Дудниченко Максим (Э 1715)

Отчет по дисциплине "Имитационное моделирование"

В [1]:

```
from IPython.display import Image
```

Генераторы случайных чисел

В [2]:

```
import numpy as np
import pandas as pd
from math import log, sin, cos, sqrt
import plotly.express as px
```

Метод серединных квадратов

В [3]:

```
# Вспомогательная функция
# Убирает последние offset цифр,
# затем берет последние length цифр

# Использует остаток от деления и целочисленное деление,
# поскольку эти операции быстрее, чем перевод в строку
def cut(x, offset, length):
    x = x // 10**offset
    x = x % 10**length
    return x
```

В [4]:

```
# 123456789
# Убираем 2 последние -> 1234567
# Берем 4 последние -> 4567
cut(123456789, 2, 4)
```

Out[4]:

```
4567
```

В [5]:

```
# n_iterations: число итераций
# n_digits: число цифр в x0
def middle_square(x, n_iterations, n_digits=6):
    for i in range(n_iterations):
        x = x**2
        x = cut(x, n_digits//2, n_digits)
    return x
```

В [6]:

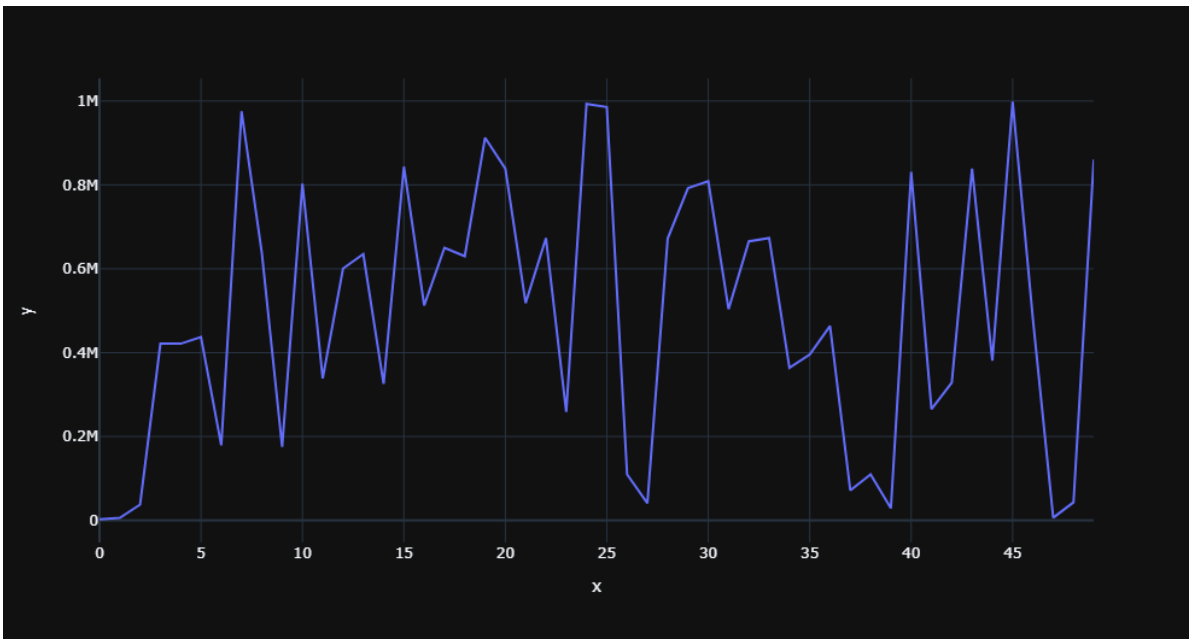
```
for i in range(10):
    print(middle_square(2478, i, n_digits=4))
```

```
2478
1404
9712
3229
4264
1816
2978
8684
4118
9579
```

В [7]:

```
# Число в зависимости от итерации
Image('plots/newplot.png')
```

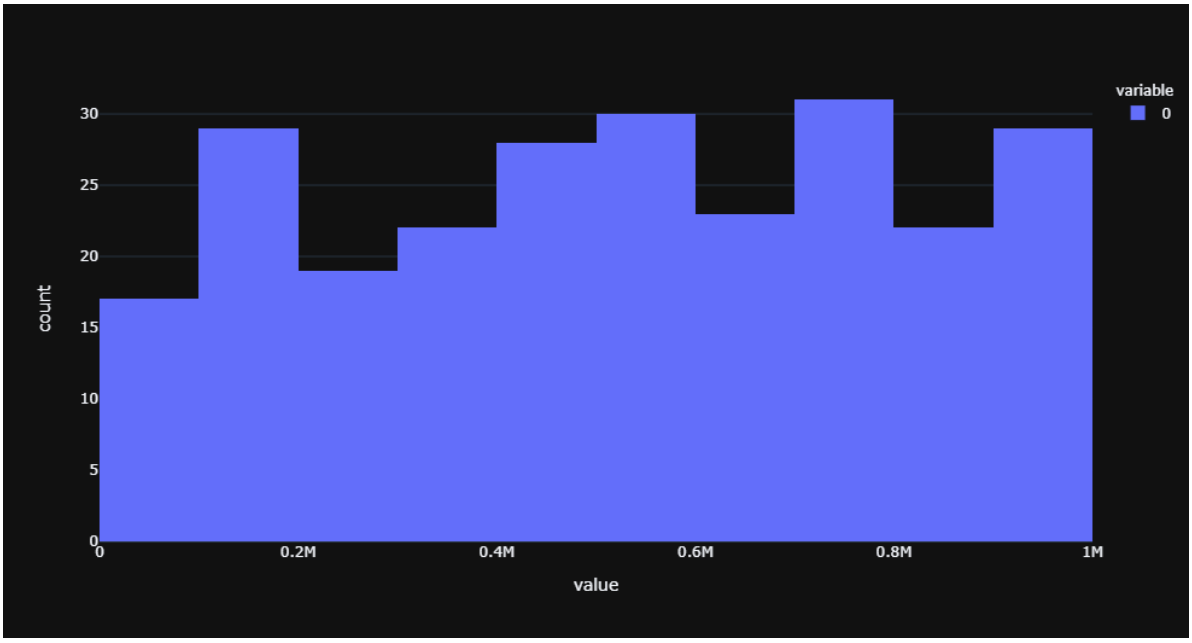
Out[7]:



In [8]:

```
# Гистограмма
Image('plots/newplot (1).png')
```

Out[8]:



Метод серединных произведений

In [9]:

```
def middle_product(x0, x1, n_iterations=10):
    for i in range(n_iterations):
        # Для метода нужно хранить предыдущее число
        # Обновляем два числа одновременно
        x0, x1 = x1, cut(x0*x1, 2, 4)
    return x0, x1
```

B [10]:

```
for i in range(10):
    print(middle_product(4866,5843, i))
```

```
(4866, 5843)
(5843, 4320)
(4320, 2417)
(2417, 4414)
(4414, 6686)
(6686, 5120)
(5120, 2323)
(2323, 8937)
(8937, 7606)
(7606, 9748)
```

Линейный метод

B [11]:

```
def pseudo_random_linear(x, a, c, m, n_iterations):
    result = []
    for i in range(n_iterations):
        result.append(x)
        x = (a*x + c) % m
    return result
```

B [12]:

```
sequence = pseudo_random_linear(x=7, a=7, c=7, m=10, n_iterations=12)
sequence
```

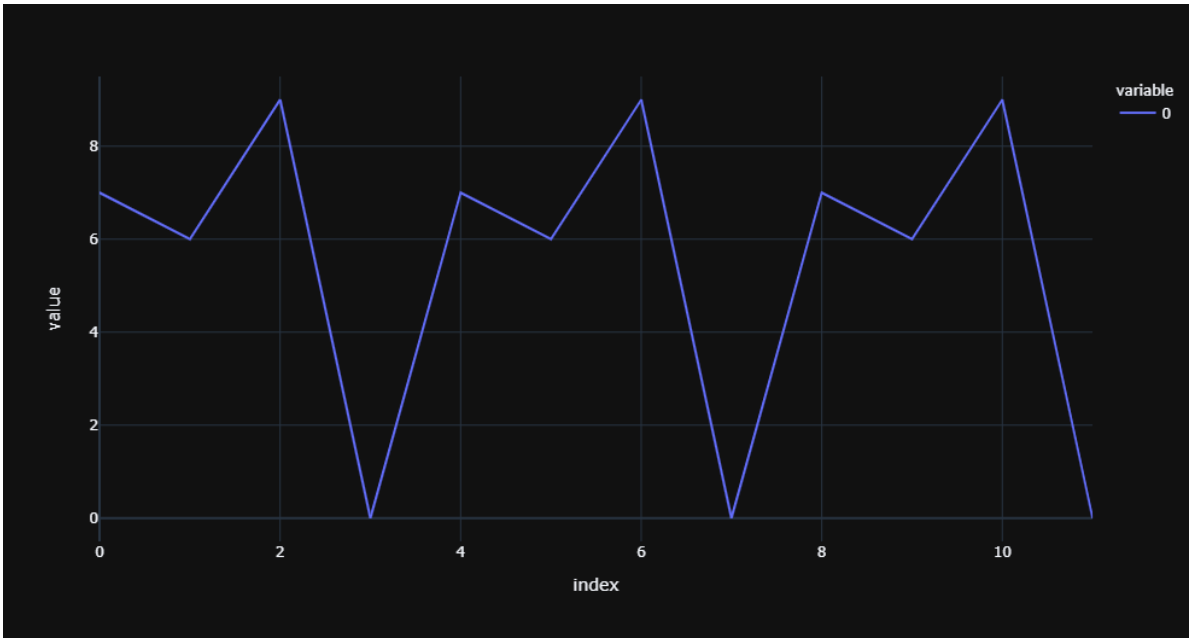
Out[12]:

```
[7, 6, 9, 0, 7, 6, 9, 0, 7, 6, 9, 0]
```

В [13]:

```
# Генерируемая последовательность
# Периодичность = 4
Image('plots/newplot (2).png')
```

Out[13]:



Экспоненциальное распределение

В [14]:

```
# Общая функция для метода обратной функции
# inverse_function: обратная функция накопленной вероятности (CDF)
# random_number: случайное число [0, 1]
# distribution_parameters: словарь параметров распределения
def random_inverse_method(inverse_function, random_number, **distribution_parameters):
    return inverse_function(random_number, **distribution_parameters)
```

Вывод обратной функции для экспоненциального рапределения

$$y = 1 - e^{-ax}$$

$$e^{-ax} = 1 - y$$

$$-ax = ln(1 - y)$$

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

В [15]:

```
def inverse_exp(y, lambda_):
    return log(1 - y) / -lambda_
```

В [16]:

```
result = [random_inverse_method(inverse_exp, np.random.rand(), lambda_=1)
          for i in range(1000)]
```

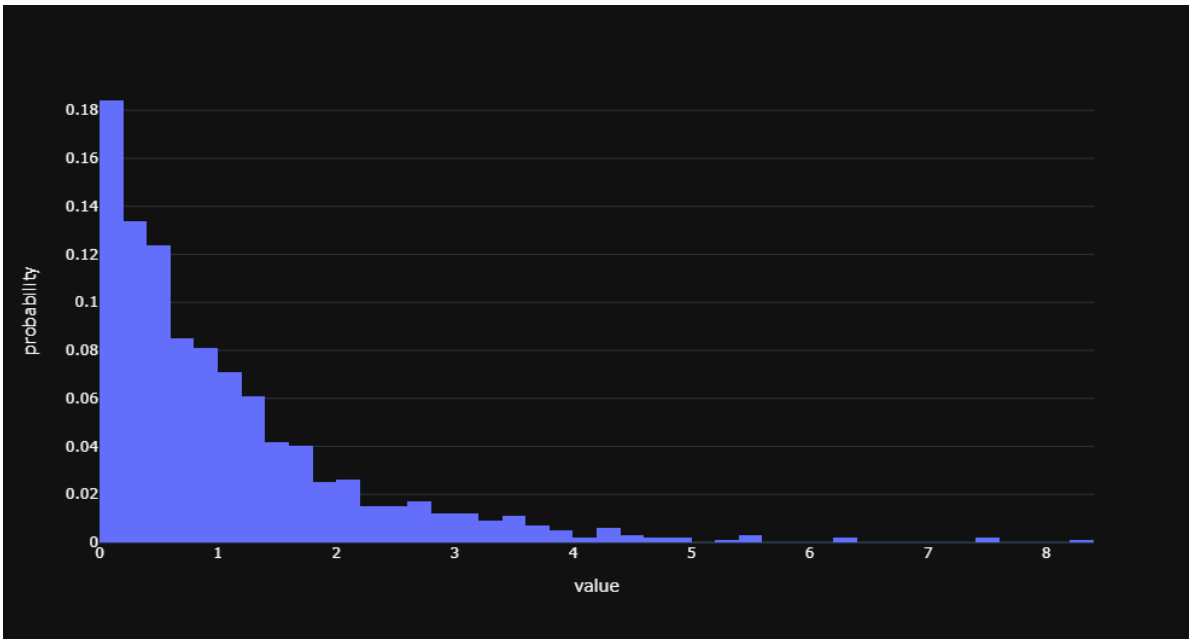
In [17]:

```
fig = px.histogram(result, histnorm='probability', template='plotly_dark')
fig['layout']['showlegend'] = False
# fig.show()
```

In [18]:

```
# Гистограмма экспоненциального распределения
Image('plots/newplot (3).png')
```

Out[18]:



Треугольное распределение

Вывод обратной функции для экспоненциального рапределения

Для первой части (x <= c)

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

$$(x-a)^2 = y(b-a)(c-a)$$

$$x - a = \sqrt{y(b-a)(c-a)}$$

$$x = a + \sqrt{y(b-a)(c-a)}$$

Для второй части (x >= c)

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

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

$$(b-x)^2 = (b-a)(b-c)(1-y)$$

$$b - x = \sqrt{(b-a)(b-c)(1-y)}$$

$$x = b - \sqrt{(b-a)(b-c)(1-y)}$$

B [19]:

```
def inverse_triangular(y, a, b, c):
    if y <= (c-a)/(b-a):
        return a + sqrt(y*(b-a)*(c-a))
    else:
        return b - sqrt((b-a)*(b-c)*(1-y))
```

B [20]:

```
result = [random_inverse_method(inverse_triangular, np.random.rand(), a=1, b=5, c=2)
          for i in range(100000)]
```

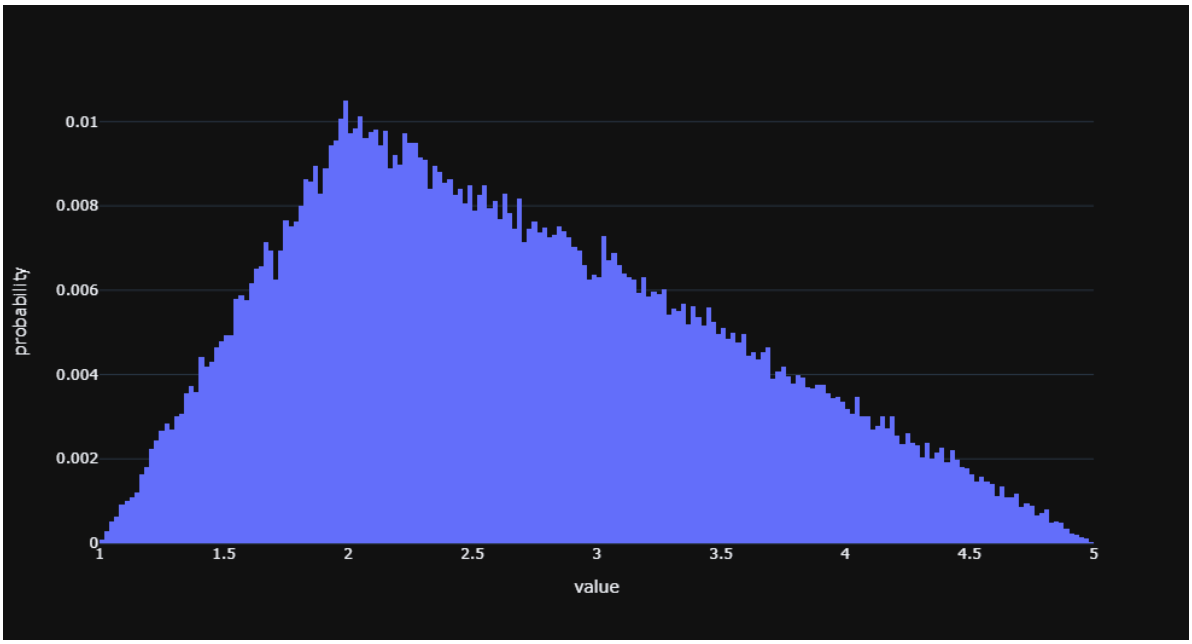
B [21]:

```
fig = px.histogram(result, histnorm='probability', template='plotly_dark')
fig['layout']['showlegend'] = False
# fig.show()
```

B [22]:

```
# Гистограмма треугольного распределения
Image('plots/newplot (4).png')
```

Out[22]:



Нормальное распределение

Метод обратной функции не подходит

Метод Бокса-Мюллера

B [23]:

```
def method1():
    r = np.random.rand()
    phi = np.random.rand()

    z0 = cos(2*np.pi*phi) * sqrt(-2*log(r))
    z1 = sin(2*np.pi*phi) * sqrt(-2*log(r))

    return z0, z1
```

B [24]:

```
method1()
```

Out[24]:

```
(-0.6081429878251547, 0.3123770666253117)
```

B [25]:

```
pairs = [method1() for i in range(10000)]
x0 = [x[0] for x in pairs]
x1 = [x[1] for x in pairs]
```

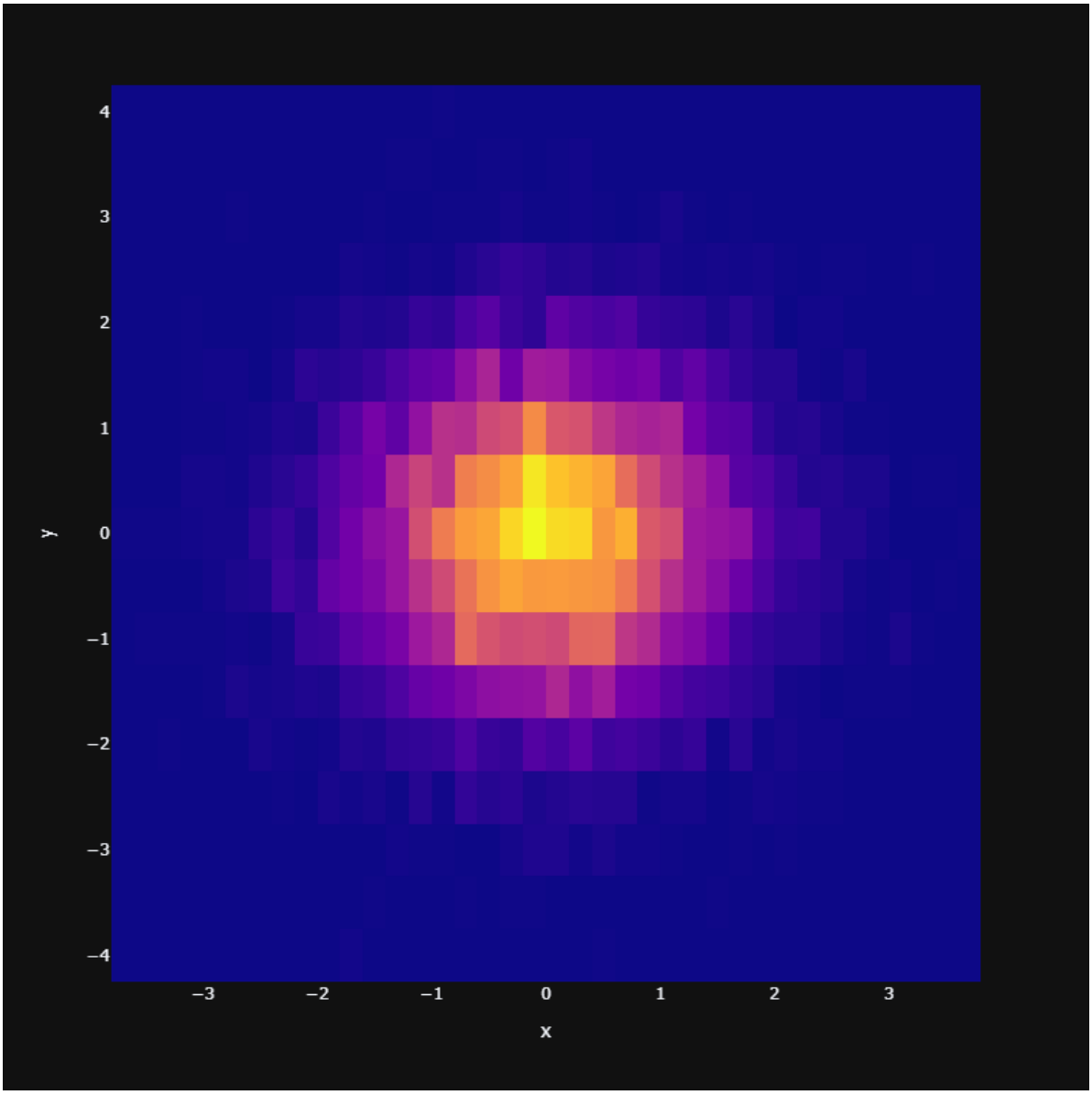
B [26]:

```
fig = px.density_heatmap(x=x0, y=x1, template='plotly_dark')
fig['layout']['height'] = 800
fig['layout']['width'] = 800
fig['layout']['coloraxis']['showscale'] = False
# fig.show()
```

B [27]:

```
1 # 2D Гистограмма
2 Image('plots/newplot (5).png')
```

Out[27]:



Метод ЦПТ

В [28]:

```
def clt_generator(n_layers):
    # Генерируем матрицу случайных чисел
    clt = np.random.uniform(0, 1, (n_layers, 10000))
    # Суммируем по столбцам
    clt = np.sum(clt, axis=0)
    # Нормализация
    clt = sqrt(12/n_layers) * (clt - n_layers/2)
    return clt
```

В [29]:

```
def plot_histogram(data):
    fig = px.histogram(data, histnorm='probability', template='plotly_dark')
    fig['layout']['showlegend'] = False
    fig.show()
```

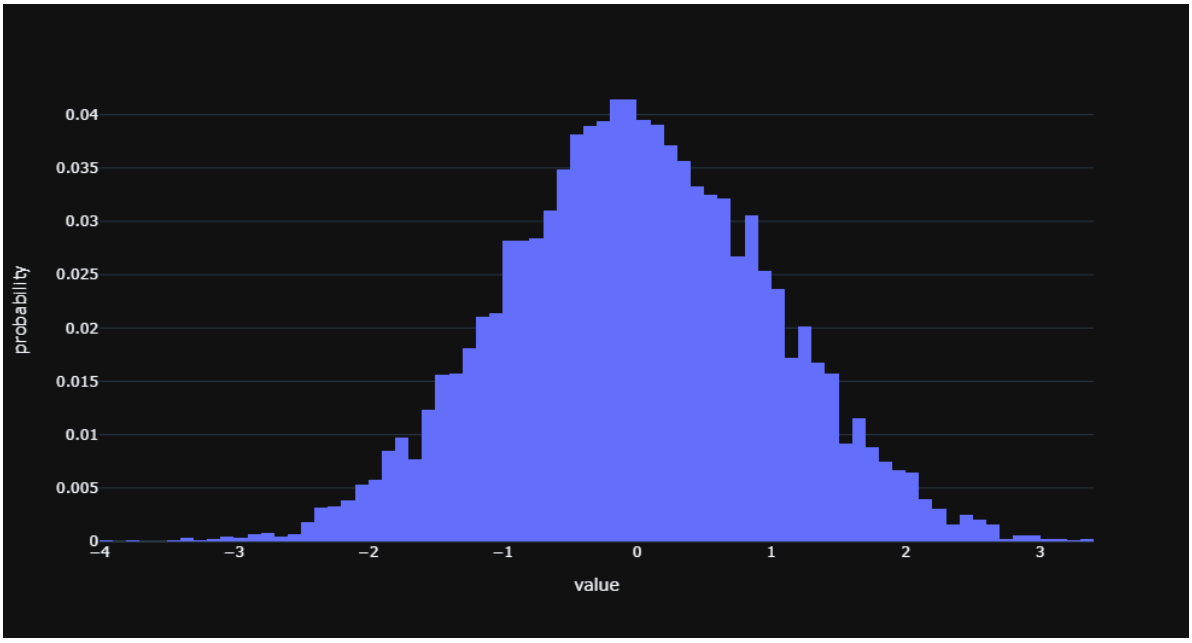
В [30]:

```
# for i in range(1, 13):
#     plot_histogram(clt_generator(i))
```

В [31]:

```
# Гистограмма (12 слагаемых)
Image('plots/newplot (9).png')
```

Out[31]:



Интеграл методом Монте-Карло

$\int_0^1 sin(\pi x)\,dx$

В [32]:

```
def monte_carlo_integrate():
    inside = 0
    total = 0
    values = []
    for i in range(1_000):
        x = np.random.rand()
        y = np.random.rand()
        # Проверяем, попадает ли в область
        if y <= np.sin(np.pi*x):
            inside += 1
        total += 1
        values.append(inside/total)
    return values
```

B [33]:

```
def plot_convergence(simulation_function, n_simulations):
    y = [simulation_function() for i in range(n_simulations)]
    x = [i[-1] for i in y]

    fig = px.line(y=y, template='plotly_dark')
    fig.update_traces(line_color='white')
    fig['data'][-1]['line']['color'] = 'red'
    fig['layout']['showlegend'] = False
    fig.show()

    fig = px.histogram(x=x, histnorm='probability', template='plotly_dark')
    fig['layout']['showlegend'] = False
    fig.show()
```

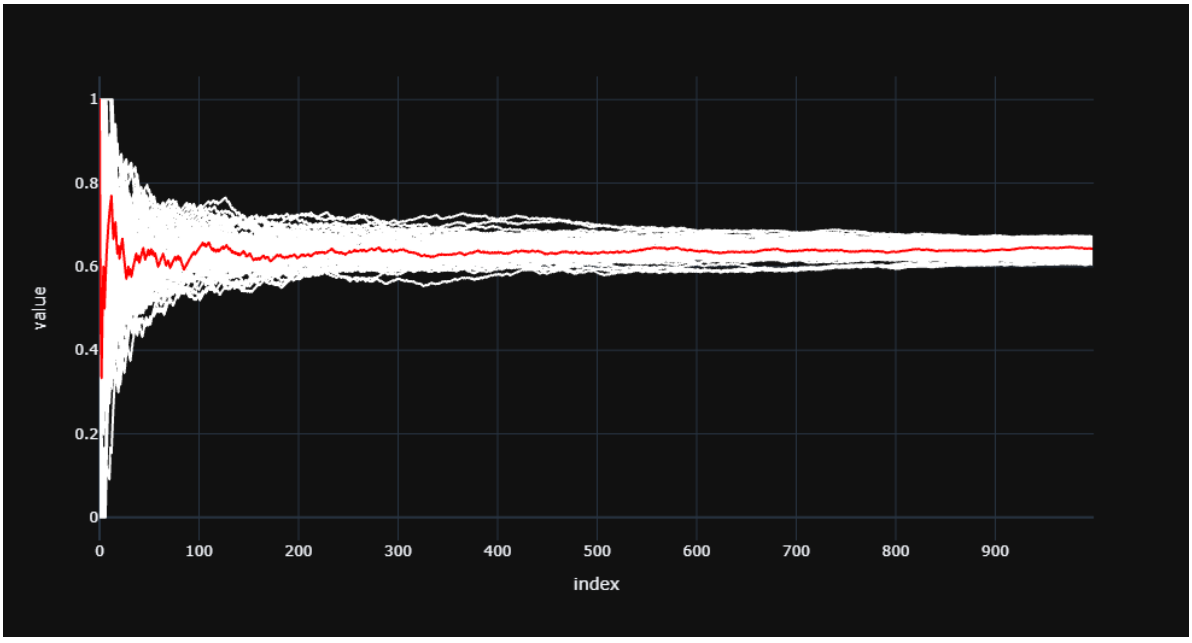
B [34]:

```
# plot_convergence(monte_carlo_integrate, 100)
```

B [35]:

```
# Сходимость в зависимости от итерации
Image('plots/newplot (10).png')
```

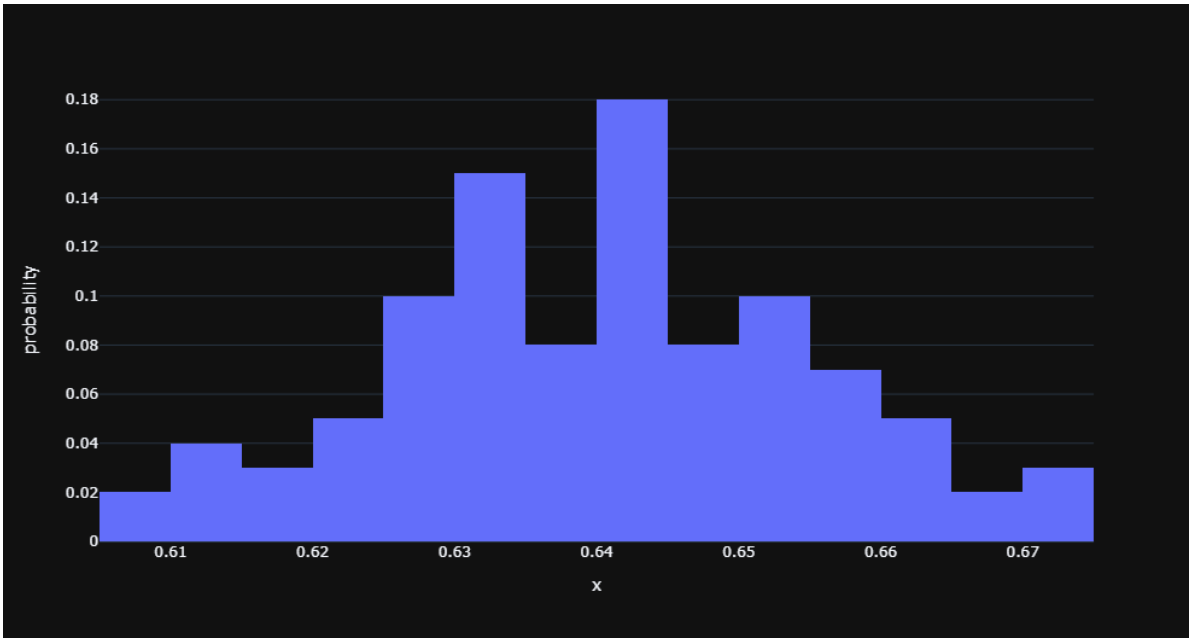
Out[35]:



In [36]:

```
# Гистограмма для последней итерации
Image('plots/newplot (11).png')
```

Out[36]:



Моделирование многоканальной СМО

In [37]:

```
import pandas as pd
import plotly.express as px
from numpy.random import exponential, choice
```

В [38]:

```python
# Класс для обработки событий
class EventTimeline:

    def __init__(self):
        # Список событий в формате: [function, arguments]
        # function: функция, которая меняет состояние модели
        # arguments: аргументы для function (время возникновения события, номер кассы,
        self.timeline = []

    def add_event(self, event):
        self.timeline.append(event)

    def get_next_event(self):
        # Сортируем события по времени наступления
        self.timeline.sort(key=lambda event: event[1]['time'])
        # Возвращаем самое раннее событие, удаляем его из списка
        return self.timeline.pop(0)
```

В [39]:

```
class Model:


    def __init__(self, n_units, **kwargs):
        self.in_intensity = kwargs['in_intensity']
        self.out_intensity = kwargs['out_intensity']
        self.max_queue = kwargs['max_queue']
        self.max_wait_time = kwargs['max_wait_time']

        self.event_timeline = EventTimeline()
        # Счетчик агентов
        self.client_id = 0
        # Начинаем модель с прибытия клиента
        # TODO: Можно изменить
        self.event_timeline.add_event(
            [self.customer_arrives, {'time': 0}]
        )
        # Занятость узлов обслуживания
        self.is_free = [True for _ in range(n_units)]
        # Очередь: список id клиентов
        self.queue = []
        # Сбор статистики (занятость узлов)
        self.table = pd.DataFrame(columns=[j for j in range(n_units)])


    # Прибытие клиента
    def customer_arrives(self, time):
        # Обновляем счетчик клиентов
        self.client_id += 1
        # Если есть свободные узлы
        if any(self.is_free):
            # Обслуживаем клиента
            self.serve_customer(time)

        # Если очередь меньше заданной
        elif len(self.queue) <= self.max_queue:
            # Добавляем клиента в очередь
            self.queue.append(self.client_id)
            # Добавляем уход из очереди в список событий
            self.event_timeline.add_event(
                [self.customer_leaves_queue,
                 {
                     'time': time + self.max_wait_time,
                     'id_': self.client_id
                 }]
            )

        # Следующий клиент прибудет через случайный промежуток времени
        self.event_timeline.add_event(
            [self.customer_arrives, {'time': time + exponential(self.in_intensity)}]
        )

    # Обслуживание клиента
    def serve_customer(self, time):
        # Находим свободный элемент
        free_index = [i for i, isfree in enumerate(self.is_free) if isfree]
        index = choice(free_index)
        # Узел становится занят
        self.is_free[index] = False
```

```
        # Клиент уходит через случайный промежуток времени
        self.event_timeline.add_event(
            [self.customer_leaves,
             {
                 'time': time + exponential(self.out_intensity),
                 'index': index
             }]
        )

    # Клиент уходит после обслуживания
    def customer_leaves(self, time, index):
        # Освобождаем узел
        self.is_free[index] = True
        # Если есть очередь:
        if self.queue:
            # Убираем из очереди
            self.queue.pop(0)
            # Обслуживаем клиента
            self.serve_customer(time)

    # Клиент уходит из очереди
    def customer_leaves_queue(self, time, id_):
        # Если клиент в очереди
        # Проверка нужна на случай, если обслуживание
        # началось раньше, чем кончилось терпение
        if id_ in self.queue:
            index = self.queue.index(id_)
            self.queue.pop(index)

    # Симуляция
    def simulation(self, n_iterations):
        for i in range(n_iterations):
            # Получаем следующее событие
            current_function, current_arguments = self.event_timeline.get_next_event()
            # Собираем статистику
            self.table.loc[current_arguments['time'], :] = self.is_free
            # Запускаем событие
            current_function(**current_arguments)
```

В [40]:

```
parameters = dict(
    in_intensity=1,
    out_intensity=5,
    max_queue=20,
    max_wait_time=25
)
model = Model(10, **parameters)
```

В [41]:

```
model.simulation(100)
```

In [42]:

```
# Статистика занятости узлов по времени
model.table.head()
```

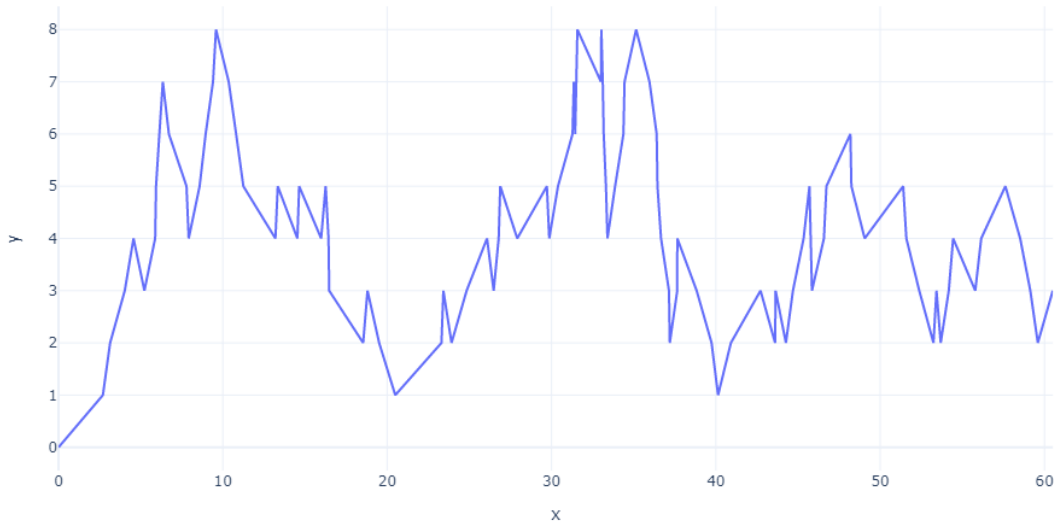
Out[42]:

	0	1	2	3	4	5	6	7	8	9
0.000000	True	True	True	True	True	True	True	True	True	True
1.935808	False	True	True	True	True	True	True	True	True	True
2.383370	False	True	True	True	True	False	True	True	True	True
2.622147	False	True	False	True	True	False	True	True	True	True
2.759364	False	True	False	True	True	False	True	True	False	True

In [43]:

```
# Общее число занятых узлов
data = (model.table == False).sum(axis=1)
fig = px.line(x=data.index, y=data.values, template='plotly_white')
# fig.show()
Image('plots/newplot (12).png')
```

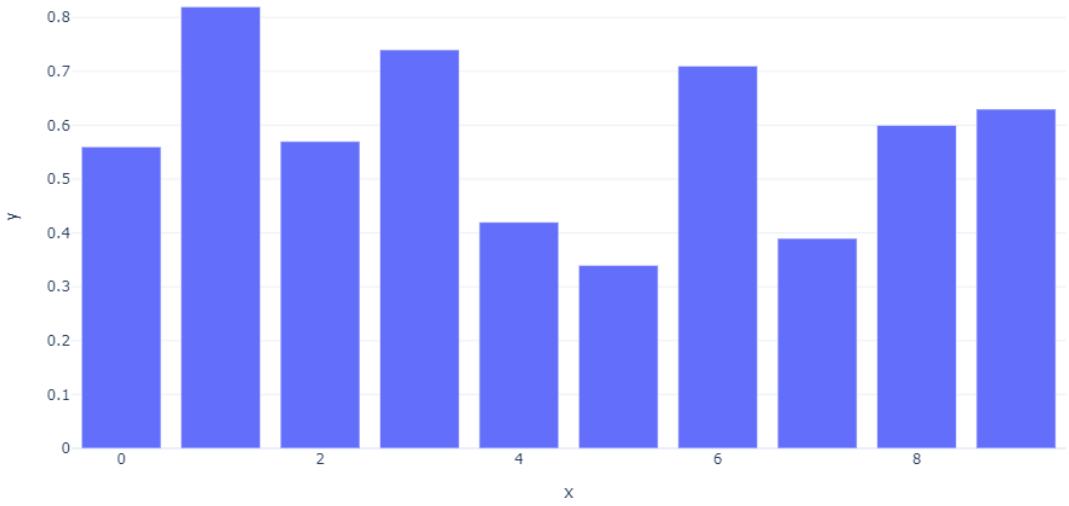
Out[43]:



In [44]:

```
# Средняя занятость по узлам
data = model.table.mean()
fig = px.bar(x=data.index, y=data.values, template='plotly_white')
# fig.show()
Image('plots/newplot (13).png')
```

Out[44]:



Игра "Жизнь" Конвея

In [45]:

```
import numpy as np
from scipy.signal import convolve2d
import plotly.graph_objects as go
import plotly.express as px
```

В [46]:

```
@np.vectorize
# Возвращает новое состояние клетки
def logic(state, n_neighbours):
    if state == 0 and n_neighbours == 3:
        return 1
    elif state == 1 and (n_neighbours == 3 or n_neighbours == 2):
        return 1
    return 0
```

В [47]:

```
# Можно менять для другой меры соседства
filt = np.array([
    [1, 1, 1],
    [1, 0, 1],
    [1, 1, 1]
])
```

В [48]:

```
def game_of_life(board, n_iterations):
    boards = [board]
    for i in range(n_iterations):
        # Считаем число соседей с использованием свертки
        n_neighbours = convolve2d(board, filt, mode='same', boundary='fill', fillvalue=
        board = logic(board, n_neighbours)
        boards.append(board)
    return boards
```

В [49]:

```
N = 250
p = 0.5
n_iterations = 100
```

В [50]:

```
board = np.random.choice([0, 1], [N, N], p=[1-p, p])
heatmaps = game_of_life(board, n_iterations)
```

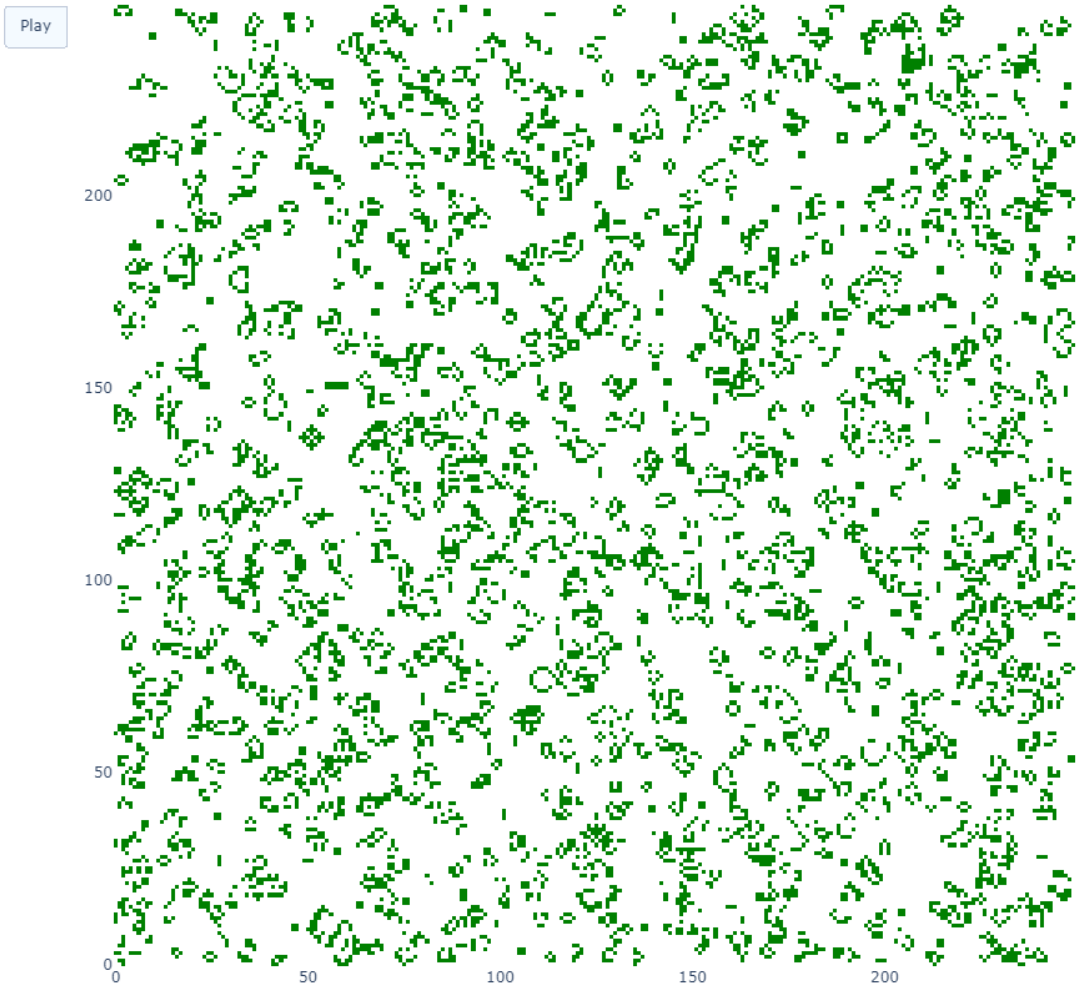
В [51]:

```
# Анимация
fig = go.Figure(data=go.Heatmap(z=heatmaps[0]),
                frames=[go.Frame(data=go.Heatmap(z=heatmaps[i])) for i in range(n_itera

fig.update_layout(
    updatemenus=[
        {'type': "buttons", 'visible': True,
        'buttons': [dict(label="Play", method="animate", args=[None])]}
    ])

fig.update_traces(showscale=False, colorscale=['white', 'green'])
fig.layout.height = 950
fig.layout.width = 950
# fig.show()
```

In [52]:

```
# Поле после нескольких итераций
Image('plots/newplot (14).png')
```

Out[52]:



Модель сегрегации Шеллинга

In [53]:

```
import numpy as np
import pandas as pd
from scipy.signal import convolve2d
import plotly.graph_objects as go
import plotly.express as px
from random import shuffle, choice
```

В [54]:

```
def Shelling_segregation_model(board, n_iterations, preferences, min_happiness):
    history = []
    for _ in range(n_iterations):
        history.append(board.copy())
        board = pd.DataFrame(board)
        filt = np.array([
            [1, 1, 1],
            [1, 0, 1],
            [1, 1, 1]
        ])
        # Определяем счастье каждой клетки
        # Используем операцию свертки
        happiness = sum(
            np.where(board==k, convolve2d(board.replace(preferences[k]), filt,
                                          mode='same', boundary='fill', fillvalue=0), 0
            for k in [1, 2, 3]
        )
        # Определяем, кто переедет
        will_move = ((happiness < min_happiness) & (board != 0))

        rows, columns = np.where(will_move)
        will_move = list(zip(rows, columns))
        shuffle(will_move)

        rows, columns = np.where(board == 0)
        empty = list(zip(rows, columns))
        shuffle(empty)

        # Переезд
        for i, j in will_move:

            agent_type = board.iloc[i, j]
            new_i, new_j = choice(empty)

            empty.append((i, j))
            empty.remove((new_i, new_j))

            board.iloc[new_i, new_j] = agent_type
            board.iloc[i, j] = 0

    return history
```

В [55]:

```
preferences = {
    1: {1: 1, 2: 1, 3:0},
    2: {1: 1, 2: 1, 3:0},
    3: {1: 0, 2: 0, 3:1}
}
min_happiness = 5
board = np.random.choice([0, 1, 2, 3], (50, 50), p=[0.1, 0.3, 0.3, 0.3])
```

В [56]:

```
heatmaps = Shelling_segregation_model(board, 25, preferences, min_happiness)
```

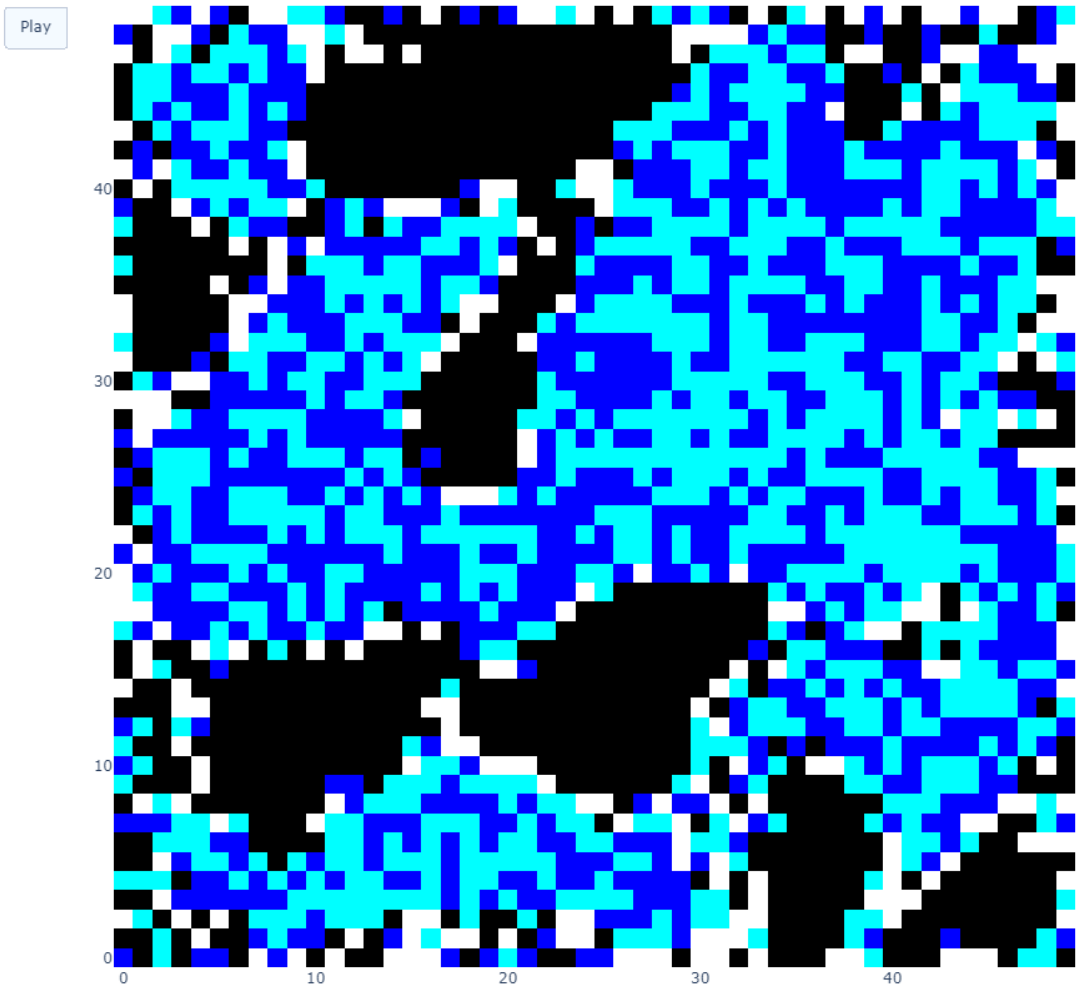
В [57]:

```
# Анимация
fig = go.Figure(data=go.Heatmap(z=heatmaps[0]),
                frames=[go.Frame(data=go.Heatmap(z=heatmaps[i])) for i in range(25)])
fig.update_layout(
    updatemenus=[
        {'type': "buttons", 'visible': True,
        'buttons': [dict(label="Play", method="animate", args=[None])]}
    ])
fig.update_traces(showscale=False, colorscale=['white', 'blue', 'cyan', 'black'])
fig.layout.height = 950
fig.layout.width = 950
# fig.show()
```

В [58]:

```
# Результат модели после нескольких итераций
# Синие и черные разделяются на островки
Image('plots/newplot (15).png')
```

Out[58]:



Модель распространения инноваций

В [59]:

```
import numpy as np
import pandas as pd
import random
import plotly.graph_objects as go
import plotly.express as px
```

В [60]:

```
class Agent:

    def __init__(self):
        self.x = np.random.uniform(0, 100)
        self.y = np.random.uniform(0, 100)

        self.product = 0
        # Чувствительность к рекламе
        self.ad_probability = np.random.uniform(0, 1)
        # Чувствительность к советам друзей
        self.friend_probability = np.random.uniform(0, 1)

    # Получение совета от друга
    def get_friend_offer(self, product):
        if np.random.rand() < self.friend_probability:
            self.product = product

    # Получение рекламы
    def get_ad_offer(self, product):
        if np.random.rand() < self.ad_probability:
            self.product = product
```

В [61]:

```
class Population:

    def __init__(self, n_agents):
        self.population = [Agent() for i in range(n_agents)]
        self.stat = []

    def update(self, n_ads_1=10, n_ads_2=10):

        # Реклама 1 продукта
        for _ in range(n_ads_1):
            # "Центр" рекламы
            center_x = np.random.uniform(0, 100)
            center_y = np.random.uniform(0, 100)
            # Попавшие в область потребители
            for agent in self.population:
                if (agent.x - center_x)**2 + (agent.y - center_y)**2 < 5**2:
                    agent.get_ad_offer(1)

        # Реклама 2 продукта
        for _ in range(n_ads_2):
            center_x = np.random.uniform(0, 100)
            center_y = np.random.uniform(0, 100)

            for agent in self.population:
                if (agent.x - center_x)**2 + (agent.y - center_y)**2 < 5**2:
                    agent.get_ad_offer(2)

        # Рекомендации друзей
        for agent1 in [agent for agent in self.population if agent.product != 0]:

            neighbours = [agent2 for agent2 in self.population if
                          (agent2.x - agent1.x)**2 + (agent2.y - agent1.y)**2 < 5**2]

            for agent2 in neighbours:
                agent2.get_friend_offer(agent1.product)

        # Сбор статистики
        s = [[a.x, a.y, a.product] for a in self.population]
        self.stat.append(pd.DataFrame(s))
```

В [62]:

```
p = Population(250)
```

В [63]:

```
for _ in range(100):
    p.update()
```

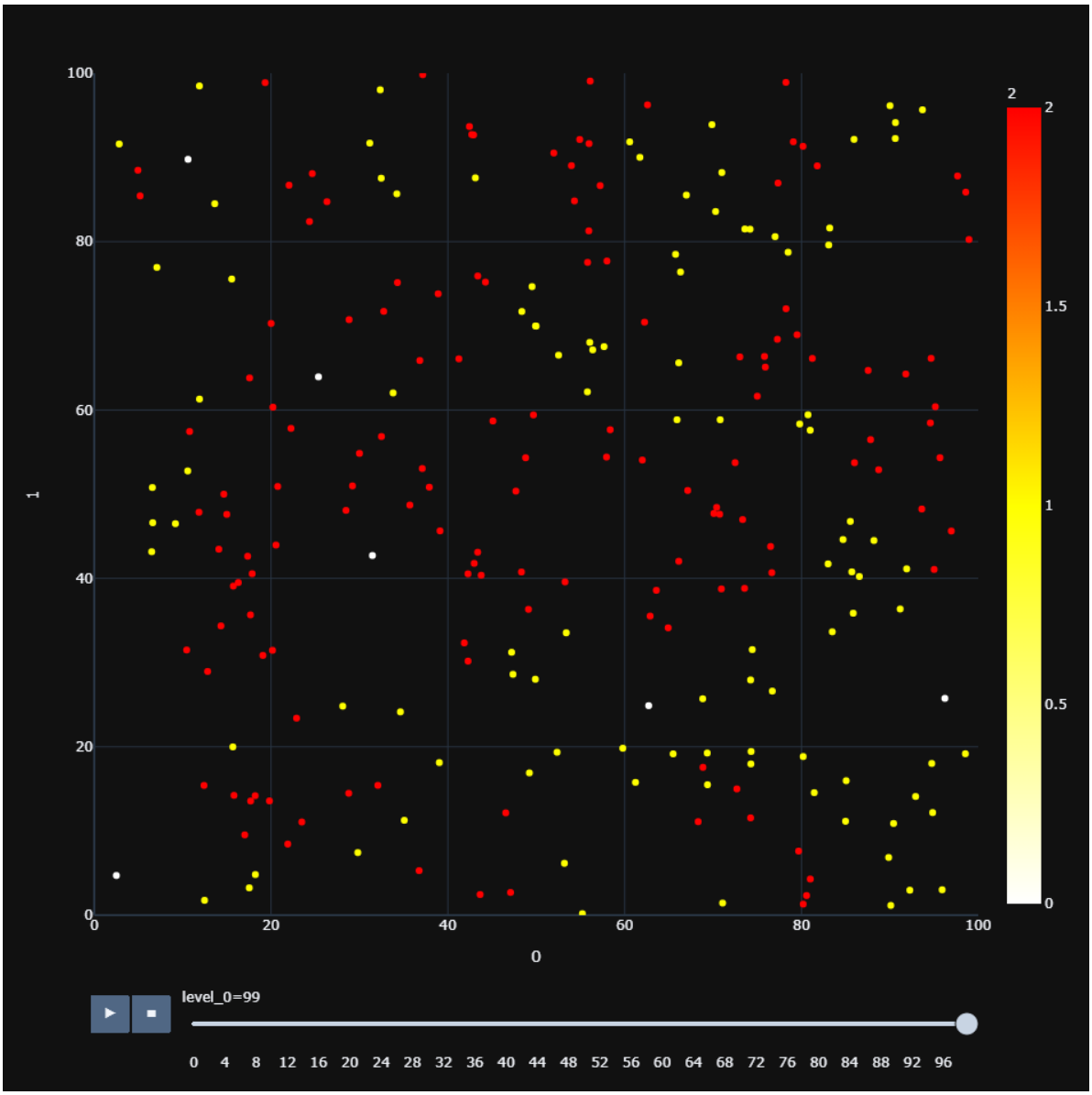
В [64]:

```
# Анимация
fig = px.scatter(pd.concat(p.stat, keys=[i for i in range(len(p.stat))]).reset_index(),
           x=0, y=1, animation_frame="level_0",
           color=2, color_continuous_scale=['white', 'yellow', 'red'],
           range_x=[0,100], range_y=[0,100],
           template='plotly_dark')
fig.layout.height = 950
fig.layout.width = 950
# fig.show()
```

В [65]:

```
# Результат модели (100 итераций)
Image('plots/newplot (16).png')
```

Out[65]:



Агентная модель

Финансовые пузыри

В [66]:

```
# Вспомогательная функция для графиков
def show_plot(*arrays):
    colors = ['rgb(255,255,255)',
              'rgb(255,170,0)',
              'rgb(255,64,0)',
              'rgb(255,0,0)']

    data = [go.Scatter(y=array, marker_color=color) for array, color in zip(arrays,colo

    fig = go.Figure(data=data)
    fig.update_layout(template='plotly_dark')
    fig.update_layout(showlegend=False)
    fig.show()
```

Общая схема модели

Инвесторы

1. Ожидаемая цена
2. Решение {-1, 0, 1}
3. Модуль объема покупок или продаж
4. Объем покупок := решение * модуль объема

Рынок

1. Обновляет цену актива в зависимости от объема торгов
2. Передает агентам информацию о цене

Консервативные инвесторы

Покупают недооцененные активы и продают переоцененные

$$E(P_{t+1}) = P_t + \lambda(P^* - P_t)$$

B [67]:

```
class ConservativeInvestor:


    def __init__(self, lambda_, p_star, delta_1):
        self.lambda_ = lambda_
        self.p_star = p_star
        self.delta_1 = delta_1

    def expectations(self, price_t):
        return price_t + self.lambda_ * (self.p_star - price_t)

    def decision(self, price_t):

        if price_t <= self.p_star - self.delta_1:
            return 1

        elif self.p_star - self.delta_1 < price_t < self.p_star + self.delta_1:
            return 0

        elif price_t >= self.p_star - self.delta_1:
            return -1

    def volume(self, price_t):

        if price_t <= self.p_star - self.delta_1:
            return 1 - np.exp(-abs(price_t - (self.p_star - self.delta_1)))

        elif self.p_star - self.delta_1 < price_t < self.p_star + self.delta_1:
            return 0

        elif price_t >= self.p_star + self.delta_1:
            return 1 - np.exp(-abs(price_t - (self.p_star + self.delta_1)))

    def trade(self, price_t):
        return self.decision(price_t) * self.volume(price_t)
```

B [68]:

```
conservative = ConservativeInvestor(lambda_=0.2, p_star=1, delta_1=0.1)
```

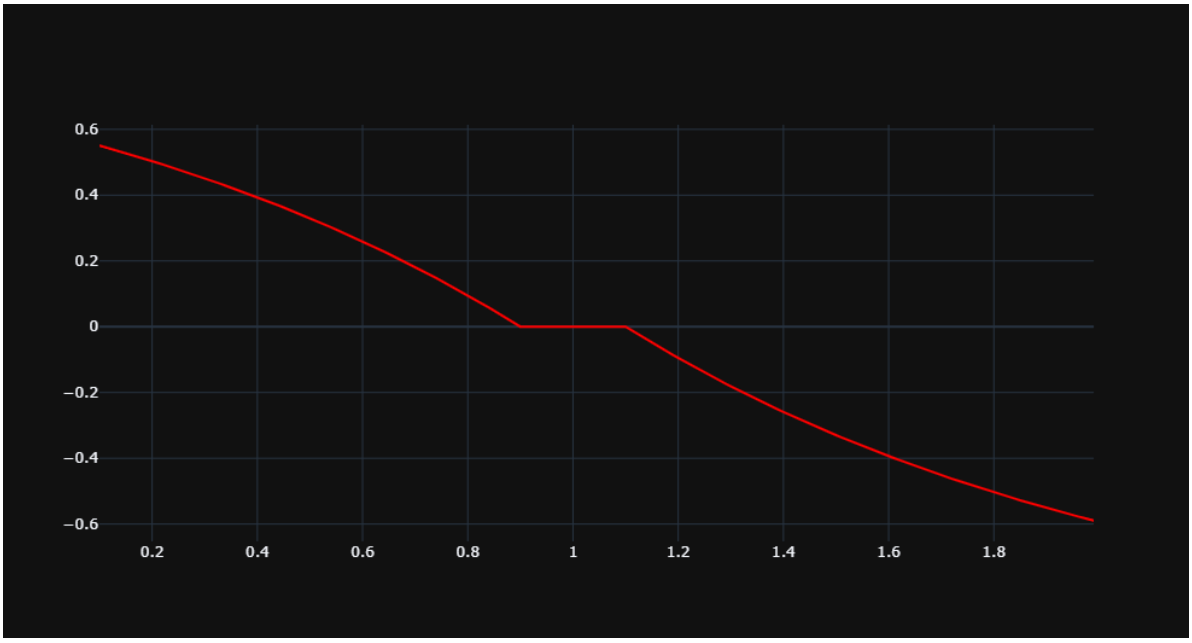
B [69]:

```
x = np.arange(0.1, 2.0, 0.01)
trade_list = [conservative.trade(i) for i in x]

fig = go.Figure(data=go.Scatter(x=x, y=trade_list, marker_color='rgb(255,0,0)'))
# fig.update_layout(template='plotly_dark')
# fig.show()
```

B [70]:

```
# Консервативные инвесторы
# Покупка / продажа актива в зависимости от цены
Image('plots/newplot (17).png')
```

Out[70]:



Спекулятивный инвестор

Скупают растущие в цене активы

(учитывают несколько последних изменений)

Также подвержены влиянию новостей (ϵ)

$$E(P_{t+1}) = P_t + \sum_{i=1}^{n} w_i(P_{t-i} - P_{t-i-1}) + \epsilon_t$$

$$\epsilon_t = N(0, \sigma^2)$$

B [71]:

```python
class SpeculativeInvestor:


    def __init__(self, w, delta_2, sigma2):
        self.w = w
        self.n = len(w)
        self.delta_2 = delta_2
        self.sigma2 = sigma2


    def expectations(self, prices):
        noise = np.random.normal(0, self.sigma2)
        diff = [prices[-i] - prices[-i-1] for i in range(1, self.n + 1)]
        lag = np.dot( self.w, diff )
        return prices[-1] + lag + noise


    def decision(self, prices):
        price_t = prices[-1]
        expected = self.expectations(prices)

        if expected >= price_t + self.delta_2:
            return 1

        elif price_t - self.delta_2 < expected < price_t + self.delta_2:
            return 0

        elif expected <= price_t - self.delta_2:
            return -1


    def volume(self, prices):
        price_t = prices[-1]
        expected = self.expectations(prices)

        if expected >= price_t + self.delta_2:
            return 1 - np.exp(-abs(expected - (price_t + self.delta_2)))

        elif price_t - self.delta_2 < expected < price_t + self.delta_2:
            return 0

        elif expected <= price_t - self.delta_2:
            return 1 - np.exp(-abs(expected - (price_t - self.delta_2)))


    def trade(self, prices):
        return self.decision(prices) * self.volume(prices)
```

Шумовой инвестор

Смотрят только на последнее изменение цены

$\Delta = P_t - P_{t-1}$

B [72]:

```
class NoiseInvestor:


    def __init__(self, delta_3):
        self.delta_3 = delta_3


    def decision(self, prices):
        price_delta = prices[-1] - prices[-2]

        if price_delta >= self.delta_3:
            return 1

        elif -self.delta_3 < price_delta < self.delta_3:
            return 0

        elif price_delta <= -self.delta_3:
            return -1


    def volume(self, prices):
        price_delta = prices[-1] - prices[-2]

        if price_delta >= self.delta_3:
            return 1 - np.exp(-abs(price_delta - self.delta_3))

        elif -self.delta_3 < price_delta < self.delta_3:
            return 0

        elif price_delta <= -self.delta_3:
            return 1 - np.exp(-abs(price_delta + self.delta_3))


    def trade(self, prices):
        return self.decision(prices) * self.volume(prices)
```

Рынок

В [73]:

```
def simulation(lambda_=0.2,
               p_star=1,
               delta_1=2,
               w=[0.08, 0.07, 0.06, 0.05],
               delta_2=0.02,
               sigma2=0.1,
               delta_3=0.3,
               alpha=0.3, # Доли инвесторов
               beta=0.6,
               theta=0.25, # Влияние на рынок
               T=10_000,
               start='rising',
               random_seed=42):

    np.random.seed(random_seed)

    # Создаем агентов
    conservative = ConservativeInvestor(lambda_, p_star, delta_1)
    speculative = SpeculativeInvestor(w, delta_2, sigma2)
    noise = NoiseInvestor(delta_3)

    # Начало симуляции
    if start == 'neutral':
        prices = [p_star] * (speculative.n + 1)
    elif start == 'rising':
        prices = [1 + 0.20*i for i in range(speculative.n + 1)]
    elif start == 'falling':
        prices = [1 - 0.05*i for i in range(speculative.n + 1)]

    for t in range(T):

        trade_1 = conservative.trade(prices[-1])
        trade_2 = speculative.trade(prices)
        trade_3 = noise.trade(prices)
        # Итоговый объем торгов
        trade_volume = alpha * trade_1 + beta * trade_2 + (1-alpha-beta) * trade_3
        # Добавляем новую цену
        new_price = prices[-1] + theta * trade_volume
        prices.append(new_price)

    return prices
```

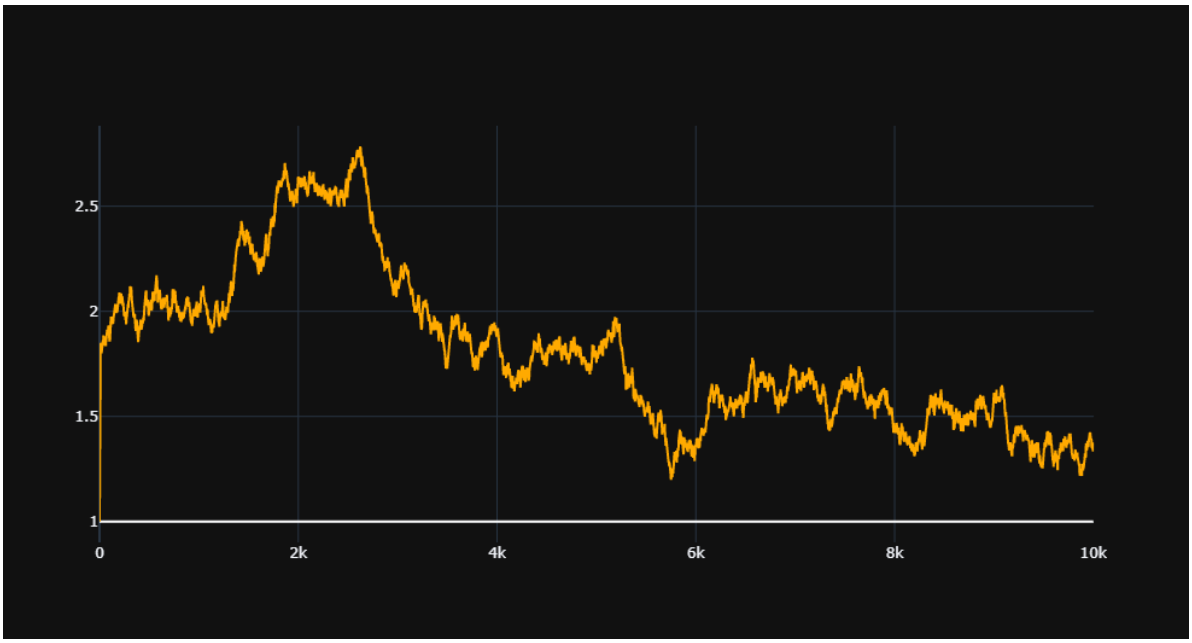
В [74]:

```
fair = [1] * 10_000
```

B [75]:

```
# Пример пузыря
# Цена поднимается, затем послепенно опускается
s1 = simulation()
# show_plot(fair, s1)
Image('plots/newplot (18).png')
```

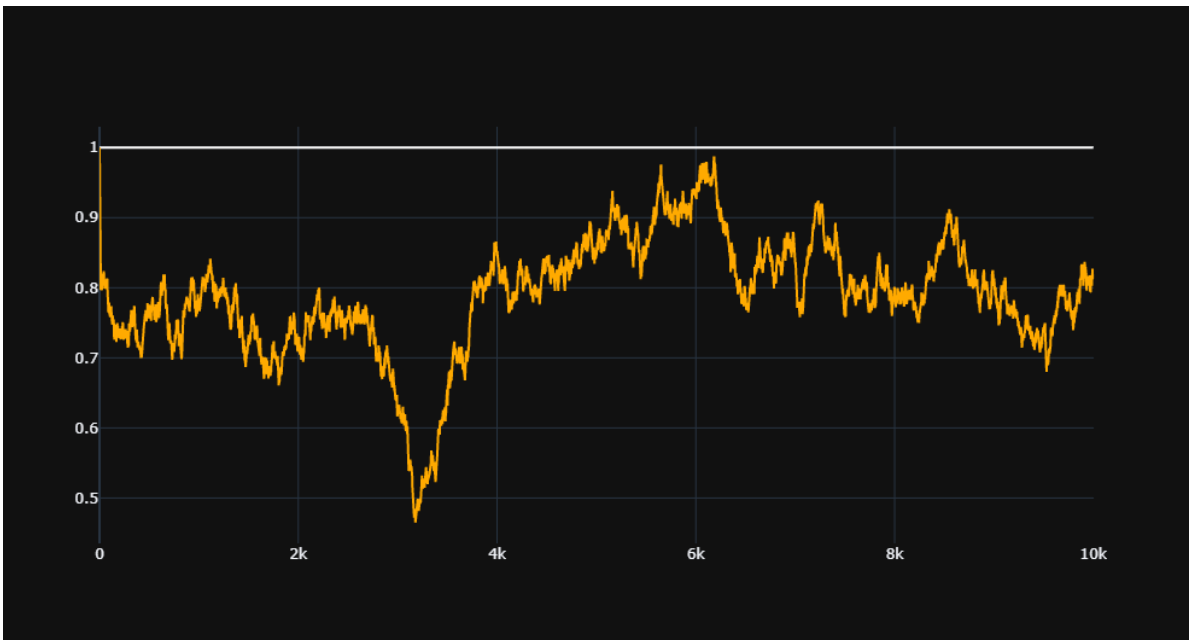
Out[75]:



B [76]:

```
# Пример "Антипузыря"
# Начинается с падения, затем цена растет
s1 = simulation(start='falling',
                theta=0.1,
                random_seed=101)
# show_plot(fair, s1)
Image('plots/newplot (19).png')
```

Out[76]:



Влияние параметров

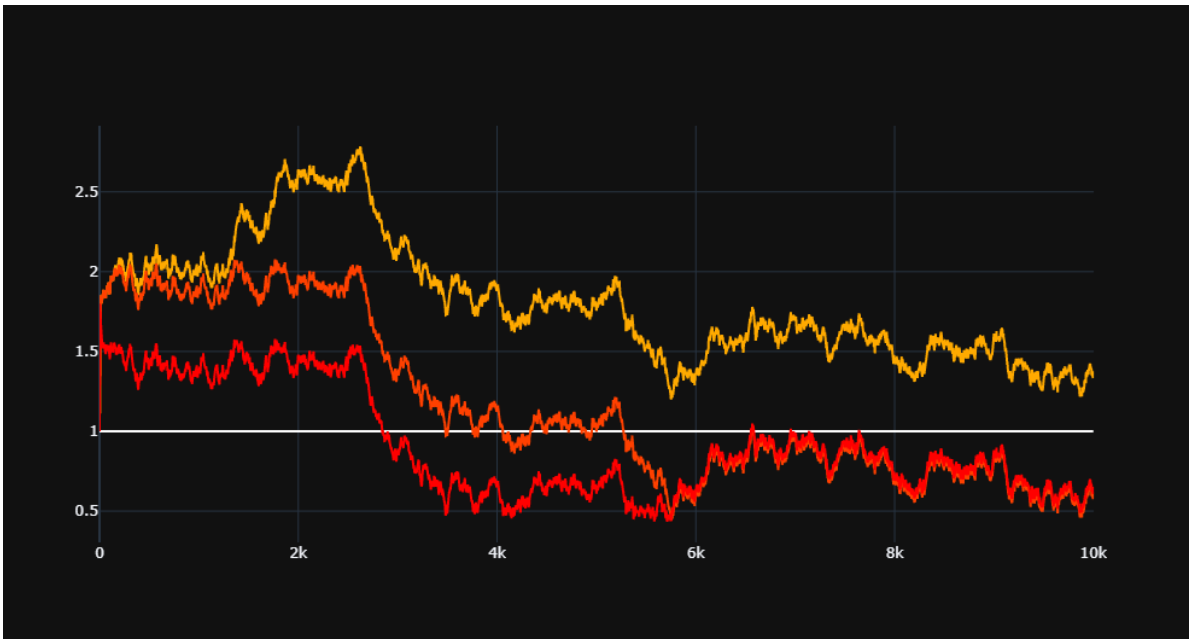
Чем краснее линия, тем больше значение параметра

Активность консервативных инвесторов

B [77]:

```
s1 = simulation(delta_1=2)
s2 = simulation(delta_1=1)
s3 = simulation(delta_1=0.5)

# show_plot(fair, s1, s2, s3)
Image('plots/newplot (20).png')
```

Out[77]:

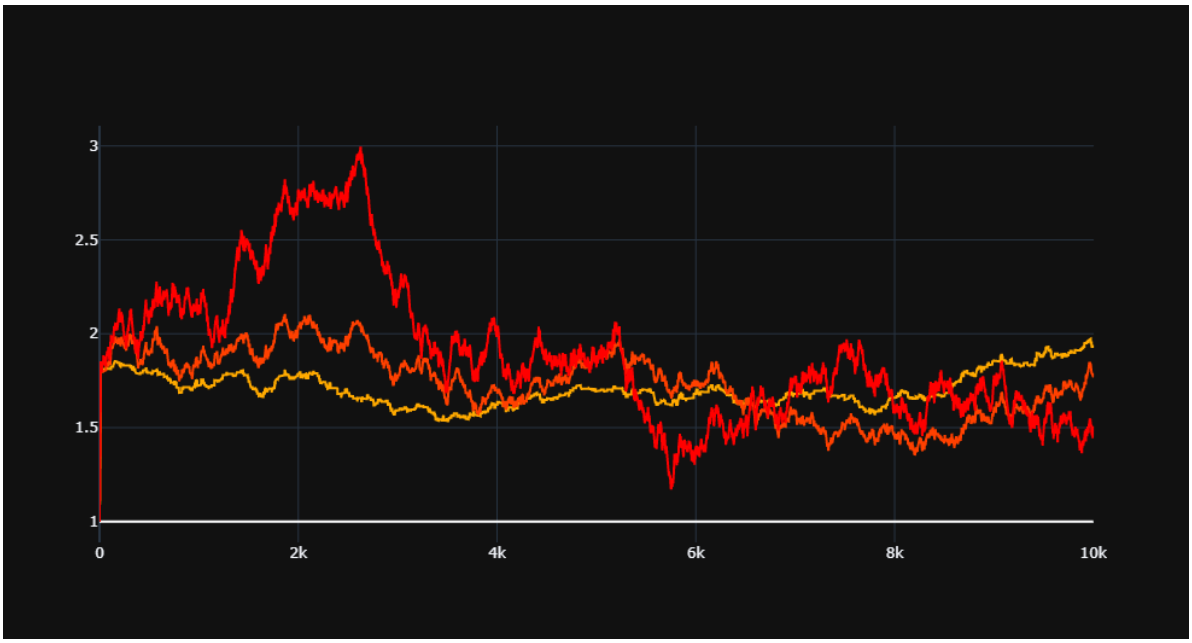


Активность спекулянтов

В [78]:

```
s1 = simulation(delta_2=0.1)
s2 = simulation(delta_2=0.05)
s3 = simulation(delta_2=0.01)

# show_plot(fair, s1, s2, s3)
Image('plots/newplot (21).png')
```

Out[78]:

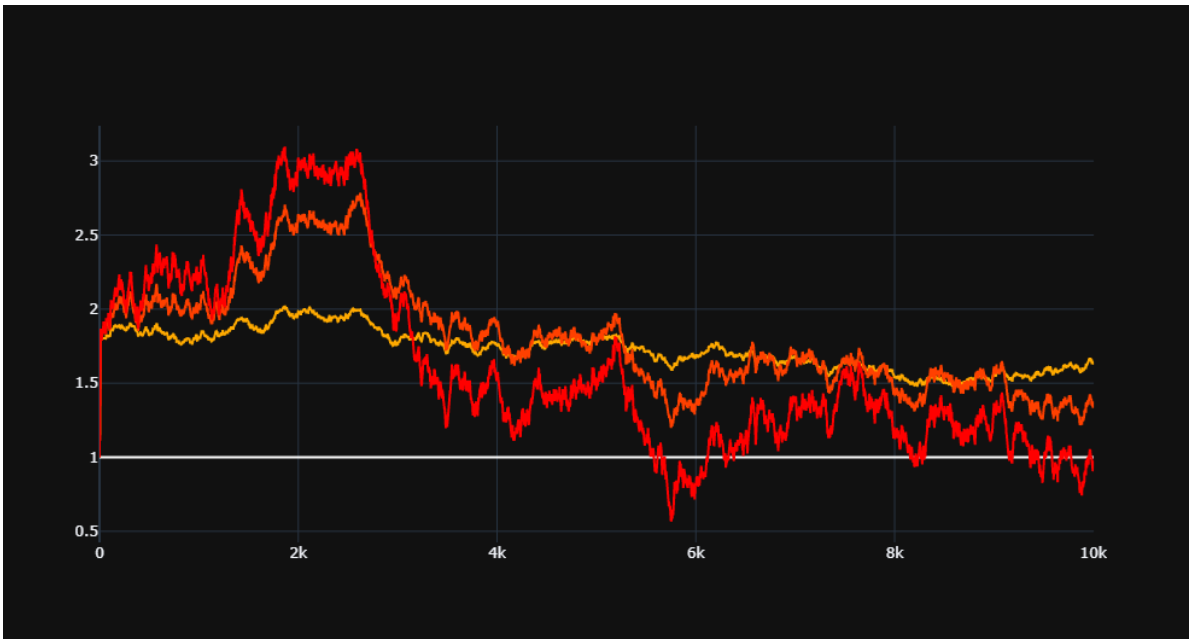


Сила новостей

B [79]:

```
s1 = simulation(sigma2=0.05)
s2 = simulation(sigma2=0.10)
s3 = simulation(sigma2=0.15)

# show_plot(fair, s1, s2, s3)
Image('plots/newplot (22).png')
```

Out[79]:

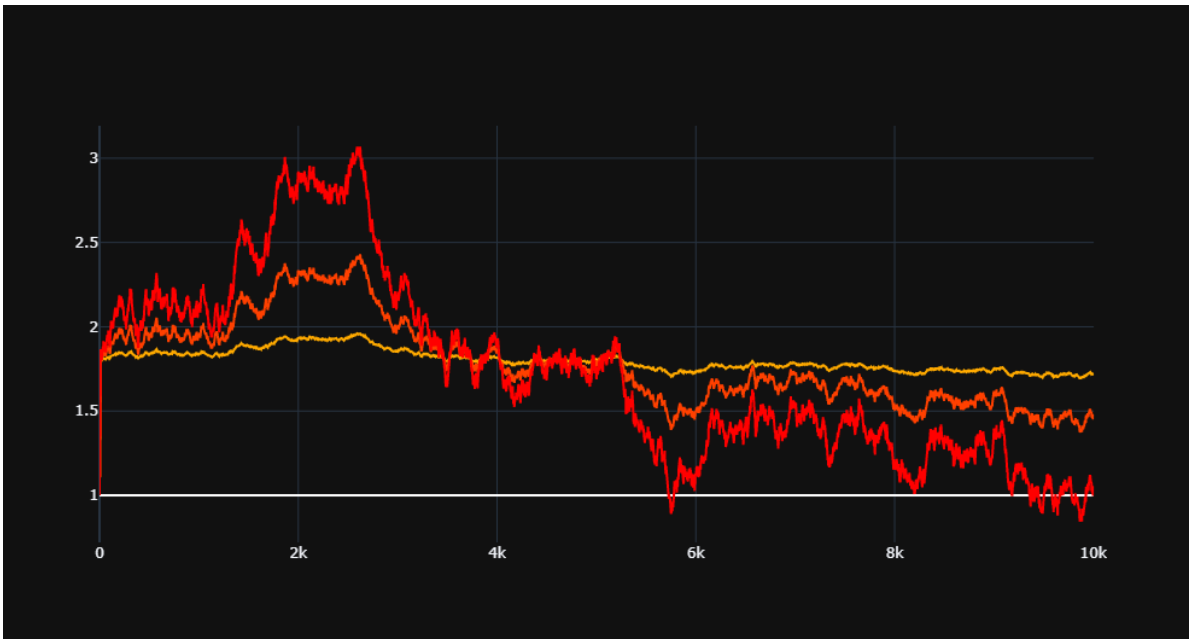


Доля спекулятивных инвесторов

B [80]:

```
s1 = simulation(alpha=0.8, beta=0.1)
s2 = simulation(alpha=0.4, beta=0.4)
s3 = simulation(alpha=0.1, beta=0.8)

# show_plot(fair, s1, s2, s3)
Image('plots/newplot (23).png')
```

Out[80]:

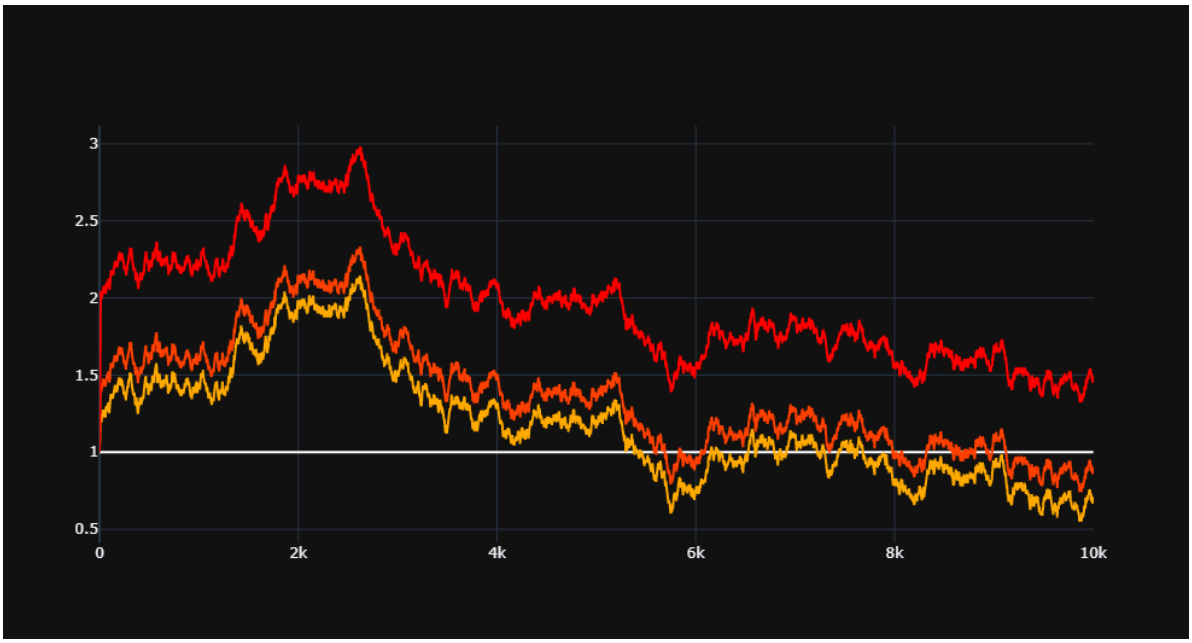


Равномерность распределения весов спекулянтов

В [81]:

```
s1 = simulation(w=[0.10] * 1)
s2 = simulation(w=[0.05] * 2)
s3 = simulation(w=[0.02] * 5)

# show_plot(fair, s1, s2, s3)
Image('plots/newplot (24).png')
```

Out[81]:



Ряд Тейлора

Вычисляем следующее слагаемое на основе предыдущего

В [82]:

```
def exponent_taylor(x, k):
    current = 1; sum_ = 1
    for i in range(1, k+1):
        sum_ += (current := current * x / i)
    return sum_
```

B [83]:

```
exponent_taylor(1, 8)
```

Out[83]:

```
2.71827876984127
```

B [84]:

```
def sin_taylor(x, k):
    current = x; sum_ = x
    for i in range(1, k+1):
        sum_ += (current := -current * x**2 / (2*i) / (2*i+1) )
    return sum_
```

B [85]:

```
sin_taylor(0.6, 50)
```

Out[85]:

```
0.5646424733950354
```

B [86]:

```
def sin_taylor_epsilon(x, epsilon=0.001):
    current = x; sum_ = x; i = 1
    while abs(current) > epsilon:
        sum_ += (current := -current * x**2 / (2*i) / (2*i+1) )
        i += 1
    return sum_
```

B [87]:

```
sin_taylor_epsilon(0.6, 0.00001)
```

Out[87]:

```
0.5646424457142857
```

Подготовка студентов к экзамену

В [88]:

```
class Student:

    def __init__(self):
        self.question_time_list = []

    def train_one_question(self):
        # Первое прочтение
        first_time = np.random.triangular(20, 30, 40)
        first_quality = np.random.triangular(0, 0.8, 1)

        # Воспроизведение
        if first_quality < 0.2:
            repeat_time = first_time
        elif first_quality < 0.5:
            repeat_time = first_time / 2
        elif first_quality < 0.8:
            repeat_time = first_time / 3
        else:
            repeat_time = 0

        # Обновляем время
        total_time = first_time + repeat_time
        self.question_time_list.append(total_time)
        return total_time

    # Подготовка к экзамену
    def train_to_exam(self, n_questions=20):
        for i in range(n_questions):
            self.train_one_question()

    def get_cumulative_time(self):
        return pd.Series(self.question_time_list).cumsum()
```

В [89]:

```
# Группа студентов
class StudentGroup:

    def __init__(self, n_students):
        self.population = [Student() for i in range(n_students)]

    def train_to_exam(self, n_questions=20):
        for student in self.population:
            student.train_to_exam(n_questions)

    def get_stat(self):
        df = pd.DataFrame()
        for i, student in enumerate(self.population):
            df[i] = student.get_cumulative_time()
        return df
```

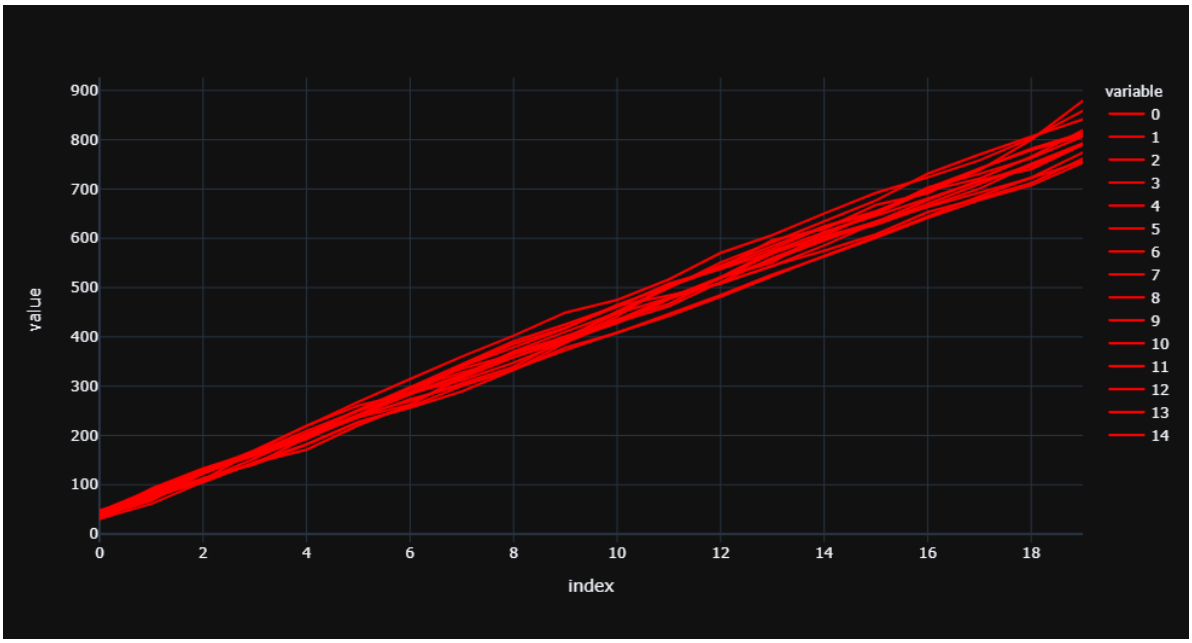
В [90]:

```
a = StudentGroup(15)
a.train_to_exam()
df = a.get_stat()
```

In [91]:

```
# X: Индекс вопроса
# Y: Время на подготовку
fig = px.line(df, template='plotly_dark')
fig.update_traces(line_color='red')
# fig.show()
Image('plots/newplot (25).png')
```

Out[91]:



Варьирование параметров модели

В модели финансовых пузырей рассмотрим, насколько высоко может подняться пузырь

In [92]:

```
from random import choice
```

In [93]:

```
# Перебираемые параметры
search_space = dict(
    lambda_=[0.1, 0.2, 0.3, 0.4],
    delta_1=[1, 1.5, 2, 2.5],
    w=[[0.08, 0.07, 0.06, 0.05], [0.1, 0.1, 0.1], [0.5, 0.2]],
    delta_2=[0.01, 0.02, 0.03, 0.04],
    sigma2=[0.1, 0.2, 0.3],
    delta_3=[0.1, 0.2, 0.3],
    alpha=[0.3, 0.4, 0.5],
    beta=[0.6, 0.5, 0.4],
    theta=[0.2, 0.3, 0.4]
)
```

В [100]:

```
max_peak = 0
plot = None
for _ in range(50):
    # Случайно выбираем параметры из списка возможных
    kwargs = {k: choice(v) for k, v in search_space.items()}
    s = simulation(**kwargs)
    peak_value = max(s)
    # Проверяем
    if peak_value > max_peak:
        max_peak = peak_value
        plot = s

print(max_peak)
```

```
3.689369874278196
```

В [103]:

```
# Наблюдаем постепенный рост цены,
# затем резкое падение и
# постепенное восстановление к справедливой цене

Image('plots/newplot (26).png')
```

Out[103]: