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»

Направление: 01.03.02
Обучающийся: Бронников Егор Игоревич
Группа: ПМ-1901

Санкт-Петербург
2022

Содержание

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 $F^{-1}(x)$.

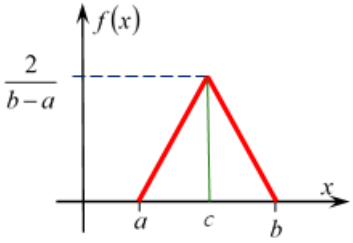

Рис. 1: График плотности треугольного закона

Решение:
Плотность распределения $f(x)$
Уравнение по двум точкам:

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

1) Рассмотрим случай $x \in [a,c]$:

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

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

2) Рассмотрим случай $x \in [c,b]$:

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

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

Плотность треугольного распределения:

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

Функция распределения $F(x)$

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

1) Рассмотрим случай $x \in [a,c]$:

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

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 $F_2(x)$.

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

$$F_2(x) = \frac{c-a}{b-a} + \int_c^x \frac{2(b-t)}{(b-c)(b-a)}\,dt = \frac{c-a}{b-a} + \frac{t(2b-t)}{(b-c)(b-a)}\,\Bigg|_c^x =$$

$$= \frac{c-a}{b-a} + \frac{(2b-c-x)(x-c)}{(b-a)(b-c)} = 1 - \frac{(x-b)^2}{(b-a)(b-c)}$$

Функция распределения треугольного закона:

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

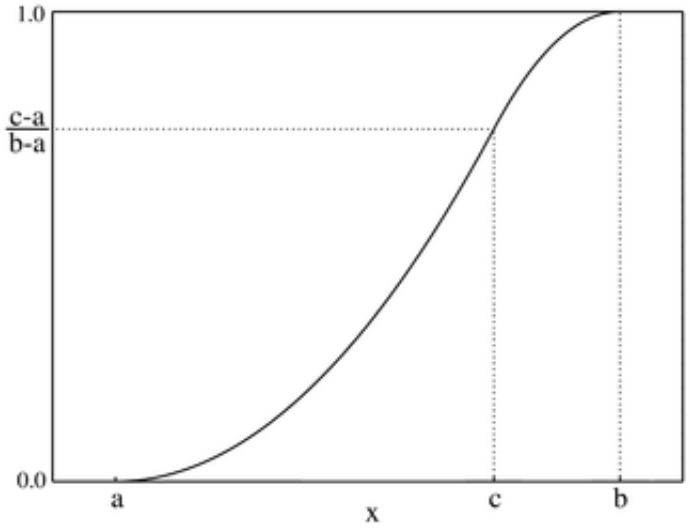

Рис. 2: График функции распределения треугольного закона

Обратная функция $F^{-1}(x)$

Пусть $y = F(x)$.
У функции распределения есть ограничение: $0 < y < 1$, которое стоит учитывать.

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

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

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

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

Обратная функция $F^{-1}(x)$:

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

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)

```python
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()
```

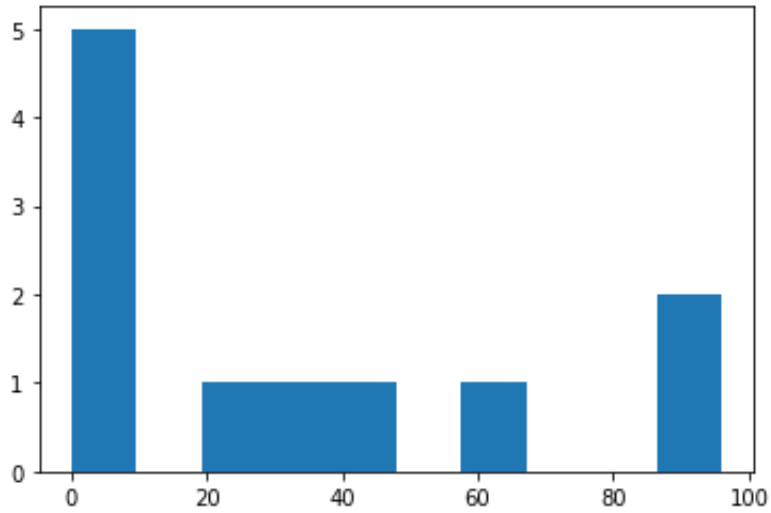

Рис. 4: Результаты генерации случайных чисел методом серединных квадратов

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-$value > 0.05$, 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:

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

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()
```

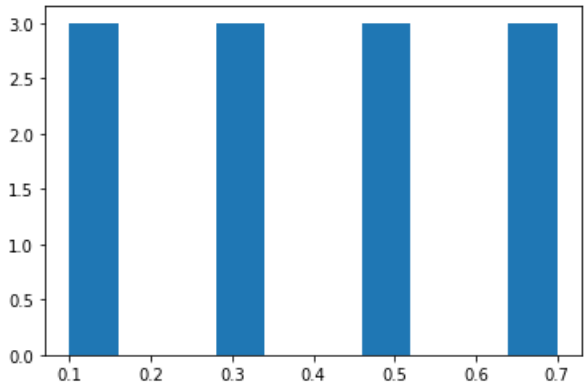

Рис. 7: Результаты генерации случайных чисел линейным конгруэнтным счётчиком на интервале [0,1)

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=0.30000000000000004, 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)
```

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

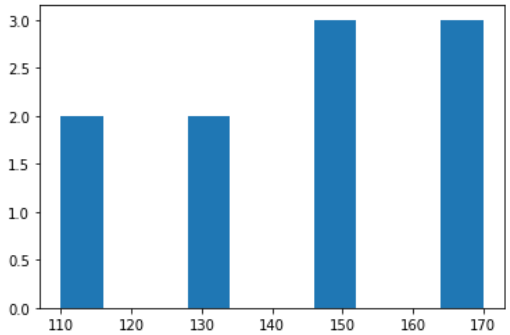

Рис. 10: Результаты генерации случайных чисел линейным конгруэнтным счётчиком на интервале [a,b)

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 $p\text{-}value > 0.05$, 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()
```

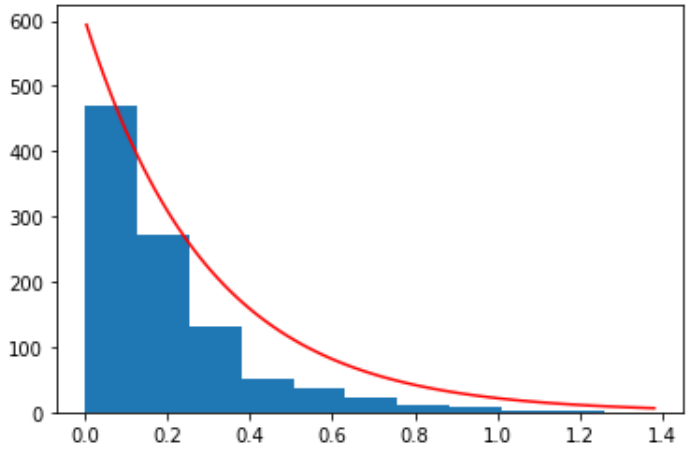

Рис. 13: Результаты генерации случайных чисел методом обратной функции для экспоненциального закона

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 $p\text{-}value > 0.05$, 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()
```

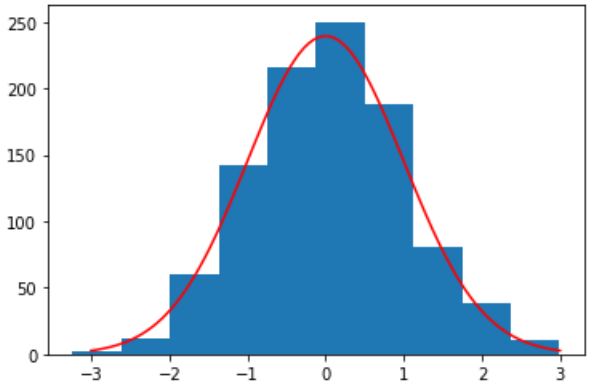

Рис. 18: Результаты применения преобразования Бокса-Мюллера

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()
```

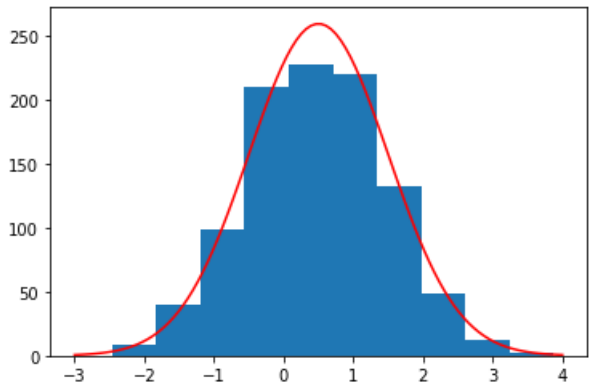

Рис. 21: Результаты применения центральной предельной теоремы

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:

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

Функция распределения треугольного закона:

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

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

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

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:

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

```python
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

```python
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()
```

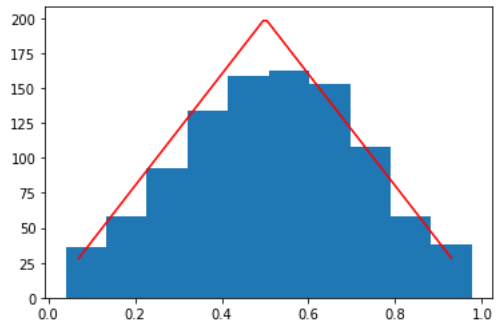

Рис. 24: Результаты генерации случайных чисел методом обратной функции для треугольного закона

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
```

```
0.041666666666666664
```

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 стоить 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()
```

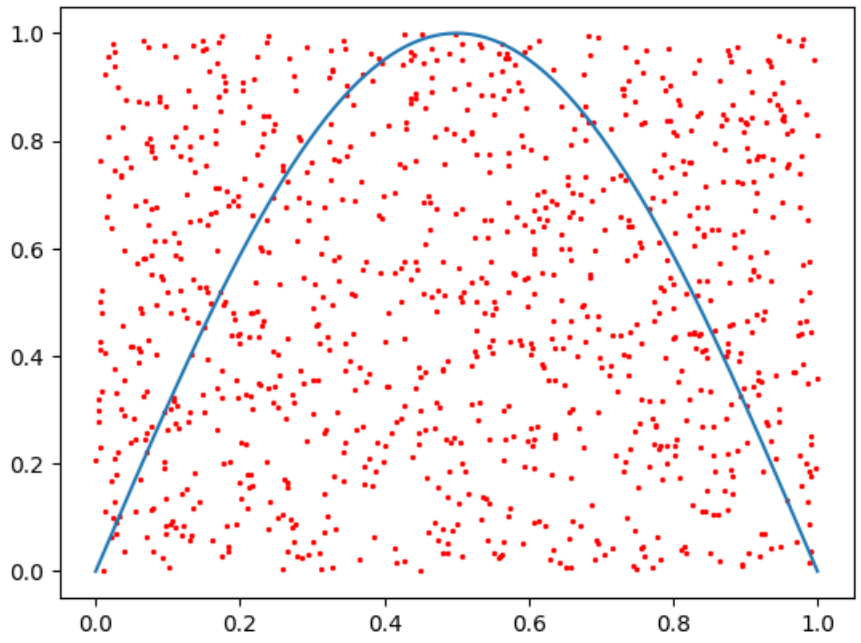

Рис. 28: Пример применения метода Монте-Карло

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)

```python
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)

```python
points = [i for i in range(1, 10000, 50)]
res = np.array([monte_carlo(p) for p in points])
accuracy = 2/np.pi - res
```

```python
plt.plot(accuracy)
plt.plot()
```

```
[]
```

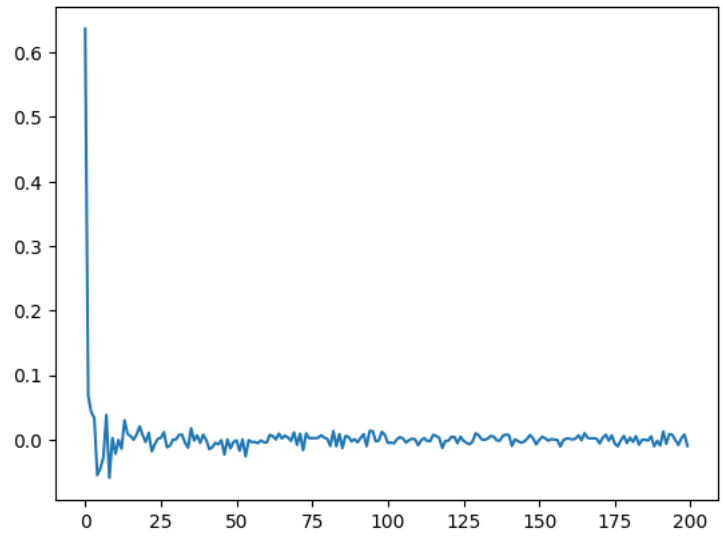

Рис. 30: Зависимость точности результатов от количества точек

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 стоить 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)

```
points = [i for i in range(1, 100000, 50)]
res = np.array([coin(p) for p in points])
accuracy = 0.5 - res
```

```
plt.plot(accuracy)
plt.plot()
```

```
[]
```

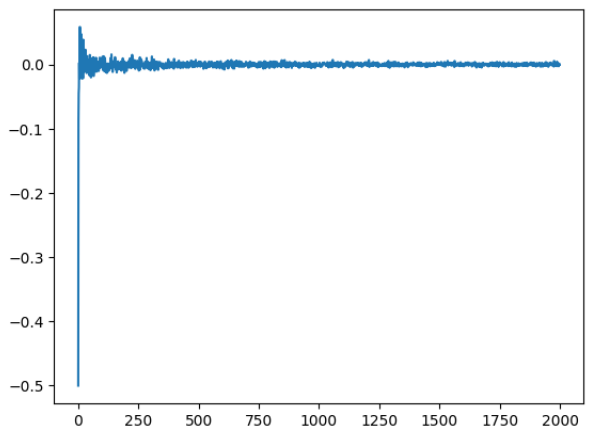

Рис. 32: Зависимость точности результатов от количества точек

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.

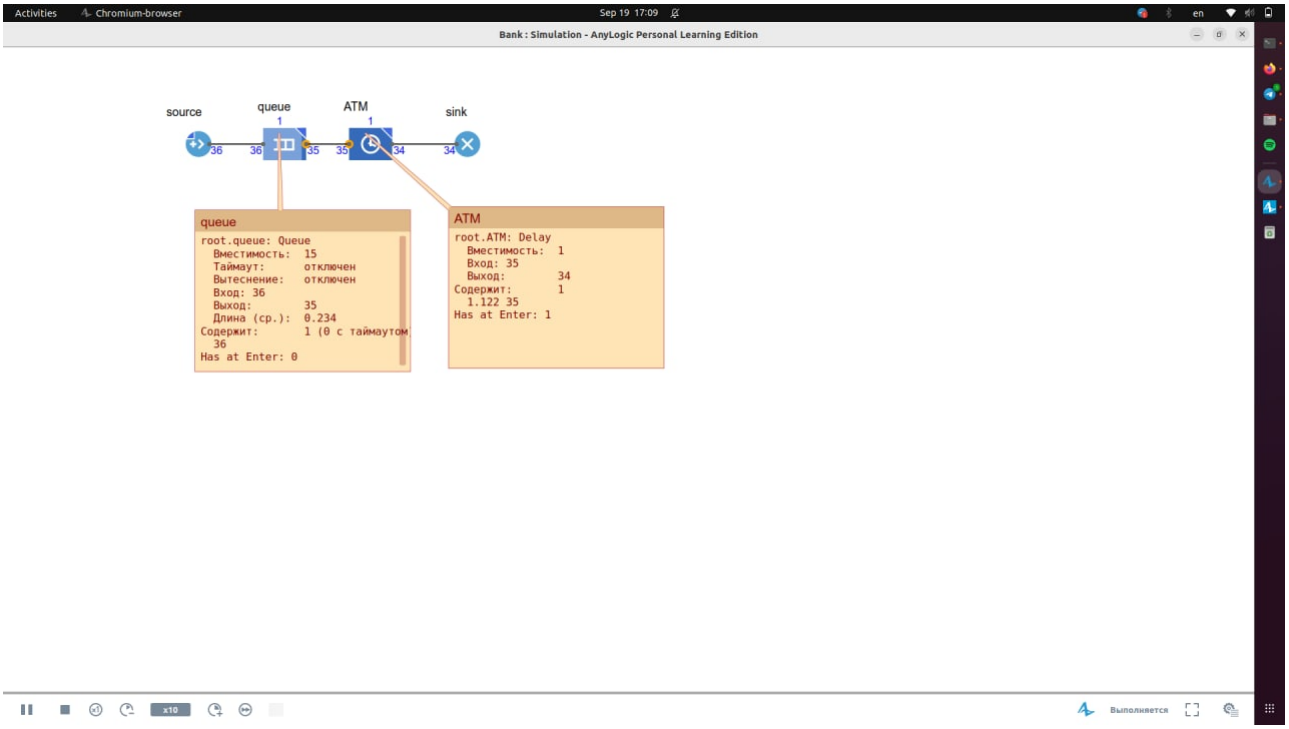

Рис. 33: Схема модели простейшей СМО

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.

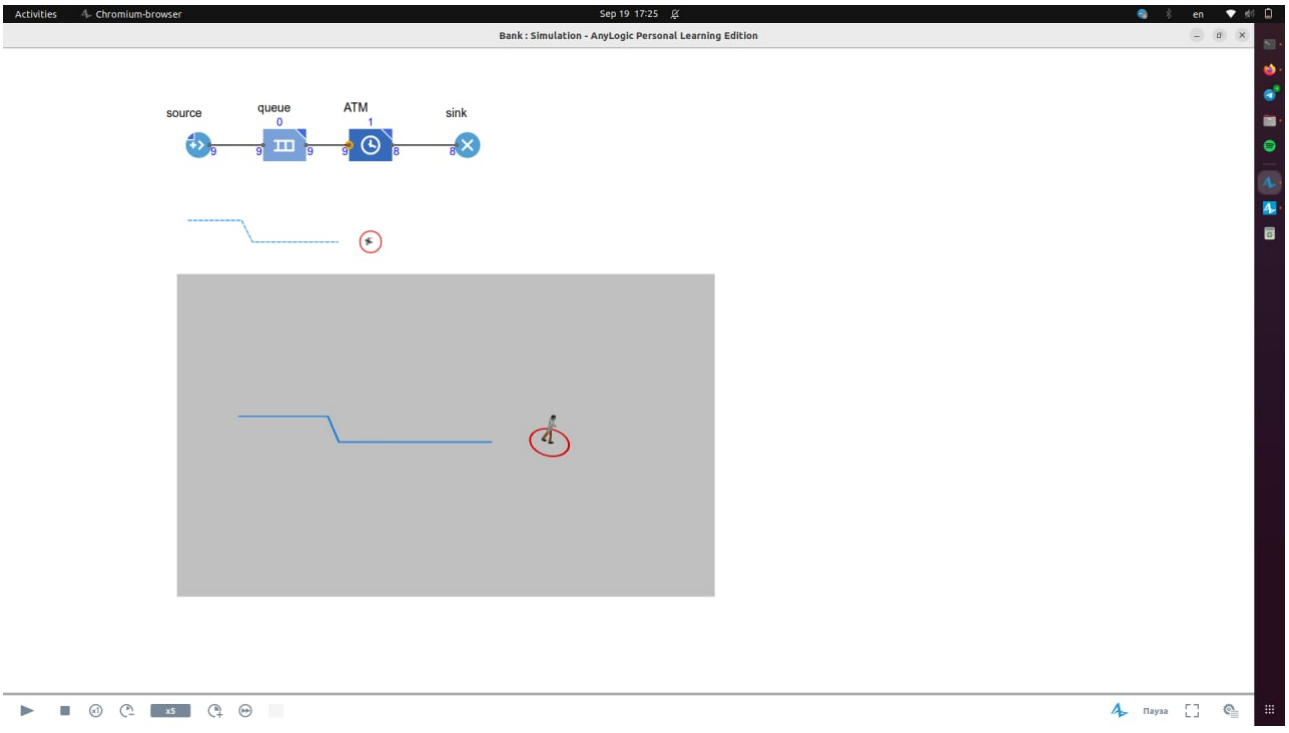

Рис. 34: Анимация модели в среде AnyLogic

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 прибытья 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.

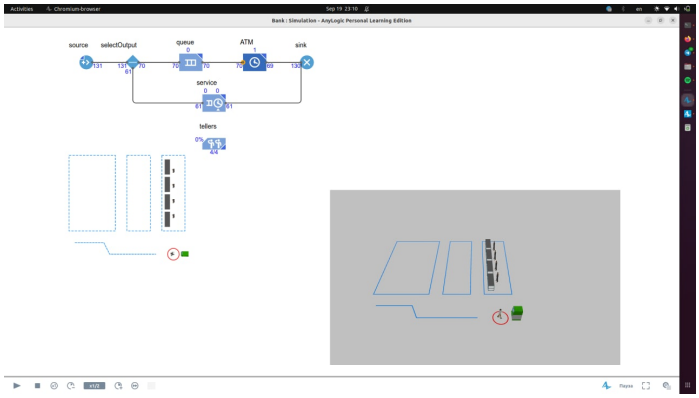

Рис. 35: Усложнение модели СМО в среде AnyLogic

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:

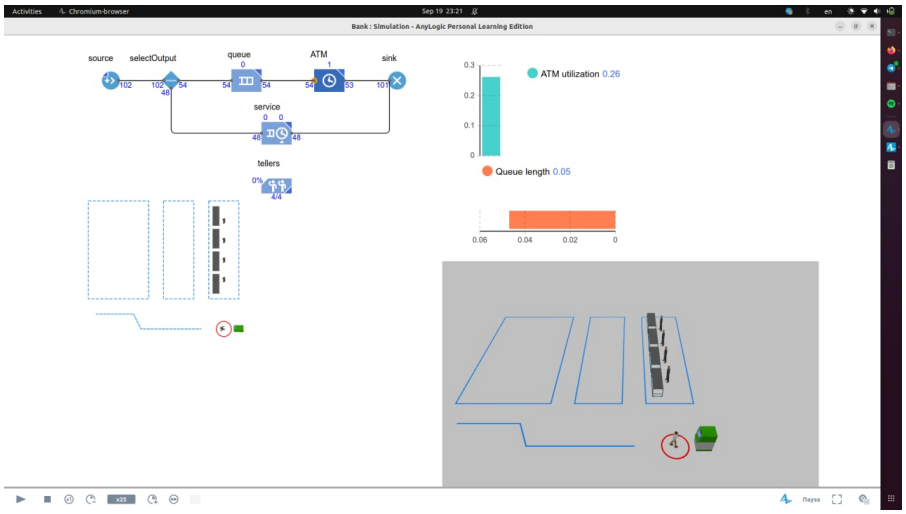

Рис. 36: Добавление статистики среднего времени занятости банкомата и средней длины очереди

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

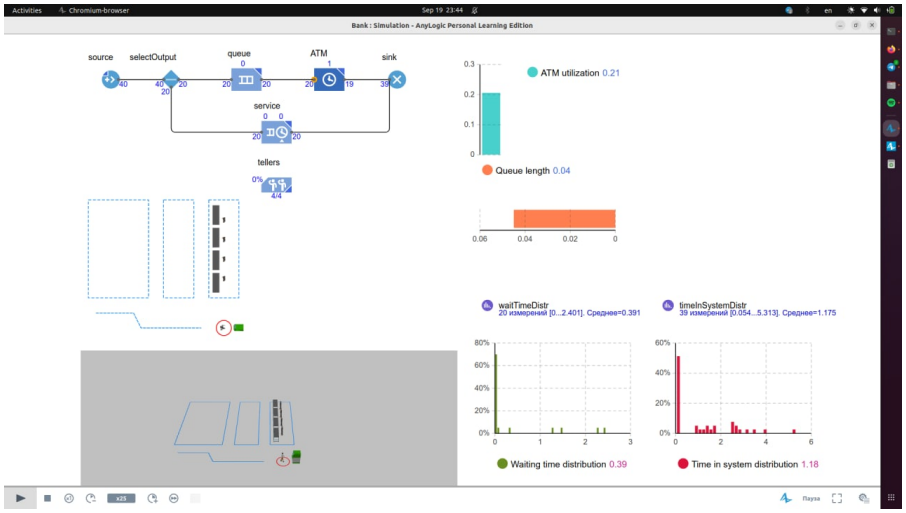

Рис. 37: Добавление статистики времени ожидания и времени обслуживания в системе

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 подсчёт времени требования в системе

Был реализован метод *get_queue*, который для каждого требования сопоставляет количество требований, который находятся 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[:requireme
    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()    # ср
    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
    arriving_per_unit_of_time = number_of_requirements / self.requirements[-1]        # количество требований

    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