Имитационное моделирование: практикум

К.Ю. Поляков,
д. т. н., Санкт-Петербург

Введение

Слова "имитационное моделирование" воспринимаются с благоговейным трепетом всеми, кто не очень хорошо понимает, что это такое. Между тем это одно из важных направлений в современной науке, чрезвычайно полезное прежде всего при решении *практических* задач управления сложными системами.

Задачи, которые приходится решать в жизни, обладают, как правило, одной важной особенностью: они не решаются (или очень трудно решаются) с помощью существующих теоретических методов. Достаточно вспомнить, что не существует (и в ближайшее время не предвидится) общих способов решения нелинейных дифференциальных уравнений. Это значит, что реальные системы очень часто настолько сложны, что мы не можем теоретически предсказать их поведение в интересующих нас условиях и тем более не можем сразу установить, как следует изменить систему для того, чтобы она стала выполнять нужные нам функции.

В такой ситуации появляется два возможных направления движения:

1) упростить математическую модель до такой степени, чтобы ее можно было исследовать теоретически (например, "убрать" все нелинейности в модели системы, линеаризовать ее);

2) применить методы имитационного моделирования [1].

Первый подход (упрощение модели) очень важен, и им не стоит пренебрегать, поскольку только он позволяет получать хотя бы приближенные теоретические результаты и сделать качественные выводы. В то же время эти выводы (полученные на упрощенной, то есть на *другой* модели) обязательно нуждаются в проверке либо на реальном объекте, либо путем компьютерного моделирования с использованием полной модели.

Во многих случаях мы можем достаточно подробно описать поведение

Автор благодарит к. ф.-м. н. Е.А. Еремина за обсуждение статьи и полезные замечания.

сложной системы, то есть алгоритм ее работы при известных входных данных. Это позволяет *сымитировать* работу системы в интересующих нас условиях и посмотреть, что же с ней будет происходить. Такой подход и получил название *имитационного моделирования*. Особенно широко он стал применяться с внедрением компьютерных технологий, которые позволили выполнять огромный объем вычислений за приемлемое время. Нередко цель имитационного моделирования — определить оптимальные значения параметров объекта путем многократных экспериментов с моделью, методом проб и ошибок.

Имитационное моделирование имеет свои сильные стороны:

• применимо даже в тех случаях, когда не для всех элементов системы найдено строгое математическое описание;

• при наличии достаточных вычислительных ресурсов может быть выполнено практически при любой степени детализации модели.

В то же время нужно понимать, что имитационное моделирование

• возникло не "от хорошей жизни", оно появилось потому, что иначе (проще и быстрее) решить задачу не удается;

• дает только количественные результаты для тех случаев, которые фактически моделировались; при этом ничего нельзя сказать о поведении системы при других условиях.

Примером может служить отладка программы для встроенного контроллера какого-нибудь устройства. Очень часто специальных средств отладки для нужного контроллера не существует. Однако, зная систему команд этого контроллера, несложно построить его компьютерную имитационную модель и отлаживать программу на компьютере общего назначения. Такой прием, называемый *кросс-программированием*, тоже можно считать вариантом имитационного моделирования.

Как правило, реальные сложные системы подвержены влиянию случайных факторов, поэтому используемые в этих случаях имитационные модели относятся к классу стохастических, или *вероятностных*, моделей. Одна из таких моделей — модель работы банка — рассматривается в учебнике информатики углубленного уровня [2–3], который вышел в 2013 году в издательстве "Бином". По мнению авторов этого учебника, очень полезно начинать изучение темы "Имитационное моделирование" не с использования готовых программных пакетов, а с "ручного" программирования, которое значительно лучше позволяет понять "что же там происходит".

В настоящей статье вероятностная модель из учебника [3] исследуется подробно с несколькими стадиями усложнения. Изложенный в статье материал далеко выходит за рамки учебника и может быть использован для стимулирования исследовательской работы заинтересованных школьников, например, в рамках элективных курсов.

Все работы можно выполнять в любой среде программирования на языке Паскаль, например, в *АЛГО* [4], *PascalABC.NET* [5], *FreePascal* [6] или *Delphi*. Все полные программы, рассмотренные в статье, можно найти в личном кабинете в электронных приложениях к этому номеру.

Модель работы банка

Предприятия сферы обслуживания (магазины, банки, узлы телефонной связи, станции "скорой помощи" и т.п.) — очень интересные, но очень сложные объекты. На вход таких систем поступает поток заявок на обслуживание — это могут быть, например, покупатели, входящие в магазин. Все заявки нужно обработать, для этого предприятие имеет несколько точек обслуживания (например, касс в магазине). При этом большую роль играет случайность:

• заявки поступают через случайные промежутки времени;

• время обслуживания — случайная величина.

Такие системы называют *системами массового обслуживания* [7]. Их теоретическое описание, как правило, достаточно сложно, поэтому часто применяют имитационные модели, которые позволяют найти решение задач с помощью многократных компьютерных экспериментов.

Рассмотрим простую модель работы банка. Клиенты входят через случайные промежутки времени, их обслуживают несколько кассиров, причем время обслуживания — также случайная величина. Будем считать, что все клиенты становятся в общую очередь, и при освобождении кассы начинается обслуживание того клиента, который стоит в этой очереди первым. Требуется определить, сколько кассиров нужно для того, чтобы клиент затратил на визит в банк не более M минут (с учетом обслуживания).

Сначала построим *детерминированную модель*, в которой случайность не учитывается. Будем считать, что за одну минуту входит ровно P клиентов, причем каждый клиент обслуживается ровно T минут.

Если количество касс равно K, то за T минут будет обслужено K клиентов. За это же время в банк войдут $P \cdot T$ новых клиентов. Несложно понять, что если $K < P \cdot T$, клиенты входят быстрее, чем их успевают обслуживать, поэтому очередь будет постоянно расти. Если же $K > P \cdot T$, очереди вообще не будет, из этого условия и нужно выбирать количество касс. Например, при $P = 2$ и $T = 2$ для обслуживания нужно не менее 10 касс.

Теперь усложним модель: введем в нее случайные события. Будем считать, что

• за одну минуту в банк входит случайное число клиентов, от 0 до P_{max};

• на обслуживание каждого клиента требуется от T_{min} до T_{max} минут.

Остается определить, как распределены случайные данные (количество входящих за одну минуту и время обслуживания) внутри заданных интервалов. Сначала для простоты мы будем считать, что в обоих случаях распределение *равномерное*. Это значит, что можно применить стандартные генераторы случайных чисел, которые дают именно равномерно распределенные значения.

Теперь построим имитационную модель и выполним моделирование ситуации для достаточно большого интервала времени L (например, для 8-часового рабочего дня $L = 8 \cdot 60 = 480$ минут).

Моделирование выполняется в дискретном времени с постоянным шагом, единица времени — одна минута. Число клиентов, находящихся в помещении банка, изменяется по закону

$$N_{i+1} = N_i + P_i - R_i$$

где N_i — количество клиентов в банке в i-ю минуту; P_i — количество клиентов, вошедших за i-ю минуту, а R_i — количество клиентов, обслуженных за эту минуту. Если считать, что N клиентов равномерно распределяются по K кассам, получаем, что средняя длина очереди в одну кассу равна $q = \frac{N}{K}$, а среднее время ожидания равно $q \cdot T = \frac{N \cdot T}{K}$.

Количество вошедших P_i — это случайное целое число в интервале от 0 до P_{max}. На языке Паскаль его можно получить так:

```
P := random(PMax + 1);
```

Определить число обслуженных R_i оказывается не так просто. Если кассир обслуживает клиента за T минут, то можно считать, что за одну минуту он сделает часть работы, равную $\frac{1}{T}$. Если предположить, что скорость работы кассиров одинакова, то K касс за одну минуту обслужат $\frac{K}{T}$ клиентов. Чтобы учесть случайное время обслуживания, величину T будем определять заново для каждой минуты случайным образом. Это случайное вещественное число в интервале от T_{min} до T_{max}:

```
T := Tmin + random * (Tmax – Tmin);
```

Тогда среднее время ожидания (с учетом времени обслуживания)

$$\Delta t = q \cdot T = \frac{N}{K} \cdot T$$

будет меняться, поскольку N и T — случайные величины. Именно поэтому невозможно *гарантировать*, что клиент *точно* не будет ждать больше положенного времени M. Вместо этого мы можем (с помощью имитационного моделирования) подсчитать, какую *долю времени* период ожидания Δt будет больше, чем допустимое значение M. Для определения этой доли нужно подсчитать количество "плохих" минут и разделить его на общее время моделирования L.

Если доля "плохого" времени получилась, например, 0,95 (или 95%), то клиент практически всегда вынужден будет ждать слишком долго, и количество касс нужно увеличивать. Если эта доля равна 0,05, то время ожидания будет больше допустимого только в 5% случаев, и такой результат можно считать приемлемым.

Таким образом, нужно составить программу, которая запрашивает количество касс, выполняет моделирование работы банка в течение рабочего дня и выводит долю "плохих" минут. Задаем начальные значения переменных:

```
{ максимальное число входящих за 1 мин. }
Pmax := 4;
{ минимальное время обслуживания }
Tmin := 1;
{ максимальное время обслуживания }
Tmax := 9;
{ период моделирования в минутах (8 часов)}
L := 480;
M := 15;    { допустимое время ожидания }
N := 0;     { сначала в банке никого нет }
Count := 0; { счетчик "плохих" минут }
```

Затем нужно ввести с клавиатуры количество касс K:

```
  write('Введите количество касс: ');
  readln(K);
```

Основной цикл моделирует работу банка в течение L минут:

```
for i := 1 to L do begin
  P := random(PMax + 1);
{ равномерное распределение }
  T := Tmin + random * (Tmax - Tmin);
{ равномерное распределение }
  R := round(K / T);
{ число обслуженных }
  N := N + P - R;
{ число клиентов в банке }
  if N < 0 then N := 0;
  dT := N / K * T;
{ среднее затраченное время }
  if dT > M then Inc(count)
end;
```

На каждом шаге последовательно вычисляются

• случайное число входящих клиентов P;

• случайное время обслуживания T;

• число клиентов R, обслуженных за эту минуту;

• число клиентов в помещении банка N; если оно получилось отрицательным, то используется нулевое значение;

• время ожидания; если оно больше допустимого времени M, увеличивается счетчик "плохих" минут.

После окончания работы цикла остается вывести результат — отношение `count/L`, которое представляет собой долю "плохих" минут. Нужно провести серию экспериментов с моделью и выбрать минимальное значение K, при котором доля

“плохих” минут будет менее 5%. Помните, что в модели используются случайные числа, поэтому при каждом новом запуске программы результаты расчетов будут немного меняться. Для того чтобы результаты были более стабильными, значение L лучше увеличить, скажем, до 480 000 (если при этом время моделирования не становится недопустимо большим).

Выполнив моделирование в среде PascalABC.NET при различных значениях K для L = 480 000, автор получил следующие результаты:

Рис. 1

— из которых следует, что для обеспечения допустимого времени ожидания достаточно 9 касс.

Обратите внимание, что мы выбрали исходные данные так, чтобы среднее число входящих клиентов и среднее время обслуживания совпадали с теми же данными для детерминированной модели. Поэтому интересно сравнить результаты, полученные с помощью моделей двух типов: детерминированной и вероятностной. Как вы помните, с помощью детерминированной модели мы получили минимальное значение K = 10, что близко к результатам проведенного вероятностного моделирования.

Распределение Пуассона

В рассмотренной вероятностной модели мы предполагали, что количество входящих за одну минуту распределено равномерно в диапазоне $[0, P_{max}]$. На самом деле, согласно теории массового обслуживания [7], количество заявок, поступающих за минуту, описывается *распределением Пуассона*. Это распределение характеризует дискретную случайную величину, принимающую целые неотрицательные значения. Вероятность того, что эта случайная величина равна заданному числу k, вычисляется по формуле

$$p(k) = \frac{\lambda^k}{k!} \cdot e^{-\lambda}. \qquad (*)$$

Здесь λ — среднее значение случайной величины, и $k! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot k$ обозначает факториал числа k. Гистограмма, иллюстрирующая распределение Пуассона для $\lambda = 1$, показана на *рис.* 2.

Как же получить на компьютере последовательность псевдослучайных чисел с таким распределением? Как правило, в современных системах программирования есть встроенный генератор

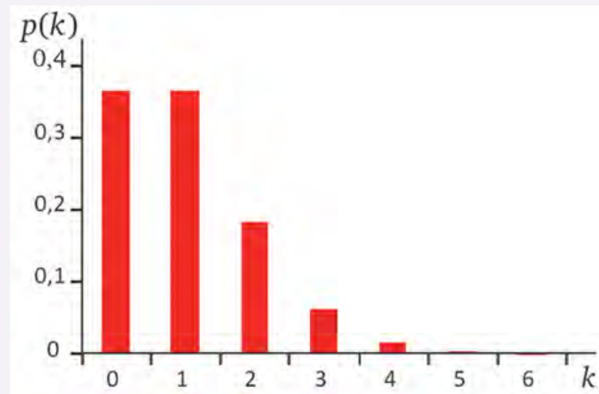

Рис. 2

псевдослучайных чисел с равномерным распределением. Оказывается, с помощью математических методов из такой последовательности чисел можно (теоретически) получить последовательность чисел с *любым* желаемым распределением. Для этого существует несколько различных методов, наиболее популярный из них — метод *обратной функции*. Ниже мы кратко изложим основные идеи этого метода и покажем, как применить его для получения случайных чисел, “распределенных по Пуассону”.

Одна из универсальных характеристик любого распределения случайных чисел — функция распределения $F(x)$, которая определяется для любого вещественного x как вероятность того, что случайная величина будет меньше или равна x. Функция $F(x)$ непрерывна, она равна нулю при $x \to -\infty$ и стремится к единице при $x \to \infty$.

Вспомним, что распределение Пуассона описывает целые неотрицательные значения, то есть $F(x) = 0$ для $x < 0$. Вероятность того, что $x = 0$ — ненулевая, она определяется по формуле, которая следует из (*):

$$p(0) = \frac{\lambda^0}{0!} \cdot e^{-\lambda} = e^{-\lambda} \qquad (**)$$

Это означает, что при $x = 0$ функция распределения $F(x)$ изменяется скачком с нуля до значения $e^{-\lambda}$. Аналогичные скачки будут происходить при всех целых значениях x, тогда как при всех нецелых x функция $F(x)$ сохраняет постоянное значение. Таким образом, для любого дискретного распределения функция распределения получается ступенчатой. На *рис.* 3 показана функция распределения Пуассона для среднего значения $\lambda = 1$.

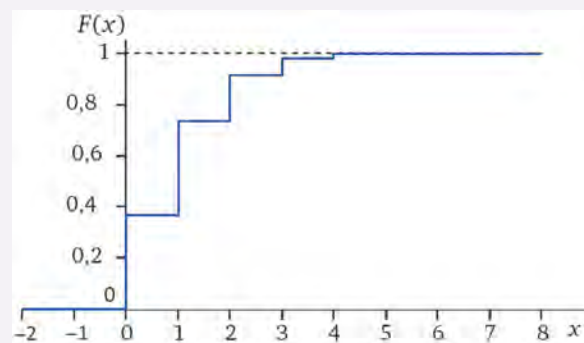

Рис. 3

Обратите внимание, что $F(x) \neq 0$ при $x = 0$, как и следует из формулы (**).

Вспомним, что мы хотим получить последовательность псевдослучайных чисел, соответствующую этой функции распределения. Предположим, что мы имеем генератор псевдослучайных вещественных чисел с равномерным распределением на интервале (0;1). Получив с помощью этого датчика очередное (псевдо-)случайное число z, нужно отложить это значение на вертикальной оси, провести горизонтальный отрезок до пересечения с графиком функции $F(x)$ и затем опустить вертикально вниз перпендикуляр к оси абсцисс. Значение x, полученное таким образом, и есть нужное нам случайное число. Очевидно, что в нашем случае все получаемые таким образом значения $k = x$ будут целыми неотрицательными числами. На *рис.* 4 графически показана эта процедура для $z = 0{,}5$.

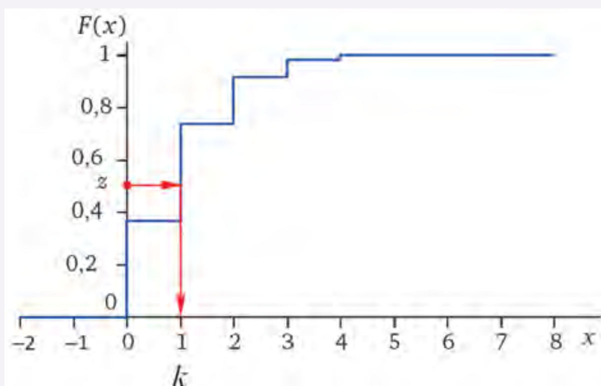

Рис. 4

Как же доказать, что новая последовательность будет иметь распределение Пуассона? Для дискретной случайной величины это сделать очень просто. По графику на *рис.* 4 видно, что вероятность получить $k = 0$ равна (с учетом равномерного распределения величины z) величине скачка функции $F(x)$ при $x = 0$, то есть $p(0)$. Аналогично вероятность получить $k = 1$ равна величине скачка $F(x)$ при $x = 1$, то есть $p(1)$, и т.д.

Теперь остается запрограммировать процедуру для перехода от случайной величины z к случайному целому числу k. Начинаем накапливать значение $F(x)$, начиная с $x = 0$. Если выполняется условие[1]

$$z \le F(0+\varepsilon) = p(0),$$

то сразу получаем $k = 0$. Если это условие не выполняется, проверяем второе условие

$$z \le F(1+\varepsilon) = p(0) + p(1)$$

при выполнении которого $k = 1$, и т.д. Процедура на языке Паскаль, использующая такой подход и формулу (*) для вычисления $p(k)$, приводится ниже. Здесь параметр функции **Lam** — это *среднее* количество клиентов, входящих за одну минуту:

[1] Здесь $F(0 + \varepsilon)$ означает предел значения функции $F(x)$ при стремлении x к нулю справа.

```
{-------------------------------------
  POISSON
  Генератор псевдослучайных целых чисел
  с распределением Пуассона.
  Вход: Lam — среднее число заявок за единицу времени
  Выход: случайное число заявок
-------------------------------------}
function Poisson(Lam: integer): integer;
var s, r, z: real;
    k: integer;
begin
  r := exp(-Lam); s := r; k := 0;
  z := random;
  while s < z do begin
    k := k + 1;
    r := r * Lam / k;
    s := s + r
  end;
  Poisson := k
end;
```

Выполним моделирование, немного изменив основную программу: теперь в основном цикле количество входящих клиентов за одну минуту определяется с помощью функции **Poisson**, параметром которой служит среднее значение, то есть $P_{max}/2$:

```
for i := 1 to L do begin
  P := Poisson(PMax div 2);
  ...
end;
```

Результат моделирования может показаться неожиданным: мы получим практически ту же самую кривую, что и на *рис.* 1. Это означает, что допущение о равномерности распределения количества входящих *в данной задаче* может быть использовано, поскольку не искажает результат в сравнении с более точной моделью, использующей распределение Пуассона. Тем не менее нужно понимать, что при других исходных данных этот вывод может оказаться неверным, на то это и имитационная модель. Поэтому лучше всегда использовать наиболее точную из имеющихся моделей.

Новая модель обслуживания

Еще одно слабое звено нашей модели — определение числа клиентов, обслуженных за текущую минуту. Ранее предполагалось, что производительность всех касс меняется одинаково каждую минуту, при этом случайное время обслуживания равномерно распределено в интервале от T_{min} до T_{max}. Скорее всего эта модель далека от реальности.

Используем другую модель обслуживания: все клиенты, находящиеся в банке, делятся на две группы: тех, кто в данный момент обслуживается, и тех, кто ждет своей очереди. Будем хранить общую длину очереди в переменной **NWait**. Для каждой кассы запомним время, оставшееся до окончания обслуживания текущего клиента, в массиве

```
var ServiceTime: array[1..100] of integer;
```

Предполагается, что время обслуживания — это целое число минут, и количество касс K не превышает 100.

Когда **P** клиентов в течение очередной минуты входят в банк, они сразу становятся в очередь, то есть значение переменной **NWait** увеличивается на **P**:

```
  NWait := NWait + P;
```

Каждую минуту все кассы выполняют очередную часть обслуживания своих клиентов, в результате некоторые кассы могут освободиться. В программе эту работу будет выполнять процедура **DoService**:

```
{----------------------------------------
DO SERVICE
Очередной цикл обслуживания за единицу
времени.Некоторые кассы могут освободиться.
----------------------------------------}
procedure DoService;
var i: integer;
begin
  for i := 1 to K do
    if ServiceTime[i] > 0 then
      Dec(ServiceTime[i])
end;
```

В этой процедуре, которая вызывается на каждом шаге моделирования, оставшееся время для всех *K* касс уменьшается на единицу.

После того как кассы выполнили очередной шаг обслуживания, нужно попытаться распределить клиентов, стоящих в очереди, по свободным кассам. Для этой цели используем процедуру **StartService**:

```
{----------------------------------------
START SERVICE
Распределение очереди по свободным кассам.
----------------------------------------}
procedure StartService;
var i: integer;
begin
  if NWait > 0 then
    for i := 1 to K do
      if ServiceTime[i] = 0 then begin
        ServiceTime[i] :=
        Tmin + random(Tmax - Tmin + 1);
        Dec(NWait);
        if NWait = 0 then break
      end
end;
```

Если очередь не пуста, мы просматриваем все кассы и пытаемся найти освободившуюся, у которой оставшееся время обслуживания равно нулю. Затем определяем случайное время обслуживания и записываем его в массив **ServiceTime**. В этом варианте процедуры предполагается, что время обслуживания — это случайная целая величина с равномерным распределением на отрезке $[T_{min}; T_{max}]$.

Теперь надо определить среднее время ожидания для только что вошедших клиентов. С учетом времени обслуживания, оно складывается из двух составляющих:

• времени обслуживания тех, кто уже находится у касс (массив **ServiceTime**);

• ожидаемого времени обслуживания для всех клиентов, стоящих в очереди, которое можно оценить как **NWait*Tcp**, где **Tcp** — среднее время обслуживания.

Полученное общее время нужно разделить на количество касс *K*.

С учетом всех приведенных выше рассуждений, основной цикл программы будет выглядеть так:

```
Count := 0;
NWait := 0;
Tcp := (Tmax + Tmin) div 2;
for i := 1 to L do begin
  P := Poisson(Pmax div 2);
{ распределение Пуассона }
  NWait := NWait + P;
  DoService;
{ работа касс }
  StartService;
{ распределение очереди }
  sumT := 0;
  for j := 1 to K do
{ оставшееся время обслуживания работающих
  касс}
    sumT := sumT + ServiceTime[j];
  dT := (NWait * Tcp + sumT) / K;
  if dT > M then Inc(count)
end;
```

Напомним, что результатом работы программы при заданном количестве касс *K* будет доля "плохих" минут — отношение **count/L**, которое для нормальной работы должно быть не более 5%, или 0,05.

Моделирование с помощью нового варианта программы (обозначим его как "вариант 3") приводит к следующим результатам:

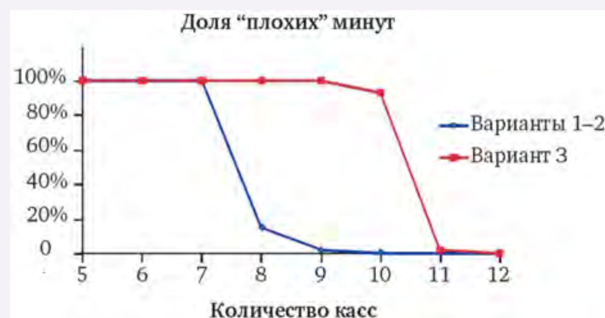

Рис. 5

Видим, что ситуация несколько изменилась: по вновь полученному графику (красная линия на *рис.* 5) можно сделать вывод, что количество касс должно быть не менее 11.

Экспоненциальное распределение

Несмотря на то что последняя модель обслуживания уже намного более реалистичная, чем первая, в ней осталось сильное допущение о том, что случайное время обслуживания имеет *равномерное* распределение. В теории массового обслуживания [7] обычно предполагают, что распределение этой величины *экспоненциальное*[2] (или *показательное*). Экспоненциальное распределение описывается *плотностью вероятности*:

---

[2] Нужно отметить, что это предположение связано в значительной степени с тем, что именно для этого случая удается получить аналитические решения многих задач (см. [1,7]).

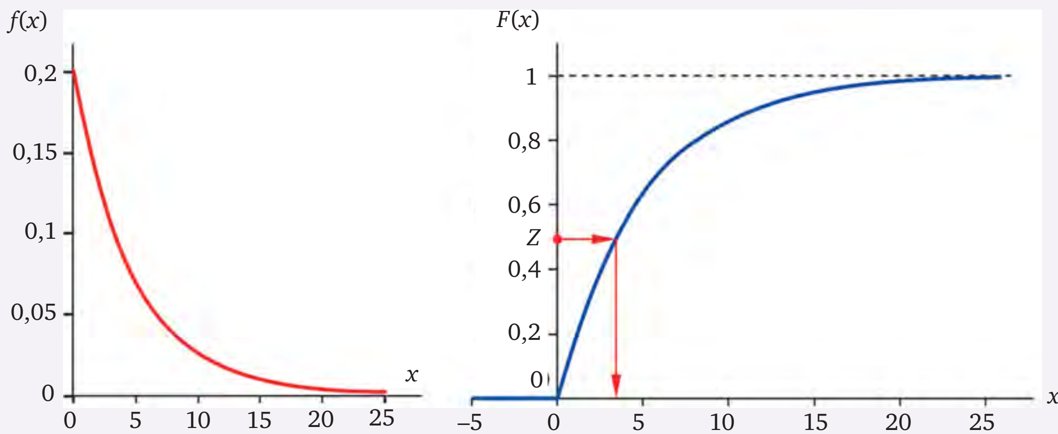

Рис. 6

$$f(x)=\begin{cases}\mu e^{-\mu x}, & x\ge 0\\ 0, & x<0\end{cases}$$

и соответствующей функцией распределения

$$F(x)=\begin{cases}1-e^{-\mu x}, & x\ge 0\\ 0, & x<0\end{cases}$$

В этих формулах параметр μ — это величина, обратная к среднему времени обслуживания $T_{ср}$, то есть $\mu=\frac{1}{T_{cp}}$. На *рис.* 6 показаны графики плотности вероятности и функции распределения для $T_{ср} = 5$.

Для того чтобы получить последовательность случайных чисел с экспоненциальным распределением, можно использовать метод обратной функции, который мы уже обсуждали выше (см. *рис.* 3 на с. 7). Фактически нам нужно найти обратную функцию для $F(x)$. В данном случае это легко сделать аналитически из равенства:

$$z=1-e^{-\mu x},$$

откуда сразу следует, что

$$x=-\frac{1}{\mu}\ln(1-z)=-T_{cp}\ln(1-z).$$

Таким образом, получив случайную величину z с равномерным распределением на интервале (0;1), мы можем найти соответствующее значение x, причем последовательность этих чисел будет иметь экспоненциальное распределение.

Учитывая, что последовательность значений $1 - z$ также имеет равномерное распределение на (0;1), для преобразования обычно используют формулу $x=-T_{cp}\ln z$. Поскольку мы договорились, что время обслуживания составляет целое число минут, полученное таким образом значение нужно округлить вверх (до ближайшего большего целого числа). Функция, которая выдает очередное случайное время обслуживания, может выглядеть так:

```
{---------------------------------------
  FULL SERVICE TIME
  Случайное время обслуживания
  (экспоненциальное распределение)
---------------------------------------}
function  FullServiceTime(Tcp:  integer):
integer;
var z: real;
begin
  z := random;
  FullServiceTime := Ceil(-ln(z) * Tcp)
end;
```

Параметр функции — это среднее время обслуживания. Функция `Ceil` выполняет округление вверх, в среде *FreePascal* для использования этой функции нужно подключить модуль *Math*.

Теперь изменим в процедуре `StartService` одну строку, где определяется случайное время обслуживания для очередного клиента:

```
  ...
ServiceTime[i] := FullServiceTime(Tcp);
  ...
```

Выполнив моделирование с учетом этих изменений, получаем новый график (он обозначен на *рис.* 7 как “вариант 4”):

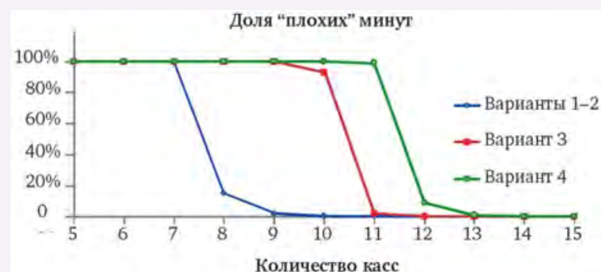

Рис. 7

По графику (см. зеленую линию на *рис.* 7) определяем, что количество касс должно быть не менее 13.

Распределение Эрланга

Как следует из *рис.* 6, используя экспоненциальное распределение для оценки времени обслуживания, мы считаем, что чаще всего клиенты обслуживаются очень быстро — плотность вероятности максимальна при $x = 0$. Как уже отмечалось [1,7], это допущение в теории массового обслуживания принимается главным образом из-за (относительной) простоты аналитического исследования. В реальной ситуации существует

некоторое “наиболее ожидаемое” время обслуживания, не равное нулю. Это справедливо, в частности, когда процедура обслуживания складывается из нескольких операций со случайным временем выполнения. Поэтому для уточнения модели часто используют распределение Эрланга, которое имеет плотность вероятности

$$f(x)=\begin{cases}\dfrac{\mu^k x^{k-1} e^{-\mu x}}{(k-1)!}, & x \ge 0 \\ 0, & x<0\end{cases}$$

Параметр k можно рассматривать как количество операций при обслуживании клиента. Для того чтобы обеспечить заданное среднее время обслуживания T_{cp}, значение μ вычисляется по формуле $\mu = \dfrac{k}{T_{cp}}$. Легко доказать, что экспоненциальное распределение — это частный случай распределения Эрланга[3] при $k = 1$.

На *рис.* 8 показана плотность вероятности для распределения Эрланга при $k = 2$ и среднем времени обслуживания $T_{cp} = 5$.

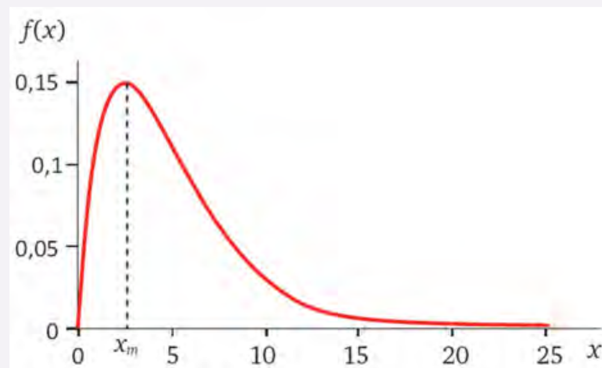

Рис. 8

Наиболее вероятное время обслуживания (точка максимума этой функции) вычисляется по формуле

$$x_m = \frac{k-1}{\mu} = \frac{k-1}{k} \cdot T_{cp}.$$

В данном случае получаем $x_m = \dfrac{T_{cp}}{2} = 2{,}5$, а при увеличении k это значение стремится к T_{cp}.

Для того чтобы генерировать последовательность случайных чисел с распределением Эрланга, достаточно на каждом шаге вычислять среднее арифметическое k независимых значений, имеющих экспоненциальное распределение с тем же параметром μ. Таким образом, функция, определяющая случайное время обслуживания, приобретает следующий вид:

```
{---------------------------------------
  FULL SERVICE TIME
  Случайное время обслуживания
  (распределение Эрланга с параметром k)
---------------------------------------}
function FullServiceTime(Tcp, k: integer):
integer;
var z: real;
    i: integer;
begin
  z := 0;
  for i := 1 to k do
    z := z - ln(random);
  FullServiceTime := Ceil(z * Tcp / k)
end;
```

Остальные части программы не изменяются.

Выполнив моделирование с помощью нового варианта программы, обнаружим, что результаты в нашем случае мало изменились в сравнении с вариантом 4.

Дискретно-событийное моделирование

До этого момента мы выполняли моделирование с постоянным шагом по времени и предполагали, что время обслуживания составляет целое число минут. Однако в профессиональных средствах имитационного моделирования систем такой подход практически не применяется. Вместо этого чаще всего используют *дискретно-событийное моделирование* [1], при котором предполагается, что

- система меняет свое состояние только в отдельные моменты времени (дискретно);
- эти моменты времени совпадают с *событиями*, происходящими в системе.

Вернемся снова к нашей модели работы банка. С точки зрения рассмотренной задачи состояние системы определяется двумя значениями:

- количеством клиентов, стоящих в очереди (его мы храним в переменной **NWait**), и
- количеством занятых касс (или, что то же самое, количеством клиентов, которые в данный момент обслуживаются), — для хранения этого значения введем новую целую переменную **NBusy**.

В системе происходят два типа событий: вход очередного клиента и окончание обслуживания клиента на одной из касс, они могут произойти в любой момент времени. Модель меняет состояние только тогда, когда в системе произошло какое-то событие, при этом модельное время “продвигается” вперед до момента этого события.

Для работы с событиями в программе введем следующие типы данных:

```
type
  TEventType = (evIn, evOut);
  TEvent = record
    time: real;
    evType: TEventType;
  end;
```

Тип **TEventType** — это тип события: **evIn** означает вход нового клиента, а **evOut** — окончание обслуживания и выход клиента. Объект типа **TEvent** — это структура (запись), содержащая два поля: планируемое время события **time** и тип этого события **evType**.

[3] Напомним, что 0! = 1.

Организуем *очередь событий*, в которой предстоящие события расставлены в порядке возрастания времени, так что ближайшее событие стоит в голове очереди. Соответствующая структура данных называется *очередь с приоритетом* [8], ее элементами будут записи типа **TEvent**, а приоритетом — время (поле **time**). Рамки статьи не позволяют углубляться в детали реализации такой очереди, нам важно только то, что она поддерживает операции “добавить событие” и “получить событие с наивысшим приоритетом” (с наименьшим значением поля **time**). В программе эти операции будут выполнять подпрограммы, имеющие такие заголовки[4]:

```
{ добавить событие }
procedure AddToQueue(E: TEvent);
{ получить событие из очереди }
function GetFromQueue: TEvent;
```

Основной цикл программы моделирования будет выглядеть так:

```
{ количество клиентов в очереди }
NWait := 0;
NBusy := 0;     { количество занятых касс }
symTime := 0;  { модельное время }
NewClient;     { вход первого клиента }
while symTime < L do begin
  { выбрать событие из очереди }
  E := GetFromQueue;
  HandleEvent(E);    { обработать событие }
end;
```

Здесь вещественная переменная **symTime** — это *модельное время*; процедура **NewClient** записывает в очередь событие прихода очередного (в данном случае — первого) клиента, а процедура **HandleEvent** “продвигает” модельное время и обрабатывает очередное событие, взятое из очереди. Цикл заканчивается, когда модельное время станет не меньше, чем заданное общее время моделирования **L**.

Теперь займемся реализацией новых процедур. Процедура **NewClient** создает новое событие типа **evIn** (вход нового клиента), определяет случайное время этого события и записывает событие в очередь:

```
{---------------------------------------
  NEW CLIENT
 Планирование времени входа очередного
 клиента.
---------------------------------------}
procedure NewClient;
var E: TEvent;
begin
  E.evType := evIn;
  E.time := symTime + ArriveTime;
  AddToQueue(E);
end;
```

Время события (поле **E.time**) — это сумма текущего модельного времени и случайного интервала между входом двух клиентов. В теории массового обслуживания [1,7] обычно предполагают, что этот интервал имеет экспоненциальное распределение, поэтому функцию **ArriveTime** можно записать так:

```
{---------------------------------------
  ARRIVE TIME
  Случайное время между входом клиентов.
---------------------------------------}
function ArriveTime: real;
begin
  ArriveTime := -ln(random) / Pcp;
end;
```

В глобальной переменной **Pcp** хранится среднее количество клиентов, входящих в банк за одну минуту.

Нам будет нужна еще одна процедура, которая выбирает из очереди очередного клиента и направляет его в освободившуюся кассу:

```
{---------------------------------------
  START SERVICE
  Начало обслуживания очередного клиента.
---------------------------------------}
procedure StartService;
var E: TEvent;
begin
  if (NWait > 0) and (NBusy < K) then begin
    NBusy := NBusy + 1;
    NWait := NWait – 1;
    E.evType := evOut;
    E.time := symTime + FullServiceTime(Tcp, 2);
    AddToQueue(E);
  end;
end;
```

Эта процедура срабатывает, когда число ожидающих в очереди **NWait** больше нуля и количество занятых касс **NBusy** меньше их общего количества **K** (есть свободные кассы). В этом случае число занятых касс увеличивается, число стоящих в очереди уменьшается, а в очередь записывается новое (планируемое) событие типа **evOut** (клиент освободился). Время обслуживания вычисляется с помощью функции **FullServiceTime**, которая генерирует случайную последовательность, имеющую распределение Эрланга с заданными средним значением **Tcp** и параметром **k**:

```
{---------------------------------------
  FULL SERVICE TIME
  Случайное время обслуживания
  (распределение Эрланга порядка k)
---------------------------------------}
function FullServiceTime(Tcp, k: integer): real;
var
  z: real;
  i: integer;
begin
  z := 0;
  for i := 1 to k do
    z := z – ln(random);
  FullServiceTime := z * Tcp / k;
end;
```

Новая версия этой функции в отличие от предыдущей возвращает вещественное число (без округления).

Теперь мы готовы написать процедуру **HandleEvent**:

[4] Полный текст программы моделирования, где используется один из простейших вариантов реализации такой очереди, приведен на диске в приложении к этому номеру.

```
{-------------------------------------
  HANDLE EVENT
  Обработка очередного события
-------------------------------------}
procedure HandleEvent(E: TEvent);
begin
{ продвинуть модельное время }
  symTime := E.time;
  case E.evType of    { выбор типа события }
   evIn:             { вход нового клиента }
     begin
       NewClient;
       NWait := NWait + 1;
     end;
   evOut:     { завершение обслуживания }
     NBusy := NBusy – 1;
  end;
  StartService;
end;
```

Эта процедура продвигает модельное время до времени переданного ее события. Затем, в зависимости от типа события, либо добавляем в очередь нового клиента, либо уменьшаем количество занятых касс. После этого вызывается процедура **StartService**, которая при наличии свободных касс пытается начать обслуживание клиентов из очереди.

Как же нам оценить работу этой модели? Наиболее простой вариант — использовать для оценки среднюю длину очереди в одну кассу $\bar{q}$, которая вычисляется по формуле:

$$\bar{q} = \frac{1}{K} \cdot \frac{1}{T} \int_0^T Q(t)dt,$$

где $Q(t)$ — текущая длина общей очереди в момент времени t, K — количество касс, а T — общее время моделирования. Поскольку состояние системы между моментами событий не меняется, функция $Q(t)$ — кусочно-постоянная:

Рис. 9

На *рис. 9* приведена диаграмма, показывающая изменение длины общей очереди $Q(t)$. В моменты времени t_2 и t_3 очередь увеличивается на единицу при входе клиентов. В моменты t_1, t_4, t_5 и t_6 заканчивается обслуживание очередного клиента, и длина очереди уменьшается. Как следует из графика, расчет $\bar{q}$ может быть выполнен достаточно просто:

- перед началом основного цикла переменная **Qcp** обнуляется;
- сразу после выборки очередного события из очереди добавляем очередную “порцию интеграла” (площадь прямоугольника) к значению этой переменной:

```
Qcp := Qcp + (E.Time – symTime) * NWait;
```

Выражение в скобках представляет собой промежуток времени от предыдущего события до текущего (только что выбранного), а значение переменной **NWait** равно длине очереди в этот период;

- после окончания цикла разделим значение переменной **Qcp** на общее время моделирования, которое хранится в переменной **symTime**, и количество касс **K**:

```
Qcp := Qcp / symTime / K;
```

Это и будет средняя длина очереди в одну кассу.

Цикл в основной программе приобретает такой вид:

```
while symTime < L do begin
  E := GetFromQueue;
{ выбрать событие из очереди }
  Qcp := Qcp + (E.Time – symTime)*NWait;
  HandleEvent(E);
end;
Qcp := Qcp / symTime / K;
```

Полную программу для моделирования читатели могут найти на диске в приложении к этому номеру. Для исходных данных, использованных при моделировании в предыдущих вариантах программы, мы получаем, что при $K < 11$ длина очереди недопустимо велика (она может превышать 2000 человек, что говорит о явной перегрузке), а при $K = 11$ банк успевает обслуживать всех входящих клиентов (средняя длина очереди меньше 1). Таким образом, согласно этой модели, нужно 11 касс.

Дополнительные критерии

Кроме средней длины очереди, для оценки качества работы системы массового обслуживания часто используют еще две характеристики: среднее время ожидания $\bar{W}$ (до начала обслуживания) и средний коэффициент использования (или эффективность) касс $\bar{b}$. Добавим в программу вычисление этих величин.

Время ожидания для клиента — это разница во времени между моментом начала обслуживания и моментом входа в банк. Для вычисления этой величины нужно где-то хранить время входа для каждого клиента, стоящего в очереди. Выделим для этой цели массив **timeIn**, предполагая, что длина очереди не будет больше, чем **MAXWAIT**:

```
const MAXWAIT = 10000;
var timeIn: array[1..MAXWAIT] of real;
```

Как вы помните, в любой момент количество клиентов в очереди хранится в переменной **NWait**, при этом первые **NWait** элементов массива **timeIn** будут содержать время входа для всех клиентов, ожидающих обслуживания.

Если прибыл очередной клиент, он ставится в очередь. В процедуре **HandleEvent** обработка события **evIn** теперь запишется так:

```
NewClient;
NWait := NWait + 1;
timeIn[NWait] := symTime;
```

В последней строке в первый свободный элемент массива **timeIn** записывается время прибытия нового клиента, которое совпадает с текущим модель-

ным временем. Конечно, сюда нужно добавить проверку на переполнение очереди, но мы оставим это читателю в качестве упражнения.

В переменной **Wcp** будем накапливать суммарное время ожидания для всех клиентов, которое после окончания моделирования разделим на число обслуженных клиентов, — это и будет среднее время ожидания.

Когда очередной клиент из очереди направляется в освободившуюся кассу, мы добавляем к переменной **Wcp** разницу между текущим временем и временем его прибытия (время ожидания), увеличиваем счетчик обслуженных клиентов **count** и сдвигаем все элементы массива **timeIn** на одну позицию к началу массива[5]:

```
Wcp := Wcp + symTime — timeIn[1];
Count := count + 1;
for i := 1 to NWait - 1 do
  timeIn[i] := timeIn[i + 1];
```

В конце программы вычисляется среднее время ожидания:

```
  Wcp := Wcp / count;
```

Остается найти коэффициент использования касс $\bar{b}$. Обозначим через $B(t)$ количество занятых касс в момент времени t. Тогда $\bar{b}$ вычисляется как среднее значение аналогично $\bar{q}$:

$$\bar{b} = \frac{1}{K} \cdot \frac{1}{T} \int_0^T B(t)dt.$$

Поскольку состояние системы между моментами событий не меняется, функция $B(t)$ — кусочно-постоянная, и расчет $\bar{b}$ может быть выполнен следующим образом:

- перед началом основного цикла переменная **Bcp** обнуляется;
- сразу после выборки очередного события из очереди добавляем “порцию интеграла” (площадь прямоугольника) к значению этой переменной:

```
  Bcp := Bcp + (E.Time — symTime) * NBusy;
```

- после окончания цикла разделим значение переменной **Bcp** на общее время моделирования, которое хранится в переменной **symTime**, и количество касс **K**:

```
  Bcp := Bcp / symTime / K;
```

Это и будет средняя эффективность касс.

В таблице приводятся результаты моделирования при различных значениях K (для $L = 100\,000$):

Количество касс, K	9	10	11	12
Средняя длина очереди, $\bar{q}$	107	7	0,5	0,15
Среднее время ожидания, $\bar{W}$	492	37	2,5	0,88
Средняя эффективность касс, $\bar{b}$	1,00	0,99	0,91	0,84

Эти данные показывают, что для обеспечения допустимого времени ожидания клиентов нужно установить 11 касс. При этом эффективность работы касс составляет 0,91, что вполне приемлемо.

[5] Это самый простой, но неэффективный способ организации очереди. Лучше использовать массив, замкнутый в кольцо (см. [3]), чтобы не перемещать элементы.

Модель с отказами

Во всех предыдущих моделях мы предполагали, что клиенты банка — люди настойчивые и целеустремленные до такой степени, что они всегда встают в очередь и терпеливо ждут начала обслуживания, независимо от длины этой очереди. На практике это, конечно, не так. Действительно, увидев очередь в кассу из 10 человек, бо́льшая часть реальных клиентов развернется и уйдет, выразив про себя или вслух недовольство работой банка. Таким образом, часть заявок останутся необслуженными — мы получили так называемую *модель с отказами* (или модель с потерями).

Пусть в момент входа клиента средняя длина очереди в одной кассе равна q. Для моделирования ситуации отказа вычислим вероятность отказа $p(q)$, используя следующие идеи:

- если очереди нет, клиент никогда не отказывается от обслуживания, то есть $p(0) = 0$;
- при возрастании длины очереди вероятность отказа повышается и асимптотически стремится к 1 при $q \to \infty$;
- при $q = 2$ около 60% клиентов отказываются от стояния в очереди и уходят.

Этим условиям удовлетворяет, например, функция $p(q) = 1 - e^{-0,5q}$, график которой показан на *рис.* 10.

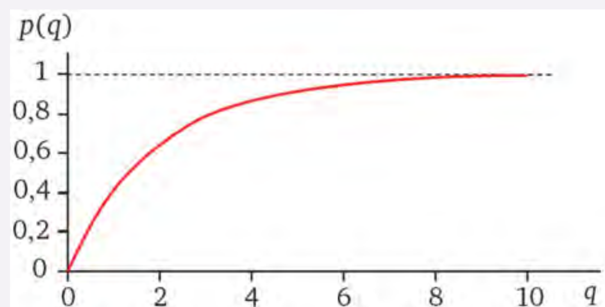

Рис. 10

Заметим, что средняя длина очереди q в одну кассу может быть вычислена как отношение общей длины очереди (переменная **NWait**) к количеству касс **K**. Перепишем соответствующим образом процедуру **HandleEvent**:

```
{----------------------------------------
  HANDLE EVENT
  Обработка события. Модель с отказами.
----------------------------------------}
procedure HandleEvent(E: TEvent);
begin
  symTime := E.time;
  case E.evType of
    evIn:
      begin
        NewClient;
        { отказ }
        if random <
           1-exp(-0.5 * NWait / K) then
          count0 := count0 + 1
        else begin
          NWait := NWait + 1;
          timeIn[NWait] := symTime;
        end;
      end;
```

```
    evOut:
      NBusy := NBusy - 1;
  end;
  StartService;
end;
```

Здесь переменная **count0** — это счетчик отказов. Отказ происходит тогда, когда случайное число, полученное при вызове функции **random**, меньше, чем вероятность отказа $p(q) = 1 - e^{-0,5q}$.

Для того чтобы после окончания моделирования вычислить долю необслуженных заявок (долю отказов, долю потерянных клиентов), нужно разделить **count0** на общее количество вошедших клиентов, которое равно **count + count0** (напомним, что в переменной **count** хранится число обслуженных клиентов).

Выполнив моделирование после этих изменений, получаем следующие результаты (*рис.* 11).

В данной ситуации длина очереди $\bar{q}$ и связанное с ней среднее время ожидания $\bar{W}$ ничего не говорят об эффективности работы банка. На первый план выходят два других показателя — эффективность работы касс и доля отказов. Они противоречат друг другу в том смысле, что при увеличении количества касс доля отказов уменьшается (это хорошо!), но зато падает и эффективность работы касс, что на практике означает лишние расходы. Поэтому приходится искать компромиссное решение. Пусть, например, наиболее важна доля отказов, и она должна быть не более 10%. По последнему графику определяем, что нужно выбрать $K = 10$ (как в детерминированной модели!), при этом эффективность работы касс оказывается достаточно высокой — около 90%.

Построенную имитационную модель можно уточнять и дальше. Например, полезно предусмотреть возможность отказа оборудования касс, перерывы на обед (когда некоторые кассы закрываются) и т.д. Эти изменения читатели уже могут сделать самостоятельно.

Заключение

Подведем некоторые итоги. Для решения задачи мы использовали одну детерминированную модель (согласно которой требуется не менее 10 касс) и несколько вероятностных, которые привели к различным ответам. Все результаты достаточно близки, то есть в данной задаче при выбранных параметрах можно было использовать даже простейшую детерминированную модель. Еще раз отметим, что при других исходных данных это может быть совсем не так.

В результате проведенных вычислительных экспериментов учащиеся должны понять, что

- имитационные модели используются тогда, когда задачу не удается (невозможно или очень сложно) решить аналитически;
- любая имитационная модель дает только численное решение для тех входных данных, которые использовались в проведенных сеансах моделирования ("прогонах" модели); что будет при других исходных данных — неизвестно;
- результаты имитационного моделирования зависят от принятых допущений;
- проверить результаты имитационного моделирования и таким образом установить адекватность (или неадекватность) модели можно только с помощью эксперимента, проведенного на объекте-оригинале.

Лишь после этого имеет смысл переходить к использованию готовых пакетов для имитационного моделирования типа GPSS [9] или AnyLogic [10]. Такой подход позволяет избежать очень коварных ловушек нашего времени — "нажимания на кнопки" без понимания сути происходящих процессов и абсолютного доверия к тем числам, которые получаются в результате работы какой-либо программы.

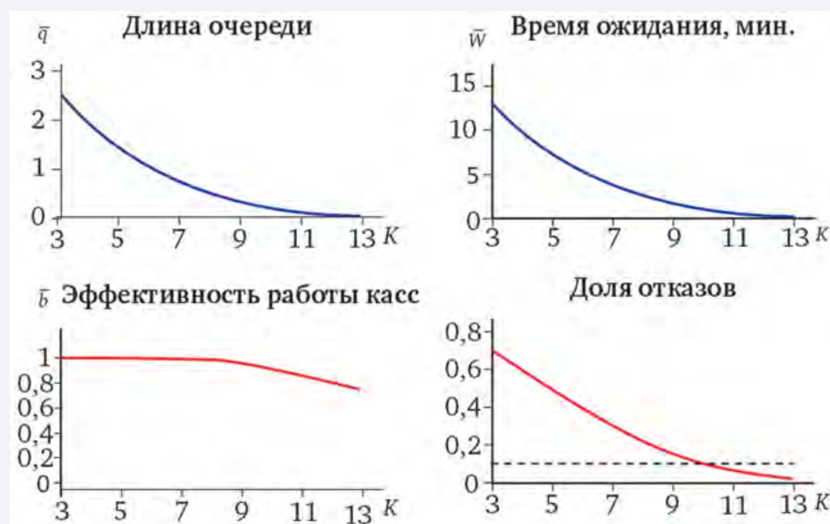

Рис. 11

Литература

1. *Лоу А.М., Кельтон В.Д.* Имитационное моделирование. СПб.: Питер, 2004.
2. *Поляков К.Ю., Еремин Е.А.* Информатика. 10-й класс. Углубленный уровень. В двух частях. М.: Бином, 2013.
3. *Поляков К.Ю., Еремин Е.А.* Информатика. 11-й класс. Углубленный уровень. В двух частях. М.: Бином, 2013.
4. Программа АЛГО [Электронный ресурс] URL: *http://petriv.ho.com.ua/algo/rus*.
5. PascalABC.NET. Современное программирование на языке Паскаль [Электронный ресурс] URL: *http://pascalabc.net*.
6. FreePascal [Электронный ресурс] URL: *http://www.freepascal.org*.
7. *Гнеденко Б.В., Коваленко И.Н.* Введение в теорию массового обслуживания. М.: URSS: КомКнига, 2013.
8. *Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К.* Алгоритмы: построение и анализ = Introduction to Algorithms / Под ред. *И.В. Красикова*. М.: Вильямс, 2005.
9. GPSS: имитационное моделирование систем [Электронный ресурс] URL: *http://gpss.ru*.
10. AnyLogic: Многоподходное имитационное моделирование [Электронный ресурс] URL: *http://www.anylogic.ru*.