A Java Application for Stochastic System Simulation and Gradient Estimation

Guowei Sun
Applied Mathematics
University of Maryland

Qian Wang
Electrical and Computer Engineering
University of Maryland

Abstract

In operations research and management science, discrete event simulation is a widely applicable technique to study complex, read world like systems. The sub-area "simulation optimization" deals with optimization of such simulation programs with three important features. First, no known functional form of the performance measure or objective function w.r.t decision variables is known prior. Second, the simulation is usually expensive to run. Third, each simulation result is a realization of an unknown random variable. Till so far, most research effort has been devoted to find gradient estimators for these types of problems. But the known gradient estimation techniques is not implemented in any commercial simulation software package. In our project, we develop the simulation program for three most important and fundamental stochastic systems, and implement well established gradient estimation techniques.

1 Building Blocks: Queueing Simulation

Single Server Queue

Single-server queue and queueing networks are arguably the most studied operations research model [1]. Since almost all interesting quantities of single server queues are well studied with established theoretical results, they now serve as "community standard" test model for designing novel optimization or simulation techniques [4].

An illustration of a single server queue would be as follows:

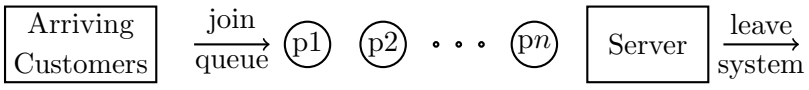

Figure 1: An Illustration of a Queue

For our project, we focus on the simplest M/M/1/∞ queue. Where M/M/1/∞ is kendall's notation [3] for a family of well behaved queues

- The arrival process is a Possion Process. With arrival rate λ
- The service times follows independent exponential distributions with rate μ

- There is one single server
- The queue can be infinitely long

Queueing Networks

A set of connected queues is used to model real world activities, such as shopping malls, Disney land, etc. Each node would be a single server queue, and the leaving customers can either leave the system or enter another queue in the network.

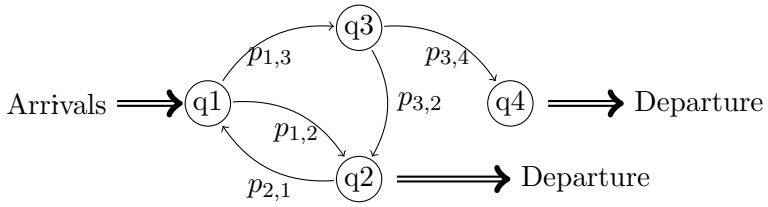

Figure 2: An Illustration of a Queueing Network

The network can be modelled as a directed graph, where the $p_{i,j}$ on each edge is the probability of a departure from queue i to enter queue j. Each queue in the network would have a service rate μ_i. One natural question is, given the transition probabilities of the network, what service rate should be placed on each node to minimize expected overall customer waiting time.

Sample Path Simulation

$N(t)$: number of customers in system at time t. $N(t)$ is a Markov process. In each run of the simulation, a realization of $N(t)$, can be obtained. Which is usually referred to a sample path.

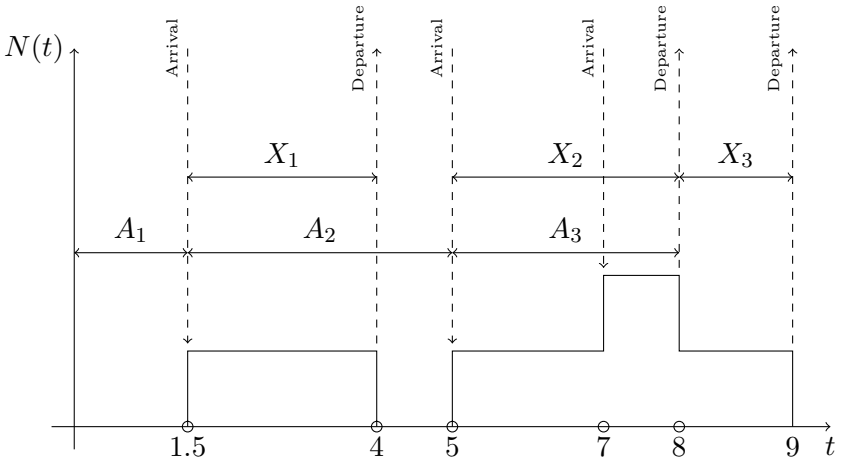

Figure 3: A Sample Path for a M/M/1/∞ Queue

An example of sample path with is given in 3.Where

- Inter-Arrival Times: $A_1 = 1.5, A_2 = 3.5, A_3 = 3$
- Service Times: $X_1 = 2.5, X_2 = 3, X_3 = 1$

2 Simulation,Optimization and Gradient Estimation

Simulation

Though theoretical results for simple models such as the single server queues are obtained. Most models for realistic real world systems is too complex to derive any closed form description of the system performance. Therefore, using computer program to simulate a system and perform statistical analysis using the simulated result is used extensively in practice. Matlab recently provided "simEvent" tool for such simulations. Other commercial simulation software such Arena, are also available.

The general idea are similar

- use pseudo random numbers to simulate certain input distributions to model randomness in the system
- collect simulation output and perform statistical analysis

But, most of the software do not provide optimization routine or functions, which is actually the core motivation of building such simulation programs.

Optimization

A typical stochastic optimization problem could be the following:
Given a $M/M/1/\infty$ queue with arrival rate λ and service rate μ. Suppose we have a cost associated with the waiting times of customers, denoted as c_1 for each unit of time. And a training cost for servers, which can be modelled as $\frac{c_2}{\mu}$. So, bigger μ would mean less waiting time but more training cost, while smaller μ increases waiting time but decrease training cost. Our objective would be to

$$\min_{\mu} E[W(\mu)] + \frac{c}{\mu}$$

The waiting time $W(\mu)$ is a random variable dependent on the service rate μ. We optimize something that is related to the expectation of this random variable. This type of formulation is rather standard for OR research, where

$$\min_{\theta} E[J(\theta)]$$

θ is a parameter, that is often related to the input distribution parameters, e.g. the service rate μ in a single server queue. $J(\theta)$ is some measure of performance, which is a random variable.

Gradient Estimators

Most optimization algorithms use gradient to guide the search for optimal points, such as gradient descent, Newton's method and their variants. In our context, our objective function is of the form

$$E(J(\theta))$$

The corresponding gradient would be

$$\frac{dE[J(\theta)]}{d\theta}$$

We do not have the closed form expression for $E(J(\theta))$. The only information we can obtain is samples of $J(\theta)$ from each simulation run. But, under some regularity conditions, we can exchange gradient operator with expectation operation

$$\frac{dE[J(\theta)]}{d\theta} = E[\frac{dJ(\theta)}{d\theta}]$$

From this formulation, we can collect "samples of gradients", just like collecting "samples of performance measures". Most research effort in this field is devoted to finding a gradient estimator that can be calculated directly from each simulation run. E.g, collecting the gradient of waiting time w.r.t service rate in a single run of queueing simulation. A comprehensive review of how to derive such gradient estimators are given in [2].

For our program, we only need to implement the gradient estimators for M/M/1/∞ queues. Which is given by

- IPA, infinitesimal perturbation analysis estimator

$$\frac{d\bar{T}_N}{d\theta} = \frac{1}{N}\sum_{m=1}^{M}\sum_{i=n_{m-1}+1}^{n_m}\sum_{j=n_{m-1}+1}^{i}\frac{dX_j}{d\theta}$$

  Where M is the number of busy periods observed and n_m is the index of the last customer served in the mth busy period.

- WD, weak derivative estimator

$$(\frac{d\bar{T}}{d\theta})_{WD} = c(\theta)\sum_{i=1}^{N}[\bar{T}_N(A_1,...,A_N,...,X_i^{(2)},...) - \bar{T}_N(A_1,...,A_N,...,X_i^{(1)},...)]$$

  Where, $X_i^{(j)} \sim F_2^{(j)}$, $(c(\theta), F_2^{(1)}, F_2^{(2)})$ is a weak derivative of F_2

- LR/SF, likelihood ratio/score function estimator

$$\begin{aligned}(\frac{d\bar{T}}{d\theta})_{LR} =& \frac{1}{N}\sum_{m=1}^{M}\{\sum_{i=n_{m-1}+1}^{n_m}T_i\sum_{i=n_{m-1}}^{n_m}\frac{\partial \ln f_2(X_i;\theta)}{\partial\theta}\}\\ &-\frac{1}{N}\sum_{m=1}^{M}\{(n_m-n_{m-1})\sum_{i=n_{m-1}+1}^{n_m}\frac{\partial \ln f_2(X_i;\theta)}{\partial\theta}\}\frac{1}{N}\sum_{j=1}^{N}T_j\end{aligned}$$

The above estimators are all single run estimators, meaning we can collect one sample of gradient for each simulation run. These estimators are heavily researched, but due to its theoretical complexity, and difficulty to generalize, they are usually not implemented in simulation software such as Arena and Matlab.

Java Application

The ideal goal is to provide simulation program with the following feature

- Provide a simple user-interface for setting up simulations.
  - Input distribution specification: arrival process, service times, etc
  - Random seed specification
  - Gradient estimator choice, IPA or WD or LR/SF
  - Network construction: how to connect the nodes of queues, and the transition probabilities on each edge.
- Optimization Functionality
  - Apply stochastic approximation algorithms to optimize choice of parameters
  - If possible, code the SPSA (simultaneous perturbation stochastic approximation) algorithm for high-dimensional problems

3 Java Graphical User Interface (GUI) Implementation

Basics for GUI

We build our Graphical User Interface (GUI) for our stochastic system simulation using two sets of Java APIs for graphics programming: AWT (Abstract Windowing Toolkit) and Swing. The hierarchy of the AWT Container classes is as follows Fig 4:

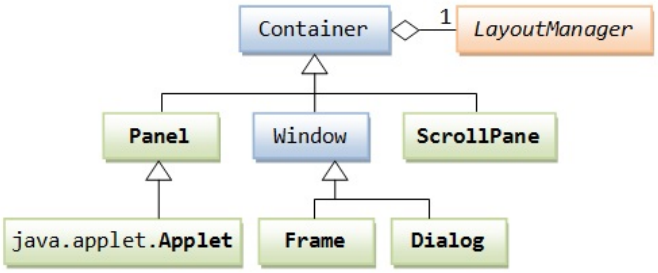

Figure 4: AWT Container Class Diagram

- Top-Level Containers: Frame, Dialog and Applet
  Each GUI program has a top-level container. The commonly-used top-level containers in AWT are Frame, Dialog and Applet: A Frame provides the "main window" for the GUI application, which has a title bar (containing an icon, a title, the minimize, maximize/restore-down and close buttons), an optional menu bar, and the content display area. To write a GUI program, we typically start with a subclass extending from java.awt.

- Secondary Containers: Panel and ScrollPane

  Secondary containers are placed inside a top-level container or another secondary container. AWT also provide these secondary containers: Panel is a rectangular box under a higher-level container used to layout a set of related GUI components in pattern such as grid or flow.

Layout Managers and Panel

The top level layout of our graphical user interface is shown as following, it is built in the ***startWindow.class***, which generates two sub windows: QueueWindow and NetworkWindow. The class hierarchy for the ***startWindow*** is shown in Fig 5.

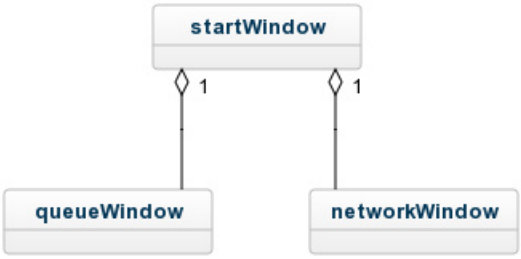

Figure 5: Class Hierarchy of startWindow

And the Layout of the startWindow Panel are shown in Fig 6.

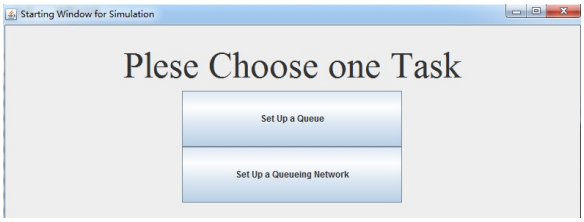

Figure 6: Layout of startWindow

Queue Window GUI

When clicking on the corresponding button "Set Up a Queue" a new window will start for the user to set parameters and do the simulation. the layout of the "Set Up a Queue" panel is shown in Fig 7.

In order to set up the layout of a container such as the QueueWindow in this case, we need to following the three steps in the instruction:

1 Construct an instance of the chosen layout object.

2 Invoke the ***setLayout()*** method of the Container, with the layout object created as the argument.

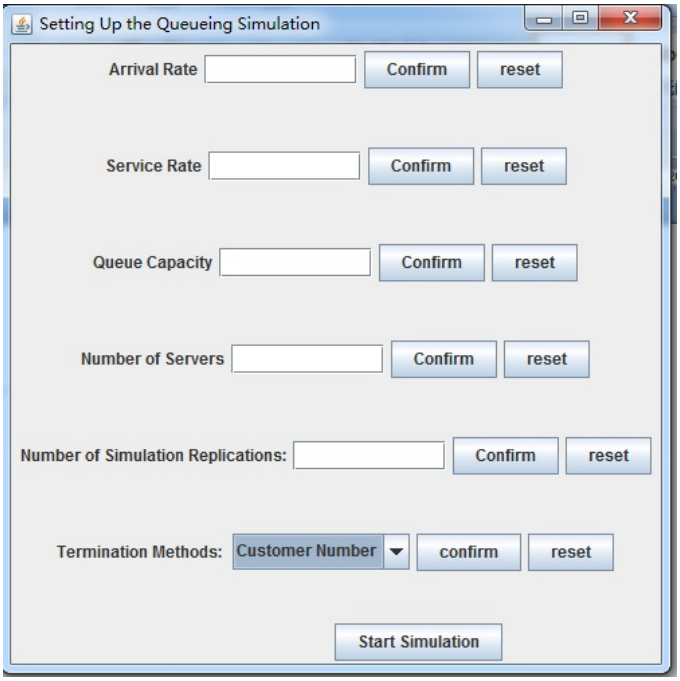

Figure 7: Layout of Set up Queue Window

3 Place the GUI components into the Container using the ***add ()*** method in the correct order.

As the figure shows there are 6 sub components in this panel, each component is an instance of the type of classes. The class hierarchy of the queue window is shown in Fig 8. And the setup and place method to construct the queueWindow is shown in the following codes segment.

```
private class confirmButtonHandlerInt implements ActionListener{
    public queueWindow(){
     setTitle("Setting Up the Queueing Simulation");

     // construct the input panel
     JPanel qPane = new JPanel();
     qPane.setLayout(new BoxLayout(qPane, BoxLayout.PAGE_AXIS));
     this.arrivalPane = new inputSequenceDouble("Arrival Rate","
    Confirm");
     qPane.add(arrivalPane);
     this.servicePane = new inputSequenceDouble("Service Rate","
    Confirm");
     qPane.add(servicePane);
     ...
     }
       }
```

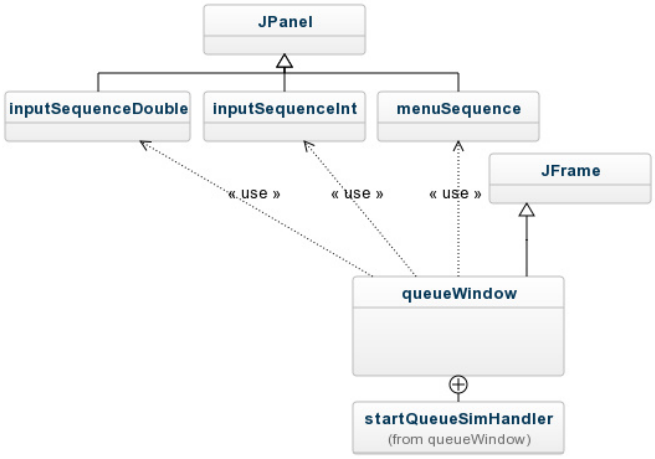

Figure 8: Class Hierarchy of queueWindow

There are totally three ***inputSequenceInt*** type objects, two ***inputSequence-Double*** type objects and one type of ***menuSequence*** object in the Layout of Queue simulation panel. The arrival rate and service rate are input type as double, so it uses the ***inputSequenceDouble.class*** to build the panel. Otherwise the capacity of the system, the service number and the replication number are integer type inputs and it uses the ***inputSequenceInt.class***.

The ***inputSequenceInt*** and ***inputSequenceDouble*** classes have the similar structure; the only difference is the input value is set to Integer for capacity and the number of server. And the arrival rate and service rate are type of double.

We choose the ***inputSequenceInt.class*** as an example to show the inner structure of this class. The class hierarchy of ***inputSequenceInt*** is shown in Fig 9.

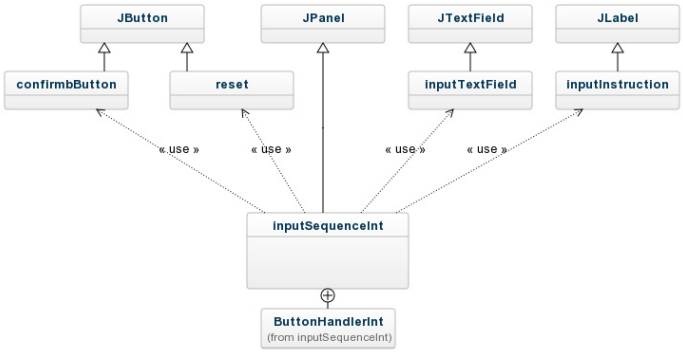

Figure 9: Class Hierarchy of inputSequenceInt

The ***inputSequenceInt*** class extends from ***JPanel*** abstract class in java.awt

and it has 4 components as 1 ***JLabel*** 1 ***JTextField*** and two ***JButton***s : reset and confirm.

We use a named Inner Class as Event Listener. A nested class is useful if you need a small class which relies on the enclosing outer class for its private variables and methods. It is ideal in an event-driven environment for implementing event handlers. This is because the event handling methods (in a listener) often require access to the private variables (e.g., a private TextField) of the outer class.

```
private class confirmButtonHandlerInt implements ActionListener{
    public void actionPerformed(ActionEvent e){
      // fill in the event for the confirmation button
      String s = inputTextField.getText().replaceAll("\\s+","");
      System.out.println(s);
      boolean v;
      try{
        Integer.parseInt(s);
        v = true;
      } catch(NumberFormatException en){
        v = false;
      }
      ...
      }
```

In this example, we define a new class called ***confirmButtonHandlerInt***, and create an instance of ***ButtonHandler*** as the ActionEvent listener for the confirmButton. The ***confirmButtonHandlerInt*** needs to implement the ActionListener interface, and override the ***actionPerformed()*** handler.

The ***menuSequence*** is used to create a panel for the user selecting the termination type. It has a label , two buttons and a combo Box to select the termination method. It also has an inner class called ***boxHandler*** implementing Action Event listener. The class hierarchy of ***menuSequence*** is shown in Fig 10.

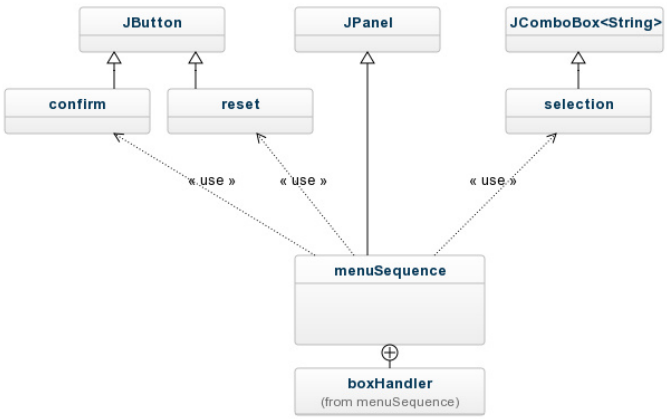

Figure 10: Class Hierarchy of imenuSequenceclass

Network Window GUI

We also build a Network window for the user to define the Queueing Network inputs. It separates from the simple Queue window, and when the user clicking on the button " Set up a Queueing Network", it will pop up a new Frame which shown in Fig 11.

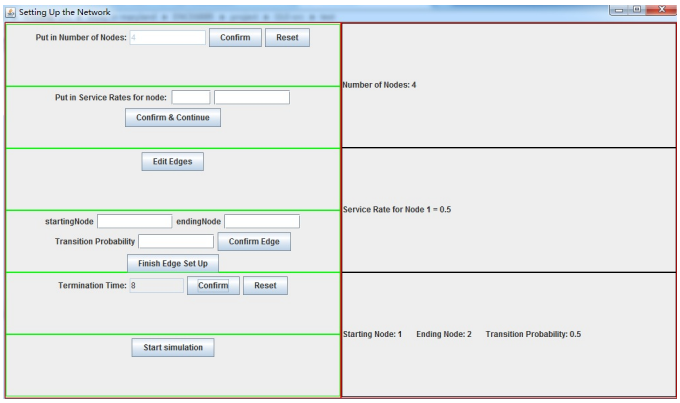

Figure 11: Layout of NetworkWiondow

There is an input sequence defined in the interface which user needs to follow to initiate the queue network.

1. Put in number of Nodes: the user initialize the number of Nodes in the queue network.
2. Put in the service rates for each node.
3. Edit Edge, initialize the edges in the network, also set the transfer probability from Node A to Node B.
4. Set terminate time.
5. Start simulation.

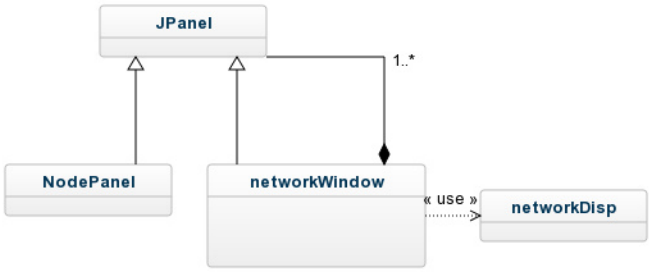

Figure 12: Class Hierarchy of Queueing Network Window

We utilized the composite design pattern to build the network queue window shown in Fig 12. As the steps we discussed above shows, we need 5 sub panels embedded in the Network Queue Window.

The right side of the Queue Network Window is the Display network panel, which is to provide the user view of the setup parameters. As if you enter the number of nodes and by clicking on the confirm button, the right side panel will display the nodes number you just entered. We implement this in the ***networkDisp.class***.

4 QUEUEING SIMULATION

This part is an introduction to our queueing simulation program using information provided by the user through the UI.

CLASSES and METHODS

- Customer:
  - Contains basic information, such as service time and inter-arrival time for the customer.
  - Recording methods: records the time of possible events including arriving into the system, starting being served and departing the system
  - Information Retrieval methods: provide necessary information about the Customer to be used by other classes and methods.
- WaitQueue:
  - Customers waiting to be served is contained in an ArrayList of Customers.
  - Add and Remove methods in response to arrival and departing events. Update the Customer information according to different actions.
  - Sort methods, which can sort the Customers according to their service time, allowing "shortest processing time" queue discipline.
- server:
  - An ArrayList of Customers which is being served
  - Add and remove methods to add and remove customers from server. At the same time, check if addition/deletion is possible based on the current system state, and update the Customer information(its arriving to serving time, its departing time) accordingly.
  - Sort methods: to order the Customers based on their departing time. So in an departure event,the one with the smallest departing time will leave the system.
- randTimes: This class is to generate samples from the most common distribution used in queueing simulation.
  - Exponential Distribution: use inverse transform method
  - Erlang Distribution: Summation of n exponential random variables generated through inverse transform method
  - Uniform Distribution: translation and scaling of the random number generated from java.Math.random() method.
- state: This class is keeping track of the simulation process and print the output into a result file.
  - File, FileWrite: to create a directory, and text files to write our output

  - Keeping track of state variables: time, event, and number of customer in the system.
- departedCustomers: This class collects the customers departed from the queueing system, and calculate the desired statistics, or performance measures from the information contained with the Customer class.
- runSum: this class is combining all the classes, read the simulation set up and running the simulation.

SIMULATION PROCEDURE

The simulation procedure can be stated as

Simulation: Queueing Simulation Procedure

procedure SET UP SIMULATION (Through UI)
  Generate ArrayList of Customers c
  Servers s, Customers in queue q
  System Clock *clock*
  Output writer *state*
  Generate $X_i \sim f_1$,$A_i \sim f_2$
  while Termination not reached **do**
    Identify next event ▷ +1 for arrival, -1 for departure
    update c
    update q
    update s
    update *clock*
    update acting Customer
    record event
  Finish Output
Finish Simulation

Test Run of Queueing Simulation

Here, we demonstrate a test run of the queueing simulation program, and how to perform some preliminary analysis on the simulation result.

- In the queue window, set up our simulation conditions 13
  The basic set up are
  - arrival rate = 4
  - service rate = 4
  - queue capacity = 10
  - number of servers = 4
  - replications = 10
  - terminate after 100 customers

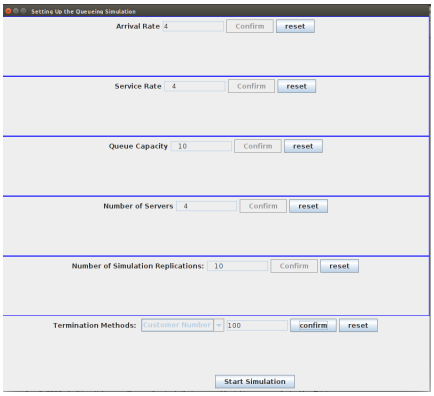

Figure 13: Simulation Set Up for out Test Run

- Error Checking: If invalid input is detected. Then an error message will be displayed, and a correct input should be entered. For instance:14 A double

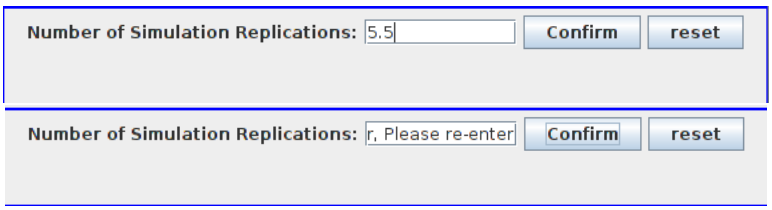

Figure 14: Error Checking for the Inputs

  is entered where an integer is required. A message is displayed, and the field should be re-entered.

- Start Simulation: Press the start simulation button.
- Simulation Output: A new directory **Output** will be created under the current directory. If totally n simulation is required. Then n files, each containing one independent simulation run is created. A sample result from a test run specified above is demonstrated in figure15
- Processing of Simulation Result: Once a simulation is completed, we can collect the simulation result from the output files, and calculate performance measures. One simulation result is processed, and the sample path is plotted using R. See figure 16

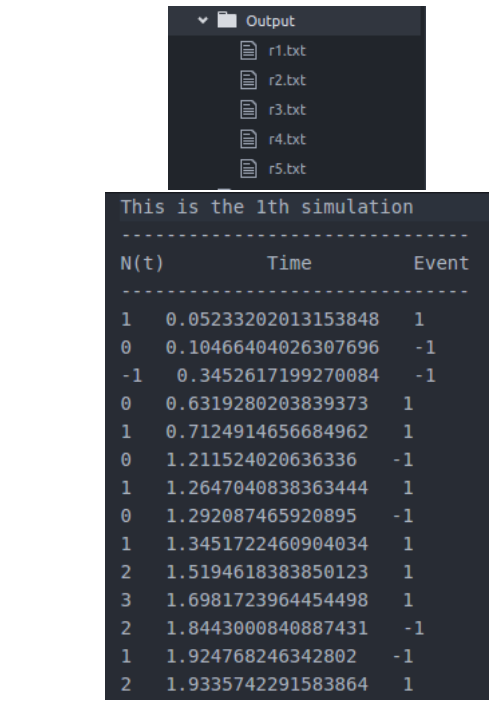

Figure 15: Simulation Output Directory and a Sample output file

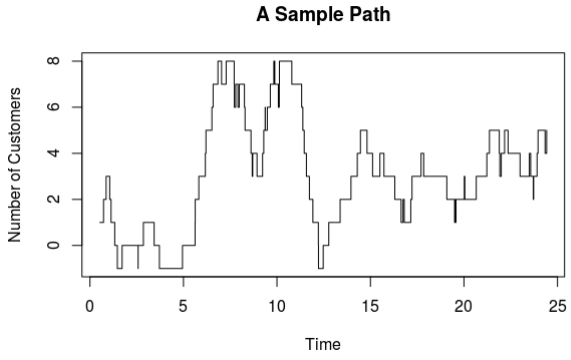

Figure 16: Plot of a Sample Path

5 Continuing Work

Queueing Network

We did not finish this part due to time constraints. But a basic frame is constructed already.

- vertex: a class contains servers, add and remove methods, and a method to randomly pick the next destination using given probabilities.
- clock: keeping track of system evolution and calculate the next event.

Calculation on the Run

In the current program, we collect the simulation result and then calculate statistics we want from the result. This requires significant amount of memory to store all the information, or a time to process the output files. Which is not ideal. In later updates, we should try to update our statistics during the simulation. In this way, we can significantly reduce the memory requirement and data processing time.

More Detailed UI

The design of our UI can still be improved. To name two things

- Enable output path selection.
- Automatically close the window when the simulation is finished.

In the operations research community. Queueing and queueing network are important standard test problems. Therefore, our effort of polishing this application will continue. And it is highly likely that it will be used in our own research in the future.

References

[1] Michael C Fu. Optimization for simulation: Theory vs. practice. *INFORMS Journal on Computing*, 14(3):192–215, 2002.

[2] Michael C Fu. Gradient estimation. *Handbooks in operations research and management science*, 13:575–616, 2006.

[3] David G. Kendall. Stochastic processes occurring in the theory of queues and their analysis by the method of the imbedded markov chain. *Ann. Math. Statist.*, 24(3):338–354, 09 1953.

[4] Vidyadhar G Kulkarni. *Modeling and analysis of stochastic systems*. CRC Press, 2009.