Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

ОТЧЕТ

по лабораторной работе №4

по курсу «Моделирование»

на тему: «Обслуживающий аппарат»

Студент ИУ7-73Б _____ Лагутин Д. В.

(Группа) (Подпись, дата) (Фамилия И. О.)

Преподаватель _____ Рудаков И. В.

(Подпись, дата) (Фамилия И. О.)

Москва, 2023 г.

Цель работы

Целью работы является моделирование системы, состоящей из генератора, буфера и обслуживающего аппарата. Генератор выдает сообщения, распределенные по равномерному закону, они приходят в память, обслуживающий аппарат обрабатывает каждое из них согласно нормальному распределению. Необходимо определить оптимальную длину очереди, при которой не будет потерянных сообщений. Использовать принципы Δt и событийный. Задаваемая часть сообщений попадает в очередь повторно.

Принципы организации управляющей программы

Принцип Δt

Данный принцип заключается в последовательном анализе состояний всех блоков в момент $t + \Delta t$ по заданному состоянию блоков в момент t. При этом новое состояние блоков определяется в соответствии с их алгоритмическим описанием с учетом действующих случайных факторов, задаваемых распределениями вероятности. В результате такого анализа принимается решение о том, какие общесистемные события должны имитироваться программной моделью на данный момент времени.

Основной недостаток этого принципа: значительные затраты машинного времени на реализацию моделирования системы. А при недостаточно малом Δt появляется опасность пропуска отдельных событий в системе, что исключает возможность получения адекватных результатов при моделировании.

Событийный принцип

Характерное свойство систем обработки информации заключается в том, что состояния отдельных устройств изменяются в дискретные моменты времени, совпадающие с моментами времени поступления сообщений в систему, времени поступления окончания задачи, времени поступления аварийных сигналов и т.д. Поэтому моделирование и продвижение времени в системе удобно проводить, используя событийный принцип, при котором состояние всех блоков имитационной модели анализируется лишь в момент появления какого-либо события. Момент поступления следующего события определяется минимальным значением из списка будущих событий, представляющего собой совокупность моментов ближайшего изменения состояния каждого из блоков системы.

Моделируемая система

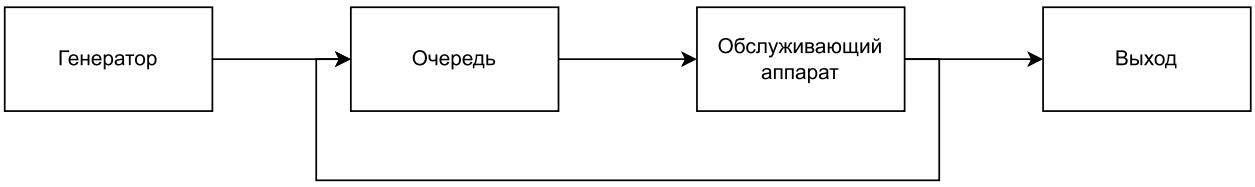


Рисунок 1 – Схема моделируемой системы

Текст программы

Листинг 1 – Реализация элементов системы

```
1  class Model
2  {
3      public:
4          Model(std::string name);
5          virtual ~Model(void) = 0;
6
7          virtual const std::string &getName(void) const;
8
9          virtual void
10         setModifier(std::shared_ptr<RequestModifier>);
11         virtual std::shared_ptr<RequestModifier>
12         getModifier(void) const;
13
14     private:
15         const std::string name;
16         std::shared_ptr<RequestModifier> modifier;
17 };
18
19 class Request
20 {
21     public:
22         virtual ~Request(void) = 0;
23 };
24
25 class Sender;
26
27 class Receiver
28 {
29     public:
30         Receiver(std::shared_ptr<Pipe> pipe);
31         virtual ~Receiver(void) = 0;
32
33         void connectInPipe(std::shared_ptr<Pipe> pipe);
34         std::shared_ptr<Pipe> inpipe(void);
35         void askSender(void);
36
```

```
37          void registerSender(std::shared_ptr<Sender> sender);
38
39      private:
40          std::shared_ptr<Pipe> _inpipe;
41          std::list<std::shared_ptr<Sender>> _senders;
42  };
43
44  class Sender
45  {
46      public:
47          Sender(std::shared_ptr<Pipe> pipe);
48          virtual ~Sender(void) = 0;
49
50          void connectOutPipe(std::shared_ptr<Pipe> pipe);
51          std::shared_ptr<Pipe> outpipe(void);
52
53          virtual void callback(void) = 0;
54
55      private:
56          std::shared_ptr<Pipe> _outpipe;
57  };
58
59  class Runner
60  {
61      public:
62          virtual ~Runner(void) = default;
63          virtual void run(void) = 0;
64  };
65
66  class BasicPipe : public Pipe
67  {
68      public:
69          BasicPipe(std::string name);
70          virtual ~BasicPipe(void) override = default;
71
72          virtual bool empty(void) const override;
73          virtual bool push(std::shared_ptr<Request>) override;
74          virtual std::shared_ptr<Request> pop(void) override;
75
76          virtual void newSet(void) override;
77          virtual void dropCurrentSet(void) override;
```

```
78          virtual void clear(void) override;
79
80      private:
81          std::mutex mutex;
82          std::list<std::list<std::shared_ptr<Request>>> sets;
83  };
84
85  class Buffer : public Model, public Sender, public Receiver
86  {
87      public:
88          Buffer(std::string name, const size_t size,
89                 std::shared_ptr<Pipe> inpipe,
90                 std::shared_ptr<Pipe> outpipe);
91          virtual ~Buffer(void) override = default;
92
93          virtual size_t read(void);
94          virtual size_t send(size_t amount = 1);
95
96          virtual size_t used(void) const;
97
98          virtual void callback(void) override;
99
100     private:
101         const size_t size;
102         std::list<std::shared_ptr<Request>> memory;
103         std::mutex mutex;
104 };
105
106 class Gate : public Model, public Receiver
107 {
108     public:
109         class Out : public Model, public Sender
110         {
111             public:
112                 Out(std::string name,
113                     std::shared_ptr<Pipe> outpipe);
114                 virtual ~Out(void) override = default;
115
116                 virtual bool send(std::shared_ptr<Request>);
117
118                 virtual void callback(void) override;
```

```
119         };
120
121     public:
122         Gate(std::string name, std::shared_ptr<Pipe> inpipe,
123              std::list<std::shared_ptr<Pipe>> outpipes);
124         virtual ~Gate(void) override = default;
125
126         virtual void
127         setModifier(std::shared_ptr<RequestModifier>)
128         override;
129
130         virtual bool read(void);
131         virtual bool redirect(std::string name);
132         virtual const std::list<std::string> &list(void);
133
134     private:
135         std::list<std::shared_ptr<Out>> outs;
136         std::list<std::string> names;
137         std::shared_ptr<Request> current = nullptr;
138 };
139
140 class RequestCreator
141 {
142     public:
143         virtual ~RequestCreator(void) = default;
144         virtual std::shared_ptr<Request> create(void) = 0;
145 };
146
147 class Generator : public Model, public Sender
148 {
149     public:
150         Generator(std::string name,
151                   std::shared_ptr<Pipe> outpipe = nullptr);
152         virtual ~Generator(void) override = default;
153         virtual void callback(void) override;
154
155         virtual void
156         setCreator(std::shared_ptr<RequestCreator> creator);
157
158         virtual void generate(const size_t amount = 1);
159
```

```
160     private:
161         std::shared_ptr<RequestCreator> cretator = nullptr;
162 };
163
164 class Processor : public Model, public Sender,
165                   public Receiver
166 {
167     public:
168         Processor(std::string name,
169                   std::shared_ptr<Pipe> inpipe = nullptr,
170                   std::shared_ptr<Pipe> outpipe = nullptr);
171         virtual ~Processor(void) override = default;
172
173         virtual bool read(void);
174         virtual bool isActive(void) const;
175         virtual bool release(void);
176
177         virtual void callback(void) override;
178
179     private:
180         std::shared_ptr<Request> active = nullptr;
181         std::list<std::shared_ptr<Request>> current;
182 };
183
184 class Terminator : public Model, public Receiver
185 {
186     public:
187         Terminator(std::string name,
188                    std::shared_ptr<Pipe> inpipe = nullptr);
189         virtual ~Terminator(void) override = default;
190
191         virtual void read(void);
192         std::list<std::shared_ptr<Request>> getDone(void);
193
194     private:
195         std::list<std::shared_ptr<Request>> done;
196 };
197
198 class StatatisticsBlock : public Model
199 {
200     public:
```

```
201         using ModelMap =
202         std::unordered_map<std::string,
203                            std::shared_ptr<Model>>;
204         class Strategy
205         {
206             public:
207                 virtual ~Strategy(void) = default;
208                 virtual void
209                 execute(const ModelMap &model) = 0;
210         };
211
212     public:
213         StatatisticsBlock(std::string name,
214                           std::list<std::shared_ptr<Model>>);
215         virtual ~StatatisticsBlock(void) override = default;
216
217         virtual void
218         registerStrategy(std::shared_ptr<Strategy> strategy);
219         virtual void write(void);
220
221     private:
222         std::list<std::shared_ptr<Strategy>> strategies;
223         ModelMap map;
224 };
```

Листинг 2 – Реализация подхода Δt

```
 1 class TimeModel
 2 {
 3     public:
 4         virtual ~TimeModel(void) = default;
 5         virtual size_t priority(void) = 0;
 6         virtual void tick(double time) = 0;
 7         virtual void
 8         setModifier(std::shared_ptr<RequestModifier>) = 0;
 9 };
10
11 class TimeRunner : public Runner
12 {
13     public:
14         TimeRunner(size_t requests, double time, double step,
15                    std::shared_ptr<TimeRequestModifier>,
```

```
16                      std::list<std::shared_ptr<TimeModel>>);
17          virtual ~TimeRunner(void) override = default;
18
19          virtual void run(void) override;
20
21      private:
22          const size_t requests;
23          const double end;
24          const double step;
25          std::shared_ptr<TimeRequestModifier> modifier;
26          std::map<size_t,
27                   std::list<std::shared_ptr<TimeModel>>> \
28          items;
29  };
30
31  TimeRunner::TimeRunner(size_t requests, double time,
32                         double step,
33                         std::shared_ptr<TimeRequestModifier>,
34                         std::list<std::shared_ptr<TimeModel>>)
35      : requests(requests), end(std::abs(time)),
36        step(std::abs(step)), modifier(modifier)
37  {
38      if (nullptr == this->modifier)
39          throw std::logic_error("Nullptr modifier");
40
41      for (auto &item : items)
42      {
43          if (nullptr == item)
44              throw std::logic_error("Nullptr modifier");
45
46          auto iter = this->items.find(item->priority());
47
48          if (this->items.end() == iter)
49              this->items.emplace(item->priority(),
50              std::list<std::shared_ptr<TimeModel>>({item}));
51          else
52              (*iter).second.push_back(item);
53      }
54  }
55
56  void TimeRunner::run(void)
```

```
57 {
58     double time = 0;
59     this->modifier->setTime(time);
60     auto modifier = this->modifier->getModifier();
61 
62     for (auto &pair : this->items)
63         for (auto &item : pair.second)
64             item->setModifier(modifier);
65 
66     for (; this->end > time
67            && this->requests > this->modifier->getPassed();
68            time += this->step)
69         for (auto &pair : this->items)
70             for (auto &item : pair.second)
71                 item->tick(time);
72 }
```

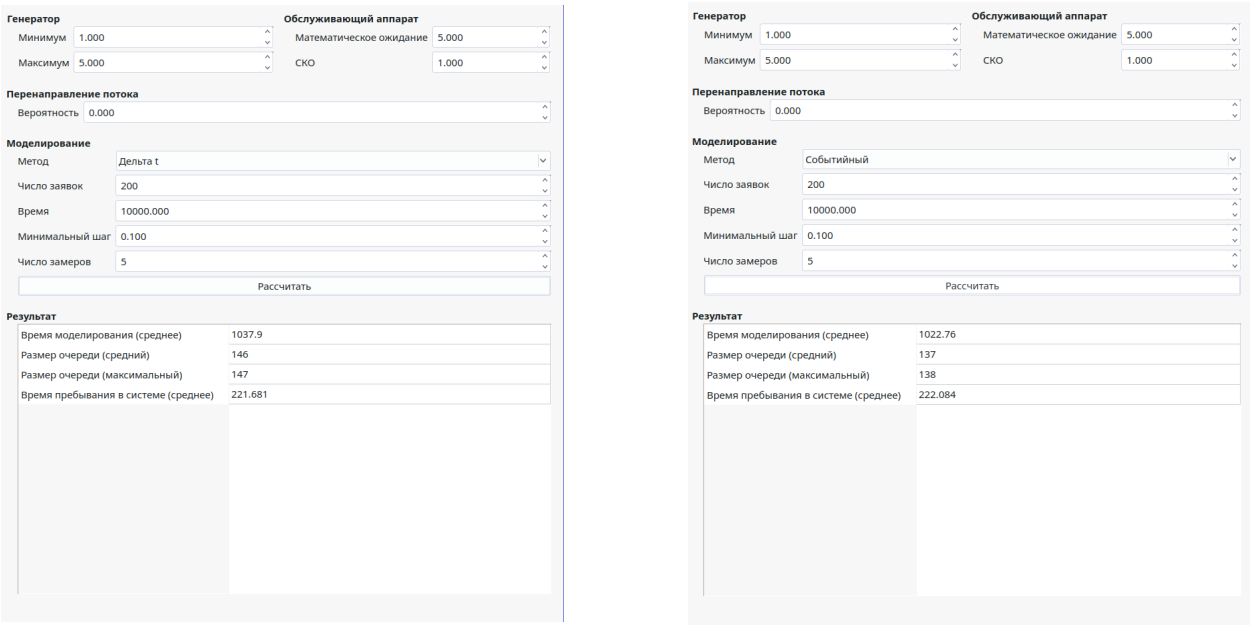
Листинг 3 – Реализация событийного подхода

```
 1 class EventModel
 2 {
 3     public:
 4         virtual ~EventModel(void) = default;
 5         virtual void event(void) = 0;
 6         virtual double nextEvent(void) const = 0;
 7         virtual void generateNextEvent(void) = 0;
 8         virtual void
 9         setModifier(std::shared_ptr<RequestModifier>) = 0;
10 };
11 
12 class EventRunner : public Runner
13 {
14     public:
15         EventRunner(size_t requests, double time,
16                     std::shared_ptr<EventRequestModifier>,
17                     std::list<std::shared_ptr<EventModel>>);
18         virtual ~EventRunner(void) override = default;
19 
20         virtual void run(void) override;
21 
22     private:
23         const size_t requests;
```

```
24          const double end;
25          std::shared_ptr<EventRequestModifier> modifier;
26          std::list<std::shared_ptr<EventModel>> items;
27  };
28
29  EventRunner::EventRunner(size_t requests, double time,
30
        std::shared_ptr<EventRequestModifier>,
31
        std::list<std::shared_ptr<EventModel>>)
32      : requests(requests), end(time), modifier(modifier),
33        items(items)
34  {
35      if (nullptr == this->modifier)
36          throw std::logic_error("Nullptr modifier");
37
38      for (auto &item : items)
39          if (nullptr == item)
40              throw std::logic_error("Nullptr modifier");
41  }
42
43  void EventRunner::run(void)
44  {
45      double time = 0;
46      this->modifier->setTime(time);
47      auto modifier = this->modifier->getModifier();
48
49      for (auto &item : this->items)
50          item->setModifier(modifier);
51
52      while (this->end > time
53             && this->requests > this->modifier->getPassed())
54      {
55          auto iter = this->items.begin(), next = iter;
56
57          for (; this->items.end() != iter; ++iter)
58              if ((*iter)->nextEvent() < (*next)->nextEvent())
59                  next = iter;
60
61          auto model = *next;
62          time = model->nextEvent();
```

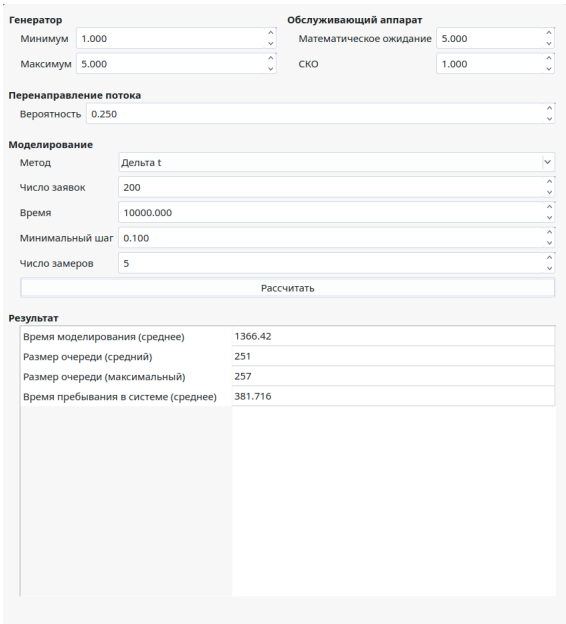
```
63            model->event();
64            model->generateNextEvent();
65        }
66    }
```

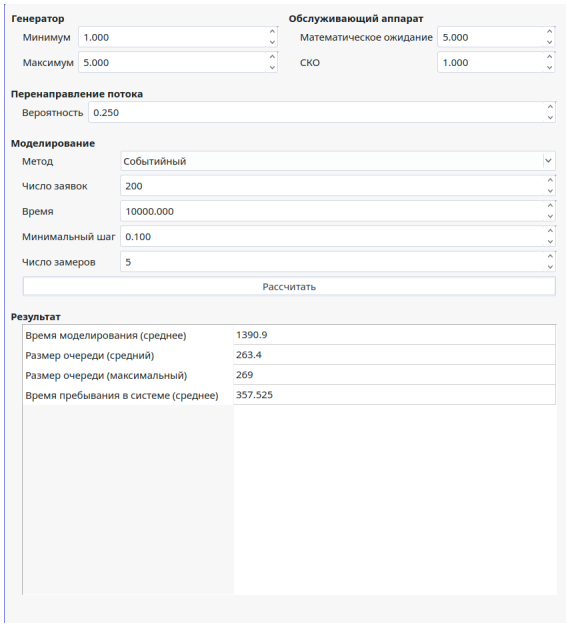
Результаты работы



Ожидаемое время моделирования: 1000

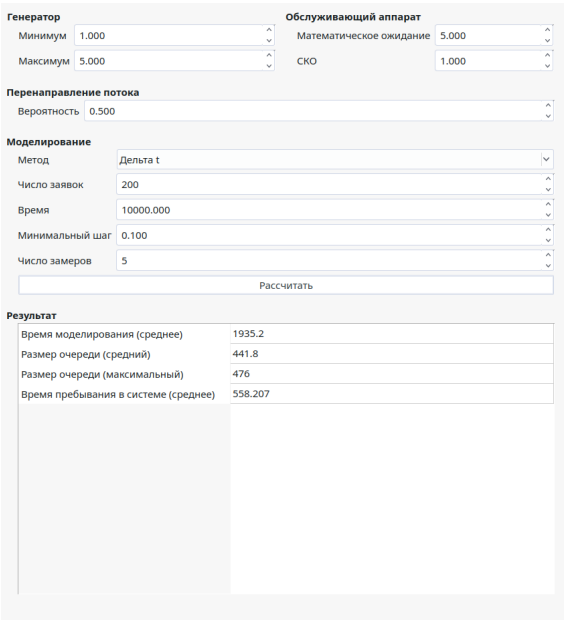
Рисунок 2 – Результаты работы для $a = 1, b = 5, \mu = 5, \sigma = 1, p = 0, n = 200$

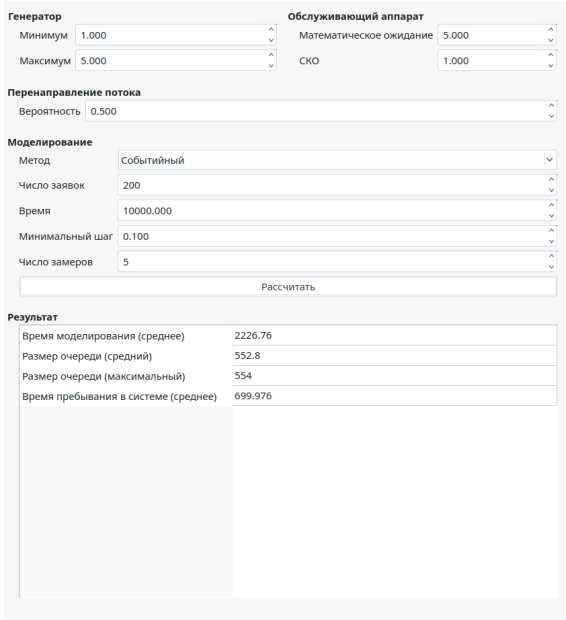




Ожидаемое время моделирования: 1333.33

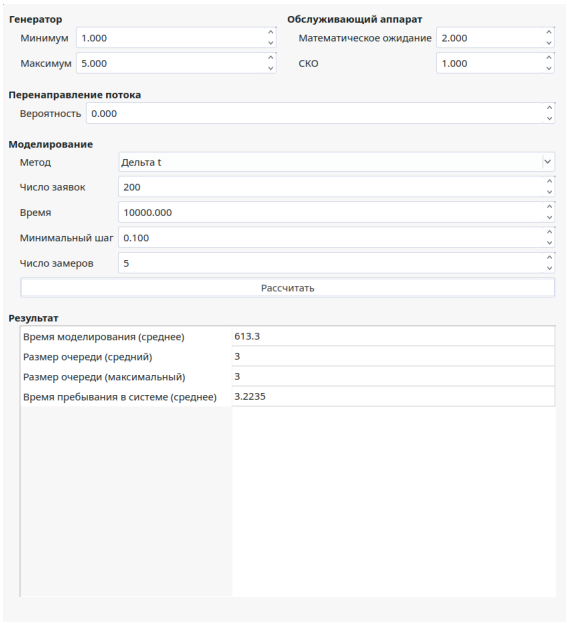
Рисунок 3 – Результаты работы для $a = 1, b = 5, \mu = 5, \sigma = 1, p = 0.25, n = 200$

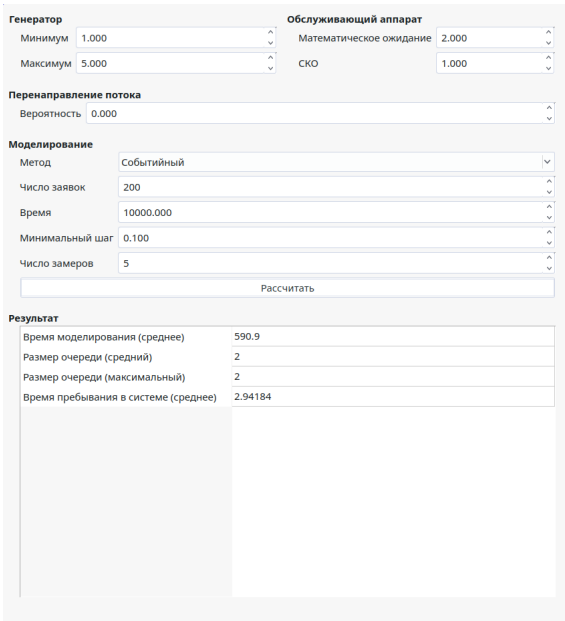




Ожидаемое время моделирования: 2000

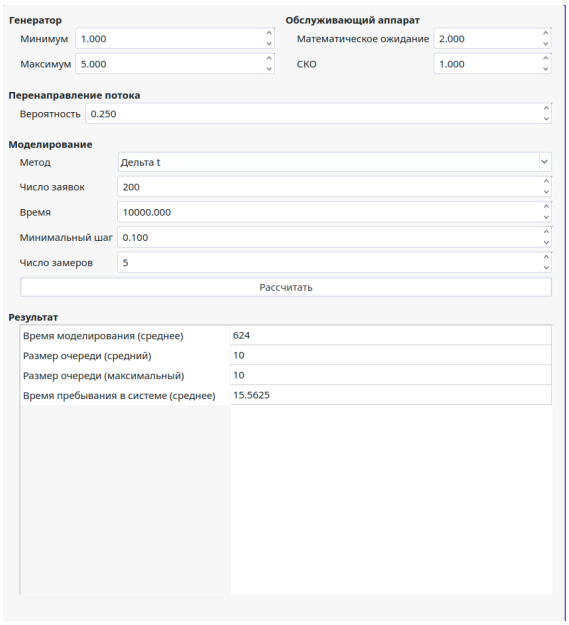
Рисунок 4 – Результаты работы для $a = 1, b = 5, \mu = 5, \sigma = 1, p = 0.5, n = 200$

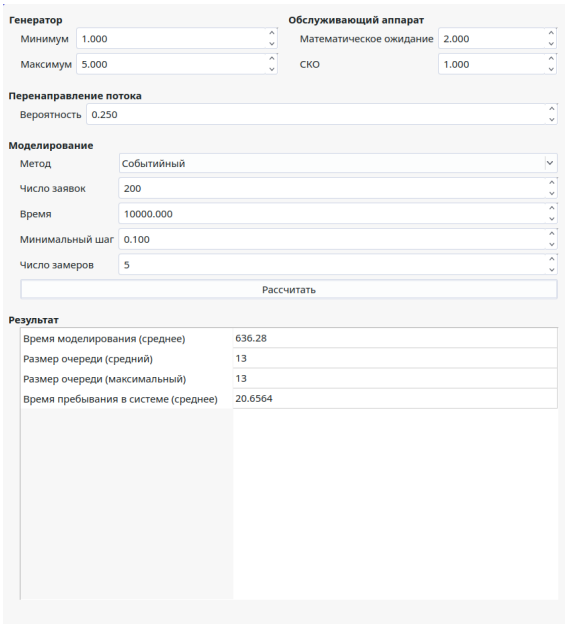




Ожидаемое время моделирования: 600

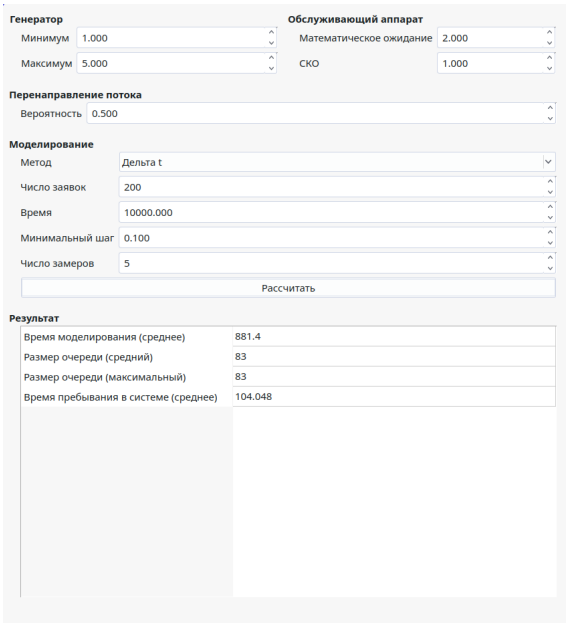
Рисунок 5 – Результаты работы для $a = 1, b = 5, \mu = 2, \sigma = 1, p = 0, n = 200$

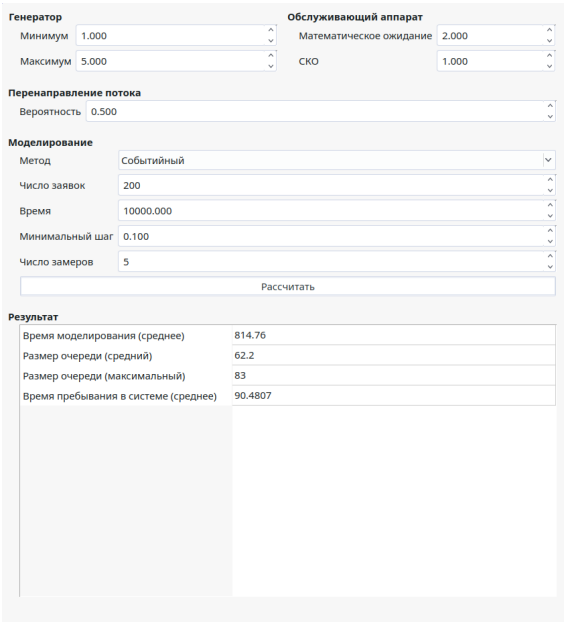




Ожидаемое время моделирования: 600

Рисунок 6 – Результаты работы для $a = 1, b = 5, \mu = 2, \sigma = 1, p = 0.25, n = 200$





Ожидаемое время моделирования: 800

Рисунок 7 – Результаты работы для $a = 1, b = 5, \mu = 2, \sigma = 1, p = 0.5, n = 200$

Вывод

В ходе выполнения работы была промоделирована системы, состоящей из генератора, буфера и обслуживающего аппарата с использованием методов протягивания модельного времени (Δt и событийный). Была рассмотрена зависимость длины очереди от параметров системы.