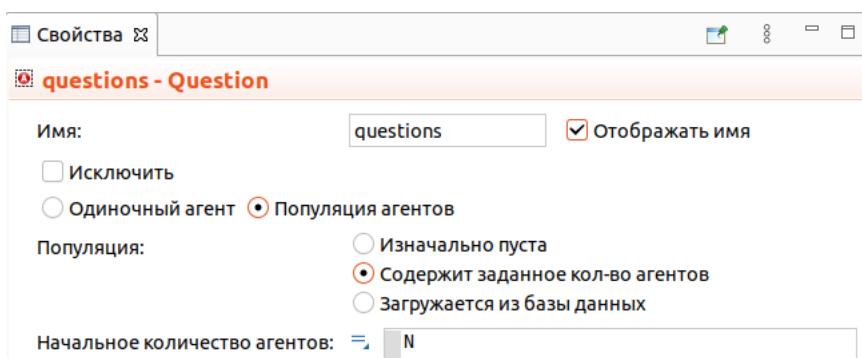


Рис. 126: Настройка популяции

Каждый из вопросов имеет несколько состояний. (Рисунок 127)

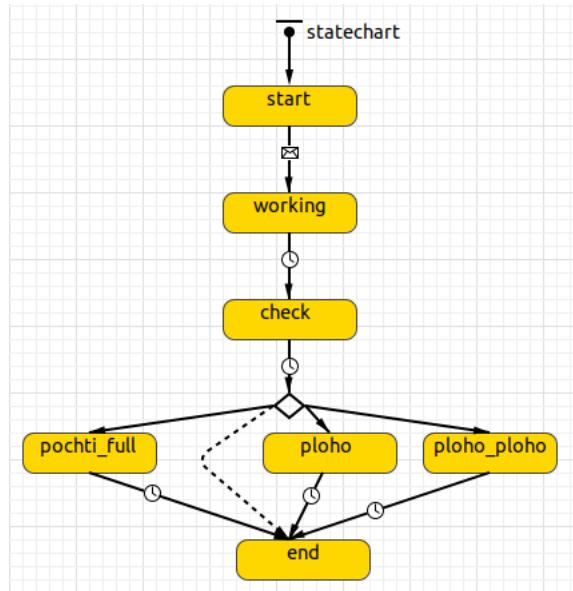


Рис. 127: Диаграмма состояний одного агента популяции

Переход между состояниями осуществляется в соответствии с правилами описанными в условии. Так как студент начинает подготовку вопроса при получении сообщения, то необходимо перед началом симуляции отправить сообщение о начале подготовки к первому вопросу. (Рисунок 128)

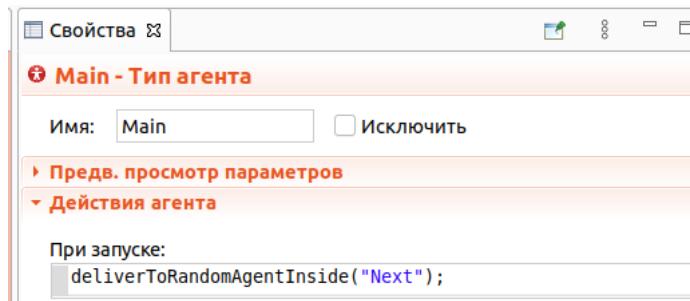


Рис. 128: Отправка сообщения первому агенту из популяции вопросов

Если запустить симуляцию, то будет следующий результат. (Рисунок 129)

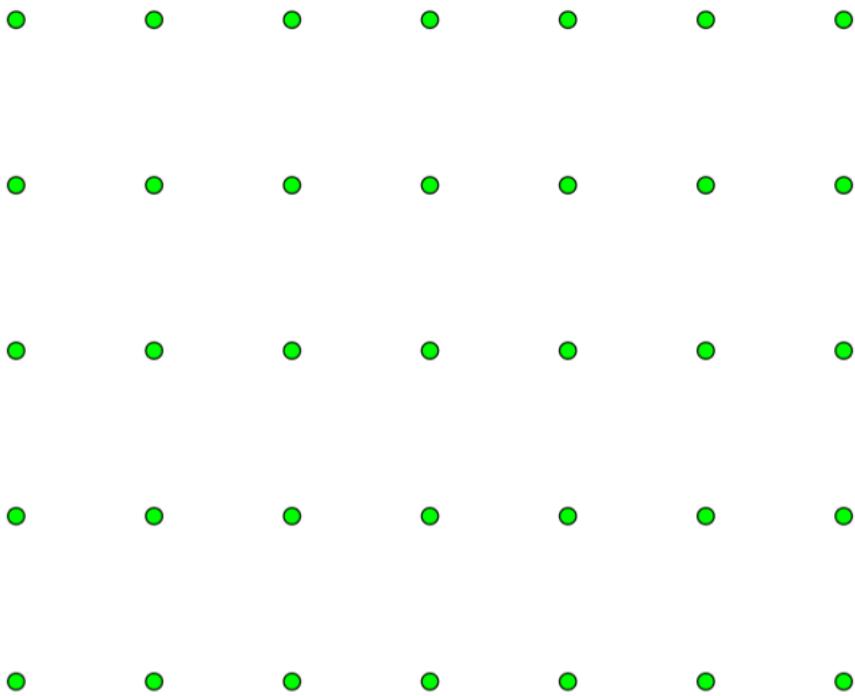


Рис. 129: Результат работы модели

Таким образом, была получена модель подготовки студента к зачёту.

Связи агентов – добавление, удаление, список, количество

Задание:

1. Создать популяцию агентов. Отобразить агенты и их связи. Для визуализации агентов использовать круг и текст с номером агента в популяции.
2. Предусмотреть:
 - добавление в популяцию нового агента отображение обновленной популяции;
 - добавление/удаление связи между случайными агентами;
 - добавление/удаление связи между агентами с заданными номерами. Номера задавать с помощью выпадающих списков.
3. Выделить цветом агента с максимальным количеством связей, вывести количество связей.
4. Выделить цветом агентов, связанных с заданным.
5. Предусмотреть удаление сделанных цветом отметок.

Решение:

Для начала необходимо создать популяцию агентов. У агентов будет их порядковый номер, текстовая метка которая отражает данную информацию и текстовая метка, которая показывает сколько других агентов связано с текущим. (Рисунок 130)

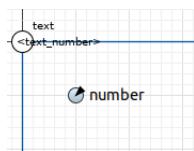


Рис. 130: Параметры популяции

Добавление новых агентов осуществляется по кнопке, также в данной кнопке осуществляется добавление нового агента в ComboBox. (Рисунок 131)

```
▼ Действие
add_humans(humans.size() + 1);
applyLayout();

String[] items = new String[humans.size()];
for (int i = 1; i <= humans.size(); ++i) {
    items[i - 1] = String.valueOf(i);
}
combobox.setItems(items);
combobox1.setItems(items);
```

Рис. 131: Реализация логики добавления нового агента

В самом начале при запуске модели осуществляется добавление всех сгенерированных агентов в ComboBox. (Рисунок 132)

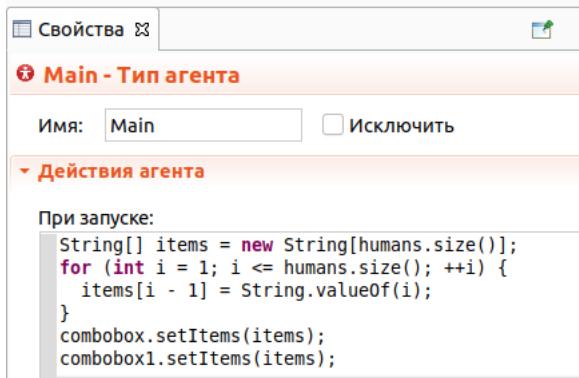


Рис. 132: Задание начальных значений выпадающих списков

Добавление связи между случайными агентами также осуществляется через кнопку. В данном алгоритме делается 100 раз попытка найти подходящую случайную вершину, то есть такую, которая не является текущей вершиной или вершиной-соседом с которым уже есть связь. Если за 100 итераций, такая вершина нашлась, то с ней возникает связь. (Рисунок 133)

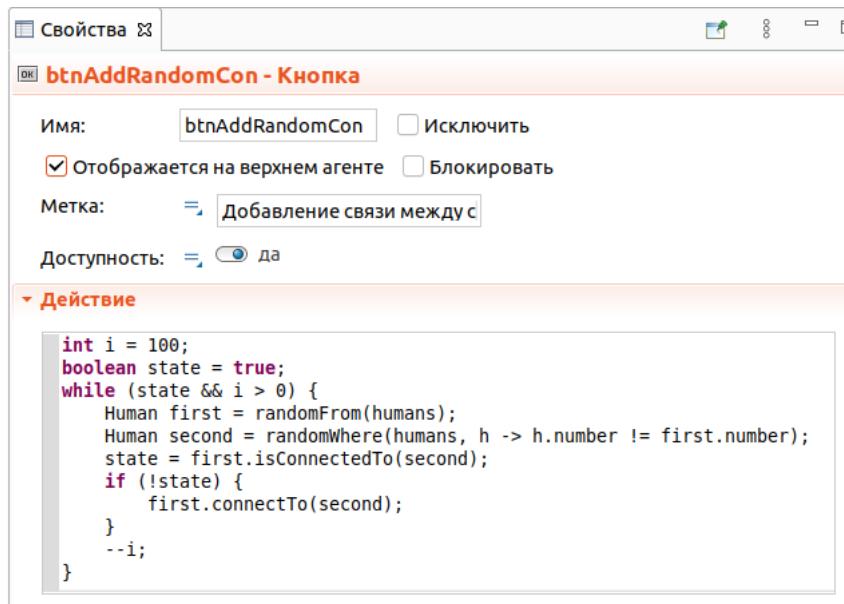


Рис. 133: Добавление связи между случайными агентами

Удаление связи между случайными агентами также реализовано по кнопке. В данном алгоритме делается 100 раз попытка найти подходящую случайную вершину, то есть такую, которая не является текущей вершиной

или вершиной-соседом с которым нет связи. Если за 100 итераций, такая вершина нашлась, то связь с ней удаляется. (Рисунок 134)

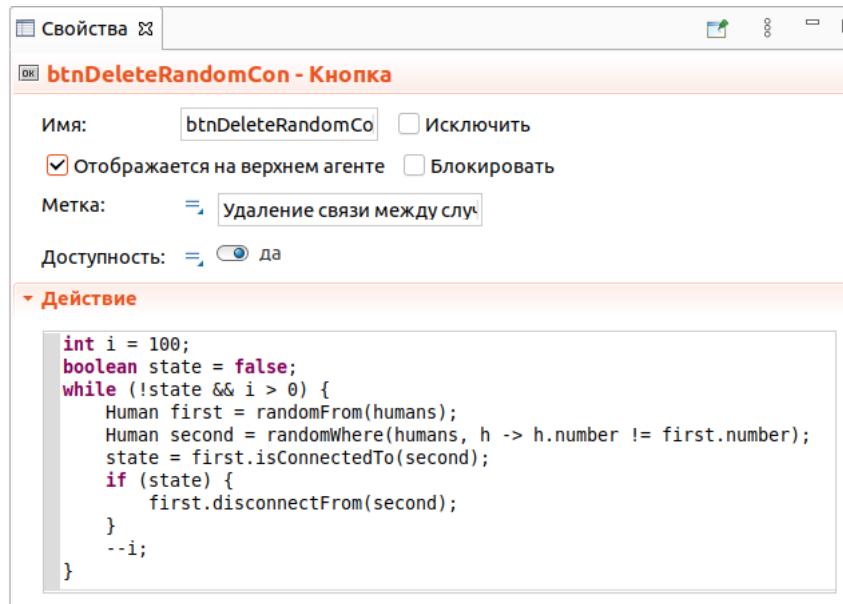


Рис. 134: Удаление связи между случайными агентами

Была реализована кнопка, по которой добавляется связь между двумя заданными вершинами. Вершины задаются через ComboBox. Если связь вершин является недопустимой, то ничего не происходит. (Рисунок 135)

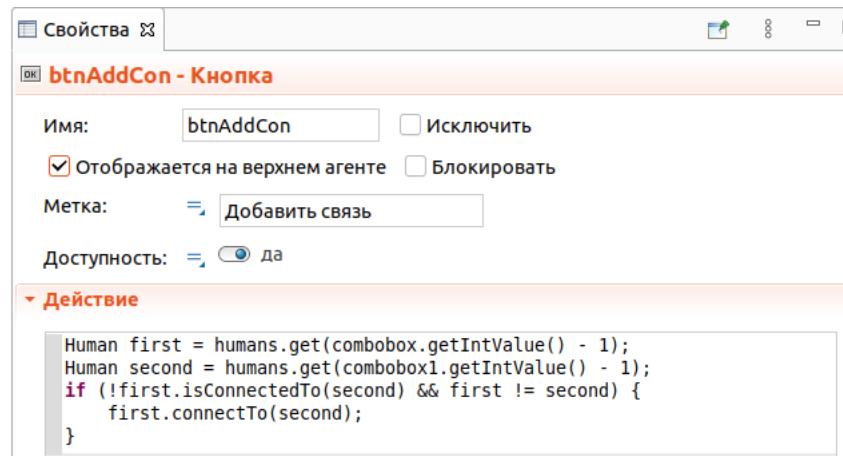


Рис. 135: Добавление связи между заданными вершинами

По аналогии с добавлением связи между двумя заданными вершинами, была реализована кнопка удаление связи между двумя заданными вершинами. (Рисунок 136)

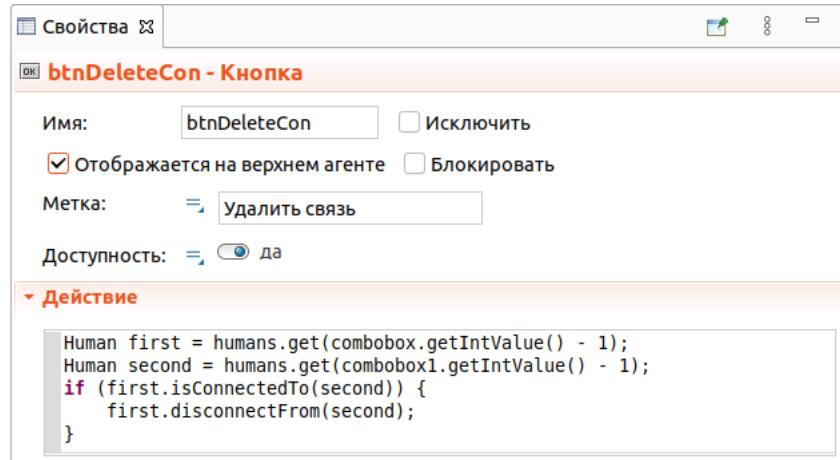


Рис. 136: Удаление связи между заданными вершинами

Далее в событии реализовано обновление агента с максимальным числом связей, он красился в красный, а всего его соседи красились в зелёный, остальные агенты в белый. (Рисунок 137)

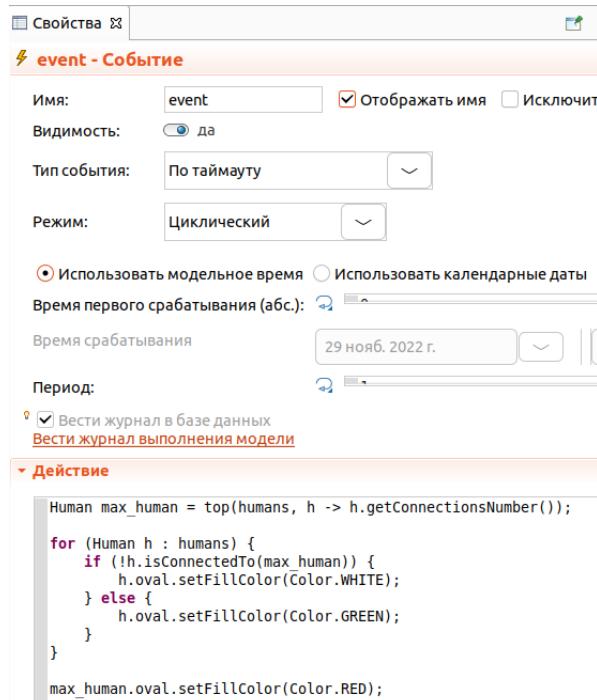


Рис. 137: Реализация события

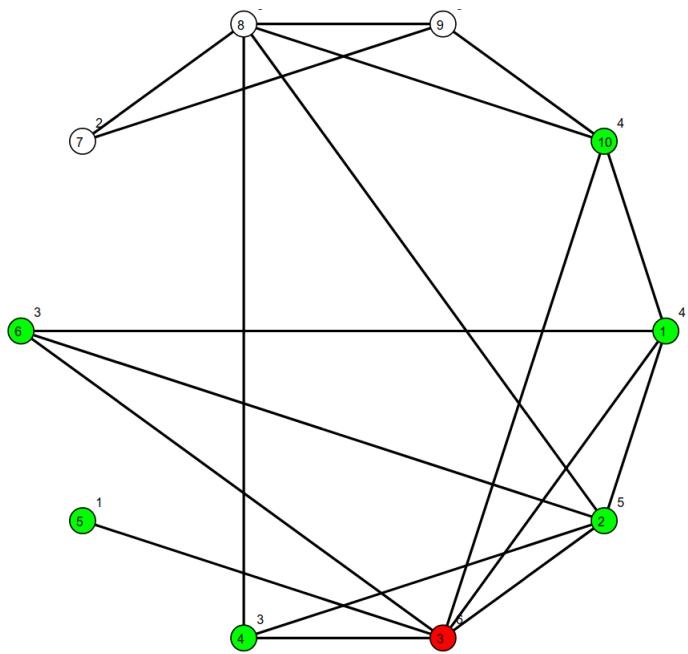


Рис. 138: Визуализация работы модели

Таким образом, в ходе реализации модели были разобраны основные инструменты для работы со связями агентов.

Работа с ГИС-картами

Доставка с покупками