

# JSL - Java Simulation Library

Manuel D. Rossetti

2021-05-19



# Contents

<b>Simulation Modeling</b>	<b>xi</b>
0.1 Simulation Modeling . . . . .	xi
0.2 Why Simulate? . . . . .	xii
0.3 Types of Systems and Simulation Models . . . . .	xiv
0.4 Simulation: Descriptive or Prescriptive Modeling? . . . . .	xviii
0.5 Randomness in Simulation . . . . .	xix
0.6 Simulation Languages . . . . .	xx
0.7 Simulation Methodology . . . . .	xxi
0.8 Overview of the Java Simulation Library . . . . .	xxviii
0.9 Exercises . . . . .	xxx
<b>Random Number Generation</b>	<b>xxxiii</b>
0.10 Random Number Generator . . . . .	xxxiii
0.11 Random Package . . . . .	xxxiv
0.11.1 Creating and Using Streams . . . . .	xxxviii
0.11.2 Common Random Numbers . . . . .	xl
0.11.3 Creating and Using Antithetic Streams . . . . .	xlii
0.12 Frequently Asked Questions . . . . .	xliii
<b>Random Variate Generation and Probability Modeling</b>	<b>xlv</b>
0.13 Continuous and Discrete Random Variables . . . . .	xlv
0.14 Overview of Generation Algorithms . . . . .	xlvi
0.15 Creating and Using Random Variables . . . . .	xlviii
0.16 Modeling Probability Distributions . . . . .	li
<b>Collecting Statistics</b>	<b>lv</b>
0.17 Creating and Using a Statistic . . . . .	lvii
0.18 Histograms and Frequencies . . . . .	lxii
0.19 Batch Statistics . . . . .	lxviii
0.20 Summary . . . . .	lxxii
<b>Monte Carlo Methods</b>	<b>lxxv</b>
0.21 Simple Monte Carlo Integration . . . . .	lxxv

0.22 Simulating the Game of Craps . . . . .	lxxviii
<b>Introduction to Discrete Event Modeling</b>	<b>lxix</b>
0.23 Introduction . . . . .	lxix
0.24 Discrete-Event Dynamic Systems . . . . .	lxxxii
0.25 How the Discrete-Event Clock Works . . . . .	lxxxiii
0.26 Simulating a Queueing System By Hand . . . . .	lxxxviii
0.27 Modeling DEDS in the JSL . . . . .	xcvii
0.27.1 Event Scheduling . . . . .	xcvii
0.27.2 Simple Event Scheduling Examples . . . . .	cii
0.27.2.1 Implementing Event Actions Using the EventAction-Ifc Interface . . . . .	cii
0.27.2.2 Overview of Simulation Run Context . . . . .	civ
0.27.2.3 Simulating a Poisson Process . . . . .	cv
0.27.3 Up and Down Component Example . . . . .	cvi
0.27.4 Modeling a Simple Queueing System . . . . .	cxi
0.28 Summary . . . . .	cxxi
<b>Modeling with Queues, Resources, and Stations</b>	<b>cxxiii</b>
0.29 Terminology of Simulation Modeling . . . . .	cxxiii
0.30 Entities and Attributes . . . . .	cxxiv
0.31 Event Generators . . . . .	cxxv
0.32 The Station Package . . . . .	cxxx
0.32.1 Modeling Simple Queueing Stations . . . . .	cxxxiv
0.33 Sharing a Resource . . . . .	cxli
0.34 Complex System Example . . . . .	cxlvi
0.34.1 Conceptualizing the Model . . . . .	cxlvi
0.34.2 Implementing the Model . . . . .	cxlviii
0.34.3 Model Results . . . . .	clv
0.35 Summary . . . . .	clvi
<b>Analyzing Simulation Output</b>	<b>clix</b>
0.36 Types of Statistical Variables . . . . .	clx
0.37 Types of Simulation With Respect To Output Analysis . . . . .	clxv
0.38 Analysis of Finite Horizon Simulations . . . . .	clxvii
0.38.1 Determining the Number of Replications . . . . .	clxix
0.39 Finite Horizon Example . . . . .	clxxi
0.39.1 Conceptualizing the Model . . . . .	clxxii
0.39.2 Sequential Sampling for Finite Horizon Simulations . . . . .	clxxx
0.40 Analysis of Infinite Horizon Simulations . . . . .	clxxxiii
0.40.1 Assessing the Effect of Initial Conditions . . . . .	clxxxviii
0.40.2 Performing the Method of Replication-Deletion . . . . .	cxciv
0.40.2.1 Determining the Warm Up Period . . . . .	cxcv
0.40.3 The Method of Batch Means . . . . .	cxcix
0.40.4 Performing the Method of Batch Means . . . . .	cciv
0.41 Comparing System Configurations . . . . .	ccvii

0.41.1 Comparing Two Systems . . . . .	ccviii
0.41.1.1 Analyzing Two Independent Samples . . . . .	ccix
0.41.1.2 Analyzing Two Dependent Samples . . . . .	ccxiii
0.41.1.3 Using Common Random Numbers . . . . .	ccxv
0.41.2 Multiple Comparisons . . . . .	ccxvi
0.42 Summary . . . . .	ccxvi
<b>Appendix</b>	<b>ccxvii</b>
<b>Miscellaneous Utility Classes</b>	<b>ccxvii</b>
.1 Reporting . . . . .	ccxvii
.2 JSLMath Class . . . . .	ccxviii
.3 The JSL Database . . . . .	ccxix
.3.1 The JSL Database Structure . . . . .	ccxix
.3.2 Creating and Using a Default JSL Database . . . . .	ccxxi
.3.3 Creating and Using JSL Databases . . . . .	ccxxiii
.3.4 Querying the JSL Database . . . . .	ccxxvi
.3.5 Additional Functionality . . . . .	ccxxvii
<b>Generating Pseudo-Random Numbers and Random Variates</b>	<b>ccxxxix</b>
.4 Pseudo Random Numbers . . . . .	ccxxx
.4.1 Random Number Generators . . . . .	ccxxxii
.5 Generating Random Variates from Distributions . . . . .	ccxxxvii
.5.1 Inverse Transform Method . . . . .	ccxxxvii
.5.2 Convolution . . . . .	cclvii
.5.3 Acceptance/Rejection . . . . .	ccl
.5.4 Mixture Distributions, Truncated Distributions, and Shifted Random Variables . . . . .	ccli
.6 Summary . . . . .	cclviii
.7 Exercises . . . . .	cclviii
<b>Probability Distribution Modeling</b>	<b>cclxvii</b>
.8 Random Variables and Probability Distributions . . . . .	cclxix
.9 Modeling with Discrete Distributions . . . . .	cclxxiv
.10 Fitting Discrete Distributions . . . . .	cclxxiv
.10.1 Fitting a Poisson Distribution . . . . .	cclxvii
.10.2 Visualizing the Data . . . . .	cclxxvi
.10.3 Estimating the Rate Parameter for the Poisson Distribution .	cclxxxii
.10.4 Chi-Squared Goodness of Fit Test for Poisson Distribution .	cclxxxiv
.10.5 Chi-Squared Goodness of Fit Test . . . . .	cclxxxv
.10.6 Using the fitdistrplus R Package on Discrete Data . . . . .	cclxxxix
.10.7 Fitting a Discrete Empirical Distribution . . . . .	ccxcii
.11 Modeling with Continuous Distributions . . . . .	ccxciv
.12 Fitting Continuous Distributions . . . . .	ccxcvi
.12.1 Visualizing the Data . . . . .	ccxcvii
.12.2 Statistically Summarize the Data . . . . .	ccxcvii

.12.3	Hypothesizing and Testing a Distribution . . . . .	ccc
.12.4	Kolmogorov-Smirnov Test . . . . .	cccvii
.12.5	Visualizing the Fit . . . . .	cccxix
.12.6	Using the Input Analyzer . . . . .	cccxv
.13	Testing Uniform (0,1) Pseudo-Random Numbers . . . . .	cccxii
.13.1	Chi-Squared Goodness of Fit Tests for Pseudo-Random Numbers . . . . .	cccxiii
.13.2	Higher Dimensional Chi-Squared Test . . . . .	cccxv
.13.3	Kolmogorov-Smirnov Test for Pseudo-Random Numbers . .	cccxvii
.13.4	Testing for Independence and Patterns in Pseudo-Random Numbers . . . . .	cccxix
.14	Additional Distribution Modeling Concepts . . . . .	cccxxi
.15	Summary . . . . .	cccxxiv
.16	Exercises . . . . .	cccxxv
	<b>Discrete Distrbutions</b>	<b>cccxli</b>
	<b>Continuous Distrbutions</b>	<b>cccxlv</b>
	<b>Statistical Tables</b>	<b>cccxlix</b>

# Preface



The online version of this book is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License<sup>1</sup>.

The purpose of this book is to provide an overview of the Java Simulation Library (JSL). The JSL facilitates simulation modeling by providing Java libraries that ease the development of simulation models. The JSL has packages for random variable generation, statistical analysis, event calendar management, and other common constructs that are needed when performing Monte Carlo and discrete event simulation modeling.

## JSL Project Page

The JSL project is based on a git repository<sup>2</sup>

## Open Source Dependencies

The JSL uses the following open source libraries.

- Apache Commons Math<sup>3</sup>
- Apache POI<sup>4</sup>
- Apache Derby<sup>5</sup>
- Apache Commons IO<sup>6</sup>
- OpenCSV<sup>7</sup>
- Google Guava<sup>8</sup>
- Tablesaw<sup>9</sup>
- Java Object Oriented Query (jooq)<sup>10</sup>

---

<sup>1</sup><http://creativecommons.org/licenses/by-nc-nd/4.0/>

<sup>2</sup><https://git.uark.edu/jslfork/JSL>

<sup>3</sup><https://commons.apache.org/proper/commons-math/>

<sup>4</sup><https://poi.apache.org/>

<sup>5</sup><https://db.apache.org/derby/>

<sup>6</sup><https://commons.apache.org/proper/commons-io/>

<sup>7</sup><http://opencsv.sourceforge.net/>

<sup>8</sup><https://github.com/google/guava>

<sup>9</sup><https://github.com/jtablesaw/tablesaw>

<sup>10</sup><https://www.jooq.org>

- SL4J<sup>11</sup>
- JavaFX<sup>12</sup>

---

<sup>11</sup><https://www.slf4j.org/>

<sup>12</sup><https://openjfx.io/>

# About the Author

**Dr. Manuel Rossetti<sup>13</sup>, P.E.** Dr. Rossetti is a Professor of Industrial Engineering and the Director for the NSF I/UCRC Center for Excellence in Logistics and Distribution (CELDI<sup>14</sup>) at the University of Arkansas. Dr. Rossetti has published over 120 journal and conference articles in the areas of simulation, logistics/inventory, and healthcare and has been the PI or Co-PI on funded research projects totaling over 5.4 million dollars. He was selected as a Lilly Teaching Fellow in 1997/98 and was voted Best IE Teacher by IE students in 2007, 2009, and 2017. He won the IE Department Outstanding Teacher Award in 2001-02, 2007-08, and 2010-11. He received the College of Engineering Imhoff Teaching Award in 2012 and was elected an IIE Fellow. In 2013, the UA Alumni Association awarded Dr. Rossetti the Charles and Nadine Baum Faculty Teaching Award, the highest award for teaching at the university. In 2015, Dr. Rossetti served as program chair for the Winter Simulation Conference. He is also the author of the book, *Simulation Modeling and Arena*<sup>15</sup>, published by John Wiley & Sons.

Dr. Rossetti grew up in Canton, Ohio and is an ardent Cleveland sports fan. He received his Ph.D. and MSIE degrees in Industrial and Systems Engineering from The Ohio State University and his BSIE degree from the University of Cincinnati. He is a registered professional engineer in the State of Arkansas.

---

<sup>13</sup><https://sites.uark.edu/rossetti/>

<sup>14</sup><https://celdi.org/>

<sup>15</sup><https://www.wiley.com/en-us/Simulation+Modeling+and+Arena%2C+2nd+Edition-p-9781118607916>



# Simulation Modeling

## LEARNING OBJECTIVES

- To be able to describe what computer simulation is
- To be able to discuss why simulation is an important analysis tool
- To be able to list and describe the various types of computer simulations
- To be able to describe a simulation methodology

In this book, you will learn how to model systems within a computer environment in order to analyze system design configurations. The models that you will build and exercise are called simulation models. When developing a simulation model, the modeler attempts to represent the system in such a way that the representation assumes or mimics the pertinent outward qualities of the system. This representation is called a simulation model. When you execute the simulation model, you are performing a simulation. In other words, simulation is an instantiation of the act of simulating. A simulation is often the next best thing to observing the real system. If you have confidence in your simulation, you can use it to infer how the real system will operate. You can then use your inference to understand and improve the system's performance.

## 0.1 Simulation Modeling

In general, simulations can take on many forms. Almost everyone is familiar with the board game Life. In this game, the players imitate life by going to college, getting a job, getting married, etc. and finally retiring. This board game is a simulation of life. As another example, the military performs war game exercises which are simulations of battlefield conditions. Both of these simulations involve a physical representation of the thing being simulated. The board game, the rules, and the players represent the simulation model. The battlefield, the rules of engagement, and the combatants are also physical representations. No wishful thinking will make the simulations that you develop in this book real. This is the first rule to remember about simulation. A simulation is only a model (representation) of the real thing. You can make your simulations as realistic as time and technology allows, but they are not the real thing. As you would never confuse a toy airplane with a real airplane, you should never confuse

a simulation of a system with the real system. You may laugh at this analogy, but as you apply simulation to the real world you will see analysts who forget this rule. Don't be one.

All the previous examples involved a physical representation or model (real things simulating other real things). In this book, you will develop computer models that simulate real systems. Ravindran et al. (1987) define computer simulation as: "A numerical technique for conducting experiments on a digital computer which involves logical and mathematical relationships that interact to describe the behavior of a system over time." Computer simulations provide an extra layer of abstraction from reality that allows fuller control of the progression of and the interaction with the simulation. In addition, even though computer simulations are one step removed from reality, they are often capable of providing constructs which cannot be incorporated into physical simulations. For example, an airplane flight simulator can have emergency conditions for which it would be too dangerous or costly to provide in a physical based simulation training scenario. This representational power of computer modeling is one of the main reasons why computer simulation is used.

## 0.2 Why Simulate?

Imagine trying to analyze the following situation. Patients arrive at an emergency room. The arrival of the patients to the emergency department occurs randomly and may vary with the day of the week and even the hour of the day. The hospital has a triage station, where the arriving patient's condition is monitored. If the patient's condition warrants immediate attention, the patient is expedited to an emergency room bed to be attended by a doctor and a nurse. In this case, the patient's admitting information may be obtained from a relative. If the patient does not require immediate attention, the patient goes through the admitting process, where the patient's information is obtained. The patient is then directed to the waiting room, to wait for allocation to a room, a doctor, and a nurse. The doctors and nurses within the emergency department must monitor the health of the patients by performing tests and diagnosing the patient's symptoms. This occurs on a periodic basis. As the patient receives care, the patient may be moved to and require other facilities (MRI, X-ray, etc.). Eventually, the patient is either discharged after receiving care or admitted to the main hospital. The hospital is interested in conducting a study of the emergency department in order to improve the care of the patients while better utilizing the available resources. To investigate this situation, you might need to understand the behavior of certain measures of performance:

- The average number of patients that are waiting.
- The average waiting time of the patients and their average total time in the emergency department.
- The average number rooms required per hour.
- The average utilization of the doctors and nurses (and other equipment).

Because of the importance of emergency department operations, the hospital has historical records available on the operation of the department through its patient tracking system. With these records, you might be able to estimate the current performance of the emergency department. Despite the availability of this information, when conducting a study of the emergency department you might want to propose changes to how the department will operate (e.g. staffing levels) in the future. Thus, you are faced with trying to predict the future behavior of the system and its performance when making changes to the system. In this situation, you cannot realistically experiment with the actual system without possibly endangering the lives or care of the patients. Thus, it would be better to model the system and to test the effect of changes on the model. If the model has acceptable fidelity, then you can infer how the changes will affect the real system. This is where simulation techniques can be utilized.

If you are familiar with operations research and industrial engineering techniques, you may be thinking that the emergency department can be analyzed by using queueing models. Later chapters of this book will present more about queueing models; however, for the present situation, the application of queueing models will most likely be inadequate due to the complex policies for allocating nurses, doctors, and beds to the patients. In addition, the dynamic nature of this system (the non-stationary arrivals, changing staffing levels, etc.) cannot be well modeled with current analytical queueing models. Queueing models might be used to analyze portions of the system, but a total analysis of the dynamic behavior of the entire system is beyond the capability of these types of models. But, a total analysis of the system is not beyond simulation modeling.

Simulation may be the preferred modeling methodology if the understanding gained from developing and using a simulation model is worth the time and cost associated with developing and using the model. Good uses of simulation include:

- Understanding how complex interactions in the system effect performance.
- Understanding how randomness effects performance.
- Comparing a fixed set of design alternatives to determine which design meets the performance goals under which conditions
- Training people to prepare them for dealing with events that may be disruptive to the actual system.
- The model will be used repeatedly for decision making.
- When the decision associated with the problem has a high cost so that the cost of building the model and evaluating the design is worth its development.
- When the current system does not yet exist and you need to ensure that the chosen design will meet specifications.

Simulation modeling activities encapsulate all three major modeling methods of data analytics: descriptive, predictive, and prescriptive. Descriptive modeling uses historical data to describe what happened in order to understand past behavior of a system. Predictive modeling uses historical data to develop models that help us understand

future behavior in order to answer what *may* happen. Descriptive modeling summarizes past data for understanding. Predictive modeling uses past data to predict future behavior. Prescriptive modeling indicates what should be done and is integral to answering questions involving system design.

A simulation model is both a descriptive and predictive model. In addition, when coupled with stochastic optimization methods or a rigorous design process that evaluates and recommends designs, a simulation model becomes an integral part of the prescriptive modeling process. A simulation model *describes* how a system works by encapsulating that description within the operating runs/constructs of the model. A simulation model uses descriptive models (input models, summary statistics). A simulation model *predicts* future system response. A simulation model can be used to predict future behavior through running what-if scenarios. Simulation is inherently a predictive modeling methodology. Unlike other predictive modeling techniques found in data analytics, such as regression, neural networks, random forests, etc., simulation explicitly incorporates *domain* knowledge into the modeling activity by incorporating system operating behavior. The behavior is modeled through the physical and logical rules that apply to the relationships between the components of the system. Unlike pure statistical predictive models, simulation has the advantage of explicitly representing relationships rather than just relying on discovering relationships.

A key advantage of simulation modeling is that it has the capability of modeling the entire system and its complex inter-relationships. The representational power of simulation provides the flexible modeling that is required for capturing complex processes. As a result, all the important interactions among the different components of the system can be accounted for within the model. The modeling of these interactions is inherent in simulation modeling because simulation imitates the behavior of the real system (as closely as necessary). The prediction of the future behavior of the system is then achieved by monitoring the behavior of different modeling scenarios as a function of simulated time.

Real world systems are often too complex for analytical models and often too expensive to experiment with directly. Simulation models allow the modeling of this complexity and enable low cost experimentation to make inferences about how the actual system might behave.

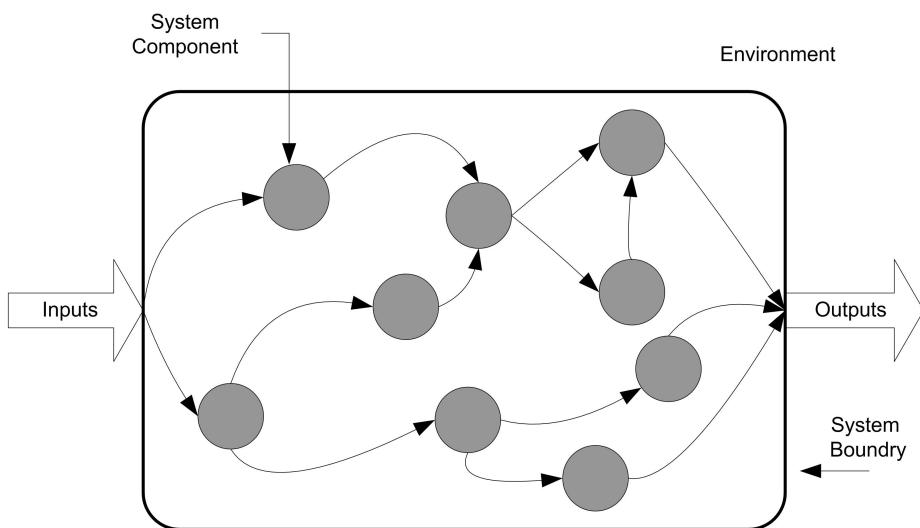
### 0.3 Types of Systems and Simulation Models

The main purpose of a simulation model is to allow observations about a particular system to be collected as a function of time. So far the word *system* has been used in much of the discussion, without formally discussing what a system is. According to (Blanchard and Fabrycky, 1990) a system is a set of inter-related components working together towards a common objective. The standard for systems engineering provides a deeper definition:

"A system is a composite of people, products, and processes that provide a capability to satisfy stated needs. A complete system includes the

facilities, equipment (hardware and software), materials, services, data, skilled personnel, and techniques required to achieve, provide, and sustain system effectiveness." (Command, 1991)

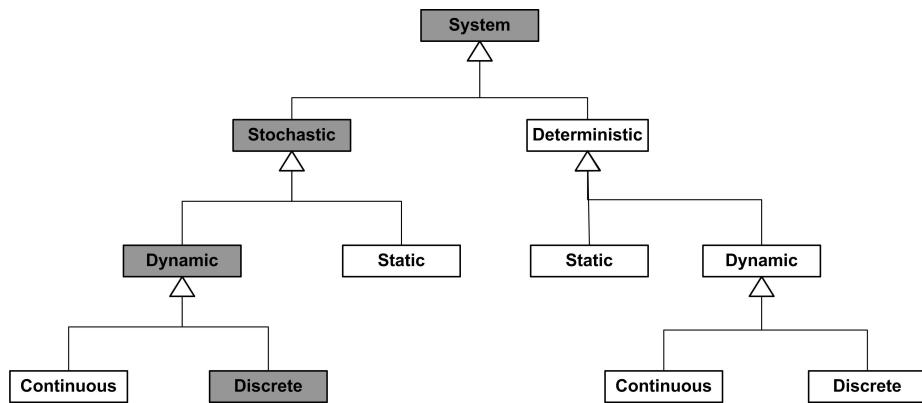
Figure1 illustrates the fact that a system is embedded within an environment and that typically a system requires inputs and produces output using internal components. How you model a particular system will depend upon the intended use of the model and how you perceive the system. The modeler's view of the system colors how they conceptualize it. For example, for the emergency room situation, "What are the system boundaries? Should the ambulance dispatching and delivery process be modeled? Should the details of the operating room be modeled?" Clearly, the emergency room has these components, but your conceptualization of it as a system may or may not include these items, and thus, your decisions regarding how to conceptualize the system will drive the level of abstraction within your modeling. An important point to remember is that two perfectly logical and rational people can look at the same thing and conceptualize that thing as two entirely different systems based on their "Weltanschauung" or world view.



**Figure 1:** Conceptualization of a System

Because how you conceptualize a system drives your modeling, it is useful to discuss some general system classifications. Systems might be classified by whether or not they are man-made (e.g. manufacturing system) or whether they are natural (e.g. solar system). A system can be physical (e.g. an airport) or conceptual (e.g. a system of equations). If stochastic or random behavior is an important component of the system then the system is said to be stochastic, if not then it is considered deterministic. One of the more useful ways to look at a system is whether it changes with respect to time. If a system does not change significantly with respect to time it is said to be static, else it is called dynamic. If a system is dynamic, you might want to consider how it evolves

with respect to time. A dynamic system is said to be discrete if the state of the system changes at discrete points in time. A dynamic system is said to be continuous if the state of the system changes continuously with time. This dichotomy is purely a function of your level of abstraction. If conceptualizing a system as discrete, serves our purposes then you can call the system discrete. Figure 2 illustrates this classification of systems. This book primarily examines stochastic, dynamic, discrete systems.



**Figure 2:** General Types of Systems

The main purpose of a simulation model is to allow observations about a particular system to be gathered as a function of time. From that standpoint, there are two distinct types of simulation: 1) discrete event and 2) continuous.

Just as discrete systems change at discrete points in time, in discrete event simulation observations are gathered at selected points in time when certain changes take place in the system. These selected points in time are called events. On the other hand, continuous simulation requires that observations be collected continuously at every point in time (or at least that the system is described for all points in time). The types of models to be examined in this book are called discrete-event simulation models.

To illustrate the difference between the two types of simulation, contrast a fast food service counter with that of oil loading facility that is filling tankers. In the fast food service counter system, changes in the status of the system occur when a customer either arrives to place an order or when the customer receives their food. At these two events, measures such as queue length and waiting time will be affected. At all the other points in time, these measures remain either unchanged (e.g. queue length) or not yet ready for observation (e.g. waiting time of the customer). For this reason, the system does not need to be observed on a continuous basis. The system need only be observed at selected discrete points in time, resulting in the applicability of a discrete-event simulation model.

In the case of the oil tanker loading example, one of the measures of performance is the amount of oil in each tanker. Because the oil is a liquid, it cannot be readily divided into discrete components. That is, it flows continuously into the tanker. It is not necessary

(or practical) to track each molecule of oil individually, when you only care about the level of the oil in the tanker. In this case, a model of the system must describe the rate of flow over time and the output of the model is presented as a function of time. Systems such as these are often modeled using differential equations. The solution of these equations involves numerical methods that integrate the state of the modeled system over time. This, in essence, involves dividing time into small equal intervals and stepping through time.

Often both the discrete and continuous viewpoints are relevant in modeling a system. For example, if oil tanker arrives at the port to be filled, we have an arrival event that changes the state of the system. This type of modeling situation is called combined continuous discrete modeling.

A system and our resulting model depends on how we characterize the state of the system. The **state** of a system is the set of properties/variables that describe the system at any time  $\{x_1(t), x_2(t), \dots\}$  where  $x_1(t)$  is a variable that represents a system property value at time  $t$ . Variables that take on a countable set of values are **discrete**. Variables that take on an uncountable set of values are said to be **continuous**. For discrete variables, we can define a mapping from the set of integers to each possible value. Even in the discrete case, there may be an infinite number of values. Continuous variables are represented by the set of real numbers. That is, there are an uncountably infinite number of possible values that the variable can take on.

A discrete system has all discrete variables in its state. A continuous system has all continuous variables in its state. A combined continuous-discrete system has both types of variables in defining its state. Consider an airplane:

- If we are interested in the number of parts operating to specification at any time  $t$ , then the state is  $\{N_1(t), N_2(t), \dots\}$  where  $N_1(t) = 1$  if part 1 is operating and 0 if part 1 is not operating to specification. The state vector consists of all discrete variables. This is a discrete system.
- If we are interested in the temperature of each part at time  $t$ , then the state is  $\{T_1(t), T_2(t), \dots\}$ , where  $T_1(t)$  is the temperature in Celsius of part 1 at time  $t$ , etc. The state vector consists of all continuous variables. This is continuous system.
- If we are interested in the velocity of a plane at time  $t$  and the number of wheels deployed at time  $t$ , then the state is  $\{v(t), n(t)\}$  where  $v(t)$  is the velocity of the plane in meters/second and  $n(t)$  is  $\{0, 1, 2, 3, 4\}$  wheels. Then the state vector consists of both continuous and discrete variables that change with time. This is a combined continuous/discrete system.

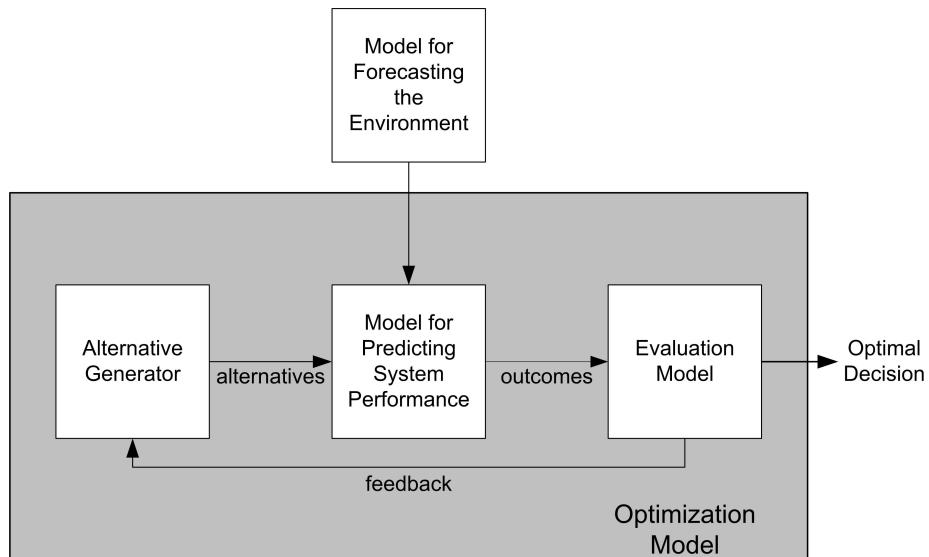
Static systems are systems for which time is not a significant factor. In other words, that the state does not evolve over time. Dynamic systems are systems for which system state changes with respect to time. In a deterministic system, the variables are not governed by underlying random processes. In a stochastic system, some of the variables are governed by underlying random processes. Time can change continuously or at discrete points. When time changes only at discrete points in time, we call these

points, events.

Some simulation languages have modeling constructs for both continuous and discrete modeling; however, this book does not cover the modeling of continuous or combined continuous discrete systems. There are many useful references on this topic. We will be modeling discrete-event dynamic stochastic systems in this textbook.

## 0.4 Simulation: Descriptive or Prescriptive Modeling?

A descriptive model describes how a system behaves. Simulation is at its heart a descriptive modeling technique. Simulation is used to depict the behaviors or characteristics of existing or proposed systems. However, a key use of simulation is to convey the *required* behaviors or properties of a proposed system. In this situation, simulation is used to prescribe a solution. A prescriptive model tells us what to do. In other words, simulation can also be used for prescriptive modeling. Figure 3 illustrates the concept of using simulation to recommend a solution.



**Figure 3:** Using Simulation for Prescriptive Analysis

In the figure, a simulation model is used for predicting the behavior of the system. Input models are used to characterize the system and its environment. An evaluative model is used to evaluate the output of the simulation model to understand how the output compares to desired goals. The alternative generator is used to generate different scenarios to be feed into the simulation model for evaluation. Through a feedback mechanism the inputs can be changed based on the evaluation of the outputs and eventually a recommended solution can be achieved.

For example, in the modeling a drive up pharmacy, suppose that the probability of a

customer waiting longer than 3 minutes in line had to be less than 10%. To form design alternatives, the inputs (e.g. number of pharmacists, possibly the service process) can be varied. Each alternative can then be evaluated to see if the waiting time criteria is met. In this simple situation, you might act as your own alternative generator and the evaluative model is as simple as meeting a criteria; however, in more complex models, there will often be hundreds of inputs to vary and multiple competing objectives. In such situations, simulation optimization and heuristic search methods are often used. This is an active and important area of research within simulation.

## 0.5 Randomness in Simulation

In most real-life situations, the arrival process and the service process occur in a random fashion. Even though the processes may be random, it does not mean that you cannot describe or model the randomness. To have any hope of simulating the situation, you must be able to model the randomness. One of the ways to model this randomness is to describe the phenomenon as a random variable governed by a particular probability distribution. For example, if the arrivals to the bank occur according to a Poisson process, then from probability theory it is known that the distribution of inter-arrival times is an exponential distribution. In general, information about how the customers arrive must be secured either through direct observation of the system or by using historical data. If neither source of information is available, then some plausible assumptions must be made to describe the random process by a probability model.

If historical data is available, there are two basic choices for how to handle the modeling. The first choice is to develop a probability model given the data. The second choice is to try to drive the simulation directly from the historical data. The latter approach is not recommended. First of all, it is extremely unlikely that the captured data will be in a directly usable form. Secondly, it is even more unlikely that the data will be able to adequately represent all the modeling scenarios that you will need through the course of experimenting with the model. For example, suppose that you only have 1 day's worth of arrival data, but you need to simulate a month's worth of system operation. If you simply re-drive your simulation using the 1 day's worth of data, you are not simulating different days! It is much more advisable to develop probability models either from historical data or from data that you capture in developing your model. Appendix .7 discusses some of the tools and techniques for modeling probability distributions.

Once a probability model has been developed, statistical theory provides the means for obtaining random samples based on the use of uniformly distributed random numbers on the interval  $(0,1)$ . These random samples are then used to map the future occurrence of an event on the time scale. For example, if the inter-arrival time is exponential then a random sample drawn from that distribution would represent the time interval until the occurrence of the next arrival. The process of generating random numbers and random variables within simulation is presented in Appendix .3.5.

## 0.6 Simulation Languages

Discrete event simulation normally involves a tremendous volume of computation. Consequently, the use of computers to carry out these computations is essential; however, the volume of computations is not the only obstacle in simulation. If you consider the bank teller example discussed in the previous sections, you will discover that it involves a complex logical structure that requires special expertise before it can be translated into a computer model. Attempting to implement the simulation model, from scratch, in a general purpose language such as FORTRAN, Visual Basic, C/C++, or Java will require above average programming skills. In the absence of specialized libraries for these languages that try to relieve the user from some of the burden, simulation as a tool would be relegated to "elite" programmers. Luckily, the repetitive nature of computations in simulation allows the development of computer libraries that are applicable to simulation modeling situations. For example, libraries or packages must be available for ordering and processing events chronologically, as well as generating random numbers and automatically collecting statistics. Such a library for simulating discrete-event systems in Java is available from the author, see (Rossetti, 2008).

The computational power and storage capacity of computers has motivated the development of specialized simulation languages. Some languages have been developed for continuous or discrete simulations. Others can be used for combined continuous and discrete modeling. All simulation languages provide certain standard programming facilities and will differ in how the user will take advantage of these facilities. There is normally some trade-off between how flexible the language is in representing certain modeling situations. Usually, languages that are highly flexible in representing complex situations require more work (and care) by the user to account for how the model logic is developed. Some languages are more programming oriented (e.g. SIMSCRIPT) and others are more "drag and drop" (e.g. ProModel, Arena, etc.).

The choice of a simulation language is a difficult one. There are many competing languages, each having their own advantages and disadvantages. The Institute for Operations Research and Management Science (INFORMS) often has a yearly product review covering commercial simulation languages, see for example (<http://lionhrtpub.com/orms/>). In addition to this useful comparison, you should examine the Winter Simulation Conference ([www.wintersim.org](http://www.wintersim.org)). The conference has hands on exhibits of simulation software and the conference proceedings often have tutorials for the various software packages. Past proceedings have been made available electronically through the generous support of the INFORMS Society for Simulation (<http://www.informs-sim.org/wscpapers.html>).

was chosen for this textbook because of the author's experience utilizing the software, its ease of use, and the availability of student versions of the software. While all languages have flaws, using a simulation language is essential in performing high performance simulation studies. Most, if not all simulation companies have strong support to assist the user in learning their software. has a strong academic and industrial user base and is very competitive in the simulation marketplace. Once you learn one simu-

lation language well, it is much easier to switch to other languages and to understand which languages will be more appropriate for certain modeling situations.

is fundamentally a process description based language. That is, when using , the modeler describes the process that an "entity" experiences while flowing through or using the elements of the system. You will learn about how facilitates process modeling throughout this textbook.

## 0.7 Simulation Methodology

This section presents a brief overview of the steps of simulation modeling by discussing the process in the context of a methodology. A methodology is simply a series of steps to follow. Since simulation involves systems modeling, a simulation methodology based on the general precepts of solving a problem through systems analysis is presented here. A general methodology for solving problems can be stated as follows:

1. Define the problem
2. Establish measures of performance for evaluation
3. Generate alternative solutions
4. Rank alternative solutions
5. Evaluate and Iterate during process
6. Execute and evaluate the solution

This methodology can be referred to by using the first letter of each step. The DEGREE methodology for problem solving represents a series of steps that can be used during the problem solving process. The first step helps to ensure that you are solving the right problem. The second step helps to ensure that you are solving the problem for the right reason, i.e. your metrics must be coherent with your problem. Steps 3 and 4 ensure that the analyst looks at and evaluates multiple solutions to the problem. In other words, these steps help to ensure that you develop the right solution to the problem. A good methodology recognizes that the analyst needs to evaluate how well the methodology is doing. In step 5, the analyst evaluates how the process is proceeding and allows for iteration. Iteration is an important concept that is foreign to many modelers. The concept of iteration recognizes that the problem solving process can be repeated until the desired degree of modeling fidelity has been achieved. Start the modeling at a level that allows it to be initiated and do not try to address the entire situation in each of the steps. Start with small models that work and build them up until you have reached your desired goals. It is important to get started and get something established on each step and continually go back in order to ensure that the model is representing reality in the way that you intended. The final step is often over looked. Simulation is often used to recommend a solution to a problem. Step 6 indicates that if you have the opportunity you should execute the solution by implementing the decisions. Finally,

you should always follow up to ensure that the projected benefits of the solution were obtained.

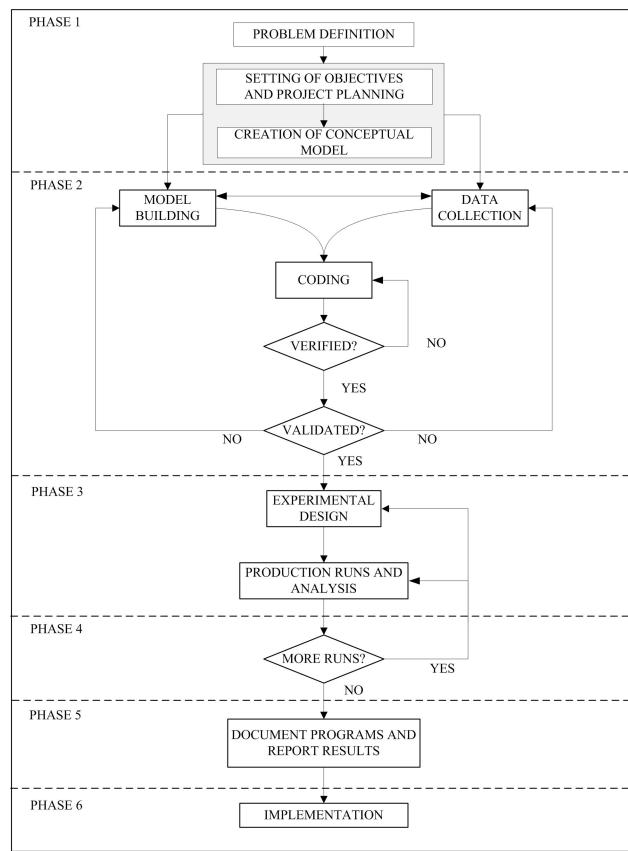
The DEGREE problem solving methodology should serve you well; however, simulation involves certain unique actions that must be performed during the general overall problem solving process. When applying DEGREE to a problem that may require simulation, the general DEGREE approach needs to be modified to explicitly consider how simulation will interact with the overall problem solving process.

Figure 4 represents a refined general methodology for applying simulation to problem solving.

1. Problem Formulation
  1. Define the problem
  2. Define the system
  3. Establish performance metrics
  4. Build conceptual model
  5. Document model assumptions
2. Simulation Model Building
  1. Model translation
  2. Input data modeling
  3. Verification
  4. Validation
3. Experimental Design and Analysis
  1. Preliminary Runs
  2. Final experiments
  3. Analysis of results
4. Evaluate and Iterate
  1. Documentation
  2. Model manual
  3. User manual
5. Implementation

The first phase, problem formulation, captures the essence of the first two steps in the DEGREE process. The second phase, model building, captures the essence of step 3 of the DEGREE process. When building models, you are either explicitly or implicitly developing certain design alternatives. The third phase, experimental design and analysis, encapsulates some of steps 3 and 4 of the DEGREE process. In designing

experiments, design alternatives are specified and when analyzing experiments their worth is being evaluated with respect to problem objectives. The fourth phase, evaluate and iterate, captures the notion of iteration. Finally, the fifth and sixth phases, documentation and implementation complete the simulation process. Documentation is essential when trying to ensure the ongoing and future use of the simulation model, and implementation recognizes that simulation projects often fail if there is no follow through on the recommended solutions.



**Figure 4:** General Simulation Methodology

The problem formulation phase of the study consists of five primary activities:

1. Defining the problem
2. Defining the system
3. Establishing performance metrics.
4. Building conceptual models
5. Documenting modeling assumptions

A problem starts with a perceived need. These activities are useful in developing an appreciation for and an understanding of what needs to be solved. The basic output of the problem definition activity is a problem definition statement. A problem definition statement is a narrative discussion of the problem. A problem definition statement is necessary to accurately and concisely represent the problem for the analyst and for the problem stakeholders. This should include all the required assumptions made during the modeling process. It is important to document your assumptions so that you can examine their effect on the model during the verification, validation, and experimental analysis steps of the methodology. This ensures that the problem is well understood and that all parties agree upon the nature of the problem and the goals of the study. The general goals of a simulation study often include:

- Comparison: To compare system alternatives and their performance measures across various factors (decision variables) with respect to some objectives
- Optimization: This is a special case of comparison in which you try to find the system configuration that optimizes performance subject to constraints.
- Prediction: To predict the behavior of the system at some future point in time.
- Investigation: To learn about and gain insight into the behavior of the system given various inputs.

These general goals will need to be specialized to the problem under study. The problem definition should include a detailed description of the objectives of the study, the desired outputs from the model, and the types of scenarios to be examined or decisions to be made.

The second activity of this phase produces a definition of the system. A system definition statement is necessary to accurately and concisely define the system, particularly its boundaries. The system definition statement is a narrative, which often contains a pictorial representation of the major elements of the system. This ensures that the simulation study is focused on the appropriate areas of interest to the stakeholders and that the scope of the project is well understood.

When defining the problem and the system, one should naturally begin to develop an understanding of how to measure system performance. The third activity of problem formulation makes this explicit by encouraging the analyst to define the required performance measures for the model. To meaningfully compare alternative scenarios, objective and measurable metrics describing the performance of the system are necessary. The performance metrics should include quantitative statistical measures from any models used in the analysis (e.g. simulation models), quantitative measures from the systems analysis, (e.g. cost/benefits), and qualitative assessments (e.g. technical feasibility, human, operational feasibility). The focus should be placed on the performance measures that are considered to be the most important to system decision-makers and tied directly to the objectives of the simulation study. Evaluation of alternatives can then proceed in an objective and unbiased manner to determine which system scenario performs the best according to the decision maker's preferences.

The problem definition statement, the system definition statement, and explicit performance metrics set the stage for more detailed modeling. These activities should be captured in a written form. Within this text, you will develop models of certain “ready-made” book problems. One way to accomplish the problem formulation phase of a simulation study is to consider writing yourself a “book problem”. You will need enough detail in these documents that a simulation analyst (you) can develop a model in any simulation language for the given situation.

With a good understanding of the problem and of the system under study, you should be ready to begin your detailed model formulations. Model formulation does not mean a computer program. You should instead use conceptual modeling tools: conceptual diagrams, flow charts, etc. prior to any use of software to implement a model. The purpose of conceptual modeling tools is to convey a more detailed system description so that the model may be translated into a computer representation. General descriptions help to highlight the areas and processes of the system that the model will simulate. Detailed descriptions assist in simulation model development and coding efforts. Some relevant diagramming constructs include:

1. Context Diagrams: A context diagram assists in conveying the general system description. The diagram is a pictorial representation of the system that often includes flow patterns typically encountered. Context diagrams are often part of the system description document. There are no rules for developing context diagrams. If you have an artistic side, here is your opportunity to shine!
2. Activity Diagrams: An activity diagram is a pictorial representation of the process for an entity and its interaction with resources while within the system. If the entity is a temporary entity (i.e. it flows through the system) the activity diagram is called an activity flow diagram. If the entity is permanent (i.e. it remains in the system throughout its life) the activity diagram is called an activity cycle diagram. Activity diagrams will be used extensively within this text.
3. Software engineering diagrams: Because simulation entails software development, the wide-variety of software engineering diagramming techniques can be utilized to provide information for the model builder. Diagrams such as flow charts, database diagrams, IDEF (ICAM DEFinition language) diagrams, UML (unified modeling language) diagrams, state charts, etc are all useful in documenting complex modeling situations. These techniques assist development and coding efforts by focusing attention on describing, and thus understanding, the elements within the system. Within this text, activity diagrams will be augmented with some simple flow chart symbols and some simple state diagrams will be used to illustrate a variety of concepts.

In your modeling, you should start with an easy conceptual model that captures the basic aspects and behaviors of the system. Then, you should begin to add details, considering additional functionality. Finally, you should always remember that the complexity of the model has to remain proportional to the quality of the available data and the degree of validity necessary to meet the objectives of the study. In other words, don't try to model the world!

After developing a solid conceptual model of the situation, simulation model building can begin. During the simulation model building phase, alternative system design configurations are developed based on the previously developed conceptual models. Additional project planning is also performed to yield specifications for the equipment, resources, and timing required for the development of the simulation models. The simulation models used to evaluate the alternative solutions are then developed, verified, validated, and prepared for analysis. Within the context of a simulation project this process includes:

- Input Data Preparation: Input data is analyzed to determine the nature of the data and to determine further data collection needs. Necessary data is also classified into several areas. This classification establishes different aspects of the model that are used in model development.
- Model Translation: Description of the procedure for coding the model, including timing and general procedures and the translation of the conceptual models into computer simulation program representations.
- Verification: Verification of the computer simulation model is performed to determine whether or not the program performs as intended. To perform model verification, model debugging is performed to locate any errors in the simulation code. Errors of particular importance include improper flow control or entity creation, failure to release resources, and logical/arithmetic errors or incorrectly observed statistics. Model debugging also includes scenario repetition utilizing identical random number seeds, "stressing" the model through a sensitivity analysis (varying factors and their levels) to ensure compliance with anticipated behavior, and testing of individual modules within the simulation code.
- Validation: Validation of the simulation model is performed to determine whether or not the simulation model adequately represents the real system. The simulation model is shown to personnel (of various levels) associated with the system in question. Their input concerning the realism of the model is critical in establishing the validity of the simulation. In addition, further observations of the system are performed to ensure model validity with respect to actual system performance. A simple technique is to statistically compare the output of the simulation model to the output from the real system and to analyze whether there is a significant (and practical) difference between the two.

Model translation will be a large component of each chapter as you learn how to develop simulation models. Verification and validation techniques will not be a major component of this text, primarily because the models will be examples made for educational purposes. This does not mean that you should ignore this important topic. You are encouraged to examine many of the useful references on validation, see for example (Balci, 1997) and (Balci, 1998).

After you are confident that your model has been verified and validated to suit your purposes, you can begin to use the model to perform experiments that investigate the goals and objectives of the project. Preliminary simulation experiments should be performed to set the statistical parameters associated with the main experimental

study. The experimental method should use the simulation model to generate benchmark statistics of current system operations. The simulation model is then altered to conform to a potential scenario and is re-run to generate comparative statistics. This process is continued, cycling through suggested scenarios and generating comparative statistics to allow evaluation of alternative solutions. In this manner, objective assessments of alternative scenarios can be made.

For a small set of alternatives, this “one at a time” approach is reasonable; however, often there are a significant number of design factors that can affect the performance of the model. In this situation, the analyst should consider utilizing formal experimental design techniques. This step should include a detailed specification of the experimental design (e.g. factorial, etc) and any advanced output analysis techniques (e.g. batching, initialization bias prevention, variance reduction techniques, multiple comparison procedures, etc.) that may be required during the execution of the experiments. During this step of the process, any quantitative models developed during the previous steps are exercised. Within the context of a simulation project, the computer simulation model is exercised at each of the design points within the stipulated experimental design.

Utilizing the criteria specified by system decision-makers, and utilizing the simulation model’s statistical results, alternative scenarios should then be analyzed and ranked. A methodology should be used to allow the comparison of the scenarios that have multiple performance measures that trade-off against each other.

If you are satisfied that the simulation has achieved your objectives then you should document and implement the recommended solutions. If not, you can iterate as necessary and determine if any additional data, models, experimentation, or analysis is needed to achieve your modeling objectives. Good documentation should consist of at least two parts: a technical manual, which can be used by the same analyst or by other analysts, and a user manual. A good technical manual is very useful when the project has to be modified, and it can be a very important contribution to software reusability and portability. The approach to documenting the example models in this text can be used as an example for how to document your models. In addition to good model development documentation, often the simulation model will be used by non-analysts. In this situation, a good user manual for how to use and exercise the model is imperative. The user manual is a product for the user who may not be an expert in programming or simulation issues; therefore clearness and simplicity should be its main characteristics. If within the scope of the project, the analyst should also develop implementation plans and follow through with the installation and integration of the proposed solutions. After implementation, the project should be evaluated as to whether or not the proposed solution met the intended objectives.

In this textbook, we will use an open source library for performing stochastic discrete event simulation called the Java Simulation Library (JSL). The next section provides a brief overview.

## 0.8 Overview of the Java Simulation Library

The JSL (a Java Simulation Library) is a simulation library for Java. The JSL's current version has packages that support random number generation, statistical collection, basic reporting, and discrete-event simulation modeling.

The purpose of the JSL is to support education and research within simulation. Current simulation languages hide the implementation details of the workings of a simulation. As such, students exposed to a simulation language such as (ProModel, Arena, and AutoMod, etc.) are able to learn the practice of simulation without having a detailed understanding of the fundamental mechanisms that enable the technology. This has both advantages and disadvantages. The advantage is that more engineers are capable of using and applying simulation technology to improve systems. The disadvantage is that their modeling abilities depend heavily on a particular software package and simulation modeling has the potential to become a black-box technology. I've seen many users who can build complicated models, but have limited or no understanding of how the simulation actually works.

When teaching simulation, especially at the undergraduate level, simulation languages enable students with introductory programming skills to model systems and perform experiments. Within a typical introductory semester course on simulation it is possible to cover the basics of simulation (random numbers, modeling, and statistical analysis) along with the coverage of a tool (Arena, ProModel, AutoMod, etc.) It is difficult to teach students how simulation works based on a general purpose programming language due to the reduced emphasis on more advanced programming skills especially for non-computer science majors. The JSL assists in bridging this gap by providing a standard library in Java for simulation modeling. One of the fundamental design goals of the JSL is to clearly demonstrate how one can implement the fundamental mechanisms that enable simulation technology. By studying the implementation details students will gain a deeper understanding of how simulation works. They will become better simulation modelers, practitioners, and users of commercial simulation languages.

The JSL also supports research within simulation. Naturally, the JSL can be used as a modeling tool to simulate complex systems. Simulators, such as a Supply Chain Simulator, can be built based on the JSL framework. In addition, the design of the JSL provides a framework for the testing of simulation artifacts through a well-defined class and interface hierarchy. The structure of the JSL permits the easy switching of various components within a simulation, the event calendar, random number generator, statistics, etc. For example, the efficiency of different event calendars can be easily tested by simply providing an event calendar to the JSL that implements the interface `CalendarIfc`. Different algorithms can be “plugged into” the framework for testing.

This document presents an overview of a simulation library for Java (JSL). The library is divided into Java packages (calendar, examples, exceptions, modeling, observers, testing, and utilities). As necessary, these packages may contain sub-packages, which implement various aspects of the library. The JSL is organized as three open source projects: `JSLCore`, `JSLEExamples`, and `JSLEExtensions`. Each of these projects is further

organized into java packages.

JSLCore - jsl is the main package that holds all sub-packages

- calendar - The calendar package implements classes that provide event calendar processing
- simulation - The simulation package implements the main classes involved in constructing and running simulation.
- modeling - The modeling package is the heart of the JSL. In the modeling package, supporting model elements such as queues, resources, variables, commands, etc. are implemented. This package will be discussed in detail in this document.
  - observers - The observers package provides support classes for observing model elements during the simulation run. The JSL is designed to allow each model element to have observers attached to it thereby implementing the classic Observer pattern. Observers can be used to collect statistics, write output to files, display model element state, etc. Proper understanding of the observer pattern is essential in maximizing the use of the JSL.
  - utilities - The utilities package provides support classes that are used by the JSL. The random, statistics, reporting, and database related sub-packages will be discussed in this document.

JSLExtensions - jslx is the main package that holds all sub-packages

- dbutilities - The dbutilities package provides support for using databases with the JSL. It also includes some functionality related to csv and Excel files
- fxutilities - The fxutilities provides some simple support for javafx
- statistics - The statistics package provide additional statistical functionality that leverages the additional classes available within the jslx package.

JSLEXamples - examples is the main package that holds all sub-packages

- entity - examples related to modeling entity flow and process description
- hospitalward - an example of modeling a hospital ward
- inventoy - some simple inventory policy models
- jobshop - models the classic job shop example from Law and Kelton's textbook
- jockeying - models queues with jockeying of customers
- modelement - examples related to the use of model elements
- models - many example models
- montecarlo - examples focused on using random numbers and Monte Carlo methods
- queueing - examples related to systems involving queues
- resource - examples that illustrate resource constructs
- spatial - examples related to the spatial package in JSLCore
- station - examples illustrating the station package in JSLCore
- utilities - miscellaneous examples of using constructs in the utilities
- variables - examples of using variables within models

The purpose of the JSL is to provide support for the development of discrete-event sim-

ulation programs within Java. This document provides an overview of the functionality and use of the classes found within the JSL. Additional information is available through the JavaDoc API. We begin our discussion with the utilities package within the JSLCore. These support packages can be used independently of building discrete-event simulation models.

## 0.9 Exercises

---

**Exercise 0.1.** Using the resources at (<http://www.informs-sim.org/wscpapers.html>) find an application of simulation to a real system and discuss why simulation was important to the analysis.

---

**Exercise 0.2.** Customers arrive to a gas station with two pumps. Each pump can reasonably accommodate a total of two cars. If all the space for the cars is full, potential customers will balk (leave without getting gas). What measures of performance will be useful in evaluating the effectiveness of the gas station? Describe how you would collect the inter-arrival and service times of the customers necessary to simulate this system.

---

**Exercise 0.3.** Classify the systems as either being discrete or continuous:

- Electrical Capacitor (You are interested in modeling the amount of current in a capacitor at any time  $t$ ).
  - On-line gaming system. (You are interested in modeling the number of people playing Halo 4 at any time  $t$ .)
  - An airport. (You are interested in modeling the percentage of flights that depart late on any given day).
- 

**Exercise 0.4.** Classify the systems as either being discrete or continuous:

- Parking lot
  - Level of gas in Fayetteville shale deposit
  - Printed circuit board manufacturing facility
- 

**Exercise 0.5.** Classify the systems as either being discrete or continuous:

- Classify the systems as either being discrete or continuous:
  - Elevator system (You are interested in modeling the number of people waiting on each floor and traveling within the elevators.)
  - Judicial system (You are interested in modeling the number of cases waiting for trial.)
  - The in-air flight path of an airplane as it moves from an origin to a destination.
- 

**Exercise 0.6.** What is model conceptualization? Give an example of something that might be produced during model conceptualization.

---

**Exercise 0.7.** The act of implementing the model in computer code, including timing and general procedures and the representation of the conceptual model into a computer simulation program is called: \_\_\_\_\_.

---

**Exercise 0.8.** Which of the following does the problem formulation phase of simulation not include? - Define the system - Establish performance metrics - Verification - Build conceptual models

---

**Exercise 0.9.** *Fill in the blank* The general goals of a simulation include the \_\_\_\_\_ of system alternatives and their performance measures across various factors (decision variables) with respect to some objectives.

---

**Exercise 0.10.** *Fill in the blank* The general goals of a simulation include the \_\_\_\_\_ of system behavior at some future point in time.

---

**Exercise 0.11.** *True or False* Verification of the simulation model is performed to determine whether the simulation model adequately represents the real system.

---



# Random Number Generation

## LEARNING OBJECTIVES

- To be able to generate random numbers using the Java Simulation Library (JSL)
- To understand how to control random number streams within the JSL

## 0.10 Random Number Generator

This section discusses how to random number generation is implemented within the JSL. The purpose is to present how these concepts can be put into practice.

The random number generator used within the JSL is described in L'Ecuyer et al. (2002) and has excellent statistical properties. It is based on the combination of two multiple recursive generators resulting in a period of approximately  $3.1 \times 10^{57}$ . This is the same generator that is now used in many commercial simulation packages. The generator used in the JSL is defined by the following equations.

$$\begin{aligned} R_{1,i} &= (1,403,580R_{1,i-2} - 810,728R_{1,i-3}) \bmod (2^{32} - 209) \\ R_{2,i} &= (527,612R_{2,i-1} - 1,370,589R_{2,i-3}) \bmod (2^{32} - 22,853) \\ Y_i &= (R_{1,i} - R_{2,i}) \bmod (2^{32} - 209) \\ U_i &= \frac{Y_i}{2^{32} - 209} \end{aligned}$$

To illustrate how this generator works, consider generating an initial sequence of pseudo-random numbers from the generator. The generator takes as its initial seed a vector of six initial values  $(R_{1,0}, R_{1,1}, R_{1,2}, R_{2,0}, R_{2,1}, R_{2,2})$ . The first initially generated value,  $U_i$ , will start at index 3. To produce five pseudo random numbers using this generator we need an initial seed vector, such as:  
 $\{R_{1,0}, R_{1,1}, R_{1,2}, R_{2,0}, R_{2,1}, R_{2,2}\} = \{12345, 12345, 12345, 12345, 12345, 12345\}$

Using the recursive equations, the resulting random numbers are as follows:

	i=3	i=4	i=5	i=6	i=7
$Z_{1,i-3} =$	12345	12345	12345	3023790853	3023790853
$Z_{1,i-2} =$	12345	12345	3023790853	3023790853	3385359573
$Z_{1,i-1} =$	12345	3023790853	3023790853	3385359573	1322208174
$Z_{2,i-3} =$	12345	12345	12345	2478282264	1655725443
$Z_{2,i-2} =$	12345	12345	2478282264	1655725443	2057415812
$Z_{2,i-1} =$	12345	2478282264	1655725443	2057415812	2070190165
$Z_{1,i} =$	3023790853	3023790853	3385359573	1322208174	2930192941
$Z_{2,i} =$	2478282264	1655725443	2057415812	2070190165	1978299747
$Y_i =$	545508589	1368065410	1327943761	3546985096	951893194
$U_i =$	0.127011122076	0.318527565471	0.309186015655	0.82584686312	0.221629915834

While it is beyond the scope of this document to explore the theoretical underpinnings of this generator, it is important to note that the generator allows multiple independent streams to be defined along with sub-streams.

The fantastic thing about this generator is the sheer size of the period. Based on their analysis, L'Ecuyer et al. (2002) state that it will be “approximately 219 years into the future before average desktop computers will have the capability to exhaust the cycle of the (generator) in a year of continuous computing.” In addition to the period length, the generator has an enormous number of streams, approximately  $1.8 \times 10^{19}$  with stream lengths of  $1.7 \times 10^{38}$  and sub-streams of length  $7.6 \times 10^{22}$  numbering at  $2.3 \times 10^{15}$  per stream. Clearly, with these properties, you do not have to worry about overlapping random numbers when performing simulation experiments. The generator was subjected to a rigorous battery of statistical tests and is known to have excellent statistical properties.

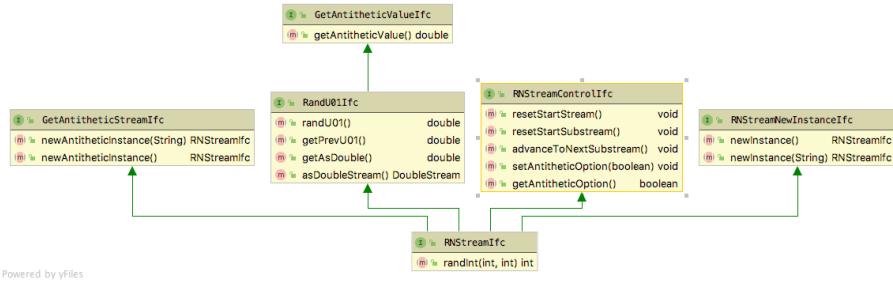
## 0.11 Random Package

The concepts within L'Ecuyer et al. (2002) have been implemented within the `jsl.utilities.random.rng`<sup>16</sup> package in the JSL. A key organizing principle for the `random` package is the use of Java interfaces. A Java interface allows classes to act like other classes. It is a mechanism by which a class can promise to have certain behaviors (i.e. methods). The JSL utilizes interfaces to separate random number generation concepts from stream control concepts.

Figure 5 shows the important interfaces within the `jsl.utilities.random.rng` package. The `RandU01Ifc` defines the methods for getting the next pseudo-random number and the previous pseudo-random number via `randU01()` and `getPrevU01()`. The `randInt(int i, int j)` method can be used to generate a random integer uniformly over the range from  $i$  to  $j$ . The methods `getAsDouble()` and `asDoubleStream()` permit the random number stream to act as a Java stream as defined within the Java Stream API<sup>17</sup>. The `GetAn-`

<sup>16</sup><https://rossetti.git-pages.uark.edu/JSL-Documentation/jsl/utilities/random/rng/package-summary.html>

<sup>17</sup><https://docs.oracle.com/javase/8/docs/api/?java/util/stream/Stream.html>



**Figure 5:** Random Number Stream Interfaces

`GetAntitheticStreamIfc` and `RNStreamNewInstanceIfc` interfaces allow a new object instance to be created from the stream. In the case of the `GetAntitheticStreamIfc` interface the created stream will produce antithetic variates from the stream. If  $U$  is a pseudo-random number, then  $1 - U$  is the antithetic variate of  $U$ .

The `RNStreamControlIfc` defines methods for controlling the underlying stream of pseudo-random numbers.

- `resetStartStream()` - positions the random number generator at the beginning of its stream. This is the same location in the stream as assigned when the random number generator was created and initialized.
- `resetStartSubstream()` - resets the position of the random number generator to the start of the current substream. If the random number generator has advanced into the substream, then this method resets to the beginning of the substream.
- `advanceToNextSubStream()` - positions the random number generator at the beginning of its next substream. This method move through the current substream and positions the generator at the beginning of the next substream.
- `setAntitheticOption(boolean flag)` - if the flag is true, the generator should start producing antithetic variates with the next call to `randU01()`. If the flag is false, the generator should stop producing antithetic variates.
- `getAntitheticOption()` - returns whether the antithetic option has been set.

The `RNStreamIfc` interface assumes that the underlying pseudo-random number generator can produce multiple streams that can be further divided into substreams. The reset methods allow the user to move within the streams. Classes that implement the `RNStreamControlIfc` can manipulate the streams in a well-defined manner.

To create an concrete instance of a stream, we must have a random number stream provider. This functionality is defined by the `RNStreamProviderIfc` interface and its concrete implementation, `RNStreamProvider`. Figure 6 illustrates the functionality available for creating random number streams. This interface conceptualizes the creation of random number streams as a process of making a sequence of streams numbered 1, 2, 3, ...

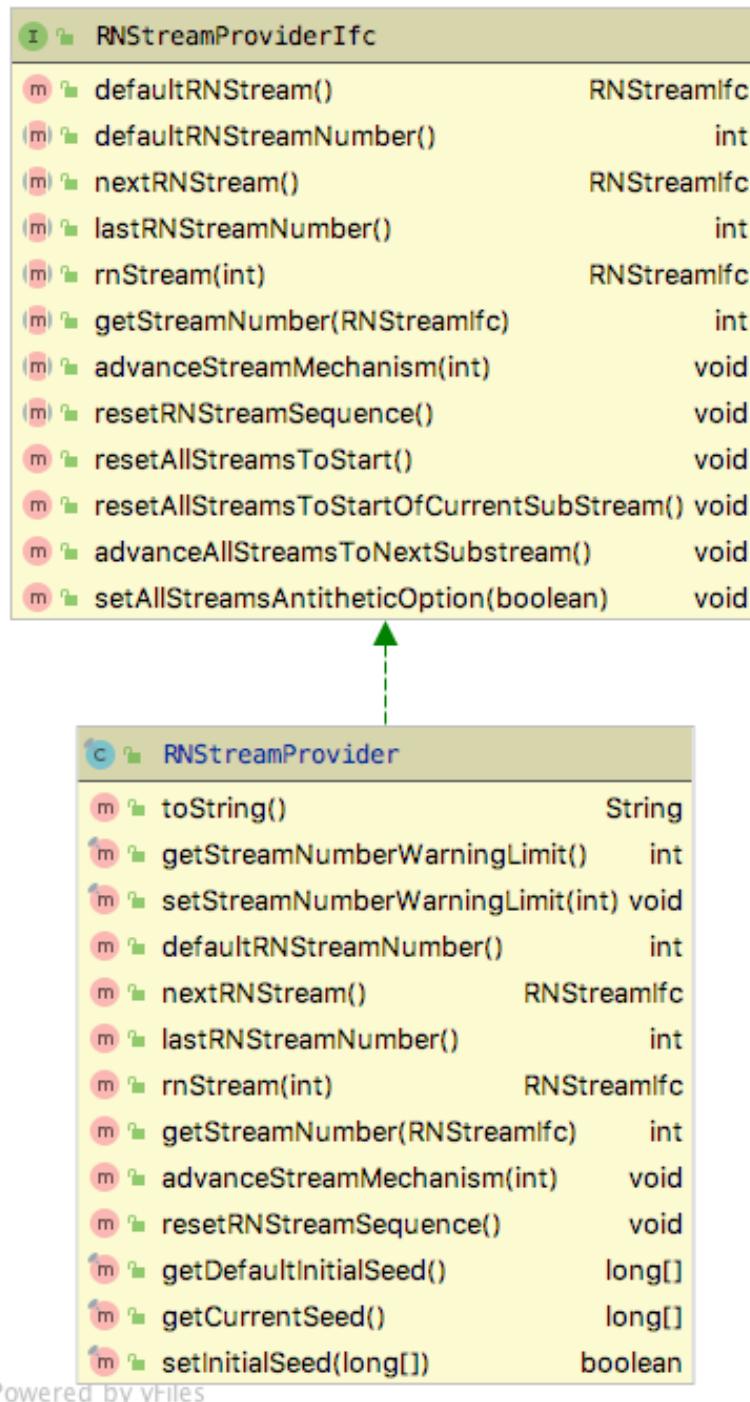


Figure 6: RNStreamProviderIfc Interface

A random number stream provider must define a default stream, which can be retrieved via the `defaultRNStream()` method. For the JSL, the default stream is the first stream created and is labeled with the sequence number 1. The sequence number of a stream can be used to retrieve a particular stream from the provider. The following methods allow for creation and access to streams.

- `nextRNStream()` - returns the next random number stream associated with the provider. Each call to `nextRNStream()` makes a new stream in the sequence of streams.
- `lastRNStreamNumber()` - returns the number of the stream that was last made. This indicates how many streams have been made. If 0 is returned, then no streams have been made by the provider.
- `rnStream(int k)` - returns the  $k^{th}$  stream. If  $k$  is greater than `lastRNStreamNumber()` then `lastRNStreamNumber()` is advanced according to the additional number of streams by creating any intermediate streams. For example, if `lastRNStreamNumber() = 10` and  $k = 15$ , then streams 11, 12, 13, 14, 15 are assumed provided and stream 15 is returned and `lastRNStreamNumber()` now equals 15. If  $k$  is less than or equal to `lastRNStreamNumber()`, then no new streams are created and `lastRNStreamNumber()` stays at its current value and the  $k^{th}$  stream is returned.
- `getStreamNumber(RNStreamIfc stream)` - returns the stream number of the instance of a stream.
- `advanceStreamMechanism(int n)` - advances the underlying stream mechanism by the specified number of streams, without actually creating the streams. The value of `lastRNStreamNumber()` remains the same after advancing through the streams. In other words, this method should act as if `nextRNStream()` was not called but advance the underlying stream mechanism as if  $n$  streams had been provided.
- `resetRNStreamSequence()` - Causes the random number stream provider to act as if it has never created any streams. Thus, the next call to `nextRNStream()` will return the 1<sup>st</sup> stream.

The random number stream provider also facilitates the control of all streams that have been created. This functionality is similar to how the position within an individual stream can be manipulated, except the provider performs the functionality on all streams that it has created. The following methods perform this function.

- `resetAllStreamsToStart()` - resets all created streams to the start of their stream.
- `resetAllStreamsToStartOfCurrentSubStream()` - resets all created streams to the start of their current sub-stream.
- `advanceAllStreamsToNextSubstream()` - advances all created streams to the start of their next sub-stream.
- `setAllStreamsAntitheticOption(boolean option)` - changes all created streams to have their antithetic option either off = false or on = true.

Many random number generators require the specification of a seed to start the generated sequence. Even though the generator within the JSL use seeds, there really is not any need to utilize the seeds because of the well defined methods for moving within the streams. Now, let's illustrate how to create and manipulate streams.

### 0.11.1 Creating and Using Streams

To create a random number stream, the user must utilize an instance of `RNStreamProvider`. This process is illustrated in the following code. This code creates two instances of `RNStreamProvider` and gets the first stream from each instance. The instances of `RNStreamProvider` use the exact same underlying default seeds. Thus, they produce *exactly the same* sequence of streams.

```
// make a provider for creating streams
RNStreamProvider p1 = new RNStreamProvider();
// get the first stream from the provider
RNStreamIfc p1s1 = p1.nextRNStream();
// make another provider, the providers are identical
RNStreamProvider p2 = new RNStreamProvider();
// thus the first streams returned are identical
RNStreamIfc p2s1 = p2.nextRNStream();
System.out.printf("%3s %15s %15s %n", "n", "f1s1", "f2s2");
for (int i = 0; i < 5; i++) {
    System.out.printf("%3d %15f %15f %n", i, p1s1.randU01(), p2s1.randU01());
}
```

Thus, in the following code output, the randomly generated values are exactly the same for the two streams.

n	p1s1	p2s2
1	0.728510	0.728510
2	0.965587	0.965587
3	0.996184	0.996184
4	0.114988	0.114988
5	0.973145	0.973145

There is really very little need for the general programmer to create a `RNStreamProvider` because the JSL supplies default provider that can be used to provide a virtually infinite number of streams. The need for directly accessing the functionality of `RNStreamProvider` is for very fine control of stream creation in such situations like running code on different computers in parallel. While the providers produce the same streams, you can force one provider to be different from another provider by manipulating the seeds. In addition, the provider can control all streams that it produces. So, unless you are trying to do some advanced work that involves coordinating multiple streams, you should not need to create multiple instances of `RNStreamProvider`.

Because the most common use case is to just have a single provider of streams, the JSL facilitates this through the `JSLRandom` class. The `JSLRandom` class has a wide range of static methods to facilitate random variate generation. The most important methods include:

- `nextRNStream()` - calls the underlying default `RNStreamProvider` to create a new ran-

dom number stream

- `rnStream(int k)` - returns the  $k^{th}$  stream from the default `RNstreamProvider`
- `getDefaultRNStream()` - calls the underlying default `RNstreamProvider` for its default stream

In the following code example, these methods are used to create streams that are used to generate random numbers. The first line of the code uses the static method `getDefau-ltRNStream()` of `JSLRandom` to get the default stream and then generates three random numbers. The stream is then advanced and three new random numbers are generated. Then, the stream is reset to its starting (initial seed) and it then repeats the original values. Finally, the a new stream is created via `JSLRandom.nextRNStream()` and then used to generate new random numbers. From a conceptual standpoint, each stream contains an independent sequence of random numbers from any other stream (unless of course they are made from different providers). They are conceptually infinite and independent due to their enormous periods.

```
RNStreamIfc s1 = JSLRandom.getDefaultRNStream();
System.out.println("Default stream is stream 1");
System.out.println("Generate 3 random numbers");
for (int i = 1; i <= 3; i++) {
    System.out.println("u = " + s1.randU01());
}
s1.advanceToNextSubstream();
System.out.println("Advance to next sub-stream and get some more random numbers");
for (int i = 1; i <= 3; i++) {
    System.out.println("u = " + s1.randU01());
}
System.out.println("Notice that they are different from the first 3.");
s1.resetStartStream();
System.out.println("Reset the stream to the beginning of its sequence");
for (int i = 1; i <= 3; i++) {
    System.out.println("u = " + s1.randU01());
}
System.out.println("Notice that they are the same as the first 3.");
System.out.println("Get another random number stream");
RNStreamIfc s2 = JSLRandom.nextRNStream();
System.out.println("2nd stream");
for (int i = 1; i <= 3; i++) {
    System.out.println("u = " + s2.randU01());
}
System.out.println("Notice that they are different from the first 3.");
```

The resulting output from this code is as follows. Again, the methods of the `RNStreamControlIfc` interface that permit movement within a stream are extremely useful for controlling the randomness associated with a simulation.

```

Default stream is stream 1
Generate 3 random numbers
u = 0.12701112204657714
u = 0.3185275653967945
u = 0.3091860155832701
Advance to next sub-stream and get some more random numbers
u = 0.0793989897973463
u = 0.4803395047575741
u = 0.8583222470551328
Notice that they are different from the first 3.
Reset the stream to the beginning of its sequence
u = 0.12701112204657714
u = 0.3185275653967945
u = 0.3091860155832701
Notice that they are the same as the first 3.
Get another random number stream
2nd stream
u = 0.7285097861965271
u = 0.9655872822837334
u = 0.9961841304801171
Notice that they are different from the first 3.

```

### 0.11.2 Common Random Numbers

Common random numbers (CRN) is a Monte Carlo method that has different experiments utilize the same random numbers. CRN is a variance reduction technique that allows the experimenter to block out the effect of the random numbers used in the experiment. To facilitate the use of common random numbers the JSL has the aforementioned stream control mechanism. One way to implement common random numbers is to use two instances of `RNStreamProvider` as was previously illustrated. In that case, the two providers produce the same sequence of streams and thus those streams can be used on the different experiments. An alternative method that does not require the use of two providers is to create a copy of the stream directly from the stream instance. The following code clones the stream instance.

```

// get the default stream
RNStreamIfc s = JSLRandom.getDefaultRNStream();
// make a clone of the stream
RNStreamIfc clone = s.newInstance();
System.out.printf("%3s %15s %15s %n", "n", "U", "U again");
for (int i = 0; i < 3; i++) {
    System.out.printf("%3d %15f %15f %n", i+1, s.randU01(), clone.randU01());
}

```

Since the instances have the same underlying state, they produce the same random

numbers. Please note that the cloned stream instance is not produced by the underlying `RNStreamProvider` and thus it is not part of the set of streams managed or controlled by the provider.

n	U	U again
1	0.127011	0.127011
2	0.318528	0.318528
3	0.309186	0.309186

An alternative method is to just use the `resetStartStream()` method of the stream to reset the stream to the desired location in its sequence and then reproduce the random numbers. This is illustrated in the following code.

```
RNStreamIfc s = JSLRandom.getDefaultRNStream();
// generate regular
System.out.printf("%3s %15s %n", "n", "U");
for (int i = 0; i < 3; i++) {
    double u = s.randU01();
    System.out.printf("%3d %15f %n", i+1, u);
}
// reset the stream and generate again
s.resetStartStream();
System.out.println();
System.out.printf("%3s %15s %n", "n", "U again");
for (int i = 0; i < 3; i++) {
    double u = s.randU01();
    System.out.printf("%3d %15f %n", i+1, u);
}
```

Notice that the generated numbers are the same.

n	U
1	0.127011
2	0.318528
3	0.309186
n	U again
1	0.127011
2	0.318528
3	0.309186

Thus, a experiment can be executed, then the random numbers reset to the desired location. Then, by changing the experimental conditions and re-running the simulation, the same random numbers are used. If many streams are used, then by accessing the `RNStreamProvider` you can reset all of the controlled streams with one call and then perform the next experiment.

### 0.11.3 Creating and Using Antithetic Streams

Recall that if a pseudo-random number is called  $U$  then its antithetic value is  $1 - U$ . There are a number of methods to access antithetic values. The simplest is to create an antithetic instance from a given stream. This is illustrated in the following code. Please note that the antithetic stream instance is not produced by the underlying `RNStreamProvider` and thus it is not part of the set of streams managed or controlled by the provider. The new instance process directly creates the new stream based on the current stream so that it has the same underling state and it is set to produce antithetic values.

```
// get the default stream
RNStreamIfc s = JSRandom.getDefaultRNStream();
// make its antithetic version
RNStreamIfc as = s.newAntitheticInstance();
System.out.printf("%3s %15s %15s %15s %n", "n", "U", "1-U", "sum");
for (int i = 0; i < 5; i++) {
    double u = s.randU01();
    double au = as.randU01();
    System.out.printf("%3d %15f %15f %15f %n", i+1, u, au, (u+au));
}
```

Notice that the generated values sum to 1.0.

n	U	1-U	sum
1	0.127011	0.872989	1.000000
2	0.318528	0.681472	1.000000
3	0.309186	0.690814	1.000000
4	0.825847	0.174153	1.000000
5	0.221630	0.778370	1.000000

An alternate method that does not require the creation of another stream involves using the `resetStartStream()` and `setAntitheticOption(boolean flag)` methods of the current stream. If you have a stream, you can use the `setAntitheticOption(boolean flag)` to cause the stream to start producing antithetic values. If you use the `resetStartStream()` method and then set the antithetic option to true, the stream will be set to its initial starting point and then produce antithetic values.

```
RNStreamIfc s = JSRandom.getDefaultRNStream();
// generate regular
System.out.printf("%3s %15s %n", "n", "U");
for (int i = 0; i < 5; i++) {
    double u = s.randU01();
    System.out.printf("%3d %15f %n", i+1, u);
}
// generate antithetic
```

```
s.resetStartStream();
s.setAntitheticOption(true);
System.out.println();
System.out.printf("%3s %15s %n", "n", "1-U");
for (int i = 0; i < 5; i++) {
    double u = s.randU01();
    System.out.printf("%3d %15f %n", i+1, u);
}
```

Notice that the second set of random numbers is the complement of the first set in this output. Of course, you can also create multiple instances of `RNStreamProvider`, and then create streams and set one of the streams to produce antithetic values.

n	U
1	0.127011
2	0.318528
3	0.309186
4	0.825847
5	0.221630

n	1-U
1	0.872989
2	0.681472
3	0.690814
4	0.174153
5	0.778370

## 0.12 Frequently Asked Questions

1. **What are pseudo-random numbers?** Numbers generated through an algorithm that appear to be random, when in fact, they are created by a deterministic process.
2. **Why do we want to control randomness within simulation models?** By controlling randomness, we can better ascertain if changes in simulation responses are due to factors of interest or due to underlying statistical variation caused by sampling. Do you think that it is better to compare two systems using the same inputs or different inputs? Suppose we have a work process that we have redesigned. We have the old process and the new process. Would it be better to test the difference in the process by using two different workers or the same worker? Most people agree that using the same worker is better. This same logic applies to randomness. Since we can control which pseudo-random number we use, it is better to test the difference between two model alternatives by using the same pseudo-random numbers. We use seeds and streams to do this.
3. **What are seeds and streams?** A random number stream is a sub-sequence of

pseudo-random numbers that start at particular place with a larger sequence of pseudo-random numbers. The starting point of a sequence of pseudo-random numbers is called the seed. A seed allows us to pick a particular stream. Having multiple streams is useful to assign different streams to different sources of randomness within a model. This facilitates the control of the use of pseudo-random numbers when performing experiments.

4. **How come my simulation results are always the same?** Random number generators in computer simulation languages come with a default set of streams that divide the “circle” up into independent sets of random numbers. The streams are only independent if you do not use up all the random numbers within the subsequence. These streams allow the randomness associated with a simulation to be controlled. During the simulation, you can associate a specific stream with specific random processes in the model. This has the advantage of allowing you to check if the random numbers are causing significant differences in the outputs. In addition, this allows the random numbers used across alternative simulations to be better synchronized. Now a common question in simulation can be answered. That is, “If the simulation is using random numbers, why to I get the same results each time I run my program?” The corollary to this question is, “If I want to get different random results each time I run my program, how do I do it?” The answer to the first question is that the underlying random number generator is starting with the same seed each time you run your program. Thus, your program will use the same pseudo random numbers today as it did yesterday and the day before, etc. The answer to the corollary question is that you must tell the random number generator to use a different seed (or alternatively a different stream) if you want different invocations of the program to produce different results. The latter is not necessarily a desirable goal. For example, when developing your simulation programs, it is desirable to have repeatable results so that you can know that your program is working correctly.
5. **How come my simulation results are unexpectedly different?** Sometimes by changing the order of method calls you change the sequence of random numbers that are assigned to various things that happen in the model (e.g. attribute, generated service times, paths taken, etc.). Please see the FAQ “How come my results are always the same?”. Now, the result can sometimes be radically different if different random numbers are used for different purposes. By using streams, you reduce this possibility and increase the likelihood that two models that have different configurations will have differences due to the change and not due to the random numbers used.

# Random Variate Generation and Probability Modeling

## LEARNING OBJECTIVES

- To be able to generate random variates using the Java Simulation Library (JSL)
- To understand how to use the JSL for basic probability computations

The JSL has the capability to generate random variates from both discrete and continuous distributions. The `jsl.utilities.random.rvariable`<sup>18</sup> package supports this functionality. The package has a set of Java interfaces that define the behavior associated with random variables. Concrete sub-classes of specific random variables are created by sub-classing `AbstractRVariable`. As shown in Figure 7, every random variable has access to an object that implements the `RNStreamIfc` interface. This gives it the ability to generate pseudo-random numbers and to control the streams. The `GetValueIfc` interface is the key interface because in this context it returns a random value from the random variable. For example, if `d` is a reference to an instance of a sub-class of type `AbstractRVariable`, then `d.getValue()` generates a random value.

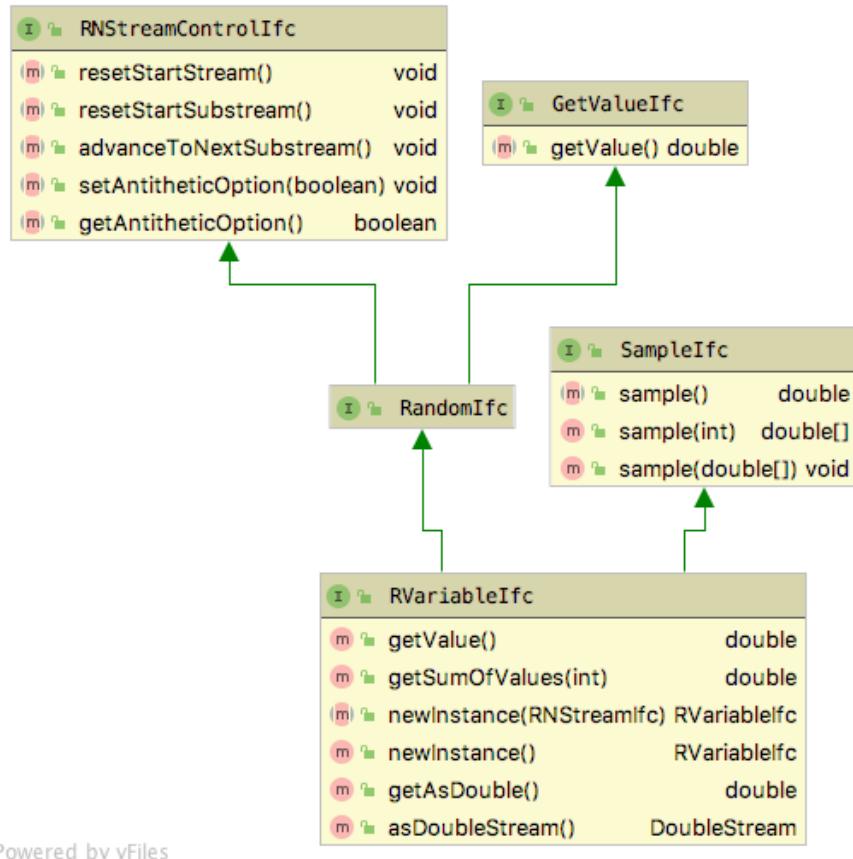
## 0.13 Continuous and Discrete Random Variables

The names and parameters associated with the continuous random variables are as follows:

- `BetaRV(double alpha1, double alpha2)`
- `ChiSquaredRV(double degreesOfFreedom)`
- `ExponentialRV(double mean)`
- `GammaRV(double shape, double scale)`
- `GeneralizedBetaRV(double alpha1, double alpha2, double min, double max)`
- `JohnsonBRV(double alpha1, double alpha2, double min, double max)`
- `LaplaceRV(double mean, double scale)`
- `LogLogisticRV(double shape, double scale)`

---

<sup>18</sup><https://rossetti.git-pages.uark.edu/JSL-Docu.../random/rvariable/package-summary.html>



**Figure 7:** Random Variable Interfaces

- LognormalRV(double mean, double variance)
- NormalRV(double mean, double variance)
- PearsonType5RV(double shape, double scale)
- PearsonType6RV(double alpha1, double alpha2, double beta)
- StudentTRV(double dof)
- TriangularRV(double min, double mode, double max)
- UniformRV(double lowerLimit, double upperLimit)
- WeibullRV(double shape, double scale)

The names and parameters associated with the discrete random variables are as follows:

- BernoulliRV(double prob)
- BinomialRV(double prob, int numTrials)
- ConstantRV(double value), a degenerate probability mass on a single value that

cannot be changed

- `DEmpiricalRV(double[] value, double[] cdf)`, the value array holds the values that can be returned, while the cdf array holds the cumulative probabilities associated with the values
- `DUniformRV(int minimum, int maximum)`
- `GeometricRV(double prob)`, range is 0 to infinity
- `NegativeBinomialRV(double prob, double numSuccess)`, range is 0 to infinity. The number of failures before the  $r^{th}$  success.
- `PoissonRV(double mean)`
- `ShiftedGeometricRV(double prob)`, range is 1 to infinity
- `VConstantRV(double value)`, a degenerate probability mass on a single value that can be changed

## 0.14 Overview of Generation Algorithms

As you can see, the name of the distribution followed by the letters RV designate the class names. Implementations of these classes extend the `AbstractRVariable` class, which implements the `RVariableIfc` interface. Users simply create an instance of the class and then use it to get a sequence of values that have the named probability distribution. In order to implement a new random variable (i.e. some random variable that is not already implemented) you can extend the class `AbstractRVariable`. This provides a basic template for what is expected in the implementation. However, it implies that you need to implement all of the required interfaces. The key method to implement is the protected `generate()` method, which should return the generated random value.

In almost all cases, the JSL utilizes the inverse transform method for generating random variates. Thus, there is a one to one mapping of the underlying pseudo-random number and the resulting random variate. Even in the case of distributions that do not have closed form inverse cumulative distribution functions, the JSL utilizes numerical methods to approximate the function whenever feasible. For example, the JSL uses a rational function approximation, see Cody (1969), to implement the inverse cumulative distribution function for the standard normal distribution. The inversion for the gamma distribution is based on Algorithm AS 91 for inverting the chi-squared distribution and exploiting its relationship with the gamma. The beta distribution also uses numerical methods to compute the cumulative distribution function as well as bi-section search to determine the inverse for cumulative distribution function.

The JSL implements the `BernoulliRV`, `DUniformRV`, `GeometricRV`, `NegativeBinomialRV`, and `ShiftedGeometricRV` classes using the methods described in Chapter 2 of Rossetti (2015). While more efficient methods may be available, the `PoissonRV` and `BinomialRV` distributions are implemented by searching the probability mass functions. Both search methods use an approximation to get close to the value of the inverse and then search up or down through the cumulative distribution function. Because of this both distributions use numerically stable methods to compute the cumulative distribution function values. The `DEmpiricalRV` class also searches through the cumulative distribution

function.

## 0.15 Creating and Using Random Variables

The following example code illustrates how to create a normal random variable and how to generate values.

```
// create a normal mean = 20.0, variance = 4.0 random variable
NormalRV n = new NormalRV(20.0, 4.0);
System.out.printf("%3s %15s %n", "n", "Values");
// generate some values
for (int i = 0; i < 5; i++) {
    // getValue() method returns generated values
    double x = n.getValue();
    System.out.printf("%3d %15f %n", i+1, x);
}
```

The resulting output is what you would expect.

n	Values
1	21.216624
2	23.639128
3	25.335884
4	17.599163
5	23.858350

Alternatively, the user can use the `sample()` method to generate an array of values that can be later processed. The following code illustrates how to do that with a triangular distribution.

```
// create a triangular random variable with min = 2.0, mode = 5.0, max = 10.0
TriangularRV t = new TriangularRV(2.0, 5.0, 10.0);
// sample 5 values
double[] sample = t.sample(5);
System.out.printf("%3s %15s %n", "n", "Values");
for (int i = 0; i < sample.length; i++) {
    System.out.printf("%3d %15f %n", i+1, sample[i]);
}
```

Again, the output is what we would expect.

n	Values
1	3.515540
2	6.327783
3	4.382075
4	7.392228

5            8.409238

It is important to note that the full range of functionality related to stream control is also available for random variables. That is, the underlying stream can be reset to its start, can be advanced to the next substream, can generate antithetic variates, etc. Each new instance of a random variable is supplied with its own unique stream that is not shared with another other random variable instances. Since this underlying random number generator has an enormous number of streams, approximately  $1.8 \times 10^{19}$ , it is very unlikely that the user will create so many streams as to start reusing them. However, the streams that are used by random variable instances can be supplied directly so that they may be shared. The following following code example illustrates how to assign a specific stream by passing a specific stream instance into the constructor of the random variable.

```
// get stream 3
RNStreamIfc stream = JSLRandom.rnStream(3);
// create a normal mean = 20.0, variance = 4.0, with the stream
NormalRV n = new NormalRV(20.0, 4.0, stream);
System.out.printf("%3s %15s %n", "n", "Values");
for (int i = 0; i < 5; i++) {
    // getValue() method returns generated values
    double x = n.getValue();
    System.out.printf("%3d %15f %n", i+1, x);
}
```

As a final example, the discrete empirical distribution requires a little more setup. The user must supply the set of values that can be generated as well as an array holding the cumulative distribution probability across the values. The following code illustrates how to do this.

```
// values is the set of possible values
double[] values = {1.0, 2.0, 3.0, 4.0};
// cdf is the cumulative distribution function over the values
double[] cdf = {1.0/6.0, 3.0/6.0, 5.0/6.0, 1.0};
//create a discrete empirical random variable
DEmpiricalRV n1 = new DEmpiricalRV(values, cdf);
System.out.println(n1);
System.out.printf("%3s %15s %n", "n", "Values");
for (int i = 1; i <= 5; i++) {
    System.out.printf("%3d %15f %n", i+1, n1.getValue());
}
```

While the preferred method for generating random values from random variables is to create instance of the appropriate random variable class, the JSL also provide a set of functions for generating random values within the `JSLRandom` class. For all the previously listed random variables, there is a corresponding function that will generate a

random value. For example, the method `rNormal()` will generate a normally distributed value. Each method is named with an "r" in front of the distribution name. By using static import of `JSLRandom` the user can more conveniently call these methods. The following code example illustrates how to do this.

```
// use import static jsl.utilities.random.rvariable.JSLRandom.*;
// at the top of your java file
double v = rUniform(10.0, 15.0); // generate a U(10, 15) value
double x = rNormal(5.0, 2.0); // generate a Normal(mu=5.0, var= 2.0) value
double n = rPoisson(4.0); //generate from a Poisson(mu=4.0) value
System.out.printf("v = %f, x = %f, n = %f %n", v, x, n);
```

In addition to random values through these static methods, the `JSLRandom` class provides a set of methods for randomly selecting from arrays and lists and for creating permutations of arrays and lists. In addition, there is a set of methods for sampling from arrays and lists without replacement. The following code provide examples of using these methods.

```
// create a list
List<String> strings = Arrays.asList("A", "B", "C", "D");
// randomly pick from the list, with equal probability
for (int i=1; i<=5; i++){
    System.out.println(randomlySelect(strings));
}

// create an array to hold a population of values
double[] y = new double[10];
for (int i = 0; i < 10; i++) {
    y[i] = i + 1;
}

// create the population
DPopulation p = new DPopulation(y);
System.out.println(p);

// permute the population
p.permute();
System.out.println(p);

// directly permute the array using JSLRandom
System.out.println("Permuting y");
JSLRandom.permutation(y);
System.out.println(DPopulation.toString(y));

// sample from the population
```

```

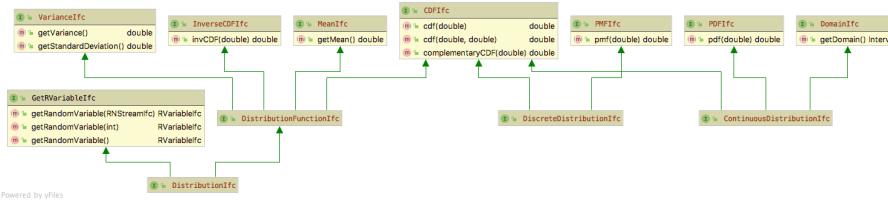
double[] x = p.sample(5);
System.out.println("Sampling 5 from the population");
System.out.println(DPopulation.toString());

// create a string list and permute it
List<String> strList = new ArrayList<>();
strList.add("a");
strList.add("b");
strList.add("c");
strList.add("d");
System.out.println(strList);
JSLRandom.permutation(strList);
System.out.println(strList);

```

## 0.16 Modeling Probability Distributions

The `jsl.utilities.random.rvariable` package is the key package for generating random variables; however, it does not facilitate performing calculations involving the underlying probability distributions. To perform calculations involving probability distributions, you should use the `jsl.utilities.distribution` package. This package has almost all the same distributions represented within the `jsl.utilities.random.rvariable` package.



**Figure 8:** Distribution Interfaces

Figure 8 illustrates the interfaces used to define probability distributions. First, the interface, `CDFIfc` serves as the basis for discrete distributions via the `DiscreteDistributionIfc` interface, for continuous distributions via the `ContinuousDistributionIfc` interface and the general `DistributionIfc` interface. The discrete distributions such as the geometric, binomial, etc. implement the `DiscreteDistributionIfc` and `PMFIfc` interfaces. Similarly, continuous distributions like the normal, uniform, etc. implement the `ContinuousDistributionIfc` and `PDFIfc` interfaces. All concrete implementations of distributions extend from the abstract base class `Distribution`, which implements the `DistributionIfc` interface. Thus, all distributions have the following capabilities:

- `cdf(double b)` - computes the cumulative probability,  $F(b) = P(X \leq b)$
- `cdf(double a, double b)` - computes the cumulative probability,  $P(a \leq X \leq b)$

- `complementaryCDF(double b)` - computes the cumulative probability,  $1 - F(b) = P(X > b)$
- `getMean()` - returns the expected value (mean) of the distribution
- `getVariance()` - returns the variance of the distribution
- `getStandardDeviation()` - returns the standard deviation of the distribution
- `invCDF(double p)` - returns the inverse of the cumulative distribution function  $F^{-1}(p)$ . This is performed by numerical search if necessary

Discrete distributions have a method called `pmf(double k)` that returns the probability associated with the value  $k$ . Continuous distributions have a probability density function,  $f(x)$ , implemented in the method, `pdf(double x)`. Finally, all distributions know how to create random variables through the `GetRVariableIfc` interface that provides the following methods.

- `RVariableIfc getRandomVariable(RNStreamIfc stream)` - returns a new instance of a random variable based on the current values of the distribution's parameters that uses the supplied stream
- `RVariableIfc getRandomVariable(int streamNum)` - returns a new instance of a random variable based on the current values of the distribution's parameters that uses the supplied stream number
- `RVariableIfc getRandomVariable()` - returns a new instance of a random variable based on the current values of the distribution's parameters that uses a newly created stream

As an example, the following code illustrates some calculations for the binomial distribution.

```
// make and use a Binomial(p, n) distribution
int n = 10;
double p = 0.8;
System.out.println("n = " + n);
System.out.println("p = " + p);
Binomial bnDF = new Binomial(p, n);
System.out.println("mean = " + bnDF.getMean());
System.out.println("variance = " + bnDF.getVariance());
// compute some values
System.out.printf("%3s %15s %15s %n", "k", "p(k)", "cdf(k)");
for (int i = 0; i <= 10; i++) {
    System.out.printf("%3d %15.10f %15.10f %n", i, bnDF.pmf(i), bnDF.cdf(i));
}
```

The output shows the mean, variance, and basic probability calculations.

```
n = 10
p = 0.8
mean = 8.0
variance = 1.5999999999999996
```

k	p(k)	cdf(k)
0	0.0000001024	0.0000001024
1	0.0000040960	0.0000041984
2	0.0000737280	0.0000779264
3	0.0007864320	0.0008643584
4	0.0055050240	0.0063693824
5	0.0264241152	0.0327934976
6	0.0880803840	0.1208738816
7	0.2013265920	0.3222004736
8	0.3019898880	0.6241903616
9	0.2684354560	0.8926258176
10	0.1073741824	1.0000000000

The `jsl.utilities.random.rvariable` package creates instances of random variables that are immutable. That is, once you create a random variable, its parameters cannot be changed. However, distributions permit their parameters to be changed and they also facilitate the creation of random variables. The following code uses the `setParameters()` method to change the parameters of the previously created binomial distribution and then creates a random variable based on the mutated distribution.

```
// change the probability and number of trials
bnDF.setParameters(0.5, 20);
System.out.println("mean = " + bnDF.getMean());
System.out.println("variance = " + bnDF.getVariance());
// make random variables based on the distributions
RVariateIfc brv = bnDF.getRandomVariable();
System.out.printf("%3s %15s %n", "n", "Values");
// generate some values
for (int i = 0; i < 5; i++) {
    // getValue() method returns generated values
    int x = (int)brv.getValue();
    System.out.printf("%3d %15d %n", i+1, x);
}
```

The results are as we would expect. Similar calculations can be made for continuous distributions. In most cases, the concrete implementations of the various distributions have specialize methods beyond those generic methods described here. Please refer to the java docs for further details.

n	Values
1	11
2	14
3	16
4	7

There are a number of useful static methods defined for the binomial, normal, gamma, and Student-T distributions. Specifically, for the binomial distribution, has the following static methods

- `binomialPMF(int j, int n, double p)` - directly computes the probability for the value  $j$
- `binomialCDF(int j, int n, double p)` - directly computes the cumulative distribution function for the value  $j$
- `binomialCCDF(int j, int n, double p)` - directly computes the complementary cumulative distribution function for the value of  $j$
- `binomialInvCDF(double x, int n, double p)` - directly computes the inverse cumulative distribution function

These methods are designed to perform their calculations in a numerically stable manner to ensure numerical accuracy. The normal distribution has the following static methods for computations involving the standard normal distribution.

- `stdNormalCDF(double z)` - the cumulative probability for a  $Z \sim N(0, 1)$  random variable, i.e.  $F(z) = P(Z \leq z)$
- `stdNormalComplementaryCDF(double z)` - returns  $1 - P(Z \leq z)$
- `stdNormalInvCDF(double p)` - returns  $z = F^{-1}(p)$  the inverse of the cumulative distribution function

The Student-T distribution also has two static convenience methods to facilitate computations.

- `getCDF(double dof, double x)` - computes the cumulative distribution function for  $x$  given the degrees of freedom
- `getInvCDF(double dof, double p)` - computes the inverse cumulative distribution function or t-value for the supplied probability given the degrees of freedom.

Within the gamma distribution there are some convenience methods for computing the gamma function, the natural logarithm of the gamma function, the incomplete gamma function, and the digamma function (derivative of the natural logarithm of the gamma function).

# Collecting Statistics

## LEARNING OBJECTIVES

- To be able to collect statistics using classes within the JSL
- To understand the basics of statistical computations supported by the JSL

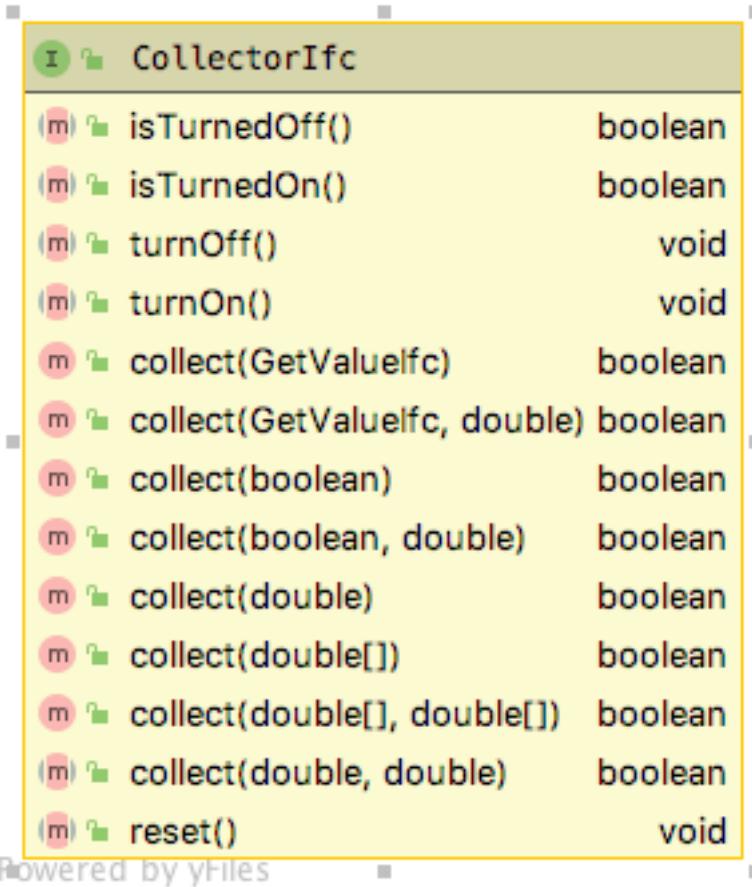
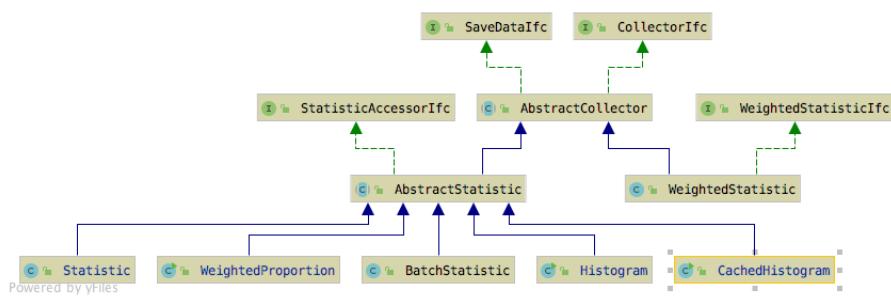
The JSL has a wide variety of classes that support statistic computations. A main theme in understanding the usage of the classes within the `jsl.utilities.statistics`<sup>19</sup> package is the concept of collection. This concept is encapsulated within the interface, `CollectorIfc` interface. The methods of the `CollectorIfc` interface are illustrated in Figure 9.

Something is a collector, if it implements the `CollectorIfc` interface. The implication is that those values presented to the various `collect` methods will be observed and tabulated into various quantities based on the presented values. The `collect` method has been overridden to facilitate collection of double values, arrays of double values, and boolean values. The two parameter `collect` methods permit collection of a second set of values. The second parameter is meant to represent the weight associated with the first parameter. As we will see, this facilitates the collection of weighted statistics. Collection can be turned on or off and can be reset. Turning collection off should cause the presented values to be ignored during the off period. Resetting a collector should set the state of the collector as if no values had been presented. Thus, resetting a collector should clear all previous collection results.

Figure 10 presents the major classes and interfaces within the statistics package. The `CollectorIfc` interface is implemented within the abstract base class `AbstractCollector`, which serves as the basis for various concrete implementations of statistical collectors. The `SaveDataIfc` interface defines methods for indicating whether or not the data collected should be saved into arrays. There are two major kinds of statistics one of which assumes that the values presented must be weighted, the `WeightedStatisticIfc` interface and the `WeightedStatistic` class. While the other branch of classes, derived from `AbstractStatistic` do not necessarily have to be weighted. The main classes to be discussed here are `Statistic` and `Histogram`.

---

<sup>19</sup><https://rossetti.git-pages.uark.edu/JSL-Documentation/jsl/utilities/statistic/package-summary.html>

**Figure 9:** CollectorIfc Interface**Figure 10:** Major Classes and Interfaces in the Statistics Package

## 0.17 Creating and Using a Statistic

The `statistic` class has a great deal of functionality. It accumulates summary statistics on the values presented to it via its `collect` methods. Recall also that since the `statistic` class implements the `CollectorIfc` interface, you can use the `reset()` method to clear all accumulated statistics and reuse the `statistic` instance. The major statistical quantities are tabulated in Figure ??.

StatisticAccessorIfc	
(m)  getName()	String
(m)  getCount()	double
(m)  getSum()	double
(m)  getWeightedSum()	double
(m)  getWeightedSumOfSquares()	double
(m)  getAverage()	double
(m)  getSumOfWeights()	double
(m)  getWeightedAverage()	double
(m)  getDeviationSumOfSquares()	double
(m)  getVariance()	double
(m)  getStandardDeviation()	double
(m)  getMin()	double
(m)  getMax()	double
(m)  getLastValue()	double
(m)  getLastWeight()	double
(m)  getKurtosis()	double
(m)  getSkewness()	double
(m)  getStandardError()	double
(m)  getHalfWidth()	double
(m)  getHalfWidth(double)	double
(m)  getConfidenceLevel()	double
(m)  getConfidenceInterval()	Interval
(m)  getRelativeError()	double
(m)  getRelativeWidth()	double
(m)  getRelativeWidth(double)	double
(m)  getConfidenceInterval(double)	Interval
(m)  getLag1Covariance()	double
(m)  getLag1Correlation()	double
(m)  getVonNeumannLag1TestStatistic()	double
(m)  getVonNeumannLag1TestStatisticPValue()	double
(m)  getNumberMissing()	double
(m)  getLeadingDigitRule(double)	int
(m)  toString()	String
(m)  getStatistics(double[])	void
(m)  getStatistics()	double[]

Powered By yFiles

As can be seen in Fig-

ure ??, the statistic class not only computes the standard statistical quantities such

as the count, average, and variance, it also has functionality to compute confidence intervals, skewness, kurtosis, the minimum, the maximum, and lag 1 covariance and correlation. The computed confidence intervals are based on the assumption that the observed data are normally distributed or that the sample size is large enough to justify using the central limit theorem to assume that the sampling distribution is normal. Thus, we can assume that the confidence intervals are approximate. The summary statistics are computed via efficient one pass algorithms that do not require any observed data to be stored. The algorithms are designed to minimize issues related to numerical precision within the calculated results. The `toString()` method of the `Statistic` class has been overridden to contain all of the computed values. Let's illustrate the usage of the `Statistic` class with some code. In this code, first we create a normal random variable to be able to generate some data. Then, two statistics are created. The first statistic directly collects the generated values. The second statistic is designed to collect  $P(X \geq 20.0)$  by observing whether or not the generated value meets this criteria as defined by the boolean expression  $x \geq 20.0$ .

```
// create a normal mean = 20.0, variance = 4.0 random variable
NormalRV n = new NormalRV(20.0, 4.0);
// create a Statistic to observe the values
Statistic stat = new Statistic("Normal Stats");
Statistic pGT20 = new Statistic("P(X>=20)");
// generate 100 values
for (int i = 1; i <= 100; i++) {
    // getValue() method returns generated values
    double x = n.getValue();
    stat.collect(x);
    pGT20.collect(x >= 20.0);
}
System.out.println(stat);
```

The results for the statistics collected directly on the observations from the `toString()` method are as follows.

```
ID 1
Name Normal Stats
Number 100.0
Average 20.370190128861807
Standard Deviation 2.111292233346322
Standard Error 0.2111292233346322
Half-width 0.4189261806189412
Confidence Level 0.95
Confidence Interval [19.951263948242865, 20.78911630948075]
Minimum 15.020744984423821
Maximum 25.33588436770212
Sum 2037.0190128861807
Variance 4.457554894588499
```

```

Weighted Average 20.370190128861797
Weighted Sum 2037.0190128861796
Sum of Weights 100.0
Weighted Sum of Squares 41935.76252316213
Deviation Sum of Squares 441.2979345642614
Last value collected 21.110736402119805
Last weighted collected 1.0
Kurtosis -0.534855387072145
Skewness 0.20030433873223502
Lag 1 Covariance -0.973414579833684
Lag 1 Correlation -0.22057990840016864
Von Neumann Lag 1 Test Statistic -2.2136062395518343
Number of missing observations 0.0
Lead-Digit Rule(1) -1

```

Of course, this is probably more output than what you need, but you can use the methods illustrated in Figure ?? to access specific desired quantities. Notice that in the code example that the  $P(X \geq 20.0)$  is also collected. This is done by using the boolean expression  $x >= 20.0$  within the `collect()` method. This expression evaluates to either true or false. The true values are presented as 1.0 and the false values as 0.0. Thus, this expression acts as an indicator variable and facilitates the estimation of probabilities. The results from the statistics can be pretty printed by using the `StatisticReporter` class, which takes a list of objects that implement the `StatisticAccessorIfc` interface and facilitates the writing and printing of various statistical summary reports.

```

StatisticReporter reporter = new StatisticReporter(List.of(stat, pGT20));
System.out.println(reporter.getHalfWidthSummaryReport());

```

Half-Width Statistical Summary Report - Confidence Level (95.000)%

Name	Count	Average	Half-Width
<hr/>			
<hr/>			
Normal Stats	100	20.3702	0.4189
$P(X \geq 20)$	100	0.5100	0.0997
<hr/>			
<hr/>			

The `Statistic` class has a number of very useful static methods that work on arrays and compute various statistical quantities.

- `int getIndexOfMin(double[] x)` - returns the index of the element that is smallest. If there are ties, the first found is returned.
- `double getMin(double[] x)` - returns the element that is smallest. If there are ties, the first found is returned.
- `int getIndexOfMax(double[] x)` - returns the index of the element that is largest. If there are ties, the first found is returned.

- `double getMax(double[] x)` - returns the element that is largest. If there are ties, the first found is returned.
- `double getMedian(double[] data)` - returns the value that has 50 percent of the data above and below it.
- `int countLessEqualTo(double[] data, double x)` - returns the count of the elements that are less than or equal to  $x$
- `int countLessThan(double[] data, double x)` - returns the count of the elements that are less than  $x$
- `int countGreaterEqualTo(double[] data, double x)` - returns the count of the elements that are greater than or equal to  $x$
- `int countGreaterThan(double[] data, double x)` - returns the count of the elements that are greater than  $x$
- `double[] getOrderStatistics(double[] data)` - returns a sorted copy of the supplied array ordered from smallest to largest
- `long estimateSampleSize(double desiredHW, double stdDev, double level)` - returns the approximate sample size necessary to reach the desired half-width at the specified confidence level given the estimate of the sample standard deviation.
- `Statistic collectStatistics(double[] x, double[] w)` - returns an instance of `Statistic` that summarizes the array of values and the supplied weights.
- `collectStatistics(double[] x)` - returns an instance of `Statistic` that summarizes the array of values

## 0.18 Histograms and Frequencies

A histogram tabulates counts and frequencies of observed data over a set of contiguous intervals. Let  $b_0, b_1, \dots, b_k$  be the breakpoints (end points) of the class intervals such that  $(b_0, b_1], (b_1, b_2], \dots, (b_{k-1}, b_k]$  form  $k$  disjoint and adjacent intervals. The intervals do not have to be of equal width. Also,  $b_0$  can be equal to  $-\infty$  resulting in interval  $(-\infty, b_1]$  and  $b_k$  can be equal to  $+\infty$  resulting in interval  $(b_{k-1}, +\infty)$ . Define  $\Delta b_j = b_j - b_{j-1}$  and if all the intervals have the same width (except perhaps for the end intervals),  $\Delta b = \Delta b_j$ . To count the number of observations that fall in each interval, we can use the count function:

$$c(\vec{x} \leq b) = \#\{x_i \leq b\} \quad i = 1, \dots, n$$

$c(\vec{x} \leq b)$  counts the number of observations less than or equal to  $x$ . Let  $c_j$  be the observed count of the  $x$  values contained in the  $j^{th}$  interval  $(b_{j-1}, b_j]$ . Then, we can determine  $c_j$  via the following equation:

$$c_j = c(\vec{x} \leq b_j) - c(\vec{x} \leq b_{j-1})$$

The key parameters of a histogram are:

- The first bin lower limit ( $b_0$ ): This is the starting point of the range over which the data will be tabulated.
- The number of bins ( $k$ )
- The width of the bins, ( $\Delta b$ )

Histogram	
m	makeHistogram(double, int)
m	makeHistogram(double, double, int)
m	makeHistogram(double, double, int, String)
m	makeHistogram(double, double, int, String, double[])
m	collect(double, double)
b	binIndex(double)
m	reset()
m	getBinNumber(double)
m	getUnderFlowCount()
m	getOverFlowCount()
m	getBin(double)
m	getBin(int)
m	getBins()
m	getBinCount(double)
m	getBinCount(int)
m	getBinFraction(int)
m	getBinFraction(double)
m	getCumulativeBinCount(double)
m	getCumulativeBinCount(int)
m	getCumulativeBinFraction(int)
m	getCumulativeBinFraction(double)
m	getCumulativeCount(int)
m	getCumulativeCount(double)
m	getCumulativeFraction(int)
m	getCumulativeFraction(double)
m	getTotalCount()
m	getFirstBinLowerLimit()
m	getLastBinUpperLimit()
m	toString()
m	getAverage()
m	getConfidenceLevel()
m	getCount()
m	getDeviationSumOfSquares()
m	getHalfWidth(double)
m	getKurtosis()
m	getLag1Correlation()
m	getLag1Covariance()
m	getLastValue()
m	getLastWeight()
m	getMax()
m	getMin()
m	getObsWeightedSum()
m	getSkewness()
m	getStandardDeviation()
m	getStandardError()
m	getSum()
m	getSumOfWeights()
m	getVariance()
m	getVonNeumannLag1TestStatistic()
m	getVonNeumannLag1TestStatisticPValue()
m	getWeightedAverage()
m	getWeightedSum()
m	getWeightedSumOfSquares()
m	getLeadingDigitRule(double)
m	main(String[])

Powered by Yhies

Figure ?? presents the methods

of the `Histogram` class. The `Histogram` class is utilized in a very similar manner as the `Statistic` class by collecting observations. The observations are then tabulated into the bins. The `Histogram` class allows the user to tabulate the bin contents via the `collect()` methods inherited from the `AbstractStatistic` base class. Since data may fall below the first bin and after the last bin, the implementation also provides counts for those occurrences. Since a `Histogram` is a sub-class of `AbstractStatistic`, it also implements the `statisticAccessorIfc` to provide summary statistics on the data tabulated within the bins. The `Histogram` class also provides static methods to create histograms based on a range (lower limit to upper limit) with a given number of bins. In this case, an appropriate bin width is computed.

In some cases, the client may not know in advance the appropriate settings for the number of bins or the width of the bins. In this situation, one can use the `CachedHistogram` class, which first collects the data in a temporary cache array. Once the cache has been filled up, the `CachedHistogram` computes a reasonable lower limit, number of bins, and bin width based on the statistics collected over the cache. The underlying histogram is available via the `getHistogram()` method after the cache has been used.

```
ExponentialRV d = new ExponentialRV(2);
// create a histogram with lower limit 0.0, 20 bins, of width 0.1
Histogram h = new Histogram(0.0, 20, 0.1);
for (int i = 1; i <= 100; ++i) {
    h.collect(d.getValue());
}
System.out.println(h);
```

```
Histogram: Histogram
-----
Number of bins = 20
Bin width = 0.1
First bin starts at = 0.0
Last bin ends at = 2.0
Under flow count = 0.0
Over flow count = 45.0
Total bin count = 55.0
Total count = 100.0
-----
Bin Range      Count Total Prob CumProb
1 [0.00,0.10)   2.0   2.0 0.036364 0.036364
2 [0.10,0.20)   5.0   7.0 0.090909 0.127273
3 [0.20,0.30)   5.0  12.0 0.090909 0.218182
4 [0.30,0.40)   2.0  14.0 0.036364 0.254545
5 [0.40,0.50)   7.0  21.0 0.127273 0.381818
6 [0.50,0.60)   3.0  24.0 0.054545 0.436364
7 [0.60,0.70)   3.0  27.0 0.054545 0.490909
8 [0.70,0.80)   3.0  30.0 0.054545 0.545455
```

```

 9 [0.80,0.90)    2.0  32.0 0.036364 0.581818
10 [0.90,1.00)    2.0  34.0 0.036364 0.618182
11 [1.00,1.10)    5.0  39.0 0.090909 0.709091
12 [1.10,1.20)    6.0  45.0 0.109091 0.818182
13 [1.20,1.30)    2.0  47.0 0.036364 0.854545
14 [1.30,1.40)    2.0  49.0 0.036364 0.890909
15 [1.40,1.50)    3.0  52.0 0.054545 0.945455
16 [1.50,1.60)    1.0  53.0 0.018182 0.963636
17 [1.60,1.70)    1.0  54.0 0.018182 0.981818
18 [1.70,1.80)    1.0  55.0 0.018182 1.000000
19 [1.80,1.90)    0.0  55.0 0.000000 1.000000
20 [1.90,2.00)    0.0  55.0 0.000000 1.000000

```

The JSL will also tabulate count frequencies when the values are only integers. This is accomplished with the `IntegerFrequency` class. Figure 11 indicates the methods of the `IntegerFrequency` class. The object can return information on the counts and proportions. It can even create a `DEmpiricalCDF` distribution based on the observed data.

In the following code example, an instance of the `IntegerFrequency` class is created. Then, an instance of a binomial random variable is used to generate a sample of 10,000 observations. The sample is then collected by the `IntegerFrequency` class's `collect()` method.

```

IntegerFrequency f = new IntegerFrequency("Frequency Demo");
BinomialRV bn = new BinomialRV(0.5, 100);
double[] sample = bn.sample(10000);
f.collect(sample);
System.out.println(f);

```

As can be noted in the output, only those integers that are actually observed are tabulated in terms of the count of the number of times the integer is observed and its proportion. The user does not have to specify the range of possible integers; however, instances of `IntegerFrequency` can be created that specify a lower and upper limit on the tabulated values. The overflow and underflow counts then tabulate when observations fall outside of the specified range.

```

Frequency Tabulation Frequency Demo
-----
Number of cells = 39
Lower limit = -2147483648
Upper limit = 2147483647
Under flow count = 0
Over flow count = 0
Total count = 10000
-----
Value      Count      Proportion
31        1        1.0E-4

```

C IntegerFrequency	
m	getName() String
m	setName(String) void
m	collect(int[]) void
m	collect(int) void
m	collect(double) void
m	collect(double[]) void
m	reset() void
m	getUnderFlowCount() int
m	getOverFlowCount() int
m	getValues() int[]
m	getFrequencies() int[]
m	getProportions() double[]
m	getCumulativeFrequency(int) int
m	getCumulativeProportion(int) double
m	getValueFrequencies() int[][]
m	getValueProportions() double[][]
m	getValueCumulativeProportions() double[][]
m	getNumberOfCells() int
m	getTotalCount() int
m	getFrequency(int) int
m	getProportion(int) double
m	getFrequencies(int[]) int[]
m	getCellList() List<Cell>
m	createDEmpiricalCDF() DEmpiricalCDF
m	getCells() SortedSet<Cell>
m	toString() String
m	getStatistic() Statistic

Powered by yFiles

Figure 11: IntegerFrequency Class

```
33   4   4.0E-4
34   5   5.0E-4
35   9   9.0E-4
36  17   0.0017
37  28   0.0028
38  41   0.0041
39  74   0.0074
40 100   0.01
41 192   0.0192
42 236   0.0236
43 277   0.0277
44 406   0.0406
45 453   0.0453
46 564   0.0564
47 653   0.0653
48 741   0.0741
49 762   0.0762
50 750   0.075
51 768   0.0768
52 783   0.0783
53 679   0.0679
54 600   0.06
55 484   0.0484
56 407   0.0407
57 324   0.0324
58 210   0.021
59 155   0.0155
60 108   0.0108
61  74   0.0074
62  41   0.0041
63  15   0.0015
64  15   0.0015
65  17   0.0017
66  3   3.0E-4
67  1   1.0E-4
69  1   1.0E-4
70  1   1.0E-4
71  1   1.0E-4
```

---

Finally, the JSL provides the ability to define labeled states and to tabulate frequencies and proportions related to the visitation and transition between the states. This functionality is available in the `StateFrequency` class. The following code example creates an instance of `StateFrequency` by providing the number of states. The states are returned in a `List` and then 10,000 states are randomly selected from the list with equal probability using the `JSLRandom` functionality to randomly select from lists. The randomly

selected state is then observed via the `collect()` method.

```
// number of states is 6
StateFrequency sf = new StateFrequency(6);
List<State> states = sf.getStates();
for(int i=1;i<=10000;i++){
    State state = JSLRandom.randomlySelect(states);
    sf.collect(state);
}
System.out.println(sf);
```

The output is what you would expect based on selecting the states with equal probability. Notice that the `StateFrequency` class not only tabulates the visits to the states, similar to `IntegerFrequency`, it also counts and tabulates the transitions between states. These detailed tabulations are available via the various methods of the class. See the Java docs for further details.

```
State Frequency Tabulation for: Identity#1
State Labels
State{id=1, number=0, name='State:0'}
State{id=2, number=1, name='State:1'}
State{id=3, number=2, name='State:2'}
State{id=4, number=3, name='State:3'}
State{id=5, number=4, name='State:4'}
State{id=6, number=5, name='State:5'}
State transition counts
[288, 272, 264, 282, 265, 286]
[283, 278, 283, 286, 296, 266]
[286, 298, 263, 264, 247, 282]
[271, 263, 275, 279, 280, 294]
[274, 305, 273, 281, 296, 268]
[254, 277, 282, 270, 313, 255]
State transition proportions
[0.17380808690404345, 0.16415208207604104, 0.15932407966203982, 0.17018708509354255, 0.15992757996378998, 0.172601086300543
[0.16725768321513002, 0.16430260047281323, 0.16725768321513002, 0.1690307328605201, 0.17494089834515367, 0.157210401891252
[0.174390243902439, 0.18170731707317073, 0.1603658536585366, 0.16097560975609757, 0.15060975609756097, 0.1719512195121951]
[0.16305655836341756, 0.1582430806257521, 0.1654632972322503, 0.16787003610108303, 0.1684717208182912, 0.17689530685920576
[0.1614614024749558, 0.17972893341190335, 0.16087212728344136, 0.16558632881555688, 0.17442545668827342, 0.157925751325869
[0.15384615384615385, 0.16777710478497881, 0.17080557238037553, 0.16353725015142337, 0.18958207147183526, 0.15445184736523

Frequency Tabulation Identity#1
-----
Number of cells = 6
Lower limit = 0
Upper limit = 5
Under flow count = 0
```

```

Over flow count = 0
Total count = 10000
-----
Value      Count   Proportion
0       1657    0.1657
1       1693    0.1693
2       1640    0.164
3       1662    0.1662
4       1697    0.1697
5       1651    0.1651
-----

```

## 0.19 Batch Statistics

In simulation, we often collect data that is correlated, that is not independent. This causes difficulty in developing valid confidence intervals for the estimators. Grouping the data into batches and computing the average of each batch is one methodology for reducing the dependence within the data. The idea is that the average associated with each batch will tend to be less dependent, especially the larger the batch size. The method of batch means provides a mechanism for developing an estimator for  $Var[\bar{X}]$ .

The method of batch means is based on observations  $(X_1, X_2, X_3, \dots, X_n)$ . The idea is to group the output into batches of size  $b$ , such that the averages of the data within a batch are more nearly independent and possibly normally distributed.

$$\underbrace{X_1, X_2, \dots, X_b}_{batch 1} \cdots \underbrace{X_{b+1}, X_{b+2}, \dots, X_{2b}}_{batch 2} \cdots \\ \underbrace{X_{(j-1)b+1}, X_{(j-1)b+2}, \dots, X_{jb}}_{batch j} \cdots \underbrace{X_{(k-1)b+1}, X_{(k-1)b+2}, \dots, X_{kb}}_{batch k}$$

Let  $k$  be the number of batches each of size  $b$ , where,  $b = \lfloor \frac{n}{k} \rfloor$ . Define the  $j^{th}$  batch mean (average) as:

$$\bar{X}_j(b) = \frac{1}{b} \sum_{i=1}^b X_{(j-1)b+i}$$

Each of the batch means are treated like observations in the batch means series. For example, if the batch means are re-labeled as  $Y_j = \bar{X}_j(b)$ , the batching process simply produces another series of data,  $(Y_1, Y_2, Y_3, \dots, Y_k)$  which may be more like a random sample. Why should they be more independent? Typically, in auto-correlated processes the lag- $k$  auto-correlations decay rapidly as  $k$  increases. Since, the batch means are formed from batches of size  $b$ , provided that  $b$  is large enough the data within a batch is conceptually far from the data in other batches. Thus, larger batch sizes are

good for ensuring independence; however, as the batch size increases the number of batches decreases and thus variance of the estimator will increase.

To form a  $(1 - \alpha)\%$  confidence interval, we simply treat this new series like a random sample and compute approximate confidence intervals using the sample average and sample variance of the batch means series:

$$\bar{Y}(k) = \frac{1}{k} \sum_{j=1}^k Y_j$$

The sample variance of the batch process is based on the  $k$  batches:

$$S_b^2(k) = \frac{1}{k-1} \sum_{j=1}^k (Y_j - \bar{Y}^2)$$

Finally, if the batch process can be considered independent and identically distributed the  $1 - \alpha$  level confidence interval can be written as follows:

$$\bar{Y}(k) \pm t_{\alpha/2, k-1} \frac{S_b(k)}{\sqrt{k}}$$

The `BatchStatistic` class within the `statistic` package implements a basic batching process. The `BatchStatistic` class works with data as it is presented to its `collect` method. Since we do not know in advance how much data we have, the `BatchStatistic` class has rules about the minimum number of batches and the size of batches that can be formed. Theory indicates that we do not need to have a large number of batches and that it is better to have a relatively small number of batches that are large in size.

Three attributes of the `BatchStatistic` class that are important are:

- `myMinNumBatches` – This represents the minimum number of batches required. The default value for this attribute is determined by `BatchStatistic.MIN_NUM_BATCHES`, which is set to 20.
- `myMinBatchSize` – This represents the minimum size for forming initial batches. The default value for this attribute is determined by `BatchStatistic.MIN_NUM_OBS_PER_BATCH`, which is set to 16.
- `myMaxNumBatchesMultiple` – This represents a multiple of minimum number of batches which is used to determine the upper limit (maximum) number of batches. For example, if `myMaxNumBatchesMultiple = 2` and the `myMinNumBatches = 20`, then the maximum number of batches we can have is 40 ( $2 * 20$ ). The default value for this attribute is determined by `BatchStatistic.MAX_BATCH_MULTIPLE`, which is set to 2.

The `BatchStatistic` class uses instances of the `Statistic` class to do its calculations. The bulk of the processing is done in two methods, `collect()` and `collectBatch()`. The `collect()` method simply uses an instance of the `Statistic` class (`myStatistic`) to collect statistics. When the amount of data collected (`myStatistic.getCount()`) equals the current batch size (`myCurrentBatchSize`) then the `collectBatch()` method is called to form a batch.

```

public final boolean collect(double value, double weight) {
    // other code
    myTotNumObs = myTotNumObs + 1.0;
    myValue = value;
    myWeight = weight;
    myStatistic.collect(myValue, myWeight);
    if (myStatistic.getCount() == myCurrentBatchSize) {
        b = collectBatch();
    }
}

```

Referring to the collectBatch() method in the following code, the batches that are formed are recorded in an array called bm[]. After recording the batch average, the statistic is reset for collecting the next batch of data. The number of batches is recorded and if this has reached the maximum number of batches (as determined by the batch multiple calculation), we rebatch the batches back down to the minimum number of batches by combining adjacent batches according to the batch multiple.

```

private boolean collectBatch() {
    boolean b = true;
    // increment the current number of batches
    myNumBatches = myNumBatches + 1;
    // record the average of the batch
    bm[myNumBatches] = myStatistic.getWeightedAverage();
    // collect running statistics on the batches
    b = myBMStatistic.collect(bm[myNumBatches]);
    // reset the within batch statistic for next batch
    myStatistic.reset();
    // if the number of batches has reached the maximum then rebatch down to
    // min number of batches
    if (myNumBatches == myMaxNumBatches) {
        myNumRebatches++;
        myCurrentBatchSize = myCurrentBatchSize * myMaxNumBatchesMultiple;
        int j = 0; // within batch counter
        int k = 0; // batch counter
        myBMStatistic.reset(); // clear for collection across new batches
        // loop through all the batches
        for (int i = 1; i <= myNumBatches; i++) {
            myStatistic.collect(bm[i]); // collect across batches old batches
            j++;
            if (j == myMaxNumBatchesMultiple) { // have enough for a batch
                //collect new batch average
                b = myBMStatistic.collect(myStatistic.getAverage());
                k++; //count the batches
                bm[k] = myStatistic.getAverage(); // save the new batch average
            }
        }
    }
}

```

```

        myStatistic.reset(); // reset for next batch
        j = 0;
    }
}

myNumBatches = k; // k should be minNumBatches
myStatistic.reset(); //reset for use with new data
}
return b;
}

```

There are a variety of procedures that have been developed that will automatically batch the data as it is collected. The JSL has a batching algorithm based on the procedure implemented within the Arena simulatin language. When a sufficient amount of data has been collected batches are formed. As more data is collected, additional batches are formed until  $k = 40$  batches are collected. When 40 batches are formed, the algorithm collapses the number of batches back to 20, by averaging each pair of batches. This has the net effect of doubling the batch size. This process is repeated as more data is collected, thereby ensuring that the number of batches is between 20 and 39. In addition, the procedure also computes the lag-1 correlation so that independence of the batches can be tested.

The `BatchStatistic` class also provides a public `rebatchToNumberOfBatches()` method to allow the user to rebatch the batches to a user supplied number of batches. Since the `BatchStatistic` class implements the `StatisticalAccessorIfc` interface, it can return the sample average, sample variance, minimum, maximum, etc. of the batches. Within the discrete-event modeling constructs of the JSL, batching can be turned on to collect batch statistics during a replication. The use of these constructs will be discussed when the discrete-event modeling elements of the JSL are presented.

The following code illustrates how to create and use a `BatchStatistic`.

```

ExponentialRV d = new ExponentialRV(2);
// number of observations
int n = 1000;
// minimum number of batches permitted
// there will not be less than this number of batches
int minNumBatches = 40;
// minimum batch size permitted
// the batch size can be no smaller than this amount
int minBatchSize = 25;
// maximum number of batch multiple
// The multiple of the minimum number of batches
// that determines the maximum number of batches
// e.g. if the min. number of batches is 20
// and the max number batches multiple is 2,

```

```
// then we can have at most 40 batches
int maxNBMultiple = 2;
// In this example, since 40*25 = 1000, the batch multiple does not matter
BatchStatistic bm = new BatchStatistic(minNumBatches, minBatchSize, maxNBMultiple);
for (int i = 1; i <= n; ++i) {
    bm.collect(d.getValue());
}
System.out.println(bm);
double[] bma = bm.getBatchMeanArrayCopy();
int i=0;
for(double x: bma){
    System.out.println("bm(" + i + ") = " + x);
    i++;
}
// this rebatches the 40 down to 10
Statistic s = bm.rebatchToNumberOfBatches(10);
System.out.println(s);
```

## 0.20 Summary

The `jsl.utilities.statistic` package defines a lot of functionality. Here is a summary of some of the useful classes and interfaces.

1. `CollectorIfc` defines a set of `collect()` methods for collecting data. The method is overridden to permit the collection of a wide variety of data type. The `collect()` method is designed to collect values and a weight associated with the value. This allows the collection of weighted statistics. `AbstractCollector` is an abstract base class for building concrete sub-classes.
2. `SaveDataIfc` defines methods for saving the observed data to arrays.
3. `WeightedStatisticIfc` defines statistics that are computed on weighted data values. `WeightedStatistic` is a concrete implementation of the interface.
4. `AbstractStatistic` is an abstract base class for defining statistics. Sub-classes of `AbstractStatistic` compute summary statistics of some kind.
5. `Histogram` defines a class to collect statistics and tabulate data into bins.
6. `Statistic` is a concrete implementation of `AbstractStatistic` allowing for a multitude of summary statistics.
7. `BatchStatistic` is also a concrete implementation of `AbstractStatistic` that provides for summarizing data via a batching process.
8. `IntegerFrequency` tabulates integer values into a frequencies by observed values, similar to a histogram.
9. `StateFrequency` facilitates defining labeled states and tabulating visitation and transition statistics.
10. `StatisticXY` collects statistics on  $(x, y)$  pairs computing statistics on the  $x$  and  $y$  values separately, as well as the covariance and correlation between the observa-

tions within a pair.

The most important class within the statistics package is probably the Statistic class. This class summarizes the observed data into summary statistics such as: minimum, maximum, average, variance, standard deviation, lag-1 correlation, and count. In addition, confidence intervals can be formed on the observations based on the student-t distribution. Finally, there are useful static methods for computing statistics on arrays and for estimating sample sizes. The reader is encourage to review the JSL documentation for all of the functionality, including the ability to write nicely printed statistical results.



# Monte Carlo Methods

## LEARNING OBJECTIVES

- To be able to use the JSL to perform basic Monte Carlo simulation
- To illustrate generating and collecting statistics for Monte Carlo simulation

This chapter illustrates how to use the JSL for simple Monte-Carlo simulations. The term Monte Carlo generally refers to the set of methods and techniques that estimate quantities by repeatedly sampling from models/equations represented in a computer. As such, this terminology is somewhat synonymous with computer simulation itself. The term Monte Carlo gets its origin from the Monte Carlo casino in the Principality of Monaco, where gambling and games of chance are well known. There is no one Monte Carlo method. Rather there is a collection of algorithms and techniques. In fact, the ideas of random number generation and random variate generation previously discussed form the foundation of Monte Carlo methods.

For the purposes of this chapter, we limit the term Monte Carlo methods to those techniques for generating and estimating the expected values of random variables, especially in regards to static simulation. In static simulation, the notion of time is relatively straightforward with respect to system dynamics. For a static simulation, time ‘ticks’ in a regular pattern and at each ‘tick’ the state of the system changes (new observations are produced).

## 0.21 Simple Monte Carlo Integration

In this example, we illustrate one of the fundamental uses of Monte Carlo methods: estimating the area of a function. Suppose we have some function,  $g(x)$ , defined over the range  $a \leq x \leq b$  and we want to evaluate the integral:

$$\theta = \int_a^b g(x)dx$$

Monte Carlo methods allow us to evaluate this integral by couching the problem as an estimation problem. It turns out that the problem can be translated into estimat-

ing the expected value of a well-chosen random variable. While a number of different choices for the random variable exist, we will pick one of the simplest for illustrative purposes. Define  $Y$  as follows with  $X \sim U(a, b)$ :

$$Y = (b - a) g(X) \quad (1)$$

Notice that  $Y$  is defined in terms of  $g(X)$ , which is also a random variable. Because a function of a random variable is also a random variable, this makes  $Y$  a random variable. Thus, the expectation of  $Y$  can be computed as follows:

$$E[Y] = (b - a) E[g(X)] \quad (2)$$

Now, let us derive  $E[g(X)]$ . By definition,

$$E_X[g(x)] = \int_a^b g(x) f_X(x) dx$$

And, the probability density function for a  $X \sim U(a, b)$  random variable is:

$$f_X(x) = \begin{cases} \frac{1}{b-a} & a \leq x \leq b \\ 0 & \text{otherwise} \end{cases}$$

Therefore,

$$E_X[g(x)] = \int_a^b g(x) f_X(x) dx = \int_a^b g(x) \frac{1}{b-a} dx \quad (3)$$

Substituting into Equation (2), yields,

$$\begin{aligned} E[Y] &= E[(b - a) g(X)] = (b - a) E[g(X)] \\ &= (b - a) \int_a^b g(x) \frac{1}{b-a} dx \\ &= \int_a^b g(x) dx = \theta \end{aligned}$$

Therefore, by estimating the expected value of  $Y$ , we can estimate the desired integral. From basic statistics, we know that a good estimator for  $E[Y]$  is the sample average of observations of  $Y$ . Let  $Y_1, Y_2, \dots, Y_n$  be a random sample of observations of  $Y$ . Let  $X_i$  be the  $i^{th}$  observation of  $X$ . Substituting each  $X_i$  into Equation (1) yields the  $i^{th}$  observation of  $Y$ ,

$$Y_i = (b - a) g(X_i)$$

Then, the sample average of is:

$$\begin{aligned}\bar{Y}(n) &= \frac{1}{n} \sum_{i=1}^n Y_i = (b - a) \frac{1}{n} \sum_{i=1}^n (b - a) g(X_i) \\ &= (b - a) \frac{1}{n} \sum_{i=1}^n g(X_i)\end{aligned}$$

where  $X_i \sim U(a, b)$ . Thus, by simply generating  $X_i \sim U(a, b)$ , plugging the  $X_i$  into the function of interest,  $g(x)$ , taking the average over the values and multiplying by  $(b - a)$ , we can estimate the integral. This works for any integral and it works for multi-dimensional integrals. While this discussion is based on a single valued function, the theory scales to multi-dimensional integration through the use of multi-variate distributions.

Suppose that we want to estimate the area under  $f(x) = x^{\frac{1}{2}}$  over the range from 1 to 4. That is, we want to evaluate the integral:

$$\theta = \int_1^4 x^{\frac{1}{2}} dx = \frac{14}{3} = 4.66\bar{6}$$

According to the previously presented theory, we need to generate  $X_i \sim U(1, 4)$  and then compute  $\bar{Y}$ , where  $Y_i = (4 - 1)\sqrt{X_i} = 3\sqrt{X_i}$ . In addition, for this simple example, we can easily check if our Monte Carlo approach is working because we know the true area.

```
double a = 1.0;
double b = 4.0;
UniformRV ucdf = new UniformRV(a, b);
Statistic stat = new Statistic("Area Estimator");
int n = 100; // sample size
for(int i=1;i<n;i++){
    double x = ucdf.getValue();
    double gx = Math.sqrt(x);
    double y = (b-a)*gx;
    stat.collect(y);
}
System.out.printf("True Area = %10.3f\n", 14.0/3.0);
System.out.printf("Area estimate = %10.3f\n", stat.getAverage());
System.out.println("Confidence Interval");
System.out.println(stat.getConfidenceInterval());
```

```

True Area =      4.667
Area estimate =   4.781
Confidence Interval
[4.608646560421988, 4.952515649272401]

```

## 0.22 Simulating the Game of Craps

Consider the game of “craps” as played in Las Vegas. The basic rules of the game are as follows: one player, the “shooter”, rolls a pair of dice. If the outcome of that roll is a 2, 3, or 12, the shooter immediately loses; if it is a 7 or an 11, the shooter wins. In all other cases, the number the shooter rolls on the first toss becomes the “point”, which the shooter must try to duplicate on subsequent rolls. If the shooter manages to roll the point before rolling a 7, the shooter wins; otherwise the shooter loses. It may take several rolls to determine whether the shooter wins or loses. After the first roll, only a 7 or the point have any significance until the win or loss is decided. Using the JSL random and statistic packages give answers and corresponding estimates to the following questions. Be sure to report your estimates in the form of confidence intervals.

- a) Before the first roll of the dice, what is the probability the shooter will ultimately win?
- b) What is the expected number of rolls required to decide the win or loss?

The solution to this problem involves the use of the Statistic class to collect the probability that the shooter will win and for the expected number of rolls. The following code illustrates the approach. The `DUniformRV` class is used to represent the two dice. Two statistics are created to estimate the probability of winning and to estimate the expected number of rolls required to decide a win or a loss. A for loop is used to simulate 5000 games. For each game, the logic of winning, losing or matching the first point. When the game is ended the instances of `Statistic` are used to collect the outcomes.

```

DUniformRV d1 = new DUniformRV(1, 6);
DUniformRV d2 = new DUniformRV(1, 6);
Statistic probOfWinning = new Statistic("Prob of winning");
Statistic numTosses = new Statistic("Number of Toss Statistics");
int numGames = 5000;
for (int k = 1; k <= numGames; k++) {
    boolean winner = false;
    int point = (int) d1.getValue() + (int) d2.getValue();
    int numberoftoss = 1;

    if (point == 7 || point == 11) {
        // automatic winner
        winner = true;
    }
    else {
        while (point != d1.getValue() + d2.getValue()) {
            numberoftoss++;
            d1.setValue((int) (Math.random() * 6) + 1);
            d2.setValue((int) (Math.random() * 6) + 1);
        }
        if (numberoftoss > 11) {
            winner = false;
        }
    }
    probOfWinning.addValue(winner);
    numTosses.addValue(numberoftoss);
}

```

```

} else if (point == 2 || point == 3 || point == 12) {
    // automatic loser
    winner = false;
} else { // now must roll to get point
    boolean continueRolling = true;
    while (continueRolling == true) {
        // increment number of tosses
        numberoftoss++;
        // make next roll
        int nextRoll = (int) d1.getValue() + (int) d2.getValue();
        if (nextRoll == point) {
            // hit the point, stop rolling
            winner = true;
            continueRolling = false;
        } else if (nextRoll == 7) {
            // crapped out, stop rolling
            winner = false;
            continueRolling = false;
        }
    }
}
probOfWinning.collect(winner);
numTosses.collect(numberoftoss);
}
StatisticReporter reporter = new StatisticReporter();
reporter.addStatistic(probOfWinning);
reporter.addStatistic(numTosses);
System.out.println(reporter.getHalfWidthSummaryReport());

```

As we can see from the results of the statistics reporter, the probability of winning is just a little less than 50 percent.

Half-Width Statistical Summary Report - Confidence Level (95.000)%

Name	Count	Average	Half-Width
<hr/>			
<hr/>			
Prob of winning	5000	0.4960	0.0139
Number of Toss Statistics	5000	3.4342	0.0856
<hr/>			
<hr/>			



# Introduction to Discrete Event Modeling

## LEARNING OBJECTIVES

- To be able to recognize and define the characteristics of a discrete-event dynamic system (DEDS)
- To be able to explain how time evolves in a DEDS
- To be able to develop and read an activity flow diagram
- To be able to create, run, and examine the results of a JSL model of a simple DEDS

## 0.23 Introduction

In Chapter 0.20, we explored how to develop models in the JSL for which time is not a significant factor. In the case of the news vendor problem, where we simulated each day's demand, time advanced at regular intervals. In the case of the area estimation problem, time was not a factor in the simulation. These types of simulations are often termed static simulations. In the next section, we begin our exploration of simulation where time is an integral component in driving the behavior of the system. In addition, we will see that time will not necessarily advance at regular intervals (e.g. hour 1, hour 2, etc.). This will be the focus of the rest of the textbook.

In this chapter, we explore the JSL simulation software platform for developing and executing simulation models. We will begin our study of the major emphasis of this textbook: modeling discrete-event dynamic systems. Recall that a discrete-event dynamic system (DEDS) is a system that evolves dynamically through time. This chapter will introduce how time evolves for DEDSs and illustrate how the JSL can be used to develop models for simple discrete-event system.

We can think of a system as a set of inter-related objects that work together to accomplish a common objective. The objects within a system are the concepts, abstractions,

things, and processes with definable boundaries and unique identity given our modeling perspective. Within the traditional simulation parlance, objects of particular interest within a system have often been called entities; however, this text takes a broader view to include other modeling elements of interest within the system (e.g. resources).

A discrete event system is a system in which the state of the system changes only at discrete points in time. There are essentially two fundamental viewpoints for modeling a discrete event system within simulation: the event view and the process view. These views are simply different representations for the same system. In the event view, the system and its elements are conceptualized as reacting to events. In the process view, the movement of entities through their processes implies the events within the system in the system. This chapter focuses on the event view.

The objects within the system change state because events occur within the system. In addition, the changes in state for an object may cause additional events to occur, affecting other objects within the system. Through the propagation of events and object state changes, the entire system state evolves through time. The state of the system is essentially the collection of states associated with all the objects in the system. The notion of modeling what happens to a system at particular events in time is the basis of discrete event modeling.

## 0.24 Discrete-Event Dynamic Systems

Nelson (1995) states that the event-view “defines system events by describing what happens to the system as it encounters entities”. In addition, Nelson (1995) states that the process view “implies system events by describing what happens to an entity as it encounters the system”. One can consider the event-view as taking the perspective of the system and the process-view as taking the perspective of the entity. The process-view describes the life cycles of objects within the system. Because of the way in which the process-view has been implemented through a network transaction flow approach within many simulation packages, the term entity has taken on the connotation of objects that flow or move within the system. To avoid confusion, the more general term, object, will be used to encompass non-flowing objects as well as flowing objects within the system.

To understand the event-view, the concepts of event, activity, state, and process must be clearly defined. Rumbaugh et al. (1991) defines state and its relationship to events and activities as “an abstraction of the attribute values and links of an object. Sets of values are grouped together into a state according to properties that affect the gross behavior of the object. A state corresponds to the interval between two events. A state has duration; it occupies an interval of time. A state is often associated with an activity that takes time to complete or with the value of an object satisfying some condition. In defining states, we ignore those attributes that do not affect the behavior of the object and we lump together in a single state all combinations of attribute values and links that have the same response to events.” Thus, the state of an object should entail only those aspects that are required for the modeling.

An event is something that happens at an instant in time that corresponds to a change in object state. An event is a one-way transmission of information from one object to another that causes an action resulting in a change in state. An event is said to be a scheduled event (or timed, determined, bounded) if its occurrence can be expressed as a function of system time and can thus be scheduled in time. For example, suppose you have a doctor's appointment at 11 am. The beginning of the appointment represents a scheduled event. An event is said to be a conditional event if it is dependent upon the outcome of certain conditions that cannot be predicted with certainty in advance (e.g. the availability of a server). Conditional events are sometimes called unbounded or contingent. If a scheduled event ends an activity, then the activity is said to be a scheduled or timed activity; otherwise, it is said to be a conditional activity.

Associated with an event is an action that is an instantaneous operation, i.e. it takes zero simulated time to complete. In contrast, an activity is an operation that takes simulated time to complete. An activity can be associated with the state of an object over an interval. Activities are defined by the occurrence of two events that represent the activity's beginning time and ending time and mark the entrance and exit of the state associated with the activity. In the simulation literature, it is common to refer to activities that take a specified duration of time as simply activities. To be more specific, it is useful to classify activities of a specified duration as timed activities. Getting your haircut is an example of a timed activity. There is a clear beginning and a clear ending for the activity. An activity that encompasses an interval of time that is unspecified or of indefinite length is called a conditional activity. An example of a conditional activity is a customer waiting in a queue for the barber to become available. The length of the activity depends on when the barber becomes available.

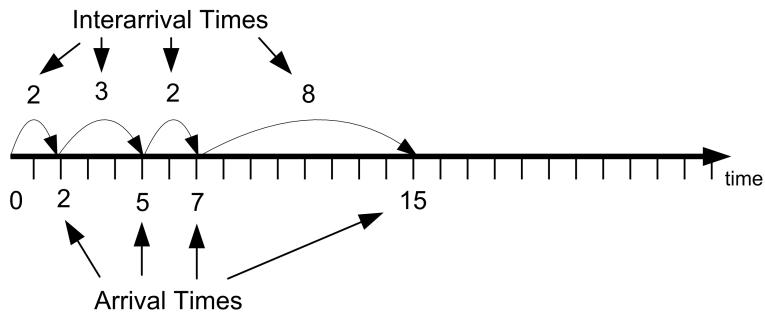
In the next section, we begin our exploration of simulation where time is an integral component in driving the behavior of the system. In addition, we will see that time will not necessarily advance at regular intervals (e.g. hour 1, hour 2, etc.). As discussed, an event occurs at a specific point in time, thus we need to understand how the simulation time clock works.

## 0.25 How the Discrete-Event Clock Works

This section introduces the concept of how time evolves in a discrete event simulation. This topic will also be revisited in future chapters after you have developed a deeper understanding for many of the underlying elements of simulation modeling.

In discrete-event dynamic systems, an event is something that happens at an instant in time which corresponds to a change in system state. An event can be conceptualized as a transmission of information that causes an action resulting in a change in system state. Let's consider a simple bank which has two tellers that serve customers from a single waiting line. In this situation, the system of interest is the bank tellers (whether they are idle or busy) and any customers waiting in the line. Assume that the bank opens at 9 am, which can be used to designate time zero for the simulation. It should be clear that if the bank does not have any customers arrive during the day that the

state of the bank will not change. In addition, when a customer arrives, the number of customers in the bank will increase by one. In other words, an arrival of a customer will change the state of the bank. Thus, the arrival of a customer can be considered an event.



**Figure 12:** Customer Arrival Process

**Table 2:** Customer Time of Arrivals

Customer	Time of arrival	Time between arrival
1	2	2
2	5	3
3	7	2
4	15	8

Figure 12 illustrates a time line of customer arrival events. The values for the arrival times in Table 2 have been rounded to whole numbers, but in general the arrival times can be any real valued numbers greater than zero. According to the figure, the first customer arrives at time two and there is an elapse of three minutes before customer 2 arrives at time five. From the discrete-event perspective, *nothing* is happening in the bank from time [0, 2); however, at time 2, an arrival event occurs and the subsequent actions associated with this event need to be accounted for with respect to the state of the system. An event denotes an instance of time that changes the state of the system. Since an event causes actions that result in a change of system state, it is natural to ask: What are the actions that occur when a customer arrives to the bank?

- The customer enters the waiting line.
- If there is an available teller, the customer will immediately exit the line and the available teller will begin to provide service.

- If there are no tellers available, the customer will remain waiting in the line until a teller becomes available.

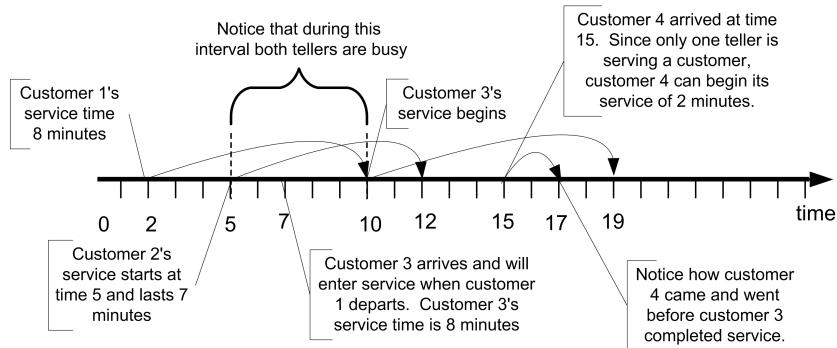
Now, consider the arrival of the first customer. Since the bank opens at 9 am with no customer and all the tellers idle, the first customer will enter and immediately exit the queue at time 9:02 am (or time 2) and then begin service. After the customer completes service, the customer will exit the bank. When a customer completes service (and departs the bank) the number of customers in the bank will decrease by one. It should be clear that some actions need to occur when a customer completes service. These actions correspond to the second event associated with this system: the customer service completion event. What are the actions that occur at this event?

- The customer departs the bank.
- If there are waiting customers, the teller indicates to the next customer that he/she will serve the customer. The customer will exit the waiting line and will begin service with the teller.
- If there are no waiting customers, then the teller will become idle.

Figure 13 contains the service times for each of the four customers that arrived in Figure 12. Examine the figure with respect to customer 1. Based on the figure, customer 1 can enter service at time two because there were no other customers present in the bank. Suppose that it is now 9:02 am (time 2) and that the service time of customer 1 is known *in advance* to be 8 minutes as indicated in Table 3. From this information, the time that customer 1 is going to complete service can be pre-computed. From the information in the figure, customer 1 will complete service at time 10 (current time + service time =  $2 + 8 = 10$ ). Thus, it should be apparent, that for you to recreate the bank's behavior over a time period that you must have knowledge of the customer's service times. The service time of each customer coupled with the knowledge of when the customer began service allows the time that the customer will complete service and depart the bank to be computed in advance. A careful inspection of Figure 13 and knowledge of how banks operate should indicate that a customer will begin service either immediately upon arrival (when a teller is available) or coinciding with when another customer departs the bank after being served. This latter situation occurs when the queue is not empty after a customer completes service. The times that service completions occur and the times that arrivals occur constitute the pertinent events for simulating this banking system.

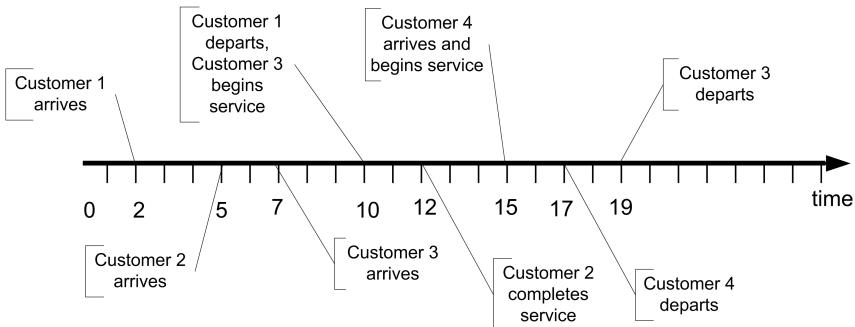
**Table 3:** Service Time for First Four Customers

Customer	Service Time Started	Service Time	Service Time Completed
1	2	8	10
2	5	7	12
3	10	9	19
4	15	2	17



**Figure 13:** Customer Service Process

If the arrival and the service completion events are combined, then the time ordered sequence of events for the system can be determined. Figure 14 illustrates the events ordered by time from smallest event time to largest event time. Suppose you are standing at time two. Looking forward, the next event to occur will be at time 5 when the second customer arrives. When you simulate a system, you must be able to generate a sequence of events so that, at the occurrence of each event, the appropriate actions are invoked that change the state of the system.



**Figure 14:** Events Ordered by Time Process

Time	Event	Comment
0	Bank Opens	
2	Arrival	Customer 1 arrives, enters service for 8 minutes, one teller becomes busy
5	Arrival	Customer 2 arrives, enters service for 7 minutes, the second teller becomes busy
7	Arrival	Customer 3 arrives, waits in queue

Time	Event	Comment
10	Service Completion	Customer 1 completes service, customer 3 exits the queue and enters service for 9 minutes
12	Service Completion	Customer 2 completes service, no customers are in the queue so a teller becomes idle
15	Arrival	Customer 4 arrives, enters service for 2 minutes, one teller becomes busy
17	Service Completion	Customer 4 completes service, a teller becomes idle
19	Service Completion	Customer 3 completes service

The real system will simply evolve over time; however, a simulation of the system must recreate the events. In simulation, events are created by adding additional logic to the normal state changing actions. This additional logic is responsible for scheduling future events that are implied by the actions of the current event. For example, when a customer arrives, the time to the next arrival can be generated and scheduled to occur at some future time. This can be done by generating the time until the next arrival and adding it to the current time to determine the actual arrival time. Thus, all the arrival times do not need to be pre-computed prior to the start of the simulation. For example, at time two, customer 1 arrived. Customer 2 arrives at time 5. Thus, the time between arrivals (3 minutes) is added to the current time and customer 2's arrival at time 5 is scheduled to occur. Every time an arrival occurs this additional logic is invoked to ensure that more arrivals will continue within the simulation.

Adding additional scheduling logic also occurs for service completion events. For example, since customer 1 immediately enters service at time 2, the service completion of customer 1 can be scheduled to occur at time 12 (current time + service time = 2 + 10 = 12). Notice that other events may have already been scheduled for times less than time 12. In addition, other events may be inserted before the service completion at time 12 actually occurs. From this, you should begin to get a feeling for how a computer program can implement some of these ideas.

Based on this intuitive notion for how a computer simulation may execute, you should realize that computer logic processing need only occur at the event times. That is, the state of the system does not change between events. Thus, it is *not necessary* to step incrementally through time checking to see if something happens at time 0.1, 0.2, 0.3, etc. (or whatever time scale you envision that is fine enough to not miss any events). Thus, in a discrete-event dynamic system simulation the clock does not “tick” at regular intervals. Instead, the simulation clock jumps from event time to event time. As the simulation moves from the current event to the next event, the current simulation time is updated to the time of the next event and any changes to the system associated with the next event are executed. This allows the simulation to evolve over time.

## 0.26 Simulating a Queueing System By Hand

This section builds on the concepts discussed in the previous section in order to provide insights into how discrete event simulations operate. In order to do this, we will simulate a simple queueing system by hand. That is, we will process each of the events that occur as well as the state changes in order to trace the operation of the system through time.

Consider again the simple banking system described in the previous section. To simplify the situation, we assume that there is only one teller that is available to provide service to the arriving customers. Customers that arrive while the teller is already helping a customer form a single waiting line, which is handled in a first come, first served manner. Let's assume that the bank opens at 9 am with no customers present and the teller idle. The time of arrival of the first eight customers is provided in the following table.

Customer Number	Time of Arrival	Service Time
1	3	4
2	11	4
3	13	4
4	14	3
5	17	2
6	19	4
7	21	3
8	27	2

We are going to process these customers in order to recreate the behavior of this system over from time 0 to 31 minutes. To do this, we need to be able to perform the “book-keeping” that a computer simulation model would normally perform for us. Thus, we will need to define some variables associated with the system and some attributes associated with the customers. Consider the following system variables.

### System Variables

- Let  $t$  represent the current simulation clock time.
- Let  $N(t)$  represent the number of customers in the system (bank) at any time  $t$ .
- Let  $Q(t)$  represent the number of customers waiting in line for the teller at any time  $t$ .
- Let  $B(t)$  represent whether or not the teller is busy (1) or idle (0) at any time  $t$ .

Because we know the number of tellers available, we know that the following relationship holds between the variables:

$$N(t) = Q(t) + B(t)$$

Note also that, knowledge of the value of  $N(t)$  is sufficient to determine the values of  $Q(t)$  and  $B(t)$  at any time  $t$ . For example, if we know that there are 3 customers in the bank, then  $N(t) = 3$ , and since the teller must be serving 1 of those customers,  $B(t) = 1$  and there must be 2 customers waiting,  $Q(t) = 2$ . In this situation, since  $N(t)$ , is sufficient to determine all the other system variables, we can say that  $N(t)$  is the system's *state variable*. The state variable(s) are the minimal set of system variables that are necessary to represent the system's behavior over time. We will keep track of the values of all of the system variables during our processing in order to collect statistics across time.

Attributes are properties of things that move through the system. In the parlance of simulation, we call the things that move through the system entity instances or entities. In this situation, the entities are the customers, and we can define a number of attributes for each customer. Here, customer is a type of entity or entity type. If we have other things that flow through the system, we identify the different types (or entity types). The attributes of the customer entity type are as follows. Notice that each attribute is subscripted by  $i$ , which indicates the  $i^{\text{th}}$  customer instance.

#### Entity Attributes

- Let  $ID_i$  be the identity number of the customer.  $ID_i$  is a unique number assigned to each customer that identifies the customer from other customers in the system.
- Let  $A_i$  be the arrival time of the  $i^{\text{th}}$  customer.
- Let  $S_i$  be the time the  $i^{\text{th}}$  customer started service.
- Let  $D_i$  be the departure time of the  $i^{\text{th}}$  customer.
- Let  $ST_i$  be the service time of the  $i^{\text{th}}$  customer.
- Let  $T_i$  be the total time spent in the system of the  $i^{\text{th}}$  customer.
- Let  $W_i$  be the time spent waiting in the queue for the  $i^{\text{th}}$  customer.

Clearly, there are also relationships between these attributes. We can compute the total time in the system for each customer from their arrival and departure times:

$$T_i = D_i - A_i$$

In addition, we can compute the time spent waiting in the queue for each customer as:

$$W_i = T_i - ST_i = S_i - A_i$$

As in the previous section, there are two types of events: arrivals and departures. Let  $E(t)$  be (A) for an arrival event at time  $t$  and be (D) for a departure event at time  $t$ . As we process each event, we will keep track of when the event occurs and the type of event. We will also keep track of the state of the system after each event has been processed. To make this easier, we will keep track of the system variables and the entity attributes within a table as follows.

**Table 6:** Hand Bank Simulation Bookkeeping Table.

System Variables				Entity Attributes								
t	E(t)	N(t)	B(t)	Q(t)	ID(i)	A(i)	S(i)	ST(i)	D(i)	T(i)	W(i)	Pending E(t)
0	NA	0	0	0	NA	NA	NA	NA	NA	NA	NA	NA

**Table 7:** Hand Bank Simulation Bookkeeping Table.

System Variables				Entity Attributes								
t	E(t)	N(t)	B(t)	Q(t)	ID(i)	A(i)	S(i)	ST(i)	D(i)	T(i)	W(i)	Pending E(t)
0	NA	0	0	0	NA	NA	NA	NA	NA	NA	NA	NA
3	A	1	1	0	1	3	4	3	NA	NA	O	E(7) = D(1), E(11) = A(2)

In the table, we see that the initial state of the system is empty and idle. Since there are no customers in the bank, there are no tabulated attribute values within the table. Reviewing the provided information, we see that customer 1 arrives at  $t = 3$  and has a service time of 4 minutes. Thus,  $ID_1 = 1$ ,  $A_1 = 3$ , and  $ST_1 = 4$ . We can enter this information directly into our bookkeeping table. In addition, because the bank was empty and idle when the customer arrived, we can note that the time that customer 1 starts service is the same as the time of their arrival and that they did not spend any time waiting in the queue. Thus,  $S_1 = 3$  and  $W_1 = 0$ . The table has been updated as follows.

Because customer 1 arrived to an empty system, they immediately started service at time 3. Since we know the service time of the customer,  $ST_1 = 4$ , and the current time,  $t = 3$ , we can determine that customer 1, will depart from the system at time 7 ( $t = 3 + 4 = 7$ ). That means we will have a departure event for customer 1 at time 7. According to the provided data, the next customer, customer 2, will arrive at time 11. Thus, we have two pending events, a departure of customer 1 at time 7 and the arrival of customer 2 at time 11. This fact is noted in the column labeled “Pending E(t)” for pending events. Here  $E(7) = D(1)$ ,  $E(11) = A(2)$  indicates that customer 1 with depart,  $D_1$ , at time 7,  $E(7)$  and customer 2 will arrive,  $A_2$ , at the event at time 11,  $E(11)$ . Clearly, the next event time will be the minimum of 7 and 11, which will be the departure of the first customer. Thus, our bookkeeping table can be updated as follows.

Since there are no customers in the bank and only the one pending event, the next event will be the arrival of customer 2 at time 11. The table can be updated as follows.

Since the pending event set is  $E(13) = A(3)$ ,  $E(15) = D(2)$  the next event will be the arrival of the third customer at time 13 before the departure of the second customer at time 15. We will now have a queue form. Updating our bookkeeping table, yields:

Notice that in the last table update, we did not update  $S_i$  and  $W_i$  because customer 3 had to wait in queue and did not start service. Customer 3 will start service, when customer 2 departs. Reviewing the pending event set, we see that the next event will

**Table 8:** Hand Bank Simulation Bookkeeping Table.

System Variables					Entity Attributes							
t	E(t)	N(t)	B(t)	Q(t)	ID(i)	A(i)	S(i)	ST(i)	D(i)	T(i)	W(i)	Pending E(t)
0	NA	0	0	0	NA	NA	NA	NA	NA	NA	NA	NA
3	A	1	1	0	1	3	4	3	NA	NA	0	E(7) = D(1), E(11) = A(2)
7	D	0	0	0	1	3	4	3	7	4	0	E(11) = A(2)

**Table 9:** Hand Bank Simulation Bookkeeping Table.

System Variables					Entity Attributes							
t	E(t)	N(t)	B(t)	Q(t)	ID(i)	A(i)	S(i)	ST(i)	D(i)	T(i)	W(i)	Pending E(t)
0	NA	0	0	0	NA	NA	NA	NA	NA	NA	NA	NA
3	A	1	1	0	1	3	4	3	NA	NA	0	E(7) = D(1), E(11) = A(2)
7	D	0	0	0	1	3	4	3	7	4	0	E(11) = A(2)
11	A	1	1	0	2	11	4	11	NA	NA	0	E(13) = A(3), E(15) = D(2)

be the arrival of customer 4 at time 14, which yields the following bookkeeping table.

Notice that we now have 3 customers in the system, 1 in service and 2 waiting in the queue. Reviewing the pending event set, we see that customer 2 will finally complete service and depart at time 15.

Because customer 2 completes service at time 15 and customer 3 is waiting in the line, we see that customer 3's attributes for  $S_i$  and  $W_i$  were updated within the table. Since customer 3 has started service and we know their service time of 4 minutes, we know that they will depart at time 19. The pending events set has been updated accordingly and indicates that the next event will be the arrival of customer 5 at time 17.

Now, we have a very interesting situation with the pending event set. We have two events that are scheduled to occur at the same time, the departure of customer 3 at time 19 and the arrival of customer 6 at time 19. In general, the order in which events

**Table 10:** Hand Bank Simulation Bookkeeping Table.

System Variables					Entity Attributes							
t	E(t)	N(t)	B(t)	Q(t)	ID(i)	A(i)	S(i)	ST(i)	D(i)	T(i)	W(i)	Pending E(t)
0	NA	0	0	0	NA	NA	NA	NA	NA	NA	NA	NA
3	A	1	1	0	1	3	4	3	NA	NA	0	E(7) = D(1), E(11) = A(2)
7	D	0	0	0	1	3	4	3	7	4	0	E(11) = A(2)
11	A	1	1	0	2	11	4	11	NA	NA	0	E(13) = A(3), E(15) = D(2)
13	A	2	1	1	3	13	4	15	NA	NA	2	E(14) = A(4), E(15) = D(2)

**Table 11:** Hand Bank Simulation Bookkeeping Table.

System Variables					Entity Attributes							
t	E(t)	N(t)	B(t)	Q(t)	ID(i)	A(i)	S(i)	ST(i)	D(i)	T(i)	W(i)	Pending E(t)
0	NA	0	0	0	NA	NA	NA	NA	NA	NA	NA	NA
3	A	1	1	0	1	3	4	3	NA	NA	0	E(7) = D(1), E(11) = A(2)
7	D	0	0	0	1	3	4	3	7	4	0	E(11) = A(2)
11	A	1	1	0	2	11	4	11	NA	NA	0	E(13) = A(3), E(15) = D(2)
13	A	2	1	1	3	13	4	15	NA	NA	2	E(14) = A(4), E(15) = D(2)
14	A	3	1	2	4	14	3	19	NA	NA	5	E(15) = D(2), E(17) = A(5)

**Table 12:** Hand Bank Simulation Bookkeeping Table.

System Variables					Entity Attributes							
t	E(t)	N(t)	B(t)	Q(t)	ID(i)	A(i)	S(i)	ST(i)	D(i)	T(i)	W(i)	Pending E(t)
0	NA	0	0	0	NA	NA	NA	NA	NA	NA	NA	NA
3	A	1	1	0	1	3	4	3	NA	NA	0	E(7) = D(1), E(11) = A(2)
7	D	0	0	0	1	3	4	3	7	4	0	E(11) = A(2)
11	A	1	1	0	2	11	4	11	NA	NA	0	E(13) = A(3), E(15) = D(2)
13	A	2	1	1	3	13	4	15	NA	NA	2	E(14) = A(4), E(15) = D(2)
14	A	3	1	2	4	14	3	19	NA	NA	5	E(15) = D(2), E(17) = A(5)
15	D	2	1	1	2	11	4	11	15	4	0	E(17) = A(5), E(19) = D(3)

**Table 13:** Hand Bank Simulation Bookkeeping Table.

System Variables					Entity Attributes							
t	E(t)	N(t)	B(t)	Q(t)	ID(i)	A(i)	S(i)	ST(i)	D(i)	T(i)	W(i)	Pending E(t)
0	NA	0	0	0	NA	NA	NA	NA	NA	NA	NA	NA
3	A	1	1	0	1	3	4	3	NA	NA	0	E(7) = D(1), E(11) = A(2)
7	D	0	0	0	1	3	4	3	7	4	0	E(11) = A(2)
11	A	1	1	0	2	11	4	11	NA	NA	0	E(13) = A(3), E(15) = D(2)
13	A	2	1	1	3	13	4	15	NA	NA	2	E(14) = A(4), E(15) = D(2)
14	A	3	1	2	4	14	3	19	NA	NA	5	E(15) = D(2), E(17) = A(5)
15	D	2	1	1	2	11	4	11	15	4	0	E(17) = A(5), E(19) = D(3)
17	A	3	1	2	5	17	2	22	NA	NA	5	E(19) = D(3), E(19) = A(6)

**Table 14:** Hand Bank Simulation Bookkeeping Table.

System Variables				Entity Attributes								
t	E(t)	N(t)	B(t)	Q(t)	ID(i)	A(i)	S(i)	ST(i)	D(i)	T(i)	W(i)	Pending E(t)
0	NA	0	0	0	NA	NA	NA	NA	NA	NA	NA	NA
3	A	1	1	0	1	3	4	3	NA	NA	0	$E(7) = D(1), E(11) = A(2)$
7	D	0	0	0	1	3	4	3	7	4	0	$E(11) = A(2)$
11	A	1	1	0	2	11	4	11	NA	NA	0	$E(13) = A(3), E(15) = D(2)$
13	A	2	1	1	3	13	4	15	NA	NA	2	$E(14) = A(4), E(15) = D(2)$
14	A	3	1	2	4	14	3	19	NA	NA	5	$E(15) = D(2), E(17) = A(5)$
15	D	2	1	1	2	11	4	11	15	4	0	$E(17) = A(5), E(19) = D(3)$
17	A	3	1	2	5	17	2	22	NA	NA	5	$E(19) = D(3), E(19) = A(6)$
19	D	2	1	1	3	13	4	15	19	6	2	$E(19) = A(6)$
19	A	3	1	2	6	19	4	24	NA	NA	5	$E(21) = A(7), E(22) = D(4)$
21	A	4	1	3	7	21	3	28	NA	NA	7	$E(22) = D(4), E(24) = D(5)$
22	D	3	1	2	4	14	3	19	22	8	5	$E(24) = D(5), E(27) = A(8)$
24	D	2	1	1	5	17	2	22	24	7	5	$E(27) = A(8), E(28) = D(6)$
27	A	3	1	2	8	27	2	31	NA	NA	4	$E(28) = D(6), E(31) = D(7)$
28	D	2	1	1	6	19	4	24	28	9	5	$E(31) = D(7)$
31	D	1	1	0	7	21	3	28	31	10	7	$E(33) = D(8)$

are processed that occur at the same time may affect how future events are processed. That is, the order of event processing may change the behavior of the system over time and thus the order of processing is, in general, important. However, in this particular simple system, the order of processing will not change what happens in the future. We will simply have an update of the state variables at the same time. In this instance, we will process the departure event first and then the arrival event. If you are not convinced that this will not make a difference, then I encourage you to change the ordering and continue the processing. In more complex system simulations, a priority indicator is attached to the events so that the events can be processed in a consistent manner. Rather than continuing the step-by-step processing of the events through time 31, we will present the completed table. We leave it as an exercise for the reader to continue the processing of the customers. The completed bookkeeping table at time 31 is as follows.

Given that we have simulated the bank over the time frame from 0 to 31 minutes, we can now compute performance statistics for the bank and measure how well it is operating. Two statistics that we can easily compute are the average time spent waiting in line and the average time spent in the system.

Let  $n$  be the number of customers observed to depart during the simulation. In this simulation, we had  $n = 7$  customers depart during the time frame of 0 to 31 minutes. The time frame over which we analyze the performance of the system is call the simu-

lation time horizon. In this case, the simulation time horizon is fixed and known in advance. When we perform a computer simulation experiment, the time horizon is often referred to as the *simulation run length* (or *simulation replication length*). To estimate the average time spent waiting in line and the average time spent in the system, we can simply compute the sample averages ( $\bar{T}$  and  $\bar{W}$ ) of the observed quantities ( $T_i$  and  $W_i$ ) for each departed customer.

$$\bar{T} = \frac{1}{7} \sum_{i=1}^7 T_i = \frac{4 + 4 + 6 + 8 + 7 + 9 + 10}{7} = \frac{48}{7} \cong 6.8571$$

$$\bar{W} = \frac{1}{7} \sum_{i=1}^7 W_i = \frac{0 + 0 + 2 + 5 + 5 + 5 + 7}{7} = \frac{24}{7} \cong 3.4286$$

Besides the average time spent in the system and the average time spent waiting, we might also want to compute the average number of customers in the system, the average number of customers in the queue, and the average number of busy tellers. You might be tempted to simply average the values in the  $N(t)$ ,  $B(t)$ , and  $Q(t)$  columns of the bookkeeping table. Unfortunately, that will result in an incorrect estimate of these values because a simple average will not take into account how long each of the variables persisted with its values over time. In this situation, we are really interested in computed a time average. This is because the variables,  $N(t)$ ,  $B(t)$ , and  $Q(t)$  are time-based variables. This type of variable is always encountered in discrete event simulations.

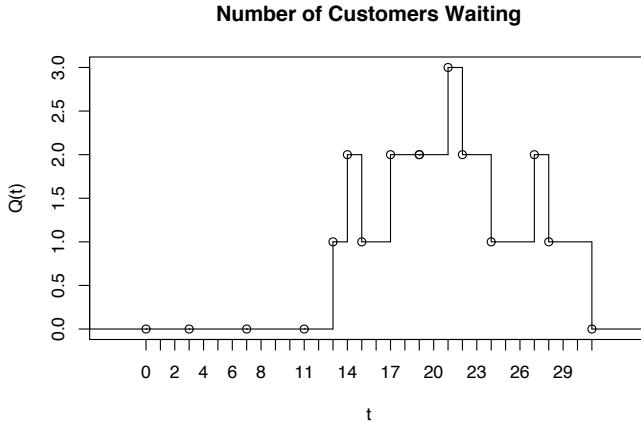
To make the discussion concrete, let's focus on the number of customers in the queue,  $Q(t)$ . Notice the number of customers in the queue,  $Q(t)$  takes on constant values during intervals of time corresponding to when the queue has a certain number of customers.  $Q(t) = \{0, 1, 2, 3, \dots\}$ . The values of  $Q(t)$  form a step function in this particular case. The following figure illustrates the step function nature of this type of variable. A realization of the values of variable is called a sample path.

That is, for a given (realized) sample path,  $Q(t)$  is a function that returns the number of customers in the queue at time  $t$ . The mean value theorem of calculus for integrals states that given a function,  $f(\bullet)$ , continuous on an interval  $(a, b)$ , there exists a constant,  $c$ , such that

$$\int_a^b f(x) dx = f(c)(b - a)$$

$$f(c) = \frac{\int_a^b f(x) dx}{(b - a)}$$

The value,  $f(c)$ , is called the mean value of the function. A similar function can be defined for  $Q(t)$ . This function is called the time-average (and represents the *mean value* of the  $Q(t)$  function):



**Figure 15:** Number of Customers Waiting in Queue for the Bank Simulation

$$\bar{Q}(n) = \frac{\int_{t_0}^{t_n} Q(t) dt}{t_n - t_0}$$

This function represents the *average with respect to time* of the given state variable. This type of statistical variable is called time-based because  $Q(t)$  is a function of time.

In the particular case where  $Q(t)$  represents the number of customers in the queue,  $Q(t)$  will take on constant values during intervals of time corresponding to when the queue has a certain number of customers. Let  $Q(t) = q_k$  for  $t_{k-1} \leq t \leq t_k$ . Then, the time-average can be rewritten as follows:

$$\bar{Q}(t) = \frac{\int_{t_0}^{t_n} Q(t) dt}{t_n - t_0} = \sum_{k=1}^n \frac{q_k(t_k - t_{k-1})}{t_n - t_0}$$

Note that  $q_k(t_k - t_{k-1})$  is the area under the curve,  $Q(t)$  over the interval  $t_{k-1} \leq t \leq t_k$  and because  $t_n - t_0 = (t_1 - t_0) + (t_2 - t_1) + (t_3 - t_2) + \dots + (t_{n-1} - t_{n-2}) + (t_n - t_{n-1})$

we can write,

$$t_n - t_0 = \sum_{k=1}^n t_k - t_{k-1}$$

The quantity  $t_n - t_0$  represents the total time over which the variable is observed. Thus, the time average is simply the area under the curve divided by the amount of time over which the curve is observed. From this equation, it should be noted that each value of  $q_k$  is weighted by the length of time that the variable has the value. If we define,  $w_k = (t_k - t_{k-1})$ , then we can re-write the time average as:

$$\bar{Q}(t) = \frac{\int_{t_0}^{t_n} Q(t) dt}{t_n - t_0} = \sum_{k=1}^n \frac{q_k(t_k - t_{k-1})}{t_n - t_0} = \frac{\sum_{k=1}^n q_k w_k}{\sum_{k=1}^n w_k}$$

This form of the equation clearly shows that each value of  $q_k$  is weighted by:

$$\frac{w_k}{\sum_{i=1}^n w_k} = \frac{w_k}{t_n - t_0} = \frac{(t_k - t_{k-1})}{t_n - t_0}$$

This is why the time average is often called the time-weighted average. If  $w_k = 1$ , then the time-weighted average is the same as the sample average.

Now we can compute the time average for  $Q(t)$ ,  $N(t)$  and  $B(t)$ . Using the following formula and noting that  $t_n - t_0 = 31$

$$\bar{Q}(t) = \sum_{k=1}^n \frac{q_k(t_k - t_{k-1})}{t_n - t_0}$$

We have that the numerator computes as follows:

$$\begin{aligned} \sum_{k=1}^n q_k(t_k - t_{k-1}) &= 0(13-0) + 1(14-13) + 2(15-14) + 1(17-15) + 2(19-17) \\ &\quad + 1(19-19) + 2(21-19) + 3(22-21) + 2(24-22) + \\ &\quad 1(27-24) + 2(28-27) + 1(31-28) = 28 \end{aligned}$$

And, the final time-weighted average number in the queue ss:

$$\bar{Q}(t) = \frac{28}{31} \cong 0.903$$

The average number in the system and the average number of busy tellers can also be computed in a similar manner, resulting in:

$$\bar{N}(t) = \frac{52}{31} \cong 1.677$$

$$\bar{B}(t) = \frac{24}{31} \cong 0.7742$$

The value of  $\bar{B}(t)$  is most interesting for this situation. Because there is only 1 teller, the fraction of the tellers that are busy is 0.7742. This quantity represents the *utilization* of the teller. The utilization of a resource represents the proportion of time that the resource is busy. Let  $c$  represent the number of units of a resource that are available. Then the utilization of the resource is defined as:

$$\bar{U} = \frac{\bar{B}(t)}{c} = \frac{\int_{t_0}^{t_n} B(t) dt}{c(t_n - t_0)}$$

Notice that the numerator of this equation is simply the total time that the resource is busy. So, we are computing the total time that the resource is busy divided by the total time that the resource could be busy,  $c(t_n - t_0)$ , which is considered the utilization.

In this simple example, when an arrival occurs, you must determine whether or not the customer will enter service or wait in the queue. When a customer departs the system, whether or not the server will become idle must be determined, by checking the queue. If the queue is empty, then the server becomes idle; otherwise the next customer in the queue begins service. In order to develop a computer simulation model of this situation, the actions that occur at an event must be represented within code. For example, the pseudo-code for the arrival event would be:

*Pseudo-code for Arrival Event*

1. schedule arrival time of next customer
  1. AT = generate time to next arrival according to inter-arrival distribution
  2. schedule arrival event at,  $t + AT$
2. check status of the servers (idle or busy)
  1. if idle, do
    1. allocate a server to the customer
    2. ST = generate service time according to the service time distribution
    3. schedule departure event at,  $t + ST$
  2. if busy, do
    1. increase number in queue by 1 (place customer in queue)

*Pseudo-code for Departure Event*

1. if queue is not empty
  1. remove next customer from the queue
  2. allocate the server to the next customer in the queue
  3. ST = generate service time according to the service time distribution
  4. schedule departure event at,  $t + ST$
2. else the queue is empty
  1. make the customer's server idle

Notice how the arrival event schedules the next arrival and the departure event may schedule the next departure event (provided the queue is not empty). Within the JSL, this pseudo-code must be represented in a class method that will be called at the appropriate event time.

To summarize, discrete-event modeling requires two key abilities:

- The ability to represent the actions associated with an event in computer code (i.e. in a class method).
- The ability to schedule events so that they will be called at the appropriate event time, i.e. the ability to have the event's logic called at the appropriate event time.

Thus, the scheduling and execution of events is critical to discrete-event modeling. The scheduling of additional events within an event is what makes the system progress through time (jumping from event to event). The JSL supports the scheduling and execution of events within its calendar and model packages. The next section overviews the libraries available within the JSL for discrete-event modeling.

## 0.27 Modeling DEDS in the JSL

Discrete event modeling within the JSL is facilitated by two packages: 1) the `jsl.simulation` package and 2) the `jsl.calendar` package. The `jsl.simulation` package has classes that implement behavior associated with simulation events and models. The `jsl.calendar` package has classes that facilitate the scheduling and processing of events.

### 0.27.1 Event Scheduling

The following classes within the `jsl.simulation` package work together to provide for the scheduling and execution of events:

- `Simulation` - An instance of `Simulation` holds the model and facilitates the running of the simulation according to run parameters such as simulation run length and number of replications.
- `Executive` - The `Executive` controls the execution of the events and works with the `calendar` package to ensure that events are executed and the appropriate model logic is called at the appropriate event time. This class is responsible for placing the events on a calendar, allowing events to be canceled, and executing the events in the correct order.
- `JSLEvent` – This class represents a simulation event. The attributes of `JSLEvent` provide information about the name, time, priority, and type of the event. The user can also check whether or not the event is canceled or if it has been scheduled. In addition, a general attribute of type `Object` is associated with an event and can be used to pass information along with the event.
- `ModelElement` – This class serves as a base class for all classes that are within a simulation model. It provides access to the scheduler and ensures that model elements participate in common simulation actions (e.g. warm up, initialization, etc.).
- `Model` - An instance of a `Model` is required to serve as the parent to any model elements within a simulation model. It is the top-level container for model elements.

- `SchedulingElement` – `SchedulingElement` is a sub-class of `ModelElement` and facilitates the scheduling of events within a simulation model. Classes that require significant event scheduling should sub-class from `SchedulingElement`.

Figure 16 illustrates the relationships between the classes `Model`, `ModelElement`, `SchedulingElement`, `Simulation`, and `Executive`. The `ModelElement` class represents the primary building block for JSL models. A `ModelElement` represents something (an element) that can be placed within an instance of a `Model`. The `Model` class subclasses `ModelElement`. Every `ModelElement` can contain many other instances of `ModelElement`. As such, an instance of a `Model` can also contain model elements. There can only be one instance of the `Model` class within the simulation. It acts as the parent (container) for all other model elements. Model elements in turn also hold other model elements. Instances arranged in this pattern form an object hierarchy that represents the simulation model. The instance of a `Simulation` holds (references) the instance of the `Model` class. `Simulation` also references the `Executive`. The instance of the `Executive` uses an instance of a class that implements the `CalendarIfc` interface. The simulation also references an instance of the `Experiment` class. The `Experiment` class allows the specification and control of the run parameters for the simulation. Every instance of a `ModelElement` must be a child of another `ModelElement` or the `Model`. This implies that instances of `ModelElement` have access to the main model, which has access to an instance of `Simulation` and thus the instance of the `Executive`. Therefore sub-classes of `ModelElement` have access to the `Executive` and can schedule events.

The `SchedulingElement` class is a special kind of `ModelElement` that facilitates the scheduling of events. `SchedulingElement` is an abstract base class for building other classes that schedule events. Sub-classes of the `SchedulingElement` class will have a set of methods that facilitate the scheduling of events.

Figure 17 illustrates the protected methods of the `SchedulingElement` class. Since these methods are protected, sub-classes will have them available through inheritance. There are two major types of scheduling methods: one for rescheduling already existing events and another for scheduling new events. The scheduling of new events results in the creation of a new `JSLEvent` and the placement of the event on the event calendar via the `Executive`.

The following code listing illustrates the key method for scheduling events within the class `SchedulingElement`. Notice that the instance of the `Executive` is called via `getExecutive()`. In addition, the user can supply information for the creation of an event such as its time, name, and priority. The user can also provide an instance of classes that implement the `EventActionIfc` interface. This interface promises to have an `action()` method. Within the `action()` method the user can provide the code that is necessary to represent the state changes associated with the event.

```
/** Creates an event and schedules it onto the event calendar. This is the main scheduling method that
 * all other scheduling methods call. The other methods are just convenience methods for this method.
 *
 * @param <T> the type associated with the attached message
```

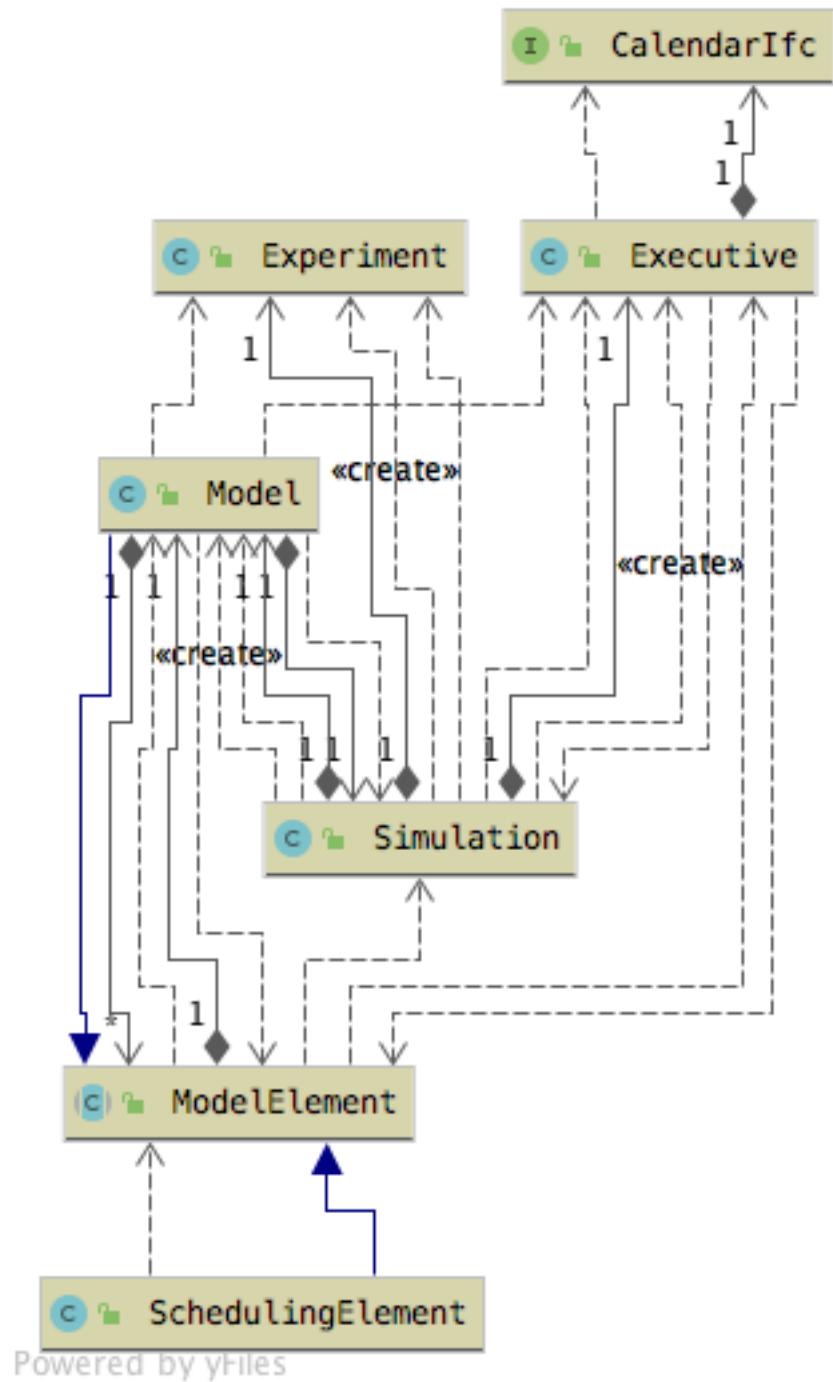
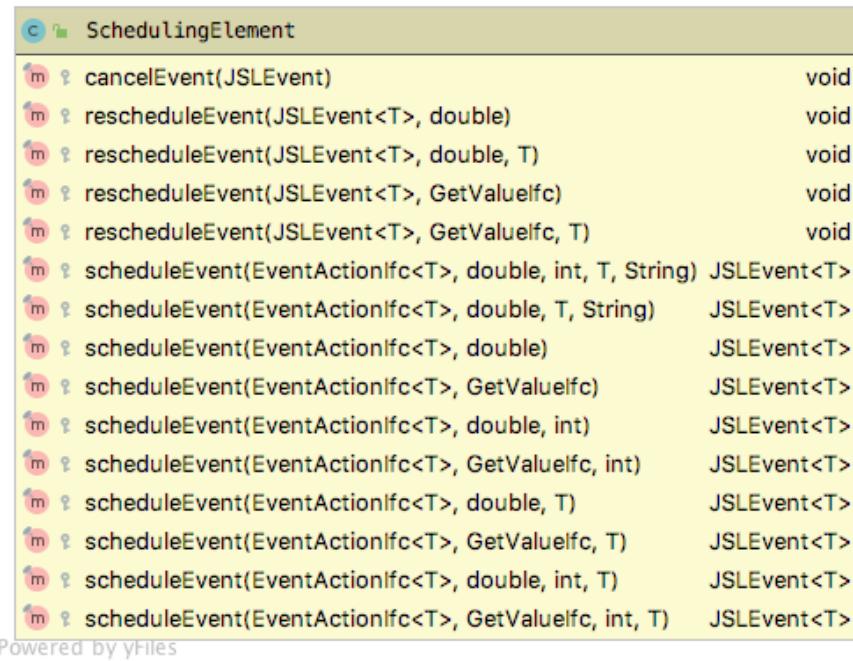


Figure 16: `jsl.simulation` Package and Relationships

**Figure 17:** SchedulingElement and its scheduling methods

```

* @param action represents an ActionListener that will handle the change of state logic
* @param time represents the inter-event time, i.e. the interval from the current time to when the
*           event will need to occur
* @param priority is used to influence the ordering of events
* @param message is a generic Object that may represent data to be transmitted with the event
* @param name the name of the event
* @return a valid JSLEvent
*/
protected final <T> JSLEvent<T> scheduleEvent(EventActionIfc<T> action, double time, int priority, T message, String name)
{
    JSLEvent<T> event = getExecutive().scheduleEvent(action, time, priority, message, name, this);
    return (event);
}

```

There are two ways that the user can provide event action code: 1) provide a class that implements the `EventActionIfc` interface and supply it when scheduling the event or 2) by treating the `EventActionIfc` as a functional interface and using Java 8's functional method representation. Providing an inner class that implements the `EventActionIfc` interface will be illustrated here.

### 0.27.2 Simple Event Scheduling Examples

This section presents two simple examples to illustrate event scheduling. The first example illustrates the scheduling of events using the `EventActionIfc` interface. The second example shows how to simulate a Poisson process and collect simple statistics.

#### 0.27.2.1 Implementing Event Actions Using the `EventActionIfc` Interface

In the first example, there will be two events scheduled with actions. The time between the events will all be deterministic. The specifics of the events are as follows:

1. Event Action One: This event occurs only once at time 10.0 and schedules the Action Two event to occur 5.0 time units later. It also prints out a message.
2. Event Action Two: This event prints out a message. Then, it schedules an action one event 15 time units into the future. It also reschedules itself to reoccur in 20 minutes.

The following code listing provides the code for this simple event example. Let's walk carefully through the construction and execution of this code.

First, the class sub-classes from `SchedulingElement`. This enables the class to have access to all the scheduling methods within `SchedulingElement` and provides one method that needs to be overridden: `initialize()`. Every `ModelElement` has an `initialize()` method. The base class, `ModelElement`'s `initialize()` method does nothing. However, the `initialize()` method is *critical* to properly modeling using instances of `ModelElement` within the JSL architecture. The purpose of the `initialize()` method is to provide code that can occur once at the beginning of each replication of the simulation, prior to the execution of any events. Thus, the `initialize()` method is the perfect location to schedule initial events onto the event calendar so that when the replications associated with the simulation are executed initial events will be on the calendar ready for execution. Notice that in this example the `initialize()` method does two things:

1. schedules the first event action one event at time 10.0 via the call: `scheduleEvent(myEventActionOne, 10.0)`
2. schedules the first action two event at time 20.0 via the call: `scheduleEvent(myEventActionTwo, 20.0)`

```
public class SchedulingEventExamples extends SchedulingElement {

    private final EventActionOne myEventActionOne;
    private final EventActionTwo myEventActionTwo;

    public SchedulingEventExamples(ModelElement parent) {
        this(parent, null);
    }
}
```

```

public SchedulingEventExamples(ModelElement parent, String name) {
    super(parent, name);
    myEventActionOne = new EventActionOne();
    myEventActionTwo = new EventActionTwo();
}

@Override
protected void initialize() {
    // schedule a type 1 event at time 10.0
    scheduleEvent(myEventActionOne, 10.0);
    // schedule an event that uses myEventAction for time 20.0
    scheduleEvent(myEventActionTwo, 20.0);
}

private class EventActionOne extends EventAction {
    @Override
    public void action(JSLEvent event) {
        System.out.println("EventActionOne at time : " + getTime());
    }
}

private class EventActionTwo extends EventAction {
    @Override
    public void action(JSLEvent jsle) {
        System.out.println("EventActionTwo at time : " + getTime());
        // schedule a type 1 event for time t + 15
        scheduleEvent(myEventActionOne, 15.0);
        // reschedule the EventAction event for t + 20
        rescheduleEvent(jsle, 20.0);
    }
}

public static void main(String[] args) {
    Simulation s = new Simulation("Scheduling Example");
    new SchedulingEventExamples(s.getModel());
    s.setLengthOfReplication(100.0);
    s.run();
}
}

```

The call `scheduleEvent(myEventAction, 20.0)` schedules an event 20 time units into the future where the event will be handled via the instance of the class `EventActionTwo`, which implements the `EventActionIfc` interface. The reference `myEventActionTwo` refers to an object of type `EventActionTwo`, which is an instance of the inner classes defined within `SchedulingEventExamples`. This variable is defined as class attribute on line 4

and an instance is created in the constructor on line 11. To summarize, the `initialize()` method is used to schedule the initial occurrences of the two types of events. The `initialize()` method occurs right before time 0.0. That is, it occurs right before the simulation clock starts.

Now, let us examine the actions that occur for the two types of events. Within the `action()` method of `EventActionOne`, we see the following code:

```
System.out.println("EventActionOne at time : " + getTime());
```

Here a simple message is printed that includes the simulation time via the inherited `getTime()` method of the `ModelElement` class. Thus, by implementing the `action()` method of the `EventActionIfc` interface, you can supply the logic that occurs when the event is executed by the simulation executive. In the implemented `EventActionTwo` class, a simple message is printed and event action one is scheduled. In addition, the `rescheduleEvent()` method is used to reschedule the event that was supplied as part of the `action()` method. Since the attributes of the event remain the same, the event is “recycled” to occur at the new scheduled time.

The main method associated with the `SchedulingEventExamples` class indicates how to create and run a simulation model. The first line of the main method creates an instance of a `simulation`. The next line makes an instance of `SchedulingEventExamples` and attaches it to the simulation model. The method call, `s.getModel()` returns an instance of the `Model` class that is associated with the instance of the `simulation`. The next line sets up the simulation to run for 100 time units and the last line tells the simulation to begin executing via the `run()` method. The output of the code is as follows:

```
EventActionOne at time : 10.0
EventActionTwo at time : 20.0
EventActionOne at time : 35.0
EventActionTwo at time : 40.0
EventActionOne at time : 55.0
EventActionTwo at time : 60.0
EventActionOne at time : 75.0
EventActionTwo at time : 80.0
EventActionOne at time : 95.0
EventActionTwo at time : 100.0
```

Notice that event action one output occurs at time 10.0. This is due to the event that was scheduled within the `initialize()` method. Event action two occurs for the first time at time 20.0 and then every 20 time units. Notice that event action one occurs at time 35.0. This is due to the event being scheduled in the `action` method of event action two.

### 0.27.2.2 Overview of Simulation Run Context

When the simulation runs, much underlying code is executed. At this stage it is not critically important to understand how this code works; however, it is useful to un-

derstand, in a general sense, what is happening. The following outlines the basic processes that are occurring when `s.run()` occurs:

1. Setup the simulation experiment
2. For each replication of the simulation:
  - a. Initialize the replication
  - b. Initialize the executive and calendar
  - c. Initialize the model and all model elements
  - d. While there are events on the event calendar or the simulation is not stopped
    - i. Determine the next event to execute
    - ii. Update the current simulation time to the time of the next event
    - iii. Call the `action()` method of the instance of `EventActionIfc` that was attached to the next event.
    - iv. Execute the actions associated with the next event
  - e. Execute end of replication model logic
3. Execute end of simulation experiment logic

Step 2(b) initializes the executive and calendar and ensures that there are no events at the beginning of the simulation. It also resets the simulation time to 0.0. Then, step 2(c) initializes the model. In this step, the `initialize()` methods of all of the model elements are executed. This is why it was important to implement the `initialize()` method in the example and have it schedule the initial events. Then, step 2(c) begins the execution of the events that were placed on the calendar. In looking at the code listings, it is not possible to ascertain how the `action()` methods are actually invoked unless you understand that during step 2(c) each scheduled event is removed from the calendar and its associated action called. In the case of the event action one and two events in the example, these actions are specified in the `action()` method of `EventActionOne` and `EventActionTwo`. After all the events in the calendar are executed or the simulation is not otherwise stopped, the replication is ended. Any clean up logic (such as statistical collection) is executed at the end of the replication. Finally, after all replications have been executed, any logic associated with ending the simulation experiment is invoked. Thus, even though the code does not directly call the event logic it is still invoked by the simulation executive because the events are scheduled. Thus, if you schedule events, you can be assured that the logic associated with the events will be executed.

### 0.27.2.3 Simulating a Poisson Process

The second simple example illustrates how to simulate a Poisson process. Recall that a Poisson process models the number of events that occur within some time interval. For a Poisson process the time between events is exponentially distributed with a mean

that is the reciprocal of the rate of occurrence for the events. For simplicity, this example simulates a Poisson process with rate 1 arrival per unit time. Thus, the mean time between events is 1.0 time unit. In this case the action is very simple, incrementing a counter that is tracking the number of events that have occurred.

The code for this example is as follows.

```
public class SimplePoissonProcess extends SchedulingElement {

    private final RandomVariable myTBE;
    private final Counter myCount;
    private final EventHandler myEventHandler;

    public SimplePoissonProcess(ModelElement parent) {
        this(parent, null);
    }

    public SimplePoissonProcess(ModelElement parent, String name) {
        super(parent, name);
        myTBE = new RandomVariable(this, new ExponentialRV(1.0));
        myCount = new Counter(this, "Counts events");
        myEventHandler = new EventHandler();
    }

    @Override
    protected void initialize() {
        scheduleEvent(myEventHandler, myTBE.getValue());
    }

    private class EventHandler extends EventAction {
        @Override
        public void action(JSLEvent evt) {
            myCount.increment();
            scheduleEvent(myEventHandler, myTBE.getValue());
        }
    }

    public static void main(String[] args) {
        Simulation s = new Simulation("Simple PP");
        new SimplePoissonProcess(s.getModel());
        s.setLengthOfReplication(20.0);
        s.setNumberOfReplications(50);
        s.run();
        SimulationReporter r = s.makeSimulationReporter();
        r.printAcrossReplicationSummaryStatistics();
    }
}
```

```

        System.out.println("Done!");
    }
}

```

There are a few new elements of this code to note. First, this example uses two new JSL model elements: `RandomVariable` and `Counter`. A `RandomVariable` is a sub-class of `ModelElement` that is used to represent random variables within a simulation model. The `RandomVariable` class must be supplied an instance of a class that implements the `RandomIfc` interface. Recall that implementations of the `RandomIfc` interface have a `get-value()` method that returns a random value and permit random number stream control. The supplied stream control is important when utilized advanced simulation statistical methods. For example, stream control is used to advance the state of the underlying stream to the next substream at the end of every replication of the model. This helps in synchronizing the use of random numbers in certain types of experimental setups.

A `Counter` is also a sub-class of `ModelElement` which facilitates the incrementing and decrementing of a variable and the statistical collection of the variable across replications. The value of the variable associated with the instance of a `Counter` is automatically reset to 0.0 at the beginning of each replication. Lines 2 and 3 within the constructor create the instances of the `RandomVariable` and the `Counter`.

Since we are modeling a Poisson process, the `initialize()` method is used to schedule the first event using the random variable that represents the time between events. This occurs on the only line of the `initialize()` method. The event logic, found in the inner class `EventHandler`, causes the counter to be incremented. Then, the next arrival is scheduled to occur. Thus, it is very easy to model an arrival process using this pattern. The last items to note are in the `main()` method of the class, where the simulation is created and run. In setting up the simulation, the run length is set to 20 time units and the number of replications associated with the simulation is set to 50.

A replication represents a sample path of the simulation that starts and ends under the same conditions. Thus, statistics collected on each replication represent independent and identically distributed observations of the simulation model's execution. In this example, there will be 50 observations of the counter observed. Since we have a Poisson process with rate 1 event per time unit and we are observing the process for 20 time units, we should expect that about 20 events should occur on average.

Right after the `run()` method is called, an instance of a `SimulationReporter` is created for the simulation. A `SimulationReporter` has the ability to write out statistical output associated with the simulation. The code uses the `printAcrossReplicationSummaryStatistics()` method to write out a simple summary report across the 50 replications for the `Counter`. Note that using the `Counter` to count the events provided for automatically collected statistics across the replications for the counter. As you can see from the output, the average number of events is close to the theoretically expected amount.

Across Replication Statistical Summary Report

```

Sat Dec 31 13:02:53 EST 2016
Simulation Results for Model: Simple_PP_Model

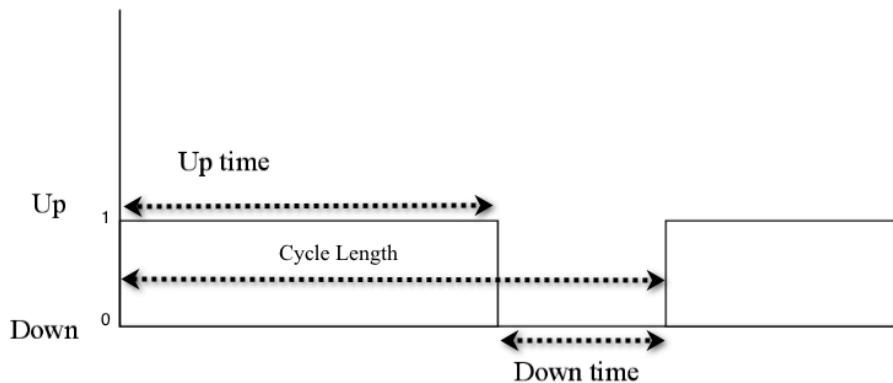
Number of Replications: 50
Length of Warm up period: 0.0
Length of Replications: 20.0

-----
Counters
-----
Name          Average      Std. Dev.   Count
-----
Counts events    20.500000     4.870779   50.000000
-----

```

### 0.27.3 Up and Down Component Example

This section further illustrates DEDS modeling with a component of a system that is subject to random failures. The component has two states UP and DOWN. The time until failure is random and governed by an exponential distribution with a mean of 1.0 time units. This represents the time that the component is in the UP state. Once the component fails, the component goes into the DOWN state. The time that the component spends in the DOWN state is governed by an exponential distribution with a mean of 2.0 time units. In this model, we are interested in estimating the proportion of time that the component is in the UP state and tracking the number of failures over the running time of the simulation. In addition, we are interested in measuring the cycle length of the component. The cycle length is the time between entering the UP state. The cycle length should be equal to the sum of the time spent in the up and down states.



**Figure 18:** Up and Down Component Process

Figure 18 illustrates the dynamics of the component over time.

The following steps are useful in developing this model:

1. Conceptualize the system/objects and their states
2. Determine the events and the actions associated with the events
3. Determine how to represent the system and objects as ModelElements
4. Determine how to initialize and run the model

The first step is to conceptualize how to model the system and the state of the component. A model element, UpDownComponent, will be used to model the component. To track the state of the component, it is necessary to know whether or not the component is UP or DOWN. A variable can be used to represent this state. However, since we need to estimate the proportion of time that the component is in the UP state, a TimeWeighted variable will be used. TimeWeighted is a sub-class of ModelElement that facilitates the observation and statistical collection of time-based variables. Time-bases variables, which are discussed further in the next Chapter, are a type of variable that changes values at particular instants of time. Time-based variables must have time-weighted statistics collected. Time-weighted statistics weights the value of the variable by the proportion of time that the variable is set to a value. To collect statistics on the cycle length we can use a ResponseVariable. ResponseVariable is a sub-class of ModelElement that can take on values within a simulation model and allows easy statistical observation of its values during the simulation run. This class provides observation-based statistical collection. Further discussion of observation-based statistics will be presented in the next Chapter.

Because this system is so simple the required performance measures can be easily computed theoretically. According to renewal theory, the probability of being in the UP state in the long run should be equal to:

$$P(UP) = \frac{\theta_u}{\theta_u + \theta_d} = \frac{1.0}{1.0 + 2.0} = 0.\overline{33}$$

where  $\theta_u$  is the mean of the up-time distribution and  $\theta_d$  is the mean of the down-time distribution. In addition, the expected cycle length should be  $\theta_u + \theta_d = 3.0$ .

The UpDownComponent class extends the SchedulingElement class and has object references to instances of RandomVariable, TimeWeighted, ResponseVariable, and Counter classes. Within the constructor of UpDownComponent, we need to create the instances of these objects for use within the class, as shown in the following code fragment.

```
public class UpDownComponent extends SchedulingElement {

    public static final int UP = 1;
    public static final int DOWN = 0;
    private RandomVariable myUpTime;
    private RandomVariable myDownTime;
    private TimeWeighted myState;
    private ResponseVariable myCycleLength;
```

```

private Counter myCountFailures;
private final UpChangeAction myUpChangeAction = new UpChangeAction();
private final DownChangeAction myDownChangeAction = new DownChangeAction();
private double myTimeLastUp;

public UpDownComponent(ModelElement parent) {
    this(parent, null);
}

public UpDownComponent(ModelElement parent, String name) {
    super(parent, name);
    RVariateIfc utd = new ExponentialRV(1.0);
    RVariateIfc dtd = new ExponentialRV(2.0);
    myUpTime = new RandomVariable(this, utd, "up time");
    myDownTime = new RandomVariable(this, dtd, "down time");
    myState = new TimeWeighted(this, "state");
    myCycleLength = new ResponseVariable(this, "cycle length");
    myCountFailures = new Counter(this, "count failures");
}

```

Lines 3 and 4 define two constants to represent the up and down states. Lines 5-9 declare additional references needed to represent the up and down time random variables and the variables that need statistical collection (myState, myCycleLength, and myCountFailures). Lines 10 and 11 define and create the event actions associated with the end of the up-time and the end of the down time. The variable myTimeLastUp is used to keep track of the time that the component last changed into the UP state, which allows the cycle length to be collected. In lines 20-23, the random variables for the up and downtime are constructed using exponential distributions.

As shown in the following code listing, the `initialize()` method sets up the component. The variable `myTimeLastUp` is set to 0.0 in order to assume that the last time the component was in the UP state started at time 0.0. Thus, we are assuming that the component starts the simulation in the UP state. Finally, in line 7 the initial event is scheduled to cause the component to go down according to the uptime distribution. This is the first event and then the component can start its regular up and down pattern. In the action associated with the change to the UP state, line 18 sets the state to UP. Line 22 schedules the time until the component goes down. Line 16 causes statistics to be collected on the value of the cycle length. The code `getTime() - myTimeLastUp` represents the elapsed time since the value of `myTimeLastUp` was set (in line 20), which represents the cycle length. The `DownChangeAction` is very similar. Line 31 counts the number of failures (times that the component has gone down). Line 32 sets the state of the component to DOWN and line 34 schedules when the component should next transition into the UP state.

```

public void initialize() {
    // assume that the component starts in the UP state at time 0.0
    myTimeLastUp = 0.0;
    myState.setValue(UP);
    // schedule the time that it goes down
    scheduleEvent(myDownChangeAction, myUpTime.getValue());
    //schedule(myDownChangeAction).name("Down").in(myUpTime).units();
}

private class UpChangeAction extends EventAction {

    @Override
    public void action(JSLEvent event) {
        // this event action represents what happens when the component goes up
        // record the cycle length, the time btw up states
        myCycleLength.setValue(getTime() - myTimeLastUp);
        // component has just gone up, change its state value
        myState.setValue(UP);
        // record the time it went up
        myTimeLastUp = getTime();
        // schedule the down state change after the uptime
        scheduleEvent(myDownChangeAction, myUpTime.getValue());
    }
}

private class DownChangeAction extends EventAction {

    @Override
    public void action(JSLEvent event) {
        // component has just gone down, change its state value
        myCountFailures.increment();
        myState.setValue(DOWN);
        // schedule when it goes up afer the down time
        scheduleEvent(myUpChangeAction, myDownTime.getValue());
    }
}

```

The following listing presents the code to construct and execute the simulation. This code is very similar to previously presented code for running a simulation. In line 3, the simulation is constructed. Then, in line 5, the model associated with the simulation is accessed. This model is then used to construct an instance of the UpDownComponent in line 7. Finally, lines 9-15 represent getting a SimulationReporter, running the simulation, and causing output to be written to the console.

```

public static void main(String[] args) {
    // create the simulation
    Simulation s = new Simulation("UpDownComponent");
    s.turnOnDefaultEventTraceReport();
    s.turnOnLogReport();
    // get the model associated with the simulation
    Model m = s.getModel();
    // create the model element and attach it to the model
    UpDownComponent tv = new UpDownComponent(m);
    // make the simulation reporter
    SimulationReporter r = s.makeSimulationReporter();
    // set the running parameters of the simulation
    s.setNumberOfReplications(5);
    s.setLengthOfReplication(5000.0);
    // tell the simulation to run
    s.run();
    r.printAcrossReplicationSummaryStatistics();
}

```

The results for the average time in the up state and the cycle length are consistent with the theoretically computed results.

```

-----
Across Replication Statistical Summary Report
Sat Dec 31 18:31:27 EST 2016
Simulation Results for Model: UpDownComponent_Model

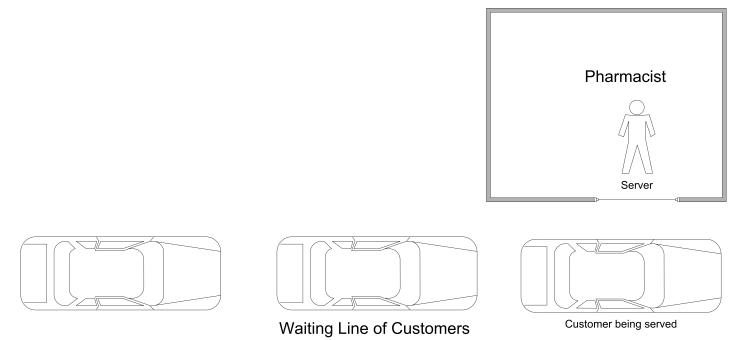
Number of Replications: 30
Length of Warm up period: 0.0
Length of Replications: 5000.0
-----
Response Variables
-----
Name          Average      Std. Dev.      Count
-----
state         0.335537    0.005846    30.000000
cycle length  2.994080    0.040394    30.000000
-----
Counters
-----
Name          Average      Std. Dev.      Count
-----
count failures 1670.066667  22.996152    30.000000

```

This example only scratches the surface of what is possible. Imagine if there were 20 components. We could easily create 20 of instances and add them to the model. Even more interesting would be to define the state of the system based on which components were in the up state. This would be the beginning of modeling the reliability of a complex system. This type of modeling can be achieved by making the individual model elements (e.g. UpDownComponent) more reusable and allow the modeling objects to interact in complex ways. More complex modeling will be the focus of the next chapter.

#### 0.27.4 Modeling a Simple Queueing System

This example considers a small pharmacy that has a single line for waiting customers and only one pharmacist. Assume that customers arrive at a drive through pharmacy window according to a Poisson distribution with a mean of 10 per hour. The time that it takes the pharmacist to serve the customer is random and data has indicated that the time is well modeled with an exponential distribution with a mean of 3 minutes. Customers who arrive to the pharmacy are served in the order of arrival and enough space is available within the parking area of the adjacent grocery store to accommodate any waiting customers.



**Figure 19:** Drive Through Pharmacy

The drive through pharmacy system can be conceptualized as a single server waiting line system, where the server is the pharmacist. An idealized representation of this system is shown in Figure 19. If the pharmacist is busy serving a customer, then additional customers will wait in line. In such a situation, management might be interested in how long customers wait in line, before being served by the pharmacist. In addition, management might want to predict if the number of waiting cars will be large. Finally, they might want to estimate the utilization of the pharmacist in order to ensure that he or she is not too busy.

When modeling the system first question to ask is: *What is the system?* In this situation, the system is the pharmacist and the potential customers as idealized in Figure 19. Now you should consider the entities of the system. An entity is a conceptual thing

of importance that flows through a system potentially using the resources of the system. Therefore, one of the first questions to ask when developing a model is: *What are the entities?* In this situation, the entities are the customers that need to use the pharmacy. This is because customers are discrete things that enter the system, flow through the system, and then depart the system.

Since entities often use things as they flow through the system, a natural question is to ask: *What are the resources that are used by the entities?* A resource is something that is used by the entities and that may constrain the flow of the entities within the system. Another way to think of resources is to think of the things that provide service in the system. In this situation, the entities “use” the pharmacist in order to get their medicine. Thus, the pharmacist is a resource.

Another useful conceptual modeling tool is the *activity diagram*. An *activity* is an operation that takes time to complete. An activity is associated with the state of an object over an interval of time. Activities are defined by the occurrence of two events which represent the activity’s beginning time and ending time and mark the entrance and exit of the state associated with the activity. An activity diagram is a pictorial representation of the process (steps of activities) for an entity and its interaction with resources while within the system. If the entity is a temporary entity (i.e. it flows through the system) the activity diagram is called an activity flow diagram. If the entity is permanent (i.e. it remains in the system throughout its life) the activity diagram is called an activity cycle diagram. The notation of an activity diagram is very simple, and can be augmented as needed to explain additional concepts:

Queues: shown as a circle with queue labeled inside

Activities: shown as a rectangle with appropriate label inside

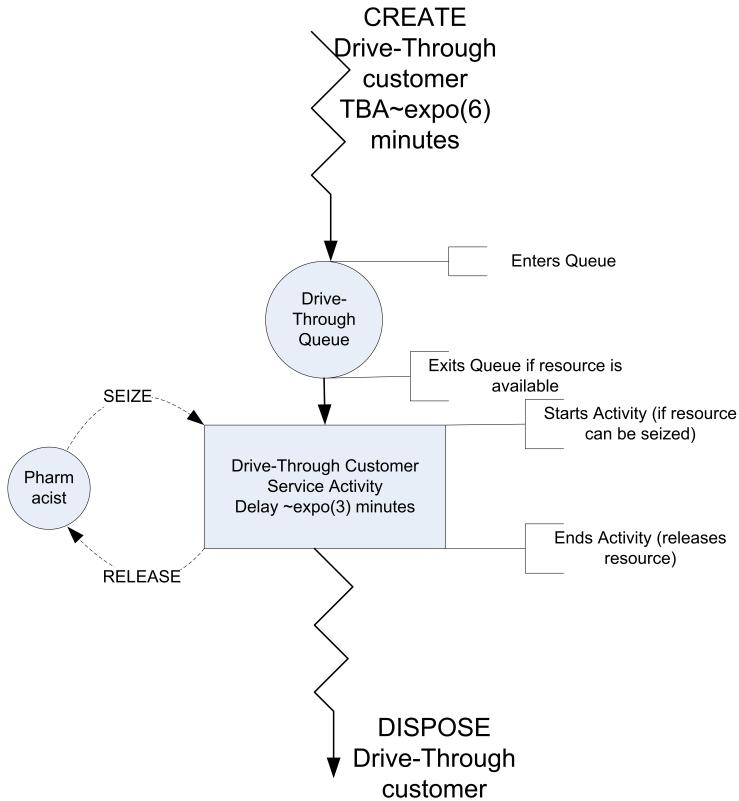
Resources: shown as small circles with resource labeled inside

Lines/arcs: indicating flow (precedence ordering) for engagement of entities in activities or for obtaining resources. Dotted lines are used to indicate the seizing and releasing of resources.

zigzag lines: indicate the creation or destruction of entities

Activity diagrams are especially useful for illustrating how entities interact with resources. In addition, activity diagrams help in finding the events and in identifying some the state changes that must be modeled. Activity diagrams are easy to build by hand and serve as a useful communication mechanism. Since they have a simple set of symbols, it is easy to use an activity diagram to communicate with people who have little simulation background. Activity diagrams are an excellent mechanism to document a conceptual model of the system before building the model.

Figure 20 shows the activity diagram for the pharmacy situation. The diagram describes the life of an entity within the system. The zigzag lines at the top of the diagram indicate the creation of an entity. Consider following the life of the customer through the pharmacy. Following the direction of the arrows, the customers are first created and then enter the queue. Notice that the diagram clearly shows that there is a queue



**Figure 20:** Activity Diagram of Drive through Pharmacy

for the drive-through customers. You should think of the entity flowing through the diagram. As it flows through the queue, the customer attempts to start an activity. In this case, the activity requires a resource. The pharmacist is shown as a resource (circle) next to the rectangle that represents the service activity.

The customer requires the resource in order to start its service activity. This is indicated by the dashed arrow from the pharmacist (resource) to the top of the service activity rectangle. If the customer does not get the resource, they wait in the queue. Once they receive the number of units of the resource requested, they proceed with the activity. The activity represents a delay of time and in this case the resource is used throughout the delay. After the activity is completed, the customer releases the pharmacist (resource). This is indicated by another dashed arrow, with the direction indicating that the units of the resource are being put back or released. After the customer completes its service activity, the customer leaves the system. This is indicated with the zigzag lines going to no-where and indicating that the object leaves the system and is disposed. The conceptual model of this system can be summarized as follows:

System: The system has a pharmacist that acts as a resource, customers that act as

entities, and a queue to hold the waiting customers. The state of the system includes the number of customers in the system, in the queue, and in service.

**Events:** Arrivals of customers to the system, which occur within an inter-event time that is exponentially distributed with a mean of 6 minutes.

**Activities:** The service time of the customers are exponentially distributed with a mean of 3 minutes.

**Conditional delays:** A conditional delay occurs when an entity has to wait for a condition to occur in order to proceed. In this system, the customer may have to wait in a queue until the pharmacist becomes available.

With an activity diagram and pseudo-code such as this available to represent a solid conceptual understanding of the system, you can begin the model development process.

In the current example, pharmacy customers arrive according to a Poisson process with a mean of  $\lambda = 10$  per hour. According to probability theory, this implies that the time between arrivals is exponentially distributed with a mean of  $(1/\lambda)$ . Thus, for this situation, the mean time between arrivals is 6 minutes.

$$\frac{1}{\lambda} = \frac{1 \text{ hour}}{10 \text{ customers}} \times \frac{60 \text{ minutes}}{1 \text{ hour}} = \frac{6 \text{ minutes}}{\text{customers}}$$

Let's assume that the pharmacy is open 24 hours a day, 7 days a week. In other words, it is always open. In addition, assume that the arrival process does not vary with respect to time. Finally, assume that management is interested in understanding the long term behavior of this system in terms of the average waiting time of customers, the average number of customers, and the utilization of the pharmacist.

To simulate this situation over time, you must specify how long to run the model. Ideally, since management is interested in long run performance, you should run the model for an infinite amount of time to get long term performance; however, you probably don't want to wait that long! For the sake of simplicity, assume that 10,000 hours of operation is long enough.

The logic of this model follows very closely the discussion of the bank teller example. The following code listing presents the definition of the variables and their creation.

```
public class DriveThroughPharmacy extends SchedulingElement {

    private int myNumPharmacists;
    private Queue<QObject> myWaitingQ;
    private RandomIfc myServiceRS;
    private RandomIfc myArrivalRS;
    private RandomVariable myServiceRV;
    private RandomVariable myArrivalRV;
    private TimeWeighted myNumBusy;
```

```

private TimeWeighted myNS;
private ResponseVariable mySysTime;
private ArrivalEventAction myArrivalEventAction;
private EndServiceEventAction myEndServiceEventAction;
private Counter myNumCustomers;

public DriveThroughPharmacy(ModelElement parent) {
    this(parent, 1,
        new ExponentialRV(1.0), new ExponentialRV(0.5));
}

public DriveThroughPharmacy(ModelElement parent, int numServers) {
    this(parent, numServers, new ExponentialRV(1.0), new ExponentialRV(0.5));
}

public DriveThroughPharmacy(ModelElement parent, int numServers, RandomIfc ad, RandomIfc sd) {
    super(parent);
    setNumberOfPharmacists(numServers);
    setServiceRS(sd);
    setArrivalRS(ad);
    myWaitingQ = new Queue<>(this, "PharmacyQ");
    myNumBusy = new TimeWeighted(this, 0.0, "NumBusy");
    myNS = new TimeWeighted(this, 0.0, "# in System");
    mySysTime = new ResponseVariable(this, "System Time");
    myNumCustomers = new Counter(this, "Num Served");
    myArrivalEventAction = new ArrivalEventAction();
    myEndServiceEventAction = new EndServiceEventAction();
}

```

The RandomVariable class is used to model the time between arrivals and the service time random variables. The TimeWeighted class is used to model the number of busy servers and the number of customers in the system. A ResponseVariable is used to model the time spent in the system. There are two events represented by implementing the EventActionIfc interface to model the arrival event and the end of service event. Finally, a new JSL class, the Queue class, is used to model the waiting line within the system. The Queue class is a sub-class of ModelElement that is able to hold instances of the class QoSObject and will automatically collect statistics on the number in the queue and the time spent in the queue. The following code listing shows the logic required to model the arrivals to the pharmacy.

```

protected void initialize() {
    super.initialize();
    // start the arrivals
    scheduleEvent(myArrivalEventAction, myArrivalRV);
}

```

```

}

private class ArrivalEventAction extends EventAction {
    @Override
    public void action(JSLEvent event) {
        // schedule the next arrival
        scheduleEvent(myArrivalEventAction, myArrivalRV);
        enterSystem();
    }
}

private void enterSystem() {
    myNS.increment(); // new customer arrived
    QObject arrivingCustomer = new QObject(getTime());

    myWaitingQ.enqueue(arrivingCustomer); // enqueue the newly arriving customer
    if (myNumBusy.getValue() < myNumPharmacists) { // server available
        myNumBusy.increment(); // make server busy
        QObject customer = myWaitingQ.removeNext(); //remove the next customer
        // schedule end of service, include the customer as the event's message
        scheduleEvent(myEndServiceEventAction, myServiceRV, customer);
    }
}

```

In line 4 the first arrival event is scheduled within the `initialize()` method. The `ArrivalEventAction` implementation schedules the next arrival using the time between arrival random variable and calls a private method named `enterSystem()`. This method handles the logic for when a customer enters the system. First, the number in the system is incremented and then the customer is enqueued. In line 17, an instance of a `QObject` is created and its creation time is supplied as the current simulation time using `getTime()`. Then, the instance of the `Queue` class, `myWaitingQ`, is used to place the customer in the queue using the `enqueue()` method. Note that even if the customer receives immediate service, we still need to place the customer in the queue because we need to correctly record that there was a zero wait time. In lines 19-24, the number of busy servers is checked against the number of pharmacists. If a server is available, then the server is made busy, the customer is removed from the queue, and the end of service for the customer is scheduled. The scheduling of the end of service is particularly important to note. In line 23, the reference to the `QObject` is supplied when calling the `scheduleEvent()` method. This method has signature:

```

/** Creates an event and schedules it onto the event calendar
 * @param <T> the type associated with the attached message
 * @param action represents an ActionListener that will handle the change of state logic
 * @param time represents the inter-event time, i.e. the interval from the current time to when the

```

```

*           event will need to occur
* @param message is a generic Object that may represent data to be transmitted with the event
* @return a valid JSLEvent
*/
protected final <T> JSLEvent<T> scheduleEvent(EventActionIfc<T> action, GetValueIfc time, T message) {
    return (scheduleEvent(action, time.getValue(), JSLEvent.DEFAULT_PRIORITY, message, getName()));
}

```

Notice that the last argument of the method is of type  $\tau$ , which is defined as a generic parameter for the method. Thus, using this method, any instance of any class can be supplied. Notice also that the `GetValueIfc` interface is used to supply the time. This is why just the name of the service time random variable can be used. The method automatically called the `getValue()` method of the argument in order to determine the time until the event's occurrence.

The following code fragment presents the logic associated with the end of service event.

```

private class EndServiceEventAction implements EventActionIfc<Q0bject> {

    @Override
    public void action(JSLEvent<Q0bject> event) {
        myNumBusy.decrement(); // customer is leaving server is freed
        if (!myWaitingQ.isEmpty()) { // queue is not empty
            Q0bject customer = myWaitingQ.removeNext(); //remove the next customer
            myNumBusy.increment(); // make server busy
            // schedule end of service
            scheduleEvent(myEndServiceEventAction, myServiceRV, customer);
        }
        departSystem(event.getMessage());
    }

    private void departSystem(Q0bject departingCustomer) {
        mySysTime.setValue(getTime() - departingCustomer.getCreateTime());
        myNS.decrement(); // customer left system
        myNumCustomers.increment();
    }
}

```

First the number of busy servers is decremented because the service is becoming idle. Then, the queue is checked to see if it is not empty. If the queue is not empty, then the next customer must be removed, the server made busy again and the customer scheduled into service. Finally, the private method, `departSystem()` is called. The argument to this method is `event.getMessage()`. The `getMessage()` method of `JSLEvent` will return the object that was scheduled with the event. Since the `JSLEvent` was provided a `qob-`

ject for the generic type in the signature of the `action()` method, we do not need to cast this object to type `Qobject`. The `departSystem()` method simply collects statistics using the `ResponseVariable`, `mySysTime` and the `TimeWeighted`, `myNS`. These objects represent the system time and the number in the system, respectively.

The following method can be used to run the model based on a desired number of servers.

```
public static void runModel(int numServers) {
    Simulation sim = new Simulation("Drive Through Pharmacy");
    sim.setNumberOfReplications(30);
    sim.setLengthOfReplication(20000.0);
    sim.setLengthOfWarmUp(5000.0);
    // add DriveThroughPharmacy to the main model
    DriveThroughPharmacy dtp = new DriveThroughPharmacy(sim.getModel(), numServers);
    dtp.setArrivalRS(new ExponentialRV(6.0));
    dtp.setServiceRS(new ExponentialRV(3.0));

    sim.run();
    sim.printHalfWidthSummaryReport();
}
```

The reports indicate that customers wait about 3 minutes on average in the line. The utilization of the pharmacist is about 50%. This means that about 50% of the time the pharmacist was busy. For this type of system, this is probably not a bad utilization, considering that the pharmacist probably has other in-store duties. The reports also indicate that there was less than one customer on average waiting for service.

```
Across Replication Statistical Summary Report
Mon Jan 02 16:05:23 CST 2017
Simulation Results for Model: Drive Through Pharmacy_Model
```

Name	Average	Std. Dev.	Count
PharmacyQ : Number In Q	0.490898	0.065510	30.000000
PharmacyQ : Time In Q	2.960012	0.351724	30.000000
NumBusy	0.495274	0.016848	30.000000
# in System	0.986173	0.080988	30.000000
System Time	5.950288	0.403270	30.000000

---

This single server waiting line system is a very common situation in practice. In fact, this exact situation has been studied mathematically through a branch of operations research called queuing theory. For specific modeling situations, formulas for the long term performance of queuing systems can be derived. This particular pharmacy model happens to be an example of an M/M/1 queuing model. The first M stands for Markov arrivals, the second M stands for Markov service times, and the 1 represents a single server. Markov was a famous mathematician who examined the exponential distribution and its properties. According to queuing theory, the expected number of customer in queue,  $L_q$ , for the M/M/1 model is:

$$\begin{aligned} L_q &= \frac{\rho^2}{1 - \rho} \\ \rho &= \lambda/\mu \\ \lambda &= \text{arrival rate to queue} \\ \mu &= \text{service rate} \end{aligned}$$

In addition, the expected waiting time in queue is given by  $W_q = L_q/\lambda$ . In the pharmacy model,  $\lambda = 1/6$ , i.e. 1 customer every 6 minutes on average, and  $\mu = 1/3$ , i.e. 1 customer every 3 minutes on average. The quantity,  $\rho$ , is called the utilization of the server. Using these values in the formulas for  $L_q$  and  $W_q$  results in:

$$\begin{aligned} \rho &= 0.5 \\ L_q &= \frac{0.5 \times 0.5}{1 - 0.5} = 0.5 \\ W_q &= \frac{0.5}{1/6} = 3 \text{ minutes} \end{aligned}$$

In comparing these analytical results with the simulation results, you can see that they match to within statistical error. Later in this text, the analytical treatment of queues and the simulation of queues will be developed. These analytical results are available for this special case because the arrival and service distributions are exponential; however, simple analytical results are not available for many common distributions, e.g. lognormal. With simulation, you can easily estimate the above quantities as well as many other performance measures of interest for wide ranging queuing situations. For example, through simulation you can easily estimate the chance that there are 3 or more cars waiting.

## 0.28 Summary

This chapter introduced how to model discrete event dynamic systems using the JSL. The JSL facilitates the model building process, the model running process, and the output analysis process.

The model elements covered included:

`Model`: Used to hold all model elements. Automatically created by the `Simulation` class.

`ModelElement`: Used as an abstract base class for creating new model elements for a simulation.

`SchedulingElement`: Used as an abstract base class for creating new model elements for a simulation that need to be able to schedule events.

`RandomVariable`: A sub-class of `ModelElement` used to model randomness within a simulation.

`ResponseVariable`: A sub-class of `ModelElement` used to collect statistics on observation-based variables.

`TimeWeighted`: A sub-class of `ModelElement` used to collect statistics on time-weighted variables in the model.

`Counter`: A sub-class of `ModelElement` used to count occurrences and collect statistics.

`Simulation`: Used to create and control a simulation model.

`SimulationReporter`: Used to gather and report statistics on a simulation model.

`JSLEvent`: Used to model different events scheduled in time during a simulation.

`EventActionIfc`: An interface used to define an `action()` method that represents event logic within the simulation.

The JSL has many other facets that have yet to be touched upon. Not only does the JSL allow the modeler to build and analyze simulation models, but it also facilitates data collection, statistical analysis, and experimentation.

The next chapter will dive deeper into how to use the JSL to model discrete-event oriented simulation situations.

# Modeling with Queues, Resources, and Stations

## LEARNING OBJECTIVES

- To be able to define and explain the key elements of discrete event modeling
- To be able to model arrival processes
- To be able to model a simple queueing station
- To be able to model inter-connected stations
- To be able to introductory resource concepts

In this chapter, a method for modeling the operation of a system by describing its components is presented. In the simplest sense, a system can be thought of as a set of objects, where an object is an element of the system that interacts with other objects. This chapter takes an object-oriented view of the system by representing the dynamic behavior of the system via objects that react to discrete-events.

## 0.29 Terminology of Simulation Modeling

When developing a simulation model using the event-view, there are a number of terms and concepts that are often used. Before learning some of these concepts in more detail, it is important that you begin with an understanding of some of the vocabulary used within simulation. The following terms will be used throughout the text:

**System** A set of inter-related components that act together over time to achieve common objectives.

**Parameters** Quantities that are properties of the system that do not change. These are typically quantities (variables) that are part of the environment that the modeler feels cannot be controlled or changed. Parameters are typically model inputs in the form of variables.

**Variables** Quantities that are properties of the system (as a whole) that change or are determined by the relationships between the components of the system as it evolves through time.

**System State** A "snap shot" of the system at a particular point in time characterized by the values of the variables that are necessary for determining the future evolution of the system from the present time. The minimum set of variables that are necessary to describe the future evolution of the system is called the system's state variables.

**Entity** An object of interest in the system whose movement or operation within the system may cause the occurrence of events.

**Attribute** A property or variable that is associated with an entity.

**Event** An instantaneous occurrence or action that changes the state of the system at a particular point in time.

**Activity** An interval of time bounded by two events (start event and end event).

**Resource** A limited quantity of items that are used (e.g. seized and released) by entities as they proceed through the system. A resource has a capacity that governs the total quantity of items that may be available. All the items in the resource are homogeneous, meaning that they are indistinguishable. If an entity attempts to seize a resource that does not have any units available it must wait in a queue.

**Queue** A location that holds entities when their movement is constrained within the system.

**Future Event List** A list that contains the time ordered sequence of events for the simulation.

When developing models, it will be useful to identify the elements of the system that fit some of these definitions. An excellent place to develop an understanding of these concepts is with entities because entities represent things that flow through and are processed by the system.

## 0.30 Entities and Attributes

When modeling a system, there are often many types of entities. For example, consider a retail store. Besides customers, the products might also be considered as entities. The products are received by the store and wait on the shelves until customers select them for purchase. Entities may come in groups and then are processed individually or they might start out as individual units that are formed into groups. For example, a truck arriving to the store may be an entity that consists of many pallets that contain products. The customers select the products from the shelves and during the check out process the products are placed in bags. The customers then carry their bags to their cars. Entities are uniquely identifiable within the system. If there are two customers in the store, they can be distinguished by the *values* of their attributes. For example,

considering a product as an entity, it may have attributes *serial number*, *weight*, *category*, and *price*. The set of attributes for a type of entity is called its *attribute set*. While all products might have these attributes, they do not necessarily have the same values for each attribute. For example, consider the following two products:

- (serial number = 12345, weight = 8 ounces, category = green beans, price = \$0.87)
- (serial number = 98765, weight = 8 ounces, category = corn, price = \$1.12)

The products carry or retain these attributes and their values as they move through the system. In other words, attributes are attached to or associated with entities. The values of the attributes might change during the operation of the system. For example, a mark down on the price of green beans might occur after some period of time. Attributes can be thought of as variables that are attached to entities.

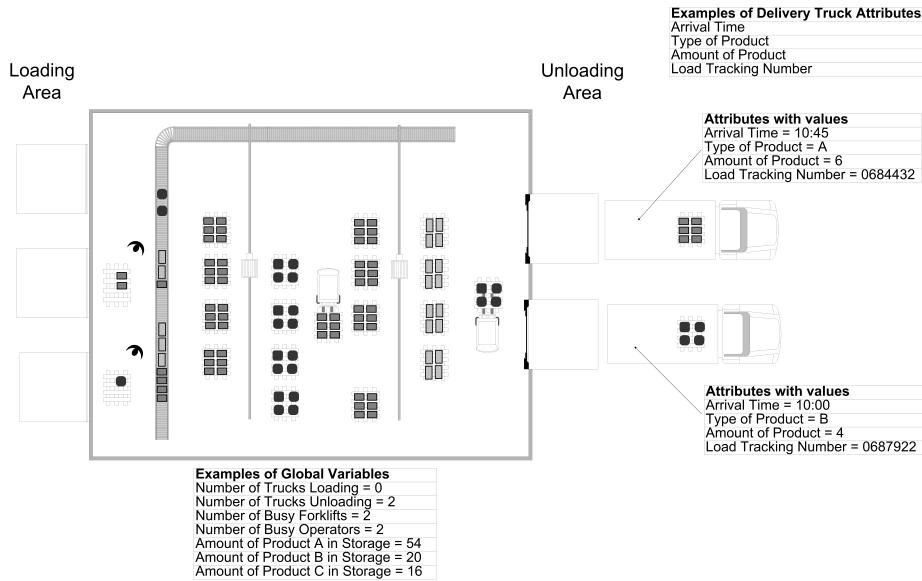
Not all information in a system is local to the entities. For example, the number of customers in the store, the number of carts, and the number of check out lanes are all characteristics of the system. These types of data are called *system attributes*. In simulation models, that take on an object-oriented nature, this information can be modeled within a class that represents the system as a whole. By making these quantities visible at the system level, the information can be shared between the different components of the system.

Figure 21 illustrates the difference between global (system) variables and entities with their attributes in the context of a warehouse. In the figure, the trucks are entities with attributes: arrival time, type of product, amount of product, and load tracking number. Notice that both of the trucks have these attributes, but each truck has different *values* for their attributes. The figure also illustrates examples of system-wide variables, such as, number of trucks loading, number of trucks unloading, number of busy forklifts, etc. This type of information belongs to the whole system.

Once a basic understanding of the system is accomplished through by conceptualizing system variables, the entities, and their attributes and various system components, you must start to understand the events that may occur within the system. In order for entities to flow through the system, there needs to be mechanisms for causing events to occur that represent the arrival of objects to the system. The following section describes how the JSL allows for the modeling of a pattern of events.

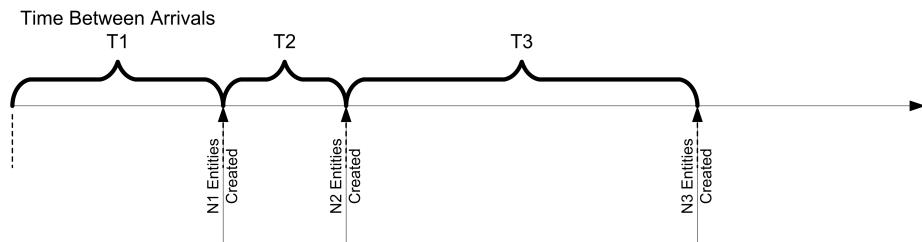
## 0.31 Event Generators

A basic mechanism by which the occurrence of a patterned sequence of events is modeled is through the `EventGenerator` class. The `EventGenerator` class defines a repeating pattern for a sequence of events. The time between arrivals specifies an ordered sequence of events in time at which events are created and introduced to the model. At each event, the modeler can define the actions (state changes) that need to occur. The first event is governed by the specification of the time until the first event, which may be stochastic. The maximum number of events to occur in the sequence can be specified.



**Figure 21:** Variables and Attributes within a System

Figure 22 illustrates a compound arrival process where the time of the first arrival is given by  $T_1$ , the time of the second arrival is given by  $T_1 + T_2$ , and the time of the third arrival is given by  $T_1 + T_2 + T_3$ . In the figure, the number of arriving entities at each arriving event is given by  $N_1$ ,  $N_2$ , and  $N_3$  respectively. The modeler can easily specify this type of arrival process using the EventGenerator class.



**Figure 22:** Example Arrival Process

For example, to specify a Poisson arrival process with mean rate  $\lambda$ , an EventGenerator can be used. Why does this specify a Poisson arrival process? Because the time between arrivals for a Poisson process with rate  $\lambda$  is exponentially distributed with the mean of the exponential distribution being  $1/\lambda$ . The EventGenerator needs to have its time between arrival distribution specified as an exponential distribution with mean equal to  $1/\lambda$ . To specify a compound arrival process, use a random variable to represent the number of occurrences per event. For example, suppose you have a compound Poisson

process<sup>20</sup> where the distribution for the number created at each arrival is governed by a discrete distribution.

$$P(X = x) = \begin{cases} 0.2 & x = 1 \\ 0.3 & x = 2 \\ 0.5 & x = 3 \end{cases}$$

The following code illustrates how to use an `EventGenerator` for this compound Poisson process with the number of arrivals specified with a discrete empirical random variable. Let's go through this example before exploring the details of the `EventGenerator` class. In line 3 the `EventGenerator` is declared and in line 17 the instance of the `EventGenerator` is created. Notice how the random variable representing the time between arrivals is provided to the `EventGenerator` constructor. The `EventGenerator` class uses classes that implement the `EventGeneratorActionIfc` interface. This interface defines a single method, called `generate()`, which is called to supply logic when the generated event occurs. An inner class called `Arrivals` implements the `EventGeneratorActionIfc` interface. Notice how a counter is used to count the number of events and a different counter is used to count the number of arrivals based on the random variable representing the number of occurrences governed by the discrete empirical distribution function.

```
public class EventGeneratorCPP extends SchedulingElement {

    protected EventGenerator myArrivalGenerator;
    protected Counter myEventCounter;
    protected Counter myArrivalCounter;
    protected RandomVariable myTBA;
    protected RandomVariable myNumArrivals;

    public EventGeneratorCPP(ModelElement parent) {
        this(parent, 1.0, null);
    }

    public EventGeneratorCPP(ModelElement parent, double tba, String name) {
        super(parent, name);
        double[] values = {1, 2, 3};
        double[] cdf = {0.2, 0.5, 1.0};
        myNumArrivals = new RandomVariable(this, new DEmpiricalRV(values, cdf));
        myTBA = new RandomVariable(this, new ExponentialRV(tba));
        myEventCounter = new Counter(this, "Counts Events");
        myArrivalCounter = new Counter(this, "Counts Arrivals");
        myArrivalGenerator = new EventGenerator(this, new Arrivals(), myTBA, myTBA);
    }
}
```

---

<sup>20</sup>See (Ross, 1997) for more information on the theory of compound Poisson processes.

```

protected class Arrivals implements EventGeneratorActionIfc {
    @Override
    public void generate(EventGenerator generator, JSLEvent event) {
        myEventCounter.increment();
        int n = (int)myNumArrivals.getValue();
        myArrivalCounter.increment(n);
    }
}

```

The `EventGenerator` class allows for the periodic generation of events similar to that achieved by “Create” modules in other simulation languages. This class works in conjunction with the `EventGeneratorActionIfc` interface, which is used to listen and react to the events that are generated by this class. Users of the class can supply an instance of an `EventGeneratorActionIfc` to provide the actions that take place when the event occurs. Alternatively, if no `EventGeneratorActionIfc` is supplied, by default the `generator(JSLEvent event)` method of this class will be called when the event occurs. Thus, sub-classes can simply override this method to provide behavior for when the event occurs. If no instance of an `EventGeneratorActionIfc` instance is supplied and the `generate()` method is not overridden, then the events will still occur; however, no meaningful actions will take place. The key input parameters to the `EventGenerator` include:

**time until the first event** This parameter is specified with an object that implements the `RandomIfc`. It should be used to represent a positive real value that represents the time after time 0.0 for the first event to occur. If this parameter is not supplied, then the first event occurs at time 0.0.

**time between events** This parameter is specified with an object that implements the `RandomIfc`. It should be used to represent a positive real value that represents the time between events. If this parameter is not supplied, then the time between events is positive infinity.

**time until last event** This parameter is specified with an object that implements the `RandomIfc`. It should be used to represent a positive real value that represents the time that the generator should stop generating. When the generator is created, this variable is used to set the ending time of the generator. Each time an event is to be scheduled the ending time is checked. If the time of the next event is past this time, then the generator is turned off and the event will not be scheduled. The default is positive infinity.

**maximum number of events** A value of type long that supplies the maximum number of events to generate. Each time an event is to be scheduled, the maximum number of events is checked. If the maximum has been reached, then the generator is turned off. The default is `Long.MAX_VALUE`. This parameter cannot be `Long.MAX_VALUE` when the time until next always returns a value of 0.0.

**listener** This parameter can be used to supply an instance of `EventGeneratorListener`.

Ifc interface to supply logic to occur when the event occurs.

The most common use case for an `EventGenerator` is very similar to the compound Poisson process example. The `EventGenerator` is setup to run with a time between events until the simulation completes; however, there are a number of other possibilities that are facilitated through various method associated with the `EventGenerator` class. The first possibility is to sub-class the `EventGenerator` to make a custom generator for objects of a specific class. To facilitate this the user need only implement the `generate()` method that is part of the `EventGenerator` class. For example, you could design classes to create customers, parts, trucks, demands, etc.

In addition to customization through sub-classes, there are a number of useful methods that are available for controlling the `EventGenerator`.

**`turnOffGenerator()`** This method allows an `EventGenerator` to be turned off. The next scheduled generation event will not occur. This method will cancel a previously scheduled generation event if one exists. No future events will be scheduled after turning off the generator. Once the generator has been turned off, it cannot be restarted until the next replication.

**`turnOnGenerator(GetValueIfc t)`** If the generator was not started upon initialization at the beginning of a replication, then this method can be used to start the generator. The generator will be started  $t$  time units after the call. If this method is used when the generator is already started it does nothing. If this method is used after the generator is done it does nothing. If this method is used after the generator has been suspended it does nothing. In other words, if the generator is already on, this method does nothing.

**`suspend()`** This method suspends the event generation pattern. The generator is still on, but the generation of events is suspended. The next scheduled generation event is canceled.

**`resume()`** If the generator is suspended then this method causes the event generator to proceed with the event generation pattern by scheduling a new event according to the time between event distribution.

**`isSuspended()`** Checks if the generator is suspended.

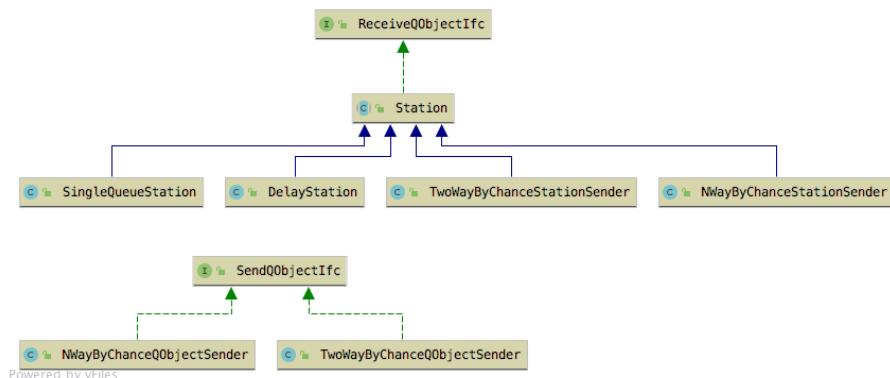
**`isGeneratorDone()`** Checks if the generator has been turned off. The generator can be turned off via the `turnOffGenerator()` method or it may turn off when it has reached its time until last event or if the maximum number of events is reached. As previously noted, once a generator has been turned off, it cannot be turned on again within the same replication.

In considering these methods, a generator can turn itself off (as an action) within or caused by the code within its `generate()` method or in the supplied `EventGeneratorListenerIfc` interface. It might also `suspend()` itself in a similar manner. Of course, a class that has a reference to the generator may also turn it off or suspend it. To resume a suspended event generator, it is necessary to schedule an event whose action invokes the

`resume()` method. Obviously, this can be within a sub-class of `EventGenerator` or within another class that has a reference to the event generator.

### 0.32 The Station Package

This section describes a JSL package that facilitates the modeling of system components that can send or receive entities. Many systems have as a basic component a location where something happens or where processing can occur. A station represents this concept. The goal of this package is to illustrate the development and use of common components that help with the modeling of simple queueing system modeling. First, an overview of the package will be presented and then the operation of important classes discussed. Then, a number of examples will illustrate how to utilize the classes.



**Figure 23:** Major Classes within Station Package

Figure 23 illustrates the primary classes and interfaces within the station package. There are two key interfaces: `SendQ0bjectIfc` and `ReceiveQ0bjectIfc`.

```

public interface SendQ0bjectIfc {

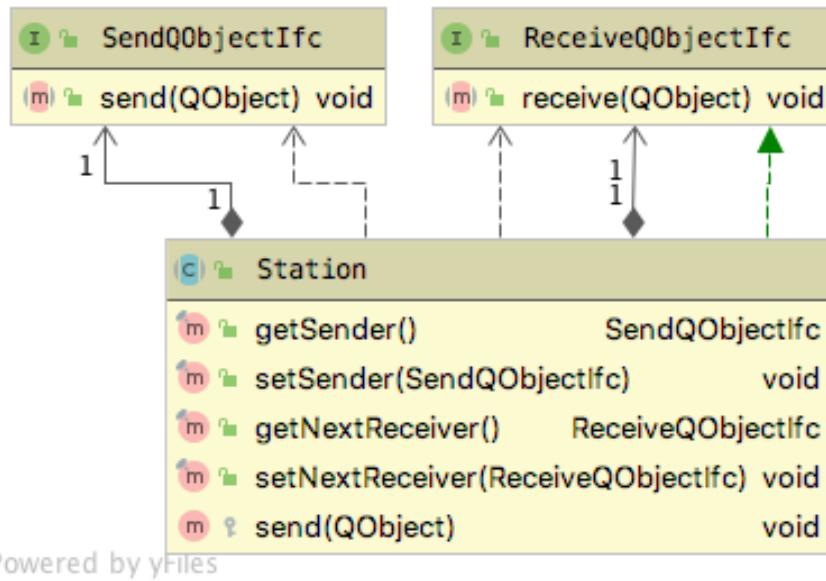
    void send(Q0bject q0bj);
}

public interface ReceiveQ0bjectIfc {

    void receive(Q0bject q0bj);
}
  
```

The interfaces define single methods, `send(Q0bject q0bj)` and `receive(Q0bject q0bj)`, which permit implementors to promise the capability of receiving and sending objects of type `Q0bject`. Figure 24 illustrates the connection between the `Station` class and

`SendQ0bjectIfc` and `ReceiveQ0bjectIfc`. This design permits the development of model elements that know how to send instances of `Q0bject` and how to receive instances of `Q0bject`. A `Station` is an abstract base class that implements the `ReceiveQ0bjectIfc` interface. Sub-classes of `station` will have a `receive()` method that can be called to tell the station that an instance of a `Q0bject` should be received.



Powered by yFiles

**Figure 24:** The Station Class

The following listing shows the implementation of the `station` class. The `station` class may have a reference to an object that implements the `SendQ0bjectIfc` interface and may have an object that implements the `ReceiveQ0bjectIfc` interface. An object that implements the `SendQ0bjectIfc` interface can be supplied to define behavior for sending a received instance of a `Q0bject` onwards within the system. If the `SendQ0bjectIfc` interface attribute is not supplied, then the `ReceiveQ0bjectIfc` object is used to indicate the next location for receipt. The protected method, `send(Q0bject qobj)`, indicates this logic. The UML diagram for the `station` class is provided in Figure 24. Notice that `station` implements the `ReceiveQ0bjectIfc` interface, may have a reference to a `ReceiveQ0bjectIfc` instance, and has a reference to an instance of a `SendQ0bjectIfc` interface.

If the client does not supply either an object that implements the `ReceiveQ0bjectIfc` interface or an object that implements the `SendQ0bjectIfc` interface, then an exception will be thrown. The object that implements the `ReceiveQ0bjectIfc` interface is meant to be a direct connection (i.e. something that can directly receive instances of `Q0bject`). Sometimes, the logic to determine where to send the `Q0bject` is complex. In which case, the user can either sub-class the `station` class or provide an instance of a class that implements the `SendQ0bjectIfc` interface. This allows the sending logic to be delegated to

another class rather than forcing the user to sub-class the `Station` class. This flexibility can be confusing to users of the class.

```
public abstract class Station extends SchedulingElement implements ReceiveQObjectIfc {  
  
    /**  
     * Can be supplied in order to provide logic  
     * to send the QObject to its next receiver  
     */  
    private SendQObjectIfc mySender;  
  
    /** Can be used to directly tell the receiver to receive the departing  
     * QObject  
     *  
     */  
    private ReceiveQObjectIfc myNextReceiver;  
  
    /**  
     *  
     * @param parent the parent model element  
     */  
    public Station(ModelElement parent) {  
        this(parent, null, null);  
    }  
  
    /**  
     *  
     * @param parent the parent model element  
     * @param name a unique name  
     */  
    public Station(ModelElement parent, String name) {  
        this(parent, null, name);  
    }  
  
    /**  
     *  
     * @param parent the parent model element  
     * @param sender can be null, represents something that can send QObjects  
     * @param name a unique name  
     */  
    public Station(ModelElement parent, SendQObjectIfc sender, String name) {  
        super(parent, name);  
        setSender(sender);  
    }  
}
```

```
/**  
 * A Station may or may not have a helper object that implements the  
 * SendQ0bjectIfc interface. If this helper object is supplied it will  
 * be used to send the processed Q0bject to its next location for  
 * processing.  
 * @return the thing that will be used to send the completed Q0bject  
 */  
public final SendQ0bjectIfc getSender()  
{  
    return mySender;  
}  
  
/**  
 * A Station may or may not have a helper object that implements the  
 * SendQ0bjectIfc interface. If this helper object is supplied it will  
 * be used to send the processed Q0bject to its next location for  
 * processing.  
 * @param sender the thing that will be used to send the completed Q0bject  
 */  
public final void setSender(SendQ0bjectIfc sender)  
{  
    mySender = sender;  
}  
  
/**  
 * A Station may or may not have a helper object that implements the  
 * ReceiveQ0bjectIfc interface. If this helper object is supplied and  
 * the SendQ0bjectIfc helper is not supplied, then the object that implements  
 * the ReceiveQ0bjectIfc will be the next receiver for the Q0bject when using  
 * default send() method.  
 * @return the thing that should receive the completed Q0bject, may be null  
 */  
public final ReceiveQ0bjectIfc getNextReceiver()  
{  
    return myNextReceiver;  
}  
  
/**  
 * A Station may or may not have a helper object that implements the  
 * ReceiveQ0bjectIfc interface. If this helper object is supplied and  
 * the SendQ0bjectIfc helper is not supplied, then the object that implements  
 * the ReceiveQ0bjectIfc will be the next receiver for the Q0bject when using  
 * default send() method.  
 * @param receiver the thing that should receive the completed Q0bject, may be null  
 */  
public final void setNextReceiver(ReceiveQ0bjectIfc receiver)  
{  
    myNextReceiver = receiver;  
}
```

```

}

/**
 * A Station may or may not have a helper object that implements the
 * SendQ0bjectIfc interface. If this helper object is supplied it will
 * be used to send the processed Q0bject to its next location for
 * processing.
 *
 * A Station may or may not have a helper object that implements the
 * ReceiveQ0bjectIfc interface. If this helper object is supplied and
 * the SendQ0bjectIfc helper is not supplied, then the object that implements
 * the ReceiveQ0bjectIfc will be the next receiver for the Q0bject
 *
 * If neither helper object is supplied then a runtime exception will
 * occur when trying to use the send() method
 * @param q0bj the completed Q0bject
 */
protected void send(Q0bject q0bj) {
    if (getSender() != null) {
        getSender().send(q0bj);
    } else if (getNextReceiver() != null) {
        getNextReceiver().receive(q0bj);
    } else {
        throw new RuntimeException("No valid sender or receiver");
    }
}

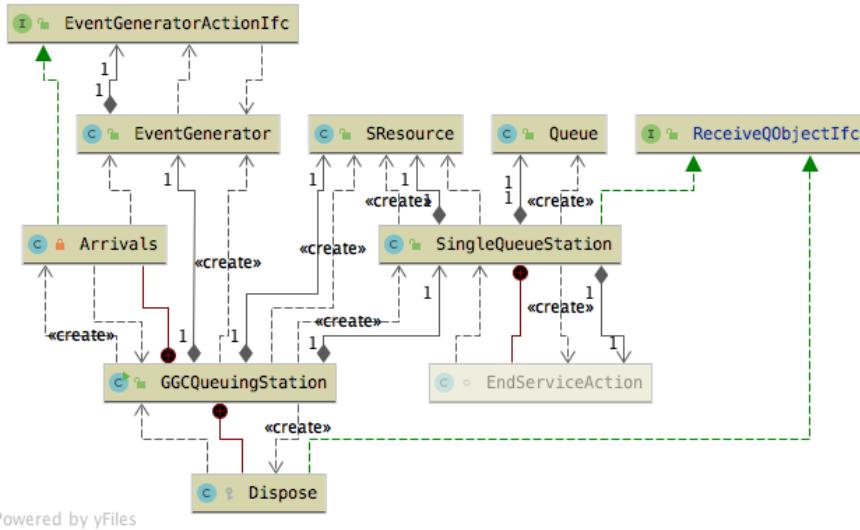
}

```

### 0.32.1 Modeling Simple Queueing Stations

In this section, the station package is used to implement a simple queueing station. Recall the pharmacy model. In that model, we had a Poisson arrival process with customers arriving to a drive through pharmacy window. The service time associated with the pharmacy was exponentially distributed and there was one pharmacist. That situation was labeled as a M/M/1 queue. In this example, we generalize the queueing station to a G/G/c queue. That is, any general (G) arrival process, any general (G) service process, and (c) servers. We will model the arrival process using an `EventGenerator` and model the queueing with a `Queue` class. In addition, we will generalize the concept of the servers by encapsulating them in a class called `SResource` (for simple resource). Finally, the `Queue` and the `SResource` classes will be combined into one class that subclasses `Station`, called `singleQueueStation`. This is illustrated in Figure 25.

In the figure, the `GGCQueueingStation` class uses inner classes for handling the arrivals and for disposing customers that have completed service. In addition, `GGCQueueingSta-`



**Figure 25:** GGC Queueing Station Classes

tion uses an instance of **SingleQueueStation** to process the customers after arrival.

Let's first examine the organization of the **GGCQueueingStation** class and how it works with the **Station** package. Then, we will present how the **SingleQueueStation** class was implemented. As shown in Figure 25 the **GGCQueueingStation** class uses an instance of **EventGenerator**. The event generator's action is called **Arrivals**, which is implemented as an inner class of **GGCQueueingStation**. Secondly, **GGCQueueingStation** uses an instance of the class **Dispose**, which implements the **ReceiveQ0bjectIfc** interface. This instance is used as the receiver after processing by the **SingleQueueStation** class. The class collects some system level statistics.

The following code listing shows the code for the **GGCQueueingStation** class. The constructor takes in instances of **RandomIfc** interfaces to represent the arrival and service processes. The arrival generator and statistical collection model elements are created within the constructor. The third line of the constructor creates the resource that is used by the arriving customers. The fifth line of the constructor creates an instance of a **SingleQueueStation** to handle the customers. Notice that the next line defines the next receiver for the station to be an instance of the class **Dispose**. When the **SingleQueueStation** is done, the **send()** method is called (as previously described) and since a receiver has been attached, the station uses this receiver to receive the **qobject** that was completed. In this case, the **qobject** has statistics collected as it leaves the system. The only other code to note is how the **EventGenerator**'s action is used to send the **qobject** to the **SingleQueueStation**. Within the **Arrivals** class, the **receive()** method of the **SingleQueueStation** instance is called with the created **qobject**. In essence, the **GGCQueueingStation** class simply defines and hooks up the various components needed

to model the situation. The `ReceiveQObjectIfc` and the `Station` concept play an integral role in facilitating how the object instances are connected.

```

public class GGCQueuingStation extends ModelElement {

    private final EventGenerator myArrivalGenerator;
    private final SingleQueueStation mySQS;
    private ResponseVariable mySystemTime;
    private TimeWeighted myNumInSystem;
    private final SResource myServers;
    private final RandomVariable mySTRV;

    public GGCQueuingStation(ModelElement parent, RandomIfc tba, RandomIfc st,
        int numServers) {
        this(parent, tba, st, numServers, null);
    }

    public GGCQueuingStation(ModelElement parent, RandomIfc tba, RandomIfc st,
        int numServers, String name) {
        super(parent, name);
        myArrivalGenerator = new EventGenerator(this, new Arrivals(), tba, tba);
        myServers = new SResource(this, numServers, "Servers");
        mySTRV = new RandomVariable(this, st);
        mySQS = new SingleQueueStation(this, myServers, mySTRV, "Station");
        mySQS.setNextReceiver(new Dispose());
        mySystemTime = new ResponseVariable(this, "System Time");
        myNumInSystem = new TimeWeighted(this, "Num in System");
    }

    private class Arrivals implements EventGeneratorActionIfc {

        @Override
        public void generate(EventGenerator generator, JSLEvent event) {
            myNumInSystem.increment();
            mySQS.receive(new QObject(getTime()));
        }
    }

    protected class Dispose implements ReceiveQObjectIfc {

        @Override
        public void receive(QObject qObj) {
            // collect final statistics
            myNumInSystem.decrement();
        }
    }
}

```

```

    mySystemTime.setValue(getTime() - qObj.getCreateTime());
}

}

```

The last piece of this puzzle is how the `SingleQueueStation` class implements the abstract base class `Station` and becomes a component that can be reused in various models. Recall the logic for handling arrivals and service completions from the pharmacy example. In that code, there was the need to have the customer enter the queue and then check if the pharmacist was available. If the pharmacist was available, then the customer was removed from the queue and the end of service scheduled. In addition, when the service is completed, if the queue is not empty, then the next customer is started into service. The `SingleQueueStation` generalizes this same logic by using an instance of `SResource` to represent the servers.

The following code listing illustrates the critical code for the `SingleQueueStation` class. The `receive()` method (which must be implemented by a sub-class of `Station`), shows the `QObject` being placed in the queue. Then, the resource is checked to see if it is available and then the next customer is served via the call to `serveNext()`. Upon the end of service the `EndServiceAction` is called. Notice that the resource is released, and then the queue is checked. Finally, the `send()` method is used to send the `QObject` to wherever it is intended to go.

```

protected double getServiceTime(QObject customer) {
    double t;
    if (getUseQobjectServiceTimeOption()) {
        Optional<GetValueIfc> valueObject = customer.getValueObject();
        if (valueObject.isPresent()){
            t = valueObject.get().getValue();
        } else {
            throw new IllegalStateException("Attempted to use QObject.getValueObject() when no object was set");
        }
    } else {
        t = getServiceTime().getValue();
    }
    return t;
}

/**
 * Called to determine which waiting QObject will be served next Determines
 * the next customer, seizes the resource, and schedules the end of the
 * service.
 */
protected void serveNext() {
    QObject customer = myWaitingQ.removeNext(); //remove the next customer
}

```

```

myResource.seize();
// schedule end of service
scheduleEvent(myEndServiceAction, getServiceTime(customer), customer);
}

@Override
public void receive(QObject customer) {
    myNS.increment(); // new customer arrived
    myWaitingQ.enqueue(customer); // enqueue the newly arriving customer
    if (isResourceAvailable()) { // server available
        serveNext();
    }
}

class EndServiceAction implements EventActionIfc<QObject> {

    @Override
    public void action(JSLEvent<QObject> event) {
        QObject leavingCustomer = event.getMessage();
        myNS.decrement(); // customer departed
        myResource.release();
        if (isQueueNotEmpty()) { // queue is not empty
            serveNext();
        }
        send(leavingCustomer);
    }
}

```

This is the same `send()` method that was previously discussed. Thus, because the receiver was set to be the instance of `Dispose`, the departing customer will be sent to the `GGCQueueingStation` where the statistics will be collected, as previously noted.

Figure 26 shows the `SResource` class. The `SResource` class models a resource that has a defined capacity, which represents the number of units of the resource that can be in use. The capacity of the resource represents the maximum number of units available for use. For example, if the resource has capacity 3, it may have 2 units busy and 1 unit idle. A resource cannot have more units busy than the capacity. A resource is considered busy when it has 1 or more units busy. A resource is considered idle when all available units are idle. Units of the resource can be seized via the `seize()` methods and released via the `release()` methods. It is an error to attempt to seize more units than are currently available. In addition, it is an error to try to `release()` more units than are currently in use (busy). Statistics on resource utilization and number of busy units is automatically collected.

The following listing illustrates how to build and run an instance of a `GGCQueueingStation` by simulating a M/M/2 queue.

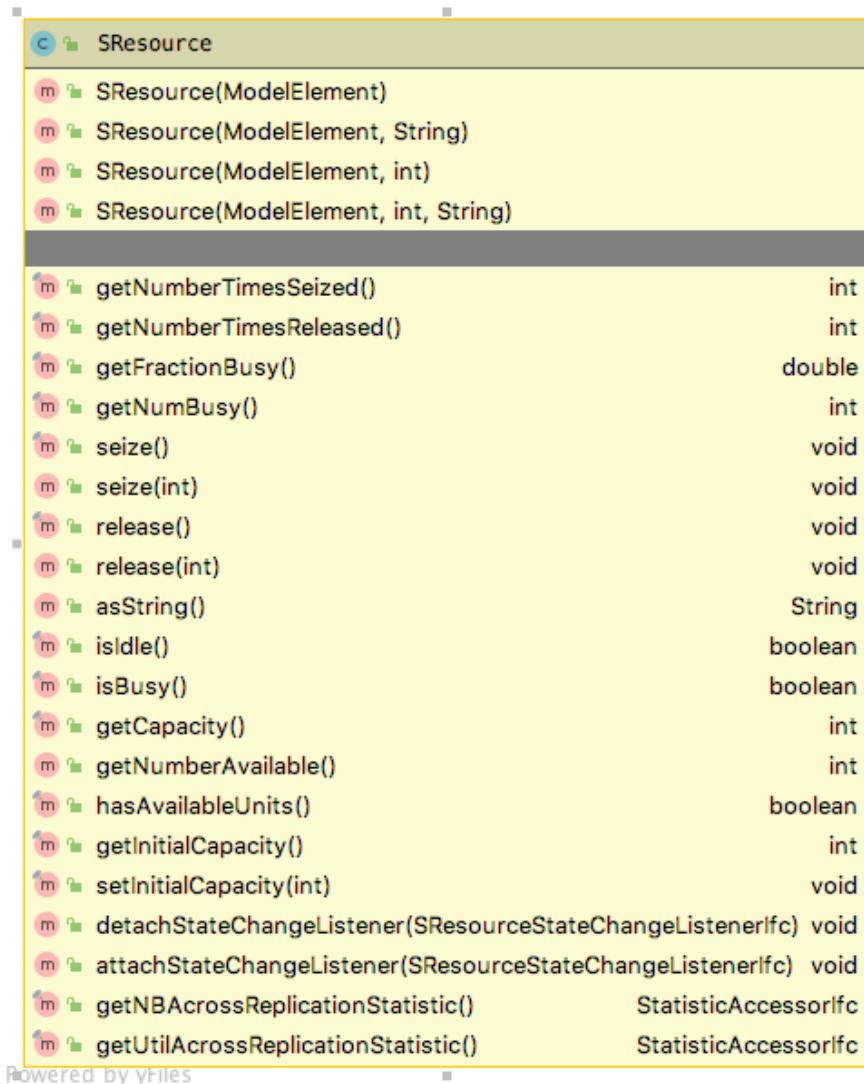


Figure 26: The SResource Class

```

public static void main(String[] args) {
    Simulation sim = new Simulation("M/M/2");
    // get the model
    Model m = sim.getModel();
    // add system to the main model
    ExponentialRV tba = new ExponentialRV(1);
    ExponentialRV st = new ExponentialRV(.8);
    int ns = 2;
    GGCQueueingStation system = new GGCQueueingStation(m, tba, st, ns);
    // set the parameters of the experiment
    sim.setNumberOfReplications(30);
    sim.setLengthOfReplication(20000.0);
    sim.setLengthOfWarmUp(5000.0);
    SimulationReporter r = sim.makeSimulationReporter();
    System.out.println("Simulation started.");
    sim.run();
    System.out.println("Simulation completed.");
    r.printAcrossReplicationSummaryStatistics();
}

```

This is the same as any JSL run. In this case, the `GGCQueueingStation` class used a `SingleQueueStation` to do, essentially, all of the work. Now, imagine many queueing stations organized into a system. Because the `SingleQueueStation` is built as a component that can be reused, it is available for more complicated modeling.

The following code illustrates how easy it is to model a tandem queue using the `SingleQueueStation` class. A tandem queue is a sequence of two queueing stations in series, where the customers departing the first station go on for further processing at the second station. The constructor shows the creation of two stations, where the receiver of the first station is set to the second station (line 9). The receiver for the second station is set to an instance of a class that implements the `ReceiveQ0bjectIfc` interface and collects the total time in the system and the number of customers in the system. This should begin to indicate how complex networks of queueing stations can be formed and simulated.

```

public TandemQueue(ModelElement parent, String name) {
    super(parent, name);
    myTBA = new RandomVariable(this, new ExponentialRV(1.0/1.1));
    myST1 = new RandomVariable(this, new ExponentialRV(0.8));
    myST2 = new RandomVariable(this, new ExponentialRV(0.7));
    myArrivalGenerator = new EventGenerator(this, new Arrivals(), myTBA, myTBA);
    myStation1 = new SingleQueueStation(this, myST1, "Station1");
    myStation2 = new SingleQueueStation(this, myST2, "Station2");
    myStation1.setNextReceiver(myStation2);
}

```

```

myStation2.setNextReceiver(new Dispose());
mySysTime = new ResponseVariable(this, "System Time");
myNumInSystem = new TimeWeighted(this, "NumInSystem");
}

protected class Arrivals implements EventGeneratorActionIfc {

    @Override
    public void generate(EventGenerator generator, JSLEvent event) {
        myNumInSystem.increment();
        myStation1.receive(new QObject(getTime()));
    }

}

protected class Dispose implements ReceiveQobjectIfc {

    @Override
    public void receive(QObject qobj) {
        // collect system time
        mySysTime.setValue(getTime() - qobj.getCreateTime());
        myNumInSystem.decrement();
    }

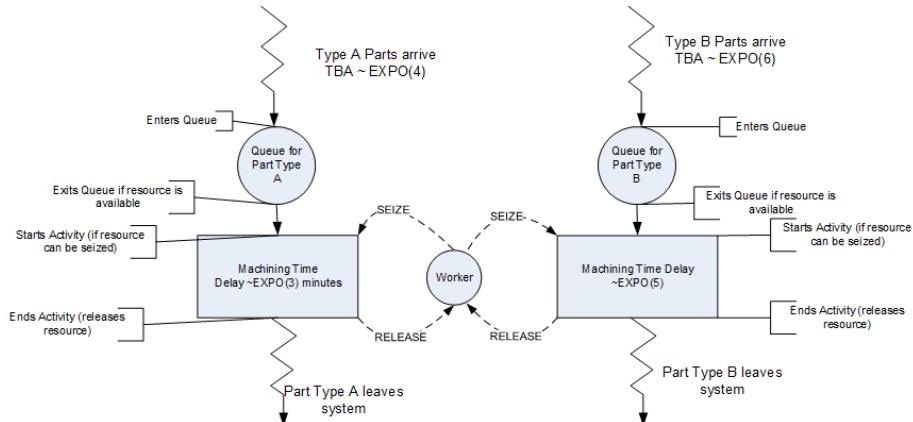
}

```

## 0.33 Sharing a Resource

In the previous queueing situations, at each station there was one resource that was used during the service operation. It is often the case that a resource can be shared across multiple activities. For example, suppose a worker is assigned to attend two machining processes. The resource is said to be shared.

Figure 27 illustrates this notion of sharing a resource between two activities. Notice that in the figure, there are two queues, one each for the two different types of parts. If there had been only a single queue to hold both parts, the modeling is very similar to the single queue station modeling that has already been presented. With only one queue, we would need to use a queue discipline to order the parts in the queue and to schedule the end of service according to the appropriate service time distribution for the given part. In the case of two queues, we need to determine how the resource will select the queue in addition to the queue discipline. For the sake of simplicity, let us assume that the queue discipline is first in first out (FIFO). Then, in the case of two queues, we need a mechanism or rule to determine which queue will be processed first, when the resource becomes available.



**Figure 27:** A Resource Shared Between Two Activities

How many events are there for the situation in Figure 27? From the strict definition of an event, there are six events in this situation. There are the two arrival events for each type of part. There are two different begin service events and two different end service events. From an implementation perspective, we could model this situation with only two events. We can have a single event that handles both types of arrivals and a single end service event that handles the departure of either type of part. The reason we only need one end service event is because the begin service event logic can be incorporated in the end of service event because whenever a service completes a waiting part will start into service. How many events should you use in your modeling? You should implement the events in a way that facilitates your perspective and your modeling. The trade-off is that when the event logic is consolidated, the logic may be more complicated.

The following listing presents the constructor for the shared queue example. Notice that two instances of the `Queue` and `EventGenerator` classes are used. The two event generators are used to implement the arrival processes for the two types of parts. The two queues are used to hold the two types of parts. An instance of `SResource` is used to represent the workers that are shared between the two activities. The activity times are represented by two `RandomVariable` instances. A `TimeWeighted` instance and a `ResponseVariable` instance are used to collect statistics on the number of parts in the system and the time spent in the system, respectively.

```

public class SharedResource extends SchedulingElement {

    private final EventGenerator myTypeAGenerator;
    private final EventGenerator myTypeBGenerator;
    private final TimeWeighted myNumInSystem;
    private final ResponseVariable mySystemTime;
    private final SResource myServers;
  
```

```

private final Queue<QObject> myTypeAWaitingQ;
private final Queue<QObject> myTypeBWaitingQ;
private final EndServiceEventAction myEndServiceEventAction;
private final RandomVariable myServiceRVTypeA;
private final RandomVariable myServiceRVTypeB;

public SharedResource(ModelElement parent, int numServers, RVariableIfc tbaA,
                      RVariableIfc tbaB, RVariableIfc stA, RVariableIfc stB, String name) {
    super(parent, name);
    myTypeAGenerator = new EventGenerator(this, new TypeAArrivals(), tbaA, tbaA);
    myTypeBGenerator = new EventGenerator(this, new TypeBArrivals(), tbaB, tbaB);
    myServiceRVTypeA = new RandomVariable(this, stA, "Service RV A");
    myServiceRVTypeB = new RandomVariable(this, stB, "Service RV B");
    myServers = new SResource(this, numServers, "Servers");
    myTypeAWaitingQ = new Queue<>(this, getName() + "_QA");
    myTypeBWaitingQ = new Queue<>(this, getName() + "_QB");
    mySystemTime = new ResponseVariable(this, "System Time");
    myNumInSystem = new TimeWeighted(this, "Num in System");
    myEndServiceEventAction = new EndServiceEventAction();
}

```

The following listing presents the arrivals of the parts via the generator listeners and the beginning of service. In line 5, the number of parts is incremented. Then, in line 6, the arriving part is represented with a new `Qobject`, which is immediately enqueued. The if statement starting at line 7, checks if the resource has available units and if so, begins serving the next part. In the `serveNext()` method, the two queues are checked. Notice that the queue for type A parts is checked first. If the queue is not empty, then the part is removed and started into service using the service time distribution for type A parts. If there are no type A parts, the queue for type B parts is checked in a similar fashion. Because the type A queue is checked first, it will have priority over the type B part queue.

```

private class TypeAArrivals implements EventGeneratorActionIfc {

    @Override
    public void generate(EventGenerator generator, JSLEvent event) {
        myNumInSystem.increment();
        myTypeAWaitingQ.enqueue(new QObject(getTime()));
        if (myServers.hasAvailableUnits()) { // server available
            serveNext();
        }
    }
}

```

```

private class TypeBArrivals implements EventGeneratorActionIfc {

    @Override
    public void generate(EventGenerator generator, JSLEvent event) {
        myNumInSystem.increment();
        myTypeBWaitingQ.enqueue(new QObject(getTime()));
        if (myServers.hasAvailableUnits()) { // server available
            serveNext();
        }
    }

    private void serveNext() {
        //logic to choose next from queues
        // if both have waiting parts, assume part type A has priority
        if (myTypeAWaitingQ.isNotEmpty()) {
            QObject partA = myTypeAWaitingQ.removeNext(); //remove the next customer
            myServers.seize();
            // schedule end of service
            scheduleEvent(myEndServiceEventAction, myServiceRVTypeA, partA);
        } else if (myTypeBWaitingQ.isNotEmpty()) {
            QObject partB = myTypeBWaitingQ.removeNext(); //remove the next customer
            myServers.seize();
            // schedule end of service
            scheduleEvent(myEndServiceEventAction, myServiceRVTypeB, partB);
        }
    }
}

```

The following code listing presents the end of service event logic for the shared resource example. The end of service logic is very similar to logic that we have seen for the departure from a queueing station. In the end of service action, the part that is leaving is passed to the outer class for statistical collection in the `departingSystem()` method. The resource is released and the queues are checked to see if a part is waiting. Note that both queues are checked, such that if either queue has a waiting part, the logic for serving the next part is invoked.

```

private boolean checkQueues() {
    return (myTypeAWaitingQ.isNotEmpty() || myTypeBWaitingQ.isNotEmpty());
}

private class EndServiceEventAction implements EventActionIfc<QObject> {

```

```

@Override
public void action(JSLEvent<QObject> event) {
    QObject leavingPart = event.getMessage();
    myServers.release();
    if (checkQueues()) { // queue is not empty
        serveNext();
    }
    departSystem(leavingPart);
}

private void departSystem(QObject leavingPart) {
    mySystemTime.setValue(getTime() - leavingPart.getCreateTime());
    myNumInSystem.decrement(); // part left system
}

```

Finally, the following code shows how to setup and run the example. The system is configured with the time between arrivals for type A parts and type B parts having exponential distributions with means of 4 and 6 time units, respectively. The service time distributions are also exponential with means of 3 and 5, respectively for type A and B parts. The capacity of the resource is set at 2 workers.

```

public static void main(String[] args) {
    Simulation sim = new Simulation("Shared Resource Example");
    // get the model
    Model m = sim.getModel();
    // add to the main model
    RVariableIfc tbaA = new ExponentialRV(4.0);
    RVariableIfc tbaB = new ExponentialRV(6.0);
    RVariableIfc stA = new ExponentialRV(3.0);
    RVariableIfc stB = new ExponentialRV(5.0);
    SharedResource sr = new SharedResource(m, 2, tbaA, tbaB, stA, stB, "SR");
    // set the parameters of the experiment
    sim.setNumberOfReplications(30);
    sim.setLengthOfReplication(20000.0);
    sim.setLengthOfWarmUp(5000.0);
    SimulationReporter r = sim.makeSimulationReporter();
    System.out.println("Simulation started.");
    sim.run();
    System.out.println("Simulation completed.");
    r.printAcrossReplicationSummaryStatistics();
}

```

The results indicate that the type B parts wait significantly longer on average than the

type A parts. It should be apparent that with some additional design that the shared resource example could be generalized into a class that could be used in other simulation models similar to how the single station queue has been generalized. This is one of the important advantages of an object-oriented approach to simulation modeling.

---

```

Number of Replications: 30
Length of Warm up period: 5000.0
Length of Replications: 20000.0
-----
-----
Response Variables
-----
-----
Name          Average      Std. Dev.    Count
-----
-----
Servers_Util   0.792550    0.015226    30.00
0000
Servers_#Busy Units 1.585100    0.030453    30.00
0000
SR_QA : Number In Q 0.589924    0.044712    30.00
0000
SR_QA : Time In Q   2.361027    0.155425    30.00
0000
SR_QB : Number In Q 1.948387    0.363318    30.00
0000
SR_QB : Time In Q   11.664922   2.087974    30.00
0000
System Time       9.891819    0.908085    30.00
0000
Num in System     4.123410    0.413465    30.00
0000
-----
```

---

## 0.34 Complex System Example

This section presents a more complex system that illustrates how to model entities that flow through a system and how to coordinate the flow. The example will reuse the single queue station modeling of previous examples; however, the entities (objects that flow in the system) require synchronization.

Suppose production orders for tie-dye T-shirts arrive to a production facility according to a Poisson process with a mean rate of 4 per hour. There are two basic psychedelic de-

signs involving either red or blue dye. For some reason the blue shirts are a little more popular than the red shirts so that when an order arrives about 70% of the time it is for the blue dye designs. In addition, there are two different package sizes for the shirts, 3 and 5 units. There is a 25% chance that the order will be for a package size of 5 and a 75% chance that the order will be for a package size of 3. Each of the shirts must be *individually* hand made to the customer's order design specifications. The time to produce a shirt (of either color) is uniformly distributed within the range of 3 to 5 minutes. There is currently one worker who is setup to make either shirt. When an order arrives to the facility, its type (red or blue) is determined and the pack size is determined. Then, the appropriate number of white (un-dyed) shirts are sent to the shirt makers with a note pinned to the shirt indicating the customer order, its basic design, and the pack size for the order. Meanwhile, the paperwork for the order is processed by a worker and a customized packaging letter and box is prepared to hold the order. It takes the paperwork worker between 8 to 10 minutes to make the box and print a custom thank you note. After the packaging is made the paperwork waits prior to final inspection for the shirts associated with the order. After the shirts are combined with the packaging, they are inspected by a packaging worker which is distributed according to a triangular distribution with a minimum of 5 minutes, a most likely value of 10 minutes, and a maximum value of 15 minutes. Finally, the boxed customer order is sent to shipping.

### 0.34.1 Conceptualizing the Model

Before proceeding you might want to jot down your answers to the following modeling recipe questions and then compare how you are doing with respect to what is presented in this section. The modeling recipe questions are:

- What is the system? What information is known by the system?
- What are the required performance measures?
- What are the entities? What information must be recorded or remembered for each entity? How are entities introduced into the system?
- What are the resources that are used by the entities? Which entities use which resources and how?
- What are the process flows? Sketch the process or make an activity flow diagram
- Develop pseudo-code for the situation
- Implement the model

The entities can be conceptualized as the arriving orders. Since the shirts are processed individually, they should also be considered entities. In addition, the type of order (red or blue) and the size of the order (3 or 5) must be tracked. Since the type of the order and the size of the order are properties of the order, attributes can be used to model this information. The resources are the two shirt makers, the paperwork worker, and the packager. The flow is described in the scenario statement: orders arrive, shirts made, meanwhile packaging is made. Then, orders are assembled, inspected, and finally shipped. It should be clear that an EventGenerator, setup to generate Poisson

arrivals can create the orders, but if shirts are entities, how should they be modeled and created? After this, there will be two types of entities in the model, the orders (paperwork) and the shirts. The shirts can be made and meanwhile the paperwork for the order can be processed. When the shirts for an order are made, they need to be combined together and associated with the order.

The activity diagram for this situation is given in Figure 28. After the order is created, the process separates into the order making process and the shirt making process. Notice that the orders and shirts must be synchronized together after each of these processes.

### 0.34.2 Implementing the Model

If it was not for the coordination between the orders (paperwork/packaging) and the shirts in this system, the modeling would be a straightforward application of concepts that have already been presented. The processing of the shirts, the paperwork, and the final packaging can all be modeled with instances of the `SingleQueueStation` class, where the shirts go to one instance, the paperwork goes to another instance, and the combined final order goes to the third instance. Thus, if it was not for the fact that shirts must be combined into an order and the order has to be combined with its paperwork before packaging, the classes within the station package could handle this modeling. In fact, the constructor for the implementation takes exactly this approach as illustrated in the following code listing.

```
public TieDyeTShirts(ModelElement parent, String name) {
    super(parent, name);
    myTBOrders = new RandomVariable(this, new ExponentialRV(15));
    myOrderGenerator = new EventGenerator(this, new OrderArrivals(),
        myTBOrders, myTBOrders);
    DEmpiricalRV type = new DEmpiricalRV(new double[]{1.0, 2.0}, new double[] {0.7, 1.0});
    DEmpiricalRV size = new DEmpiricalRV(new double[]{3.0, 5.0}, new double[] {0.75, 1.0});
    myOrderSize = new RandomVariable(this, size);
    myOrderType = new RandomVariable(this, type);
    myShirtMakingTime = new RandomVariable(this, new UniformRV(3, 5));
    myPaperWorkTime = new RandomVariable(this, new UniformRV(8, 10));
    myPackagingTime = new RandomVariable(this, new TriangularRV(5, 10, 15));
    myShirtMakers = new SResource(this, 1, "ShirtMakers_R");
    myPackager = new SResource(this, 1, "Packager_R");
    myShirtMakingStation = new SingleQueueStation(this, myShirtMakers,
        myShirtMakingTime, "Shirt_Station");
    myWorker = new SResource(this, 1, "PW-Worker");
    myPWStation = new SingleQueueStation(this, myWorker,
        myPaperWorkTime, "PW_Station");
    myPackagingStation = new SingleQueueStation(this, myPackager,
        myPackagingTime, "Packing_Station");
```

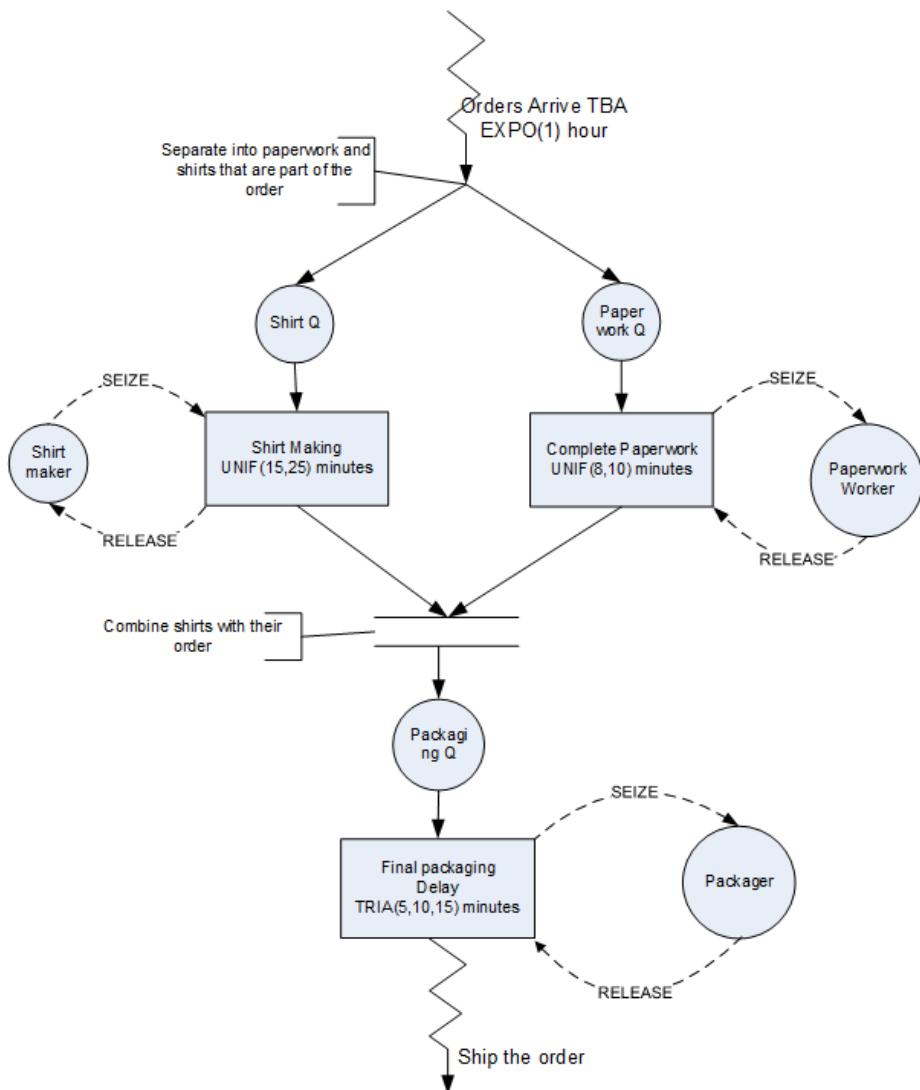


Figure 28: Activity Diagram for Tie Dye T-Shirts System

```
// need to set senders/receivers
myShirtMakingStation.setNextReceiver(new AfterShirtMaking());
myPWStation.setNextReceiver(new AfterPaperWork());
myPackagingStation.setNextReceiver(new Dispose());
mySystemTime = new ResponseVariable(this, "System Time");
myNumInSystem = new TimeWeighted(this, "Num in System");
}
```

In the constructor, lines 3 and 4, shows the specification for the `EventGenerator`. Then, `DEmpiricalRV` random variables are setup to model the order size and the order type, where 1 represents blue shirts and 2 represents red shirts. Lines 10-12 show the modeling of the shirt making time, paperwork time, and packaging time all using instances of the `RandomVariable` class. In lines 13-15 the workers are modeled with instances of the `SResource` class, and in lines 16-21, the `SingleQueueStation` class is used to model the use of the resources and the activities for shirt making, paperwork, and packaging. All that is left is connecting the stations together, which is accomplished in lines 23-25 by setting the receivers for the stations. The ‘magic’ of modeling the order coordination happens in the receivers and how orders are modeled.

Before exploring that implementation, we can explore how orders enter and leave the system. The following listing presents how the orders are generated and how the orders leave the system.

```
private class OrderArrivals implements EventGeneratorActionIfc {

    @Override
    public void generate(EventGenerator generator, JSLEvent event) {
        myNumInSystem.increment();
        Order order = new Order();
        List<Order.Shirt> shirts = order.getShirts();

        for (Order.Shirt shirt : shirts) {
            myShirtMakingStation.receive(shirt);
        }
        myPWStation.receive(order.getPaperWork());
    }
}

protected class Dispose implements ReceiveQ0bjectIfc {

    @Override
    public void receive(Q0bject q0bj) {
        // collect final statistics
    }
}
```

```

    myNumInSystem.decrement();
    mySystemTime.setValue(getTime() - qObj.getCreateTime());
    Order o = (Order) qObj;
    o.dispose();
}

}

```

An implementation of an `EventGeneratorListenerIfc` interface called `OrderArrivals` is provided to the `EventGenerator`. In line 5, the number of orders in the system is incremented. Then, in line 6, a new order is made. The shirts associated with the orders are then retrieved (line 7) and then sent to the shirt making station (lines 9-11). Finally, line 12 retrieves the paperwork from the order (`order.getPaperWork()`) and sends it to the paperwork station. The implementation of the logic for orders to leave the system is implemented in similar disposal logic as we have previously seen (starting at line 18). First, the number of orders is decremented and the time that an order spent in the system collected. Then, the order is disposed.

The modeling of the synchronization of the shirts and the paperwork comes down to the following fact: when all the shirts are completed and the paperwork is completed, then the order is completed and can be sent to the packaging station. If the paperwork activity finishes before all the shirts are completed, the order will be completed when the last shirt is done. If all the shirts are completed before the paperwork is completed, then the order waits until the paperwork is done. Thus, when a shirt associated with an order is completed, we can check if the paper work is done and if all shirts are done, the order can proceed. When the paperwork is completed, we need to check if all shirts are done and if so the order can proceed. Both shirt completion and paperwork completion are events and these events are modeled within the `SingleQueueStation` class. However, right after the events occur the `SingleQueueStation` class uses its attached `QobjectReceiverIfc` instance. The logic for checking for order completion can be added to the receivers. This logic is presented in the following listing.

```

protected class AfterShirtMaking implements ReceiveQObjectIfc {

    @Override
    public void receive(QObject qObj) {
        Order.Shirt shirt = (Order.Shirt) qObj;
        shirt.setDoneFlag();
    }
}

protected class AfterPaperWork implements ReceiveQObjectIfc {

    @Override

```

```

public void receive(QObject qObj) {
    Order.PaperWork pw = (Order.PaperWork) qObj;
    pw.setDoneFlag();
}

}

```

Because of how the Order class is implemented, this logic is not particularly interesting. As shown in the code listing, all that occurs is to indicate the shirt is completed (line 6) and that the paperwork is completed (line 16). The Order class is implemented such that it is notified whenever a shirt is completed and when the paperwork is completed. If during this notification process, the entire order is completed, the order will be sent to the packaging station. This is the logic that we will explore next.

Because orders may need to wait, we are going to model them by sub-classing from the `QObject` class. This allows the use of the `Queue` class. In addition, shirts and paperwork are also entities that wait. So, they will also be modeled as sub-classes of the `QObject` class. Orders have a size and a type. In addition, orders contain shirts and paperwork. The following listing shows the constructor and instance variables for the `Order` class.

```

private class Order extends QObject {

    private int myType;
    private int mySize;
    private PaperWork myPaperWork;
    private List<Shirt> myShirts;
    private int myNumCompleted;
    private boolean myPaperWorkDone;

    public Order(double creationTime, String name) {
        super(creationTime, name);
        myNumCompleted = 0;
        myPaperWorkDone = false;
        myType = (int) myOrderType.getValue();
        mySize = (int) myOrderSize.getValue();
        myShirts = new ArrayList<>();
        for (int i = 1; i <= mySize; i++) {
            myShirts.add(new Shirt());
        }
        myPaperWork = new PaperWork();
    }
}

```

The `Order` class is a private inner class of the `TieDyeTShirts` class. This allows orders access to all the instance variables and methods of the `TieDyeTShirts` class. It is declared

private since its usage is only within the `TieDyeTShirts` class. `Order` extends `QObject` and has instance variables to represent the type and size of the orders (lines 3 and 4). Lines 5 and 6 represent the associations between the order and its shirts (held in a list) and its paperwork. In the constructor body, the type and size are set from the random variables declared in the `TieDyeTShirts` class. In addition, a list holding the shirts is filled and the paperwork is created. The number of completed shirts is noted as zero and the fact that the paperwork is not completed is saved in an attribute.

The following listing illustrates the key methods associated with modeling the behavior of the orders.

```
public boolean isComplete() {
    return ((areShirtsDone()) && (isPaperWorkDone()));
}

public boolean areShirtsDone() {
    return (myNumCompleted == mySize);
}

public boolean isPaperWorkDone() {
    return (myPaperWorkDone);
}

public int getNumShirtsCompleted() {
    return myNumCompleted;
}

private void shirtCompleted() {
    if (areShirtsDone()) {
        throw new IllegalStateException("The order already has all its shirts.");
    }
    // okay not complete, need to add shirt
    myNumCompleted = myNumCompleted + 1;
    if (isComplete()) {
        TieDyeTShirts.this.orderCompleted(this);
    }
}

private void paperWorkCompleted() {
    if (isPaperWorkDone()) {
        throw new IllegalStateException("The order already has paperwork.");
    }
    myPaperWorkDone = true;
    if (isComplete()) {
        TieDyeTShirts.this.orderCompleted(this);
    }
}
```

```
    }  
}
```

The three methods, `isCompleted()`, `areShirtsDone()`, and `isPaperWorkDone()` all check on the status of the order. The two methods `shirtCompleted()` and `paperWorkCompleted()` are used by shirts and paperwork to update the order's state. The `shirtCompleted()` method increments the number of shirts completed and if the order is completed, the outer class, `TieDyeTShirts` is notified by calling its `orderCompleted()` method. In addition, the `paperWorkCompleted()` method does the same thing when it is completed. These two methods are called by shirts and paperwork.

The following code presents the Shirt and PaperWork classes. The Shirt and PaperWork classes are inner classes of the Order class.

```
protected class Shirt extends QObject {  
  
    protected boolean myDoneFlag = false;  
  
    public Shirt() {  
        this(getTime());  
    }  
  
    public Shirt(double creationTime) {  
        this(creationTime, null);  
    }  
  
    public Shirt(double creationTime, String name) {  
        super(creationTime, name);  
    }  
  
    public Order getOrder() {  
        return Order.this;  
    }  
  
    public void setDoneFlag() {  
        if (myDoneFlag == true) {  
            throw new IllegalStateException("The shirt is already done.");  
        }  
        myDoneFlag = true;  
        Order.this.shirtCompleted();  
    }  
  
    public boolean isCompleted() {  
        return myDoneFlag;  
    }  
}
```

```

}

protected class PaperWork extends Shirt {

    @Override
    public void setDoneFlag() {
        if (myDoneFlag == true) {
            throw new IllegalStateException("The paperwork is already done.");
        }
        myDoneFlag = true;
        Order.this.paperWorkCompleted();
    }
}

```

The key methods are the `setDoneFlag()` methods in both classes. Notice that when a shirt is completed the call `Order.this.shirtCompleted()` occurs. Similar logic occurs within the `Paperwork` class. This is the notification that they are completed so that the `Order` class is notified when they are completed.

### 0.34.3 Model Results

The following table presents the results of the simulation. From the utilization of the shirt making resource it is clear that more than one shirt maker is necessary.

**Table 16:** Across Replication Statistics for Tie-Dye T-Shirts Example, Number of Replications 50

Response Name	$\bar{x}$	$s$
ShirtMakers_R:Util	0.934393	0.009719
ShirtMakers_R:BusyUnits	0.934393	0.009719
Packager_R:Util	0.667315	0.006774
Packager_R:BusyUnits	0.667315	0.006774
Shirt_Station:Q:NumInQ	27.041158	7.192692
Shirt_Station:Q:TimeInQ	115.486332	29.686151
Shirt_Station:NS	27.975551	7.200011
PW-Worker:Util	0.600392	0.006394
PW-Worker:BusyUnits	0.600392	0.006394
PW_Station:Q:NumInQ	0.452418	0.020481
PW_Station:Q:TimeInQ	6.780610	0.256060
PW_Station:NS	1.052810	0.025685
Packing_Station:Q:NumInQ	0.015327	0.000827
Packing_Station:Q:TimeInQ	0.229741	0.011867
Packing_Station:NS	0.682642	0.007135

Response Name	$\bar{x}$	$s$
System Time	134.366279	29.657064
Num in System	8.982821	2.067345

## 0.35 Summary

In this chapter, you have learned a few fundamental JSL model elements that facilitate the modeling of simple queueing situations. The classes and interfaces covered included:

**EventGenerator** Used to cause a sequence of events to occur according to a pattern of time.

**EventGeneratorActionIfc** Used to define a method that is called when an EventGenerator's event is invoked

**Station** An abstract base class that facilitates the receiving and send of QObjects within a model.

**ReceiveQObjectIfc** An interface that defines a receive(QObjectIfc) method for receiving instances of the QQObject class.

**SendQObjectIfc** An interface that defines a send(QObjectIfc) method for sending instances of the QQObject class.

**SingleQueueStation** A concrete implementation of the abstract Station class that allows received instances of QObjects to wait in a Queue in order to use a resource (SResource).

**SResource** A simple resource class that models units of capacity that can be seized and released.

The **EventGenerator** class provides a reusable component that facilitates the modeling of arrival processes. This was illustrated by modeling a compound Poisson process as well as illustrating how to create objects of user-defined classes (e.g. Order, Shirt, etc.). The station package with the JSL allows the construction of networks of stations where processing may occur. The **SingleQueueStation** class is one example of the kind of station that can be built. By using the **ReceiveQObjectIfc** and the **SendQObjectIfc** a wide variety of additional components can be built while taking advantage of the sending and receiving paradigm that is often common in simulation modeling. We also introduced the **SResource** class for modeling simple resource situations where a resource has a capacity of units that can be used. How to model a simple shared resource was also illustrated. With just these components you could model a wide-variety of situations and build additional reusable components.

As you perform your modeling, I strongly encourage you to plan your simulation model carefully (on paper) prior to entering it into a computer model. If you just sit down and try to program at the computer, the effort can result in confusing spaghetti code. You

should have a plan for defining your classes and use a naming convention for things that you use in the model. I advise that you use some basic object-oriented program development methods such as the Unified Modeling Language (UML) diagrams and Class Responsibility Collaboration (CRC) cards. In addition, you should list out the logic of your model in some sort of pseudo-code. You should treat simulation model development like a programming effort.

The next couple of chapters will build upon the modeling foundations learned in this chapter.



# Analyzing Simulation Output

## LEARNING OBJECTIVES

- To be able to recognize the different types of statistical quantities used within and produced by simulation models
- To be able to analyze finite horizon simulations via the method of replications
- To be able to analyze infinite horizon simulations via the method of batch means and the method of replication-deletion
- To be able to compare simulation alternatives and make valid decisions based on the statistical output of a simulation

Because the inputs to the simulation are random, the outputs from the simulation are also random. You can think of a simulation model as a function that maps inputs to outputs. This chapter presents the statistical analysis of the outputs from simulation models.

In addition, a number of issues that are related to the proper execution of simulation experiments are presented. For example, the simulation outputs are dependent upon the input random variables, input parameters, and the initial conditions of the model. Initial conditions refer to the starting conditions for the model, i.e. whether or not the system starts “empty and idle”. The effect of initial conditions on steady state simulations will be discussed in this chapter.

Input parameters are related to the controllable and uncontrollable factors associated with the system. For a simulation model, *all* input parameters are controllable; however, in the system being modeled we typically have control over only a limited set of parameters. Thus, in simulation you have the unique ability to control the random inputs into your model. This chapter will discuss how to take advantage of controlling the random inputs.

Input parameters can be further classified as decision variables. That is, those parameters of interest that you want to change in order to test model configurations for decision-making. The structure of the model itself may be considered a decision variable when you are trying to optimize the performance of the system. When you change

the input parameters for the simulation model and then execute the simulation, you are simulating a different design alternative.

This chapter describes how to analyze the output from a single design alternative and how to analyze the results of multiple design alternatives. To begin the discussion you need to build an understanding of the types of statistical quantities that may be produced by a simulation experiment.

## 0.36 Types of Statistical Variables

A simulation experiment occurs when the modeler sets the input parameters to the model and executes the simulation. This causes events to occur and the simulation model to evolve over time. During the execution of the simulation, the behavior of the system is observed and various statistical quantities computed. When the simulation reaches its termination point, the statistical quantities are summarized in the form of output reports.

A simulation experiment may be for a single replication of the model or may have multiple replications. A *replication* is the generation of one sample path which represents the evolution of the system from its initial conditions to its ending conditions. If you have multiple replications within an experiment, each replication represents a different sample path, starting from the same initial conditions and being driven by the same input parameter settings. Because the randomness within the simulation can be controlled, the underlying random numbers used within each replication of the simulation can be made to be independent. Thus, as the name implies, each replication is an independently generated “repeat” of the simulation.

Within a single sample path (replication), the statistical behavior of the model can be observed.

**Definition 0.1**(Within Replication Statistic). The statistical quantities collected during a replication are called *within replication statistics*.

**Definition 0.2**(Across Replication Statistic). The statistical quantities collected across replications are called *across replication statistics*. Across replication statistics are collected based on the observation of the final values of within replication statistics.

*Within replication statistics* are collected based on the observation of the sample path and include observations on entities, state changes, etc. that occur during a sample path execution. The observations used to form within replication statistics are not likely to be independent and identically distributed. Since across replication statistics are formed from the final values of within replication statistics, one observation per replication is available. Since each replication is considered independent, the observations that form the sample for across replication statistics are likely to be independent and identically distributed. The statistical properties of within and across

replication statistics are inherently different and require different methods of analysis. Of the two, within replication statistics are the more challenging from a statistical standpoint.

For within replication statistical collection there are two primary types of observations: *tally* and *time-persistent*. Tally data represent a sequence of equally weighted data values that do not persist over time. This type of data is associated with the duration or interval of time that an object is in a particular state or how often the object is in a particular state. As such it is observed by marking (tallying) the time that the object enters the state and the time that the object exits the state. Once the state change takes place, the observation is over (it is gone, it does not persist, etc.). If we did not observe the state change, then we would have missed the observation. The time spent in queue, the count of the number of customers served, whether or not a particular customer waited longer than 10 minutes are all examples of tally data.

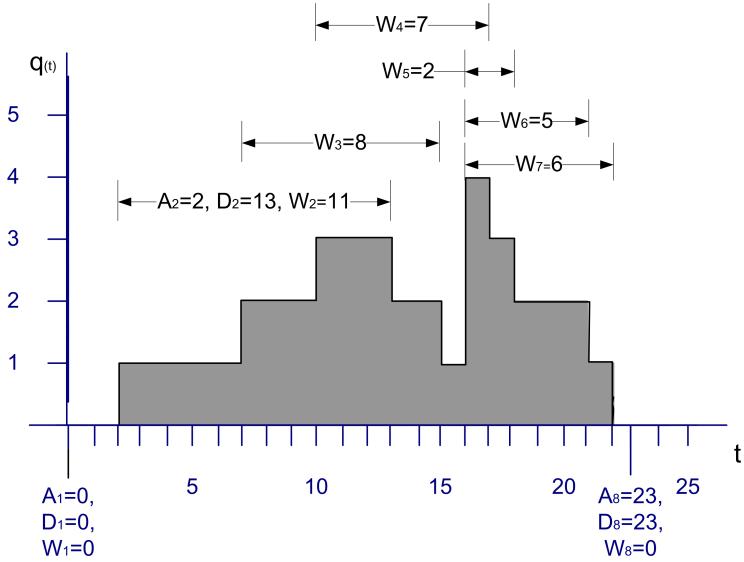
Time-persistent observations represent a sequence of values that persist over some specified amount of time with that value being weighted by the amount of time over which the value persists. These observations are directly associated with the values of the state variables within the model. The value of a time-persistent observation persists in time. For example, the number of customers in the system is a common state variable. If we want to collect the average number of customers in the system *over time*, then this will be a time-persistent statistic. While the value of the number of customers in the system changes at discrete points in time, it holds (or persists with) that value over a duration of time. This is why it is called a time-persistent variable.

Figure 29 illustrates a single sample path for the number of customers in a queue over a period of time. From this sample path, events and subsequent statistical quantities can be observed.

- Let  $A_i$   $i = 1 \dots n$  represent the time that the  $i^{th}$  customer enters the queue
- Let  $D_i$   $i = 1 \dots n$  represent the time that the  $i^{th}$  customer exits the queue
- Let  $W_i = D_i - A_i$   $i = 1 \dots n$  represent the time that the  $i^{th}$  customer spends in the queue

Thus,  $W_i$   $i = 1 \dots n$  represents the sequence of wait times for the queue, each of which can be individually observed and tallied. This is tally type data because the customer enters a state (the queued state) at time  $A_i$  and exits the state at time  $D_i$ . When the customer exits the queue at time  $D_i$ , the waiting time in queue,  $W_i = D_i - A_i$  can be observed or tallied.  $W_i$  is only observable at the instant  $D_i$ . This makes  $W_i$  tally based data and, once observed, its value never changes again with respect to time. Tally data is most often associated with an entity that is moving through states that are implied by the simulation model. An observation becomes available each time the entity enters and subsequently exits the state.

With tally data it is natural to compute the sample average as a measure of the central tendency of the data. Assume that you can observe  $n$  customers entering and existing the queue, then the average waiting time across the  $n$  customers is given by:



**Figure 29:** Sample Path for Tally and Time-Persistent Data

$$\bar{W}(n) = \frac{1}{n} \sum_{i=1}^n W_i$$

Many other statistical quantities, such as the minimum, maximum, and sample variance, etc. can also be computed from these observations. Unfortunately, within replication data is often (if not always) correlated with respect to time. In other words, within replication observations like,  $W_i, i = 1 \dots n$ , are not statistically independent. In fact, they are likely to also not be identically distributed. Both of these issues will be discussed when the analysis of infinite horizon or steady state simulation models is presented.

The other type of statistical variable encountered within a replication is based on time-persistent observations. Let  $q(t), t_0 < t \leq t_n$  be the number of customers in the queue at time  $t$ . Note that  $q(t) \in \{0, 1, 2, \dots\}$ . As illustrated in Figure 29,  $q(t)$  is a function of time (a step function in this particular case). That is, for a given (realized) sample path,  $q(t)$  is a function that returns the number of customers in the queue at time  $t$ .

The mean value theorem of calculus for integrals states that given a function,  $f(\cdot)$ , continuous on an interval  $[a, b]$ , there exists a constant,  $c$ , such that

$$\int_a^b f(x)dx = f(c)(b - a)$$

The value,  $f(c)$ , is called the mean value of the function. A similar function can be

defined for  $q(t)$ . In simulation, this function is called the time-average:

$$\bar{L}_q(n) = \frac{1}{t_n - t_0} \int_{t_0}^{t_n} q(t) dt$$

This function represents the average with respect to time of the given state variable. This type of statistical variable is called time-persistent because  $q(t)$  is a function of time (i.e. it persists over time).

In the particular case where  $q(t)$  represents the number of customers in the queue,  $q(t)$  will take on constant values during intervals of time corresponding to when the queue has a certain number of customers. Let  $q(t) = q_k$  for  $t_{k-1} \leq t \leq t_k$  and define  $v_k = t_k - t_{k-1}$  then the time-average number in queue can be rewritten as follows:

$$\begin{aligned}\bar{L}_q(n) &= \frac{1}{t_n - t_0} \int_{t_0}^{t_n} q(t) dt \\ &= \frac{1}{t_n - t_0} \sum_{k=1}^n q_k (t_k - t_{k-1}) \\ &= \frac{\sum_{k=1}^n q_k v_k}{t_n - t_0} \\ &= \frac{\sum_{k=1}^n q_k v_k}{\sum_{k=1}^n v_k}\end{aligned}$$

Note that  $q_k(t_k - t_{k-1})$  is the area under  $q(t)$  over the interval  $t_{k-1} \leq t \leq t_k$  and

$$t_n - t_0 = \sum_{k=1}^n v_k = (t_1 - t_0) + (t_2 - t_1) + \cdots + (t_{n-1} - t_{n-2}) + (t_n - t_{n-1})$$

is the total time over which the variable is observed. Thus, the time average is simply the area under the curve divided by the amount of time over which the curve is observed. From this equation, it should be noted that each value of  $q_k$  is weighted by the length of time that the variable has the value. This is why the time average is often called the time-weighted average. If  $v_k = 1$ , then the time average is the same as the sample average.

With time-persistent data, you often want to estimate the percentage of time that the variable takes on a particular value. Let  $T_i$  denote the *total* time during  $t_0 < t \leq t_n$  that the queue had  $q(t) = i$  customers. To compute  $T_i$ , you sum all the rectangles corresponding to  $q(t) = i$ , in the sample path. Because  $q(t) \in \{0, 1, 2, \dots\}$  there are an infinite number of possible value for  $q(t)$  in this example; however, within a finite sample path you can only observe a finite number of the possible values. The ratio of  $T_i$  to  $T = t_n - t_0$  can be used to estimate the percentage of time the queue had  $i$  customers. That is, define  $\hat{p}_i = T_i/T$  as an estimate of the proportion of time that the queue had  $i$  customers during the interval of observation.

Let's look at an example. Consider Figure 29, which shows the change in queue length over a simulated period of 25 time units.

1. Compute the time average number in queue over the interval of time from 0 to 25.
2. Compute the percentage of time that the queue had  $\{0, 1, 2, 3, 4\}$  customers

Since the queue length is a time-persistent variable, the time average queue length can be computed as:

$$\begin{aligned}\bar{L}_q &= \frac{0(2-0) + 1(7-2) + 2(10-7) + 3(13-10) + 2(15-13) + 1(16-15)}{25} \\ &\quad + \frac{4(17-16) + 3(18-17) + 2(21-18) + 1(22-21) + 0(25-22)}{25} \\ &= \frac{39}{25} = 1.56\end{aligned}$$

To estimate the percentage of time that the queue had  $\{0, 1, 2, 3, 4\}$  customers, the values of  $v_k = t_k - t_{k-1}$  need to be summed for whenever  $q(t) \in \{0, 1, 2, 3, 4\}$ . This results in the following:

$$\begin{aligned}\hat{p}_0 &= \frac{T_0}{T} = \frac{(2-0) + (25-22)}{25} = \frac{2+3}{25} = \frac{5}{25} = 0.2 \\ \hat{p}_1 &= \frac{T_1}{T} = \frac{(7-2) + (16-15) + (22-21)}{25} = \frac{5+1+1}{25} = \frac{7}{25} = 0.28 \\ \hat{p}_2 &= \frac{T_2}{T} = \frac{3+2+3}{25} = \frac{8}{25} = 0.28 \\ \hat{p}_3 &= \frac{T_3}{T} = \frac{3+1}{25} = \frac{4}{25} = 0.16 \\ \hat{p}_4 &= \frac{T_4}{T} = \frac{1}{25} = \frac{1}{25} = 0.04\end{aligned}$$

Notice that the sum of the  $\hat{p}_i$  adds to one. To compute the average waiting time in the queue, use the supplied values for each waiting time.

$$\bar{W}(8) = \frac{\sum_{i=1}^n W_i}{n} = \frac{0 + 11 + 8 + 7 + 2 + 5 + 6 + 0}{8} = \frac{39}{8} = 4.875$$

Notice that there were two customers, one at time 1.0 and another at time 23.0 that had waiting times of zero. The state graph did not move up or down at those times. Each unit increment in the queue length is equivalent to a new customer entering (and staying in) the queue. On the other hand, each unit decrement of the queue length signifies a departure of a customer from the queue. If you assume a first-in, first-out

(FIFO) queue discipline, the waiting times of the six customers that entered the queue (and had to wait) are shown in the figure.

Now that we understand the type of data that occurs within a replication, we need to develop an understanding for the types of simulation situations that require specialized statistical analysis. The next section introduces this important topic.

## 0.37 Types of Simulation With Respect To Output Analysis

When modeling a system, specific measurement goals for the simulation responses are often required. The goals, coupled with how the system operates, will determine how you execute and analyze the simulation experiments. In planning the experimental analysis, it is useful to think of simulations as consisting of two main categories related to the period of time over which a decision needs to be made:

**Finite horizon** In a finite-horizon simulation, a well defined ending time or ending condition can be specified which clearly defines the end of the simulation. Finite horizon simulations are often called *terminating* simulations, since there are clear terminating conditions.

**Infinite horizon** In an infinite horizon simulation, there is no well defined ending time or condition. The planning period is over the life of the system, which from a conceptual standpoint lasts forever. Infinite horizon simulations are often called *steady state* simulations because in an infinite horizon simulation you are often interested in the long-term or steady state behavior of the system.

For a finite horizon simulation, an event or condition associated with the system is present which indicates the end of each simulation replication. This event can be specified in advance or its time of occurrence can be a random variable. If it is specified in advance, it is often because you do not want information past that point in time (e.g. a 3 month planning horizon). It might be a random variable in the case of the system stopping when a condition is met. For example, an ending condition may be specified to stop the simulation when there are no entities left to process. Finite horizon simulations are very common since most planning processes are finite. A few example systems involving a finite horizon include:

- Bank: bank doors open at 9am and close at 5pm
- Military battle: simulate until force strength reaches a critical value
- Filling a customer order: suppose a new contract is accepted to produce 100 products, you might simulate the production of the 100 products to see the cost, delivery time, etc.

For a finite horizon simulation, each replication represents a sample path of the model for one instance of the finite horizon. The length of the replication corresponds to the

finite horizon of interest. For example, in modeling a bank that opens at 9 am and closes at 5 pm, the length of the replication would be 8 hours.

In contrast to a finite horizon simulation, an infinite horizon simulation has no natural ending point. Of course, when you actually simulate an infinite horizon situation, a finite replication length must be specified. Hopefully, the replication length will be long enough to satisfy the goal of observing long run performance. Examples of infinite horizon simulations include:

- A factory where you are interested in measuring the steady state throughput
- A hospital emergency room which is open 24 hours a day, 7 days of week
- A telecommunications system which is always operational

Infinite horizon simulations are often tied to systems that operate continuously and for which the long-run or steady state behavior needs to be estimated.

Because infinite horizon simulations often model situations where the system is always operational, they often involve the modeling of non-stationary processes. In such situations, care must be taken in defining what is meant by long-run or steady state behavior. For example, in an emergency room that is open 24 hours a day, 365 days per year, the arrival pattern to such a system probably depends on time. Thus, the output associated with the system is also non-stationary. The concept of steady state implies that the system has been running so long that the system's behavior (in the form of performance measures) no longer depends on time; however, in the case of the emergency room since the inputs depend on time so do the outputs. In such cases it is often possible to find a period of time or cycle over which the non-stationary behavior repeats. For example, the arrival pattern to the emergency room may depend on the day of the week, such that every Monday has the same characteristics, every Tuesday has the same characteristics, and so on for each day of the week. Thus, on a weekly basis the non-stationary behavior repeats. You can then define your performance measure of interest based on the appropriate non-stationary cycle of the system. For example, you can define  $Y$  as the expected waiting time of patients *per week*. This random variable may have performance that can be described as long-term. In others, the long-run weekly performance of the system may be stationary. This type of simulation has been termed steady state cyclical parameter estimation within (Law, 2007).

Of the two types of simulations, finite horizon simulations are easier to analyze. Luckily they are the more typical type of simulation found in practice. In fact, when you think that you are faced with an infinite horizon simulation, you should very carefully evaluate the goals of your study to see if they can just as well be met with a finite planning horizon. The analysis of both of these types of simulations will be discussed in this chapter through examples.

## 0.38 Analysis of Finite Horizon Simulations

This section illustrates how tally-based and time-persistent statistics are collected within a replication and how statistics are collected across replications. Finite horizon simulations can be analyzed by traditional statistical methodologies that assume a random sample, i.e. independent and identically distributed random variables. A simulation experiment is the collection of experimental design points (specific input parameter values) over which the behavior of the model is observed. For a particular design point, you may want to repeat the execution of the simulation multiple times to form a sample at that design point. To get a random sample, you execute the simulation starting from the same initial conditions and ensure that the random numbers used within each replication are independent. Each replication must also be terminated by the same conditions. It is very important to understand that independence is achieved across replications, i.e. the replications are independent. The data *within* a replication may or may not be independent.

The method of *independent replications* is used to analyze finite horizon simulations. Suppose that  $n$  replications of a simulation are available where each replication is terminated by some event  $E$  and begun with the same initial conditions. Let  $Y_{rj}$  be the  $j^{th}$  observation on replication  $r$  for  $j = 1, 2, \dots, m_r$  where  $m_r$  is the number of observations in the  $r^{th}$  replication, and  $r = 1, 2, \dots, n$ , and define the sample average for each replication to be:

$$\bar{Y}_r = \frac{1}{m_r} \sum_{j=1}^{m_r} Y_{rj}$$

If the data are time-based then,

$$\bar{Y}_r = \frac{1}{T_E} \int_0^{T_E} Y_r(t) dt$$

$\bar{Y}_r$  is the sample average based on the observation within the  $r^{th}$  replication. It is a random variable that can be observed at the end of each replication, therefore,  $\bar{Y}_r$  for  $r = 1, 2, \dots, n$  forms a random sample. Thus, standard statistical analysis of the random sample can be performed.

To make this concrete, suppose that you are examining a bank that opens with no customers at 9 am and closes its doors at 5 pm to prevent further customers from entering. Let,  $W_{rj}, j = 1, \dots, m_r$ , represents the sequence of waiting times for the customers that entered the bank between 9 am and 5 pm on day (replication)  $r$  where  $m_r$  is the number of customers who were served between 9 am and 5 pm on day  $r$ . For simplicity, ignore the customers who entered before 5 pm but did not get served until after 5 pm. Let  $N_r(t)$  be the number of customers in the system at time  $t$  for day (replication)  $r$ . Suppose that you are interested in the mean daily customer waiting time and the mean number of customers in the bank on any given 9 am to 5 pm day, i.e. you are

interested in  $E[W_r]$  and  $E[N_r]$  for any given day. At the end of each replication, the following can be computed:

$$\bar{W}_r = \frac{1}{m_r} \sum_{j=1}^{m_r} W_{rj}$$

$$\bar{N}_r = \frac{1}{8} \int_0^8 N_r(t) dt$$

At the end of all replications, random samples:  $\bar{W}_r$  and  $\bar{N}_r$  are available from which sample averages, standard deviations, confidence intervals, etc. can be computed. Both of these samples are based on observations of within replication data.

Both  $\bar{W}_r$  and  $\bar{N}_r$  for  $r = 1, 2, \dots, n$  are averages of many observations within the replication. Sometimes, there may only be one observation based on the entire replication. For example, suppose that you are interested in the probability that someone is still in the bank when the doors close at 5 pm, i.e. you are interested in  $\theta = Pr\{N(t = 5pm) > 0\}$ . In order to estimate this probability, an indicator variable can be defined within the simulation and observed each time the condition was met or not. For this situation, an indicator variable,  $I_r$ , for each replication can be defined as follows:

$$I_r = \begin{cases} 1 & N(t = 5pm) > 0 \\ 0 & N(t = 5pm) \leq 0 \end{cases}$$

Therefore, at the end of the replication, the simulation must tabulate whether or not there are customers in the bank and record the value of this indicator variable. Since this happens only once per replication, a random sample of the  $I_r$  for  $r = 1, 2, \dots, n$  will be available after all replications have been executed. We can use the observations of the indicator variable to estimate the desired probability.

Since the analysis of the system will be based on a random sample, the key design criteria for the experiment will be the required number of replications. In other words, you need to determine the sample size.

Because confidence intervals may form the basis for decision making, you can use the confidence interval half-width in determining the sample size. For example, in estimating  $E[W_r]$  for the bank example, you might want to be 95% confident that you have estimated the true waiting time to within  $\pm 2$  minutes.

There are three related methods that are commonly used for determining the sample size for this situation:

- an iterative method based on the t-distribution,
- an approximate method based on the normal distribution, and
- the half-width ratio method

Each of these methods assumes that the observations in the sample are independent and identically distributed from a normal distribution. In addition, the methods also assume that the pilot replications are representative of the population under study. When the data are not normally distributed, you must rely on the central limit theorem to get approximate results. The assumption of normality is typically justified when across replication statistics are based on within replication averages.

Since the first two methods have previously been presented, this chapter will discuss the half-width ratio method because it is often seen in practice.

### 0.38.1 Determining the Number of Replications

If you make a pilot run of  $n_0$  replications you can use the half-width from the pilot run to determine how many replications you need to have to be close to a desired half-width bound in the full experiment. This is called the *half-width ratio method*.

Let  $h_0$  be the initial value for the half-width from the pilot run of  $n_0$  replications.

$$h_0 = t_{\alpha/2, n_0 - 1} \frac{s_0}{\sqrt{n_0}} \quad (4)$$

Solving for  $n_0$  yields:

$$n_0 = t_{\alpha/2, n_0 - 1}^2 \frac{s_0^2}{h_0^2} (\#eq : n_0) \quad (5)$$

Similarly for any  $n$ , we have:

$$n = t_{\alpha/2, n - 1}^2 \frac{s^2}{h^2} \quad (6)$$

Taking the ratio of  $n_0$  to  $n$  (equations @ref(eq:n\_0) and (6)) and assuming that  $t_{\alpha/2, n - 1}$  is approximately equal to  $t_{\alpha/2, n_0 - 1}$  and  $s^2$  is approximately equal to  $s_0^2$ , yields,

$$n \cong n_0 \frac{h_0^2}{h^2} = n_0 \left( \frac{h_0}{h} \right)^2 \quad (7)$$

Equation ((7)) is the half-width ratio equation.

In the case of an indicator variable such as,  $I_r$ , which was suggested for use in estimating the probability that there are customers in the bank after 5 pm, the sampled observations are clearly not normally distributed. In this case, since you are estimating a proportion, you can use the sample size determination techniques for estimating proportions previously described.

Now, let's look at an example. Suppose a pilot run of a simulation model estimated that the average waiting time for a customer during the day was 11.485 minutes based on an initial sample size of 15 replications with a 95% confidence interval half-width

of 1.04. Using the three sample size determination techniques, recommend a sample size to be 95% confident that you are within  $\pm 0.10$  of the true mean waiting time in the queue.

First we will do the half-width ratio method. We have that  $h_0 = 1.04$ ,  $n_0 = 15$ , and  $h = 0.1$ , thus:

$$n \cong n_0 \left( \frac{h_0}{h} \right)^2 = 15 \left( \frac{1.04}{0.1} \right)^2 = 1622.4 \cong 1623$$

To estimate a sample size based on the normal approximation method, we need to have the estimate of the initial sample standard deviation. Unfortunately, this is not directly reported, but it can be computed using Equation ((4)). Rearranging Equation ((4)) to solve for  $s_0$ , yields:

$$s_0 = \frac{h_0 \sqrt{n_0}}{t_{\alpha/2, n_0 - 1}}$$

Since we have a 95% confidence interval with  $n_0 = 15$ , we have that  $t_{0.025, 14} = 2.145$ , which yields,

$$s_0 = \frac{h_0 \sqrt{n_0}}{t_{\alpha/2, n_0 - 1}} = \frac{1.04 \sqrt{15}}{2.145} = 1.87781$$

Now, we can use the normal approximation method. By using  $h$  as the desired bound  $E$  and  $z_{0.025} = 1.96$ , we have,

$$n \geq \left( \frac{z_{\alpha/2} s}{E} \right)^2 = \left( \frac{1.96 \times 1.87781}{0.1} \right)^2 = 1354.54 \approx 1355$$

The final method is to use the iterative method based on the following equation:

$$h = t_{\alpha/2, n-1} \frac{s}{\sqrt{n}} \leq E$$

This can be accomplished easily within a spreadsheet yielding  $n = 1349$ .

The three methods resulted in following recommendations:

- an iterative method based on the t-distribution,  $n = 1349$
- an approximate method based on the normal distribution,  $n = 1355$
- the half-width ratio method,  $n = 1623$

As noted in the discussion, the half-width ratio method recommended the largest number of replications.

	A	B	C	D	E	F
1						
2	<b>alpha</b>	0.05				
3	<b>S</b>	1.87781				
4	<b>bound</b>	0.1				
5	<b>n</b>	1348.799				
6	<b>alpha/2</b>	0.025				
7	<b>t-alpha/2</b>	1.961727				
8	<b>half-width</b>	0.100304				
9	<b>difference</b>	0.000304				
10						
11						
12						
13						
14						
15						
16						

Figure 30: Iterative Method for Sample Size via Goal Seek

### 0.39 Finite Horizon Example

This section presents a fictitious system involving the production of rings. The example illustrates how to collect tally based statistics, time based statistics, and statistics that can only be collected at the end of a replication. The analysis of a finite horizon simulation will be illustrated. In addition, the system also represents another example of how to use the station package.

Every morning the sales force at LOTR Makers, Inc. makes a number of confirmation calls to customers who have previously been visited by the sales force. They have tracked the success rate of their confirmation calls over time and have determined that the chance of success varies from day to day. They have modeled the probability of success for a given day as a beta random variable with parameters  $\alpha_1 = 5$  and  $\alpha_2 = 1.5$  so that the mean success rate is about 77%. They always make 100 calls every morning. Each sales call will or will not result in an order for a pair of magical rings for that day. Thus, the number of pairs of rings to produce every day is a binomial random variable, with  $p$  determined by the success rate for the day and  $n = 100$  representing the total number of calls made. Note that  $p$  is random in this modeling.

The sales force is large enough and the time to make the confirmation calls small enough so as to be able to complete all the calls before releasing a production run for the day. In essence, ring production does not start until all the orders have been confirmed, but the actual number of ring pairs produced every day is unknown until the sales call confirmation process is completed. The time to make the calls is

negligible when compared to the overall production time.

Besides being magical, one ring is smaller than the other ring so that the smaller ring must fit snuggly inside the larger ring. The pair of rings is produced by a master ring maker and takes uniformly between 5 to 15 minutes. The rings are then scrutinized by an inspector with the time (in minutes) being distributed according to a triangular distribution with parameters (2, 4, 7) for the minimum, the mode, and the maximum. The inspection determines whether the smaller ring is too big or too small when fit inside the bigger outer ring. The inside diameter of the bigger ring,  $D_b$ , is normally distributed with a mean of 1.5 cm and a standard deviation of 0.002. The outside diameter of the smaller ring,  $D_s$ , is normally distributed with a mean of 1.49 and a standard deviation of 0.005. If  $D_s > D_b$ , then the smaller ring will not fit in the bigger ring; however, if  $D_b - D_s > tol$ , then the rings are considered too loose. The tolerance is currently set at 0.02 cm.

If there are no problems with the rings, the rings are sent to a packer for custom packaging for shipment. A time study of the packaging time indicates that it is distributed according to a log-normal distribution with a mean of 7 minutes and a standard deviation of 1 minute. If the inspection shows that there is a problem with the pair of rings they are sent to a rework craftsman. The minimum time that it takes to rework the pair of rings has been determined to be 5 minutes plus some random time that is distributed according to a Weibull distribution with a scale parameter of 15 and a shape parameter of 5. After the rework is completed, the pair of rings is sent to packaging.

Currently, the company runs two shifts of 480 minutes each. Time after the end of the second shift is considered overtime. Management is interested in investigating the following:

- The daily production time.
- The probability of overtime.
- The average number of pairs of rings in both the ring making process and the ring inspection process.
- The average time that it takes for a pair of rings to go through both the ring making process and the ring inspection process. In addition, a 95% confidence interval for the mean time to complete these processes to within  $\pm 20$  minutes is desired.

### 0.39.1 Conceptualizing the Model

Now let's proceed with the modeling of this situation. We start with answering the basic model building questions.

*What is the system? What information is known by the system?*

The system is the LOTR Makers, Inc. sales calls and ring production processes. The system starts every day with the initiation of sales calls and ends when the last pair of rings produced for the day is shipped. The system knows the following:

- Sales call success probability distribution:  $p \sim \text{beta}(\alpha_1 = 5, \alpha_2 = 1.5)$
- Number of calls to be made every morning:  $n = 100$
- Distribution of time to make the pair of rings:  $U(5, 15)$
- Distributions associated with the big and small ring diameters:  $N(\mu = 1.5, \sigma = 0.002)$  and  $N(\mu = 1.49, \sigma = 0.005)$ , respectively
- Distribution of ring-inspection time: triangular(2,4,7)
- Distribution of packaging time: lognormal( $\mu_l = 7, \sigma_l = 1$ )
- Distribution of rework time, 5 + Weibull(scale =15, shape =3)
- Length of a shift: 480 minutes

*What are the entities? What information must be recorded for each entity?*

Possible entities are the sales calls and the production job (pair of rings) for every successful sales call. For every pair of rings, the diameters must be known.

*What are the resources that are used by the entities?*

The sales calls do not use any resources. The production job uses a master craftsman, an inspector, and a packager. It might also use a rework craftsman.

*What are the activities? What are the processes? What are the events associated with the processes and activities? Write out or draw sketches of the process.*

There are two processes: sales order and production. An outline of the sales order process should look like this:

1. Start the day.
2. Determine the likelihood of calls being successful.
3. Make the calls.
4. Determine the total number of successful calls.
5. Start the production jobs.

Notice that the sales order process takes zero time and that it occurs at the beginning of each day. Thus, there appears to be an event that occurs at time 0.0, that determines the number of production jobs and sends them to production. This type of situation is best modeled using the initialize() method to make the orders to be placed into production. From the problem statement, the number of production jobs is a binomial random variable with  $n = 100$  and  $p \sim \text{BETA}(\alpha_1 = 5, \alpha_2 = 1.5)$ .

The following listing shows the constructor for this model and the initialize() method.

```
public LOTR(ModelElement parent, String name) {
    super(parent, name);
    mySalesCallProb = new RandomVariable(this, new BetaRV(5.0, 1.5));
```

```

myMakeRingTimeRV = new RandomVariable(this, new UniformRV(5, 15));
mySmallRingODRV = new RandomVariable(this, new NormalRV(1.49, 0.005 * 0.005));
myBigRingIDRV = new RandomVariable(this, new NormalRV(1.5, 0.002 * 0.002));
myInspectTimeRV = new RandomVariable(this, new TriangularRV(2, 4, 7));
myPackingTimeRV = new RandomVariable(this, new LognormalRV(7, 1));
myReworkTimeRV = new RandomVariable(this, new ShiftedRV(5.0, new WeibullRV(3, 15)));
myRingMakingStation = new SingleQueueStation(this, myMakeRingTimeRV,
                                              "RingMakingStation");
myInspectionStation = new SingleQueueStation(this, myInspectTimeRV,
                                              "InspectStation");
myReworkStation = new SingleQueueStation(this, myReworkTimeRV,
                                           "ReworkStation");
myPackagingStation = new SingleQueueStation(this, myPackingTimeRV,
                                              "PackingStation");
myRingMakingStation.setNextReceiver(myInspectionStation);
myInspectionStation.setNextReceiver(new AfterInspection());
myReworkStation.setNextReceiver(myPackagingStation);
myPackagingStation.setNextReceiver(new Dispose());
mySystemTime = new ResponseVariable(this, "System Time");
myNumInSystem = new TimeWeighted(this, "Num in System");
myNumCompleted = new Counter(this, "Num Completed");
myProbTooBig = new ResponseVariable(this, "Prob too Big");
myProbTooSmall = new ResponseVariable(this, "Prob too Small");
myProbOT = new ResponseVariable(this, "Prob of Over Time");
myEndTime = new ResponseVariable(this, "Time to Make Orders");
myNumInRMandInspection = new TimeWeighted(this, "Num in RM and Inspection");
myTimeInRMandInspection = new ResponseVariable(this, "Time in RM and Inspection");
}

@Override
protected void initialize() {
    super.initialize();
    double p = mySalesCallProb.getValue();
    int n = JSLRandom.rBinomial(p, myNumDailyCalls);
    for (int i = 0; i < n; i++) {
        myRingMakingStation.receive(new RingOrder());
        myNumInSystem.increment();
        myNumInRMandInspection.increment();
    }
}
}

```

The constructor makes the random variables that model the number of successful calls and the probability that a call is successful. Then, the initialize() method, which is automatically called at the beginning of a replication (essentially at time 0.0), uses the random variables to first determine the probability of a successful call and then deter-

mines the number of successful calls. Finally, a for-loop is used to make the orders (new RingOrder()) and send them into production at the ring making station. Also, the number of orders in the system and the number of orders at the ring making and inspection stations is incremented.

The following code listing illustrates the modeling of the orders within the system.

```
private class RingOrder extends QObject {

    private double myBigRingID;
    private double mySmallRingOuterD;
    private double myGap;
    private boolean myNeedsReworkFlag = false;
    private boolean myTooBigFlag = false;
    private boolean myTooSmallFlag = false;

    public RingOrder() {
        this(getTime(), null);
    }

    public RingOrder(double creationTime, String name) {
        super(creationTime, name);
        myBigRingID = myBigRingIDRV.getValue();
        mySmallRingOuterD = mySmallRingODRV.getValue();
        myGap = myBigRingID - mySmallRingOuterD;
        if (mySmallRingOuterD > myBigRingID) {
            myTooBigFlag = true;
            myNeedsReworkFlag = true;
        } else if (myGap > myRingTol) {
            myTooSmallFlag = true;
            myNeedsReworkFlag = true;
        }
    }
}
```

A RingOrder represents an order for a pair of rings. RingOrder is an inner class of LOTR. This was done so that the RingOrder has easy access to the random variables defined within the LOTR class. The outer ring's inner diameter and the inner ring's outer diameter are modeled using attributes for the order. The values for these attributes are set using the random variables that were defined as instance variables of the LOTR class and instantiated within the LOTR class's constructor. The size of the gap and whether or not the ring needs rework is also computed. The condition of the ring in terms of whether the gap is too big or too small is specified. Since the RingOrder class extends the QObject class it has the ability to be held by instances of the Queue class. Once the order for the rings is made it is passed into production.

An outline of the production process should be something like this:

1. Make the rings (determine sizes).
2. Inspect the rings.
3. If rings do not pass inspection, perform rework
4. Package rings and ship.

Notice that in the production of the rings there are a number of activities that take place during which various resources are used. This situation is very similar to the Tie-Dye T-Shirt example in that the rings move from ring making to inspection (possibly rework) and finally to packaging. Each of these areas can be modeled using the SingleQueueStation class. The events associated with this situation include arrival to ring making, departure from ring making, arrival to inspection, departure from inspection, arrival to rework, departure from rework, arrival to packaging and departure from packaging. Since the departure from an upstream station also represents an arrival to the downstream station, the number of events needed to model this situation can be consolidated. As mentioned, the SingleQueueStation can be used to model this situation by connecting the stations together. The construction of the stations and their connection is illustrated in the constructor for the LOTR system. Notice that the setNextReceiver() methods are used to connect the ring making station to the inspection station and to have the rework station send work to the packaging station.

One conceptually challenging aspect of this model is the fact that the rings will probabilistically go to the rework station or the packaging station. To model this situation, an inner class called AfterInspection was designed that implements the ReceiveQObjectIfc interface. An instance of this class is provided as the receiver for the inspection station. Thus, after the inspection station is done, the order for the ring (in the form of a QObject) will be sent to this logic.

In the following listing, lines 4 and 5 finish out the collection of the number of orders in the ring making and inspection stations and the collection of the time spent within those stations. Then, starting in line 6, the order is checked if it needs rework and if so, the rework station is told to receive it; otherwise, the packaging station is told to receive it.

```
protected class AfterInspection implements ReceiveQObjectIfc {
    @Override
    public void receive(QObject qObj) {
        myNumInRManInspection.decrement();
        myTimeInRManInspection.setValue(getTime() - qObj.getCreateTime());
        RingOrder order = (RingOrder) qObj;
        if (order.myNeedsReworkFlag) {
            myReworkStation.receive(order);
        } else {
            myPackagingStation.receive(order);
        }
    }
}
```

```

        }
    }

}

protected class Dispose implements ReceiveQ0bjectIfc {
    @Override
    public void receive(Q0bject q0bj) {
        // collect final statistics
        RingOrder order = (RingOrder) q0bj;
        myNumInSystem.decrement();
        mySystemTime.setValue(getTime() - order.getCreateTime());
        myNumCompleted.increment();
        myProbTooBig.setValue(order.myTooBigFlag);
        myProbTooSmall.setValue(order.myTooSmallFlag);
    }
}

@Override
protected void replicationEnded() {
    super.replicationEnded();
    myProbOT.setValue(getTime() > myOTLimit);
    myEndTime.setValue(getTime());
}

```

Similar to previous examples, the Dispose inner class collects statistics at the system level and on the orders as they depart the system. Besides the collection of the number orders in the ring making and inspection stations, management also desired the collection of the probability of over time and the time that the production run will be completed. The collection of the chance of over time depends on when all of the production are completed. The problem statement requests the estimation of the probability of overtime work. The sales order process determines the number of rings to produce. The production process continues until there are no more rings to produce for that day. The number of rings to produce is a binomial random variable as determined by the sales order confirmation process. Thus, there is no clear run length for this simulation.

In the JSL, a replication of a simulation can end based on three situations:

1. A scheduled run length
2. A terminating condition is met
3. No more events to process

Because the production for the day stops when all the rings are produced, the third situation applies for this model. The simulation will end automatically after all the rings are produced. In essence, a day's worth of production is simulated. Because

the number of rings to produce is random and it takes a random amount of time to make, inspect, rework, and package the rings, the time that the simulation will end is a random variable. If this time is less than 960 (the time of two shifts), there will not be any overtime. If this time is greater than 960, production will have lasted past two shifts and thus overtime will be necessary. To assess the chance that there is overtime, you need to record statistics on how often the end of the simulation is past 960.

Thus, the easiest way to observe the over time is to understand that a replication of the simulation will end when there are no more events to process. Just like in the case of the initialize() method being called at the start of a replication, the replicationEnded() method of all model elements will be called when the simulation replication ends. This provides for the opportunity to supply code that will be executed when the replication ends. Also note that the replicationEnded() method is called *before* any logic that might clear statistical quantities and that no additional events happen after the replication ends. Lines 3 and 4 of the replicationEnded() method implement the collection of the probability of over time and the time that the simulation ends. This is accomplished with instances of the ResponseVariable class, which were declared as instance variables of the LOTR class and instantiated within its constructor.

The method getTime() available on all model elements provides the current time of the simulation. Thus, when the simulation ends, The method getTime() will be the time that the simulation ended. In this case, it will represent the time that the last ring completed processing. To estimate the chance that there is overtime, we use a ResponseVariable to capture the time. Since this occurs one time for each replication, the number of observations of the over time will be equal to the number of replications.

Running the model results in the user defined statistics for the probability of overtime and the average time to produce the orders as shown in in the folloing table. The probability of overtime appears to be about 3%, but their is a lot of variation for these 30 replications. The average time to produce an order is about 770 minutes. While the average is under 960, there still appears to be a reasonable chance of overtime occurring. What do you think causes the overtime? Can you ecommend an alternative to reduce the likelihood of overtime?

**Table 17:** Across Replication Statistics for LOTR Example

Response Name	$\bar{x}$	$s$
RingMakingStation:R:Util	0.980777	0.007716
RingMakingStation:R:BusyUnits	0.980777	0.007716
RingMakingStation:Q:Num In Q	36.877643	7.949062
RingMakingStation:Q:Time In Q	374.334264	79.128277
RingMakingStation:NS	37.858420	7.952406
InspectStation:R:Util	0.426290	0.022779
InspectStation:R:BusyUnits	0.426290	0.022779
InspectStation:Q:Num In Q	0.001148	0.001260
InspectStation:Q:Time In Q	0.011508	0.012382
InspectStation:NS	0.427438	0.023260

Response Name	$\bar{x}$	$s$
ReworkStation:R:Util	0.127256	0.050377
ReworkStation:R:BusyUnits	0.127256	0.050377
ReworkStation:Q:Num In Q	0.003795	0.006096
ReworkStation:Q:Time In Q	0.493970	0.794616
ReworkStation:NS	0.131051	0.052790
PackingStation:R:Util	0.690843	0.033373
PackingStation:R:BusyUnits	0.690843	0.033373
PackingStation:Q:Num In Q	0.108125	0.043984
PackingStation:Q:Time In Q	1.087263	0.411139
PackingStation:NS	0.798968	0.071767
System Time	398.068177	79.425695
Num in System	39.215877	7.992176
Prob too Big	0.026853	0.019754
Prob too Small	0.041889	0.019552
Prob of Over Time	0.033333	0.182574
Time to Make Orders	770.286991	159.076748
Num in RM and Inspection	38.285858	7.954368
Time in RM and Inspection	388.641522	79.155138
Across Rep Stat Num Completed	75.866667	15.904312
Number of Replications 30		

The final issue to be handled in this example is to specify the number of replications. Based on this initial run of 30 replication, the required number of replications will be computed to ensure a 95% confidence interval for the mean time to complete the ring making and inspection processes with an error bound of  $\pm 20$  minutes. The estimated standard deviation for this time was 79.155138.

Using the normal approximation with  $\alpha = 0.05$ ,  $n_0 = 30$ ,  $s_0 = 79.155138$ , and  $E = 20$ , indicates that approximately  $n = 61$  replications are needed to meet the criteria.

$$n \geq \left( \frac{z_{\alpha/2} s_0}{E} \right)^2 = \left( \frac{1.96 \times 79.155138}{20} \right)^2 \approx 61$$

If you wanted to use the iterative method, you must first determine the standard deviation from the pilot replications. In the case of multiple replications, you can use the half-width value and equation ((4)) to compute  $s_0$ .

Rerunning the simulation with  $n = 61$ , yields a half-width of 20.77, which is very close to the criteria of 20. Note that the make and inspection time is highly variable. The methods to determine the half-width assume that the standard deviation,  $s_0$ , in the pilot runs will be similar to the standard deviation observed in the final set of replications. However, when the full set of replications is run, the actual standard deviation may be different than used in the pilot run. Thus, the half-width criterion might not be exactly met. If the assumptions are reasonably met, there will be a high likelihood

that the desired half-width will be very close to the desired criteria, as show in this example.

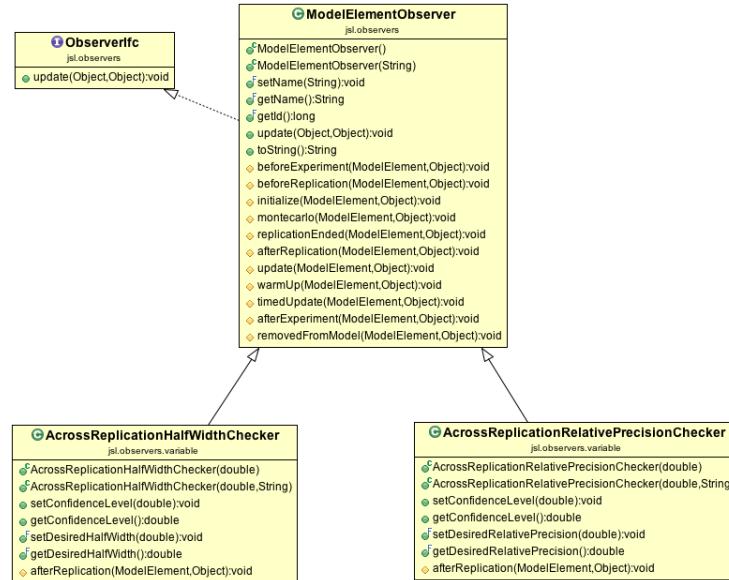
### 0.39.2 Sequential Sampling for Finite Horizon Simulations

The methods discussed for determining the sample size are based on pre-determining a *fixed* sample size and then making the replications. If the half-width equation is considered as an iterative function of  $n$ :

$$h(n) = t_{\alpha/2,n-1} \frac{s(n)}{\sqrt{n}} \leq E$$

Then, it becomes apparent that additional replications of the simulation can be executed until the desired half-width bound is met. This is called sequential sampling, and in this case the sample size of the experiment is not known in advance. The brute force method for implementing this approach would be to run and rerun the simulation each time increasing the number of replications until the criterion is met.

To implement this within the JSL, we need a way to stop or end a simulation when a criteria or condition is met. Because of the hierarchical nature of the model elements within a model and because there are common actions that occur when running a model the observer pattern can be used here.



**Figure 31:** Half-Width Observer Checking Code

Figure 31 presents part of the `jsl.observers.variable` package which defines a base class called `ModelElementObserver` that can be attached to instances of `ModelElement` and

then will be notified if various actions take place. There are a number of actions associated with a `ModelElement` that occur during a simulation that can be listened for by a `ModelElementObserver`:

- `beforeExperiment()` - This occurs prior to the first replication and before any events are executed.
- `beforeReplication()` - This occurs prior to each replication and before any events are executed. The event calendar is cleared after this action.
- `initialize()` - This occurs at the start of every replication (after `beforeReplication()` and after the event calendar is cleared) but before any events are executed. As we have seen, it is safe to schedule events in this method.
- `warmUp()` - This occurs during a replication if a warm up period has been specified for the model. The statistical accumulators are cleared during this action if applicable.
- `replicationEnded()` - This occurs at the end of every replication prior to the clearing of any statistical accumulators.
- `afterReplication()` - This occurs at the end of every replication after the statistical accumulators have been cleared for the replication.
- `afterExperiment()` - This occurs after all replications have been executed and prior to the end of the simulation.

`ModelElementObservers` are notified right *after* the `ModelElement` experiences the above mentioned actions. Thus, users of the `ModelElementObserver` need to understand that the state of the model element is available after the simulation actions of the model element have occurred.

The `AcrossReplicationHalfWidthChecker` class listens to the `afterReplication()` method of a `ResponseVariable` and checks the current value of the half-width. This is illustrated in the following code listing.

```
protected void afterReplication(ModelElement m, Object arg) {
    ResponseVariable x = (ResponseVariable) m;
    Simulation s = x.getSimulation();

    if (s == null) {
        return;
    }
    if (s.getCurrentReplicationNumber() <= 2.0) {
        return;
    }

    StatisticAccessorIfc stat = x.getAcrossReplicationStatistic();
    double hw = stat.getHalfWidth(getConfidenceLevel());
```

```
    if (hw <= myDesiredHalfWidth) {
        s.send("Half-width condition met for " + x.getName());
    }
}
```

Notice that a reference to the observed `ResponseVariable` is used to get access to the across replication statistics. If there are more than 2 replications, then the half-width is checked against a user supplied desired half-width. If the half-width criterion is met, then the simulation is told to end using the `Simulation` class's `end()` method. The `end()` method causes the simulation to not execute any future replications and to halt further execution.

The following code listing illustrates how to set up half-width checking.

```
Simulation sim = new Simulation("LOTR Example");
// get the model
Model m = sim.getModel();
// add system to the main model
LOTR system = new LOTR(m, "LOTR");
ResponseVariable rsv = m.getResponseVariable("Time in RM and Inspection");
AcrossReplicationHalfWidthChecker hwc = new AcrossReplicationHalfWidthChecker(20.0);
rsv.addObserver(hwc);
// set the parameters of the experiment
sim.setNumberOfReplications(1000);
System.out.println("Simulation started.");
sim.run();
System.out.println("Simulation completed.");
sim.printHalfWidthSummaryReport();
```

All that is needed is to get a reference to the response variable that needs to be checked so that the observer can be attached. This can be done easily in the location of the code where the response variable is created or as in this example, the name of the response variable is used to get the reference from the model. Line 6 of the listing illustrates using the name to get the reference, followed by the construction of the checker (line 7) and attaching it as an observer (line 8).

In the sequential sampling experiment there were 66 replications. This is actually more than the recommended 61 replications for the fixed half-width method. This is perfectly possible, and emphasizes the fact that in the sequential sampling method, the number of replications is actually a random variable. If you were to use different streams and re-run the sequential sampling experiment, the number of replications completed may be different each time.

**Table 18:** Half-Width Summary Report for Sequential Analysis

Response Name	$\bar{x}$	$hw$
RingMakingStation:R:Util	0.981083	0.002039
RingMakingStation:R:BusyUnits	0.981083	0.002039
RingMakingStation:Q:Num In Q	37.364044	1.963297
RingMakingStation:Q:Time In Q	380.608370	19.903070
RingMakingStation:NS	38.345127	1.964300
InspectStation:R:Util	0.426379	0.005022
InspectStation:R:BusyUnits	0.426379	0.005022
InspectStation:Q:Num In Q	0.000948	0.000263
InspectStation:Q:Time In Q	0.009558	0.002622
InspectStation:NS	0.427327	0.005109
ReworkStation:R:Util	0.127123	0.011491
ReworkStation:R:BusyUnits	0.127123	0.011491
ReworkStation:Q:Num In Q	0.004093	0.001949
ReworkStation:Q:Time In Q	0.530734	0.255424
ReworkStation:NS	0.131216	0.012288
PackingStation:R:Util	0.687782	0.006946
PackingStation:R:BusyUnits	0.687782	0.006946
PackingStation:Q:Num In Q	0.103600	0.008483
PackingStation:Q:Time In Q	1.050310	0.080851
PackingStation:NS	0.791382	0.013703
System Time	404.357533	19.948930
Num in System	39.695052	1.969934
Prob too Big	0.031764	0.004634
Prob too Small	0.038146	0.004449
Prob of Over Time	0.106061	0.076275
Time to Make Orders	782.451179	39.996433
Num in RM and Inspection	38.772454	1.964935
Time in RM and Inspection	394.965666	19.914630
Across Rep Stat Num Completed	76.803030	3.936414
Number of replications: 66		

## 0.40 Analysis of Infinite Horizon Simulations

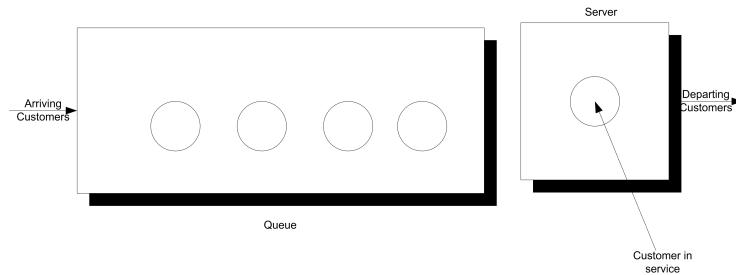
This section discusses how to plan and analyze infinite horizon simulations. When analyzing infinite horizon simulations, the primary difficulty is the nature of within replication data. In the finite horizon case, the statistical analysis is based on three basic requirements:

1. Observations are independent
2. Observations are sampled from identical distributions

3. Observations are drawn from a normal distribution (or enough observations are present to invoke the central limit theorem)

These requirements were met by performing independent replications of the simulation to generate a random sample. In a direct sense, the data within a replication do not satisfy any of these requirements; however, certain procedures can be imposed on the manner in which the observations are gathered to ensure that these statistical assumptions are not grossly violated. The following will first explain why within replication data typically violates these assumptions and then will provide some methods for mitigating the violations within the context of infinite horizon simulations.

To illustrate the challenges related to infinite horizon simulations, a simple spreadsheet simulation was developed for a M/M/1 queue.



**Figure 32:** Single Server Queueing System

Consider a single server queueing system as illustrated Figure 32.

For a single server queueing system, there is an equation that allows the computation of the waiting times of each of the customers based on knowledge of the arrival and service times. Let  $X_1, X_2, \dots$  represent the successive service times and  $Y_1, Y_2, \dots$  represent the successive inter-arrival times for each of the customers that visit the queue. Let  $E[Y_i] = 1/\lambda$  be the mean of the inter-arrival times so that  $\lambda$  is the mean arrival rate. Let  $E[X_i] = 1/\mu$  be the mean of the service times so that  $\mu$  is the mean service rate. Let  $W_i$  be the waiting time in the queue for the  $i^{th}$  customer. That is, the time between when the customer arrives until they enter service.

Lindley's equation, see (Gross and Harris, 1998), relates the waiting time to the arrivals and services as follows:

$$W_{i+1} = \max(0, W_i + X_i - Y_i)$$

The relationship says that the time that the  $(i+1)^{st}$  customer must wait is the time the  $i^{th}$  waited, plus the  $i^{th}$  customer's service time,  $X_i$  (because that customer is in front of the  $i^{th}$  customer), less the time between arrivals of the  $i^{th}$  and  $(i+1)^{st}$  customers,  $Y_i$ . If  $W_i + X_i - Y_i$  is less than zero, then the  $((i+1)^{st}$  customer arrived after the  $i^{th}$  finished service, and thus the waiting time for the  $(i+1)^{st}$  customer is zero, because his service starts immediately.

Suppose that  $X_i \sim \exp(E[X_i] = 0.7)$  and  $Y_i \sim \exp(E[Y_i] = 1.0)$ . This is a M/M/1 queue with  $\lambda = 1$  and  $\mu = 10/7$ . Thus, based on traditional queuing theory results:

$$\rho = 0.7$$

$$L_q = \frac{0.7 \times 0.7}{1 - 0.7} = 1.6\bar{3}$$

$$W_q = \frac{L_q}{\lambda} = 1.6\bar{3} \text{ minutes}$$

Lindley's equation can be readily implemented in Java as illustrated in the following code listing.

```
// inter-arrival time distribution
RandomIfc y = new ExponentialRV(1.0);
// service time distribution
RandomIfc x = new ExponentialRV(0.7);
int r = 30; // number of replications
int n = 100000; // number of customers
int d = 10000; // warm up
Statistic avgw = new Statistic("Across rep avg waiting time");
Statistic wbar = new Statistic("Within rep avg waiting time");
for (int i = 1; i <= r; i++) {
    double w = 0; // initial waiting time
    for (int j = 1; j <= n; j++) {
        w = Math.max(0.0, w + x.getValue() - y.getValue());
        wbar.collect(w); // collect waiting time
        if (j == d) { // clear stats at warmup
            wbar.reset();
        }
    }
    //collect across replication statistics
    avgw.collect(wbar.getAverage());
    // clear within replication statistics for next rep
    wbar.reset();
}
System.out.println("Replication/Deletion Lindley Equation Example");
System.out.println(avgw);
```

This implementation can be readily extended to capture the data to files for display in spreadsheets or other plotting software. As part of the plotting process it is useful to display the cumulative sum and cumulative average of the data values.

$$\sum_{i=1}^n W_i \text{ for } n = 1, 2, \dots$$

$$\frac{1}{n} \sum_{i=1}^n W_i \text{ for } n = 1, 2, \dots$$



**Figure 33:** Cumulative Average Waiting Time of 1000 Customers

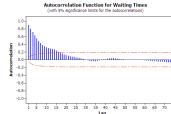
Figure 33 presents the cumulative average plot of the first 1000 customers. As seen in the plot, the cumulative average starts out low and then eventually trends towards 1.2 minutes.

A	B
1	Simple Generating Mechanism
2	Mean Service Time
3	Mean Waiting Time
4	Waiting Time in Queue
5	Length of Queue
6	Sample Variance
7	Correlation
8	Std Err
9	Std Dev
10	Skewness
11	Kurtosis
12	Argmax of Posterior
13	Posterior Std Dev
14	Posterior Correlation
15	Posterior Std Err
16	Posterior Std Dev
17	Posterior Correlation
18	Upper Limit for mean
19	Lower Limit for mean

**Figure 34:** Lindley Equation Results Across 1000 Customers

The analytical results indicate that the true long-run expected waiting time in the queue is 1.633 minutes. The average over the 1000 customers in the simulation is 1.187 minutes. The results in Figure 34 indicated that the sample average is significantly lower than the true expected average. We will explore why this occurs shortly.

The first issue to consider with this data is independence. To do this you should analyze the 1000 observations in terms of its autocorrelation.



**Figure 35:** Autocorrelation Plot for Waiting Times

From Figure 35, it is readily apparent that the data has strong positive correlation. The lag-1 correlation for this data is estimated to be about 0.9. Figure 35 clearly indicates the strong first order linear dependence between  $W_i$  and  $W_{i-1}$ . This positive dependence implies that if the previous customer waited a long time the next customer is likely to wait a long time. If the previous customer had a short wait, then the next customer is likely to have a short wait. This makes sense with respect to how a queue operates.

Strong positive correlation has serious implications when developing confidence intervals on the mean customer waiting time because the usual estimator for the sample variance:

$$S^2(n) = \frac{1}{n-1} \sum_{i=1}^n (X_i - \bar{X})^2$$

is a biased estimator for the true population variance when there is correlation in the observations. This issue will be re-examined when ways to mitigate these problems are discussed.

The second issue that needs to be discussed is that of the non-stationary behavior of the data. Non-stationary data indicates some dependence on time. More generally, non-stationary implies that the  $W_1, W_2, W_3, \dots, W_n$  are not obtained from identical distributions.

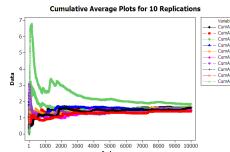
Why should the distribution of  $W_1$  not be the same as the distribution of  $W_{1000}$ ? The first customer is likely to enter the queue with no previous customers present and thus it is very likely that the first customer will experience little or no wait (the way  $W_0$  was initialized in this example allows a chance of waiting for the first customer). However, the 1000<sup>th</sup> customer may face an entirely different situation. Between the 1<sup>st</sup> and the 1000<sup>th</sup> customer there might likely be a line formed. In fact from the M/M/1 formula, it is known that the steady state expected number in the queue is 1.633. Clearly, the conditions that the 1<sup>st</sup> customer faces are different than the 1000<sup>th</sup> customer. Thus, the distributions of their waiting times are likely to be different.

This situation can be better understood by considering a model for the underlying data. A time series,  $X_1, X_2, \dots$ , is said to be *covariance stationary* if:

- The mean exists and  $\theta = E[X_i]$ , for  $i=1,2,\dots, n$
- The variance exists and  $Var[X_i] = \sigma^2 > 0$ , for  $i=1,2,\dots, n$
- The lag-k autocorrelation,  $\rho_k = cor(X_i, X_{i+k})$ , is not a function of  $i$ , i.e. the correlation between any two points in the series does not depend upon where the points are in the series, it depends only upon the distance between them in the series.

In the case of the customer waiting times, we can conclude from the discussion that it is very likely that  $\theta \neq E[X_i]$  and  $Var[X_i] \neq \sigma^2$  for each  $i=1,2,\dots, n$  for the time series.

Do you think that is it likely that the distributions of  $W_{9999}$  and  $W_{10000}$  will be similar? The argument, that the 9999<sup>th</sup> customer is on average likely to experience similar conditions as the 10000<sup>th</sup> customer, sure seems reasonable.



**Figure 36:** Multiple Sample Paths of Queueing Simulation

Figure 36 shows 10 different replications of the cumulative average for a 10000 customer simulation. From the figure, we can see that the cumulative average plots can vary significantly over the 10000 customers with the average tracking above the true expected value, below the true expected value, and possibly towards the true expected value. For the case of 10000 customers, you should notice that the cumulative average starts to approach the expected value of the steady state mean waiting time in the queue with increasing number of customers. This is the law of large numbers in action. It appears that it takes a period of time for the performance measure to *warm up* towards the true mean. Determining the warm up time will be the basic way to mitigate the problem of this non-stationary behavior.

From this discussion, we can conclude that the second basic statistical assumption of identically distributed data is not valid for within replication data. From this, we can also conclude that it is very likely that the data are not normally distributed. In fact, for the M/M/1 it can be shown that the steady state distribution for the waiting time in the queue is not a normal distribution. Thus, all three of the basic statistical assumptions are violated for the within replication data of this example. This problem needs to be addressed in order to properly analyze infinite horizon simulations.

There are two basic methods for performing infinite horizon simulations. The first is to perform multiple replications. This approach addresses independence and normality in a similar fashion as the finite horizon case, but special procedures will be needed to address the non-stationary aspects of the data. The second basic approach is to work with one very long replication. Both of these methods depend on first addressing the problem of the non-stationary aspects of the data. The next section looks at ways to mitigate the non-stationary aspect of within-replication data for infinite horizon simulations.

#### 0.40.1 Assessing the Effect of Initial Conditions

Consider the output stochastic process  $X_i$  of the simulation. Let  $F_i(x|I)$  be the conditional cumulative distribution function of  $X_i$  where  $I$  represents the initial conditions used to start the simulation at time 0. If  $F_i(x|I) \rightarrow F(x)$  when  $i \rightarrow \infty$ , for all initial conditions  $I$ , then  $F(x)$  is called the steady state distribution of the output process. (Law 2007).

In infinite horizon simulations, estimating parameters of the steady state distribution,  $F(x)$ , such as the steady state mean,  $\theta$ , is often the key objective. The fundamental difficulty associated with estimating steady state performance is that unless the system is initialized using the steady state distribution (which is not known), there is no way to directly observe the steady state distribution.

It is true that if the steady state distribution exists and you run the simulation long enough the estimators will tend to converge to the desired quantities. Thus, within the infinite horizon simulation context, you must decide on how long to run the simulations and how to handle the effect of the *initial conditions* on the estimates of performance. The initial conditions of a simulation represent the state of the system when

the simulation is started. For example, in simulating the pharmacy system, the simulation was started with no customers in service or in the line. This is referred to as *empty and idle*. The initial conditions of the simulation affect the rate of convergence of estimators of steady state performance.

Because the distributions  $F_i(x|I)$  at the start of the replication tend to depend more heavily upon the initial conditions, estimators of steady state performance such as the sample average,  $\bar{X}$ , will tend to be *biased*. A point estimator,  $\hat{\theta}$ , is an *unbiased* estimator of the parameter of interest,  $\theta$ , if  $E[\hat{\theta}] = \theta$ . That is, if the expected value of the sampling distribution is equal to the parameter of interest then the estimator is said to be unbiased. If the estimator is biased then the difference,  $E[\hat{\theta}] - \theta$ , is called the bias of,  $\hat{\theta}$ , the estimator.

Note that any individual difference between the true parameter,  $\theta$ , and a particular observation,  $X_i$ , is called error,  $\epsilon_i = X_i - \theta$ . If the expected value of the errors is not zero, then there is bias. A particular observation is not biased. Bias is a property of the estimator. Bias is analogous to being consistently off target when shooting at a bulls-eye. It is as if the sights on your gun are crooked. In order to estimate the bias of an estimator, you must have multiple observations of the estimator. Suppose that you are estimating the mean waiting time in the queue as per the previous example and that the estimator is based on the first 20 customers. That is, the estimator is:

$$\bar{W}_r = \frac{1}{20} \sum_{i=1}^{20} W_{ir}$$

and there are  $r = 1, 2, \dots, 10$  replications. The following table shows the sample average waiting time for the first 20 customers for 10 different replications.

**Table 19:** Ten Replications of 20 Customers

r	$\bar{W}_r$	$B_r = \bar{W}_r - W_q$
1	0.194114	-1.43922
2	0.514809	-1.11852
3	1.127332	-0.506
4	0.390004	-1.24333
5	1.05056	-0.58277
6	1.604883	-0.02845
7	0.445822	-1.18751
8	0.610001	-1.02333
9	0.52462	-1.10871
10	0.335311	-1.29802
$\bar{W} = 0.6797$		$\bar{B} = -0.9536$

In the table,  $B_r$  is an estimate of the bias for the  $r^{th}$  replication, where  $W_q = 1.633$ .

Upon averaging across the replications, it can be seen that  $\bar{B} = -0.9536$ , which indicates that the estimator based only on the first 20 customers has significant negative bias, i.e. on average it is less than the target value.

This is the so called *initialization bias problem* in steady state simulation. Unless the initial conditions of the simulation can be generated according to  $F(x)$ , which is not known, you must focus on methods that detect and/or mitigate the presence of initialization bias.

One strategy for initialization bias mitigation is to find an index,  $d$ , for the output process,  $X_i$ , so that  $X_i; i = d + 1, \dots$  will have substantially similar distributional properties as the steady state distribution  $F(x)$ . This is called the simulation warm up problem, where  $d$  is called the warm up point, and  $i = 1, \dots, d$  is called the warm up period for the simulation. Then, the estimators of steady state performance are based only on  $X_i; i = d + 1, \dots$ , which represent the data after deleting the warm up period.

For example, when estimating the steady state mean waiting time for each replication  $r$  the estimator would be:

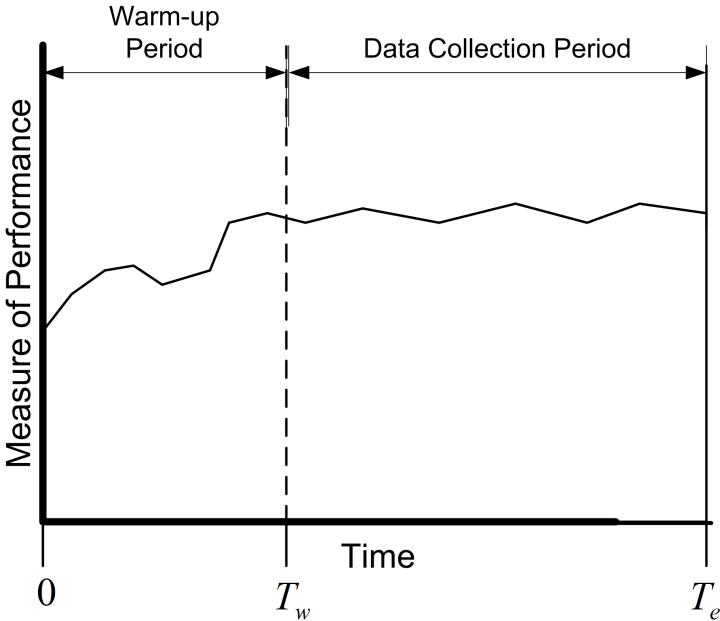
$$\bar{W}_r = \frac{1}{n-d} \sum_{i=d+1}^n W_{ir}$$

For time-based performance measures, such as the average number in queue, a time  $T_w$  can be determined past which the data collection process can begin. Estimators of time-persistent performance such as the sample average are computed as:

$$\bar{Y}_r = \frac{1}{T_e - T_w} \int_{T_w}^{T_e} Y_r(t) dt$$

Figure 37 shows the concept of a warm up period for a simulation replication. When you perform a simulation, you can easily specify a time-based warm up period using the `setLengthOfWarmUp()` method of the `Simulation` class. In fact, even for observation based data, it will be more convenient to specify the warm up period in terms of time. A given value of  $T_w$  implies a particular value of  $d$  and vice versa. Specifying a warm up period, causes an event to be scheduled for time  $T_w$ . At that time, all the accumulated statistical counters are cleared so that the net effect is that statistics are only collected over the period from  $T_w$  to  $T_e$ . The problem then becomes that of finding an appropriate warm up period.

Before proceeding with how to assess the length of the warm up period, the concept of steady state needs to be further examined. This subtle concept is often misunderstood or misrepresented. Often you will hear the phrase: *The system has reached steady state*. The correct interpretation of this phrase is that the distribution of the desired performance measure has reached a point where it is sufficiently similar to the desired steady state distribution. Steady state is a concept involving the performance measures generated by the system as time goes to infinity. However, sometimes this phrase is interpreted incorrectly to mean that the system *itself* has reached steady state.



**Figure 37:** The Concept of the Warm Up Period

Let me state emphatically that the system *never* reaches steady state. If the system itself reached steady state, then by implication it would never change with respect to time. It should be clear that the system continues to evolve with respect to time; otherwise, it would be a very boring system! Thus, it is incorrect to indicate that the system has reached steady state. Because of this, do not use the phrase: *The system has reached steady state.*

Understanding this subtle issue raises an interesting implication concerning the notion of deleting data to remove the initialization bias. Suppose that the state of the system at the end of the warm up period,  $T_w$ , is exactly the same as at  $T = 0$ . For example, it is certainly possible that at time  $T_w$  for a particular replication that the system was empty and idle. Since the state of the system at  $T_w$  is the same as that of the initial conditions, there will be no effect of deleting the warm up period for this replication. In fact there will be a negative effect, in the sense that data will have been thrown away for no reason. Deletion methods are predicated on the likelihood that the state of the system seen at  $T_w$  is more representative of steady state conditions. At the end of the warm up period, the system can be in *any of the possible* states of the system. Some states will be more likely than others. If multiple replications are made, then at  $T_w$  each replication will experience a different set of conditions at  $T_w$ . Let  $I_{T_w}^r$  be the initial conditions (state) at time  $T_w$  on replication  $r$ . By setting a warm up period and performing multiple replications, you are in essence sampling from the distribution governing the state of the system at time  $T_w$ . If  $T_w$  is long enough, then on average across the replications, you are more likely to start collecting data when the system is

in states that are more representative over the long term (rather than just empty and idle).

Many methods and rules have been proposed to determine the warm up period. The interested reader is referred to (Wilson and Pritsker, 1978), Lada et al. (2003), (Litton and Harmonosky, 2002), White et al. (2000), Cash et al. (1992), and (Rossetti and De-laney, 1995) for an overview of such methods. This discussion will concentrate on the visual method proposed in (Welch, 1983).

The basic idea behind Welch's graphical procedure is simple:

- Make  $R$  replications. Typically,  $R \geq 5$  is recommended.
- Let  $Y_{rj}$  be the  $j^{th}$  observation on replication  $r$  for  $j = 1, 2, \dots, m_r$  where  $m_r$  is the number of observations in the  $r^{th}$  replication, and  $r = 1, 2, \dots, n$ ,
- Compute the averages across the replications for each  $j = 1, 2, \dots, m$ , where  $m = \min(m_r)$  for  $r = 1, 2, \dots, n$ .

$$\bar{Y}_{.j} = \frac{1}{n} \sum_{r=1}^n Y_{rj}$$

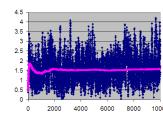
- Plot,  $\bar{Y}_{.j}$  for each  $j = 1, 2, \dots, m$
- Apply smoothing techniques to  $\bar{Y}_{.j}$  for  $j = 1, 2, \dots, m$
- Visually assess where the plots start to converge

Let's apply the Welch's procedure to the replications generated from the Lindley equation simulation. Using the 10 replications stored in a spreadsheet we can compute the average across each replication for each customer.

1									
2									
3									
4									
5									
6									
7									
8									
9									
10									
11									
12									
13									
14									
15									
16									
17									
18									
19									
20									
21									
22									
23									
24									
25									
26									
27									
28									
29									
30									
31									
32									
33									
34									
35									
36									
37									
38									
39									
40									
41									
42									
43									
44									
45									
46									
47									
48									
49									
50									
51									
52									
53									
54									
55									
56									
57									
58									
59									
60									
61									
62									
63									
64									
65									
66									
67									
68									
69									
70									
71									
72									
73									
74									
75									
76									
77									
78									
79									
80									
81									
82									
83									
84									
85									
86									
87									
88									
89									
90									
91									
92									
93									
94									
95									
96									
97									
98									
99									
100									

**Figure 38:** Computing the Averages for the Welch Plot

In Figure 38, cell B2 represents the average across the 10 replications for the 1<sup>st</sup> customer. Column D represents the cumulative average associated with column B.



**Figure 39:** Welch Plot with Superimposed Cumulative Average Line

Figure 39 is the plot of the cumulative average (column D) superimposed on the averages across replications (column B). The cumulative average is one method of smoothing the data. From the plot, you can infer that after about customer 3000 the cumu-

lative average has started to converge. Thus, from this analysis you might infer that  $d = 3000$ .

When you perform an infinite horizon simulation by specifying a warm up period and making multiple replications, you are using the method of *replication-deletion*. If the method of replication-deletion with  $d = 3000$  is used for the current example, a slight reduction in the bias can be achieved as indicated in the following table.

**Table 20:** Replication-Deletion Results,  $d = 3000$

$r$	$\bar{W}_r(d = 0)$	$\bar{W}_r(d = 3000)$	$B_r(d = 0)$	$B_r(d = 3000)$
1	1.594843	1.592421	-0.03849	-0.04091
2	1.452237	1.447396	-0.1811	-0.18594
3	1.657355	1.768249	0.024022	0.134915
4	1.503747	1.443251	-0.12959	-0.19008
5	1.606765	1.731306	-0.02657	0.097973
6	1.464981	1.559769	-0.16835	-0.07356
7	1.621275	1.75917	-0.01206	0.125837
8	1.600563	1.67868	-0.03277	0.045347
9	1.400995	1.450852	-0.23234	-0.18248
10	1.833414	1.604855	0.20008	-0.02848
	$\bar{\bar{W}} = 1.573617$	$\bar{\bar{W}} = 1.603595$	$\bar{B} = -0.05972$	$\bar{B} = -0.02974$
	$s = 0.1248$	$s = 0.1286$	$s = 0.1248$	$s = 0.1286$
95% LL	1.4843	1.5116	-0.149023	-0.121704
95% UL	1.6629	1.6959	-0.029590	0.062228

While not definitive for this simple example, the results suggest that deleting the warm up period helps to reduce initialization bias. This model's warm up period will be further analyzed using additional tools available in the next section.

In performing the method of replication-deletion, there is a fundamental trade-off that occurs. Because data is deleted, the variability of the estimator will tend to increase while the bias will tend to decrease. This is a trade-off between a reduction in bias and an increase in variance. That is, accuracy is being traded off against precision when deleting the warm up period. In addition to this trade off, data from each replication is also being thrown away. This takes computational time that could be expended more effectively on collecting usable data. Another disadvantage of performing replication-deletion is that the techniques for assessing the warm up period (especially graphical) may require significant data storage. The Welch plotting procedure requires the saving of data points for post processing after the simulation run. In addition, significant time by the analyst may be required to perform the technique and the technique is subjective.

When a simulation has many performance measures, you may have to perform a warm up period analysis for every performance measure. This is particularly important, since in general, the performance measures of the same model may converge

towards steady state conditions at different rates. In this case, the length of the warm up period must be sufficiently long enough to cover all the performance measures. Finally, replication-deletion may simply compound the bias problem if the warm up period is insufficient relative to the length of the simulation. If you have not specified a long enough warm up period, you are potentially compounding the problem for  $n$  replications.

Despite all these disadvantages, replication-deletion is very much used in practice because of the simplicity of the analysis after the warm up period has been determined. Once you are satisfied that you have a good warm up period, the analysis of the results is the same as that of finite horizon simulations. Replication-deletion also facilitates the use of experimental design techniques that rely on replicating design points.

The next section illustrates how to perform the method of replication-deletion on this simple M/M/1 model.

#### 0.40.2 Performing the Method of Replication-Deletion

The first step in performing the method of replication-deletion is to determine the length of the warm up period. This example illustrates how to:

- Save the values from observation and time-based data to files for post processing
- Make Welch plots based on the saved data
- Setup and run the multiple replications
- Interpret the results

When performing a warm-up period analysis, the first decision to make is the length of each replication. In general, there is very little guidance that can be offered other than to try different run lengths and check for the sensitivity of your results. Within the context of queueing simulations, the work by (Whitt, 1989) offers some ideas on specifying the run length, but these results are difficult to translate to general simulations.

Since the purpose here is to determine the length of the warm up period, then the run length should be bigger than what you suspect the warm up period to be. In this analysis, it is better to be conservative. You should make the run length as long as possible given your time and data storage constraints. Banks et al. (2005) offer the rule of thumb that the run length should be at least 10 times the amount of data deleted. That is,  $n \geq 10d$  or in terms of time,  $T_e \geq 10T_w$ . Of course, this is a “catch 22” situation because you need to specify  $n$  or equivalently  $T_e$  in order to assess  $T_w$ . Setting  $T_e$  very large is recommended when doing a preliminary assessment of  $T_w$ . Then, you can use the rule of thumb of 10 times the amount of data deleted when doing a more serious assessment of  $T_w$  (e.g. using Welch plots etc.)

A preliminary assessment of the current model has already been performed based on the previously described spreadsheet simulation. That assessment suggested a deletion point of at least  $d = 3000$  customers. This can be used as a starting point in the

current effort. Now,  $T_w$  needs to be determined based on  $d$ . The value of  $d$  represents the customer number for the end of the warm up period. To get  $T_w$ , you need to answer the question: How long (on average) will it take for the simulation to generate  $d$  observations. In this model, the mean number of arrivals is 1 customer per minute. Thus, the initial  $T_w$  is

$$3000 \text{ customers} \times \frac{\text{minute}}{1 \text{ customer}} = 3000 \text{ minutes}$$

and therefore the initial  $T_e$  should be 30,000 minutes. That is, you should specify 30,000 minutes for the replication length.

#### 0.40.2.1 Determining the Warm Up Period

To perform a more rigorous analysis of the warm up period, we need to run the simulation for multiple replications and capture the data necessary to produce Welch plots for the performance measures of interest. Within the JSL, this can be accomplished by using the classes within the `welch` package in the JSL utilities. There are three classes of note:

**`WelchDataFileCollector`** This class captures data associated with a `ResponseVariable` within the simulation model. Every observation from every replication is written to a binary data file. In addition, a text based data file is created that contains the meta data associated with the data collection process. The meta data file records the number of replications, the number of observations for each replication, and the time between observations for each replication.

**`WelchDataFileCollectorTW`** This class has the same functionality as the `WelchDataFileCollector` class except for TimeWeighted response variables. In this case, a discretizing interval must be provided in order to batch the time weighted observations into discrete intervals.

**`WelchDataFileAnalyzer`** This class uses the files produced by the previous two classes to produces the Welch data. That is, the average across each row of observations and the cumulative average over the observations. This class produces files from which the plots can be made.

A warm up period analysis is associated with a particular response. The goal is to determine the number of observations of the response to delete at the beginning of the simulation. We can collect the relevant data by attaching an observer to the response variable. In the following listing, lines 8 and 9 get access to the two responses from the model. Then, in lines 12-16, the data collectors are created and attached to the responses. This needs to be done before the simulation is executed. The simulation is then run and then, the `WelchDataFileAnalyzer` instances for each of the responses are made. The Welch plot data is written to a file for plotting. Finally, the Welch chart is displayed. This code illustrates the display for the system time. In a similar manner the Welch plot for the number in the system can be performed.

```

Simulation sim = new Simulation("DTP");
// get the model
Model m = sim.getModel();
// add DriveThroughPharmacy to the main model
DriveThroughPharmacy driveThroughPharmacy = new DriveThroughPharmacy(m);
driveThroughPharmacy.setArrivalRS(new ExponentialRV(1.0));
driveThroughPharmacy.setServiceRS(new ExponentialRV(0.7));
ResponseVariable rv = m.getResponseVariable("System Time");
TimeWeighted tw = m.getTimeWeighted("# in System");
File d = JSL.makeOutputSubDirectory(sim.getName());
// this creates files to capture welch data from a ResponseVariable
WelchDataFileCollector wdfc = new WelchDataFileCollector(d, "welchsystime");
rv.addObserver(wdfc);
// this creates files to collection welch data from a TimeWeighted variable
WelchDataFileCollectorTW wdfctw = new WelchDataFileCollectorTW(10, d, "numInSystem");
tw.addObserver(wdfctw);
// set the parameters of the experiment
sim.setNumberOfReplications(5);
sim.setLengthOfReplication(5000.0);
SimulationReporter r = sim.makeSimulationReporter();
System.out.println("Simulation started.");
sim.run();
System.out.println("Simulation completed.");
r.printAcrossReplicationSummaryStatistics();
System.out.println(wdfc);
WelchDataFileAnalyzer wa = wdfc.makeWelchDataFileAnalyzer();
System.out.println(wa);

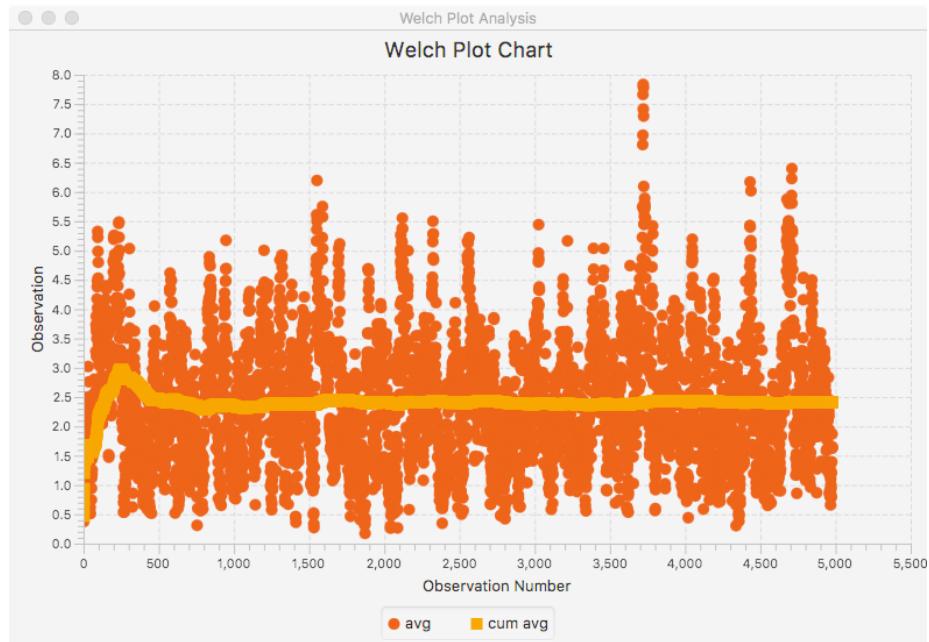
System.out.println("Writing welch data to csv");
wa.makeCSVWelchPlotDataFile();
System.out.println("Writing welch data to welch data file");
wa.makeWelchPlotDataFile();
System.out.println("Plotting the Welch plot for System Times");
WelchChart.display(wa);

```

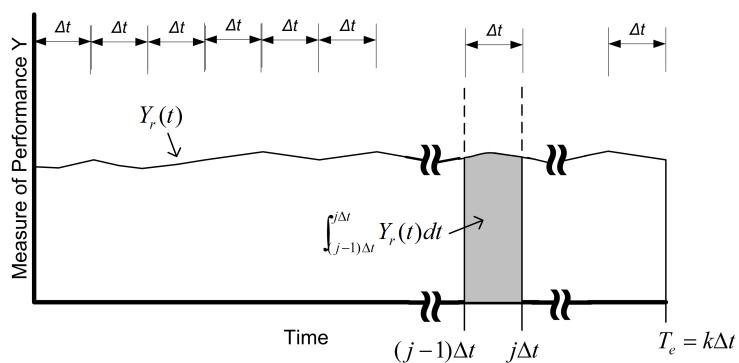
Figure 40 illustrates the plot of the data. From this plot, we can estimate the number of observations to delete as 2000.

Time-persistent observations are saved within a file from within the model such that the time of the observation and the value of the state variable at the time of change are recorded. Thus, the observations are not equally spaced in time. In order to perform the Welch plot analysis, we need to cut the data into discrete equally spaced intervals of time.

Figure 41 illustrates the concept of discretizing time-persistent data. Suppose that you divide  $T_e$  into  $k$  intervals of size  $\Delta t$ , so that  $T_e = k \times \Delta t$ . The time average over the



**Figure 40:** Welch Plot for System Time Analysis



**Figure 41:** Discretizing Time-Persistent Data

$j^{th}$  interval is given by:

$$\bar{Y}_{rj} = \frac{1}{\Delta t} \int_{(j-1)\Delta t}^{j\Delta t} Y_r(t) dt$$

Thus, the overall time average can be computed from the time average associated with each interval:

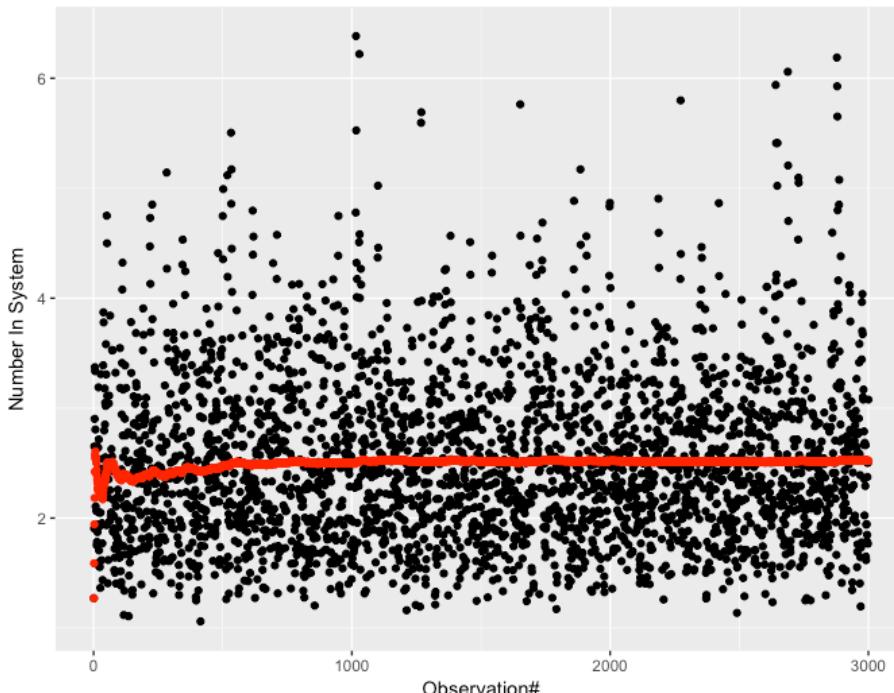
$$\begin{aligned}\bar{Y}_r &= \frac{\int_0^{T_e} Y_r(t) dt}{T_e} = \frac{\int_0^{T_e} Y_r(t) dt}{k\Delta t} \\ &= \frac{\sum_{j=1}^k \int_{(j-1)\Delta t}^{j\Delta t} Y_r(t) dt}{k\Delta t} = \frac{\sum_{j=1}^k \bar{Y}_{rj}}{k}\end{aligned}$$

Each of the  $\bar{Y}_{rj}$  are computed over intervals of time that are equally spaced and can be treated as if they are observation (tally) based data.

The computation of the  $\bar{Y}_{rj}$  for time-persistent data can be achieved by using the WelchDataFileCollectorTW class and specifying a discretization interval. Since the number in queue data is time-persistent, time based batches are selected, and the batch size is specified in terms of time. In this case, the data is being batched based on a time interval of 10 minutes in line 15 of the code listing. This produces a file which contains the  $\bar{Y}_{rj}$  as observations. This file can then be analyzed as previously illustrated.

The resulting plot is show in Figure 42. This plot is sparser than the previous plot because each observation represents the average of 10 minutes. There are 3000 observations. From the plot, we can conclude that after observation 1000, we see a steady convergence. The 1000th observation represents 10000 time units (minutes) because every observation represents 10 time units.

Once you have performed the warm up analysis, you still need to use your simulation model to estimate system performance. Because the process of analyzing the warm up period involves saving the data you could use the already saved data to estimate your system performance after truncating the initial portion of the data from the data sets. If re-running the simulation is relatively inexpensive, then you can simply set the warm up period via the Simulation class and re-run the model. Following the rule of thumb that the length of the run should be at least 10 times the warm up period, the simulation was re-run with the settings (30 replications, 10000 minute warm up period, 100,000 minute replication length). The results shown in the following table indicate that there does not appear to be any significant bias with these replication settings.



**Figure 42:** Welch Plot of Time-Persistent Number in System Data

**Table 21:** Across Replication Statistics for Drive Through Pharmacy

Response Name	$\bar{x}$	$s$
PharmacyQ:Num In Q	1.637069	0.040491
PharmacyQ:Time In Q	1.636834	0.037817
NumBusy	0.700038	0.003435
Number in System	2.337107	0.043385
System Time	2.336790	0.039586
Number of Replications	30	

The true waiting time in the queue is 1.633 and it is clear that the 95% confidence interval contains this value.

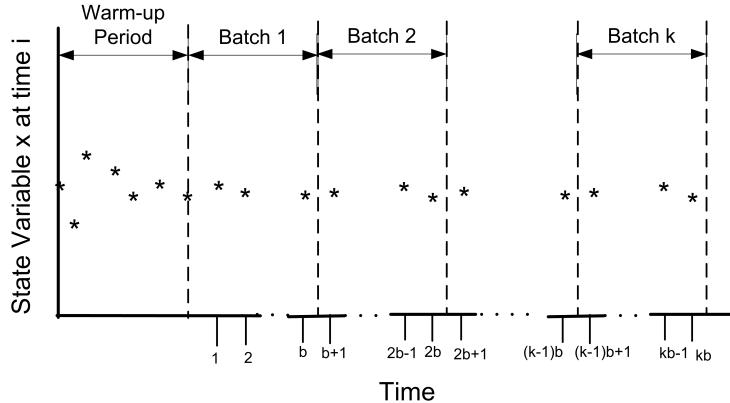
The process described here for determining the warm up period for steady state simulation is tedious and time consuming. Research into automating this process is still an active area of investigation. The recent work by (Robinson, 2005) and ? holds some promise in this regard; however, there remains the need to integrate these methods into computer simulation software. Even though determining the warm up period is tedious, some consideration of the warm up period should be done for infinite horizon simulations.

Once the warm up period has been found, you can set the warm up period using the Simulation class. Then, you can use the method of replication-deletion to perform your simulation experiments. Thus, all the discussion previously presented on the analysis of finite horizon simulations can be applied.

When determining the number of replications, you can apply the fixed sample size procedure after performing a pilot run. If the analysis indicates that you need to make more runs to meet your confidence interval half-width you have two alternatives: 1) increase the number of replications or 2) keep the same number of replications but increase the length of each replication. If  $n_0$  was the initial number of replications and

### 0.40.3 The Method of Batch Means

In the batch mean method, only one simulation run is executed. After deleting the warm up period, the remainder of the run is divided into  $k$  batches, with each batch average representing a single observation.



**Figure 43:** Illustration of the Batch Means Method

Figure 43 illustrates the concept of batching observations. The advantages of the batch means method are that it entails a long simulation run, thus dampening the effect of the initial conditions. The disadvantage is that the within replication data are correlated and unless properly formed the batches may also exhibit a strong degree of correlation.

The following presentation assumes that a warm up analysis has already been performed and that the data that has been collected occurs after the warm up period. For simplicity, the presentation assumes observation based data. The discussion also applies to time-based data that has been cut into discrete equally spaced intervals of time as previously described. Therefore, assume that a series of observations,  $(X_1, X_2, X_3, \dots, X_n)$ , is available from within the one long replication after the warm up period. As shown earlier, the within replication data can be highly correlated. In that section, it was mentioned that standard confidence intervals based on

$$S^2(n) = \frac{1}{n-1} \sum_{i=1}^n (X_i - \bar{X})^2$$

are not appropriate for this type of data. Suppose you were to ignore the correlation, what would be the harm? In essence, a confidence interval implies a certain level of confidence in the decisions based on the confidence interval. When you use  $S^2(n)$  as defined above, you will not achieve the desired level of confidence because  $S^2(n)$  is a biased estimator for the variance of  $\bar{X}$  when the data are correlated. Under the assumption that the data are covariance stationary, an assessment of the harm in ignoring the correlation can be made. For a series that is covariance stationary, one can

show that

$$\text{Var}(\bar{X}) = \frac{\gamma_0}{n} \left[ 1 + 2 \sum_{k=1}^{n-1} \left( 1 - \frac{k}{n} \right) \rho_k \right]$$

where  $\gamma_0 = \text{Var}(X_i)$ ,  $\gamma_k = \text{Cov}(X_i, X_{i+k})$ , and  $\rho_k = \gamma_k/\gamma_0$  for  $k = 1, 2, \dots, n-1$ .

When the data are correlated,  $S^2/n$  is a biased estimator of  $\text{Var}(\bar{X})$ . To show this, you need to compute the expected value of  $S^2/n$  as follows:

$$E[S^2/n] = \frac{\gamma_0}{n} \left[ 1 - \frac{2R}{n-1} \right]$$

where

$$R = \sum_{k=1}^{n-1} \left( 1 - \frac{k}{n} \right) \rho_k$$

Bias is defined as the difference between the expected value of the estimator and the quantity being estimated. In this case, the bias can be computed with some algebra as:

$$\text{Bias} = E[S^2/n] - \text{Var}(\bar{Y}) = \frac{-2\gamma_0 R}{n-1}$$

Since  $\gamma_0 > 0$  and  $n > 1$  the sign of the bias depends on the quantity  $R$  and thus on the correlation. There are three cases to consider: zero correlation, negative correlation, and positive correlation. Since  $-1 \leq \rho_k \leq 1$ , examining the limiting values for the correlation will determine the range of the bias.

For positive correlation,  $0 \leq \rho_k \leq 1$ , the bias will be negative, ( $-\gamma_0 \leq \text{Bias} \leq 0$ ). Thus, the bias is negative if the correlation is positive, and the bias is positive if the correlation is negative. In the case of positive correlation,  $S^2/n$  underestimates the  $\text{Var}(\bar{X})$ . Thus, using  $S^2/n$  to form confidence intervals will make the confidence intervals too short. You will have unjustified confidence in the point estimate in this case. The true confidence will not be the desired  $1 - \alpha$ . Decisions based on positively correlated data will have a higher than planned risk of making an error based on the confidence interval.

One can easily show that for negative correlation,  $-1 \leq \rho_k \leq 0$ , the bias will be positive ( $0 \leq \text{Bias} \leq \gamma_0$ ). In the case of negatively correlated data,  $S^2/n$  over estimates the  $\text{Var}(\bar{X})$ . A confidence interval based on  $S^2/n$  will be too wide and the true quality of the estimate will be better than indicated. The true confidence coefficient will not be the desired  $1 - \alpha$ ; it will be greater than  $1 - \alpha$ .

Of the two cases, the positively correlated case is the more severe in terms of its effect on the decision making process; however, both are problems. Thus, the naive use

of  $S^2/n$  for dependent data is highly unwarranted. If you want to build confidence intervals on  $\bar{X}$  you need to find an unbiased estimator of the  $Var(\bar{X})$ .

The method of batch means provides a way to develop (at least approximately) an unbiased estimator for  $Var(\bar{X})$ . Assuming that you have a series of data point, the method of batch means method divides the data into subsequences of contiguous batches:

$$\begin{gathered} \underbrace{X_1, X_2, \dots, X_b}_{batch1} \cdots \underbrace{X_{b+1}, X_{b+2}, \dots, X_{2b}}_{batch2} \cdots \\ \underbrace{X_{(j-1)b+1}, X_{(j-1)b+2}, \dots, X_{jb}}_{batchj} \cdots \underbrace{X_{(k-1)b+1}, X_{(k-1)b+2}, \dots, X_{kb}}_{batchk} \end{gathered}$$

and computes the sample average of the batches. Let  $k$  be the number of batches each consisting of  $b$  observations, so that  $k = \lfloor n/b \rfloor$ . If  $b$  is not a divisor of  $n$  then the last  $(n - kb)$  data points will not be used. Define  $\bar{X}_j(b)$  as the  $j^{th}$  batch mean for  $j = 1, 2, \dots, k$ , where,

$$\bar{X}_j(b) = \frac{1}{b} \sum_{i=1}^b X_{(j-1)b+i}$$

Each of the batch means are treated like observations in the batch means series. For example, if the batch means are re-labeled as  $Y_j = \bar{X}_j(b)$ , the batching process simply produces another series of data,  $(Y_1, Y_2, Y_3, \dots, Y_k)$  which may be more like a random sample. To form a  $1 - \alpha\%$  confidence interval, you simply treat this new series like a random sample and compute approximate confidence intervals using the sample average and sample variance of the batch means series:

$$\begin{aligned} \bar{Y}(k) &= \frac{1}{k} \sum_{j=1}^k Y_j \\ S_b^2(k) &= \frac{1}{k-1} \sum_{j=1}^k (Y_j - \bar{Y})^2 \\ \bar{Y}(k) &\pm t_{\alpha/2, k-1} \frac{S_b(k)}{\sqrt{k}} \end{aligned}$$

Since the original  $X$ 's are covariance stationary, it follows that the resulting batch means are also covariance stationary. One can show, see (Alexopoulos and Seila, 1998), that the correlation in the batch means reduces as both the size of the batches,  $b$  and the number of data points,  $n$  increases. In addition, one can show that  $S_b^2(k)/k$  approximates  $Var(\bar{X})$  with error that reduces as both  $b$  and  $n$  increase towards infinity.

The basic difficulty with the batch means method is determining the batch size or alternatively the number of batches. Larger batch sizes are good for independence but reduce the number of batches, resulting in higher variance for the estimator.

(Schmeiser, 1982) performed an analysis that suggests that there is little benefit if the number of batches is larger than 30 and recommended that the number of batches remain in the range from 10 to 30. However, when trying to assess whether or not the batches are independent it is better to have a large number of batches ( $> 100$ ) so that tests on the lag-k correlation have better statistical properties.

There are a variety of procedures that have been developed that will automatically batch the data as it is collected, see for example (Fishman and Yarberry, 1997), (Steiger and Wilson, 2002), and Banks et al. (2005). has its own batching algorithm. The batching algorithm is described in Kelton et al. (2004) page 311. See also (Fishman, 2001) page 254 for an analysis of the effectiveness of the algorithm.

The discussion here is based on the description in Kelton et al. (2004). When the algorithm has recorded a sufficient amount of data, it begins by forming  $k = 20$  batches. As more data is collected, additional batches are formed until  $k = 40$  batches are collected. When 40 batches are formed, the algorithm collapses the number of batches back to 20, by averaging each pair of batches. This has the net effect of doubling the batch size. This process is repeated as more data is collected, thereby ensuring that the number of batches is between 20 and 39.

For time-persistent data, the approach requires that the data be discretized as previously discussed in the section on warm up period analysis. Then, the same batch method is applied to ensure between 20 and 39 batches. In addition, the process also computes the lag-1 correlation so that a test can be performed to check if the correlation is significant by testing the hypothesis that the batch means are uncorrelated using the following test statistic, see (Alexopoulos and Seila, 1998):

$$C = \sqrt{\frac{k^2 - 1}{k - 2}} \left[ \hat{\rho}_1 + \frac{[Y_1 - \bar{Y}]^2 + [Y_k - \bar{Y}]^2}{2 \sum_{j=1}^k (Y_j - \bar{Y})^2} \right]$$

$$\hat{\rho}_1 = \frac{\sum_{j=1}^{k-1} (Y_j - \bar{Y})(Y_{j+1} - \bar{Y})}{\sum_{j=1}^k (Y_j - \bar{Y})^2}$$

The hypothesis is rejected if  $C > z_\alpha$  for a given confidence level  $\alpha$ . If the batch means do not pass the test, *Correlated* is reported for the half-width on the statistical reports.

#### 0.40.4 Performing the Method of Batch Means

Performing the method of batch means in the JSL is relatively straight forward. The following assumes that a warm up period analysis has already been performed. Batches are formed during the simulation run and the confidence intervals are based on the batches. In this situation, the primary concern will be to determine the run length that will ensure a desired half-width on the confidence intervals. Both fixed sampling and sequential sampling methods can be applied.

The following code present the process to set up batching within the JSL for the Drive Through Pharmacy Model. The key class is the `statisticalBatchingElement` class, which

must be added to the `Model` (see line 9) prior to running the simulation. Then, starting at line 20 the batch summary statistical reporting is captured. The `StatisticalBatchingElement` has one key parameter which represents the interval used to discretize the time weighted variables. If no interval is supplied, then the algorithm ensures that the initial number of batches collected before applying the previously described batching algorithm is 512.

```

Simulation sim = new Simulation("Drive Through Pharmacy");
Model m = sim.getModel();
// add DriveThroughPharmacy to the main model
DriveThroughPharmacy driveThroughPharmacy = new DriveThroughPharmacy(m);
driveThroughPharmacy.setArrivalRS(new ExponentialRV(1.0));
driveThroughPharmacy.setServiceRS(new ExponentialRV(0.7));

// create the batching element for the simulation
StatisticalBatchingElement be = new StatisticalBatchingElement(m);

// set the parameters of the experiment
sim.setNumberOfReplications(1);
sim.setLengthOfReplication(1300000.0);
sim.setLengthOfWarmUp(100000.0);
System.out.println("Simulation started.");
sim.run();
System.out.println("Simulation completed.");

// get a StatisticReport from the batching element
StatisticReporter statisticReporter = be.getStatisticReporter();

// print out the report
System.out.println(statisticReporter.getHalfWidthSummaryReport());
//System.out.println(be.toString());

// use the name of a response to get a reference to a particular response variable
ResponseVariable systemTime = m.getResponseVariable("System Time");
// access the batch statistic from the batching element
BatchStatistic batchStatistic = be.getBatchStatistic(systemTime);
// get the actual batch mean values
double[] batchMeanArrayCopy = batchStatistic.getBatchMeanArrayCopy();
System.out.println(Arrays.toString(batchMeanArrayCopy));

```

The analysis performed to determine the warm up period should give you some information concerning how long to make this single run and how long to set it's warm up period. Assume that a warm up analysis has been performed using  $n_0$  replications of length  $T_e$  and that the analysis has indicated a warm up period of length  $T_w$ . Then, we can use this information is setting up the run length and warm up period for the single

replication experiment.

As previously discussed, the method of replication deletion spreads the risk of choosing a bad initial condition across multiple replications. The method of batch means relies on only one replication. If you were satisfied with the warm up period analysis based on  $n_0$  replications and you were going to perform replication deletion, then you are willing to throw away the observations contained in at least  $n_0 \times T_w$  time units and you are willing to use the data collected over  $n_0 \times (T_e - T_w)$  time units. Therefore, the warm up period for the single replication can be set at  $n_0 \times T_w$  and the run length can be set at  $n_0 \times T_e$ . For example, suppose your warm up analysis was based on the initial results of  $n_0 = 10$ ,  $T_e = 30000$ ,  $T_w = 10000$ . Thus, your starting run length would be  $n_0 \times T_e = 10 \times 30,000 = 300,000$  and the warm period will be  $n_0 \times T_w = 100,000$ . The following table show the results based on replication deletion

**Table 22:** Replication-Deletion Half-Width Summary report  $n = 10$ ,  $T_e = 30000$ ,  $T_w = 10000$

Response Name	$n$	$\bar{x}$	$hw$
PharmacyQ:Num In Q	10	1.656893	0.084475
PharmacyQ:Time In Q	10	1.659100	0.082517
NumBusy	10	0.699705	0.004851
Num in System	10	2.356598	0.088234
System Time	10	2.359899	0.085361

This table shows the results based on batching one long replication.

**Table 23:** Batch Means Summary Report  $T_e = 300000$ ,  $T_w = 100000$

Response Name	$n$	$\bar{x}$	$hw$
PharmacyQ:Num In Q	32	1.631132	0.047416
PharmacyQ:Time In Q	24	1.627059	0.055888
NumBusy	32	0.699278	0.004185
Num in System	32	2.330410	0.049853
System Time	24	2.324590	0.057446

Suppose now you want to ensure that the half-widths from a single replication are less than a given error bound. The half-widths reported by the simulation for a single replication are based on the *batch means*. You can get an approximate idea of how much to increase the length of the replication by using the half-width sample size determination formula.

$$n \cong n_0 \left( \frac{h_0}{h} \right)^2$$

In this case, you interpret  $n$  and  $n_0$  as the number of batches. From previous results, there were 32 batches for the time-weighted variables. Based on  $T_e = 300000$  and  $T_w = 100000$  there was a total of  $T_e - T_w = 200000$  time units of observed data. This means that each batch represents  $200,000 \div 32 = 6250$  time units. Using this information in the half-width based sample size formula with  $n_0 = 32$ ,  $h_0 = 0.049$ , and  $h = 0.02$ , for the number in the system, yields:

$$n \cong n_0 \frac{h_0^2}{h^2} = 32 \times \frac{(0.049)^2}{(0.02)^2} = 192 \text{ batches}$$

Since each batch in the run had 6250 time units, this yields the need for 1,200,000 time units of observations. Because of the warm up period, you therefore need to set  $T_e$  equal to  $(1,200,000 + 100,000 = 1,300,000)$ . Re-running the simulation yields the results shown in the following table. The results show that the half-width meets the desired criteria. This approach is approximate since you do not know how the observations will be batched when making the final run.

**Table 24:** Batch Means Summary Report  $T_e = 1,300,000$ ,  $T_w = 100000$

Response Name	$n$	$\bar{x}$	$hw$
NumBusy	32	0.698973	0.001652
Num in System	32	2.322050	0.019413
PharmacyQ:Num In Q	32	1.623077	0.018586
PharmacyQ:Time In Q	36	1.625395	0.017383
System Time	36	2.324915	0.017815

Rather than trying to fix the amount of sampling, you might instead try to use a sequential sampling technique that is based on the half-width computed during the simulation run. In order to do this, we need to create a specific observer that can stop the simulation when the half-width criteria is met. Currently, the JSL does not facilitate this approach.

Once the warm up period has been analyzed, performing infinite horizon simulations using the batch means method is relatively straight forward. A disadvantage of the batch means method is that it will be more difficult to use the statistical classical experimental design methods. If you are faced with an infinite horizon simulation, then you can use either the replication-deletion approach or the batch means method readily within the JSL. In either case, you should investigate if there may be any problems related to initialization bias. If you use the replication-deletion approach, you should play it safe when specifying the warm up period. Making the warm up period longer than you think it should be is better than replicating a poor choice. When performing an infinite horizon simulation based on one long run, you should make sure that your run length is long enough. A long run length can help to “wash out” the effects of initial condition bias.

Ideally, in the situation where you have to make many simulation experiments using different parameter settings of the same model, you should perform a warm up analysis for each design configuration. In practice, this is not readily feasible when there are a large number of experiments. In this situation, you should use your common sense to pick the design configurations (and performance measures) that you feel will most likely suffer from initialization bias. If you can determine long enough warm up periods for these configurations, the other configurations should be relatively safe from the problem by using the longest warm up period found.

There are a number of other techniques that have been developed for the analysis of infinite horizon simulations including the standardized time series method, the regenerative method, and spectral methods. An overview of these methods and others can be found in (Alexopoulos and Seila, 1998) and in (Law, 2007).

So far you have learned how to analyze the data from one design configuration. A key use of simulation is to be able to compare alternative system configurations and to assist in choosing which configurations are best according to the decision criteria. The next section discusses how to compare different system configurations.

## 0.41 Comparing System Configurations

The previous sections have concentrated on estimating the performance of a system through the execution of a single simulation model. The running of the model requires the specification of the input variables (e.g. mean time between arrivals, service distribution, etc.) and the structure of the model (e.g. FIFO queue, process flow, etc.) The specification of a set of inputs (variables and/or structure) represents a particular system configuration, which is then simulated to estimate performance. To be able to simulate design configurations, you may have to build different models or you may be able to use the same model supplied with different values of the program inputs. In either situation, you now have different design configurations that can be compared. This allows the performance of the system to be estimated under a wide-variety of controlled conditions. It is this ability to easily perform these what-if simulations that make simulation such a useful analysis tool.

Naturally, when you have different design configurations, you would like to know which configurations are better than the others. Since the simulations are driven by random variables, the outputs from each configuration (e.g.  $Y^1, Y^2$ ) are also random variables. The estimate of the performance of each system must be analyzed using statistical methods to ensure that the differences in performance are not simply due to sampling error. In other words, you want to be confident that one system is statistically better (or worse) than the other system.

### 0.41.1 Comparing Two Systems

The techniques for comparing two systems via simulation are essentially the same as that found in books that cover the statistical analysis of two samples (e.g. (Mont-

gomery and Runger, 2006)). This section begins with a review of these methods. Assume that samples from two different populations (system configurations) are available:

$X_{11}, X_{12}, \dots, X_{1n_1}$  a sample of size  $n_1$  from system configuration 1

$X_{21}, X_{22}, \dots, X_{2n_2}$  a sample of size  $n_2$  from system configuration 2

The samples represent a performance measure of the system that will be used in a decision regarding which system configuration is preferred. For example, the performance measure may be the average system throughput per day, and you want to pick the design configuration that has highest throughput.

Assume that each system configuration has an unknown population mean for the performance measure of interest,  $E[X_1] = \theta_1$  and  $E[X_2] = \theta_2$ . Thus, the problem is to determine, with some statistical confidence, whether  $\theta_1 < \theta_2$  or alternatively  $\theta_1 > \theta_2$ . Since the system configurations are different, an analysis of the situation of whether  $\theta_1 = \theta_2$  is of less relevance in this context.

Define  $\theta = \theta_1 - \theta_2$  as the mean difference in performance between the two systems. Clearly, if you can determine whether  $\theta > 0$  or  $\theta < 0$  you can determine whether  $\theta_1 < \theta_2$  or  $\theta_1 > \theta_2$ . Thus, it is sufficient to concentrate on the difference in performance between the two systems.

Given samples from two different populations, there are a number of ways in which the analysis can proceed based on different assumptions concerning the samples. The first common assumption is that the observations within each sample for each configuration form a random sample. That is, the samples represent independent and identically distributed random variables. Within the context of simulation, this can be easily achieved for a given system configuration by performing replications. For example, this means that  $X_{11}, X_{12}, \dots, X_{1n_1}$  are the observations from  $n_1$  replications of the first system configuration. A second common assumption is that both populations are normally distributed or that the central limit theorem can be used so that sample averages are at least approximately normal.

To proceed with further analysis, assumptions concerning the population variances must be made. Many statistics textbooks present results for the case of the population variance being known. In general, this is not the case within simulation contexts, so the assumption here will be that the variances associated with the two populations are unknown. Textbooks also present cases where it is assumed that the population variances are equal. Rather than making that assumption it is better to test a hypothesis regarding equality of population variances.

The last assumption concerns whether or not the two samples can be considered independent of each other. This last assumption is very important within the context of simulation. Unless you take specific actions to ensure that the samples will be independent, they will, in fact, be dependent because of how simulations use (re-use) the

same random number streams. The possible dependence between the two samples is not necessarily a bad thing. In fact, under certain circumstance it can be a good thing.

The following sections first presents the methods for analyzing the case of unknown variance with independent samples. Then, we focus on the case of dependence between the samples. Finally, how to use the JSL to do the work of the analysis will be illustrated.

#### 0.41.1.1 Analyzing Two Independent Samples

Although the variances are unknown, the unknown variances are either equal or not equal. In the situation where the variances are equal, the observations can be pooled when developing an estimate for the variance. In fact, rather than just assuming equal or not equal variances, you can (and should) use an F-test to test for the equality of variance. The F-test can be found in most elementary probability and statistics books (see (Montgomery and Runger, 2006)).

The decision regarding whether  $\theta_1 < \theta_2$  can be addressed by forming confidence intervals on  $\theta = \theta_1 - \theta_2$ . Let  $\bar{X}_1$ ,  $\bar{X}_2$ ,  $S_1^2$ , and  $S_2^2$  be the sample averages and sample variances based on the two samples ( $k = 1, 2$ ):

$$\bar{X}_k = \frac{1}{n_k} \sum_{j=1}^{n_k} X_{kj}$$

$$S_k^2 = \frac{1}{n_k - 1} \sum_{j=1}^{n_k} (X_{kj} - \bar{X}_k)^2$$

An estimate of  $\theta = \theta_1 - \theta_2$  is desired. This can be achieved by estimating the difference with  $\hat{D} = \bar{X}_1 - \bar{X}_2$ . To form confidence intervals on  $\hat{D} = \bar{X}_1 - \bar{X}_2$  an estimator for the variance of  $\hat{D} = \bar{X}_1 - \bar{X}_2$  is required. Because the samples are independent, the computation of the variance of the difference is:

$$Var(\hat{D}) = Var(\bar{X}_1 - \bar{X}_2) = \frac{\sigma_1^2}{n_1} + \frac{\sigma_2^2}{n_2}$$

where  $\sigma_1^2$  and  $\sigma_2^2$  are the unknown population variances. Under the assumption of equal variance,  $\sigma_1^2 = \sigma_2^2 = \sigma^2$ , this can be written as:

$$Var(\hat{D}) = Var(\bar{X}_1 - \bar{X}_2) = \frac{\sigma^2}{n_1} + \frac{\sigma^2}{n_2} = \sigma^2 \left( \frac{1}{n_1} + \frac{1}{n_2} \right)$$

where  $\sigma^2$  is the common unknown variance. A pooled estimator of  $\sigma^2$  can be defined as:

$$S_p^2 = \frac{(n_1 - 1)S_1^2 + (n_2 - 1)S_2^2}{n_1 + n_2 - 2}$$

Thus, a  $(1 - \alpha)\%$  confidence interval on  $\theta = \theta_1 - \theta_2$  is:

$$\hat{D} \pm t_{\alpha/2,v} s_p \sqrt{\frac{1}{n_1} + \frac{1}{n_2}}$$

where  $v = n_1 + n_2 - 2$ . For the case of unequal variances, an approximate  $(1 - \alpha)\%$  confidence interval on  $\theta = \theta_1 - \theta_2$  is given by:

$$\hat{D} \pm t_{\alpha/2,v} \sqrt{S_1^2/n_1 + S_2^2/n_2}$$

where

$$v = \left\lfloor \frac{\left(\frac{(S_1^2/n_1 + S_2^2/n_2)^2}{(S_1^2/n_1)^2 + (S_2^2/n_2)^2}\right) - 2}{\frac{n_1+1}{n_1+1} + \frac{n_2+1}{n_2+1}} \right\rfloor$$

Let  $[l, u]$  be the resulting confidence interval where  $l$  and  $u$  represent the lower and upper limits of the interval with by construction  $l < u$ . Thus, if  $u < 0$ , you can conclude with  $(1 - \alpha)\%$  confidence that  $\theta = \theta_1 - \theta_2 < 0$  (i.e. that  $\theta_1 < \theta_2$ ). If  $l > 0$ , you can conclude with  $(1 - \alpha)\%$  that  $\theta = \theta_1 - \theta_2 > 0$  (i.e. that  $\theta_1 > \theta_2$ ). If  $[l, u]$  contains 0, then no conclusion can be made at the given sample sizes about which system is better. This does not indicate that the system performance is the same for the two systems. You know that the systems are different. Thus, their performance will be different. This only indicates that you have not taken enough samples to detect the true difference. If sampling is relatively cheap, then you may want to take additional samples in order to discern an ordering between the systems.

Two configurations are under consideration for the design of an airport security checkpoint. A simulation model of each design was made. The replication values of the throughput per minute for the security station for each design are provided in the following table.

	Design 1	Design 2
1	10.98	8.93
2	8.87	9.82
3	10.53	9.27
4	9.40	8.50
5	10.10	9.63
6	10.28	9.55
7	8.86	9.30
8	9.00	9.31
9	9.99	9.23
10	9.57	8.88
11		8.05
12		8.74
$\bar{x}$	9.76	9.10
$s$	0.74	0.50

	Design 1	Design 2
$n$	10	12

Assume that the two simulations were run independently of each other, using different random numbers. Recommend the better design with 95% confidence.

According to the results:

$$\hat{D} = \bar{X}_1 - \bar{X}_2 = 9.76 - 9.1 = 0.66$$

In addition, we should test if the variances of the samples are equal. This requires an  $F$  test, with  $H_0 : \sigma_1^2 = \sigma_2^2$  versus  $H_1 : \sigma_1^2 \neq \sigma_2^2$ . Based on elementary statistics, the test statistic is:  $F_0 = S_1^2/S_2^2$ . The rejection criterion is to reject  $H_0$  if  $F_0 > f_{\alpha/2, n_1-1, n_2-1}$  or  $F_0 < f_{1-\alpha/2, n_1-1, n_2-1}$ , where  $f_{p,u,v}$  is the upper percentage point of the  $F$  distribution. Assuming a 0.01 significance level for the  $F$  test, we have  $F_0 = (0.74)^2/(0.50)^2 = 2.12$ . Since  $f_{0.005, 9, 11} = 5.54$  and  $f_{0.995, 9, 11} = 0.168$ , there is not enough evidence to conclude that the variances are different at the 0.01 significance level. The value of  $f_{p,u,v}$  can be determined in as F.INV.RT(p, u, v). Note also that  $f_{1-p,u,v} = 1/f_{p,v,u}$ . In R, the formula is  $f_{p,u,v} = qt(1-p, u, v)$ , since R provides the quantile function, not the upper right tail function.

Since the variances can be assumed equal, we can use the pooled variance, which is:

$$\begin{aligned} S_p^2 &= \frac{(n_1 - 1)S_1^2 + (n_2 - 1)S_2^2}{n_1 + n_2 - 2} \\ &= \frac{(10 - 1)(0.74)^2 + (12 - 1)(0.5)^2}{12 + 10 - 2} \\ &= 0.384 \end{aligned}$$

Thus, a  $(1 - 0.05)\%$  confidence interval on  $\theta = \theta_1 - \theta_2$  is:

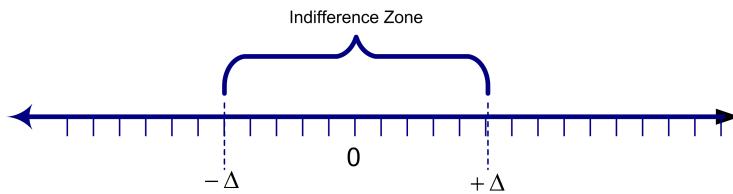
$$\begin{aligned} \hat{D} &\pm t_{\alpha/2, v} s_p \sqrt{\frac{1}{n_1} + \frac{1}{n_2}} \\ 0.66 &\pm t_{0.025, 20} (\sqrt{0.384}) \sqrt{\frac{1}{10} + \frac{1}{12}} \\ 0.66 &\pm (2.086)(0.6196)(0.428) \\ 0.66 &\pm 0.553 \end{aligned}$$

where  $v = n_1 + n_2 - 2 = 10 + 12 - 2 = 20$ . Since this results in an interval [0.10, 1.21] that does not contain zero, we can conclude that design 1 has the higher throughput with 95% confidence.

The confidence interval can assist in making decisions regarding relative performance of the systems from a *statistically significant* standpoint. However, if you make a conclusion about the ordering of the system, it still may not be practically significant. That is,

the difference in the system performance is statistically significant but the actual difference is of no practical use. For example, suppose you compare two systems in terms of throughput with resulting output  $\bar{X}_1 = 5.2$  and  $\bar{X}_2 = 5.0$  with the difference statistically significant. While the difference of 0.2 may be statistically significant, you might not be able to achieve this in the actual system. After all, you are making a decision based on a *model of the system* not on the real system. If the costs of the two systems are significantly different, you should prefer the cheaper of the two systems since there is no practical difference between the two systems. The fidelity of the difference is dependent on your modeling assumptions. Other modeling assumptions may overshadow such a small difference.

The notion of practical significance is model and performance measure dependent. One way to characterize the notion of practical significance is to conceptualize a zone of performance for which you are indifferent between the two systems.



**Figure 44:** Indifference Zone Concept

Figure 44 illustrates the concept of an indifference zone around the difference between the two systems. If the difference between the two systems falls in this zone, you are indifferent between the two systems (i.e. there is no practical difference).

Using the indifference zone to model the notion of practical significance, if  $u < -\Delta$ , you can conclude confidence that  $\theta_1 < \theta_2$ , and if  $l > \Delta$ , you can conclude with confidence that  $\theta_1 > \theta_2$ . If  $l$  falls within the indifference zone and  $u$  does not (or vice versa), then there is not enough evidence to make a confident conclusion. If  $[l, u]$  is totally contained within the indifference zone, then you can conclude with confidence that there is no practical difference between the two systems.

#### 0.41.1.2 Analyzing Two Dependent Samples

In this situation, continue to assume that the observations within a sample are independent and identically distributed random variables; however, the samples themselves are not independent. That is, assume that the  $(X_{11}, X_{12}, \dots, X_{1n_1})$  and  $(X_{21}, X_{22}, \dots, X_{2n_2})$  from the two systems are dependent.

For simplicity, suppose that the difference in the configurations can be implemented using a simple parameter change within the model. For example, the mean processing time is different for the two configurations. First, run the model to produce  $(X_{11}, X_{12}, \dots, X_{1n_1})$  for configuration 1. Then, change the parameter and re-executed the model to produce  $(X_{21}, X_{22}, \dots, X_{2n_2})$  for configuration 2.

Assuming that you did nothing with respect to the random number streams, the second configuration used the same random numbers that the first configuration used. Thus, the generated responses will be correlated (dependent). In this situation, it is convenient to assume that each system is run for the same number of replications, i.e.  $n_1 = n_2 = n$ . Since each replication for the two systems uses the same random number streams, the correlation between  $(X_{1,j}, X_{2,j})$  will not be zero; however, each pair will still be independent *across* the replications. The basic approach to analyzing this situation is to compute the difference for each pair:

$$D_j = X_{1j} - X_{2j} \text{ for } j = 1, 2, \dots, n$$

The  $(D_1, D_2, \dots, D_n)$  will form a random sample, which can be analyzed via traditional methods. Thus, a  $(1 - \alpha)\%$  confidence interval on  $\theta = \theta_1 - \theta_2$  is:

$$\begin{aligned}\bar{D} &= \frac{1}{n} \sum_{j=1}^n D_j \\ S_D^2 &= \frac{1}{n-1} \sum_{j=1}^n (D_j - \bar{D})^2 \\ \bar{D} \pm t_{\alpha/2, n-1} \frac{S_D}{\sqrt{n}}\end{aligned}$$

The interpretation of the resulting confidence interval  $[l, u]$  is the same as in the independent sample approach. This is the paired-t confidence interval presented in statistics textbooks.

Assume that the two simulations were run independently using common random numbers. Recommend the better design with 95% confidence.

According to the results:

$$\bar{D} = \bar{X}_1 - \bar{X}_2 = 50.88 - 48.66 = 2.22$$

Also, we have that  $S_D^2 = (0.55)^2$ . Thus, a  $(1 - 0.05)\%$  confidence interval on  $\theta = \theta_1 - \theta_2$  is:

$$\begin{aligned}\hat{D} \pm t_{\alpha/2, n-1} \frac{S_D}{\sqrt{n}} \\ 2.22 \pm t_{0.025, 9} \frac{0.55}{\sqrt{10}} \\ 2.22 \pm (2.261)(0.1739) \\ 2.22 \pm 0.0.393\end{aligned}$$

Since this results in an interval  $[1.827, 2.613]$  that does not contain zero, we can conclude that design 1 has the higher cost with 95% confidence.

Of the two approaches (independent versus dependent) samples, the latter is much more prevalent in simulation contexts. The approach is called the method of *common random numbers (CRN)* and is a natural by product of how most simulation languages handle their assignment of random number streams.

To understand why this method is the preferred method for comparing two systems, you need to understand the method's affect on the variance of the estimator. In the case of independent samples, the estimator of performance was  $\hat{D} = \bar{X}_1 - \bar{X}_2$ . Since

$$\begin{aligned}\bar{D} &= \frac{1}{n} \sum_{j=1}^n D_j \\ &= \frac{1}{n} \sum_{j=1}^n (X_{1j} - X_{2j}) \\ &= \frac{1}{n} \sum_{j=1}^n X_{1j} - \frac{1}{n} \sum_{j=1}^n X_{2j} \\ &= \bar{X}_1 - \bar{X}_2 \\ &= \hat{D}\end{aligned}$$

The two estimators are the same, when  $n_1 = n_2 = n$ ; however, their variances are not the same. Under the assumption of independence, computing the variance of the estimator yields:

$$V_{\text{IND}} = \text{Var}(\bar{X}_1 - \bar{X}_2) = \frac{\sigma_1^2}{n} + \frac{\sigma_2^2}{n}$$

Under the assumption that the samples are not independent, the variance of the estimator is:

$$V_{\text{CRN}} = \text{Var}(\bar{X}_1 - \bar{X}_2) = \frac{\sigma_1^2}{n} + \frac{\sigma_2^2}{n} - 2\text{cov}(\bar{X}_1, \bar{X}_2)$$

If you define  $\rho_{12} = \text{corr}(\bar{X}_1, \bar{X}_2)$ , the variance for the common random number situation is:

$$V_{\text{CRN}} = V_{\text{IND}} - 2\sigma_1\sigma_2\rho_{12}$$

Therefore, whenever there is positive correlation  $\rho_{12} > 0$  within the pairs we have that,  $V_{\text{CRN}} < V_{\text{IND}}$ .

If the variance of the estimator in the case of common random numbers is smaller than the variance of the estimator under independent sampling, then a *variance reduction* has been achieved. The method of common random numbers is called a variance reduction technique. If the variance reduction results in a confidence interval for  $\theta$  that is tighter than the independent case, the use of common random numbers should be

preferred. The variance reduction needs to be big enough to overcome any loss in the number of degrees of freedom caused by the pairing. When the number of replications is relatively large ( $n > 30$ ) this will generally be the case since the student-t value does not vary appreciatively for large degrees of freedom. Notice that the method of common random numbers might backfire and cause a variance increase if there is negative correlation between the pairs. An overview of the conditions under which common random numbers may work is given in (Law, 2007).

This notion of pairing the outputs from each replication for the two system configurations makes common sense. When trying to discern a difference, you want the two systems to experience the same randomness so that you can more readily infer that any difference in performance is due to the inherent difference between the systems and not caused by the random numbers.

In experimental design settings, this is called blocking on a factor. For example, if you wanted to perform an experiment to determine whether a change in a work method was better than the old method, you should use the same worker to execute both methods. If instead, you had different workers execute the methods, you would not be sure if any difference was due to the workers or to the proposed change in the method. In this context, the worker is the factor that should be blocked. In the simulation context, the random numbers are being blocked when using common random numbers.

#### 0.41.1.3 Using Common Random Numbers

The following explores how independent sampling and common random numbers can be implemented.

IN PROGRESS

#### 0.41.2 Multiple Comparisons

IN PROGRESS

## 0.42 Summary

This chapter described many of the statistical aspects of simulation that you will typically encounter in performing a simulation study. An important aspect of performing a correct simulation analysis is to understand the type of data associated with your performance measures (time-based versus observation-based) and how to collect/analyze such data. Then in your modeling you will be faced with specifying the time horizon of your simulation. Most situations involve finite-horizons, which are fortunately easy to analyze via the method of replications. This allows a random sample to be formed across replications and to analyze the simulation output via traditional statistical techniques.

In the case of infinite horizon simulations, things are more complicated. You must first analyze the effect of any warm up period on the performance measures and de-

cide whether you should use the method of replication-deletion or the method of batch means.

Since you often want to use simulation to make a recommendation concerning a design configuration, an analysis across system configurations must be carefully planned. When performing your analysis, you should consider how and when to use the method of common random numbers and you should consider the impact of common random numbers on how you analyze the simulation results.

Now that you have a solid understanding of how to program and model using the JSL and how to analyze your results, you are ready to explore the application of the JSL to additional modeling situations involving more complicated systems. The next chapter concentrates on queueing systems. These systems form the building blocks for modeling more complicated systems in manufacturing, transportation, and service industries.

# Miscellaneous Utility Classes

## .1 Reporting

We have already introduced the `StatisticReporter` class. Beside the ability to create a string representation of a half-width summary statistical report, the class has the ability to create a string representation of the report as a LaTeX table. Also found in the `jsl.utilities.reporting`<sup>21</sup> package is the `JSL` class. This class provides ready access to methods to create text files and to write to text files. It has a static field called `out` that is a `PrintWriter`. Thus, the field can be used globally to write out to a text file called `jslOutput.txt` that is written into the `jslOutput` directory.

```
// write string to file jslOutput.txt found in directory jslOutput
// JSL.out can be used just like System.out except text goes to a file
JSL.out.println("Hello World!");
```

One nice feature of using `JSL.out` is that the output can be turned off. The field `out` is actually an instance of `LogPrintWriter`, which provides very simple logging capabilities. By setting `out.OUTPUT_ON = false` all writing via `JSL.out` will not happen. When doing small programs, this can be useful for debugging and tracing; however, this is no substitute for using a full logger. The JSL supports logging through the SL4J<sup>22</sup> logging facade. While SL4J loggers can and should be used anywhere in your code, if you want a simple global logger that is already set up, you can use the `JSL.LOGGER` field.

In addition, the `JSL` class facilitates the creation of files and instances of `PrintWriter` that automatically catch the I/O exceptions through various static methods.

```
// make a file and write some data to it, file will be directory jslOutput, by default
PrintWriter writer = JSL.makePrintWriter("data", "csv");
```

There are methods to make instances of java's `File` class, make `PrintWriter` instances, get a path to the working directory and cause `JSL.out` to be redirected to the console.

<sup>21</sup><https://rossetti.git-pages.uark.edu/JSL-Documentation/jsl/utilities/reporting/package-summary.html>

<sup>22</sup><https://www.slf4j.org/index.html>

Please see the java docs for additional details.

## .2 JSLMath Class

The `JSLMath`<sup>23</sup> class is a singleton similar to java's `Math` class that adds some additional mathematical capabilities. Many of the methods provide basic functionality involving arrays. The methods include the following methods.

- `double getDefaultNumericalPrecision()` - returns the default numerical precision that can be expected on the machine
- `boolean equal(double a, double b)` - returns true if the two doubles are equal with respect to the default numerical precision
- `boolean equal(double a, double b, double precision)` - returns true if the two doubles are equal with respect to the specified precision
- `boolean within(double a, double b, double precision)` - returns true if the absolute difference between the double is within the specified precision
- `double factorial(int n)` - returns a numerically stable computed value of the factorial
- `double binomialCoefficient(int n, int k)` - returns a numerically stable computed value of the binomial coefficient
- `double logFactorial(int n)` - returns the natural logarithm of the factorial

The following methods work on arrays. While the `Statistic` class facilitates finding the minimum and maximum of an array of doubles, the `JSLMath` also provides this capability for `long` and `int` arrays.

- `int getIndexOfMin(long[] x)`
- `long getMin(long[] x)`
- `int getIndexOfMax(long[] x)`
- `long getMax(long[] x)`
- `int getIndexOfMin(int[] x)`
- `int getMin(int[] x)`
- `int getIndexOfMax(int[] x)`
- `int getMax(int[] x)`

`JSLMath` also provides for basic array manipulation via the following methods.

- `double getRange(double[] array)` - the difference between the largest and smallest element of the array.
- `double[] getMinMaxScaledArray(double[] array)` - rescales the array based the the range of the array.
- `double[] copyWithout(int index, double[] fromA)` - copies all the element of A except that one at element index
- `double[] addConstant(double[] a, double c)` - adds a constant to all elements of the array

---

<sup>23</sup> <https://rossetti.git-pages.uark.edu/JSL-Documentation/jsl/utilities/math/JSLMath.html>

- `double[] subtractConstant(double[] a, double c)` - subtracts a constant from all elements of the array
- `double[] multiplyConstant(double[] a, double c)` - multiplies all elements of the array by a constant
- `double[] divideConstant(double[] a, double c)` - divides all elements of the array by a constant
- `double[] multiplyElements(double[] a, double[] b)` - performs row element multiplication of the arrays
- `double getSumSquares(double[] array)` - computes the sum of squares for the array
- `double getSumSquareRoots(double[] array)` - computes the sum of the square roots of the elements of the array
- `double[] addElements(double[] a, double[] b)` - performs row element addition of the arrays
- `boolean compareArrays(double[] first, double[] second)` - returns true if element pairs are all the same in the two arrays.
- `<T> List<T> getElements(List objects, Class<T> targetClass)` - finds all elements in the list that are of the same class
- `countElements(List objects, Class targetClass)` - counts how many elements in the list are of the same class

### .3 The JSL Database

The JSL has the ability to save statistical data from the simulation runs into a relational database. Any relational database can be utilized; however, the JSL directly supports the embedded Apache Derby database management system (DBMS) as well as the PostgreSQL DBMS. The JSL database functionality is built upon the Java Object-Oriented Query (jooQ<sup>24</sup>) application programming interface, which abstracts one level above directly using Java Database Connectivity (JDBC) calls. Be aware that jooQ is freely available for use only with open source databases. The JSL library provides utilities to create databases, connect to databases, import data from Excel spreadsheets, export data to Excel spreadsheets as well as perform queries.

#### .3.1 The JSL Database Structure

The JSL database consists of six tables that capture information and data concerning the execution of a simulation and resulting statistical quantities. Figure 1 presents the database diagram for the JSL\_DB schema.

- SIMULATION\_RUN – contains information about the simulation runs that are contained within the database. Such information as the name of the simulation, model, and experiment are captured. In addition, time stamps of the start and end of the experiment, the number of replications, the replication length, the length of the warm up period and options concerning stream control.

---

<sup>24</sup><https://www.jooq.org/>

- MODEL\_ELEMENT contains information about the instances of ModelElement that were used within the execution of the simulation run. A model element has an identifier that is considered unique to the simulation run. That is, the simulation run ID and the model element ID are the primary key of this table. The name of the model element, its class type, the name and ID of its parent element are also held for each entity in MODEL\_ELEMENT. The parent/child relationship permits an understanding of the model element hierarchy that was present when the simulation executed.
- WITHIN REP STAT contains information about within replication statistical quantities associated with TimeWeighted and ResponseVariables from each replication of a set of replications of the simulation. The name, count, average, minimum, maximum, weighted sum, sum of weights, weighted sum of squares, last observed value, and last observed weight are all captured.
- WITHIN REP COUNTER STAT contains information about with replication observations associated with Counters used within the model. The name of the counter and its value at the end of the replication are captured for each replication of a set of replications of the simulation.
- ACROSS REP STAT contains information about the across replication statistics associated with TimeWeighted, ResponseVariable, and Counters within the model. Statistical summary information across the replications is automatically stored.
- BATCH STAT contains information about the batch statistics associated with TimeWeighted, ResponseVariable, and Counters within the model. Statistical summary information across the batches is automatically stored.

In addition to the base tables, the JSL database contains views of its underlying constructs to facilitate simpler data extraction. Figure 2 presents the pre-defined views for the JSL database. The views, in essence, reduce the amount of information to the most likely used sets of data for the across replication, batch, and within replication captured statistical quantities. In addition, the PW\_DIFF\_WITHIN REP VIEW holds all pairwise differences for every response variable, time weighted variable, or counter from across all experiments within the database. This view reports  $(A - B)$  for every within replication ending average, where A is a simulation run that has higher simulation ID than B and they represent an individual performance measure. From this view, pairwise statistics can be computed across all replications.

The information within the SIMULATION\_RUN, MODEL\_ELEMENT, WITHIN REP STAT, WITHIN REP COUNTER STAT tables are written to the database at the end of each replication. The ACROSS REP STAT and BATCH STAT tables are filled after the entire experiment is completed. Even though the ACROSS REP STAT table could be constructed directly from the data captured within the tables holding with replication data, this is not done. Instead, the across replication statistics are directly written from the simulation after all replications of an experiment are completed.

A JSLSDatabase instance is constructed to hold the data from any JSL simulation. As such, a simulation execution can have many observers and thus could have any number of JSLSDatabase instances that collect data from the execution. The most common case

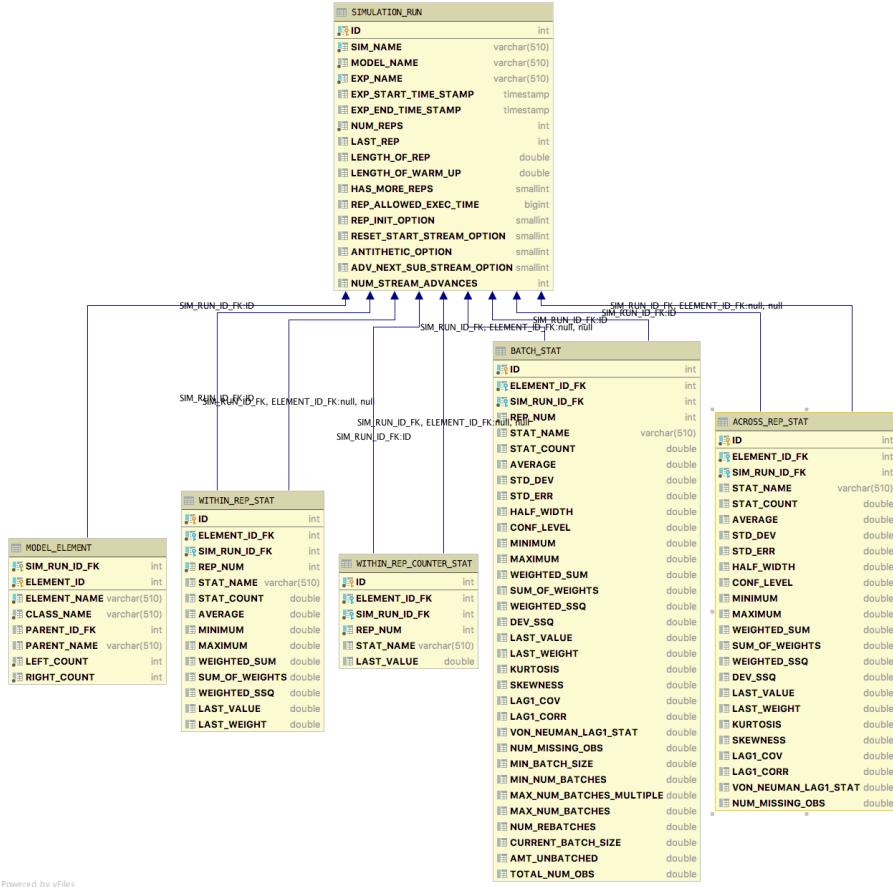


Figure 45: JSL Database Relational Diagram

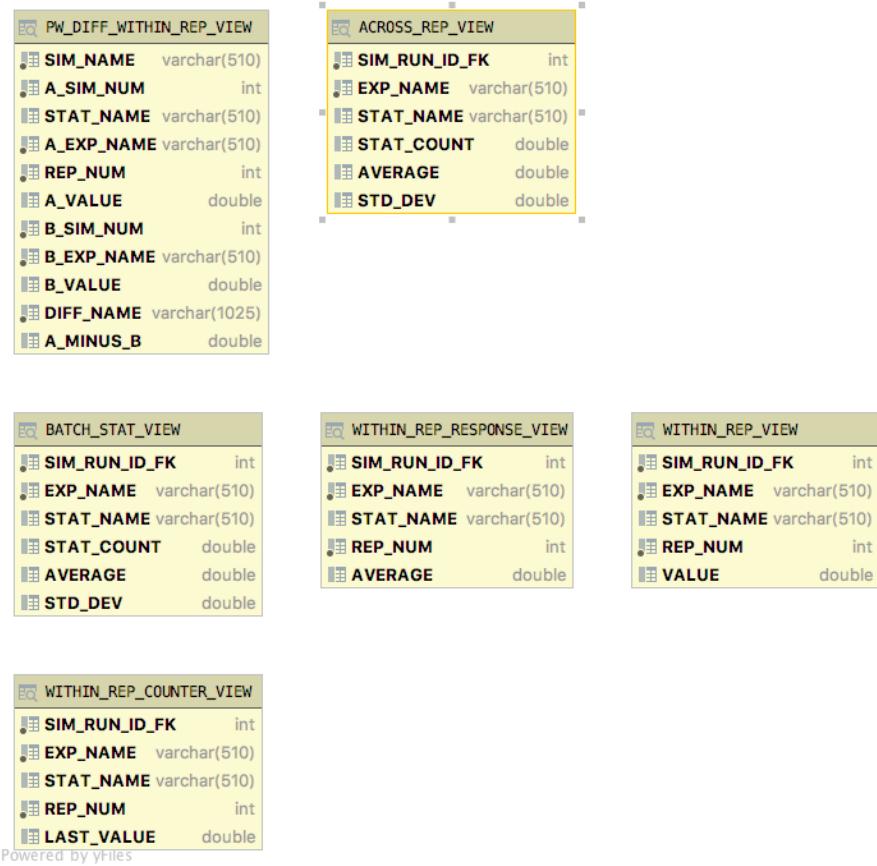
for multiple databases would be the use of an embedded database as well as a database that is stored on a database server.

### 3.2 Creating and Using a Default JSL Database

It is easy to have a database associated with a simulation. Just indicate that a default database should be created when constructing an instance of the Simulation class by providing “true” for the create default database parameter of the Simulation class:

```
Simulation sim = new Simulation("Drive Through Pharmacy", true);
```

This causes an instance of the class, JSLDATABASE, to be created and attached to the instance of the Simulation as an observer via an intermediate class called

**Figure 46:** JSL Database Views

JSLDatabaseObserver. The running of the simulation then causes data from the simulation to be stored in an Apache Derby embedded database that is contained within the directory, jslOutput/db. The name of the database will be based on the name provided for the simulation. For the previous code snippet, a database called JSldb\_DriveThroughPharmacy will be created. Any spaces in the name of the simulation are removed for the name of the database and JSldb\_ is appended. This naming is the default behavior for the default database. The database can be accessed just like any database. For example, IntelliJ's Datagrip tool can be used to connect to the database, execute queries, and export data.

One very important point to note is that constructing an instance of Simulation and providing “true” for the create default database option, **always** creates a new instance of the default database. This process involves deleting the previous default database and re-creating it without any data. If you do not want the previously created default database to be deleted and recreated, then you must take explicit steps to prevent this

from happening. The most obvious steps include:

- Do not call the constructor of Simulation with “true” for the create default database option. This will cause no new default JSL database to be created. Thus, a previous database instance will not be deleted. Or,
- Change the name of the simulation so that a differently named default database will be constructed.
- Move, copy, or rename the previously created database using operating system commands.

Note that if you execute **any** simulation that has the **same name** as a previously executed simulation and you were using the default database option, then the previous database will be deleted and recreated. This might cause you to lose previous simulation results. This behavior is the default because generally, when you re-run your simulation you want the results of that run to be written into the database.

### .3.3 Creating and Using JSL Databases

To better control the creation and use of an instance of the JSLDATABASE, I suggest that you consider creating your own instance rather than relying on the default database. The main reason to do this is if you plan to add results from multiple simulation executions to the **same** database. A database represented by a JSLDATABASE instance is just a database that has a JSL\_DB schema within it to hold data from a JSL simulation run. Provided that you can properly configure a JSL\_DB schema within a database, you can use **any** database as the backing store for JSL results. By default the JSL library facilitates the creation of embedded Apache Derby databases and PostgreSQL databases. In addition, the JSL library facilitates the creation of JSLDATABASE instances based on these two database management systems.

A number of methods are provided to create instances of a JSLDATABASE. Obviously, the constructor of JSLDATABASE can be used. The constructor has two parameters, an instance of an object that implements the DatabaseIfc interface and a Boolean parameter (clearDataOption) which controls whether or not all of the data within a possible JSL\_DB schema is removed when the JSLDATABASE instance is created. The default behavior is to not remove previous data. If the supplied DatabaseIfc interface instance does not already have a JSL\_DB schema, then one is created. If it already has a JSL\_DB schema, then the clear data option controls what happens to any previously stored data. Any of the methods of the DatabaseFactory class can be used to create an instance of the DatabaseIfc interface. Because you are most likely interested in directly making a JSLDATABASE instance, there are a number of static methods of the JSLDATABASE class that are provided for common use cases. For example, the following code snippet illustrates how to make an instance of a JSLDATABASE based on the embedded Derby database system. This will create a database named “MCB\_Db” within the jsOutput/db directory. If a database already exists with that name, then it will be deleted and a new database created.

```
JSLDatabase mcb_db = JSLDatabase.createEmbeddedDerbyJSLDatabase("MCB_Db");
```

If you want to connect to a previously created database, then use the methods of the DatabaseFactory class. For example, the following connects to an existing database found within the jslOutput/db directory and then supplies it to the JSLDatabase constructor. No previous data is lost via this process since we are only connecting to a database (not creating one).

```
DatabaseIfc database = DatabaseFactory.getEmbeddedDerbyDatabase("JSLDb_DriveThroughPharmacy");
// use the database as the backing database for the new JSLDatabase instance
JSLDatabase jslDatabase = new JSLDatabase(database);
```

The previously illustrated code examples only **create** an instance of JSLDatabase. The instance is **not** connected to an instance of Simulation and thus simulation results will not be added to the database unless additional steps are taken to hook up the JSLDatabase instance with instances of the Simulation class prior to running experiments.

The approach to connect a JSLDatabase instance with a Simulation instance involves creating an instance of JSLDatabaseObserver to monitor the simulation's execution. JSLDatabaseObserver has three required parameters in its constructor: 1) an instance of JSLDatabase, 2) and instance of Simulation, and 3) a boolean parameter (clearDataBeforeExperimentOption), which controls whether or not data from prior executions of the observed simulation will be cleared if they have the same simulation name and experiment name before each experiment (i.e. when the run() method is called on the simulation). The default value for the clear data before experiment option is true. Thus, data will be cleared from the database if the simulation name and experiment name are **the same**. If you do not want the data cleared, then set the option to false. However, if you then attempt to execute a simulation that has the same name and experiment name as one already stored in the database, an exception will be thrown. To prevent this exception change the name of the simulation or the experiment prior to running the simulation or decide to clear the data. The preferred method is to change the name of the experiment since this facilitates other analysis using JSL constructs.

Since creating and using a JSLDatabaseObserver is a common use case, the JSL library provides methods on the Simulation class to facilitate this:

- public JSLDatabaseObserver createJSLDatabaseObserver(String dbName)
- public JSLDatabaseObserver createJSLDatabaseObserver(JSLDatabase jslDatabase)
- public JSLDatabaseObserver createJSLDatabaseObserver(JSLDatabase jslDatabase, boolean clearDataBeforeExperiment)

The first method creates an embedded Derby database with the provided name in the jslOutput/db directory. The created JSLDatabaseObserver is returned and through that reference the underlying JSLDatabase can be accessed. The first of these three

methods creates a new database. The latter two only creates a new JSLServer instance and uses the supplied JSLServer. Thus, data will be added to the database. The class, UsingJSLDbExamples within the ex.running package illustrates many of the use cases presented.

As an illustration consider running a simulation multiple times within the same program execution but with different parameters. The following code illustrates how this might be achieved. The first line creates a simulation with a default database. Then the simulation is set up and executed. Notice that the experiment name is set prior to running the simulation. Then, the service time parameter is changed and the simulation is executed again. Notice that the experiment name was changed before the second simulation run. Data from both executions are captured within the default database. This code can be re-executed because a new default database is created each the Simulation instance is constructed. This clears all previous data and then all subsequent runs are captured because the experiment name is changed. If the name had not been changed before the second simulation run, then when the second simulation run executes the data from the first run will be cleared and the second run data captured because the clear data before experiment flag is true. If the clear data before experiment flag is false and you attempt to execute this code an exception would be thrown because there would be an attempt to enter data into the database that has the same simulation and experiment name. This would force you to change the name of the experiment before executing the second experiment. Because this code does change the names of the experiments, the clear data before experiment flag setting is irrelevant because the experiments have different names.

```
// make the simulation with a default database
Simulation sim = new Simulation("MultiRun", true);
// set the parameters of the experiment
sim.setNumberOfReplications(30);
sim.setLengthOfReplication(20000.0);
sim.setLengthOfWarmUp(5000.0);

// create the model element and attach it to the main model
DriverLicenseBureauWithQ driverLicenseBureauWithQ = new DriverLicenseBureauWithQ(sim.getModel());
sim.setExperimentName("1stRun");
// tell the simulation to run
System.out.println("Simulation started.");
sim.run();
System.out.println("Simulation completed.");

sim.setExperimentName("2ndRun");
driverLicenseBureauWithQ.setServiceDistributionInitialRandomSource(new ExponentialRV(0.7));

// tell the simulation to run
System.out.println("Simulation started.");
```

```

sim.run();
System.out.println("Simulation completed.");

// get the default JSL database
Optional<JSLSDatabase> db = sim.getDefaultJSLSDatabase();
if (db.isPresent()) {
    System.out.println("Printing across replication records");
    db.get().getAcrossRepStatRecords().format(System.out);
    System.out.println();
}

```

Once you have a database that contains the schema to hold JSL based data, you can continue to write results to that database as much as you want. If your database is on a server, then you can easily collect data from different simulation executions that occur on different computers by referencing the database on the server. Therefore, if you are running multiple simulation runs in parallel on different computers or in the “cloud”, you should be able to capture the data from the simulation runs into one database.

### .3.4 Querying the JSL Database

The JSL database is a database and thus it can be queried from within Java or from other programs. If you have an instance of the JSLSDatabase as in:

```

DatabaseIfc database = DatabaseFactory.getEmbeddedDerbyDatabase("JSLDb_DriveThroughPharmacy");
// use the database as the backing database for the new JSLSDatabase instance
JSLSDatabase jslDatabase = new JSLSDatabase(database);

```

You can extract information about the simulation run using the methods of the JSLSDatabase class. Since the underlying data is stored in a relational database, SQL queries can be used on the database. The discussion of writing and executing queries from within Java is beyond the scope of this discussion. To facilitate queries within Java, the JSL leverages the open source jooQ library. A few methods to be aware of include:

- `writeAllTablesAsCSV()` – writes all the tables to separate CSV files
- `writeDbToExcelWorkbook()` – writes all the tables and views to a single Excel work-book
- `getWithinRepViewRecords()` – returns a jooQ Result containing all the within replication statistical records
- `getWithinRepViewRecordsAsTablesawTable()` – returns a Tablesaw table representation of the within replication statistical data
- `getAcrossRepViewRecordsAsTablesawTable()` – returns a Tablesaw table representation of the across replication statistical data
- `getAcrossRepViewRecords()` – returns a jooQ Result containing all the across replication statistical records

- `getMultipleComparisonAnalyserFor(set of experiment name, response name)`  
 – returns an instance of the `MultipleComparisonAnalyzer` class in order to perform a multiple comparison analysis of a set of experiments on a specific response name.

The JSL database can be accessed via R or other software programs and additional analysis performed on JSL simulation data.

### 3.5 Additional Functionality

The functionality of the `JSL_DB` depends upon how `ResponseVariable`, `TimeWeighted`, and `Counter` instances are named within a JSL model. A JSL model is organized into a tree of `ModelElement` instances with the instance of the `Model` at the top of the tree. The `Model` instance for the simulation model contains instances of `ModelElement`, which are referred to as children of the parent model element. Each model element instance can have zero or more children, and those children can have children, etc. Each `ModelElement` instance must have a unique integer ID and a unique name. The unique integer ID is automatically provided when a `ModelElement` instance is created. The user can supply a name for a `ModelElement` instance when creating the instance. The name must be unique within the simulation model.

A recommended practice to ensure that model element names are unique is to use the name of the parent model element as part of the name. If the parent name is unique, then all children names will be unique relative to any other model elements. For example, in the following code `getName()` references the name of the current model element (an instance of `QueueingSystemWithQ`), which is serving as the parent for the children model element declared within the constructor.

```
public QueueingSystemWithQ(ModelElement parent, int numServers, RVariableIfc ad, RVariableIfc sd) {
    super(parent);

    myWaitingQ = new Queue(this, getName() + "_Q");
    myNumBusy = new TimeWeighted(this, 0.0, getName() + "_NumBusy");
    myNS = new TimeWeighted(this, 0.0, getName() + "_NS");
    mySysTime = new ResponseVariable(this, getName() + "_System Time");
```

The name supplied to the `TimeWeighted` and `ResponseVariable` constructors will cause the underlying statistic to have the same name. The statistic's name cannot be changed once it is set. The statistic name is important for referencing statistical data within the JSL database. One complicating factor involves using the JSL database to analyze the results from multiple simulation models. In order to more readily compare the results of the same performance measure between two different simulation models, the user should try to ensure that the names of the performance measures are the same. If the above recommended naming practice is used, the names of the statistics may depend on the order in which the model element instances are created and added to the model element hierarchy. If the model structure never changes between different

simulation models then this will not present an issue; however, if the structure of the model changes between two different simulation models (which can often be the case), the statistic names may be affected. If this issue causes problems, you can always name the desired output responses or counters exactly what you want it to be and use the same name in other simulation models.

Since the model element ID is assigned automatically based on the number of model elements created within the model, the model element numbers between two instances of the same simulation model will most likely be different. Thus, there is no guarantee that the IDs will be the same and using the model element ID as part of queries on the JSL database will have to take this into account. You can assume that the name of the underlying statistic is the same as its associated model element and since it is user definable, it is better suited for queries based on the JSL database.

# Generating Pseudo-Random Numbers and Random Variates

## LEARNING OBJECTIVES

- To be able to describe and use linear congruential pseudo-random number generation methods
- To be aware of current state of the art pseudo-random number generation methods
- To be able to define and use key terms in pseudo-random number generation methods such as streams, seeds, period, etc.
- To be able to explain the key issues in pseudo-random number testing
- To be able to derive and implement an inverse cumulative distribution function based random variate generation algorithm
- To be able to explain and implement the convolution algorithm for random variate generation
- To be able to explain and implement the acceptance rejection algorithm for random variate generation

Randomness in simulation is often modeled by using random variables and probability distributions. Thus, simulation languages require the ability to generate random variates. A random variate is an instance (or realization) of a random variable. In this section, you will learn how simulation languages allow for the generation of randomness. Generating randomness requires algorithms that permit sequences of numbers to act as the underlying source of randomness within the model. Then, given a good source of randomness, techniques have been established that permit the sequences to be transformed so that they can represent a wide variety of random variables (e.g. normal, Poisson, etc.). The algorithms that govern these procedures are described in the second part of this chapter.

## 4 Pseudo Random Numbers

This section indicates how uniformly distributed random numbers over the range from 0 to 1 are obtained within simulation programs. While commercial simulation packages provide substantial capabilities for generating random numbers, we still need to understand how this process works for the following reasons:

1. The random numbers within a simulation experiment might need to be controlled in order to take advantage of them to improve decision making.
2. In some situations, the commercial package does not have ready made functions for generating the desired random variables. In these situations, you will have to implement an algorithm to generate the random variates.

In addition, simulation is much broader than just using a commercial package. You can perform simulation in any computer language and spreadsheets. The informed modeler should know how the key inputs to simulation models are generated.

In simulation, large amount of cheap (easily computed) random numbers are required. In general, consider how random numbers might be obtained:

1. Dice, coins, colored balls
2. Specially designed electronic equipment
3. Algorithms

Clearly, within the context of computer simulation, it might be best to rely on algorithms; however, if an algorithm is used to generate the random numbers then they will not be truly random. For this reason, the random numbers that are used in computer simulation are called *pseudo random*.

**Definition .3** (Pseudo-Random Numbers). A sequence of pseudo-random numbers,  $U_i$ , is a deterministic sequence of numbers in  $(0, 1)$  having the same relevant statistical properties as a sequence of truly random  $U(0, 1)$  numbers.((Ripley, 1987))

A set of statistical tests are performed on the pseudo-random numbers generated from algorithms in order to indicate that their properties are not significantly different from a true set of  $U(0, 1)$  random numbers. The *algorithms* that produce pseudo-random numbers are called *random number generators*. In addition to passing a battery of statistical tests, the random number generators need to be fast and they need to be able to reproduce a sequence of numbers if and when necessary.

The following section discusses random number generation methods. The approach will be practical, with just enough theory to motivate future study of this area and to allow you to understand the important implications of random number generation. A more rigorous treatment of random number and random variable generation can be found such texts as (Fishman, 2006) and (Devroye, 1986).

### 4.1 Random Number Generators

Over the history of scientific computing, there have been a wide variety of techniques and algorithms proposed and used for generating pseudo-random numbers. A common technique that has been used (and is still in use) within a number of simulation environments is discussed in this text. Some new types of generators that have been recently adopted within many simulation environments, especially the one used within the JSL, will also be briefly discussed.

A linear congruential generator (LCG) is a recursive algorithm for producing a sequence of pseudo random numbers. Each new pseudo random number from the algorithm depends on the previous pseudo random number. Thus, a starting value called the *seed* is required. Given the value of the seed, the rest of the sequence of pseudo random numbers can be completely determined by the algorithm. The basic definition of an LCG is as follows

**Definition .4** (Linear Congruential Generator). A LCG defines a sequence of integers,  $R_0, R_1, \dots$  between 0 and  $m - 1$  according to the following recursive relationship:

$$R_{i+1} = (aR_i + c) \bmod m$$

where  $R_0$  is called the seed of the sequence,  $a$  is called the constant multiplier,  $c$  is called the increment, and  $m$  is called the modulus.  $(m, a, c, R_0)$  are integers with  $a > 0, c \geq 0, m > a, m > c, m > R_0$ , and  $0 \leq R_i \leq m - 1$ .

To compute a corresponding pseudo-random uniform number, we use

$$U_i = \frac{R_i}{m}$$

Notice that an LCG defines a sequence of integers and subsequently a sequence of real (rational) numbers that can be considered pseudo random numbers. Remember that pseudo random numbers are those that can “fool” a battery of statistical tests. The choice of the seed, constant multiplier, increment, and modulus, i.e. the parameters of the LCG, will determine the properties of the sequences produced by the generator. With properly chosen parameters, an LCG can be made to produce pseudo random numbers. To make this concrete, let’s look at a simple example of an LCG.

**Example .1** (Simple LCG Example). Consider an LCG with the following parameters ( $m = 8, a = 5, c = 1, R_0 = 5$ ). Compute the first nine values for  $R_i$  and  $U_i$  from the defined sequence.

\*\*\*

Let's first remember how to compute using the mod operator. The mod operator is defined as:

$$z = y \bmod m = y - m \left\lfloor \frac{y}{m} \right\rfloor$$

where  $\lfloor x \rfloor$  is the floor operator, which returns the greatest integer that is less than or equal to  $x$ . For example,

$$\begin{aligned} z &= 17 \bmod 3 \\ &= 17 - 3 \left\lfloor \frac{17}{3} \right\rfloor \\ &= 17 - 3 \lfloor 5.66 \rfloor \\ &= 17 - 3 \times 5 = 2 \end{aligned} \tag{8}$$

Thus, the mod operator returns the integer remainder (including zero) when  $y \geq m$  and  $y$  when  $y < m$ . For example,  $z = 6 \bmod 9 = 6 - 9 \lfloor \frac{6}{9} \rfloor = 6 - 9 \times 0 = 6$ .

Using the parameters of the LCG, the pseudo-random numbers are:

$$\begin{aligned} R_1 &= (5R_0 + 1) \bmod 8 = 26 \bmod 8 = 2 \Rightarrow U_1 = 0.25 \\ R_2 &= (5R_1 + 1) \bmod 8 = 11 \bmod 8 = 3 \Rightarrow U_2 = 0.375 \\ R_3 &= (5R_2 + 1) \bmod 8 = 16 \bmod 8 = 0 \Rightarrow U_3 = 0.0 \\ R_4 &= (5R_3 + 1) \bmod 8 = 1 \bmod 8 = 1 \Rightarrow U_4 = 0.125 \\ R_5 &= 6 \Rightarrow U_5 = 0.75 \\ R_6 &= 7 \Rightarrow U_6 = 0.875 \\ R_7 &= 4 \Rightarrow U_7 = 0.5 \\ R_8 &= 5 \Rightarrow U_8 = 0.625 \\ R_9 &= 2 \Rightarrow U_9 = 0.25 \end{aligned}$$

In the previous example, the  $U_i$  are simple fractions involving  $m = 8$ . Certainly, this sequence does not appear very random. The  $U_i$  can only take on rational values in the range,  $0, \frac{1}{m}, \frac{2}{m}, \frac{3}{m}, \dots, \frac{(m-1)}{m}$  since  $0 \leq R_i \leq m - 1$ . This implies that if  $m$  is small there will be gaps on the interval  $[0, 1)$ , and if  $m$  is large then the  $U_i$  will be more densely distributed on  $[0, 1)$ .

Notice that if a sequence generates the same value as a previously generated value then the sequence will repeat or cycle. An important property of a LCG is that it has a long cycle, as close to length  $m$  as possible. The length of the cycle is called the *period* of the LCG. Ideally the period of the LCG is equal to  $m$ . If this occurs, the LCG is said to achieve its full period. As can be seen in the example, the LCG is full period. Until recently, most computers were 32 bit machines and thus a common value for  $m$  is  $2^{31} - 1 = 2,147,483,647$ , which represents the largest integer number on a 32 bit computer using 2's complement integer arithmetic. This choice of  $m$  also happens to be a prime number, which leads to special properties.

A proper choice of the parameters of the LCG will allow desirable pseudo random number properties to be obtained. The following result due to (Hull and Dobell, 1962), see also (Law, 2007), indicates how to check if a LCG will have the largest possible cycle.

---

**Theorem.1** (LCG Theorem). *An LCG has full period if and only if the following three conditions hold: (1) The only positive integer that (exactly) divides both  $m$  and  $c$  is 1 (i.e.  $c$  and  $m$  have no common factors other than 1), (2) If  $q$  is a prime number that divides  $m$  then  $q$  should divide  $(a - 1)$ . (i.e.  $(a - 1)$  is a multiple of every prime number that divides  $m$ ), and (3) If 4 divides  $m$ , then 4 should divide  $(a - 1)$ . (i.e.  $(a - 1)$  is a multiple of 4 if  $m$  is a multiple of 4)*

---

Now, let's apply this theorem to the example LCG and check whether or not it should obtain full period. To apply the theorem, you must check if each of the three conditions holds for the generator.

- Condition 1:  $c$  and  $m$  have no common factors other than 1.  
The factors of  $m = 8$  are  $(1, 2, 4, 8)$ , since  $c = 1$  (with factor 1) condition 1 is true.
- Condition 2:  $(a - 1)$  is a multiple of every prime number that divides  $m$ . The first few prime numbers are  $(1, 2, 3, 5, 7)$ . The prime numbers,  $q$ , that divide  $m = 8$  are  $(q = 1, 2)$ . Since  $a = 5$  and  $(a - 1) = 4$ , clearly  $q = 1$  divides 4 and  $q = 2$  divides 4. Thus, condition 2 is true.
- Condition 3: If 4 divides  $m$ , then 4 should divide  $(a - 1)$ .  
Since  $m = 8$ , clearly 4 divides  $m$ . Also, 4 divides  $(a - 1) = 4$ . Thus, condition 3 holds.

Since all three conditions hold, the LCG achieves full period.

There are some simplifying conditions, see Banks et al. (2005), which allow for easier application of the theorem. For  $m = 2^b$ , ( $m$  a power of 2) and  $c$  not equal to 0, the longest possible period is  $m$  and can be achieved provided that  $c$  is chosen so that the greatest common factor of  $c$  and  $m$  is 1 and  $a = 4k + 1$  where  $k$  is an integer. The previous example LCG satisfies this situation.

For  $m = 2^b$  and  $c = 0$ , the longest possible period is  $(m/4)$  and can be achieved provided that the initial seed,  $R_0$  is odd and  $a = 8k + 3$  or  $a = 8k + 5$  where  $k = 0, 1, 2, \dots$ .

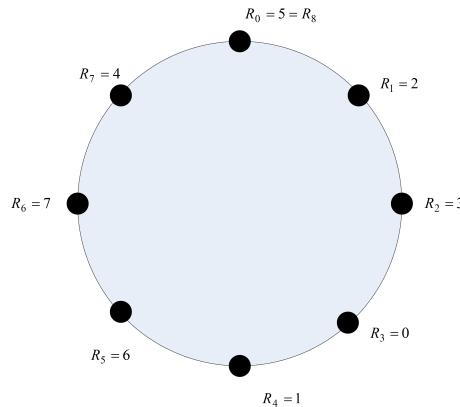
The case of  $m$  a prime number and  $c = 0$ , defines a special case of the LCG called a *prime modulus multiplicative linear congruential generator* (PMMLCG). For this case, the longest possible period is  $m - 1$  and can be achieved if the smallest integer,  $k$ , such that  $a^k - 1$  is divisible by  $m$  is  $m - 1$ .

Thirty-Two bit computers have been very common for over 20 years. In addition,  $2^{31} - 1 = 2,147,483,647$  is a prime number. Because of this,  $2^{31} - 1$  has been the choice for  $m$  with  $c = 0$ . Two common values for the multiplier,  $a$ , have been:

$$a = 630,360,016$$

$$a = 16,807$$

The latter of which was used within for a number of years. Notice that for PMMLCG's the full period cannot be achieved (because  $c = 0$ ), but with the proper selection of the multiplier, the next best period length of  $m - 1$  can be obtained. In addition, for this case  $R_0 \in \{1, 2, \dots, m - 1\}$  and thus  $U_i \in (0, 1)$ . The limitation of  $U_i \in (0, 1)$  is very useful when generating random variables from various probability distributions, since 0 cannot be realized. When using an LCG, you must supply a starting seed as an initial value for the algorithm. This seed determines the sequence that will come out of the generator when it is called within software. Since generators cycle, you can think of the sequence as a big circular list as indicated in Figure 47.



**Figure 47:** Sequence for Simple LCG Example

Starting with seed  $R_0 = 5$ , you get a sequence  $\{2, 3, 0, 1, 6, 7, 4, 5\}$ . Starting with seed,  $R_0 = 1$ , you get the sequence  $\{6, 7, 4, 5, 2, 3, 0, 1\}$ . Notice that these two sequences overlap with each other, but that the first half  $\{2, 3, 0, 1\}$  and the second half  $\{6, 7, 4, 5\}$  of the sequence do not overlap. If you only use 4 random numbers from each of these two *subsequences* then the numbers will not overlap. This leads to the definition of a stream:

**Definition .5** (Stream). The subsequence of random numbers generated from a given seed is called a random number stream.

You can take the sequence produced by the random number generator and divide it up into subsequences by associating certain seeds with streams. You can call the first subsequence stream 1 and the second subsequence stream 2, and so forth. Each stream

can be further divided into subsequences or sub-streams of non-overlapping random numbers.

In this simple example, it is easy to remember that stream 1 is defined by seed,  $R_0 = 5$ , but when  $m$  is large, the seeds will be large integer numbers, e.g.  $R_0 = 123098345$ . It is difficult to remember such large numbers. Rather than remember this huge integer, an assignment of stream numbers to seeds is made. Then, the sequence can be reference by its stream number. Naturally, if you are going to associate seeds with streams you would want to divide the entire sequence so that the number of non-overlapping random numbers in each stream is quite large. This ensures that as a particular stream is used that there is very little chance of continuing into the next stream. Clearly, you want  $m$  to be as large as possible and to have many streams that contain as large as possible number of non-overlapping random numbers. With today's modern computers even  $m$  is  $2^{31} - 1 = 2,147,483,647$  is not very big. For large simulations, you can easily run through all these random numbers.

Random number generators in computer simulation languages come with a default set of streams that divide the “circle” up into independent sets of random numbers. The streams are only independent if you do not use up all the random numbers within the subsequence. These streams allow the randomness associated with a simulation to be controlled. During the simulation, you can associate a specific stream with specific random processes in the model. This has the advantage of allowing you to check if the random numbers are causing significant differences in the outputs. In addition, this allows the random numbers used across alternative simulations to be better synchronized.

Now a common question for beginners using random number generators can be answered. That is, *If the simulation is using random numbers, why to I get the same results each time I run my program?* The corollary to this question is, *If I want to get different random results each time I run my program, how do I do it?* The answer to the first question is that the underlying random number generator is starting with the same seed each time you run your program. Thus, your program will use the same pseudo random numbers today as it did yesterday and the day before, etc. The answer to the corollary question is that you must tell the random number generator to use a different seed (or alternatively a different stream) if you want different invocations of the program to produce different results. The latter is not necessarily a desirable goal. For example, when developing your simulation programs, it is desirable to have repeatable results so that you can know that your program is working correctly. Unfortunately, many novices have heard about using the computer clock to “randomly” set the seed for a simulation program. This is a *bad* idea and very much not recommended in our context. This idea is more appropriate within a gaming simulation, in order to allow the human gamer to experience different random sequences.

Given current computing power, the previously discussed PMMLCGs are insufficient since it is likely that all the 2 billion or so of the random numbers would be used in performing serious simulation studies. Thus, a new generation of random number generators was developed that have extremely long periods. The random number generator described in L'Ecuyer et al. (2002) is one example of such a generator. It is based on

the combination of two multiple recursive generators resulting in a period of approximately  $3.1 \times 10^{57}$ . This is the same generator that is now used in many commercial simulation packages. The generator as defined in (Law, 2007) is:

$$\begin{aligned} R_{1,i} &= (1, 403, 580R_{1,i-2} - 810, 728R_{1,i-3})[\text{mod}(2^{32} - 209)] \\ R_{2,i} &= (527, 612R_{2,i-1} - 1, 370, 589R_{2,i-3})[\text{mod}(2^{32} - 22, 853)] \\ Y_i &= (R_{1,i} - R_{2,i})[\text{mod}(2^{32} - 209)] \\ U_i &= \frac{Y_i}{2^{32} - 209} \end{aligned}$$

The generator takes as its initial seed a vector of six initial values ( $R_{1,0}, R_{1,1}, R_{1,2}, R_{2,0}, R_{2,1}, R_{2,2}$ ). The first initially generated value,  $U_i$ , will start at index 3. To produce five pseudo random numbers using this generator we need an initial seed vector such as:  
 $\{R_{1,0}, R_{1,1}, R_{1,2}, R_{2,0}, R_{2,1}, R_{2,2}\} = \{12345, 12345, 12345, 12345, 12345, 12345\}$

Using the recursive equations, the resulting random numbers are as follows:

	i=3	i=4	i=5	i=6	i=7
$Z_{1,i-3} =$	12345	12345	12345	3023790853	3023790853
$Z_{1,i-2} =$	12345	12345	3023790853	3023790853	3385359573
$Z_{1,i-1} =$	12345	3023790853	3023790853	3385359573	1322208174
$Z_{2,i-3} =$	12345	12345	12345	2478282264	1655725443
$Z_{2,i-2} =$	12345	12345	2478282264	1655725443	2057415812
$Z_{2,i-1} =$	12345	2478282264	1655725443	2057415812	2070190165
$Z_{1,i} =$	3023790853	3023790853	3385359573	1322208174	2930192941
$Z_{2,i} =$	2478282264	1655725443	2057415812	2070190165	1978299747
$Y_i =$	545508589	1368065410	1327943761	3546985096	951893194
$U_i =$	0.127011122076	0.318527565471	0.309186015655	0.82584686312	0.221629915834

While it is beyond the scope of this text to explore the theoretical underpinnings of this generator, it is important to note that the use of this new generator is conceptually similar to that which has already been described. The generator allows multiple independent streams to be defined along with sub-streams.

The fantastic thing about this generator is the sheer size of the period. Based on their analysis, L'Ecuyer et al. (2002) state that it will be “approximately 219 years into the future before average desktop computers will have the capability to exhaust the cycle of the (generator) in a year of continuous computing.” In addition to the period length, the generator has an enormous number of streams, approximately  $1.8 \times 10^{19}$  with stream lengths of  $1.7 \times 10^{38}$  and sub-streams of length  $7.6 \times 10^{22}$  numbering at  $2.3 \times 10^{15}$  per stream. Clearly, with these properties, you do not have to worry about overlapping random numbers when performing simulation experiments. The generator was subjected to a rigorous battery of statistical tests and is known to have excellent statistical properties. The subject of modeling and testing different distribu-

tions is deferred to a separate part of this book.

## 5 Generating Random Variates from Distributions

In simulation, pseudo random numbers serve as the foundation for generating samples from probability distribution models. We will now assume that the random number generator has been rigorously tested and that it produces sequences of  $U_i \sim U(0, 1)$  numbers. We now want to take the  $U_i \sim U(0, 1)$  and utilize them to generate from probability distributions.

The realized value from a probability distribution is called a random variate. Simulations use many different probability distributions as inputs. Thus, methods for generating random variates from distributions are required. Different distributions may require different algorithms due to the challenges of efficiently producing the random variables. Therefore, we need to know how to generate samples from probability distributions. In generating random variates the goal is to produce samples  $X_i$  from a distribution  $F(x)$  given a source of random numbers,  $U_i \sim U(0, 1)$ . There are four basic strategies or methods for producing random variates:

1. Inverse transform or inverse cumulative distribution function (CDF) method
2. Convolution
3. Acceptance/Rejection
4. Mixture and Truncated Distributions

The following sections discuss each of these methods.

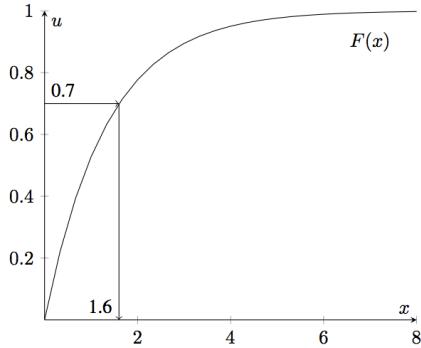
### 5.1 Inverse Transform Method

The inverse transform method is the preferred method for generating random variates provided that the inverse transform of the cumulative distribution function can be easily derived or computed numerically. The key advantage for the inverse transform method is that for every  $U_i$  use a corresponding  $X_i$  will be generated. That is, there is a one-to-one mapping between the pseudo-random number  $u_i$  and the generated variate  $x_i$ .

The inverse transform technique utilizes the inverse of the cumulative distribution function as illustrated in Figure 48, will illustrates simple cumulative distribution function. First, generate a number,  $u_i$  between 0 and 1 (along the  $U$  axis), then find the corresponding  $x_i$  coordinate by using  $F^{-1}(\cdot)$ . For various values of  $u_i$ , the  $x_i$  will be properly ‘distributed’ along the x-axis. The beauty of this method is that there is a one to one mapping between  $u_i$  and  $x_i$ . In other words, for each  $u_i$  there is a unique  $x_i$  because of the monotone property of the CDF.

The idea illustrated in Figure 48 is based on the following theorem.





**Figure 48:** Inverse Transform Method

**Theorem .2** (Inverse Transform). *Let  $X$  be a random variable with  $X \sim F(x)$ . Define another random variable  $Y$  such that  $Y = F(X)$ . That is,  $Y$  is determined by evaluating the function  $F(\cdot)$  at the value  $X$ . If  $Y$  is defined in this manner, then  $Y \sim U(0, 1)$ .*

The proof utilizes the definition of the cumulative distribution function to derive the CDF for  $Y$ .

$$\begin{aligned}
 F(y) &= P\{Y \leq y\} \\
 &= P\{F(X) \leq y\} \text{ substitute for } Y \\
 &= P\{F^{-1}(F(X)) \leq F^{-1}(y)\} \text{ apply inverse} \\
 &= P\{X \leq F^{-1}(y)\} \text{ definition of inverse} \\
 &= F(F^{-1}(y)) \text{ definition of CDF} \\
 &= y \text{ definition of inverse}
 \end{aligned}$$

Since  $P(Y \leq y) = y$  defines a  $U(0, 1)$  random variable, the proof is complete.

This result also works in reverse if you start with a uniformly distributed random variable then you can get a random variable with the distribution of  $F(x)$ . The idea is to generate  $U_i \sim U(0, 1)$  and then to use the inverse cumulative distribution function to transform the random number to the appropriately distributed random variate.

Let's assume that we have a function, `randU01()`, that will provide pseudo-random numbers on the range (0,1). Then, the following presents the pseudo-code for the inverse transform algorithm.

1.  $u = \text{randU01}()$
2.  $x = F^{-1}(u)$
3. **return**  $x$

Line 1 generates a uniform number. Line 2 takes the inverse of  $u$  and line 3 returns the random variate. The following example illustrates the inverse transform method for the exponential distribution.

The exponential distribution is often used to model the time until an event (e.g. time until failure, time until an arrival etc.) and has the following probability density function:

$$f(x) = \begin{cases} 0.0 & \text{if } x < 0 \\ \lambda e^{-\lambda x} & \text{if } x \geq 0 \end{cases}$$

with

$$E[X] = \frac{1}{\lambda}$$

$$Var[X] = \frac{1}{\lambda^2}$$


---

**Example .2** (Generating Exponential Random Variates). Consider a random variable,  $X$ , that represents the time until failure for a machine tool. Suppose  $X$  is exponentially distributed with an expected value of  $1.\overline{33}$ . Generate a random variate for the time until the first failure using a uniformly distributed value of  $u = 0.7$ .

\*\*\*

In order to solve this problem, we must first compute the CDF for the exponential distribution. For any value,  $b < 0$ , we have by definition:

$$F(b) = P\{X \leq b\} = \int_{-\infty}^b f(x) dx = \int_{-\infty}^b 0 dx = 0$$

For any value  $b \geq 0$ ,

$$\begin{aligned} F(b) &= P\{X \leq b\} = \int_{-\infty}^b f(x) dx \\ &= \int_{-\infty}^0 f(x) dx + \int_0^b f(x) dx \\ &= \int_0^b \lambda e^{\lambda x} dx = - \int_0^b e^{-\lambda x} (-\lambda) dx \\ &= -e^{-\lambda x} \Big|_0^b = -e^{-\lambda b} - (-e^0) = 1 - e^{-\lambda b} \end{aligned}$$

Thus, the CDF of the exponential distribution is:

$$F(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 - e^{-\lambda x} & \text{if } x \geq 0 \end{cases}$$

Now the inverse of the CDF can be derived by setting  $u = F(x)$  and solving for  $x = F^{-1}(u)$ .

$$\begin{aligned} u &= 1 - e^{-\lambda x} \\ x &= \frac{-1}{\lambda} \ln(1 - u) = F^{-1}(u) \end{aligned}$$

For Example .2, we have that  $E[X] = 1.33$ . Since  $E[X] = 1/\lambda$  for the exponential distribution, we have that  $\lambda = 0.75$ . Since  $u = 0.7$ , then the generated random variate,  $x$ , would be:

$$x = \frac{-1}{0.75} \ln(1 - 0.7) = 1.6053$$

Thus, if we let  $\theta = E[X]$ , the formula for generating an exponential random variate is simply:

$$x = \frac{-1}{\lambda} \ln(1 - u) = -\theta \ln(1 - u) \quad (9)$$

In the following pseudo-code, we assume that `randU01()` is a function that returns a uniformly distributed random number over the range (0,1).

1.  $u = \text{randU01}()$
2.  $x = \frac{-1}{\lambda} \ln(1 - u)$
3. return  $x$

Thus, the key to applying the inverse transform technique for generating random variates is to be able to first derive the cumulative distribution function (CDF) and then to derive its inverse function. It turns out that for many common distributions, the CDF and inverse CDF well known.

The uniform distribution over an interval  $(a, b)$  is often used to model situations where the analyst does not have much information and can only assume that the outcome is equally likely over a range of values. The uniform distribution has the following characteristics:

$$\begin{aligned} X &\sim \text{Uniform}(a, b) \\ f(x) &= \begin{cases} \frac{1}{b-a} & a \leq x \leq b \\ 0 & \text{otherwise} \end{cases} \\ E[X] &= \frac{a+b}{2} \\ \text{Var}[X] &= \frac{(b-a)^2}{12} \\ F(x) &= \begin{cases} 0.0 & x < a \\ \frac{x-a}{b-a} & a \leq x \leq b \\ 1.0 & x > b \end{cases} \end{aligned}$$

**Example .3** (Inverse CDF for Uniform Distribution). Consider a random variable,  $X$ , that represents the amount of grass clippings in a mower bag in pounds. Suppose the random variable is uniformly distributed between 5 and 35 pounds. Generate a random variate for the weight using a pseudo-random number of  $u = 0.25$ .

\*\*\*

To solve this problem, we must determine the inverse CDF algorithm for the  $U(a, b)$  distribution. The inverse of the CDF can be derived by setting  $u = F(x)$  and solving for  $x = F^{-1}(u)$ .

$$\begin{aligned} u &= \frac{x - a}{b - a} \\ u(b - a) &= x - a \\ x &= a + u(b - a) = F^{-1}(u) \end{aligned}$$

For the example, we have that  $a = 5$  and  $b = 35$  and  $u = 0.25$ , then the generated  $x$  would be:

$$F^{-1}(u) = x = 5 + 0.25 \times (35 - 5) = 5 + 7.5 = 12.5$$

Notice how the value of  $u$  is first scaled on the range  $(0, b - a)$  and then shifted to the range  $(a, b)$ . For the uniform distribution this transformation is linear because of the form of its  $F(x)$ .

1.  $u = \text{randU01}()$
2.  $x = a + u(b - a)$
3. return  $x$

For the previous distributions a closed form representation of the cumulative distribution function was available. If the cumulative distribution function can be inverted, then the inverse transform method can be easily used to generate random variates from the distribution. If no closed form analytical formula is available for the inverse cumulative distribution function, then often we can resort to numerical methods to implement the function. For example, the normal distribution is an extremely useful distribution and numerical methods have been devised to provide its inverse cumulative distribution function.

The inverse CDF method also works for discrete distributions. For a discrete random variable,  $X$ , with possible values  $x_1, x_2, \dots, x_n$  ( $n$  may be infinite), the probability distribution is called the probability mass function (PMF) and denoted:

$$f(x_i) = P(X = x_i)$$

where  $f(x_i) \geq 0$  for all  $x_i$  and

$$\sum_{i=1}^n f(x_i) = 1$$

The cumulative distribution function is

$$F(x) = P(X \leq x) = \sum_{x_i \leq x} f(x_i)$$

and satisfies,  $0 \leq F(x) \leq 1$ , and if  $x \leq y$  then  $F(x) \leq F(y)$ .

In order to apply the inverse transform method to discrete distributions, the cumulative distribution function can be searched to find the value of  $x$  associated with the given  $u$ . This process is illustrated in the following example.

---

**Example .4** (Discrete Empirical Distribution). Suppose you have a random variable,  $X$ , with the following discrete probability mass function and cumulative distribution function.

$x_i$	1	2	3	4
$f(x_i)$	0.4	0.3	0.2	0.1

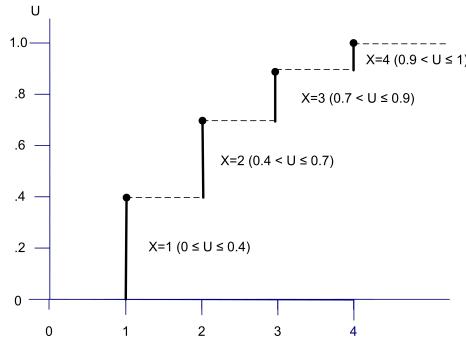
Plot the probability mass function and cumulative distribution function for this random variable. Then, develop an inverse cumulative distribution function for generating from this distribution. Finally, given  $u_1 = 0.934$  and  $u_2 = 0.1582$  are pseudo-random numbers, generate the two corresponding random variates from this PMF.

---

To solve this example, we must understand the functional form of the PMF and CDF, which are given as follows:

$$P\{X = x\} = \begin{cases} 0.4 & x = 1 \\ 0.3 & x = 2 \\ 0.2 & x = 3 \\ 0.1 & x = 4 \end{cases}$$

$$F(x) = \begin{cases} 0.0 & \text{if } x < 1 \\ 0.4 & \text{if } 1 \leq x < 2 \\ 0.7 & \text{if } 2 \leq x < 3 \\ 0.9 & \text{if } 3 \leq x < 4 \\ 1.0 & \text{if } x \geq 4 \end{cases}$$



**Figure 49:** Example Empirical CDF

Figure 49 illustrates the CDF for this discrete distribution.

Examining Figure 49 indicates that for any value of  $u_i$  in the interval,  $(0.4, 0.7]$  you get an  $x_i$  of 2. Thus, generating random numbers from this distribution can be accomplished by using the inverse of the cumulative distribution function.

$$F^{-1}(u) = \begin{cases} 1 & \text{if } 0.0 \leq u \leq 0.4 \\ 2 & \text{if } 0.4 < u \leq 0.7 \\ 3 & \text{if } 0.7 < u \leq 0.9 \\ 4 & \text{if } 0.9 < u \leq 1.0 \end{cases}$$

Suppose  $u_1 = 0.934$ , then by  $F^{-1}(u)$ ,  $x = 4$ . If  $u_2 = 0.1582$ , then  $x = 1$ . Thus, we use the inverse transform function to look up the appropriate  $x$  for a given  $u$ .

For a discrete distribution, given a value for  $u$ , pick  $x_i$ , such that  $F(x_{i-1}) < u \leq F(x_i)$  provides the inverse CDF function. Thus, for any given value of  $u$  the generation process amounts to a table lookup of the corresponding value for  $x_i$ . This simply involves searching until the condition  $F(x_{i-1}) < u \leq F(x_i)$  is true. Since  $F(x)$  is an increasing function in  $x$ , only the upper limit needs to be checked. The following presents these ideas in the form of an algorithm.

1.  $u = \text{randU010}$
2.  $i = 1$
3.  $x = x_i$
4. WHILE  $F(x) \leq u$
5.  $i = i + 1$
6.  $x = x_i$
7. END WHILE
8. RETURN  $x$

In the algorithm, if the test  $F(x) \leq u$  is true, the while loop moves to the next interval.

If the test failed,  $u > F(x_i)$  must be true. The while loop stops and  $x$  is the last value checked, which is returned. Thus, only the upper limit in the next interval needs to be tested. Other more complicated and possibly more efficient methods for performing this process are discussed in (Fishman, 2006) and (Ripley, 1987).

Using the inverse transform method, for discrete random variables, a Bernoulli random variate can be easily generated as shown in the following.

1.  $u = \text{randU01}()$
2. IF ( $u \leq p$ ) THEN
3.  $x=1$
4. ELSE
5.  $x=0$
6. END IF
7. RETURN  $x$

To generate *discrete uniform* random variables, the inverse transform method yields the following algorithm:

1.  $u = \text{randU01}()$
2.  $x = a + \lfloor (b - a + 1)u \rfloor$
3. return  $x$

Notice how the discrete uniform distribution inverse transform algorithm is different from the case of the continuous uniform distribution associated with Example .4. The inverse transform method also works for generating geometric random variables. Unfortunately, the geometric distribution has multiple definitions. Let  $X$  be the number of Bernoulli trials needed to get one success. Thus,  $X$  has range 1, 2, ... and distribution:

$$P\{X = k\} = (1 - p)^{k-1}p$$

We will call this the shifted geometric in this text. The algorithm to generate a shifted geometric random variables is as follows:

1.  $u = \text{randU01}()$
2.  $x = 1 + \left\lfloor \frac{\ln(1-u)}{\ln(1-p)} \right\rfloor$
3. return  $x$

Notice the use of the floor operator  $\lfloor \cdot \rfloor$ . If the geometric is defined as the number of failures  $Y = X - 1$  before the first success, then  $Y$  has range 0, 1, 2, ... and probability distribution:

$$P\{Y = k\} = (1 - p)^k p$$

We will call this distribution the geometric distribution in this text. The algorithm to generate a geometric random variables is as follows:

1.  $u = \text{randU01}()$
2.  $y = \left\lfloor \frac{\ln(1-u)}{\ln(1-p)} \right\rfloor$
3. return  $y$

The Poisson distribution is often used to model the number of occurrences within an interval time, space, etc. For example,

- the number of phone calls to a doctor's office in an hour
- the number of persons arriving to a bank in a day
- the number of cars arriving to an intersection in an hour
- the number of defects that occur within a length of item
- the number of typos in a book
- the number of pot holes in a mile of road

Assume we have an interval of real numbers, and that incidents occur at random throughout the interval. If the interval can be partitioned into sub-intervals of small enough length such that:

- The probability of more than one incident in a sub-intervals is zero
- The probability of one incident in a sub-intervals is the same for all intervals and proportional to the length of the sub-intervals, and
- The number of incidents in each sub-intervals is independent of other sub-intervals

Then, we have a Poisson distribution. Let the probability of an incident falling into a subinterval be  $p$ . Let there be  $n$  subintervals. An incident either falls in a subinterval or it does not. This can be considered a Bernoulli trial. Suppose there are  $n$  subintervals, then the number of incidents that fall in the large interval is a binomial random variable with expectation  $n * p$ . Let  $\lambda = np$  be a constant and keep dividing the main interval into smaller and smaller subintervals such that  $\lambda$  remains constant. To keep  $\lambda$  constant, increase  $n$ , and decrease  $p$ . What is the chance that  $x$  incidents occur in the  $n$  subintervals?

$$\binom{n}{x} \left(\frac{\lambda}{n}\right)^x \left(1 - \frac{\lambda}{n}\right)^{n-x}$$

Take the limit as  $n$  goes to infinity

$$\lim_{n \rightarrow \infty} \binom{n}{x} \left(\frac{\lambda}{n}\right)^x \left(1 - \frac{\lambda}{n}\right)^{n-x}$$

and we get the Poisson distribution:

$$P\{X = x\} = \frac{e^{-\lambda} \lambda^x}{x!} \quad \lambda > 0, \quad x = 0, 1, \dots$$

where  $E[X] = \lambda$  and  $Var[X] = \lambda$ . If a Poisson random variable represents the number of incidents in some interval, then the mean of the random variable must equal the expected number of incidents in the same length of interval. In other words, the units must match. When examining the number of incidents in a unit of time, the Poisson distribution is often written as:

$$P\{X(t) = x\} = \frac{e^{-\lambda t} (\lambda t)^x}{x!}$$

where  $X(t)$  is number of events that occur in  $t$  time units. This leads to an important relationship with the exponential distribution.

Let  $X(t)$  be a Poisson random variable that represents the number of arrivals in  $t$  time units with  $E[X(t)] = \lambda t$ . What is the probability of having no events in the interval from 0 to  $t$ ?

$$P\{X(t) = 0\} = \frac{e^{-\lambda t} (\lambda t)^0}{0!} = e^{-\lambda t}$$

This is the probability that no one arrives in the interval  $(0, t)$ . Let  $T$  represent the time until an arrival from any starting point in time. What is the probability that  $T > t$ ? That is, what is the probability that the time of the arrival is sometime after  $t$ ? For  $T$  to be bigger than  $t$ , we must not have anybody arrive before  $t$ . Thus, these two events are the same:  $\{T > t\} = \{X(t) = 0\}$ . Thus,  $P\{T > t\} = P\{X(t) = 0\} = e^{-\lambda t}$ . What is the probability that  $T \leq t$ ?

$$P\{T \leq t\} = 1 - P\{T > t\} = 1 - e^{-\lambda t}$$

This is the CDF of  $T$ , which is an exponential distribution. Thus, if  $T$  is a random variable that represents the time between events and  $T$  is exponential with mean  $1/\lambda$ , then, the number of events in  $t$  will be a Poisson process with  $E[X(t)] = \lambda t$ . Therefore, a method for generating Poisson random variates with mean  $\lambda$  can be derived by counting the number of events that occur before  $t$  when the time between events is exponential with mean  $1/\lambda$ .

**Example .5** (Generate Poisson Random Variates). Let  $X(t)$  represent the number of customers that arrive to a bank in an interval of length  $t$ , where  $t$  is measured in hours. Suppose  $X(t)$  has a Poisson distribution with mean rate  $\lambda = 4$  per hour. Use the following pseudo-random number (0.971, 0.687, 0.314, 0.752, 0.830) to generate a value of  $X(2)$ . That is, generate the number of arrivals in 2 hours.

Because of the relationship between the Poisson distribution and the exponential distribution, the time between events  $T$  will have an exponential distribution with mean  $0.25 = 1/\lambda$ . Thus, we have:

$$T_i = \frac{-1}{\lambda} \ln(1 - u_i) = -0.25 \ln(1 - u_i)$$

$$A_i = \sum_{k=1}^i T_k$$

where  $T_i$  represents the time between the  $i-1$  and  $i$  arrivals and  $A_i$  represents the time of the  $i^{th}$  arrival. Using the provided  $u_i$ , we can compute  $T_i$  (via the inverse transform method for the exponential distribution) and  $A_i$  until  $A_i$  goes over 2 hours.

$i$	$u_i$	$T_i$	$A_i$
1	0.971	0.881	0.881
2	0.687	0.290	1.171
3	0.314	0.094	1.265
4	0.752	0.349	1.614
5	0.830	0.443	2.057

Since the arrival of the fifth customer occurs after time 2 hours,  $X(2) = 4$ . That is, there were 4 customers that arrived within the 2 hours. This example is meant to be illustrative of one method for generating Poisson random variates. There are much more efficient methods that have been developed.

The inverse transform technique is general and is used when  $F^{-1}(\cdot)$  is closed form and easy to compute. It also has the advantage of using one  $U(0, 1)$  for each  $X$  generated, which helps when applying certain techniques that are used to improve the estimation process in simulation experiments. Because of this advantage many simulation languages utilize the inverse transform technique even if a closed form solution to  $F^{-1}(\cdot)$  does not exist by numerically inverting the function.

## .5.2 Convolution

Many random variables are related to each other through some functional relationship. One of the most common relationships is the convolution relationship. The distribution of the sum of two or more random variables is called the *convolution*. Let  $Y_i \sim G(y)$  be independent and identically distributed random variables. Let  $X = \sum_{i=1}^n Y_i$ . Then the distribution of  $X$  is said to be the  $n$ -fold convolution of  $Y$ . Some common random variables that are related through the convolution operation are:

- A binomial random variable is the sum of Bernoulli random variables.
- A negative binomial random variable is the sum of geometric random variables.

- An Erlang random variable is the sum of exponential random variables.
- A Normal random variable is the sum of other normal random variables.
- A chi-squared random variable is the sum of squared normal random variables.

The basic convolution algorithm simply generates  $Y_i \sim G(y)$  and then sums the generated random variables. Let's look at a couple of examples. By definition, a negative binomial distribution represents one of the following two random variables:

- The number of failures in sequence of Bernoulli trials before the  $r^{\text{th}}$  success, has range  $\{0, 1, 2, \dots\}$ .
- The number of trials in a sequence of Bernoulli trials until the  $r^{\text{th}}$  success, it has range  $\{r, r + 1, r + 2, \dots\}$

The number of failures before the  $r^{\text{th}}$  success, has range  $\{0, 1, 2, \dots\}$ . This is the sum of geometric random variables with range  $\{0, 1, 2, \dots\}$  with the same success probability.

If  $Y \sim NB(r, p)$  with range  $\{0, 1, 2, \dots\}$ , then

$$Y = \sum_{i=1}^r X_i$$

Where  $X_i \sim \text{Geometric}(p)$  with range  $\{0, 1, 2, \dots\}$ , and  $X_i$  can be generated via inverse transform with:

$$X_i = \left\lfloor \frac{\ln(1 - U_i)}{\ln(1 - p)} \right\rfloor$$

Note that  $\lfloor \cdot \rfloor$  is the floor function<sup>25</sup>.

If we have a negative binomial distribution that represents the number of trials until the  $r^{\text{th}}$  success, it has range  $\{r, r + 1, r + 2, \dots\}$ , in this text we call this a shifted negative binomial distribution.

A random variable from a “shifted” negative binomial distribution is the sum of shifted geometric random variables with range  $\{1, 2, 3, \dots\}$ . with same success probability. This geometric is called the shifted geometric distribution.

If  $T \sim NB(r, p)$  with range  $\{r, r + 1, r + 2, \dots\}$ , then

$$T = \sum_{i=1}^r X_i$$

Where  $X_i \sim \text{Shifted Geometric}(p)$  with range  $\{1, 2, 3, \dots\}$ , and  $X_i$  can be generated via inverse transform with:

---

<sup>25</sup>[https://en.wikipedia.org/wiki/Floor\\_and\\_ceiling\\_functions#:~:text=In%20the%20language%20of%20order,r,the%20integers%20into%20the%20reals.](https://en.wikipedia.org/wiki/Floor_and_ceiling_functions#:~:text=In%20the%20language%20of%20order,r,the%20integers%20into%20the%20reals.)

$$X_i = 1 + \left\lfloor \frac{\ln(1 - U_i)}{\ln(1 - p)} \right\rfloor$$

Notice that the relationship between these random variables as follows:

Let  $Y$  be the number of failures in a sequence of Bernoulli trials before the  $r^{th}$  success.

Let  $T$  be the number of trials in a sequence of Bernoulli trials until the  $r^{th}$  success,

Then, clearly,  $T = Y + r$ . Notice that we can generate  $Y$ , via convolution, as previously explained and just add  $r$  to get  $T$ .

$$Y = \sum_{i=1}^r X_i$$

Where  $X_i \sim \text{Geometric}(p)$  with range  $\{0, 1, 2, \dots\}$ , and  $X_i$  can be generated via inverse transform with:

$$X_i = \left\lfloor \frac{\ln(1 - U_i)}{\ln(1 - p)} \right\rfloor$$

TODO add a numerical example here for NB

As another example consider the requirement to generate random variables from an Erlang distribution. Suppose that  $Y_i \sim \text{Exp}(E[Y_i] = 1/\lambda)$ . That is,  $Y$  is exponentially distributed with rate parameter  $\lambda$ . Now, define  $X$  as  $X = \sum_{i=1}^r Y_i$ . One can show that  $X$  will have an Erlang distribution with parameters  $(r, \lambda)$ , where  $E[X] = r/\lambda$  and  $\text{Var}[X] = r/\lambda^2$ . Thus, an Erlang( $r, \lambda$ ) is an  $r$ -fold convolution of  $r$  exponentially distributed random variables with common mean  $1/\lambda$ .

**Example .6** (Generate Erlang Random Variates via Convolution). Use the following pseudo-random numbers  $u_1 = 0.35$ ,  $u_2 = 0.64$ ,  $u_3 = 0.14$ , generate a random variate from an Erlang distribution having parameters  $r = 3$  and  $\lambda = 0.5$ .

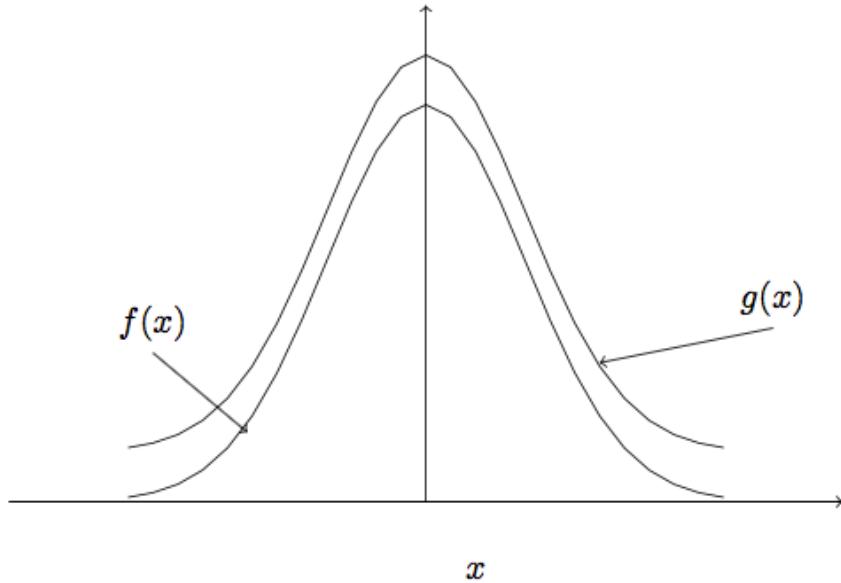
This requires generating 3 exponential distributed random variates each with  $\lambda = 0.5$  and adding them up.

$$\begin{aligned} y_1 &= \frac{-1}{\lambda} \ln(1 - u_1) = \frac{-1}{0.5} \ln(1 - 0.35) = 0.8616 \\ y_2 &= \frac{-1}{\lambda} \ln(1 - u_2) = \frac{-1}{0.5} \ln(1 - 0.64) = 2.0433 \\ y_3 &= \frac{-1}{\lambda} \ln(1 - u_3) = \frac{-1}{0.5} \ln(1 - 0.14) = 0.3016 \\ x &= y_1 + y_2 + y_3 = 0.8616 + 2.0433 + 0.3016 = 3.2065 \end{aligned}$$

Because of its simplicity, the convolution method is easy to implement; however, in a number of cases (in particular for a large value of  $n$ ), there are more efficient algorithms available.

### 5.3 Acceptance/Rejection

In the acceptance-rejection method, the probability density function (PDF)  $f(x)$ , from which it is desired to obtain a sample is replaced by a proxy PDF,  $w(x)$ , that can be sampled from more easily. The following illustrates how  $w(x)$  is defined such that the selected samples from  $w(x)$  can be used directly to represent random variates from  $f(x)$ . The PDF  $w(x)$  is based on the development of a majorizing function for  $f(x)$ . A majorizing function,  $g(x)$ , for  $f(x)$ , is a function such that  $g(x) \geq f(x)$  for  $-\infty < x < +\infty$ .



**Figure 50:** Concept of a Majorizing Function

Figure 50 illustrates the concept of a majorizing function for  $f(x)$ , which simply means a function that is bigger than  $f(x)$  everywhere.

In addition, to being a majorizing function for  $f(x)$ ,  $g(x)$  must have finite area. In other words,

$$c = \int_{-\infty}^{+\infty} g(x)dx$$

If  $w(x)$  is defined as  $w(x) = g(x)/c$  then  $w(x)$  will be a probability density function. The acceptance-rejection method starts by obtaining a random variate  $W$  from  $w(x)$ . Recall that  $w(x)$  should be chosen with the stipulation that it can be easily sampled, e.g. via the inverse transform method. Let  $U \sim U(0, 1)$ . The steps of the procedure are as provided in the following algorithm. The sampling of  $U$  and  $W$  continue until  $U \times g(W) \leq f(W)$  and  $W$  is returned. If  $U \times g(W) > f(W)$ , then the loop repeats.

1. REPEAT
2. Generate  $W \sim w(x)$
3. Generate  $U \sim U(0, 1)$
4. UNTIL  $(U \times g(W)) \leq f(W)$
5. RETURN  $W$

The validity of the procedure is based on deriving the cumulative distribution function of  $W$  given that the  $W = w$  was accepted,  $P\{W \leq x | W = w \text{ is accepted}\}$ .

The efficiency of the acceptance-rejection method is enhanced as the probability of rejection is reduced. This probability depends directly on the choice of the majorizing function  $g(x)$ . The acceptance-rejection method has a nice intuitive geometric connotation, which is best illustrated with an example.

---

**Example .7** (Acceptance-Rejection Example). Consider the following PDF over the range  $[-1, 1]$ . Develop an acceptance/rejection based algorithm for  $f(x)$ .

$$f(x) = \begin{cases} \frac{3}{4}(1-x^2) & -1 \leq x \leq 1 \\ 0 & \text{otherwise} \end{cases}$$

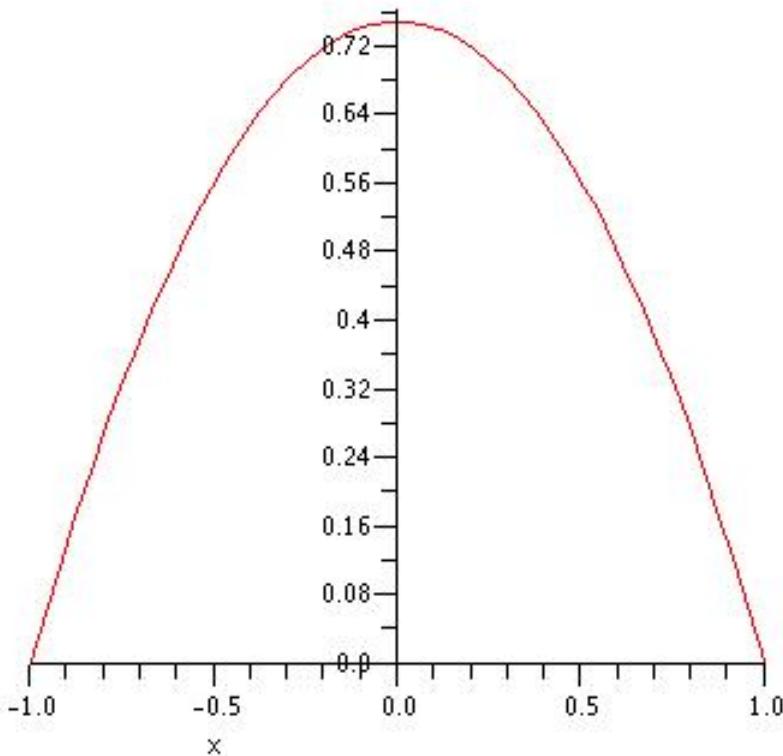

---

The first step in deriving an acceptance/rejection algorithm for the  $f(x)$  in Example .7 is to select an appropriate majorizing function. A simple method to choose a majorizing function is to set  $g(x)$  equal to the maximum value of  $f(x)$ . As can be seen from the plot of  $f(x)$  the maximum value of  $3/4$  occurs at  $x$  equal to 0. Thus, we can set  $g(x) = 3/4$ . In order to proceed, we need to construct the PDF associated with  $g(x)$ . Define  $w(x) = g(x)/c$  as the PDF. To determine  $c$ , we need to determine the area under the majorizing function:

$$c = \int_{-1}^1 g(x)dx = \int_{-1}^1 \frac{3}{4} dx = \frac{3}{2}$$

Thus,

$$w(x) = \begin{cases} \frac{1}{2} & -1 \leq x \leq 1 \\ 0 & \text{otherwise} \end{cases}$$



**Figure 51:** Plot of  $f(x)$

This implies that  $w(x)$  is a uniform distribution over the range from  $[-1, 1]$ . Based on the discussion of the continuous uniform distribution, a uniform distribution over the range from  $a$  to  $b$  can be generated with  $a + U(b - a)$ . Thus, for this case ( $a = -1$  and  $b = 1$ ), with  $b - a = 1 - (-1) = 2$ . The acceptance rejection algorithm is as follows:

1. Repeat
  - 1.1 Generate  $U_1 \sim U(0, 1)$
  - 1.2  $W = -1 + 2U_1$
  - 1.3 Generate  $U_2 \sim U(0, 1)$
  - 1.4  $f = \frac{3}{4}(1 - W^2)$
2. Until  $(U_2 \times \frac{3}{4} \leq f)$
3. Return  $W$

Steps 1.1 and 1.2 of generate a random variate,  $W$ , from  $w(x)$ . Note that  $W$  is in the range  $[-1, 1]$  (the same as the range of  $X$ ) so that step 1.4 is simply finding the height

associated with  $W$  in terms of  $f$ . Step 1.3 generates  $U_2$ . What is the range of  $U_2 \times \frac{3}{4}$ ? The range is  $[0, \frac{3}{4}]$ . Note that this range corresponds to the range of possible values for  $f$ . Thus, in step 2, a point between  $[0, \frac{3}{4}]$  is being compared to the candidate point's height  $f(W)$  along the vertical axis. If the point is under  $f(W)$ , the  $W$  is accepted; otherwise the  $W$  is rejected and another candidate point must be generated. In other words, if a point "under the curve" is generated it will be accepted.

As illustrated in the previous, the probability of acceptance is related to how close  $g(x)$  is to  $f(x)$ . The ratio of the area under  $f(x)$  to the area under  $g(x)$  is the probability of accepting. Since the area under  $f(x)$  is 1, the probability of acceptance,  $P_a$ , is:

$$P_a = \frac{1}{\int_{-\infty}^{+\infty} g(x)dx} = \frac{1}{c}$$

where  $c$  is the area under the majorizing function. For the example, the probability of acceptance is  $P_a = 2/3$ . Based on this example, it should be clear that the more  $g(x)$  is similar to the PDF  $f(x)$  the better (higher) the probability of acceptance. The key to efficiently generating random variates using the acceptance-rejection method is finding a suitable majorizing function over the same range as  $f(x)$ . In the example, this was easy because  $f(x)$  had a finite range. It is more challenging to derive an acceptance rejection algorithm for a probability density function,  $f(x)$ , if it has an infinite range because it may be more challenging to find a good majorizing function.

#### 5.4 Mixture Distributions, Truncated Distributions, and Shifted Random Variables

This section describes three random variate generation methods that build on the previously discussed methods. These methods allow for more flexibility in modeling the underlying randomness. First, let's consider the definition of a mixture distribution and then consider some examples.

**Definition .6** (Mixture Distribution). The distribution of a random variable  $X$  is a *mixture distribution* if the CDF of  $X$  has the form:

$$F_X(x) = \sum_{i=1}^k \omega_i F_{X_i}(x)$$

where  $0 < \omega_i < 1$ ,  $\sum_{i=1}^k \omega_i = 1$ ,  $k \geq 1$  and  $F_{X_i}(x)$  is the CDF of a continuous or discrete random variable  $X_i$ ,  $i = 1, \dots, k$ .

Notice that the  $\omega_i$  can be interpreted as a discrete probability distribution as follows. Let  $I$  be a random variable with range  $I \in \{1, \dots, k\}$  where  $P\{I = i\} = \omega_i$  is the

probability that the  $i^{th}$  distribution  $F_{X_i}(x)$  is selected. Then, the procedure for generating from  $F_X(x)$  is to randomly generate  $I$  from  $g(i) = \{I = i\} = \omega_i$  and then generate  $X$  from  $F_{X_I}(x)$ . The following algorithm presents this procedure.

1. Generate  $I \sim g(i)$
2. Generate  $X \sim F_{X_I}(x)$
3. return  $X$

Because mixture distributions combine the characteristics of two or more distributions, they provide for more flexibility in modeling. For example, many of the standard distributions that are presented in introductory probability courses, such as the normal, Weibull, lognormal, etc., have a single mode. Mixture distributions are often utilized for the modeling of data sets that have more than one mode.

As an example of a mixture distribution, we will discuss the hyper-exponential distribution. The hyper-exponential is useful in modeling situations that have a high degree of variability. The coefficient of variation is defined as the ratio of the standard deviation to the expected value for a random variable  $X$ . The coefficient of variation is defined as  $c_v = \sigma/\mu$ , where  $\sigma = \sqrt{\text{Var}[X]}$  and  $\mu = E[X]$ . For the hyper-exponential distribution  $c_v > 1$ . The hyper-exponential distribution is commonly used to model service times that have different (and mutually exclusive) phases. An example of this situation is paying with a credit card or cash at a checkout register. The following example illustrates how to generate from a hyper-exponential distribution.

**Example .8** (Hyper-Exponential Random Variate). Suppose the time that it takes to pay with a credit card,  $X_1$ , is exponentially distributed with a mean of 1.5 minutes and the time that it takes to pay with cash,  $X_2$ , is exponentially distributed with a mean of 1.1 minutes. In addition, suppose that the chance that a person pays with credit is 70%. Then, the overall distribution representing the payment service time,  $X$ , has an hyper-exponential distribution with parameters  $\omega_1 = 0.7$ ,  $\omega_2 = 0.3$ ,  $\lambda_1 = 1/1.5$ , and  $\lambda_2 = 1/1.1$ .

$$\begin{aligned}F_X(x) &= \omega_1 F_{X_1}(x) + \omega_2 F_{X_2}(x) \\F_{X_1}(x) &= 1 - \exp(-\lambda_1 x) \\F_{X_2}(x) &= 1 - \exp(-\lambda_2 x)\end{aligned}$$

Derive an algorithm for this distribution. Assume that you have two pseudo-random numbers,  $u_1 = 0.54$  and  $u_2 = 0.12$ , generate a random variate from  $F_X(x)$ .

In order to generate a payment service time,  $X$ , we can use the mixture distribution algorithm.

1. Generate  $u \sim U(0, 1)$
2. Generate  $v \sim U(0, 1)$

3. If ( $u \leq 7$ )
4.  $X = F_{X_1}^{-1}(v) = -1.5 \ln(1 - v)$
5. else
6.  $X = F_{X_2}^{-1}(v) = -1.1 \ln(1 - v)$
7. end if
8. return  $X$

Using  $u_1 = 0.54$ , because  $0.54 < 0.7$ , we have that  
 $X = F_{X_1}^{-1}(0.12) = -1.5 \ln(1 - 0.12) = 0.19175$

In the previous example, generating  $X$  from  $F_{X_i}(x)$  utilizes the inverse transform method for generating from the two exponential distribution functions. In general,  $F_{X_i}(x)$  for a general mixture distribution might be any distribution. For example, we might have a mixture of a Gamma and a Lognormal distribution. To generate from the individual  $F_{X_i}(x)$  one would use the most appropriate generation technique for that distribution. For example,  $F_{X_1}(x)$  might use inverse transform,  $F_{X_2}(x)$  might use acceptance/rejection,  $F_{X_3}(x)$  might use convolution, etc. This provides great flexibility in modeling and in generation.

In general, we may have situations where we need to control the domain over which the random variates are generated. For example, when we are modeling situations that involve time (as is often the case within simulation), we need to ensure that we do not generate negative values. Or, for example, it may be physically impossible to perform a task in a time that is shorter than a particular value. The next two generation techniques assist with modeling these situations.

A *truncated distribution* is a distribution derived from another distribution for which the range of the random variable is restricted. Truncated distributions can be either discrete or continuous. The presentation here illustrates the continuous case. Suppose we have a random variable,  $X$  with PDF,  $f(x)$  and CDF  $F(x)$ . Suppose that we want to constrain  $f(x)$  over interval  $[a, b]$ , where  $a < b$  and the interval  $[a, b]$  is a subset of the original support of  $f(x)$ . Note that it is possible that  $a = -\infty$  or  $b = +\infty$ . Constraining  $f(x)$  in this manner will result in a new random variable,  $X|a \leq X \leq b$ . That is, the random variable  $X$  given that  $X$  is contained in  $[a, b]$ . Random variables have a probability distribution. The question is what is the probability distribution of this new random variable and how can we generate from it.

This new random variable is governed by the conditional distribution of  $X$  given that  $a \leq X \leq b$  and has the following form:

$$f(x|a \leq X \leq b) = f^*(x) = \begin{cases} \frac{g(x)}{F(b)-F(a)} & a \leq x \leq b \\ 0 & \text{otherwise} \end{cases}$$

where

$$g(x) = \begin{cases} f(x) & a \leq x \leq b \\ 0 & \text{otherwise} \end{cases}$$

Note that  $g(x)$  is not a probability density. To convert it to a density, we need to find its area and divide by its area. The area of  $g(x)$  is:

$$\begin{aligned} \int_{-\infty}^{+\infty} g(x)dx &= \int_{-\infty}^a g(x)dx + \int_a^b g(x)dx + \int_b^{+\infty} g(x)dx \\ &= \int_{-\infty}^a 0dx + \int_a^b f(x)dx + \int_b^{+\infty} 0)dx \\ &= \int_a^b f(x)dx = F(b) - F(a) \end{aligned}$$

Thus,  $f^*(x)$  is simply a “re-weighting” of  $f(x)$ . The CDF of  $f^*(x)$  is:

$$F^*(x) = \begin{cases} 0 & \text{if } x < a \\ \frac{F(x)-F(a)}{F(b)-F(a)} & a \leq x \leq b \\ 0 & \text{if } b < x \end{cases}$$

This leads to a straight forward algorithm for generating from  $f^*(x)$  as follows:

1. Generate  $u \sim U(0, 1)$
2.  $W = F(a) + (F(b) - F(a))u$
3.  $X = F^{-1}(W)$
4. return  $X$

Lines 1 and 2 of the algorithm generate a random variable  $W$  that is uniformly distributed on  $(F(a), F(b))$ . Then, that value is used within the original distribution’s inverse CDF function, to generate a  $X$  given that  $a \leq X \leq b$ . Let’s look at an example.

**Example .9** (Generating a Truncated Random Variate). Suppose  $X$  represents the distance between two cracks in highway. Suppose that  $X$  has an exponential distribution with a mean of 10 meters. Generate a distance restricted between 3 and 6 meters using the pseudo-random number 0.23.

The CDF of the exponential distribution with mean 10 is:

$$F(x) = 1 - e^{-x/10}$$

Therefore  $F(3) = 1 - \exp(-3/10) = 0.259$  and  $F(6) = 0.451$ . The exponential distribution has inverse cumulative distribution function:

$$F^{-1}(u) = \frac{-1}{\lambda} \ln(1-u)$$

First, we generate a random number uniformly distributed between  $F(3)$  and  $F(6)$  using  $u = 0.23$ :

$$W = 0.259 + (0.451 - 0.259) \times 0.23 = 0.3032$$

Therefore, in order to generate the distance we have:

$$X = -10 \times \ln(1 - 0.3032) = 3.612$$

Lastly, we discuss shifted distributions. Suppose  $X$  has a given distribution  $f(x)$ , then the distribution of  $X + \delta$  is termed the shifted distribution and is specified by  $g(x) = f(x - \delta)$ . It is easy to generate from a shifted distribution, simply generate  $X$  according to  $F(x)$  and then add  $\delta$ .

**Example .10** (Generating a Shifted Weibull Random Variate Example). Suppose  $X$  represents the time to setup a machine for production. From past time studies, we know that it cannot take any less than 5.5 minutes to prepare for the setup and that the time after the 5.5 minutes is random with a Weibull distribution with shape parameter  $\alpha = 3$  and scale parameter  $\beta = 5$ . Using a pseudo-random number of  $u = 0.73$  generate a value for the time to perform the setup.

The Weibull distribution has a closed form cumulative distribution function:

$$F(x) = 1 - e^{-(x/\beta)^\alpha}$$

Thus, the inverse CDF function is:

$$F^{-1}(u) = \beta [-\ln(1-u)]^{1/\alpha}$$

Therefore to generate the setup time we have:

$$5.5 + 5 [-\ln(1 - 0.73)]^{1/3} = 5.5 + 5.47 = 10.97$$

Within this section, we illustrated the four primary methods for generating random variates 1) inverse transform, 2) convolution, 3) acceptance/rejection, and 4) mixture and truncated distributions. These are only a starting point for the study of random variate generation methods.

## .6 Summary

This section covered a number of important concepts used within simulation including:

- Generating pseudo-random numbers
- Generating random variates and processes

These topics provide a solid foundation for modeling random components within simulation models. Not only should you now understand how random numbers are generated you also know how to transform those numbers to allow the generation from a wide variety of probability distributions. To further your study of random variate generation, you should study the generation of multi-variate distributions.

## .7 Exercises

---

**Exercise .12.** The sequence of random numbers generated from a given seed is called a random number (a)\_\_\_\_\_.

---

**Exercise .13.** State three major methods of generating random variables from any distribution. (a)\_\_\_\_\_. (b)\_\_\_\_\_. (c)\_\_\_\_\_.

\*\*\*

**Exercise .14.** Consider the multiplicative congruential generator with ( $a = 13$ ,  $m = 64$ ,  $c = 0$ , and seeds  $X_0 = 1,2,3,4$ ). a) Using Theorem .1, does this generator achieve its maximum period for these parameters? b) Generate one period's worth of uniform random variables from each of the supplied seeds.

\*\*\*

**Exercise .15.** Consider the multiplicative congruential generator with ( $a = 11$ ,  $m = 64$ ,  $c = 0$ , and seeds  $X_0 = 1,2,3,4$ ). a) Using Theorem .1, does this generator achieve its maximum period for these parameters? b) Generate one period's worth of uniform random variables from each of the supplied seeds.

---

**Exercise .16.** Consider the linear congruential generator with ( $a = 11$ ,  $m = 16$ ,  $c = 5$ , and seed  $X_0 = 1$ ). a) Using Theorem .1, does this generator achieve its maximum period for these parameters? b) Generate 2 pseudo-random uniform numbers for this generator.

\*\*\*

**Exercise .17.** Consider the linear congruential generator with ( $a = 13$ ,  $m = 16$ ,  $c = 13$ , and seed  $X_0 = 37$ ). a) Using Theorem .1, does this generator achieve its maximum period for these parameters? b) Generate 2 pseudo-random uniform numbers for this generator.

\*\*\*

**Exercise .18.** Consider the linear congruential generator with ( $a = 8$ ,  $m = 10$ ,  $c = 1$ , and seed  $X_0 = 11$ ). a) Using Theorem .1, does this generator achieve its maximum period for these parameters? b) Generate 2 pseudo-random uniform numbers for this generator.

\*\*\*

**Exercise .19.** Consider the following discrete distribution of the random variable  $X$  whose probability mass function is  $p(x)$ .

$x$	0	1	2	3	4
$p(x)$	0.3	0.2	0.2	0.1	0.2

- a. Determine the CDF  $F(x)$  for the random variable,  $X$ .
  - b. Create a graphical summary of the CDF. See Example .4.
  - c. Create a look-up table that can be used to determine a sample from the discrete distribution,  $p(x)$ . See Example .4.
  - d. Generate 3 values of  $X$  using the following pseudo-random numbers  
 $u_1 = 0.943$ ,  $u_2 = 0.398$ ,  $u_3 = 0.372$
- 

**Exercise .20.** Consider the following uniformly distributed random numbers:

$U_1$	$U_2$	$U_3$	$U_4$	$U_5$	$U_6$	$U_7$	$U_8$
0.9396	0.1694	0.7487	0.3830	0.5137	0.0083	0.6028	0.8727

- a. Generate an exponentially distributed random number with a mean of 10 using the 1st random number.
  - b. Generate a random variate from a (12, 22) discrete uniform distribution using the 2nd random number.
-

**Exercise .21.** Consider the following uniformly distributed random numbers:

$U_1$	$U_2$	$U_3$	$U_4$	$U_5$	$U_6$	$U_7$	$U_8$
0.9559	0.5814	0.6534	0.5548	0.5330	0.5219	0.2839	0.3734

- a. Generate a uniformly distributed random number with a minimum of 12 and a maximum of 22 using  $U_8$ .
- b. Generate 1 random variate from an Erlang( $r = 2, \beta = 3$ ) distribution using  $U_1$  and  $U_2$
- c. The demand for magazines on a given day follows the following probability mass function:

$x$	40	50	60	70	80
$P(X = x)$	0.44	0.22	0.16	0.12	0.06

Using the supplied random numbers for this problem starting at  $U_1$ , generate 4 random variates from the probability mass function.

---

**Exercise .22.** Suppose that customers arrive at an ATM via a Poisson process with mean 7 per hour. Determine the arrival time of the first 6 customers using the following pseudo-random numbers via the inverse transformation method. Start with the first row and read across the table.

0.943	0.398	0.372	0.943	0.204	0.794
0.498	0.528	0.272	0.899	0.294	0.156
0.102	0.057	0.409	0.398	0.400	0.997

---

**Exercise .23.** The demand,  $D$ , for parts at a repair bench per day can be described by the following discrete probability mass function:

$D$	0	1	2
$p(D)$	0.3	0.2	0.5

Generate the demand for the first 4 days using the following sequence of  $U(0,1)$  random

numbers: 0.943, 0.398, 0.372, 0.943.

---

**Exercise .24.** The service times for a automated storage and retrieval system has a shifted exponential distribution. It is known that it takes a minimum of 15 seconds for any retrieval. The parameter of the exponential distribution is  $\lambda = 45$ . Generate two service times for this distribution using the following sequence of U(0,1) random numbers: 0.943, 0.398, 0.372, 0.943.

---

**Exercise .25.** The time to failure for a computer printer fan has a Weibull distribution with shape parameter  $\alpha = 2$  and scale parameter  $\beta = 3$ . Testing has indicated that the distribution is limited to the range from 1.5 to 4.5. Generate two random variates from this distribution using the following sequence of U(0,1) random numbers: 0.943, 0.398, 0.372, 0.943.

---

**Exercise .26.** The interest rate for a capital project is unknown. An accountant has estimated that the minimum interest rate will between 2% and 5% within the next year. The accountant believes that any interest rate in this range is equally likely. You are tasked with generating interest rates for a cash flow analysis of the project. Generate two random variates from this distribution using the following sequence of U(0,1) random numbers: 0.943, 0.398, 0.372, 0.943.

---

**Exercise .27.** Customers arrive at a service location according to a Poisson distribution with mean 10 per hour. The installation has two servers. Experience shows that 60% of the arriving customers prefer the first server. Start with the first row and read across the table determine the arrival times of the first three customers at each server.

0.943	0.398	0.372	0.943	0.204	0.794
0.498	0.528	0.272	0.899	0.294	0.156
0.102	0.057	0.409	0.398	0.400	0.997

---

**Exercise .28.** Consider the triangular distribution:

$$F(x) = \begin{cases} 0 & x < a \\ \frac{(x-a)^2}{(b-a)(c-a)} & a \leq x \leq c \\ 1 - \frac{(b-x)^2}{(b-a)(b-c)} & c < x \leq b \\ 1 & b < x \end{cases}$$

- a. Derive an inverse transform algorithm for this distribution. b. Using 0.943, 0.398, 0.372, 0.943, 0.204 generate 5 random variates from the triangular distribution with  $a = 2, c = 5, b = 10$ .
- 

**Exercise .29.** Consider the following probability density function:

$$f(x) = \begin{cases} \frac{3x^2}{2} & -1 \leq x \leq 1 \\ 0 & \text{otherwise} \end{cases}$$

- a. Derive an inverse transform algorithm for this distribution. b. Using 0.943, 0.398 generate two random variates from this distribution.
- 

**Exercise .30.** Consider the following probability density function:

$$f(x) = \begin{cases} 0.5x - 1 & 2 \leq x \leq 4 \\ 0 & \text{otherwise} \end{cases}$$

- a. Derive an inverse transform algorithm for this distribution. b. Using 0.943, 0.398 generate two random variates from this distribution.
- 

**Exercise .31.** Consider the following probability density function:

$$f(x) = \begin{cases} \frac{2x}{25} & 0 \leq x \leq 5 \\ 0 & \text{otherwise} \end{cases}$$

- a. Derive an inverse transform algorithm for this distribution. b. Using 0.943, 0.398 generate two random variates from this distribution.
-

**Exercise .32.** Consider the following probability density function:

$$f(x) = \begin{cases} \frac{2}{x^3} & x > 1 \\ 0 & x \leq 1 \end{cases}$$

- a. Derive an inverse transform algorithm for this distribution. b. Using 0.943, 0.398 generate two random variates from this distribution.
- 

**Exercise .33.** The times to failure for an automated production process have been found to be randomly distributed according to a Rayleigh distribution:

$$f(x) = \begin{cases} 2\beta^{-2}xe^{-(x/\beta)^2} & x > 0 \\ 0 & \text{otherwise} \end{cases}$$

- a. Derive an inverse transform algorithm for this distribution. b. Using 0.943, 0.398 generate two random variates from this distribution with  $\beta = 2.0$ .
- 

**Exercise .34.** Using the first two rows of random numbers from the following table, generate 5 random numbers from the negative binomial distribution with parameters ( $r = 4, p = 0.4$ ) using the convolution method the number of Bernoulli trials to get 4 successes.

0.943	0.398	0.372	0.943	0.204	0.794
0.498	0.528	0.272	0.899	0.294	0.156
0.102	0.057	0.409	0.398	0.400	0.997

---

**Exercise .35.** Using the first two rows of random numbers from the following table, generate 5 random numbers from the negative binomial distribution with parameters ( $r = 4, p = 0.4$ ) using a sequence of Bernoulli trials to get 4 successes.

0.943	0.398	0.372	0.943	0.204	0.794
0.498	0.528	0.272	0.899	0.294	0.156
0.102	0.057	0.409	0.398	0.400	0.997

---

**Exercise .36.** Suppose that the processing time for a job consists of two distributions. There is a 30% chance that the processing time is lognormally distributed with a mean of 20 minutes and a standard deviation of 2 minutes, and a 70% chance that the time is uniformly distributed between 10 and 20 minutes. Using the first row of random numbers the following table generate two job processing times. Hint:  $X \sim LN(\mu, \sigma^2)$  if and only if  $\ln(X) \sim N(\mu, \sigma^2)$ . Also, note that:

$$E[X] = e^{\mu + \sigma^2/2}$$

$$Var[X] = e^{2\mu + \sigma^2} (e^{\sigma^2} - 1)$$

---

0.943	0.398	0.372	0.943	0.204	0.794
0.498	0.528	0.272	0.899	0.294	0.156
0.102	0.057	0.409	0.398	0.400	0.997

---

**Exercise .37.** Suppose that the service time for a patient consists of two distributions. There is a 25% chance that the service time is uniformly distributed with minimum of 20 minutes and a maximum of 25 minutes, and a 75% chance that the time is distributed according to a Weibull distribution with shape of 2 and a scale of 4.5. Using the first row of random numbers from the following table generate the service time for two patients.

---

0.943	0.398	0.372	0.943	0.204	0.794
0.498	0.528	0.272	0.899	0.294	0.156
0.102	0.057	0.409	0.398	0.400	0.997

---

**Exercise .38.** If  $Z \sim N(0, 1)$ , and  $Y = \sum_{i=1}^k Z_i^2$  then  $Y \sim \chi_k^2$ , where  $\chi_k^2$  is a chi-squared random variable with  $k$  degrees of freedom. Using the first two rows of random numbers from the following table generate two  $\chi_5^2$  random variates.

---

0.943	0.398	0.372	0.943	0.204	0.794
0.498	0.528	0.272	0.899	0.294	0.156
0.102	0.057	0.409	0.398	0.400	0.997

---

**Exercise .39.** In the (a)\_\_\_\_\_ technique for generating random variates, you want the (b)\_\_\_\_\_ function to be as close as possible to

the distribution function that you want to generate from in order to ensure that the (c) \_\_\_\_\_ is as high as possible, thereby improving the efficiency of the algorithm.

---

**Exercise .40.** Prove that the acceptance-rejection method for continuous random variables is valid by showing that for any  $x$ ,

$$P\{X \leq x\} = \int_{-\infty}^x f(y)dy$$

Hint: Let E be the event that the acceptance occurs and use conditional probability.

---

**Exercise .41.** Consider the following probability density function:

$$f(x) = \begin{cases} \frac{3x^2}{2} & -1 \leq x \leq 1 \\ 0 & \text{otherwise} \end{cases}$$

- a. Derive an acceptance-rejection algorithm for this distribution. b. Using the first row of random numbers from the following table generate 2 random variates using your algorithm.

0.943	0.398	0.372	0.943	0.204	0.794
0.498	0.528	0.272	0.899	0.294	0.156
0.102	0.057	0.409	0.398	0.400	0.997

---

**Exercise .42.** This problem is based on (Cheng, 1977), see also (Ahrens and Dieter, 1972). Consider the gamma distribution:

$$f(x) = \beta^{-\alpha} x^{\alpha-1} \frac{e^{-x/\beta}}{\Gamma(\alpha)}$$

where  $x > 0$  and  $\alpha > 0$  is the shape parameter and  $\beta > 0$  is the scale parameter. In the case where  $\alpha$  is a positive integer, the distribution reduces to the Erlang distribution and  $\alpha = 1$  produces the negative exponential distribution.

Acceptance-rejection techniques can be applied to the cases of  $0 < \alpha < 1$  and  $\alpha > 1$ . For the case of  $0 < \alpha < 1$  see Ahrens and Dieter (1972). For the case of  $\alpha > 1$ , Cheng (1977) proposed the following majorizing function:

$$g(x) = \left[ \frac{4\alpha^\alpha e^{-\alpha}}{a\Gamma(\alpha)} \right] h(x)$$

where  $a = \sqrt{(2\alpha - 1)}$ ,  $b = \alpha^a$ , and  $h(x)$  is the resulting probability distribution function when converting  $g(x)$  to a density function:

$$h(x) = ab \frac{x^{a-1}}{(b + x^a)^2} \text{ for } x > 0$$

- a. Develop an inverse transform algorithm for generating from  $h(x)$
- b. Using the first two rows of random numbers from the following table, generate two random variates from a gamma distribution with parameters  $\alpha = 2$  and  $\beta = 10$  via the acceptance/rejection method.

0.943	0.398	0.372	0.943	0.204	0.794
0.498	0.528	0.272	0.899	0.294	0.156
0.102	0.057	0.409	0.398	0.400	0.997

**Exercise .43.** Parts arrive to a machine center with three drill presses according to a Poisson distribution with mean  $\lambda$ . The arriving customers are assigned to one of the three drill presses randomly according to the respective probabilities  $p_1$ ,  $p_2$ , and  $p_3$  where  $p_1 + p_2 + p_3 = 1$  and  $p_i > 0$  for  $i = 1, 2, 3$ . What is the distribution of the inter-arrival times to each drill press? Specify the parameters of the distribution. Suppose that  $p_1$ ,  $p_2$ , and  $p_3$  equal to 0.25, 0.45, and 0.3 respectively and that  $\lambda$  is equal to 12 per minute.

Using the first row of random numbers from the following table generate the first three arrival times.

0.943	0.398	0.372	0.943	0.204	0.794
0.498	0.528	0.272	0.899	0.294	0.156
0.102	0.057	0.409	0.398	0.400	0.997

---

# Probability Distribution Modeling

## LEARNING OBJECTIVES

- To be able model discrete distributions based on data
- To be able model continuous distributions based on data
- To be able to perform basic statistical tests on uniform pseudo-random numbers

When performing a simulation study, there is no substitution for actually observing the system and collecting the data required for the modeling effort. As outlined in Section 0.7, a good simulation methodology recognizes that modeling and data collection often occurs in parallel. That is, observing the system allows conceptual modeling which allows for an understanding of the input models that are needed for the simulation. The collection of the data for the input models allow further observation of the system and further refinement of the conceptual model, including the identification of additional input models. Eventually, this cycle converges to the point where the modeler has a well defined understanding of the input data requirements. The data for the input model must be collected and modeled.

Input modeling begins with data collection, probability, statistics, and analysis. There are many methods available for collecting data, including time study analysis, work sampling, historical records, and automatically collected data. Time study and work sampling methods are covered in a standard industrial engineering curriculum. Observing the time an operator takes to perform a task via a time study results in a set of observations of the task times. Hopefully, there will be sufficient observations for applying the techniques discussed in this section.

Work sampling is useful for developing the percentage of time associated with various activities. This sort of study can be useful in identifying probabilities associated with performing tasks and for validating the output from the simulation models. Historical records and automatically collected data hold promise for allowing more data to be collected, but also pose difficulties related to the quality of the data collected. In any of the above mentioned methods, the input models will only be as good as the data and processes used to collect the data.

One especially important caveat for new simulation practitioners: do not rely on the people in the system you are modeling to correctly collect the data for you. If you do rely on them to collect the data, you must develop documents that clearly define what data is needed and how to collect the data. In addition, you should train them to collect the data using the methods that you have documented. Only through careful instruction and control of the data collection processes will you have confidence in your input modeling.

A typical input modeling process includes the following procedures:

1. Documenting the process being modeled: Describe the process being modeled and define the random variable to be collected. When collecting task times, you should pay careful attention to clearly defining when the task starts and when the task ends. You should also document what triggers the task.
2. Developing a plan for collecting the data and then collect the data: Develop a sampling plan, describe how to collect the data, perform a pilot run of your plan, and then collect the data.
3. Graphical and statistical analysis of the data: Using standard statistical analysis tools you should visually examine your data. This should include such plots as a histogram, a time series plot, and an auto-correlation plot. Again, using statistical analysis tools you should summarize the basic statistical properties of the data, e.g. sample average, sample variance, minimum, maximum, quartiles, etc.
4. Hypothesizing distributions: Using what you have learned from steps 1 - 3, you should hypothesize possible distributions for the data.
5. Estimating parameters: Once you have possible distributions in mind you need to estimate the parameters of those distributions so that you can analyze whether the distribution provides a good model for the data. With current software this step, as well as steps 3, 4, and 6, have been largely automated.
6. Checking goodness of fit for hypothesized distributions: In this step, you should assess whether or not the hypothesized probability distributions provide a good fit for the data. This should be done both graphically (e.g. histograms, P-P plots and Q-Q plots) and via statistical tests (e.g. Chi-Squared test, Kolmogorov-Smirnov Test). As part of this step you should perform some sensitivity analysis on your fitted model.

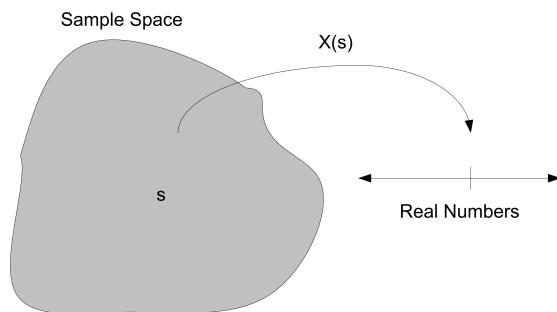
During the input modeling process and after it is completed, you should document your process. This is important for two reasons. First, much can be learned about a system simply by collecting and analyzing data. Second, in order to have your simulation model accepted as useful by decision makers, they must *believe* in the input models. Even non-simulation savvy decision makers understand the old adage “Garbage In = Garbage Out”. The documentation helps build credibility and allows you to illustrate how the data was collected.

The following section provides a review of probability and statistical concepts that are

useful in distribution modeling.

## .8 Random Variables and Probability Distributions

This section discusses some concepts in probability and statistics that are especially relevant to simulation. These will serve you well as you model randomness in the inputs of your simulation models. When an input process for a simulation is stochastic, you must develop a probabilistic model to characterize the process's behavior over time. Suppose that you are modeling the service times in the pharmacy example. Let  $X_i$  be a random variable that represents the service time of the  $i^{th}$  customer. As shown in Figure 52, a random variable is a function that assigns a real number to each outcome,  $s$ , in a random process that has a set of possible outcomes,  $S$ .



**Figure 52:** Random Variables Map Outcomes to Real Numbers

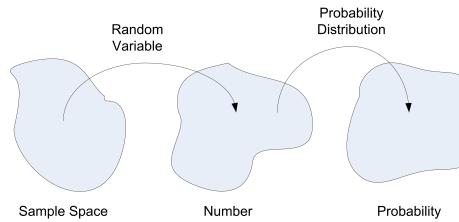
In this case, the process is the service times of the customers and the outcomes are the possible values that the service times can take on, i.e. the range of possible values for the service times.

The determination of the range of the random variable is part of the modeling process. For example, if the range of the service time random variable is the set of all possible positive real numbers, i.e.  $X_i \in \mathbb{R}^+$  or in other words,  $X_i \geq 0$ , then the service time should be modeled as a *continuous* random variable.

Suppose instead that the service time can only take on one of five discrete values 2, 5, 7, 8, 10, then the service time random variable should be modeled as a *discrete* random variable. Thus, the first decision in modeling a stochastic input process is to appropriately define a random variable and its possible range of values.

The next decision is to characterize the probability distribution for the random variable. As indicated in Figure 53, a probability distribution for a random variable is a function that maps from the range of the random variable to a real number,  $p \in [0, 1]$ . The value of  $p$  should be interpreted as the probability associated with the event represented by the random variable.

For a discrete random variable,  $X$ , with possible values  $x_1, x_2, \dots, x_n$  ( $n$  may be infinite), the function,  $f(x)$  that assigned probabilities to each possible value of the random



**Figure 53:** robability Distributions Map Random Variables to Probabilities

variable is called the probability mass function (PMF) and is denoted:

$$f(x_i) = P(X = x_i)$$

where  $f(x_i) \geq 0$  for all  $x_i$  and  $\sum_{i=1}^n f(x_i) = 1$ . The probability mass function describes the probability value associated with each discrete value of the random variable.

For a continuous random variable,  $X$ , the mapping from real numbers to probability values is governed by a probability density function,  $f(x)$  and has the properties:

1.  $f(x) \geq 0$
2.  $\int_{-\infty}^{\infty} f(x)dx = 1$  (The area must sum to 1.)
3.  $P(a \leq x \leq b) = \int_a^b f(x)dx$  (The area under  $f(x)$  between a and b.)

The probability density function (PDF) describes the probability associated with a range of possible values for a continuous random variable.

A cumulative distribution function (CDF) for a discrete or continuous random variable can also be defined. For a discrete random variable, the cumulative distribution function is defined as

$$F(x) = P(X \leq x) = \sum_{x_i \leq x} f(x_i)$$

and satisfies,  $0 \leq F(x) \leq 1$ , and for  $x \leq y$  then  $F(x) \leq F(y)$ .

The cumulative distribution function of a continuous random variable is

$$F(x) = P(X \leq x) = \int_{-\infty}^x f(u)du \text{ for } -\infty < x < \infty$$

Thus, when modeling the elements of a simulation model that have randomness, one must determine:

- Whether or not the randomness is discrete or continuous
- The form of the distribution function (i.e. the PMF or PDF)

To develop an understanding of the probability distribution for the random variable, it is useful to characterize various properties of the distribution such as the expected value and variance of the random variable.

The *expected value* of a discrete random variable  $X$ , is denoted by  $E[X]$  and is defined as:

$$E[X] = \sum_x x f(x)$$

where the sum is defined through all possible values of  $x$ . The *variance* of  $X$  is denoted by  $Var[X]$  and is defined as:

$$\begin{aligned} Var[X] &= E[(X - E[X])^2] \\ &= \sum_x (x - E[X])^2 f(x) \\ &= \sum_x x^2 f(x) - (E[X])^2 \\ &= E[X^2] - (E[X])^2 \end{aligned}$$

Suppose  $X$  is a continuous random variable with PDF,  $f(x)$ , then the expected value of  $X$  is

$$E[X] = \int_{-\infty}^{\infty} x f(x) dx$$

and the variance of  $X$  is

$$Var[X] = \int_{-\infty}^{\infty} (x - E[X])^2 f(x) dx = \int_{-\infty}^{\infty} x^2 f(x) dx - (E[X])^2$$

which is equivalent to  $Var[X] = E[X^2] - (E[X])^2$  where

$$E[X^2] = \int_{-\infty}^{\infty} x^2 f(x) dx$$

Another parameter that is often useful is the coefficient of variation. The coefficient of variation is defined as:

$$c_v = \frac{\sqrt{Var[X]}}{E[X]}$$

The coefficient of variation measures the amount of variation relative to the mean value (provided that  $E\{X\} \neq 0$ ).

To estimate  $E\{X\}$ , the sample average,  $\bar{X}(n)$ ,

$$\bar{X}(n) = \frac{1}{n} \sum_{i=1}^n X_i$$

is often used. To estimate  $Var[X]$ , assuming that the data are independent, the sample variance,  $S^2$ ,

$$S^2(n) = \frac{1}{n-1} \sum_{i=1}^n (X_i - \bar{X})^2$$

can be used. Thus, an estimator for the coefficient of variation is:

$$\hat{c}_v = \frac{s}{\bar{x}}$$

A number of other statistical quantities are also useful when trying to characterize the properties of a distribution:

- skewness - Measures the asymmetry of the distribution about its mean.
- kurtosis - Measures the degree of peakedness of the distribution
- order statistics - Used when comparing the sample data to the theoretical distribution via P-P plots or Q-Q plots.
- quantiles (1st quartile, median, 3rd quartile) - Summarizes the distribution of the data.
- minimum, maximum, and range - Indicates the range of possible values

Skewness can be estimated by:

$$\hat{\gamma}_1 = \frac{\frac{1}{n} \sum_{i=1}^n (X_i - \bar{X})^3}{[S^2]^{3/2}}$$

For a unimodal distribution, negative skew indicates that the tail on the left side of the PDF is longer or fatter than the right side. Positive skew indicates that the tail on the right side is longer or fatter than the left side. A value of skewness near zero indicates symmetry.

Kurtosis can be estimated by:

$$\hat{\gamma}_2 = \frac{n-1}{(n-2)(n-3)} ((n+1)g_2 + 6)$$

where,  $g_2$  is:

$$g_2 = \frac{\frac{1}{n} \sum_{i=1}^n (X_i - \bar{X})^4}{\left(\frac{1}{n} \sum_{i=1}^n (X_i - \bar{X})^2\right)^2} - 3$$

Order statistics are just a fancy name for the sorted data. Let  $(x_1, x_2, \dots, x_n)$  represent a sample of data. If the data is sorted from smallest to largest, then the  $i^{th}$  ordered element can be denoted as  $x_{(i)}$ . For example,  $x_{(1)}$  is the smallest element, and  $x_{(n)}$  is the largest, so that  $(x_{(1)}, x_{(2)}, \dots, x_{(n)})$  represents the ordered data and these values are called the order statistics. From the order statistics, a variety of other statistics can be computed:

1. minimum =  $x_{(1)}$
2. maximum =  $x_{(n)}$
3. range =  $x_{(n)} - x_{(1)}$

The median,  $\tilde{x}$ , is a measure of central tendency such that one-half of the data is above it and one-half of the data is below it. The median can be estimated as follows:

$$\tilde{x} = \begin{cases} x_{((n+1)/2)} & n \text{ is odd} \\ \frac{x_{(n/2)} + x_{((n/2)+1)}}{2} & n \text{ is even} \end{cases}$$

For example, consider the following data:

$$x_{(1)} = 3, x_{(2)} = 5, x_{(3)} = 7, x_{(4)} = 7, x_{(5)} = 38$$

Because  $n = 5$ , we have:

$$\frac{n+1}{2} = \frac{5+1}{2} = 3$$

$$\tilde{x} = x_{(3)} = 7$$

Suppose we have the following data:  
 $x_{(1)} = 3, x_{(2)} = 5, x_{(3)} = 7, x_{(4)} = 7$

Because  $n = 4$ , we have:

$$\begin{aligned} x_{(n/2)} &= x_{(2)} \\ x_{((n/2)+1)} &= x_{(3)} \\ \tilde{x} &= \frac{x_{(2)} + x_{(3)}}{2} = \frac{5 + 7}{2} = 6 \end{aligned}$$

The first quartile is the first 25% of the data and can be thought of as the 'median' of the first half of the data. Similarly, the third quartile is the first 75% of the data or the 'median' of the second half of the data. Different methods are used to estimate these quantities in various software packages; however, their interpretation is the same, summarizing the distribution of the data.

As noted in this section, a key decision in distribution modeling is whether the underlying random variable is discrete or continuous. The next section discusses how to model discrete distributions.

## .9 Modeling with Discrete Distributions

There are a wide variety of discrete random variables that often occur in simulation modeling. Appendix .16 summarizes the functions and characteristics some common discrete distributions. Table 44 provides an overview of some modeling situations for common discrete distributions.

**Table 44:** Common Modeling Situations for Discrete Distributions

Distribution	Modeling Situations
Bernoulli( $p$ )	independent trials with success probability $p$
Binomial( $n,p$ )	sum of $n$ Bernoulli trials with success probability $p$
Geometric( $p$ )	number of Bernoulli trials until the first success
Negative Binomial( $r,p$ )	number of Bernoulli trials until the $r^{th}$ success
Discrete Uniform( $a,b$ )	equally likely outcomes over range ( $a, b$ )
Discrete Uniform $v_1, \dots, v_n$	equally likely over values $v_i$
Poisson( $\lambda$ )	counts of occurrences in an interval, area, or volume

By understanding the modeling situations that produce data, you can hypothesize the appropriate distribution for the distribution fitting process.

## .10 Fitting Discrete Distributions

This section illustrates how to model and fit a discrete distribution to data. Although the steps in modeling discrete and continuous distributions are very similar, the processes and tools utilized vary somewhat. The first thing to truly understand is the difference between discrete and continuous random variables. Discrete distributions are used to model discrete random variables. Continuous distributions are used to model continuous random variables. This may seem obvious but it is a key source of confusion for novice modelers.

Discrete random variables take on any of a specified *countable* list of values. Continuous random variables take on any numerical value in an interval or collection of intervals. The source of confusion is when looking at a file of the data, you might not be able to tell the difference. The discrete values may have decimal places and then the modeler thinks that the data is continuous. The modeling starts with what is being collected and how it is being collected (not with looking at a file!).

### .10.1 Fitting a Poisson Distribution

Since the Poisson distribution is very important in simulation modeling, the discrete input modeling process will be illustrated by fitting a Poisson distribution. Example .11 presents data collected from an arrival process. As noted in Table 44, the Poisson distribution is a prime candidate for modeling this type of data.

---

**Example .11** (Fitting a Poisson Distribution). Suppose that we are interested in modeling the demand for a computer laboratory during the morning hours between 9 am to 11 am on normal weekdays. During this time a team of undergraduate students has collected the number of students arriving to the computer lab during 15 minute intervals over a period of 4 weeks. The observations per 15 minute interval are over viewed in Table 45. The full data set is available with the chapter files.

Since there are four 15 minute intervals in each hour for each two hour time period, there are 8 observations per day. Since there are 5 days per week, we have 40 observations per week for a total of  $40 \times 4 = 160$  observations. Check whether a Poisson distribution is an appropriate model for this data.

**Table 45:** Computer Laboratory Arrival Counts by Week, Period, and Day

Week	Period	M	T	W	TH	F
1	9:00-9:15 am	8	5	16	7	7
1	9:15-9:30 am	8	4	9	8	6
1	9:30-9:45 am	9	5	6	6	5
1	9:45-10:00 am	10	11	12	10	12
1	10:00-10:15 am	6	7	14	9	3
1	10:15-10:30 am	11	8	7	7	11
1	10:30-10:45 am	12	7	8	3	6
1	10:45-11:00 am	8	9	9	8	6
2	9:00-9:15 am	10	13	7	7	7
×	×	×	×	×	×	×
3	9:00-9:15 am	5	7	14	8	8
×	×	×	×	×	×	×
4	9:00-9:15 am	7	11	8	5	4
×	×	×	×	×	×	×
4	10:45-11:00 am	8	9	7	9	6

---

The solution to Example .11 involves the following steps:

1. Visualize the data.
2. Check if the week, period, or day of week influence the statistical properties of the count data.
3. Tabulate the frequency of the count data.
4. Estimate the mean rate parameter of the hypothesized Poisson distribution.
5. Perform goodness of fit tests to test the hypothesis that the number of arrivals

per 15 minute interval has a Poisson distribution versus the alternative that it does not have a Poisson distribution.

### .10.2 Visualizing the Data

When analyzing a data set it is best to begin with visualizing the data. We will analyze this data utilizing the R statistical software package. Assuming that the data is in a comma separate value (csv) file called *PoissonCountData.csv* in the R working directory, the following commands will read the data into the R environment, plot a histogram, plot a time series plot, and make an autocorrelation plot of the data.

```
p2 = read.csv("PoissonCountData.csv")
hist(p2$N, main="Computer Lab Arrivals", xlab = "Counts")
plot(p2$N,type="b",main="Computer Lab Arrivals", ylab = "Count", xlab = "Observation#")
acf(p2$N, main = "ACF Plot for Computer Lab Arrivals")
```

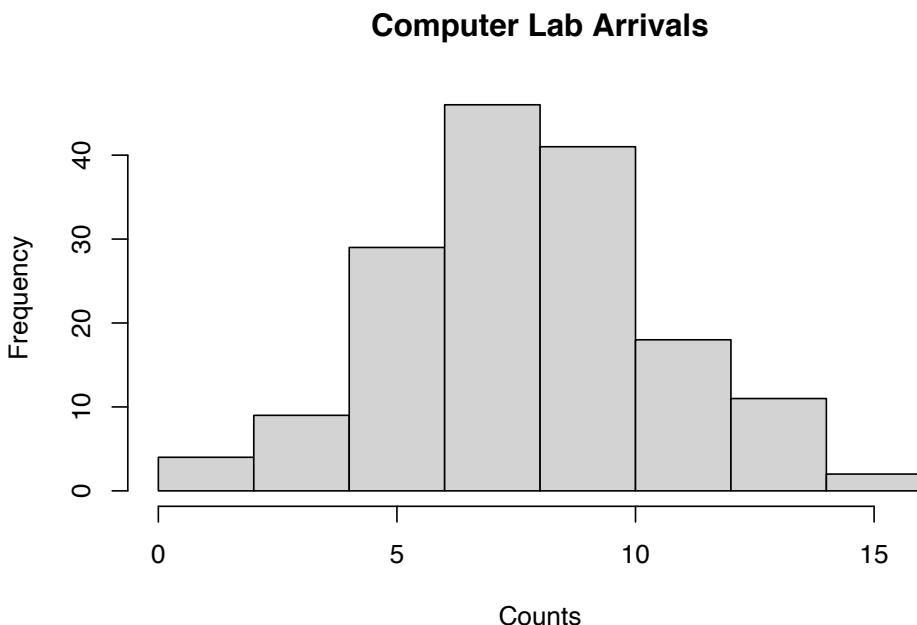
Table 46 illustrates the layout of the data frame in R. The first column indicates the week, the 2nd column represents the period of the day, the 3rd column indicates the day of the week, and the last column indicated with the variable,  $N$ , represents the count for the week, period, day combination. Notice that each period of the day is labeled numerically. To access a particular column within the data frame you use the \$ operator. Thus, the reference,  $p\$N$  accesses all the counts across all of the week, period, day combinations. The variable,  $p\$N$ , is subsequently used in the *hist*, *plot*, and *acf* commands.

As can be seen in Figure 54 the data has a somewhat symmetric shape with nothing unusual appearing in the figure. The shape of the histogram is consistent with possible shapes associated with the Poisson distribution.

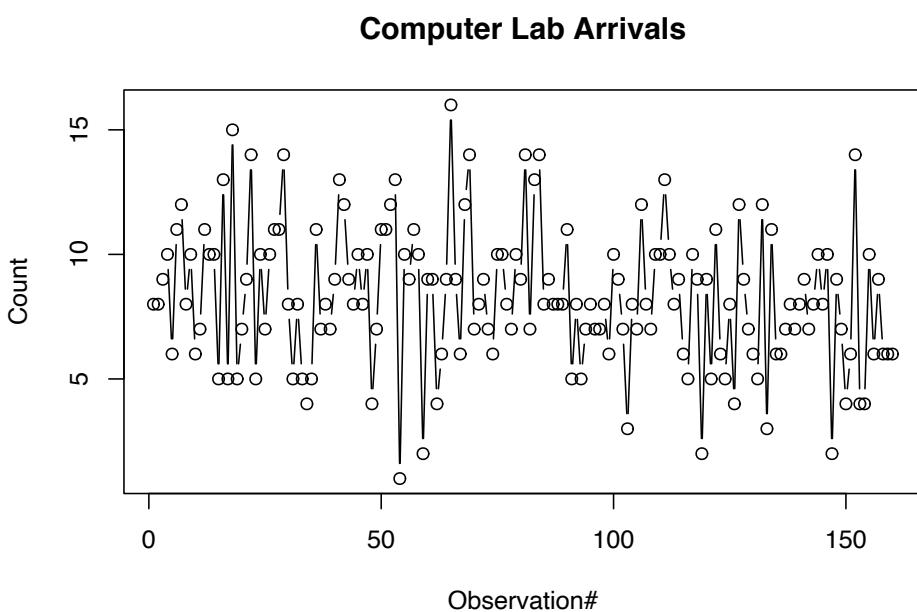
The time series plot, shown in Figure 55, illustrates no significant patterns. We see random looking data centered around a common mean value with no trends of increasing or decreasing data points and no cyclical patterns of up and down behavior.

An autocorrelation plot allows the dependence within the data to be quickly examined. An autocorrelation plot is a time series assessment tool that plots the lag-k correlation versus the lag number. In order to understand these terms, we need to provide some background on how to think about a time series. A time series is a sequence of observations ordered by observation number,  $X_1, X_2, \dots, X_n$ . A time series,  $X_1, X_2, \dots, X_n$ , is said to be *covariance stationary* if:

- The mean of  $X_i$ ,  $E[X_i]$ , exists and is a constant with respect to time. That is,  $\mu = E[X_i]$  for  $i=1, 2, \dots$
- The variance of  $X_i$ ,  $Var[X_i]$  exists and is constant with respect to time. That is,  $\sigma^2 = Var[X_i]$  for  $i=1, 2, \dots$
- The lag-k autocorrelation,  $\rho_k = cor[X_i, X_{i+k}]$ , is not a function of time  $i$  but is a function of the distance  $k$  between points. That is, the correlation between any two points in the series does not depend upon where the points are in the series, it depends only upon the distance between them in the series.



**Figure 54:** Histogram of Computer Lab Arrivals



**Figure 55:** Time Series Plot of Computer Lab Arrivals

**Table 46:** Computer Lab Arrival Data.

Week	Period	Day	N
1	1	M	8
1	2	M	8
1	3	M	9
1	4	M	10
1	5	M	6
1	6	M	11
1	7	M	12
1	8	M	8
2	1	M	10
2	2	M	6
2	3	M	7
2	4	M	11
2	5	M	10
2	6	M	10
2	7	M	5
2	8	M	13
3	1	M	5
3	2	M	15
3	3	M	5
3	4	M	7

Recall that the correlation between two random variables is defined as:

$$\text{cor}[X, Y] = \frac{\text{cov}[X, Y]}{\sqrt{\text{var}[X]\text{Var}[Y]}}$$

where the covariance,  $\text{cov}[X, Y]$  is defined by:

$$\text{cov}[X, Y] = E[(X - E[X])(Y - E[Y])] = E[XY] - E[X]E[Y]$$

The correlation between two random variables  $X$  and  $Y$  measures the strength of linear association. The correlation has no units of measure and has a range:  $[-1, 1]$ . If the correlation between the random variables is less than zero then the random variables are said to be *negatively correlated*. This implies that if  $X$  tends to be high then  $Y$  will tend to be low, or alternatively if  $X$  tends to be low then  $Y$  will tend to be high. If the correlation between the random variables is positive, the random variables are said to be *positively correlated*. This implies that if  $X$  tends to be high then  $Y$  will tend to be high, or alternatively if  $X$  tends to be low then  $Y$  will tend to be low. If the correlation is zero, then the random variables are said to be uncorrelated. If  $X$  and  $Y$  are independent random variables then the correlation between them will be zero. The

converse of this is not necessarily true, but an assessment of the correlation should tell you something about the linear dependence between the random variables.

The autocorrelation between two random variables that are  $k$  time points apart in a covariance stationary time series is given by:

$$\begin{aligned}\rho_k &= \text{cor}[X_i, X_{i+k}] = \frac{\text{cov}[X_i, X_{i+k}]}{\sqrt{\text{Var}[X_i]\text{Var}[X_{i+k}]}} \\ &= \frac{\text{cov}[X_i, X_{i+k}]}{\sigma^2} \text{ for } k = 1, 2, \dots\end{aligned}$$

A plot of  $\rho_k$  for increasing values of  $k$  is called an autocorrelation plot. The autocorrelation function as defined above is the theoretical function. When you have data, you must estimate the values of  $\rho_k$  from the actual times series. This involves forming an estimator for  $\rho_k$ . (Law, 2007) suggests plotting:

$$\hat{\rho}_k = \frac{\hat{C}_k}{S^2(n)}$$

where

$$\hat{C}_k = \frac{1}{n-k} \sum_{i=1}^{n-k} (X_i - \bar{X}(n)) (X_{i+k} - \bar{X}(n)) \quad (10)$$

$$S^2(n) = \frac{1}{n-1} \sum_{i=1}^n (X_i - \bar{X}(n))^2 \quad (11)$$

$$\bar{X}(n) = \frac{1}{n} \sum_i^n X_i \quad (12)$$

are the sample covariance, sample variance, and sample average respectively.

Some time series analysis books, see for examples Box et al. (1994), have a slightly different definition of the sample autocorrelation function:

$$r_k = \frac{c_k}{c_0} = \frac{\sum_{i=1}^{n-k} (X_i - \bar{X}(n)) (X_{i+k} - \bar{X}(n))}{\sum_{i=1}^n (X_i - \bar{X}(n))^2} \quad (13)$$

where

$$c_k = \frac{1}{n} \sum_{i=1}^{n-k} (X_i - \bar{X}(n)) (X_{i+k} - \bar{X}(n)) \quad (14)$$

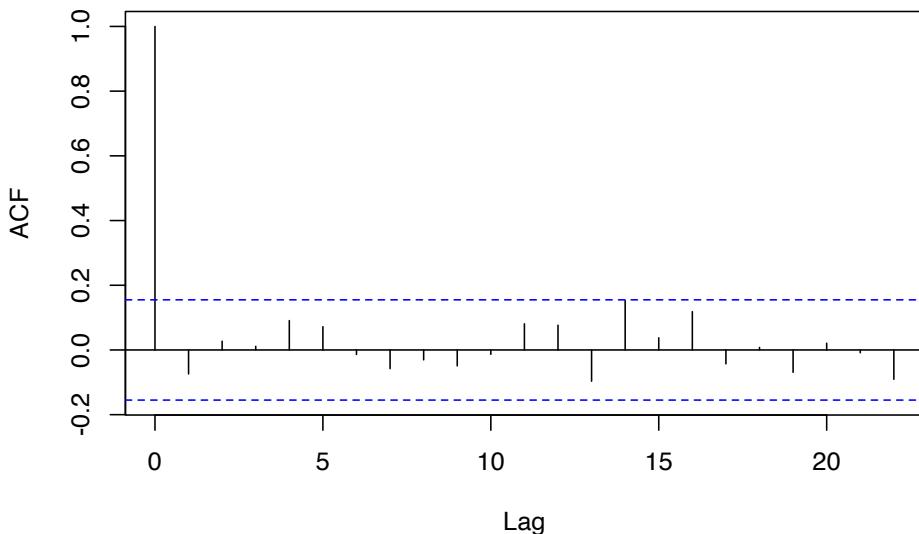
Notice that the numerator in Equation (13) has  $n - k$  terms and the denominator has  $n$  terms. A plot of  $r_k$  versus  $k$  is called a correlogram or sample autocorrelation plot. For the data to be uncorrelated,  $r_k$  should be approximately zero for the values of  $k$ . Unfortunately, estimators of  $r_k$  are often highly variable, especially for large  $k$ ; however, an autocorrelation plot should still provide you with a good idea of independence.

Formal tests can be performed using various time series techniques and their assumptions. A simple test can be performed based on the assumption that the series is white noise,  $N(0, 1)$  with all  $\rho_k = 0$ . Box et al. (1994) indicate that for large  $n$ ,  $\text{Var}(r_k) \approx \frac{1}{n}$ . Thus, a quick test of dependence is to check if sampled correlations fall within a reasonable confidence band around zero. For example, suppose  $n = 100$ , then  $\text{Var}(r_k) \approx \frac{1}{100} = 0.01$ . Then, the standard deviation is  $\sqrt{0.01} = 0.1$ . Assuming an approximate, 95% confidence level, yields a confidence band of  $\pm 1.645 \times 0.1 = \pm 0.1645$  about zero. Therefore, as long as the plotted values for  $r_k$  do not fall outside of this confidence band, it is likely that the data are independent.

A sample autocorrelation plot can be easily developed once the autocorrelations have been computed. Generally, the maximum lag should be set to no larger than one tenth of the size of the data set because the estimation of higher lags is generally unreliable.

As we can see from Figure 56, there does not appear to be any significant correlation with respect to observation number for the computer lab arrival data. The autocorrelation is well-contained within the lag correlation limits denoted with the dashed lines within the plot.

**ACF Plot for Computer Lab Arrivals**



**Figure 56:** Autocorrelation Function Plot for Computer Lab Arrivals

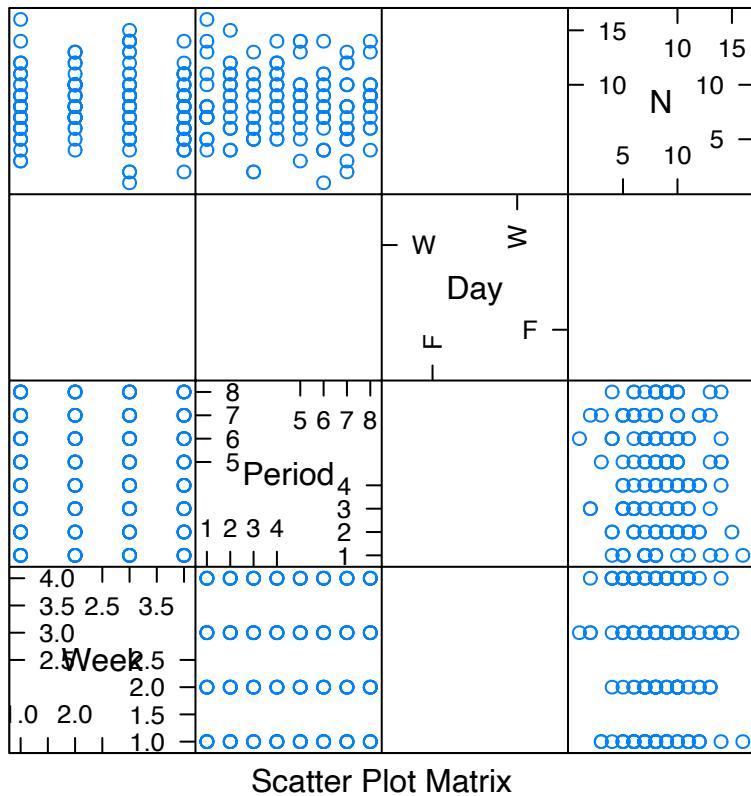
Because arrival data often varies with time, it would be useful to examine whether or not the count data depends in some manner on when it was collected. For example, perhaps the computer lab is less busy on Fridays. In other words, the counts may depend on the day of the week. We can test for dependence on various factors by using a Chi-square based contingency table test. These tests are summarized in introductory statistics books. See (Montgomery and Runger, 2006).

First, we can try to visualize any patterns based on the factors. We can do this easily with a scatter plot matrix within the *lattice* package of R. In addition, we can use the *xtabs* function to tabulate the data by week, period, and day. The *xtabs* function specifies a modeling relationship between the observations and factors. In the R listing,  $N \sim Week + Period + Day$  indicates that we believe that the count column  $N$  in the data, depends on the *Week*, *Period*, and the *Day*. This builds a statistical object that can be summarized using the *summary* command.

```
library(lattice)
splom(p2)
mytable = xtabs(N~Week + Period + Day, data=p2)
summary(mytable)
Call: xtabs(formula = N ~ Week + Period + Day, data = p2)
Number of cases in table: 1324
Number of factors: 3
Test for independence of all factors:
Chisq = 133.76, df = 145, p-value = 0.7384
```

A scatter plot matrix plots the variables in a matrix format that makes it easier to view pairwise relationships within the data. Figure 57 presents the results of the scatter plot. Within the cells, the data looks reasonably ‘uniform’. That is, there are no discernible patterns to be found. This provides evidence that there is not likely to be dependence between these factors. To formally test this hypothesis, we can use the multi-factor contingency table test provided by using the *summary* command on the output object, *myTable* of the *xtabs* command.

The results of using, *summary(mytable)* show that the chi-square test statistic has a very high p-value, 0.7384, when testing if  $N$  depends on *Week*, *Period*, and *Day*. The null hypothesis,  $H_0$ , of a contingency table test of this form states that the counts are independent of the factors versus the alternative,  $H_a$ , that the counts are dependent on the factors. Since the p-value is very high, we should not reject  $H_0$ . What does this all mean? In essence, we can now treat the arrival count observation as 160 independent observations. Thus, we can proceed with formally testing if the counts come from a Poisson distribution without worrying about time or factor dependencies within the data. If the results of the analysis indicated dependence on the factors, then we might need to fit separate distributions based on the dependence. For example, if we concluded that the days of the week were different (but the week and period did not matter), then we could try to fit a separate Poisson distribution for each day of the week. When the mean rate of occurrence depends on time, this situation warrants the investigation of using a non-homogeneous (non-stationary) Poisson process. The estimation of the parameters of a non-stationary Poisson process is beyond the scope of this text.



**Figure 57:** Scatter Plot Matrix from Lattice Package for Computer Lab Arrivals

The interested reader should refer to (Leemis, 1991) and other such references.

### .10.3 Estimating the Rate Parameter for the Poisson Distribution

Testing if the Poisson distribution is a good model for this data can be accomplished using various statistics tests for a Poisson distribution. The basic approach is to compare the hypothesized distribution function in the form of the PDF, PMF, or the CDF to a fit of the data. This implies that you have hypothesized a distribution *and* estimated the parameters of the distribution in order to compare the hypothesized distribution to the data. For example, suppose that you hypothesize that the Poisson distribution will be a good model for the count data. Then, you need to estimate the rate parameter associated with the Poisson distribution.

There are two main methods for estimating the parameters of distribution functions 1) the method of moments and 2) the method of maximum likelihood. The method of moments matches the empirical moments to the theoretical moments of the distribution and attempts to solve the resulting system of equations. The method of maximum likelihood attempts to find the parameter values that maximize the joint probability

distribution function of the sample. Estimation of the parameters from sample data is based on important statistical theory that requires the estimators for the parameters to satisfy statistical properties (e.g. unique, unbiased, invariant, and consistency). It is beyond the scope of this book to cover the properties of these techniques. The interested reader is referred to (Law, 2007) or to (Casella and Berger, 1990) for more details on the theory of these methods.

To make concrete the challenges associated with fitting the parameters of a hypothesized distribution, the maximum likelihood method for fitting the Poisson distribution will be used on the count data. Suppose that you hypothesize that the distribution of the count of the number of arrivals in the  $i^{th}$  15 minute interval can be modeled with a Poisson distribution. Let  $N$  be the number of arrivals in a 15 minute interval. Note that the intervals do not overlap and that we have shown that they can be considered independent of each other. We are hypothesizing that  $N$  has a Poisson distribution with rate parameter  $\lambda$ , where  $\lambda$  represents the expected number of arrivals per 15 minute interval.

$$f(n; \lambda) = P\{N = n\} = \frac{e^{-\lambda} (\lambda)^n}{n!}$$

Let  $N_i$  be the number of arrivals in the  $i^{th}$  interval of the  $k = 160$  intervals. The  $N_1, N_2, \dots, N_k$  form a random sample of size  $k$ . Then, the joint probability probability mass function of the sample is:

$$L(\lambda) = g(n_1, n_2, \dots, n_k; \lambda) = f(n_1; \lambda)f(n_2; \lambda) \dots f(n_k; \lambda) = \prod_{i=1}^k f(n_i; \lambda)$$

The  $(n_1, n_2, \dots, n_k)$  are observed (known values) and  $\lambda$  is unknown. The function  $L(\lambda)$  is called the likelihood function. To estimate the value of  $\lambda$  by the method of maximum likelihood, we must find the value of  $\lambda$  that maximizes the function  $L(\lambda)$ . The interpretation is that we are finding the value of the parameter that is maximizing the likelihood that it came from this sample.

Substituting the definition of the Poisson distribution into the likelihood function yields:

$$\begin{aligned} L(\lambda) &= \prod_{i=1}^k \frac{e^{-\lambda} (\lambda)^{n_i}}{n_i!} \\ &= \frac{e^{-k\lambda} \lambda^{\sum_{i=1}^k n_i}}{\prod_{i=1}^k n_i!} \end{aligned}$$

It can be shown that maximizing  $L(\lambda)$  is the same as maximizing  $\ln(L(\lambda))$ . This is called the log-likelihood function. Thus,

$$\ln(L(\lambda)) = -k\lambda + \ln(\lambda) \sum_{i=1}^k n_i - \sum_{i=1}^k \ln(n_i!)$$

Differentiating with respect to  $\lambda$ , yields,

$$\frac{d\ln(L(\lambda))}{d\lambda} = -k + \frac{\sum_{i=1}^k n_i}{\lambda}$$

When we set this equal to zero and solve for  $\lambda$ , we get

$$0 = -k + \frac{\sum_{i=1}^k n_i}{\lambda}$$

$$\hat{\lambda} = \frac{\sum_{i=1}^k n_i}{k} \quad (15)$$

If the second derivative,  $\frac{d^2\ln L(\lambda)}{d\lambda^2} < 0$  then a maximum is obtained.

$$\frac{d^2\ln L(\lambda)}{d\lambda^2} = \frac{-\sum_{i=1}^k n_i}{\lambda^2}$$

because the  $n_i$  are positive and  $\lambda$  is positive the second derivative must be negative; therefore the maximum likelihood estimator for the parameter of the Poisson distribution is given by Equation (15). Notice that this is the sample average of the interval counts, which can be easily computed using the *mean()* function within R.

While the Poisson distribution has an analytical form for the maximum likelihood estimator, not all distributions will have this property. Estimating the parameters of distributions will, in general, involve non-linear optimization. This motivates the use of software tools such as R when performing the analysis. Software tools will perform this estimation process with little difficulty. Let's complete this example using R to fit and test whether or not the Poisson distribution is a good model for this data.

#### **.10.4 Chi-Squared Goodness of Fit Test for Poisson Distribution**

In essence, a distributional test examines the hypothesis  $H_0 : X_i \sim F_0$  versus the alternate hypothesis of  $H_a : X_i \not\sim F_0$ . That is, the null hypothesis is that data come from distribution function,  $F_0$  and the alternative hypothesis that the data are not distributed according to  $F_0$ .

As a reminder, when using a hypothesis testing framework, we can perform the test in two ways: 1) pre-specifying the Type I error and comparing to a critical value or 2) pre-specifying the Type I error and comparing to a p-value. For example, in the first case, suppose we let our Type I error  $\alpha = 0.05$ , then we look up a critical value appropriate for the test, compute the test statistic, and reject the null hypothesis  $H_0$  if test statistic is too extreme (large or small depending on the test). In the second case, we compute the p-value for the test based on the computed test statistic, and then reject  $H_0$  if the p-value is less than or equal to the specified Type I error value.

This text emphasizes the p-value approach to hypothesis testing because computing the p-value provides additional information about the significance of the result. The

p-value for a statistical test is the smallest  $\alpha$  level at which the observed test statistic is significant. This smallest  $\alpha$  level is also called the observed level of significance for the test. The smaller the p-value, the more the result can be considered statistically significant. Thus, the p-value can be compared to the desired significance level,  $\alpha$ . Thus, when using the p-value approach to hypothesis testing the testing criterion is:

- If the p-value  $> \alpha$ , then do not reject  $H_0$
- If the p-value  $\leq \alpha$ , then reject  $H_0$

An alternate interpretation for the p-value is that it is the probability assuming  $H_0$  is true of obtaining a test statistic at least as *extreme* as the observed value. *Assuming* that  $H_0$  is true, then the test statistic will follow a known distributional model. The p-value can be interpreted as the chance assuming that  $H_0$  is true of obtaining a more extreme value of the test statistic than the value actually obtained. Computing a p-value for a hypothesis test allows for a different mechanism for accepting or rejecting the null hypothesis. Remember that a Type I error is

$$\alpha = P(\text{Type I error}) = P(\text{rejecting the null when it is in fact true})$$

This represents the chance you are willing to take to make a mistake in your conclusion. The p-value represents the observed chance associated with the test statistic being more extreme under the assumption that the null is true. A small p-value indicates that the observed result is rare under the assumption of  $H_0$ . If the p-value is small, it indicates that an outcome as extreme as observed is possible, but not probable under  $H_0$ . In other words, chance by itself may not adequately explain the result. So for a small p-value, you can reject  $H_0$  as a plausible explanation of the observed outcome. If the p-value is large, this indicates that an outcome as extreme as that observed can occur with high probability. For example, suppose you observed a p-value of 0.001. This means that assuming  $H_0$  is true, there was a 1 in 1000 chance that you would observe a test statistic value at least as extreme as what you observed. It comes down to whether or not you consider a 1 in 1000 occurrence a rare or non-rare event. In other words, do you consider yourself that lucky to have observed such a rare event. If you do not think of this as lucky but you still actually observed it, then you might be suspicious that something is not “right with the world”. In other words, your assumption that  $H_0$  was true is likely wrong. Therefore, you reject the null hypothesis,  $H_0$ , in favor of the alternative hypothesis,  $H_a$ . The risk of you making an incorrect decision here is what you consider a rare event, i.e. your  $\alpha$  level.

For a discrete distribution, the most common distributional test is the Chi-Squared goodness of fit test, which is the subject of the next section.

### .10.5 Chi-Squared Goodness of Fit Test

The Chi-Square Test divides the range of the data into,  $k$ , intervals (or classes) and tests if the number of observations that fall in each interval (or class) is close the expected number that should fall in the interval (or class) given the hypothesized distribution is the correct model. In the case of discrete data, the intervals are called classes and they

are mapped to groupings along the domain of the random variable. For example, in the case of a Poisson distribution, a possible set of  $k = 7$  classes could be  $\{0\}$ ,  $\{1\}$ ,  $\{2\}$ ,  $\{3\}$ ,  $\{4\}$ ,  $\{5\}$ ,  $\{6 \text{ or more}\}$ . As a general rule of thumb, the classes should be chosen so that the expected number of observations in each class is at least 5.

Let  $c_j$  be the observed count of the observations contained in the  $j^{\text{th}}$  class. The Chi-Squared test statistic has the form:

$$\chi_0^2 = \sum_{j=1}^k \frac{(c_j - np_j)^2}{np_j} \quad (16)$$

The quantity  $np_j$  is the expected number of observations that should fall in the  $j^{\text{th}}$  class when there are  $n$  observations.

For large  $n$ , an approximate  $1 - \alpha$  level hypothesis test can be performed based on a Chi-Squared test statistic that rejects the null hypothesis if the computed  $\chi_0^2$  is greater than  $\chi_{\alpha, k-s-1}^2$ , where  $s$  is the *number of estimated parameters* for the hypothesized distribution.  $\chi_{\alpha, k-s-1}^2$  is the upper critical value for a Chi-squared random variable  $\chi^2$  such that  $P\{\chi^2 > \chi_{\alpha, k-s-1}^2\} = \alpha$ . The p-value,  $P\{\chi_{k-s-1}^2 > \chi_0^2\}$ , for this test can be computed in using the following formula:

$$\text{CHISQ.DIST.RT}(\chi_0^2, k - s - 1)$$

In the statistical package *R*, the formula is:

$$\text{pchisq}(\chi_0^2, k - s - 1, \text{lower.tail} = \text{FALSE})$$

The null hypothesis is that the data come from the hypothesized distribution versus the alternative hypothesis that the data do not come from the hypothesized distribution.

Let's perform a Chi-squared goodness of fit test of computer lab data for the Poisson distribution. A good first step in analyzing discrete data is to summarize the observations in a table. We can do that easily with the R *table()* function.

```
# tabulate the counts
tCnts = table(p2$N)
tCnts

##
##  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16
##  1  3  2  7 13 16 21 25 20 21 11  7  5  6  1  1
```

From this analysis, we see that there were no week-period combinations that had 0 observations, 1 that had 1 count, 3 that had 2 counts, and so forth. Now, we will estimate the rate parameter of the hypothesized Poisson distribution using Equation (15) and tabulate the expected number of observations for a proposed set of classes.

```

# get number of observations
n = length(p2$N)
n

## [1] 160

# estimate the rate for Poisson from the data
lambda = mean(p2$N)
lambda

## [1] 8.275

# setup vector of x's across domain of Poisson
x = 0:15
x

## [1] 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

# compute the probability for Poisson
prob = dpois(x, lambda)
prob

## [1] 0.0002548081 0.0021085367 0.0087240706 0.0240638947 0.0497821822
## [6] 0.0823895116 0.1136288681 0.1343255548 0.1389429957 0.1277503655
## [11] 0.1057134275 0.0795253284 0.0548393410 0.0349073498 0.0206327371
## [16] 0.0113823933

# view the expected counts for these probabilities
prob*n

## [1] 0.04076929 0.33736587 1.39585130 3.85022316 7.96514916 13.18232186
## [7] 18.18061890 21.49208877 22.23087932 20.44005848 16.91414839 12.72405254
## [13] 8.77429457 5.58517596 3.30123794 1.82118293

```

We computed the probability associated with the Poisson distribution with  $\hat{\lambda} = 8.275$ . Recall that the Possion distribution has the following form:

$$P\{X = x\} = \frac{e^{-\lambda} \lambda^x}{x!} \quad \lambda > 0, \quad x = 0, 1, \dots$$

Then, by multiplying each probability,  $P\{X = i\} = p_i$  by the number of observations  $n = 160$ , we can get the expected number,  $n \times p_i$ , that we should observed under the hypothesized distribution. Since the expected values for  $x = 0, 1, 2, 3, 4$  are so small, we consolidate them in to one class to have the expected number to be at least 5. We

will also consolidate the range  $x \geq 13$  into a single class and the values of 11 and 12 into a class.

```
# compute the probability for the classes
# the vector is indexed starting at 1, prob[1] = P(X=0)
cProb = c(sum(prob[1:5]), prob[6:11], sum(prob[12:13]), ppois(12, lambda, lower.tail = FALSE))
cProb

## [1] 0.08493349 0.08238951 0.11362887 0.13432555 0.13894300 0.12775037 0.10571343
## [8] 0.13436467 0.07795112

# compute the expected counts for each of the classes
expected = cProb*n
expected

## [1] 13.58936 13.18232 18.18062 21.49209 22.23088 20.44006 16.91415 21.49835
## [9] 12.47218
```

Thus, we will use the following  $k = 9$  classes: {0,1,2,3,4}, {5}, ..., {10}, {11,12}, and {13 or more}. Now, we need to summarize the observed frequencies for the proposed classes.

```
# transform tabulated counts to data frame
dfCnts = as.data.frame(tCnts)
# extract only frequencies
cnts = dfCnts$Freq
cnts

## [1] 1 3 2 7 13 16 21 25 20 21 11 7 5 6 1 1

# consolidate classes for observed
observed = c(sum(cnts[1:4]), cnts[5:10], sum(cnts[11:12]), sum(cnts[13:16]))
observed

## [1] 13 13 16 21 25 20 21 18 13
```

Now, we have both the observed and expected tabulated and can proceed with the chi-squared goodness of fit test. We will compute the result directly using Equation (16).

```
# compute the observed minus expected components
chisq = ((observed - expected)^2)/expected
# compute the chi-squared test statistic
sumchisq = sum(chisq)
# chi-squared test statistic
print(sumchisq)
```

```
## [1] 2.233903
```

```
# set the degrees of freedom, with 1 estimated parameter s = 1
df = length(expected) - 1 - 1
# compute the p-value
pvalue = 1 - pchisq(sumchisq, df)
# p-value
print(pvalue)
```

```
## [1] 0.9457686
```

Based on such a high p-value, we would not reject  $H_0$ . Thus, we can conclude that there is not enough evidence to suggest that the lab arrival count data is some other distribution than the Poisson distribution. In this section, we did the Chi-squared test *by-hand*. R has a package that facilitates distribution fitting called, *fitdistrplus*. We will use this package to analyze the computer lab arrival data again in the next section.

## .10.6 Using the *fitdistrplus* R Package on Discrete Data

Fortunately, R has a very useful package for fitting a wide variety of discrete and continuous distributions call the *fitdistrplus* package. To install and load the package do the following:

```
install.packages('fitdistrplus')
library(fitdistrplus)
plotdist(p2$N, discrete = TRUE)
```

The *plotdist* command will plot the empirical probability mass function and the cumulative distribution function as illustrated in Figure 58.

To perform the fitting and analysis, you use the *fitdist* and *gofstat* commands. In addition, plotting the output from the *fitdist* function will provide a comparison of the empirical distribution to the theoretical distribution.

```
fp = fitdist(p2$N, "pois")
```

```
summary(fp)
```

```
plot(fp)
```

```
gofstat(fp)
```

```
fp = fitdist(p2$N, "pois")
summary(fp)
```

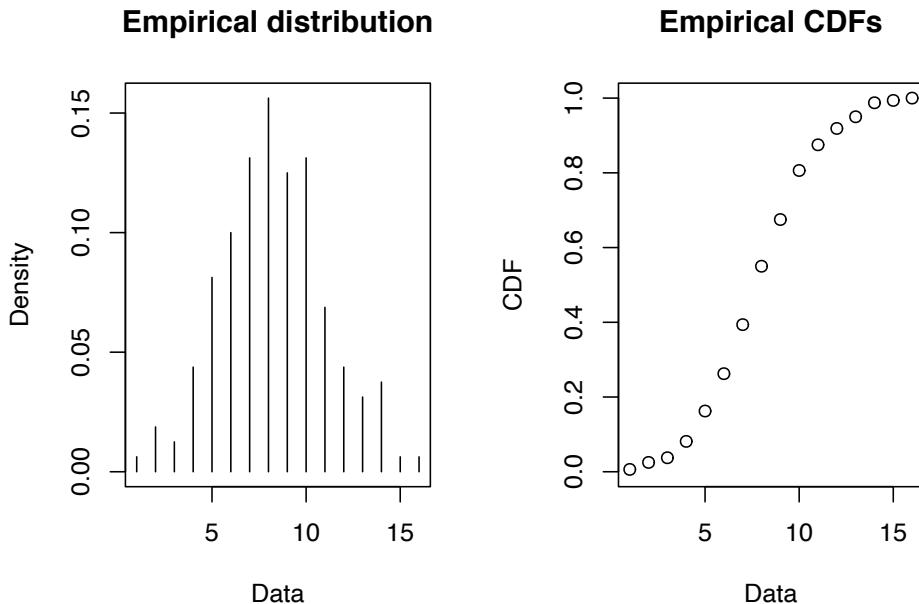
```
## Fitting of the distribution ' pois ' by maximum likelihood
```

```
## Parameters :
```

```
##           estimate Std. Error
```

```
## lambda    8.275  0.2274176
```

```
## Loglikelihood: -393.7743   AIC: 789.5485   BIC: 792.6237
```



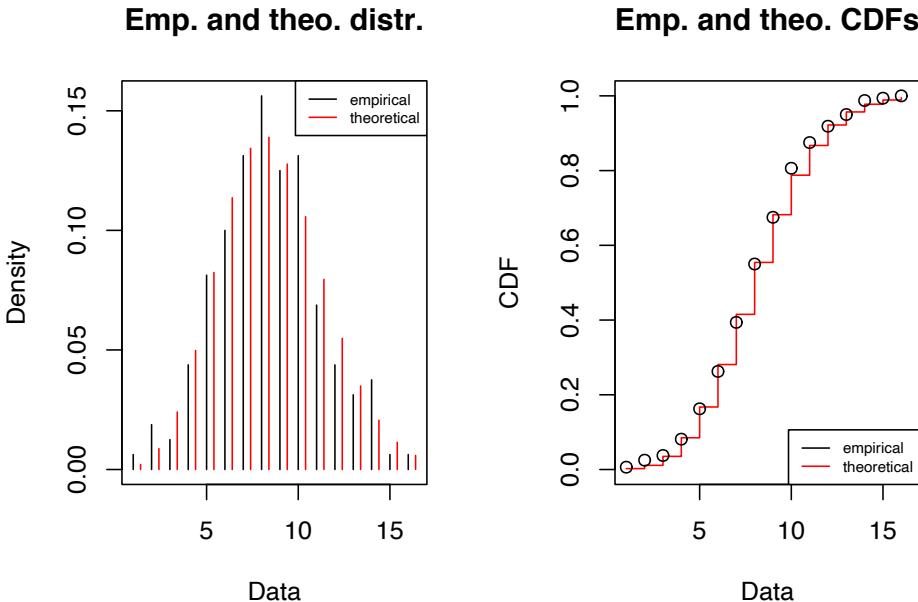
**Figure 58:** Plot of the Empirical PMF and CDF of the Computer Lab Arrivals

```
gofstat(fp)
```

```
## Chi-squared statistic:  2.233903
## Degree of freedom of the Chi-squared distribution:  7
## Chi-squared p-value:  0.9457686
## Chi-squared table:
##      obscounts theocounts
## <= 4    13.00000 13.58936
## <= 5    13.00000 13.18232
## <= 6    16.00000 18.18062
## <= 7    21.00000 21.49209
## <= 8    25.00000 22.23088
## <= 9    20.00000 20.44006
## <= 10   21.00000 16.91415
## <= 12   18.00000 21.49835
## > 12   13.00000 12.47218
##
## Goodness-of-fit criteria
##                               1-mle-pois
## Akaike's Information Criterion  789.5485
## Bayesian Information Criterion 792.6237
```

The output of the *fitdist* and the *summary* commands provides the estimate of  $\lambda =$

8.275. The result object,  $fp$ , returned by the `fitdist` can then subsequently be used to plot the fit, `plot` and perform a goodness of fit test, `gofstat`.



**Figure 59:** Plot of the Empirical and Theoretical CDF of the Computer Lab Arrivals

The `gofstat` command performs a chi-squared goodness of fit test and computes the chi-squared statistic value (here 2.233903) and the p-value (0.9457686). These are exactly the same results that we computed in the previous section. Clearly, the chi-squared test statistic p-value is quite high. Again the null hypothesis is that the observations come from a Poisson distribution with the alternative that they do not. The high p-value suggests that we should not reject the null hypothesis and conclude that a Poisson distribution is a very reasonable model for the computer lab arrival counts.

### .10.7 Fitting a Discrete Empirical Distribution

We are interested in modeling the number of packages delivered on a small parcel truck to a hospital loading dock. A distribution for this random variable will be used in a loading dock simulation to understand the ability of the workers to unload the truck in a timely manner. Unfortunately, a limited sample of observations is available from 40 different truck shipments as shown in Table 47

**Table 47:** Loading Dock Data

130	130	110	130	130	110	110	130	120	130
140	140	130	110	110	140	130	140	130	120
140	150	120	150	120	130	120	100	110	150

**Table 47:** Loading Dock Data

130	120	120	130	120	120	130	130	130	100
-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

Developing a probability model will be a challenge for this data set because of the limited amount of data. Using the following *R* commands, we can get a good understanding of the data. Assuming that the data has been read into the variable, *packageCnt*, the *hist()*, *stripchart()*, and *table()* commands provide an initial analysis.

```
packageCnt = scan("data/AppDistFitting/TruckLoadingData.txt")
hist(packageCnt, main="Packages per Shipment", xlab="#packages")
stripchart(packageCnt, method="stack", pch="o")
table(packageCnt)
```

As we can see from the *R* output, the range of the data varies between 100 and 150. The histogram, shown in Figure 60, illustrates the shape of the data. It appears to be slightly positively skewed. Figure 61 presents a dot plot of the observed counts. From this figure, we can clearly see that the only values obtained in the sample are 100, 110, 120, 130, 140, and 150. It looks as though the ordering comes in tens, starting at 100 units.

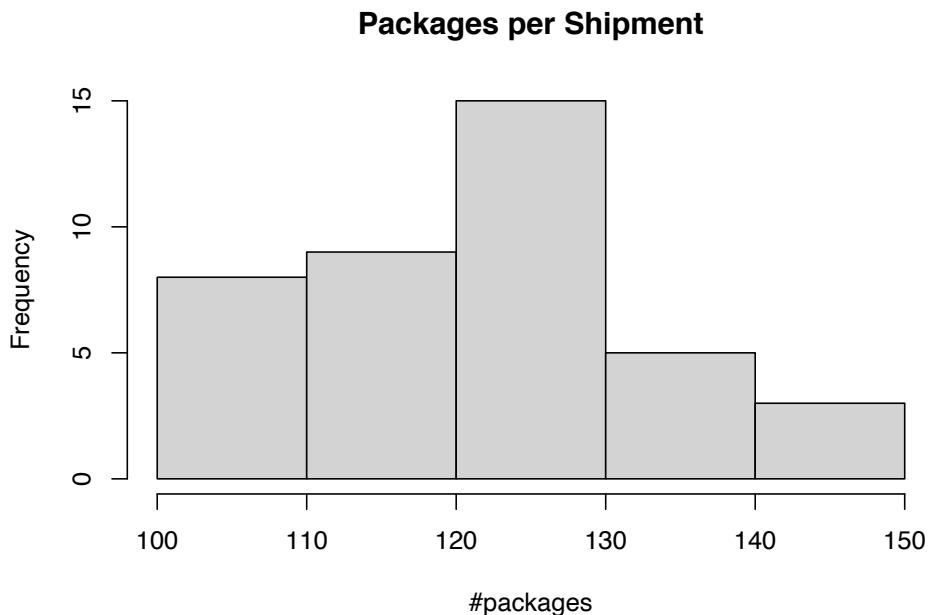
Because of the limited size of the sample and limited variety of data points, fitting a distribution will be problematic. However, we can fit a discrete empirical distribution to the proportions observed in the sample. Using the *table()* command, we can summarize the counts. Then, by dividing by 40 (the total number of observations), we can get the proportions as show in the *R* listing.

```
tp = table(packageCnt)
tp/length(packageCnt)
```

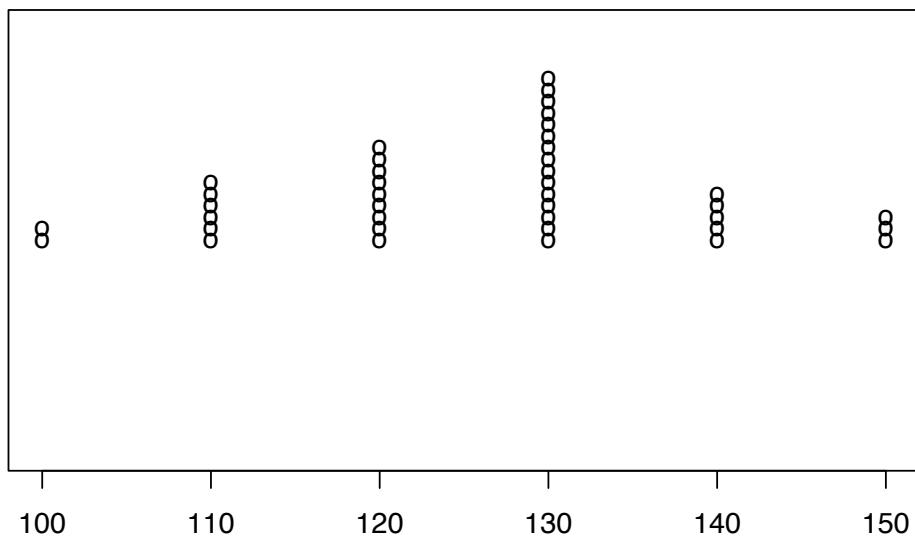
```
## packageCnt
##   100   110   120   130   140   150
## 0.050 0.150 0.225 0.375 0.125 0.075
```

Thus, we can represent this situation with the following probability mass and cumulative distribution functions.

$$P\{X = x\} = \begin{cases} 0.05 & x = 100 \\ 0.15 & x = 110 \\ 0.225 & x = 120 \\ 0.375 & x = 130 \\ 0.125 & x = 140 \\ 0.075 & x = 150 \end{cases}$$



**Figure 60:** Histogram of Package Count per Shipment



**Figure 61:** Dot Plot of Package Counter Shipment

$$F(x) = \begin{cases} 0.0 & \text{if } x < 100 \\ 0.05 & \text{if } 100 \leq x < 110 \\ 0.20 & \text{if } 110 \leq x < 120 \\ 0.425 & \text{if } 120 \leq x < 130 \\ 0.80 & \text{if } 130 \leq x < 140 \\ 0.925 & \text{if } 140 \leq x < 150 \\ 1.0 & \text{if } x \geq 150 \end{cases}$$

The previous two examples illustrated the process for fitting a discrete distribution to data for use in a simulation model. Because discrete event dynamic simulation involves modeling the behavior of a system over time, the modeling of distributions that represent the time to perform a task is important. Since the domain of time is on the set of positive real numbers, it is a continuous variable. We will explore the modeling of continuous distributions in the next section.

## .11 Modeling with Continuous Distributions

Continuous distributions can be used to model situations where the set of possible values occurs in an interval or set of intervals. Within discrete event simulation, the most common use of continuous distributions is for the modeling of the time to perform a task. Appendix .16 summarizes the properties of common continuous distributions.

The continuous uniform distribution can be used to model situations in which you have a lack of data and it is reasonable to assume that everything is equally likely within an interval. Alternatively, if you have no a priori knowledge that some events are more likely than others, then a uniform distribution seems like a reasonable starting point. The uniform distribution is also commonly used to model machine processing times that have very precise time intervals for completion. The expected value and variance of a random variable with a continuous uniform distribution over the interval (a, b) is:

$$E[X] = \frac{a + b}{2}$$

$$Var[X] = \frac{(b - a)^2}{12}$$

Often the continuous uniform distribution is specified by indicating the  $\pm$  around the expected value. For example, we can say that a continuous uniform over the range (5, 10) is the same as a uniform with  $7.5 \pm 2.5$ . The uniform distribution is symmetric over its defined interval.

The triangular distribution is also useful in situations with a lack of data if you can characterize a most likely value for the random variable in addition to its range (minimum and maximum). This makes the triangular distribution very useful when the only data that you might have on task times comes from interviewing people. It is relatively easy for someone to specify the most likely task time, a minimum task time,

and a maximum task time. You can create a survey instrument that asks multiple people familiar with the task to provide these three estimates. From, the survey you can average the responses to develop an approximate model. This is only one possibility for how to combine the survey values.

If the most likely value is equal to one-half the range, then the triangular distribution is symmetric. In other words, fifty percent of the data is above and below the most likely value. If the most likely value is closer to the minimum value then the triangular distribution is right-skewed (more area to the right). If the most likely value is closer to the maximum value then the triangular distribution is left-skewed. The ability to control the skewness of the distribution in this manner also makes this distribution attractive.

The beta distribution can also be used to model situations where there is a lack data. It is a bounded continuous distribution over the range from  $(0, 1)$  but can take on a wide variety of shapes and skewness characteristics. The beta distribution has been used to model the task times on activity networks and for modeling uncertainty concerning the probability parameter of a discrete distribution, such as the binomial. The beta distribution is commonly shifted to be over a range of values  $(a, b)$ .

The exponential distribution is commonly used to model the time between events. Often, when only a mean value is available (from the data or from a guess), the exponential distribution can be used. A random variable,  $X$ , with an exponential distribution rate parameter  $\lambda$  has:

$$E[X] = \frac{1}{\lambda}$$

$$Var[X] = \frac{1}{\lambda^2}$$

Notice that the variance is the square of the expected value. This is considered to be highly variable. The coefficient of variation for the exponential distribution is  $c_v = 1$ . Thus, if the coefficient of variation estimated from the data has a value near 1.0, then an exponential distribution may be possible choice for modeling the situation.

An important property of the exponential distribution is the lack of memory property. The lack of memory property of the exponential distribution states that given  $\Delta t$  is the time period that elapsed since the occurrence of the last event, the time  $t$  remaining until the occurrence of the next event is independent of  $\Delta t$ . This implies that,  $P\{X > \Delta t + t | X > t\} = P\{X > t\}$ . This property indicates that the probability of the occurrence of the next event is dependent upon the length of the interval since the last event, but not the absolute time of the last occurrence. It is the interval of elapsed time that matters. In a sense the process's clock resets at each event time and the past does not matter when predicting the future. Thus, it "forgets" the past. This property has some very important implications, especially when modeling the time to failure. In most situations, the history of the process does matter (such as wear and tear on the machine). In which case, the exponential distribution may not be appropriate. Other distributions of the exponential family may be more useful in these situations such as

the gamma and Weibull distributions. Why is the exponential distribution often used? Two reasons: 1) it often is a good model for many situations found in nature and 2) it has very convenient mathematical properties.

While the normal distribution is a mainstay of probability and statistics, you need to be careful when using it as a distribution for input models because it is defined over the entire range of real numbers. For example, within simulation the time to perform a task is often required; however, time must be a positive real number. Clearly, since a normal distribution can have negative values, using a normal distribution to model task times can be problematic. If you attempt to delay for negative time you will receive an error. Instead of using a normal distribution, you might use a truncated normal, see Section 5.4. Alternatively, you can choose from any of the distributions that are defined on the range of positive real numbers, such as the lognormal, gamma, Weibull, and exponential distributions. The lognormal distribution is a convenient choice because it is also specified by two parameters: the mean and variance.

Table 48 lists common modeling situations for various continuous distributions.

**Table 48:** Common Modeling Situations for Continuous Distributions

Distribution	Modeling Situations
Uniform	when you have no data, everything is equally likely to occur within an interval, machine task times
Normal	modeling errors, modeling measurements, length, etc., modeling the sum of a large number of observations
Exponential	time to perform a task, time between failures, distance between defects
Erlang	service times, multiple phases of service with each phase exponential
Weibull	time to failure, time to complete a task
Gamma	repair times, time to complete a task, replenishment lead time
Lognormal	time to perform a task, quantities that are the product of a large number of other quantities
Triangular	rough model in the absence of data assume a minimum, a maximum, and a most likely value
Beta	useful for modeling task times on bounded range with little data, modeling probability as a random variable

Once we have a good idea about the type of random variable (discrete or continuous) and some ideas about the distribution of the random variable, the next step is to fit a distributional model to the data. In the following sections, we will illustrate how to fit continuous distributions to data.

## .12 Fitting Continuous Distributions

Previously, we examined the modeling of discrete distributions. In this section, we will look at modeling a continuous distribution using the functionality available in R. This example starts with step 3 of the input modeling process. That is, the data has already been collected. Additional discussion of this topic can be found in Chapter 6 of (Law, 2007).

**Example .12** (Fitting a Gamma Distribution). Suppose that we are interested in modeling the time that it takes to perform a computer component repair task. The 100 observations are provide below in minutes. Fit an appropriate distribution to this data.

	1	2	3	4	5	6	7	8	9	10
1	15.3	10.0	12.6	19.7	9.4	11.7	22.6	13.8	15.8	17.2
2	12.4	3.0	6.3	7.8	1.3	8.9	10.2	5.4	5.7	28.9
3	16.5	15.6	13.4	12.0	8.2	12.4	6.6	19.7	13.7	17.2
4	3.8	9.1	27.0	9.7	2.3	9.6	8.3	8.6	14.8	11.1
5	19.5	5.3	25.1	13.5	24.7	9.7	21.0	3.9	6.2	10.9
6	7.0	10.5	16.1	5.2	23.0	16.0	11.3	7.2	8.9	7.8
7	20.1	17.8	14.4	8.4	12.1	3.6	10.9	19.6	14.1	16.1
8	11.8	9.2	31.4	16.4	5.1	20.7	14.7	22.5	22.1	22.7
9	22.8	17.7	25.6	10.1	8.2	24.4	30.8	8.9	8.1	12.9
10	9.8	5.5	7.4	31.5	29.1	8.9	10.3	8.0	10.9	6.2

---

### .12.1 Visualizing the Data

The first steps are to visualize the data and check for independence. This can be readily accomplished using the *hist*, *plot*, and *acf* functions in R. Assume that the data is in a file called, *taskTimes.txt* within the R working directory.

```
y = scan(file="data/AppDistFitting/taskTimes.txt")
hist(y, main="Task Times", xlab = "minutes")
plot(y,type="b",main="Task Times", ylab = "minutes", xlab = "Observation#")
acf(y, main = "ACF Plot for Task Times")
```

As can be seen in Figure 62, the histogram is slightly right skewed.

The time series plot, Figure 63, illustrates no significant pattern (e.g. trends, etc.).

Finally, the autocorrelation plot, Figure 64, shows no significant correlation (the early lags are well within the confidence band) with respect to observation number.

Based on the visual analysis, we can conclude that the task times are likely to be independent and identically distributed.

### .12.2 Statistically Summarize the Data

An analysis of the statistical properties of the task times can be easily accomplished in R using the *summary*, *mean*, *var*, *sd*, and *t.test* functions. The *summary* command sum-

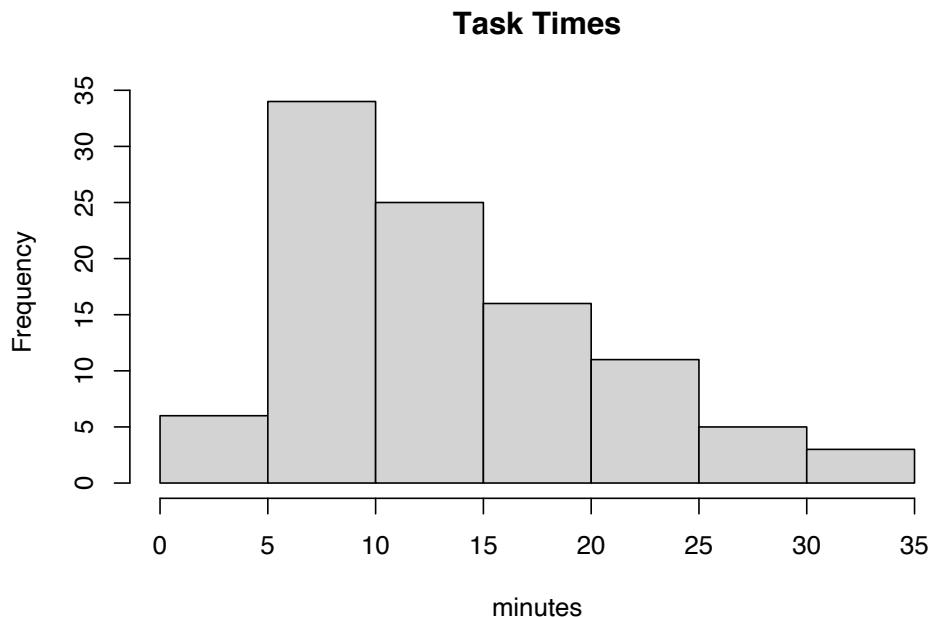


Figure 62: Histogram of Computer Repair Task Times

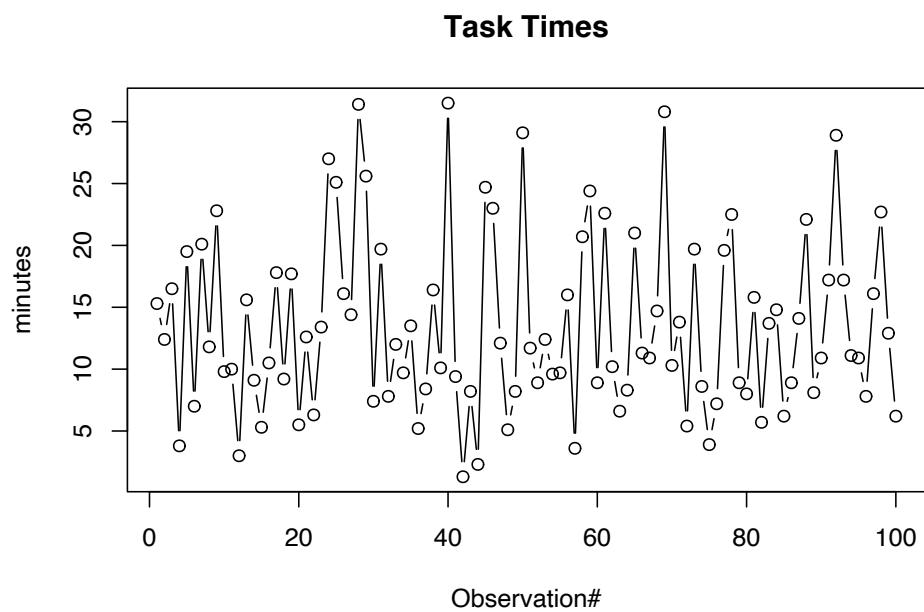
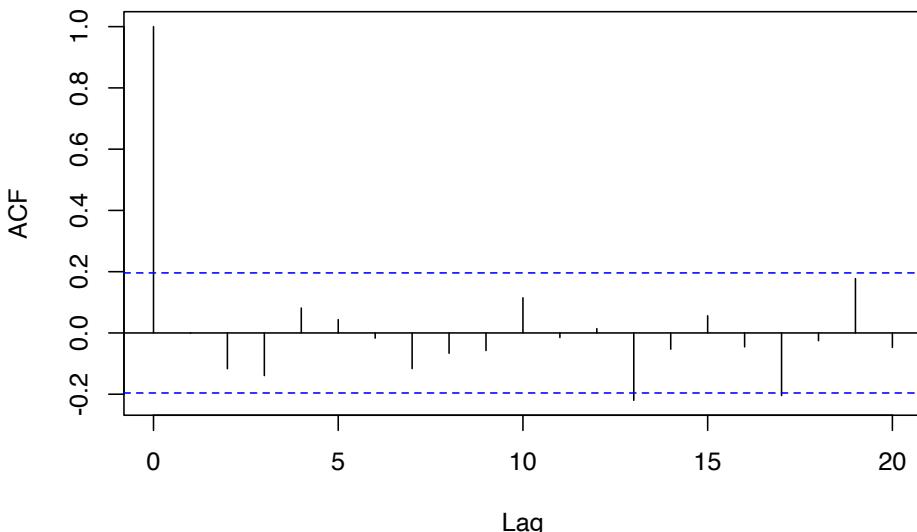


Figure 63: Time Series Plot of Computer Repair Task Times

### ACF Plot for Task Times



**Figure 64:** ACF Plot of Computer Repair Task Times

marizes the distributional properties in terms of the minimum, maximum, median, and 1st and 3rd quartiles of the data.

```
summary(y)

##      Min. 1st Qu. Median     Mean 3rd Qu.    Max.
##  1.300   8.275  11.750  13.412  17.325  31.500
```

The *mean*, *var*, *sd* commands compute the sample average, sample variance, and sample standard deviation of the data.

```
mean(y)

## [1] 13.412

var(y)

## [1] 50.44895

sd(y)

## [1] 7.102742
```

Finally, the *t.test* command can be used to form a 95% confidence interval on the mean and test if the true mean is significantly different from zero.

```
t.test(y)

##
##  One Sample t-test
##
## data: y
## t = 18.883, df = 99, p-value < 2.2e-16
## alternative hypothesis: true mean is not equal to 0
## 95 percent confidence interval:
## 12.00266 14.82134
## sample estimates:
## mean of x
## 13.412
```

The *descdist* command of the *fitdistrplus* package will also provide a description of the distribution's properties.

```
descdist(y, graph=FALSE)

##
## summary statistics
## -----
## min: 1.3 max: 31.5
## median: 11.75
## mean: 13.412
## estimated sd: 7.102742
## estimated skewness: 0.7433715
## estimated kurtosis: 2.905865
```

The median is less than the mean and the skewness is less than 1.0. This confirms the visual conclusion that the data is slightly skewed to the right. Before continuing with the analysis, let's recap what has been learned so far:

- The data appears to be stationary. This conclusion is based on the time series plot where no discernible trend with respect to time is found in the data.
- The data appears to be independent. This conclusion is from the autocorrelation plot.
- The distribution of the data is positively (right) skewed and unimodal. This conclusion is based on the histogram and from the statistical summary.

### .12.3 Hypothesizing and Testing a Distribution

The next steps involve the model fitting processes of hypothesizing distributions, estimating the parameters, and checking for goodness of fit. Distributions such as the

gamma, Weibull, and lognormal should be candidates for this situation based on the histogram. We will perform the analysis for the gamma distribution ‘by hand’ so that you can develop an understanding of the process. Then, the *fitdistrplus* package will be illustrated.

Here is what we are going to do:

1. Perform a chi-squared goodness of fit test.
2. Perform a K-S goodness of fit test.
3. Examine the P-P and Q-Q plots.

Recall that the Chi-Square Test divides the range of the data,  $(x_1, x_2, \dots, x_n)$ , into,  $k$ , intervals and tests if the number of observations that fall in each interval is close the expected number that should fall in the interval given the hypothesized distribution is the correct model.

Since a histogram tabulates the necessary counts for the intervals it is useful to begin the analysis by developing a histogram. Let  $b_0, b_1, \dots, b_k$  be the breakpoints (end points) of the class intervals such that  $(b_0, b_1], (b_1, b_2], \dots, (b_{k-1}, b_k]$  form  $k$  disjoint and adjacent intervals. The intervals do not have to be of equal width. Also,  $b_0$  can be equal to  $-\infty$  resulting in interval  $(-\infty, b_1]$  and  $b_k$  can be equal to  $+\infty$  resulting in interval  $(b_{k-1}, +\infty)$ . Define  $\Delta b_j = b_j - b_{j-1}$  and if all the intervals have the same width (except perhaps for the end intervals),  $\Delta b = \Delta b_j$ .

To count the number of observations that fall in each interval, define the following function:

$$c(\vec{x} \leq b) = \#\{x_i \leq b\} \quad i = 1, \dots, n \quad (17)$$

$c(\vec{x} \leq b)$  counts the number of observations less than or equal to  $x$ . Let  $c_j$  be the observed count of the  $x$  values contained in the  $j^{th}$  interval  $(b_{j-1}, b_j]$ . Then, we can determine  $c_j$  via the following equation:

$$c_j = c(\vec{x} \leq b_j) - c(\vec{x} \leq b_{j-1}) \quad (18)$$

Define  $h_j = c_j/n$  as the relative frequency for the  $j^{th}$  interval. Note that  $\sum_{j=1}^k h_j = 1$ . A plot of the cumulative relative frequency,  $\sum_{i=1}^j h_i$ , for each  $j$  is called a cumulative distribution plot. A plot of  $h_j$  should resemble the true probability distribution in shape because according to the mean value theorem of calculus.

$$p_j = P\{b_{j-1} \leq X \leq b_j\} = \int_{b_{j-1}}^{b_j} f(x)dx = \Delta b \times f(y) \text{ for } y \in (b_{j-1}, b_j)$$

Therefore, since  $h_j$  is an estimate for  $p_j$ , the shape of the distribution should be proportional to the relative frequency, i.e.  $h_j \approx \Delta b \times f(y)$ .

The number of intervals is a key decision parameter and will affect the visual quality of the histogram and ultimately the chi-squared test statistic calculations that are based on the tabulated counts from the histogram. In general, the visual display of the histogram is highly dependent upon the number of class intervals. If the widths of the intervals are too small, the histogram will tend to have a ragged shape. If the width of the intervals are too large, the resulting histogram will be very block like. Two common rules for setting the number of interval are:

1. Square root rule, choose the number of intervals,  $k = \sqrt{n}$ .
2. Sturges rule, choose the number of intervals,  $k = \lfloor 1 + \log_2(n) \rfloor$ .

A frequency diagram in R is very simple by using the *hist()* function. The *hist()* function provides the frequency version of histogram and *hist(x, freq=F)* provides the density version of the histogram. The *hist()* function will automatically determine breakpoints using the Sturges rule as its default. You can also provide your own breakpoints in a vector. The *hist()* function will automatically compute the counts associated with the intervals.

```
# make histogram with no plot
h = hist(y, plot = FALSE)
# show the histogram object components
h

## $breaks
## [1] 0 5 10 15 20 25 30 35
##
## $counts
## [1] 6 34 25 16 11 5 3
##
## $density
## [1] 0.012 0.068 0.050 0.032 0.022 0.010 0.006
##
## $mids
## [1] 2.5 7.5 12.5 17.5 22.5 27.5 32.5
##
## $xname
## [1] "y"
##
## $equidist
## [1] TRUE
##
## attr(),"class")
## [1] "histogram"
```

Notice how the *hist* command returns a result object. In the example, the result object is assigned to the variable *h*. By printing the result object, you can see all the tabulated

results. For example the variable `h$counts` shows the tabulation of the counts based on the default breakpoints.

```
h$counts
```

```
## [1] 6 34 25 16 11 5 3
```

The breakpoints are given by the variable `h$breaks`. Note that by default `hist` defines the intervals as right-closed, i.e.  $(b_{k-1}, b_k]$ , rather than left-closed,  $[b_{k-1}, b_k)$ . If you want left closed intervals, set the `hist` parameter, `right = FALSE`. The relative frequencies,  $h_j$  can be computed by dividing the counts by the number of observations, i.e.  $h$counts/length(y)$ .

The variable `h$density` holds the relative frequencies divided by the interval length. In terms of notation, this is,  $f_j = h_j / \Delta b_j$ . This is referred to as the density because it estimates the height of the probability density curve.

To define your own break points, put them in a vector using the `collect` command (for example: `b = c(0, 4, 8, 12, 16, 20, 24, 28, 32)`) and then specify the vector with the `breaks` option of the `hist` command if you do not want to use the default breakpoints. The following listing illustrates how to do this.

```
# set up some new break points
b = c(0,4,8,12,16,20,24,28,32)
b
```

```
## [1] 0 4 8 12 16 20 24 28 32
```

```
# make histogram with no plot for new breakpoints
hb = hist(y, breaks = b, plot = FALSE)
# show the histogram object components
hb
```

```
## $breaks
## [1] 0 4 8 12 16 20 24 28 32
##
## $counts
## [1] 6 16 30 17 12 9 5 5
##
## $density
## [1] 0.0150 0.0400 0.0750 0.0425 0.0300 0.0225 0.0125 0.0125
##
## $mids
## [1] 2 6 10 14 18 22 26 30
##
## $xname
```

```
## [1] "y"
##
## $equidist
## [1] TRUE
##
## attr(),"class")
## [1] "histogram"
```

You can also use the `cut()` function and the `table()` command to tabulate the counts by providing a vector of breaks and tabulate the counts using the `cut()` and the `table()` commands without using the `hist` command. The following listing illustrates how to do this.

```
#define the intervals
y.cut = cut(y, breaks=b)
# tabulate the counts in the intervals
table(y.cut)

## y.cut
##   (0,4]   (4,8]   (8,12]  (12,16]  (16,20]  (20,24]  (24,28]  (28,32]
##       6      16      30      17      12       9       5       5
```

By using the `hist` function in R, we have a method for tabulating the relative frequencies. In order to apply the chi-square test, we need to be able to compute the following test statistic:

$$\chi_0^2 = \sum_{j=1}^k \frac{(c_j - np_j)^2}{np_j} \quad (19)$$

where

$$p_j = P\{b_{j-1} \leq X \leq b_j\} = \int_{b_{j-1}}^{b_j} f(x)dx = F(b_j) - F(b_{j-1}) \quad (20)$$

Notice that  $p_j$  depends on  $F(x)$ , the cumulative distribution function of the hypothesized distribution. Thus, we need to hypothesize a distribution and estimate the parameters of the distribution.

For this situation, we will hypothesize that the task times come from a gamma distribution. Therefore, we need to estimate the shape ( $\alpha$ ) and the scale ( $\beta$ ) parameters. In order to do this we can use an estimation technique such as the method of moments or the maximum likelihood method. For simplicity and illustrative purposes, we will use the method of moments to estimate the parameters.

The method of moments is a technique for constructing estimators of the parameters that is based on matching the sample moments (e.g. sample average, sample variance, etc.) with the corresponding distribution moments. This method equates sample moments to population (theoretical) ones. Recall that the mean and variance of the gamma distribution are:

$$\begin{aligned} E[X] &= \alpha\beta \\ Var[X] &= \alpha\beta^2 \end{aligned}$$

Setting  $\bar{X} = E[X]$  and  $S^2 = Var[X]$  and solving for  $\alpha$  and  $\beta$  yields,

$$\begin{aligned} \hat{\alpha} &= \frac{(\bar{X})^2}{S^2} \\ \hat{\beta} &= \frac{S^2}{\bar{X}} \end{aligned}$$

Using the results,  $\bar{X} = 13.412$  and  $S^2 = 50.44895$ , yields,

$$\begin{aligned} \hat{\alpha} &= \frac{(\bar{X})^2}{S^2} = \frac{(13.412)^2}{50.44895} = 3.56562 \\ \hat{\beta} &= \frac{S^2}{\bar{X}} = \frac{50.44895}{13.412} = 3.761478 \end{aligned}$$

Then, you can compute the theoretical probability of falling in your intervals. Table 50 illustrates the computations necessary to compute the chi-squared test statistic.

**Table 50:** Chi-Squared Goodness of Fit Calculations

$j$	$b_{j-1}$	$b_j$	$c_j$	$F(b_{j-1})$	$F(b_j)$	$p_j$	$np_j$	$\frac{(c_j - np_j)^2}{np_j}$
1	0.00	5.00	6.00	0.00	0.08	0.08	7.89	0.45
2	5.00	10.00	34.00	0.08	0.36	0.29	28.54	1.05
3	10.00	15.00	25.00	0.36	0.65	0.29	28.79	0.50
4	15.00	20.00	16.00	0.65	0.84	0.18	18.42	0.32
5	20.00	25.00	11.00	0.84	0.93	0.09	9.41	0.27
6	25.00	30.00	5.00	0.93	0.97	0.04	4.20	0.15
7	30.00	35.00	3.00	0.97	0.99	0.02	1.72	0.96
8	35.00	$\infty$	0.00	0.99	1.00	0.01	1.03	1.03

Since the 7th and 8th intervals have less than 5 expected counts, we should combine them with the 6th interval. Computing the chi-square test statistic value over the 6 intervals yields:

$$\begin{aligned}
\chi_0^2 &= \sum_{j=1}^6 \frac{(c_j - np_j)^2}{np_j} \\
&= \frac{(6.0 - 7.89)^2}{7.89} + \frac{(34 - 28.54)^2}{28.54} + \frac{(25 - 28.79)^2}{28.79} + \frac{(16 - 18.42)^2}{218.42} \\
&\quad + \frac{(11 - 9.41)^2}{9.41} + \frac{(8 - 6.95)^2}{6.95} \\
&= 2.74
\end{aligned}$$

Since two parameters of the gamma were estimated from the data, the degrees of freedom for the chi-square test is  $3$  ( $\#$ intervals -  $\#$ parameters - 1 =  $6-2-1$ ). Computing the p-value yields  $P\{\chi_3^2 > 2.74\} = 0.433$ . Thus, given such a high p-value, we would not reject the hypothesis that the observed data is gamma distributed with  $\alpha = 3.56562$  and  $\beta = 3.761478$ .

The following listing provides a script that will compute the chi-square test statistic and its p-value within R.

```

a = mean(y)*mean(y)/var(y) #estimate alpha
b = var(y)/mean(y) #estmate beta
hy = hist(y, plot=FALSE) # make histogram
LL = hy$breaks # set lower limit of intervals
UL = c(LL[-1],10000) # set upper limit of intervals
FLL = pgamma(LL,shape = a, scale = b) #compute F(LL)
FUL = pgamma(UL,shape = a, scale = b) #compute F(UL)
pj = FUL - FLL # compute prob of being in interval
ej = length(y)*pj # compute expected number in interval
e = c(ej[1:5],sum(ej[6:8])) #combine last 3 intervals
cnts = c(hy$counts[1:5],sum(hy$counts[6:7])) #combine last 3 intervals
chissq = ((cnts-e)^2)/e #compute chi sq values
sumchisq = sum(chissq) # compute test statistic
df = length(e)-2-1 #compute degrees of freedom
pvalue = 1 - pchisq(sumchisq, df) #compute p-value
print(sumchisq) # print test statistic

## [1] 2.742749

print(pvalue) #print p-value

```

```

## [1] 0.4330114

```

Notice that we computed the same  $\chi^2$  value and p-value as when doing the calculations by hand.

### .12.4 Kolmogorov-Smirnov Test

The Kolmogorov-Smirnov (K-S) Test compares the hypothesized distribution,  $\hat{F}(x)$ , to the empirical distribution and does not depend on specifying intervals for tabulating the test statistic. The K-S test compares the theoretical cumulative distribution function (CDF) to the empirical CDF by checking the largest absolute deviation between the two over the range of the random variable. The K-S Test is described in detail in (Law, 2007), which also includes a discussion of the advantages/disadvantages of the test. For example, (Law, 2007) indicates that the K-S Test is more powerful than the Chi-Squared test and has the ability to be used on smaller sample sizes.

To apply the K-S Test, we must be able to compute the empirical distribution function. The empirical distribution is the proportion of the observations that are less than or equal to  $x$ . Recalling Equation (17), we can define the empirical distribution as in Equation (21).

$$\tilde{F}_n(x) = \frac{c(\vec{x} \leq x)}{n} \quad (21)$$

To formalize this definition, suppose we have a sample of data,  $x_i$  for  $i = 1, 2, \dots, n$  and we then sort this data to obtain  $x_{(i)}$  for  $i = 1, 2, \dots, n$ , where  $x_{(1)}$  is the smallest,  $x_{(2)}$  is the second smallest, and so forth. Thus,  $x_{(n)}$  will be the largest value. These sorted numbers are called the *order statistics* for the sample and  $x_{(i)}$  is the  $i^{th}$  order statistic.

Since the empirical distribution function is characterized by the proportion of the data values that are less than or equal to the  $i^{th}$  order statistic for each  $i = 1, 2, \dots, n$ , Equation (21) can be re-written as:

$$\tilde{F}_n(x_{(i)}) = \frac{i}{n} \quad (22)$$

The reason that this revised definition works is because for a given  $x_{(i)}$  the number of data values less than or equal to  $x_{(i)}$  will be  $i$ , by definition of the order statistic. For each order statistic, the empirical distribution can be easily computed as follows:

$$\begin{aligned} \tilde{F}_n(x_{(1)}) &= \frac{1}{n} \\ \tilde{F}_n(x_{(2)}) &= \frac{2}{n} \\ &\vdots \\ \tilde{F}_n(x_{(i)}) &= \frac{i}{n} \\ &\vdots \\ \tilde{F}_n(x_{(n)}) &= \frac{n}{n} = 1 \end{aligned}$$

A continuity correction is often used when defining the empirical distribution as follows:

$$\tilde{F}_n(x_{(i)}) = \frac{i - 0.5}{n}$$

This enhances the testing of continuous distributions. The sorting and computing of the empirical distribution is easy to accomplish in a spreadsheet program or in the statistical package R.

The K-S Test statistic,  $D_n$  is defined as  $D_n = \max\{D_n^+, D_n^-\}$  where:

$$\begin{aligned} D_n^+ &= \max_{1 \leq i \leq n} \left\{ \tilde{F}_n(x_{(i)}) - \hat{F}(x_{(i)}) \right\} \\ &= \max_{1 \leq i \leq n} \left\{ \frac{i}{n} - \hat{F}(x_{(i)}) \right\} \\ D_n^- &= \max_{1 \leq i \leq n} \left\{ \hat{F}(x_{(i)}) - \tilde{F}_n(x_{(i-1)}) \right\} \\ &= \max_{1 \leq i \leq n} \left\{ \hat{F}(x_{(i)}) - \frac{i-1}{n} \right\} \end{aligned}$$

The K-S Test statistic,  $D_n$ , represents the largest vertical distance between the hypothesized distribution and the empirical distribution over the range of the distribution. Table 83 contains critical values for the K-S test, where you reject the null hypothesis if  $D_n$  is greater than the critical value  $D_\alpha$ , where  $\alpha$  is the Type I significance level.

Intuitively, a large value for the K-S test statistic indicates a poor fit between the empirical and the hypothesized distributions. The null hypothesis is that the data comes from the hypothesized distribution. While the K-S Test can also be applied to discrete data, special tables must be used for getting the critical values. Additionally, the K-S Test in its original form assumes that the parameters of the hypothesized distribution are known, i.e. given without estimating from the data. Research on the effect of using the K-S Test with estimated parameters has indicated that it will be conservative in the sense that the actual Type I error will be less than specified.

The following R listing, illustrates how to compute the K-S statistic by hand (which is quite unnecessary) because you can simply use the `ks.test` command as illustrated.

```
j = 1:length(y) # make a vector to count y's
yj = sort(y) # sort the y's
Fj = pgamma(yj, shape = a, scale = b) #compute F(yj)
n = length(y)
D = max(max((j/n)-Fj),max(Fj - ((j-1)/n))) # compute K-S test statistic
print(D)
```

```
## [1] 0.05265431
```

```
ks.test(y, 'pgamma', shape=a, scale =b) # compute k-s test

##
## One-sample Kolmogorov-Smirnov test
##
## data: y
## D = 0.052654, p-value = 0.9444
## alternative hypothesis: two-sided
```

Based on the very high p-value of 0.9444, we should not reject the hypothesis that the observed data is gamma distributed with  $\alpha = 3.56562$  and  $\beta = 3.761478$ .

We have now completed the chi-squared goodness of fit test as well as the K-S test. The Chi-Squared test has more general applicability than the K-S Test. Specifically, the Chi-Squared test applies to both continuous and discrete data; however, it suffers from depending on the interval specification. In addition, it has a number of other shortcomings which are discussed in (Law, 2007). While the K-S Test can also be applied to discrete data, special tables must be used for getting the critical values. Additionally, the K-S Test in its original form assumes that the parameters of the hypothesized distribution are known, i.e. given without estimating from the data. Research on the effect of using the K-S Test with estimated parameters has indicated that it will be conservative in the sense that the actual Type I error will be less than specified. Additional advantage and disadvantage of the K-S Test are given in (Law, 2007). There are other statistical tests that have been devised for testing the goodness of fit for distributions. One such test is Anderson-Darling Test. (Law, 2007) describes this test. This test detects tail differences and has a higher power than the K-S Test for many popular distributions. It can be found as standard output in commercial distribution fitting software.

### .12.5 Visualizing the Fit

Another valuable diagnostic tool is to make probability-probability (P-P) plots and quantile-quantile (Q-Q) plots. A P-P Plot plots the empirical distribution function versus the theoretical distribution evaluated at each order statistic value. Recall that the empirical distribution is defined as:

$$\tilde{F}_n(x_{(i)}) = \frac{i}{n}$$

Alternative definitions are also used in many software packages to account for continuous data. As previously mentioned

$$\tilde{F}_n(x_{(i)}) = \frac{i - 0.5}{n}$$

is very common, as well as,

$$\tilde{F}_n(x_{(i)}) = \frac{i - 0.375}{n + 0.25}$$

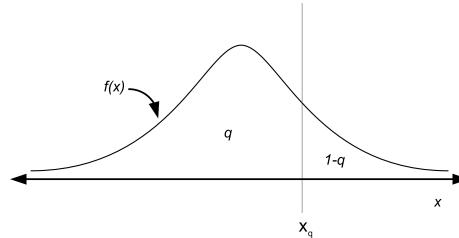
To make a P-P Plot, perform the following steps:

1. Sort the data to obtain the order statistics:  $(x_{(1)}, x_{(2)}, \dots x_{(n)})$
2. Compute  $\tilde{F}_n(x_{(i)}) = \frac{i - 0.5}{n} = q_i$  for  $i=1, 2, \dots n$
3. Compute  $\hat{F}(x_{(i)})$  for  $i=1, 2, \dots n$  where  $\hat{F}$  is the CDF of the hypothesized distribution
4. Plot  $\hat{F}(x_{(i)})$  versus  $\tilde{F}_n(x_{(i)})$  for  $i=1, 2, \dots n$

The Q-Q Plot is similar in spirit to the P-P Plot. For the Q-Q Plot, the quantiles of the empirical distribution (which are simply the order statistics) are plotted versus the quantiles from the hypothesized distribution. Let  $0 \leq q \leq 1$  so that the  $q^{th}$  quantile of the distribution is denoted by  $x_q$  and is defined by:

$$q = P(X \leq x_q) = F(x_q) = \int_{-\infty}^{x_q} f(u)du$$

As shown in Figure 65,  $x_q$  is that value on the measurement axis such that  $100q\%$  of the area under the graph of  $f(x)$  lies to the left of  $x_q$  and  $100(1-q)\%$  of the area lies to the right. This is the same as the inverse cumulative distribution function.



**Figure 65:** The Quantile of a Distribution

For example, the z-values for the standard normal distribution tables are the quantiles of that distribution. The quantiles of a distribution are readily available if the inverse CDF of the distribution is available. Thus, the quantile can be defined as:

$$x_q = F^{-1}(q)$$

where  $F^{-1}$  represents the inverse of the cumulative distribution function (not the reciprocal). For example, if the hypothesized distribution is  $N(0,1)$  then  $1.96 = \Phi^{-1}(0.975)$  so that  $x_{0.975} = 1.96$  where  $\Phi(z)$  is the CDF of the standard normal distribution. When you give a probability to the inverse of the cumulative distribution

function, you get back the corresponding ordinate value that is associated with the area under the curve, e.g. the quantile.

To make a Q-Q Plot, perform the following steps:

1. Sort the data to obtain the order statistics:  $(x_{(1)}, x_{(2)}, \dots x_{(n)})$
2. Compute  $q_i = \frac{i - 0.5}{n}$  for  $i=1, 2, \dots n$
3. Compute  $x_{q_i} = \hat{F}^{-1}(q_i)$  for where  $i = 1, 2, \dots n$  is the  $\hat{F}^{-1}$  inverse CDF of the hypothesized distribution
4. Plot  $x_{q_i}$  versus  $x_{(i)}$  for  $i = 1, 2, \dots n$

Thus, in order to make a P-P Plot, the CDF of the hypothesized distribution must be available and in order to make a Q-Q Plot, the inverse CDF of the hypothesized distribution must be available. When the inverse CDF is not readily available there are other methods to making Q-Q plots for many distributions. These methods are outlined in (Law, 2007). The following example will illustrate how to make and interpret the P-P plot and Q-Q plot for the hypothesized gamma distribution for the task times.

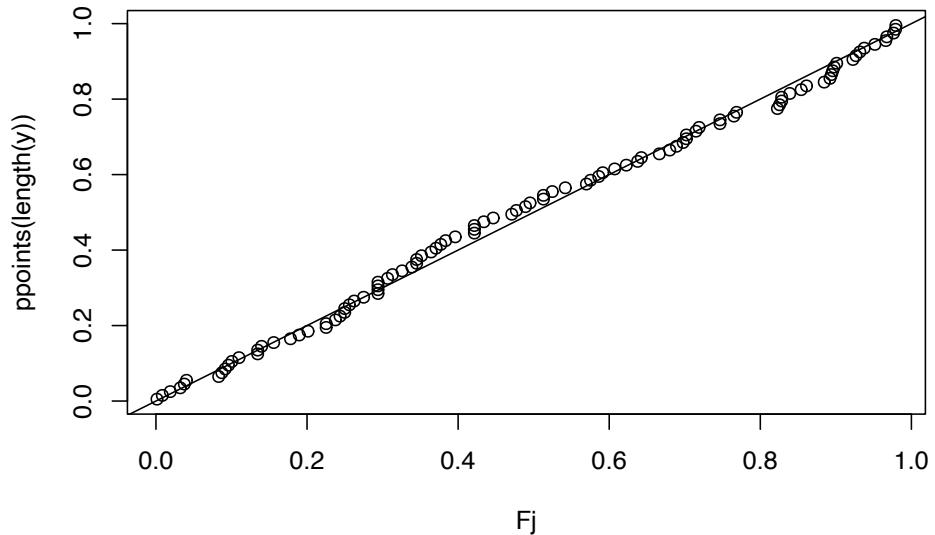
The following R listing will make the P-P and Q-Q plots for this situation.

```
plot(Fj,ppoints(length(y))) # make P-P plot
abline(0,1) # add a reference line to the plot
qqplot(y, qgamma(ppoints(length(y)), shape = a, scale = b)) # make Q-Q Plot
abline(0,1) # add a reference line to the plot
```

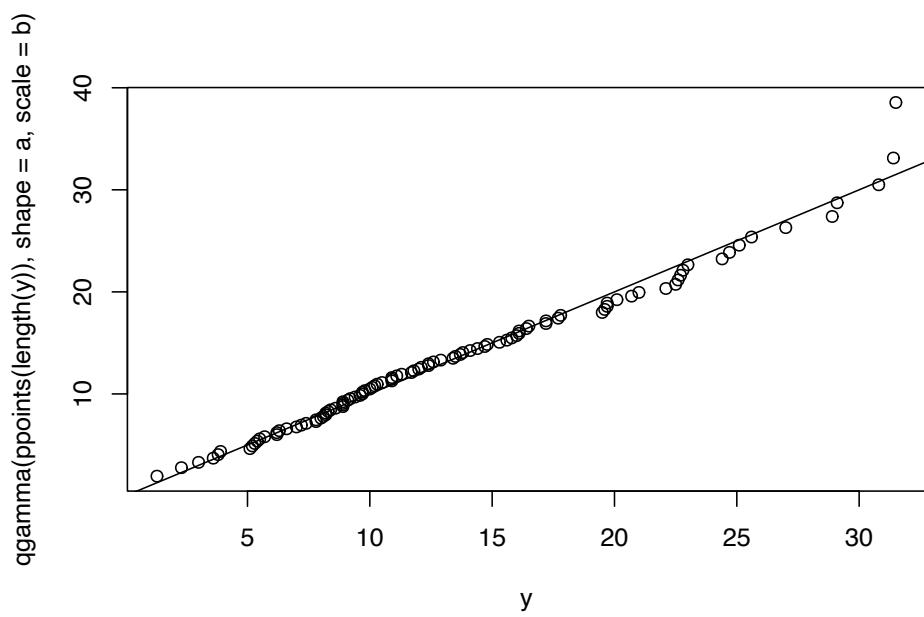
The function `ppoints()` in R will generate  $\tilde{F}_n(x_{(i)})$ . Then you can easily use the distribution function (with the “p”, as in `pgamma()`) to compute the theoretical probabilities. In R, the quantile function can be found by appending a “q” to the name of the available distributions. We have already seen `qt()` for the student-t distribution. For the normal, we use `qnorm()` and for the gamma, we use `qgamma()`. Search the R help for ‘distributions’ to find the other common distributions. The function `abline()` will add a reference line between 0 and 1 to the plot. Figure 66 illustrates the P-P plot.

The Q-Q plot should appear approximately linear with intercept zero and slope 1, i.e. a 45 degree line, if there is a good fit to the data. In addition, curvature at the ends implies too long or too short tails, convex or concave curvature implies asymmetry, and stragglers at either ends may be outliers. The P-P Plot should also appear linear with intercept 0 and slope 1. The `abline()` function was used to add the reference line to the plots. Figure 67 illustrates the Q-Q plot. As can be seen in the figures, both plots do not appear to show any significant departure from a straight line. Notice that the Q-Q plot is a little off in the right tail.

Now, that we have seen how to do the analysis ‘by hand’, let’s see how easy it can be using the `fitdistrplus` package. Notice that the `fitdist` command will fit the parameters of the distribution.



**Figure 66:** The P-P Plot for the Task Times with gamma(shape = 3.56, scale = 3.76)



**Figure 67:** The Q-Q Plot for the Task Times with gamma(shape = 3.56, scale = 3.76)

```
library(fitdistrplus)
fy = fitdist(y, "gamma")
print(fy)

## Fitting of the distribution ' gamma ' by maximum likelihood
## Parameters:
##           estimate Std. Error
## shape    3.4098479 0.46055722
## rate     0.2542252 0.03699365
```

Then, the *gofstat* function does all the work to compute the chi-square goodness of fit, K-S test statistic, as well as other goodness of fit criteria. The results lead to the same conclusion that we had before: the gamma distribution is a good model for this data.

```
gfy = gofstat(fy)
print(gfy)

## Goodness-of-fit statistics
##                               1-mle-gamma
## Kolmogorov-Smirnov statistic 0.04930008
## Cramer-von Mises statistic  0.03754480
## Anderson-Darling statistic  0.25485917
##
## Goodness-of-fit criteria
##                               1-mle-gamma
## Akaike's Information Criterion 663.3157
## Bayesian Information Criterion 668.5260
```

```
print(gfy$chisq) # chi-squared test statistic
```

```
## [1] 3.544766
```

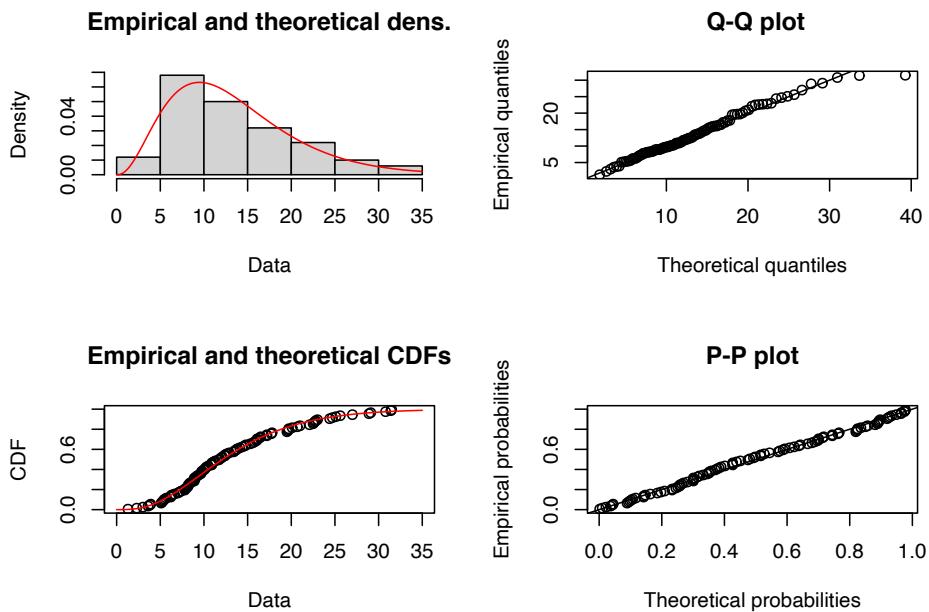
```
print(gfy$chisqvalue) # chi-squared p-value
```

```
## [1] 0.8956877
```

```
print(gfy$chisqdf) # chi-squared degrees of freedom
```

```
## [1] 8
```

Plotting the object returned from the *fitdist* command via (e.g. *plot(fy)*), produces a plot (Figure 68) of the empirical and theoretical distributions, as well as the P-P and Q-Q plots.



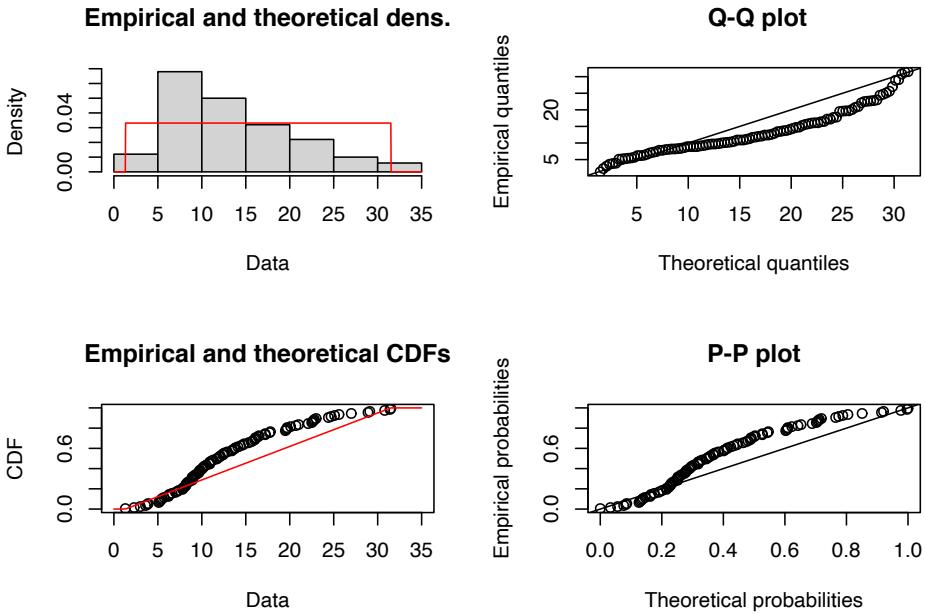
**Figure 68:** Distribution Plot from `fitdistrplus` for Gamma Distribution Fit of Computer Repair Times

Figure 69 illustrates the P-P and Q-Q plots if we were to hypothesize a uniform distribution. Clearly, the plots in Figure 69 illustrate that a uniform distribution is not a good model for the task times.

## .12.6 Using the Input Analyzer

In this section, we will use the Arena Input Analyzer to fit a distribution to service times collected for the pharmacy example. The Arena Input Analyzer is a separate program that comes with Arena. It is available as part of the free student edition of Arena.

Let  $X_i$  be the service time of the  $i^{th}$  customer, where the service time is defined as starting when the  $(i - 1)^{st}$  customer begins to drive off and ending when the  $i^{th}$  customer drives off after interacting with the pharmacist. In the case where there is no customer already in line when the  $i^{th}$  customer arrives, the start of the service can be defined as the point where the customer's car arrives to the beginning of the space in front of the pharmacist's window. Notice that in this definition, the time that it takes the car to pull up to the pharmacy window is being included. An alternative definition of service time might simply be the time between when the pharmacist asks the customer what they need until the time in which the customer gets the receipt. Both of these definitions are reasonable interpretations of service times and it is up to you to decide what sort of definition fits best with the overall modeling objectives. As you can see, input modeling is as much an art as it is a science.



**Figure 69:** Distribution Plot from `fitdistrplus` for Uniform Distribution Fit of Computer Repair Times

One hundred observations of the service time were collected using a portable digital assistant and are shown in Table 51 where the first observation is in row 1 column 1, the second observation is in row 2 column 1, the 21<sup>st</sup> observation is in row 1 column 2, and so forth. This data is available in the text file *PharmacyInputModelingExampleData.txt* that accompanies this chapter.

**Table 51:** Pharmacy Service Times

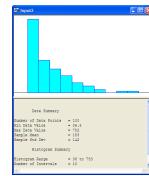
61	278.73	194.68	55.33	398.39
59.09	70.55	151.65	58.45	86.88
374.89	782.22	185.45	640.59	137.64
195.45	46.23	120.42	409.49	171.39
185.76	126.49	367.76	87.19	135.6
268.61	110.05	146.81	59	291.63
257.5	294.19	73.79	71.64	187.02
475.51	433.89	440.7	121.69	174.11
77.3	211.38	330.09	96.96	911.19
88.71	266.5	97.99	301.43	201.53
108.17	71.77	53.46	68.98	149.96
94.68	65.52	279.9	276.55	163.27
244.09	71.61	122.81	497.87	677.92
230.68	155.5	42.93	232.75	255.64

**Table 51:** Pharmacy Service Times

371.02	83.51	515.66	52.2	396.21
160.39	148.43	56.11	144.24	181.76
104.98	46.23	74.79	86.43	554.05
102.98	77.65	188.15	106.6	123.22
140.19	104.15	278.06	183.82	89.12
193.65	351.78	95.53	219.18	546.57

Prior to using the Input Analyzer, you should check the data if the observations are stationary (not dependent on time) and whether it is independent. We will leave that analysis as an exercise, since we have already illustrated the process using R in the previous sections.

After opening the Input Analyzer you should choose New from the File menu to start a new input analyzer data set. Then, using File > Data File > Use Existing, you can import the text file containing the data for analysis. The resulting import should leave the Input Analyzer looking like Figure 70.

**Figure 70:** Input Analyzer After Data Import

You should save the session, which will create a (.dft) file. Notice how the Input Analyzer automatically makes a histogram of the data and performs a basic statistical summary of the data. In looking at Figure 70, we might hypothesize a distribution that has long tail to the right, such as the exponential distribution.

The Input Analyzer will fit many of the common distributions that are available within Arena: Beta, Erlang, Exponential, Gamma, Lognormal, Normal, Triangular, Uniform, Weibull, Empirical, Poisson. In addition, it will provide the expression to be used within the Arena model. The fitting process within the Input Analyzer is highly dependent upon the intervals that are chosen for the histogram of the data. Thus, it is very important that you vary the number of intervals and check the sensitivity of the fitting process to the number of intervals in the histogram.

There are two basic methods by which you can perform the fitting process 1) individually for a specific distribution and 2) by fitting all of the possible distributions. Given the interval specification the Input Analyzer will compute a Chi-Squared goodness of fit statistic, Kolmogorov-Smirnov Test, and squared error criteria, all of which will be discussed in what follows.

Let's try to fit an exponential distribution to the observations. With the formerly imported data imported into an input window within the Input Analyzer, go to the Fit menu and select the exponential distribution. The resulting analysis is shown in the following listing.

```

Distribution Summary
Distribution: Exponential
Expression: 36 + EXPO(147)
Square Error: 0.003955

Chi Square Test
Number of intervals = 4
Degrees of freedom = 2
Test Statistic = 2.01
Corresponding p-value = 0.387

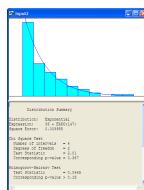
Kolmogorov-Smirnov Test
Test Statistic = 0.0445
Corresponding p-value > 0.15

Data Summary
Number of Data Points = 100
Min Data Value = 36.8
Max Data Value = 782
Sample Mean = 183
Sample Std Dev = 142

Histogram Summary
Histogram Range = 36 to 783
Number of Intervals = 10

```

The Input Analyzer has made a fit to the data and has recommended the Arena expression ( $36 + \text{EXPO}(147)$ ). What is this value 36? The value 36 is called the offset or location parameter. The visual fit of the data is shown in Figure 71



**Figure 71:** Histogram for Exponential Fit to Service Times

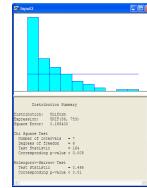
Recall the discussion in Section .5.4 concerning shifted distributions. Any distribution can have this additional parameter that shifts it along the x-axis. This can complicate parameter estimation procedures. The Input Analyzer has an algorithm which will

attempt to estimate this parameter. Generally, a reasonable estimate of this parameter can be computed via the floor of the minimum observed value,  $\lfloor \min(x_i) \rfloor$ . Is the model reasonable for the service time data? From the histogram with the exponential distribution overlaid, it appears to be a reasonable fit.

To understand the results of the fit, you must understand how to interpret the results from the Chi-Square Test and the Kolmogorov-Smirnov Test. The null hypothesis is that the data come from they hypothesized distribution versus the alternative hypothesis that the data do not come from the hypothesized distribution. The Input Analyzer shows the p-value of the tests.

The results of the distribution fitting process indicate that the p-value for the Chi-Square Test is 0.387. Thus, we would not reject the hypothesis that the service times come from the propose exponential distribution. For the K-S test, the p-value is greater than 0.15 which also does not suggest a serious lack of fit for the exponential distribution.

Figure 72 show the results of fitting a uniform distribution to the data.



**Figure 72:** Uniform Distribtuion and Histogram for Service Time Data

The following listing shows the results for the uniform distribution. The results show that the p-value for the K-S Test is smaller than 0.01, which indicates that the uniform distribution is probably not a good model for the service times.

```

Distribution Summary
Distribution: Uniform
Expression: UNIF(36, 783)
Square Error: 0.156400

Chi Square Test
Number of intervals = 7
Degrees of freedom = 6
Test Statistic = 164
Corresponding p-value < 0.005

Kolmogorov-Smirnov Test
Test Statistic = 0.495
Corresponding p-value < 0.01

Data Summary

```

```

Number of Data Points = 100
Min Data Value      = 36.8
Max Data Value      = 782
Sample Mean          = 183
Sample Std Dev       = 142

```

```

Histogram Summary
Histogram Range     = 36 to 783
Number of Intervals = 10

```

In general, you should be cautious of goodness-of-fit tests because they are unlikely to reject any distribution when you have little data, and they are likely to reject every distribution when you have lots of data. The point is, for whatever software that you use for your modeling fitting, you will need to correctly interpret the results of any statistical tests that are performed. Be sure to understand how these tests are computed and how sensitive the tests are to various assumptions within the model fitting process.

The final result of interest in the Input Analyzer's distribution summary output is the value labeled *Square Error*. This is the criteria that the Input Analyzer uses to recommend a particular distribution when fitting multiple distributions at one time to the data. The squared error is defined as the sum over the intervals of the squared difference between the relative frequency and the probability associated with each interval:

$$\text{Square Error} = \sum_{j=1}^k (h_j - \hat{p}_j)^2 \quad (23)$$

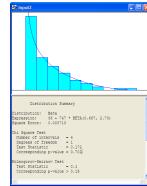
Table 52 shows the square error calculation for the fit of the exponential distribution to the service time data. The computed square error matches closely the value computed within the Input Analyzer, with the difference attributed to round off errors.

**Table 52:** Square Error Calculation

$j$	$c_j$	$b_j$	$h_j$	$\hat{p}_j$	$(h_j - \hat{p}_j)^2$
1	43	111	0.43	0.399	0.000961
2	19	185	0.19	0.24	0.0025
3	14	260	0.14	0.144	1.6E-05
4	10	335	0.1	0.0866	0.00018
5	6	410	0.06	0.0521	6.24E-05
6	4	484	0.04	0.0313	7.57E-05
7	2	559	0.02	0.0188	1.44E-06
8	0	634	0	0.0113	0.000128
9	1	708	0.01	0.0068	1.02E-05
10	1	783	0.01	0.00409	3.49E-05
Square Error					0.003969

When you select the Fit All option within the Input Analyzer, each of the possible distributions are fit in turn and the summary results computed. Then, the Input Analyzer ranks the distributions from smallest to largest according to the square error criteria. As you can see from the definition of the square error criteria, the metric is dependent upon the defining intervals. Thus, it is highly recommended that you test the sensitivity of the results to different values for the number of intervals.

Using the Fit All function results in the Input Analyzer suggesting that  $36 + 747 * \text{BETA}(0.667, 2.73)$  expression is a good fit of the model (Figure 73). The Window > Fit All Summary menu option will show the squared error criteria for all the distributions that were fit. Figure 74 indicates that the Erlang distribution is second in the fitting process according to the squared error criteria and the Exponential distribution is third in the rankings. Since the Exponential distribution is a special case of the Erlang distribution we see that their squared error criteria is the same. Thus, in reality, these results reflect the same distribution.



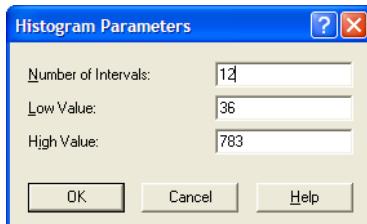
**Figure 73:** Fit All Beta Recommendation for Service Time Data

Fit All Summary	
Data File: C:\Documents and Settings\	
Function	Sq Error
Beta	0.000713
Erlang	0.00395
Exponential	0.00395
Lognormal	0.00472
Weibull	0.0063
Gamma	0.00737
Normal	0.0827
Triangular	0.0939
Uniform	0.156

**Figure 74:** Fit All Recommendation for Service Time Data

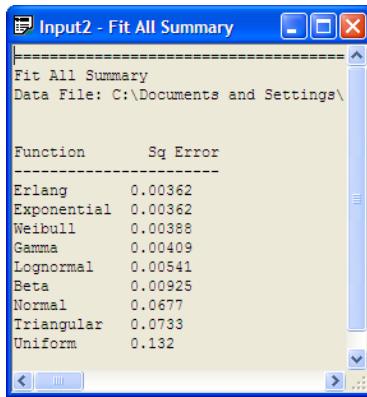
By using Options > Parameters > Histogram, the Histogram Parameters dialog can be used to change the parameters associated with the histogram as shown in Figure 75.

Changing the number of intervals to 12 results in the output provided in Figure 76, which indicates that the exponential distribution is a reasonable model based on the Chi-Square test, the K-S test, and the squared error criteria. You are encouraged to

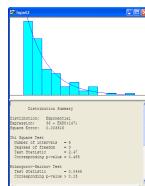


**Figure 75:** Changing the Histogram Parameters

check other fits with differing number of intervals. In most of the fits, the exponential distribution will be recommended. It is beginning to look like the exponential distribution is a reasonable model for the service time data.



**Figure 76:** Fit All with 12 Intervals



**Figure 77:** Exponential Fit with 12 Intervals

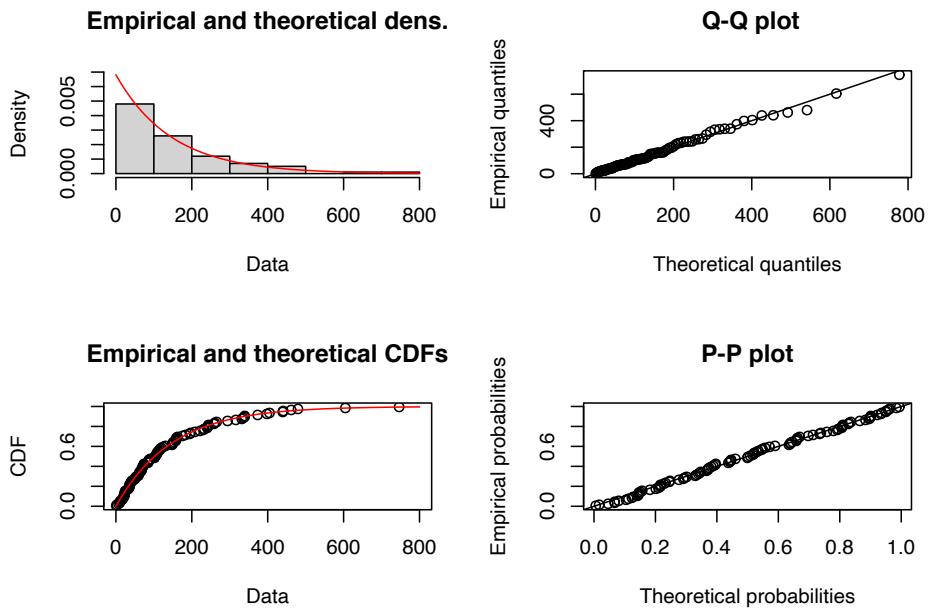
The Input Analyzer is convenient because it has the fit all summary and will recommend a distribution. However, it does not provide P-P plots and Q-Q plots. To do this, we can use the *fitdistrplus* package within R. Before proceeding with this analysis, there is a technical issue that must be addressed.

The proposed model from the Input Analyzer is:  $36 + \text{EXPO}(147)$ . That is, if  $X$  is a random variable that represents the service time then  $X \sim 36 + \text{EXPO}(147)$ , where 147 is the mean of the exponential distribution, so that  $\lambda = 1/147$ . Since 36 is a constant

in this expression, this implies that the random variable  $W = X - 36$ , has  $W \sim \text{EXPO}(147)$ . Thus, the model checking versus the exponential distribution can be done on the random variable  $W$ . That is, take the original data and subtract 36.

The following listing illustrates the R commands to make the fit, assuming that the data is in a file called *ServiceTimes.txt* within the R working directory. Figure 78 shows that the exponential distribution is a good fit for the service times based on the empirical distribution, P-P plot, and the Q-Q plot.

```
x = scan(file="ServiceTimes.txt") #read in the file
Read 100 items
w=x-36
library(fitdistrplus)
Loading required package: survival
Loading required package: splines
fw = fitdist(w, "exp")
fw
Fitting of the distribution ' exp ' by maximum likelihood
Parameters:
      estimate Std. Error
rate 0.006813019 0.0006662372
1/fw$estimate
      rate
146.7778
plot(fw)
```



**Figure 78:** Distribution Plot from *fitdistrplus* for Service Time Data

The P-P and Q-Q plots of the shifted data indicate that the exponential distribution is an excellent fit for the service time data.

## .13 Testing Uniform (0,1) Pseudo-Random Numbers

Now that we have seen the general process for fitting continuous distributions, this section will discuss the special case of testing for uniform (0,1) random variables. The reason that this is important is because these methods serve the basis for testing if pseudo-random numbers can reasonably be expected to perform as if they are U(0,1) random variates. Thus, this section provides an overview of what is involved in testing the statistical properties of random number generators. Essentially a random number generator is supposed to produce sequences of numbers that appear to be independent and identically distributed (IID)  $U(0, 1)$  random variables. The hypothesis that a sample from the generator is IID  $U(0, 1)$  must be made. Then, the hypothesis is subjected to various statistical tests. There are standard batteries of test, see for example (Soto, 1999), that are utilized. Typical tests examine:

- Distributional properties: e.g. Chi-Squared and Kolmogorov-Smirnov test
- Independence: e.g. Correlation tests, runs tests
- Patterns: e.g. Poker test, gap test

When considering the quality of random number generators, the higher dimensional properties of the sequence also need to be considered. For example, the serial test is a higher dimensional Chi-Squared test. Just like for general continuous distributions, the two major tests that are utilized to examine the distributional properties of sequences of pseudo-random numbers are the Chi-Squared Goodness of Fit test and the Kolmogorov-Smirnov test. In the case of testing for  $U(0, 1)$  random variates some simplifications can be made in computing the test statistics.

### .13.1 Chi-Squared Goodness of Fit Tests for Pseudo-Random Numbers

When applying the Chi-Squared goodness of fit to test if the data are  $U(0, 1)$ , the following is the typical procedure:

1. Divide the interval  $(0, 1)$  into  $k$  equally spaced classes so that  $\Delta b = b_j - b_{j-1}$  resulting in  $p_j = \frac{1}{k}$  for  $j = 1, 2, \dots, k$ . This results in the expected number in each interval being  $np_j = n \times \frac{1}{k} = \frac{n}{k}$
2. As a practical rule, the expected number in each interval  $np_j$  should be at least 5. Thus, in this case  $\frac{n}{k} \geq 5$  or  $n \geq 5k$  or  $k \leq \frac{n}{5}$ . Thus, for a given value of  $n$ , you should choose the number of intervals  $k \leq \frac{n}{5}$ ,
3. Since the parameters of the distribution are known  $a = 0$  and  $b = 1$ , then  $s = 0$ . Therefore, we reject  $H_0 : U_i \sim U(0, 1)$  if  $\chi^2_0 > \chi^2_{\alpha, k-1}$  or if the p-value is less than  $\alpha$

If  $p_j$  values are chosen as  $\frac{1}{k}$ , then Equation (24) can be rewritten as:

$$\chi_0^2 = \frac{k}{n} \sum_{j=1}^k \left( c_j - \frac{n}{k} \right)^2 \quad (24)$$

Let's apply these concepts to a small example.

---

**Example .13** (Testing 100 Pseudo-Random Numbers). Suppose we have 100 observations from a pseudo-random number generator. Perform a  $\chi^2$  test that the numbers are distributed  $U(0, 1)$ .

0.971	0.668	0.742	0.171	0.350	0.931	0.803	0.848	0.160	0.085
0.687	0.799	0.530	0.933	0.105	0.783	0.828	0.177	0.535	0.601
0.314	0.345	0.034	0.472	0.607	0.501	0.818	0.506	0.407	0.675
0.752	0.771	0.006	0.749	0.116	0.849	0.016	0.605	0.920	0.856
0.830	0.746	0.531	0.686	0.254	0.139	0.911	0.493	0.684	0.938
0.040	0.798	0.845	0.461	0.385	0.099	0.724	0.636	0.846	0.897
0.468	0.339	0.079	0.902	0.866	0.054	0.265	0.586	0.638	0.869
0.951	0.842	0.241	0.251	0.548	0.952	0.017	0.544	0.316	0.710
0.074	0.730	0.285	0.940	0.214	0.679	0.087	0.700	0.332	0.610
0.061	0.164	0.775	0.015	0.224	0.474	0.521	0.777	0.764	0.144

---

Since we have  $n = 100$  observations, the number of intervals should be less than or equal to 20. Let's choose  $k = 10$ . This means that  $p_j = 0.1$  for all  $j$ . The following table summarizes the computations for each interval for computing the chi-squared test statistic.

$j$	$b_{j-1}$	$b_j$	$p_j$	$c(\vec{x} \leq b_{j-1})$	$c(\vec{x} \leq b_j)$	$c_j$	$np_j$	$\frac{(c_j - np_j)^2}{np_j}$
1	0	0.1	0.1	0	13	13	10	0.9
2	0.1	0.2	0.1	13	21	8	10	0.4
3	0.2	0.3	0.1	21	28	7	10	0.9
4	0.3	0.4	0.1	28	35	7	10	0.9
5	0.4	0.5	0.1	35	41	6	10	1.6
6	0.5	0.6	0.1	41	50	9	10	0.1
7	0.6	0.7	0.1	50	63	13	10	0.9
8	0.7	0.8	0.1	63	77	14	10	1.6
9	0.8	0.9	0.1	77	90	13	10	0.9
10	0.9	1	0.1	90	100	10	10	0

Summing the last column yields:

$$\chi_0^2 = \sum_{j=1}^k \frac{(c_j - np_j)^2}{np_j} = 8.2$$

Computing the p-value for  $k - s - 1 = 10 - 0 - 1 = 9$  degrees of freedom, yields  $P\{\chi_9^2 > 8.2\} = 0.514$ . Thus, given such a high p-value, we would not reject the hypothesis that the observed data is  $U(0, 1)$ . This process can be readily implemented within a spreadsheet or performed using R. Assuming that the file, u01data.txt, contains the PRNs for this example, then the following R commands will perform the test:

```
data = scan(file="data/AppDistFitting/u01data.txt") # read in the file
b = seq(0,1, by = 0.1) # set up the break points
h = hist(data, b, right = FALSE, plot = FALSE) # tabulate the counts
chisq.test(h$counts) # perform the test
```

```
##
## Chi-squared test for given probabilities
##
## data: h$counts
## X-squared = 8.2, df = 9, p-value = 0.5141
```

Because we are assuming that the data is  $U(0, 1)$ , the chisq.test() function within R is simplified because its default is to assume that all data is equally likely. Notice that we get exactly the same result as we computed when doing the calculations manually.

### .13.2 Higher Dimensional Chi-Squared Test

The pseudo-random numbers should not only be uniformly distributed on the interval  $(0, 1)$  they should also be uniformly distributed within the unit square,  $\{(x, y) : x \in (0, 1), y \in (0, 1)\}$ , the unit cube, and so forth for higher number of dimensions  $d$ .

The serial test, described in (Law, 2007) can be used to assess the quality of the higher dimensional properties of pseudo-random numbers. Suppose the sequence of pseudo-random numbers can be formed into non-overlapping vectors each of size  $d$ . The vectors should be independent and identically distributed random vectors uniformly distributed on the  $d$ -dimensional unit hyper-cube. This motivates the development of the serial test for uniformity in higher dimensions. To perform the serial test:

1. Divide  $(0,1)$  into  $k$  sub-intervals of equal size.

2. Generate  $n$  vectors of pseudo-random numbers each of size  $d$ ,
- $$\vec{U}_1 = (U_1, U_2, \dots, U_d)$$

$$\vec{U}_2 = (U_{d+1}, U_{d+2}, \dots, U_{2d})$$

⋮

$$\vec{U}_n = (U_{(n-1)d+1}, U_{(n-1)d+2}, \dots, U_{nd})$$

3. Let  $c_{j_1, j_2, \dots, j_d}$  be the count of the number of  $\vec{U}_i$ 's having the first component in subinterval  $j_1$ , second component in subinterval  $j_2$  etc.

4. Compute

$$\chi_0^2(d) = \frac{k^d}{n} \sum_{j_1=1}^k \sum_{j_2=1}^k \dots \sum_{j_d=1}^k \left( c_{j_1, j_2, \dots, j_d} - \frac{n}{k^d} \right)^2 \quad (25)$$

5. Reject the hypothesis that the  $\vec{U}_i$ 's are uniformly distributed in the  $d$ -dimensional unit hyper-cube if  $\chi_0^2(d) > \chi_{\alpha, k^d-1}^2$  or if the p-value  $P\{\chi_{k^d-1}^2 > \chi_0^2(d)\}$  is less than  $\alpha$ .

(Law, 2007) provides an algorithm for computing  $c_{j_1, j_2, \dots, j_d}$ . This test examines how uniformly the random numbers fill-up the multi-dimensional space. This is a very important property when applying simulation to the evaluation of multi-dimensional integrals as is often found in the physical sciences.

---

**Example .14** (2-D Chi-Squared Test in R for  $U(0,1)$ ). Using the same data as in the previous example, perform a 2-Dimensional  $\chi^2$  Test using the statistical package R. Use 4 intervals for each of the dimensions.

\*\*\*

The following code listing is liberally commented for understanding the commands and essentially utilizes some of the classification and tabulation functionality in R to compute Equation (25). The code displayed here is available in the files associated with the chapter in file, 2dchisq.R.

```
nd = 100 #number of data points
data <- read.table("u01data.txt") # read in the data
d = 2 # dimensions to test
n = nd/d # number of vectors
m = t(matrix(data$V1,nrow=d)) # convert to matrix and transpose
b = seq(0,1, by = 0.25) # setup the cut points
xg = cut(m[,1],b,right=FALSE) # classify the x dimension
yg = cut(m[,2],b,right=FALSE) # classify the y dimension
xy = table(xg,yg) # tabulate the classifications
k = length(b) - 1 # the number of intervals
```

```

en = n/(k^d) # the expected number in an interval
vxy = c(xy) # convert matrix to vector for easier summing
vxymen = vxy-en # subtract expected number from each element
vxymen2 = vxymen*vxymen # square each element
schi = sum(vxymen2) # compute sum of squares
chi = schi/en # compute the chi-square test statistic
dof = (k^d) - 1 # compute the degrees of freedom
pv = pchisq(chi,dof, lower.tail=FALSE) # compute the p-value
# print out the results
cat("#observations = ", nd,"\n")
cat("#vectors = ", n, "\n")
cat("size of vectors, d = ", d, "\n")
cat("#intervals =", k, "\n")
cat("cut points = ", b, "\n")
cat("expected # in each interval, n/k^d = ", en, "\n")
cat("interval tabulation = \n")
print(xy)
cat("\n")
cat("chisq value =", chi," \n")
cat("dof =", dof," \n")
cat("p-value = ",pv," \n")

```

The results shown in the following listing are for demonstration purposes only since the expected number in the intervals is less than 5. However, you can see from the interval tabulation, that the counts for the 2-D intervals are close to the expected. The p-value suggests that we would not reject the hypothesis that there is non-uniformity in the pairs of the psuedo-random numbers. The R code can be generalized for a larger sample or for performing a higher dimensional test. The reader is encouraged to try to run the code for a larger data set.

Output:

```

#observations = 100
#vectors = 50
size of vectors, d = 2
#intervals = 4
cut points = 0 0.25 0.5 0.75 1
expected # in each interval, n/k^d = 3.125
interval tabulation =
      yg
      xg      [0,0.25) [0.25,0.5) [0.5,0.75) [0.75,1)
      [0,0.25)      5        0        3        2
      [0.25,0.5)    2        1        2        7
      [0.5,0.75)    3        3        3        7
      [0.75,1)      4        1        4        3

chisq value = 18.48
dof = 15

```

p-value = 0.2382715

### .13.3 Kolmogorov-Smirnov Test for Pseudo-Random Numbers

When applying the K-S test to testing pseudo-random numbers, the hypothesized distribution is the uniform distribution on 0 to 1. The CDF for a uniform distribution on the interval  $(a, b)$  is given by:

$$F(x) = P\{X \leq x\} = \frac{x - a}{b - a} \text{ for } a < x < b$$

Thus, for  $a = 0$  and  $b = 1$ , we have that  $F(x) = x$  for the  $U(0, 1)$  distribution. This simplifies the calculation of  $D_n^+$  and  $D_n^-$  to the following:

$$\begin{aligned} D_n^+ &= \max_{1 \leq i \leq n} \left\{ \frac{i}{n} - \hat{F}(x_{(i)}) \right\} \\ &= \max_{1 \leq i \leq n} \left\{ \frac{i}{n} - x_{(i)} \right\} \\ D_n^- &= \max_{1 \leq i \leq n} \left\{ \hat{F}(x_{(i)}) - \frac{i-1}{n} \right\} \\ &= \max_{1 \leq i \leq n} \left\{ x_{(i)} - \frac{i-1}{n} \right\} \end{aligned}$$


---

**Example .15** (K-S Test for  $U(0,1)$ ). Given the data from Example .13 test the hypothesis that the data appears  $U(0, 1)$  versus that it is not  $U(0, 1)$  using the Kolmogorov-Smirnov goodness of fit test at the  $\alpha = 0.05$  significance level.

\*\*\*

A good way to organize the computations is in a tabular form, which also facilitates the use of a spreadsheet. The second column is constructed by sorting the data. Recall that because we are testing the  $U(0, 1)$  distribution,  $F(x) = x$ , and thus the third column is simply  $F(x_{(i)}) = x_{(i)}$ . The rest of the columns follow accordingly.

$i$	$x_{(i)}$	$i/n$	$\frac{i-1}{n}$	$F(x_{(i)})$	$\frac{i}{n} - F(x_{(i)})$	$F(x_{(i)}) - \frac{i-1}{n}$
1	0.006	0.010	0.000	0.006	0.004	0.006
2	0.015	0.020	0.010	0.015	0.005	0.005
3	0.016	0.030	0.020	0.016	0.014	-0.004
4	0.017	0.040	0.030	0.017	0.023	-0.013
5	0.034	0.050	0.040	0.034	0.016	-0.006
$\times$	$\times$	$\times$	$\times$	$\times$	$\times$	$\times$
95	0.933	0.950	0.940	0.933	0.017	-0.007
96	0.938	0.960	0.950	0.938	0.022	-0.012

$i$	$x_{(i)}$	$i/n$	$\frac{i-1}{n}$	$F(x_{(i)})$	$\frac{i}{n} - F(x_{(i)})$	$F(x_{(i)}) - \frac{i-1}{n}$
97	0.940	0.970	0.960	0.940	0.030	-0.020
98	0.951	0.980	0.970	0.951	0.029	-0.019
99	0.952	0.990	0.980	0.952	0.038	-0.028
100	0.971	1.000	0.990	0.971	0.029	-0.019

Computing  $D_n^+$  and  $D_n^-$  yields

$$\begin{aligned} D_n^+ &= \max_{1 \leq i \leq n} \left\{ \frac{i}{n} - x_{(i)} \right\} \\ &= 0.038 \\ D_n^- &= \max_{1 \leq i \leq n} \left\{ x_{(i)} - \frac{i-1}{n} \right\} \\ &= 0.108 \end{aligned}$$

Thus, we have that

$$D_n = \max\{D_n^+, D_n^-\} = \max\{0.038, 0.108\} = 0.108$$

Referring to Table 83 and using the approximation for sample sizes greater than 35, we have that  $D_{0.05} \approx 1.36/\sqrt{n}$ . Thus,  $D_{0.05} \approx 1.36/\sqrt{100} = 0.136$ . Since  $D_n < D_{0.05}$ , we would not reject the hypothesis that the data is uniformly distribution over the range from 0 to 1.

The K-S test performed in the solution to Example .15 can also be readily performed using the statistical software *R*. Assuming that the file, *u01data.txt*, contains the data for this example, then the following R commands will perform the test:

```
data = scan(file="data/AppDistFitting/u01data.txt") # read in the file
ks.test(data,"punif",0,1) # perform the test
```

```
##
## One-sample Kolmogorov-Smirnov test
##
## data: data
## D = 0.10809, p-value = 0.1932
## alternative hypothesis: two-sided
```

Since the p-value for the test is greater than  $\alpha = 0.05$ , we would not reject the hypothesis that the data is  $U(0, 1)$ .

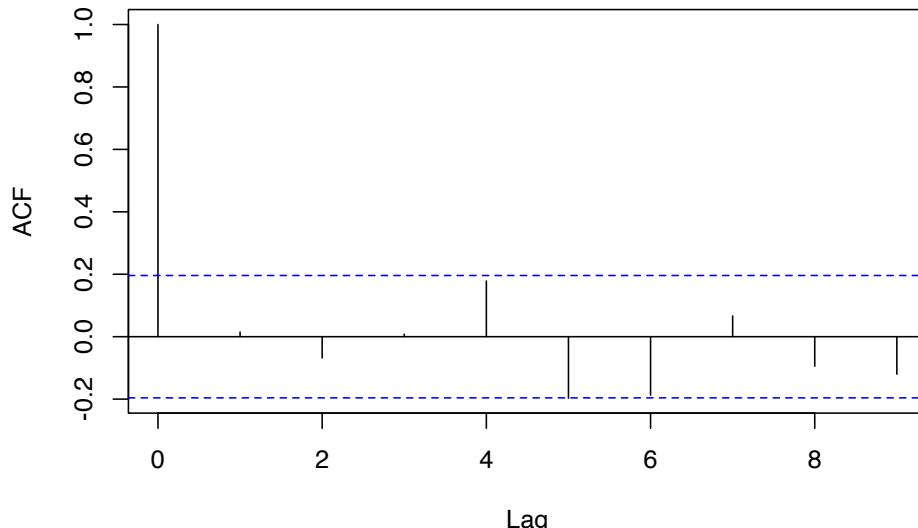
### .13.4 Testing for Independence and Patterns in Pseudo-Random Numbers

A full treatment for testing for independence and patterns within sequences of pseudo-random numbers is beyond the scope of this text. However, we will illustrate some of the basic concepts in this section.

As previously noted an analysis of the autocorrelation structure associated with the sequence of pseudo-random numbers forms a basis for testing dependence. A sample autocorrelation plot can be easily developed once the autocorrelation values have been estimated. Generally, the maximum lag should be set to no larger than one tenth of the size of the data set because the estimation of higher lags is generally unreliable.

An autocorrelation plot can be easily performed using the statistical software *R* for the data in Example .13. Using the `*acf()$` function in *R* makes it easy to get estimates of the autocorrelation estimates. The resulting plot is shown in Figure 79. The  $r_0$  value represents the estimated variance. As can be seen in the figure, the `acf()` function of *R* automatically places the confidence band within the plot. In this instance, since none of the  $r_k, k \geq 1$  are outside the confidence band, we can conclude that the data are likely independent observations.

**Autocorrelation Plot of Pseudo-Random Number Data**



**Figure 79:** Autocorrelation Plot for u01data.txt Data Set

```
rho
```

```
##
```

```
## Autocorrelations of series 'data', by lag
##
##      0     1     2     3     4     5     6     7     8     9
## 1.000  0.015 -0.068  0.008  0.179 -0.197 -0.187  0.066 -0.095 -0.120
```

The autocorrelation test examines serial dependence; however, sequences that do not have other kinds of patterns are also desired. For example, the runs test attempts to test ‘upward’ or ‘downward’ patterns. The runs up and down test count the number of runs up, down or sometimes just total number of runs versus expected number of runs. A run is a succession of similar events preceded and followed by a different event. The length of a run is the number of events in the run. The Table 56 illustrates how to compute runs up and runs down for a simple sequence of numbers. In the table, a sequence of ‘-’ indicates a run down and a sequence of ‘+’ indicates a run up. In the table, there are 8 runs (4 runs up and 4 runs down).

**Table 56:** Example Runs Up and Runs Down

0.90	0.13	0.27	0.41	0.71	0.28	0.18	0.22	0.26	0.19	0.61	0.87	0.95	0.21	0.79
-	+	+	+	-	-	-	-	+	-	+	+	+	-	+

Digit patterns can also be examined. The gap test counts the number of digits that appear between repetitions of a particular digit and uses a K-S test statistic to compare with the expected number of gaps. The poker test examines for independence based on the frequency with which certain digits are repeated in a series of numbers, (e.g. pairs, three of a kind, etc.). Banks et al. (2005) discusses how to perform these tests.

Does all this testing really matter? Yes! You should always know what pseudo-random number generator you are using within your simulation models and you should always know if the generator has passed a battery of statistical tests. The development and testing of random number generators is serious business. You should stick with well researched generators and reliable well tested software.

## .14 Additional Distribution Modeling Concepts

This section wraps up the discussion of input modeling by covering some additional topics that often come up during the modeling process.

Throughout the service time example, a continuous random variable was being modeled. But what do you do if you are modeling a discrete random variable? The basic complicating factor is that the only discrete distribution available within the Input Analyzer is the Poisson distribution and this option will only become active if the data input file only has integer values. The steps in the modeling process are essentially the same except that you cannot rely on the Input Analyzer. Commercial software will have options for fitting some of the common discrete distributions. The fitting process for a discrete distribution is simplified in one way because the bins for the frequency

diagram are naturally determined by the range of the random variable. For example, if you are fitting a geometric distribution, you need only tabulate the frequency of occurrence for each of the possible values of the random variable 1, 2, 3, 4, etc.. Occasionally, you may have to group bins together to get an appropriate number of observations per bin. The fitting process for the discrete case primarily centers on a straightforward application of the Chi-Squared goodness of fit test, which was outlined in this chapter, and is also covered in many introductory probability and statistics textbooks.

If you consider the data set as a finite set of values, then why can't you just reuse the data? In other words, why should you go through all the trouble of fitting a theoretical distribution when you can simply reuse the observed data, say for example by reading in the data from a file. There are a number of problems with this approach. The first difficulty is that the observations in the data set are a *sample*. That is, they do not necessarily cover all the possible values associated with the random variable. For example, suppose that only a sample of 10 observations was available for the service time problem. Furthermore, assume that the sample was as follows:

1	2	3	4	5	6	7	8	9	10
36.84	38.5	46.23	46.23	46.23	48.3	52.2	53.46	53.46	56.11

Clearly, this sample does not contain the high service times that were in the 100 sample case. The point is, if you resample from only these values, you will never get any values less than 36.84 or bigger than 56.11. The second difficulty is that it will be more difficult to experimentally vary this distribution within the simulation model. If you fit a theoretical distribution to the data, you can vary the parameters of the theoretical distribution with relative ease in any experiments that you perform. Thus, it is worthwhile to attempt to fit and use a reasonable input distribution.

But what if you cannot find a reasonable input model either because you have very limited data or because no model fits the data very well? In this situation, it is useful to try to use the data in the form of the empirical distribution. Essentially, you treat each observation in the sample as equally likely and randomly draw observations from the sample. In many situations, there are repeated observations within the sample (as above) and you can form a discrete empirical distribution over the values. If this is done for the sample of 10 data points, a discrete empirical distribution can be formed as shown in Table 58. Again, this limits us to only the values observed in the sample.

**Table 58:** Simple Empirical Distribution

X	PMF	CDF
36.84	0.1	0.1
38.5	0.1	0.2
46.23	0.3	0.5
48.3	0.1	0.6
53.46	0.2	0.8

$X$	PMF	CDF
55.33	0.1	0.9
56.11	0.1	1

One can also use the continuous empirical distribution, which interpolates between the distribution values.

What do you do if the analysis indicates that the data is dependent or that the data is non-stationary? Either of these situations can invalidate the basic assumptions behind the standard distribution fitting process. First, suppose that the data shows some correlation. The first thing that you should do is to verify that the data was correctly collected. The sampling plan for the data collection effort should attempt to ensure the collection of a random sample. If the data was from an automatic collection procedure then it is quite likely that there may be correlation in the observations. This is one of the hazards of using automatic data collection mechanisms. You then need to decide whether modeling the correlation is important or not to the study at hand. Thus, one alternative is to simply ignore the correlation and to continue with the model fitting process. This can be problematic for two reasons. First, the statistical tests within the model fitting process will be suspect, and second, the correlation may be an important part of the input modeling. For example, it has been shown that correlated arrivals and correlated service times in a simple queueing model can have significant effects on the values of the queue's performance measures. If you have a large enough data set, a basic approach is to form a random sample from the data set itself in order to break up the correlation. Then, you can proceed with fitting a distribution to the random sample; however, you should still model the dependence by trying to incorporate it into the random generation process. There are some techniques for incorporating correlation into the random variable generation process. An introduction to the topic is provided in (Banks et al., 2005).

If the data show non-stationary behavior, then you can attempt to model the dependence on time using time series models or other non-stationary models. Suffice to say, that these advanced techniques are beyond the scope of this text; however, the next section will discuss the modeling of a special non-stationary model, the non-homogeneous Poisson process, which is very useful for modeling time dependent arrival processes. For additional information on these methods, the interested reader is referred to (Law, 2007) and (Leemis and Park, 2006) or the references therein.

Finally, all of the above assumes that you have data from which you can perform an analysis. In many situations, you might have no data whatsoever either because it is too costly to collect or because the system that you are modeling does not exist. In the latter case, you can look at similar systems and see how their inputs were modeled, perhaps adopting some of those input models for the current situation. In either case, you might also rely on expert opinion. In this situation, you can ask an expert in the process to describe the characteristics of a probability distribution that might model the situation. This is where the uniform and the triangular distributions can be very

useful, since it is relatively easy to get an expert to indicate a minimum possible value, a maximum possible value, and even a most likely value.

Alternatively, you can ask the expert to assist in making an empirical distribution based on providing the chance that the random variable falls within various intervals. The breakpoints near the extremes are especially important to get. Table 1.8 presents a distribution for the service times based on this method.

**Table 59:** Breakpoint Based Empirical Distribution

X	PMF	CDF
(36, 100]	0.1	0.1
(100, 200]	0.1	0.2
(200 - 400]	0.3	0.6
(400, 600]	0.2	0.8
(600, $\infty$ )	0.2	1.0

Whether you have lots of data, little data, or no data, the key final step in the input modeling process is *sensitivity analysis*. Your ultimate goal is to use the input models to drive your larger simulation model of the system under study. You can spend significant time and energy collecting and analyzing data for an input model that has no significant effect on the output measures of interest to your study. You should start out with simple input models and incorporate them into your simulation model. Then, you can vary the parameters and characteristics of those models in an experimental design to assess how sensitive your output is to the changes in the inputs. If you find that the output is very sensitive to particular input models, then you can plan, collect, and develop better models for those situations. The amount of sensitivity is entirely modeler dependent. Remember that in this whole process, you are in charge, not the software. The software is only there to support your decision making process. Use the software to justify your art.

## .15 Summary

In this chapter you learned how to analyze data in order to model the input distributions for a simulation model. The input distributions drive the stochastic behavior of the simulation model.

The modeling of the input distributions requires:

- Understanding how the system being modeled works. This understanding improves overall model construction in an iterative fashion: model the system, observe some data, model the data, model the system, etc.

- Carefully collecting the data using well thought out collection and sampling plans
- Analyzing the data using appropriate statistical techniques
- Hypothesizing and testing appropriate probability distributions
- Incorporating the models in to your simulations

Properly modeling the inputs to the simulation model form a critical foundation to increasing the validity of the simulation model and subsequently, the credibility of the simulation outputs.

## .16 Exercises

The files referenced in the exercises are available in the files associated with this chapter.

---

**Exercise .44.** The observations available in the text file, *problem1.txt*, represent the count of the number of failures on a windmill turbine farm per year. Using the techniques discussed in the chapter recommend an input distribution model for this situation.

---

**Exercise .45.** The observations available in the text file, *problem2.txt*, represent the time that it takes to repair a windmill turbine on each occurrence in minutes. Using the techniques discussed in the chapter recommend an input distribution model for this situation.

---

**Exercise .46.** The observations available in the text file, *problem3.txt*, represent the time in minutes that it takes a repair person to drive to the windmill farm to repair a failed turbine. Using the techniques discussed in the chapter recommend an input distribution model for this situation.

---

**Exercise .47.** The observations available in the text file, *problem4.txt*, represent the time in seconds that it takes to service a customer at a movie theater counter. Using the techniques discussed in the chapter recommend an input distribution model for this situation.

---

**Exercise .48.** The observations available in the text file, *problem5.txt*, represent the time in hours between failures of a critical piece of computer testing equipment. Using the techniques discussed in the chapter recommend an input distribution model for this situation.

---

**Exercise .49.** The observations available in the text file, *problem6.txt*, represent the time in minutes associated with performing a lube, oil and maintenance check at the local Quick Oil Change Shop. Using the techniques discussed in the chapter recommend an input distribution model for this situation.

---

**Exercise .50.** If  $Z \sim N(0, 1)$ , and  $Y = \sum_{i=1}^k Z_i^2$  then  $Y \sim \chi_k^2$ , where  $\chi_k^2$  is a chi-squared random variable with  $k$  degrees of freedom. Setup an model to generate  $n = 32, 64, 128, 256, 1024$  observations of  $Y$  with  $k = 5$ . For each sample, fit a distribution to the sample.

---

**Exercise .51.** Consider the following sample (also found in *problem8.txt*) that represents the time (in seconds) that it takes a hematology cell counter to complete a test on a blood sample.

23.79	75.51	29.89	2.47	32.37
29.72	84.69	45.66	61.46	67.23
94.96	22.68	86.99	90.84	56.49
30.45	69.64	17.09	33.87	98.04
12.46	8.42	65.57	96.72	33.56
35.25	80.75	94.62	95.83	38.07
14.89	54.80	95.37	93.76	83.64
50.95	40.47	90.58	37.95	62.42
51.95	65.45	11.17	32.58	85.89
65.36	34.27	66.53	78.64	58.24

- a. Test the hypothesis that these data are drawn from a uniform distribution at a 95% confidence level assuming that the interval is between 0 and 100.
- b. The interval of the distribution is between  $a$  and  $b$ , where  $a$  and  $b$  are unknown parameters estimated from the data.
- c. Consider the following output for fitting a uniform distribution to a data set with the Arena Input Analyzer. Would you reject or not reject the hypothesis that the data is uniformly distributed.

```

Distribution Summary
Distribution: Uniform
Expression: UNIF(36, 783)
Square Error: 0.156400

Chi Square Test
Number of intervals = 7
Degrees of freedom = 6
Test Statistic = 164
Corresponding p-value < 0.005

Kolmogorov-Smirnov Test
Test Statistic = 0.495
Corresponding p-value < 0.01

Data Summary
Number of Data Points = 100
Min Data Value = 36.8
Max Data Value = 782
Sample Mean = 183
Sample Std Dev = 142

Histogram Summary
Histogram Range = 36 to 783
Number of Intervals = 10

```

---

**Exercise .52.** Consider the following frequency data on the number of orders received per day by a warehouse.

$j$	$c_j$	$c_j$	$np_j$	$\frac{(c_j - np_j)^2}{np_j}$
1	0	10		
2	1	42		
3	2	27		
4	3	12		
5	4	6		
6	5 or more	3		
	Totals	100		

- Compute the sample mean for this data.
- Perform a  $\chi^2$  goodness of fit test to test the hypothesis (use a 95% confidence level) that the data is Poisson distributed. Complete the provided table to show your calculations.

---

**Exercise .53.** The number of electrical outlets in a prefabricated house varies between 10 and 22 outlets. Since the time to perform the install depends on the number of outlets, data was collected to develop a probability distribution for this variable. The data set is given below and also found in file *problem10.txt*:

13	12	12	12	12	14	22	13	16	15	15	15	21	18	12	17	14	13	11	12	14	
10	10	16	12	13	11	13	11	11	12	13	16	15	12	11	14	11	12	11	11	13	17

Fit a probability model to the number of electrical outlets per prefabricated house.

---

**Exercise .54.** Test the following fact by generating instances of  $Y = \sum_{i=1}^r X_i$ , where  $X_i \sim \text{expo}(\beta)$  and  $\beta = E[X_i]$ . Using  $r = 5$  and  $\beta = 2$ . Be careful to specify the correct parameters for the exponential distribution function. The exponential distribution takes in the mean of the distribution as its parameter. Generate 10, 100, 1000 instances of  $Y$ . Perform hypothesis tests to check  $H_0 : \mu = \mu_0$  versus  $H_1 : \mu \neq \mu_0$  where  $\mu$  is the true mean of  $Y$  for each of the sample sizes generated. Use the same samples to fit a distribution using the Input Analyzer. Properly interpret the statistical results supplied by the Input Analyzer.

---

**Exercise .55.** Suppose that we are interested in modeling the arrivals to Sly's BBQ Food Truck during 11:30 am to 1:30 pm in the downtown square. During this time a team of undergraduate students has collected the number of customers arriving to the truck for 10 different periods of the 2 hour time frame for each day of the week. The data is given below and also found in file *problem12.csv*

Obs#	M	T	W	R	F	S	SU
1	13	4	4	3	8	8	9
2	6	5	7	7	5	6	8
3	7	14	10	5	5	5	10
4	12	6	10	5	12	7	4
5	6	8	8	5	4	11	9
6	10	6	9	3	3	6	4
7	9	5	5	5	7	5	4
8	7	10	11	9	7	10	13
9	8	4	2	7	6	5	7
10	9	2	6	8	7	4	9

1. Visualize the data.

2. Check if the day of the week influences the statistical properties of the count data.
  3. Tabulate the frequency of the count data.
  4. Estimate the mean rate parameter of the hypothesized Poisson distribution.
  5. Perform goodness of fit tests to test the hypothesis that the number of arrivals for the interval 11:30 am to 1:30 pm has a Poisson distribution versus the alternative that it does not have a Poisson distribution.
- 

**Exercise .56.** A copy center has one fast copier and one slow copier. The copy time per page for the fast copier is thought to be lognormally distributed with a mean of 1.6 seconds and a standard deviation of 0.3 seconds. A co-op Industrial Engineering student has collected some time study data on the time to copy a page for the slow copier. The times, in seconds, are given in the data file *problem13.txt*

---



# Discrete Distributions

## Bernoulli

Bernoulli	$Ber(p)$
Parameters:	$0 < p < 1$ , probability of success
PMF:	$P[X = 1] = p, P[X = 0] = 1 - p$
Expected Value:	$E[X] = p$
Variance:	$Var[X] = p(1 - p)$
Arena:	DISC(p,1,1.0,0[,stream])
Spreadsheet Generation:	= IF(RAND() < p, 1, 0)
Modeling:	the number of successes in one trial

## Binomial

Binomial	$Binom(n, p)$
Parameters:	$0 < p < 1$ , probability of success, $n$ , number of trials
PMF:	$P[X = x] = \binom{n}{x} p^x (1 - p)^{n-x} \quad x = 0, 1, \dots, n$
Expected Value:	$E[X] = np$
Variance:	$Var[X] = np(1 - p)$
Arena:	Not available, use convolution of Bernoulli
Spreadsheet Generation:	= BINOM.INV(n,p,RAND())
Modeling:	the number of successes in $n$ trials

## Shifted Geometric

Shifted Geometric	Shifted Geo( $p$ )
Parameters:	$0 < p < 1$ , probability of success
PMF:	$P[X = x] = p(1 - p)^{x-1} \quad x = 1, 2, \dots,$
Expected Value:	$E[X] = 1/p$
Variance:	$Var[X] = (1 - p)/p^2$
Arena:	1 + AINT(LN(1-UNIF(0,1))/LN(1-p))

<b>Shifted Geometric</b>	Shifted Geo( $p$ )
Spreadsheet Generation:	= 1 + INT(LN(1-RAND())/LN(1-p))
Modeling:	the number of trials until the first success

### Negative Binomial

<b>Negative Binomial Defn. 1</b> NB1( $r, p$ )	
Parameters:	$0 < p < 1$ , probability of success, $r^{th}$ success
PMF:	$P[X = x] = \binom{x-1}{r-1} p^r (1-p)^{x-r} \quad x = r, r+1, \dots$
Expected Value:	$E[X] = r/p$
Variance:	$Var[X] = r(1-p)/p^2$
Arena:	use convolution of shifted geometric
Spreadsheet Generation:	not available, count until $r^{th}$ success
Modeling:	the number of trials until the $r^{th}$ success

<b>Negative Binomial Defn. 2</b> NB2( $r, p$ )	
Parameters:	$0 < p < 1$ , probability of success, $r^{th}$ success
PMF:	$P[Y = y] = \binom{y+r-1}{r-1} p^r (1-p)^y \quad y = 0, 1, \dots$
Expected Value:	$E[Y] = r(1-p)/p$
Variance:	$Var[Y] = r(1-p)/p^2$
Arena:	use convolution of geometric
Spreadsheet Generation:	not available, count until $r^{th}$ success
Modeling:	the number of failures prior to the $r^{th}$ success

### Poisson

<b>Poisson</b>	Pois( $\lambda$ )
Parameters:	$\lambda > 0$
PMF:	$P[X = x] = \frac{e^{-\lambda} \lambda^x}{x!} \quad x = 0, 1, \dots$
Expected Value:	$E[X] = \lambda$
Variance:	$Var[X] = \lambda$
Arena:	POIS( $\lambda$ [, Stream])
Spreadsheet Generation:	not available, approximate with lookup table approach
Modeling:	the number of occurrences during a period of time

### Discrete Uniform

<b>Discrete Uniform</b>	$DU(a, b)$
Parameters:	$a \leq b$
PMF:	$P[X = x] = \frac{1}{b-a+1} \quad x = a, a+1, \dots, b$
Expected Value:	$E[X] = (b+a)/2$
Variance:	$Var[X] = ((b-a+1)^2 - 1) / 12$
Arena:	$a + \text{AINT}((b-a+1)*\text{UNIF}(0,1))$
Spreadsheet Generation:	=RANDBETWEEN(a,b)
Modeling:	equal occurrence over a range of integers



# Continuous Distributions

## Continuous Uniform

Uniform	$U(a, b)$
Parameters:	$a = \text{minimum}$ , $b = \text{maximum}$ , $-\infty < a < b < \infty$
PDF:	$f(x) = \frac{1}{b-a}$ for $a \leq x \leq b$
CDF:	$F(x) = \frac{x-a}{b-a}$ if $a \leq x \leq b$
Inverse CDF:	$F^{-1}(p) = a + p(b-a)$ if $0 < p < 1$
Expected Value:	$E[X] = \frac{a+b}{2}$
Variance:	$V[X] = \frac{(b-a)^2}{12}$
Arena:	UNIF(a,b[,Stream])
Spreadsheet Generation:	= a + RAND()*(b-a)
Modeling:	assumes equally likely across the range, when you have lack of data, task times

## Normal

Normal	$N(\mu, \sigma^2)$
Parameters:	$-\infty < \mu < +\infty$ (mean), $\sigma^2 > 0$ (variance)
CDF:	No closed form
Inverse CDF:	No closed form
Expected Value:	$E[X] = \mu$
Variance:	$Var[X] = \sigma^2$
Arena:	NORM( $\mu, \sigma^2$ [,Stream])
Spreadsheet Generation:	= NORM.INV(RAND(), $\mu, \sigma$ )
Modeling:	task times, errors

## Exponential

<b>Exponential</b>	EXPO( $1/\lambda$ )
Parameters:	$\lambda > 0$
PDF:	$f(x) = \lambda e^{-\lambda x}$ if $x \geq 0$
CDF:	$F(x) = 1 - e^{-\lambda x}$ if $x \geq 0$
Inverse CDF:	$F^{-1}(p) = (-1/\lambda) \ln(1-p)$ if $0 < p < 1$
Expected Value:	$E[X] = \theta = 1/\lambda$
Variance:	$Var[X] = 1/\lambda^2$
Arena:	EXPO( $\theta$ , Stream])
Spreadsheet Generation:	=(-1/ $\lambda$ )LN(1-RAND())
Modeling:	time between arrivals, time to failure highly variable task time

**Weibull**

<b>Weibull</b>	WEIB( $\beta, \alpha$ )
Parameters:	$\beta > 0$ (scale), $\alpha > 0$ (shape)
CDF:	$F(x) = 1 - e^{-(x/\beta)^\alpha}$ if $x \geq 0$
Inverse CDF:	$F^{-1}(p) = \beta [-\ln(1-p)]^{1/\alpha}$ if $0 < p < 1$
Expected Value:	$E[X] = \left(\frac{\beta}{\alpha}\right) \Gamma\left(\frac{1}{\alpha}\right)$
Variance:	$Var[X] = \left(\frac{\beta^2}{\alpha}\right) \left\{ 2\Gamma\left(\frac{2}{\alpha}\right) - \left(\frac{1}{\alpha}\right) \left(\Gamma\left(\frac{1}{\alpha}\right)\right)^2 \right\}$
Arena:	WEIB(scale, shape[, Stream])
Spreadsheet Generation:	=( $\beta$ )(-LN(1 - RAND())) $\wedge$ (1/ $\alpha$ )
Modeling:	task times, time to failure

**Erlang**

<b>Erlang</b>	Erlang( $r, \beta$ )
Parameters:	$r > 0$ , integer, $\beta > 0$ (scale)
CDF:	$F(x) = 1 - e^{(-x/\beta)} \sum_{j=0}^{r-1} \frac{(x/\beta)^j}{j!}$ if $x \geq 0$
Inverse CDF:	No closed form
Expected Value:	$E[X] = r\beta$
Variance:	$Var[X] = r\beta^2$
Arena:	ERLA( $E[X], r$ , Stream])
Spreadsheet Generation:	=GAMMA.INV(RAND(), $r, \beta$ )
Modeling:	task times, lead time, time to failure,

**Gamma**

<b>Gamma</b>	$\text{Gamma}(\alpha, \beta)$
Parameters:	$\alpha > 0$ , shape, $\beta > 0$ (scale)
CDF:	No closed form
Inverse CDF:	No closed form
Expected Value:	$E[X] = \alpha\beta$
Variance:	$Var[X] = \alpha\beta^2$
Arena:	GAMM(scale, shape[, stream])
Spreadsheet Generation:	=GAMMA.INV(RAND(), $\alpha$ , $\beta$ )
Modeling:	task times, lead time, time to failure,

**Beta**

<b>Beta</b>	$\text{BETA}(\alpha_1, \alpha_2)$
Parameters:	shape parameters $\alpha_1 > 0, \alpha_2 > 0$
CDF:	No closed form
Inverse CDF:	No closed form
Expected Value:	$E[X] = \frac{\alpha_1}{\alpha_1 + \alpha_2}$
Variance:	$Var[X] = \frac{\alpha_1\alpha_2}{(\alpha_1 + \alpha_2)^2(\alpha_1 + \alpha_2 + 1)}$
Arena:	BETA( $\alpha_1, \alpha_2$ [, stream])
Spreadsheet Generation:	BETA.INV(RAND(), $\alpha_1$ , $\alpha_2$ )
Modeling:	activity time when data is limited, probabilities

**Lognormal**

<b>Lognormal</b>	$\text{LOGN}(\mu_l, \sigma_l)$
Parameters:	$\mu = \ln \left( \mu_l^2 / \sqrt{\sigma_l^2 + \mu_l^2} \right) \quad \sigma^2 = \ln \left( (\sigma_l^2 / \mu_l^2) + 1 \right)$
CDF:	No closed form
Inverse CDF:	No closed form
Expected Value:	$E[X] = \mu_l = e^{\mu + \sigma^2/2}$
Variance:	$Var[X] = \sigma_l^2 = e^{2\mu + \sigma^2} (e^{\sigma^2} - 1)$
Arena:	LOGN( $\mu_l, \sigma_l$ [, stream])
Spreadsheet Generation:	LOGNORM.INV(RAND(), $\mu$ , $\sigma$ )
Modeling:	task times, time to failure

**Triangular**

<b>Triangular</b>	TRIA(a, m, b)
Parameters:	a = minimum, m = mode, b = maximum
CDF:	$F(x) = \frac{(x-a)^2}{(b-a)(m-a)}$ for $a \leq x \leq m$ $F(x) = 1 - \frac{(b-x)^2}{(b-a)(b-m)}$ for $m < x \leq b$
Inverse CDF:	$F^{-1}(p) = a + \sqrt{(b-a)(m-a)p}$ for $0 < p < \frac{m-a}{b-a}$ $F^{-1}(p) = b - \sqrt{(b-a)(b-m)(1-p)}$ for $\frac{m-a}{b-a} \leq p$
Expected Value:	$E[X] = (a + m + b)/3$
Variance:	$Var[X] = \frac{a^2 + b^2 + m^2 - ab - am - bm}{18}$
Arena:	TRIA(min, mode, max[, stream])
Spreadsheet Generation:	implement $F^{-1}(p)$ as VBA function
Modeling:	task times, activity time when data is limited

# Statistical Tables

**Table 79:** Cumulative Standard Normal Distribution  $z = -3.99$  to 0.0

z	-0.09	-0.08	-0.07	-0.06	-0.05	-0.04	-0.03	-0.02	-0.01	0.00
-3.9	0.000033	0.000034	0.000036	0.000037	0.000039	0.000041	0.000042	0.000044	0.000046	0.000048
-3.8	0.000050	0.000052	0.000054	0.000057	0.000059	0.000062	0.000064	0.000067	0.000069	0.000071
-3.7	0.000075	0.000078	0.000082	0.000085	0.000088	0.000092	0.000096	0.000100	0.000104	0.000108
-3.6	0.000112	0.000117	0.000121	0.000126	0.000131	0.000136	0.000142	0.000147	0.000153	0.000160
-3.5	0.000165	0.000172	0.000178	0.000185	0.000193	0.000200	0.000208	0.000216	0.000224	0.000233
-3.4	0.000242	0.000251	0.000260	0.000270	0.000280	0.000291	0.000302	0.000313	0.000325	0.000340
-3.3	0.000349	0.000362	0.000376	0.000390	0.000404	0.000419	0.000434	0.000450	0.000466	0.000482
-3.2	0.000501	0.000519	0.000538	0.000557	0.000577	0.000598	0.000619	0.000641	0.000664	0.000690
-3.1	0.000711	0.000736	0.000762	0.000789	0.000816	0.000845	0.000874	0.000904	0.000935	0.000966
-3	0.001001	0.001035	0.001070	0.001107	0.001144	0.001183	0.001223	0.001264	0.001306	0.001350
-2.9	0.001395	0.001441	0.001489	0.001538	0.001589	0.001641	0.001695	0.001750	0.001807	0.001867
-2.8	0.001926	0.001988	0.002052	0.002118	0.002186	0.002256	0.002327	0.002401	0.002477	0.002557
-2.7	0.002635	0.002718	0.002803	0.002890	0.002980	0.003072	0.003167	0.003264	0.003364	0.003474
-2.6	0.003573	0.003681	0.003793	0.003907	0.004025	0.004145	0.004269	0.004396	0.004527	0.004667
-2.5	0.004799	0.004940	0.005085	0.005234	0.005386	0.005543	0.005703	0.005868	0.006037	0.006217
-2.4	0.006387	0.006569	0.006756	0.006947	0.007143	0.007344	0.007549	0.007760	0.007976	0.008207
-2.3	0.008424	0.008656	0.008894	0.009137	0.009387	0.009642	0.009903	0.010170	0.010444	0.010724
-2.2	0.011011	0.011304	0.011604	0.011911	0.012224	0.012545	0.012874	0.013209	0.013553	0.013907
-2.1	0.014262	0.014629	0.015003	0.015386	0.015778	0.016177	0.016586	0.017003	0.017429	0.017857
-2	0.018309	0.018763	0.019226	0.019699	0.020182	0.020675	0.021178	0.021692	0.022216	0.022834
-1.9	0.023295	0.023852	0.024419	0.024998	0.025588	0.026190	0.026803	0.027429	0.028067	0.028705
-1.8	0.029379	0.030054	0.030742	0.031443	0.032157	0.032884	0.033625	0.034380	0.035148	0.036000
-1.7	0.036727	0.037538	0.038364	0.039204	0.040059	0.040930	0.041815	0.042716	0.043633	0.044557
-1.6	0.045514	0.046479	0.047460	0.048457	0.049471	0.050503	0.051551	0.052616	0.053699	0.054787
-1.5	0.055917	0.057053	0.058208	0.059380	0.060571	0.061780	0.063008	0.064255	0.065522	0.066800
-1.4	0.068112	0.069437	0.070781	0.072145	0.073529	0.074934	0.076359	0.077804	0.079270	0.080737
-1.3	0.082264	0.083793	0.085343	0.086915	0.088508	0.090123	0.091759	0.093418	0.095098	0.096777
-1.2	0.098525	0.100273	0.102042	0.103835	0.105650	0.107488	0.109349	0.111232	0.113139	0.115046
-1.1	0.117023	0.119000	0.121000	0.123024	0.125072	0.127143	0.129238	0.131357	0.133500	0.135600
-1	0.137857	0.140071	0.142310	0.144572	0.146859	0.149170	0.151505	0.153864	0.156248	0.158600

z	-0.09	-0.08	-0.07	-0.06	-0.05	-0.04	-0.03	-0.02	-0.01
-0.9	0.161087	0.163543	0.166023	0.168528	0.171056	0.173609	0.176186	0.178786	0.181385
-0.8	0.186733	0.189430	0.192150	0.194895	0.197663	0.200454	0.203269	0.206108	0.208946
-0.7	0.214764	0.217695	0.220650	0.223627	0.226627	0.229650	0.232695	0.235762	0.238848
-0.6	0.245097	0.248252	0.251429	0.254627	0.257846	0.261086	0.264347	0.267629	0.270920
-0.5	0.277595	0.280957	0.284339	0.287740	0.291160	0.294599	0.298056	0.301532	0.305098
-0.4	0.312067	0.315614	0.319178	0.322758	0.326355	0.329969	0.333598	0.337243	0.340958
-0.3	0.348268	0.351973	0.355691	0.359424	0.363169	0.366928	0.370700	0.374484	0.378318
-0.2	0.385908	0.389739	0.393580	0.397432	0.401294	0.405165	0.409046	0.412936	0.416898
-0.1	0.424655	0.428576	0.432505	0.436441	0.440382	0.444330	0.448283	0.452242	0.456198
0	0.464144	0.468119	0.472097	0.476078	0.480061	0.484047	0.488034	0.492022	0.496018

**Table 80:** Cumulative Standard Normal Distribution  $z = 0.0$  to  $3.99$ 

z	0	0.01	0.02	0.03	0.04	0.05	0.06	0.07	0.08
0	0.500000	0.503989	0.507978	0.511966	0.515953	0.519939	0.523922	0.527903	0.531882
0.1	0.539828	0.543795	0.547758	0.551717	0.555670	0.559618	0.563559	0.567495	0.571428
0.2	0.579260	0.583166	0.587064	0.590954	0.594835	0.598706	0.602568	0.606420	0.610262
0.3	0.617911	0.621720	0.625516	0.629300	0.633072	0.636831	0.640576	0.644309	0.648024
0.4	0.655422	0.659097	0.662757	0.666402	0.670031	0.673645	0.677242	0.680822	0.684386
0.5	0.691462	0.694974	0.698468	0.701944	0.705401	0.708840	0.712260	0.715661	0.719044
0.6	0.725747	0.729069	0.732371	0.735653	0.738914	0.742154	0.745373	0.748571	0.751744
0.7	0.758036	0.761148	0.764238	0.767305	0.770350	0.773373	0.776373	0.779350	0.782304
0.8	0.788145	0.791030	0.793892	0.796731	0.799546	0.802337	0.805105	0.807850	0.810574
0.9	0.815940	0.818589	0.821214	0.823814	0.826391	0.828944	0.831472	0.833977	0.836454
1	0.841345	0.843752	0.846136	0.848495	0.850830	0.853141	0.855428	0.857690	0.859924
1.1	0.864334	0.866500	0.868643	0.870762	0.872857	0.874928	0.876976	0.879000	0.881000
1.2	0.884930	0.886861	0.888768	0.890651	0.892512	0.894350	0.896165	0.897958	0.899774
1.3	0.903200	0.904902	0.906582	0.908241	0.909877	0.911492	0.913085	0.914657	0.916200
1.4	0.919243	0.920730	0.922196	0.923641	0.925066	0.926471	0.927855	0.929219	0.930564
1.5	0.933193	0.934478	0.935745	0.936992	0.938220	0.939429	0.940620	0.941792	0.942944
1.6	0.945201	0.946301	0.947384	0.948449	0.949497	0.950529	0.951543	0.952540	0.953524
1.7	0.955435	0.956367	0.957284	0.958185	0.959070	0.959941	0.960796	0.961636	0.962464
1.8	0.964070	0.964852	0.965620	0.966375	0.967116	0.967843	0.968557	0.969258	0.969944
1.9	0.971283	0.971933	0.972571	0.973197	0.973810	0.974412	0.975002	0.975581	0.976144
2	0.977250	0.977784	0.978308	0.978822	0.979325	0.979818	0.980301	0.980774	0.981234
2.1	0.982136	0.982571	0.982997	0.983414	0.983823	0.984222	0.984614	0.984997	0.985374
2.2	0.986097	0.986447	0.986791	0.987126	0.987455	0.987776	0.988089	0.988396	0.988694
2.3	0.989276	0.989556	0.989830	0.990097	0.990358	0.990613	0.990863	0.991106	0.991344
2.4	0.991802	0.992024	0.992240	0.992451	0.992656	0.992857	0.993053	0.993244	0.993434
2.5	0.993790	0.993963	0.994132	0.994297	0.994457	0.994614	0.994766	0.994915	0.995064
2.6	0.995339	0.995473	0.995604	0.995731	0.995855	0.995975	0.996093	0.996207	0.996314
2.7	0.996533	0.996636	0.996736	0.996833	0.996928	0.997020	0.997110	0.997197	0.997284
2.8	0.997445	0.997523	0.997599	0.997673	0.997744	0.997814	0.997882	0.997948	0.998014

z	0	0.01	0.02	0.03	0.04	0.05	0.06	0.07	0.08	0.09
2.9	0.998134	0.998193	0.998250	0.998305	0.998359	0.998411	0.998462	0.998511	0.998559	0.998605
3	0.998650	0.998694	0.998736	0.998777	0.998817	0.998856	0.998893	0.998930	0.998965	0.998999
3.1	0.999032	0.999065	0.999096	0.999126	0.999155	0.999184	0.999211	0.999238	0.999264	0.999289
3.2	0.999313	0.999336	0.999359	0.999381	0.999402	0.999423	0.999443	0.999462	0.999481	0.999499
3.3	0.999517	0.999534	0.999550	0.999566	0.999581	0.999596	0.999610	0.999624	0.999638	0.999651
3.4	0.999663	0.999675	0.999687	0.999698	0.999709	0.999720	0.999730	0.999740	0.999749	0.999758
3.5	0.999767	0.999776	0.999784	0.999792	0.999800	0.999807	0.999815	0.999822	0.999828	0.999835
3.6	0.999841	0.999847	0.999853	0.999858	0.999864	0.999869	0.999874	0.999879	0.999883	0.999888
3.7	0.999892	0.999896	0.999900	0.999904	0.999908	0.999912	0.999915	0.999918	0.999922	0.999925
3.8	0.999928	0.999931	0.999933	0.999936	0.999938	0.999941	0.999943	0.999946	0.999948	0.999950
3.9	0.999952	0.999954	0.999956	0.999958	0.999959	0.999961	0.999963	0.999964	0.999966	0.999967

**Table 81:** Percentage Points  $t_{\nu,\alpha}$  of the Student-t Distribution

$\nu \setminus \alpha$	0.1	0.05	0.025	0.01	0.005	0.0025	0.001	0.0005
1	3.078	6.314	12.706	31.821	63.657	127.321	318.309	636.619
2	1.886	2.920	4.303	6.965	9.925	14.089	22.327	31.599
3	1.638	2.353	3.182	4.541	5.841	7.453	10.215	12.924
4	1.533	2.132	2.776	3.747	4.604	5.598	7.173	8.610
5	1.476	2.015	2.571	3.365	4.032	4.773	5.893	6.869
6	1.440	1.943	2.447	3.143	3.707	4.317	5.208	5.959
7	1.415	1.895	2.365	2.998	3.499	4.029	4.785	5.408
8	1.397	1.860	2.306	2.896	3.355	3.833	4.501	5.041
9	1.383	1.833	2.262	2.821	3.250	3.690	4.297	4.781
10	1.372	1.812	2.228	2.764	3.169	3.581	4.144	4.587
11	1.363	1.796	2.201	2.718	3.106	3.497	4.025	4.437
12	1.356	1.782	2.179	2.681	3.055	3.428	3.930	4.318
13	1.350	1.771	2.160	2.650	3.012	3.372	3.852	4.221
14	1.345	1.761	2.145	2.624	2.977	3.326	3.787	4.140
15	1.341	1.753	2.131	2.602	2.947	3.286	3.733	4.073
16	1.337	1.746	2.120	2.583	2.921	3.252	3.686	4.015
17	1.333	1.740	2.110	2.567	2.898	3.222	3.646	3.965
18	1.330	1.734	2.101	2.552	2.878	3.197	3.610	3.922
19	1.328	1.729	2.093	2.539	2.861	3.174	3.579	3.883
20	1.325	1.725	2.086	2.528	2.845	3.153	3.552	3.850
21	1.323	1.721	2.080	2.518	2.831	3.135	3.527	3.819
22	1.321	1.717	2.074	2.508	2.819	3.119	3.505	3.792
23	1.319	1.714	2.069	2.500	2.807	3.104	3.485	3.768
24	1.318	1.711	2.064	2.492	2.797	3.091	3.467	3.745
25	1.316	1.708	2.060	2.485	2.787	3.078	3.450	3.725
26	1.315	1.706	2.056	2.479	2.779	3.067	3.435	3.707
27	1.314	1.703	2.052	2.473	2.771	3.057	3.421	3.690
28	1.313	1.701	2.048	2.467	2.763	3.047	3.408	3.674

$\nu \setminus \alpha$	0.1	0.05	0.025	0.01	0.005	0.0025	0.001	0.0005
29	1.311	1.699	2.045	2.462	2.756	3.038	3.396	3.659
30	1.310	1.697	2.042	2.457	2.750	3.030	3.385	3.646
31	1.309	1.696	2.040	2.453	2.744	3.022	3.375	3.633
32	1.309	1.694	2.037	2.449	2.738	3.015	3.365	3.622
33	1.308	1.692	2.035	2.445	2.733	3.008	3.356	3.611
34	1.307	1.691	2.032	2.441	2.728	3.002	3.348	3.601
35	1.306	1.690	2.030	2.438	2.724	2.996	3.340	3.591
40	1.303	1.684	2.021	2.423	2.704	2.971	3.307	3.551
45	1.301	1.679	2.014	2.412	2.690	2.952	3.281	3.520
50	1.299	1.676	2.009	2.403	2.678	2.937	3.261	3.496
$\infty$	1.282	1.645	1.960	2.326	2.576	2.807	3.090	3.291

**Table 82:** Percentage Points  $\chi^2_{\nu, \alpha}$  of the Chi-Square Distribution ( $\nu$ ) degrees of freedom

$\nu \setminus \alpha$	0.1	0.05	0.025	0.01	0.005	0.0025	0.001	0.0005
1	2.706	3.841	5.024	6.635	7.879	9.141	10.828	12.116
2	4.605	5.991	7.378	9.210	10.597	11.983	13.816	15.202
3	6.251	7.815	9.348	11.345	12.838	14.320	16.266	17.730
4	7.779	9.488	11.143	13.277	14.860	16.424	18.467	19.997
5	9.236	11.070	12.833	15.086	16.750	18.386	20.515	22.105
6	10.645	12.592	14.449	16.812	18.548	20.249	22.458	24.103
7	12.017	14.067	16.013	18.475	20.278	22.040	24.322	26.018
8	13.362	15.507	17.535	20.090	21.955	23.774	26.124	27.868
9	14.684	16.919	19.023	21.666	23.589	25.462	27.877	29.666
10	15.987	18.307	20.483	23.209	25.188	27.112	29.588	31.420
11	17.275	19.675	21.920	24.725	26.757	28.729	31.264	33.137
12	18.549	21.026	23.337	26.217	28.300	30.318	32.909	34.821
13	19.812	22.362	24.736	27.688	29.819	31.883	34.528	36.478
14	21.064	23.685	26.119	29.141	31.319	33.426	36.123	38.109
15	22.307	24.996	27.488	30.578	32.801	34.950	37.697	39.719
16	23.542	26.296	28.845	32.000	34.267	36.456	39.252	41.308
17	24.769	27.587	30.191	33.409	35.718	37.946	40.790	42.879
18	25.989	28.869	31.526	34.805	37.156	39.422	42.312	44.434
19	27.204	30.144	32.852	36.191	38.582	40.885	43.820	45.973
20	28.412	31.410	34.170	37.566	39.997	42.336	45.315	47.498
21	29.615	32.671	35.479	38.932	41.401	43.775	46.797	49.011
22	30.813	33.924	36.781	40.289	42.796	45.204	48.268	50.511
23	32.007	35.172	38.076	41.638	44.181	46.623	49.728	52.000
24	33.196	36.415	39.364	42.980	45.559	48.034	51.179	53.479
25	34.382	37.652	40.646	44.314	46.928	49.435	52.620	54.947
26	35.563	38.885	41.923	45.642	48.290	50.829	54.052	56.407
27	36.741	40.113	43.195	46.963	49.645	52.215	55.476	57.858
28	37.916	41.337	44.461	48.278	50.993	53.594	56.892	59.300

$\nu \setminus \alpha$	0.1	0.05	0.025	0.01	0.005	0.0025	0.001	0.0005
29	39.087	42.557	45.722	49.588	52.336	54.967	58.301	60.735
30	40.256	43.773	46.979	50.892	53.672	56.332	59.703	62.162
31	41.422	44.985	48.232	52.191	55.003	57.692	61.098	63.582
32	42.585	46.194	49.480	53.486	56.328	59.046	62.487	64.995
33	43.745	47.400	50.725	54.776	57.648	60.395	63.870	66.403
34	44.903	48.602	51.966	56.061	58.964	61.738	65.247	67.803
35	46.059	49.802	53.203	57.342	60.275	63.076	66.619	69.199
40	51.805	55.758	59.342	63.691	66.766	69.699	73.402	76.095
50	63.167	67.505	71.420	76.154	79.490	82.664	86.661	89.561
60	74.397	79.082	83.298	88.379	91.952	95.344	99.607	102.695
70	85.527	90.531	95.023	100.425	104.215	107.808	112.317	115.578
80	96.578	101.879	106.629	112.329	116.321	120.102	124.839	128.261
90	107.565	113.145	118.136	124.116	128.299	132.256	137.208	140.782
100	118.498	124.342	129.561	135.807	140.169	144.293	149.449	153.167

**Table 83:** Kolmogorov-Smirnov Test Critical Values

n	$D_{0.1}$	$D_{0.05}$	$D_{0.01}$
10	0.36866	0.40925	0.48893
11	0.35242	0.39122	0.46770
12	0.33815	0.37543	0.44905
13	0.32549	0.36143	0.43247
14	0.31417	0.34890	0.41762
15	0.30397	0.33760	0.40420
16	0.29472	0.32733	0.39201
17	0.28627	0.31796	0.38086
18	0.27851	0.30936	0.37062
19	0.27136	0.30143	0.36117
20	0.26473	0.29408	0.35241
25	0.23768	0.26404	0.31657
30	0.21756	0.24170	0.28987
35	0.20185	0.22425	0.26897
over 35	$1.22/\sqrt{n}$	$1.36/\sqrt{n}$	$1.63/\sqrt{n}$

See (Miller, 1956). Additional values can be computed at:

<http://www.ciphersbyritter.com/JAVASCRP/NORMCHIK.HTM#KolSmir>



# Bibliography

- Ahrens, J. and Dieter, V. (1972). Computer methods for sampling from the exponential and normal distributions. *Communications of the Association for Computing Machinery*, 15:873–82.
- Alexopoulos, C. and Seila, A. F. (1998). Output data analysis. In Banks, J., editor, *Handbook of Simulation*. John Wiley & Sons, New York.
- Balci, O. (1997). Principles of simulation model validation, verification, and testing. *Transactions of the Society for Computer Simulation International*.
- Balci, O. (1998). Verification, validation, and testing. In *The Handbook of Simulation*, pages 335–393. John Wiley & Sons.
- Banks, J., Carson, J., Nelson, B., and Nicol, D. (2005). *Discrete-Event System Simulation*. Prentice Hall, 4th edition.
- Blanchard, B. S. and Fabrycky, W. J. (1990). *Systems Engineering and Analysis*. Prentice-Hall, Englewood Cliffs, New Jersey.
- Box, G. E. P., Jenkins, G. M., and Reinsel, G. C. T. S. A. (1994). *Forecasting and Control*. Prentice Hall, 3rd edition.
- Casella, G. and Berger, R. (1990). *Statistical Inference*. Wadsworth & Brooks/Cole.
- Cash, C. R., Dippold, D. G., Long, J. M., Nelson, B. L., and Pollard, W. P. (1992). Evaluation of tests for initial conditions bias. In Swain, J. J., Goldsman, D., Crain, R. C., and Wilson, J. R., editors, *Proceedings of the 1992 Winter Simulation Conference*, pages 577–585.
- Cheng, R. C. (1977). The generation of gamma variables with nonintegral shape parameters. *Applied Statistics*, 26(1):71–75.
- Command, A. F. S. (1991). *ASD Directorate of Systems Engineering and DSMC Technical Management Department*.
- Devroye, L. (1986). *Non-Uniform Random Variate Generation*. Springer-Verlag, New York.
- Fishman, G. S. (2001). *Discrete-Event Simulation: Modeling, Programming, and Analysis*. Springer, New York.

- Fishman, G. S. (2006). *A First Course in Monte Carlo*. Thomson Brooks/Cole.
- Fishman, G. S. and Yarberry, L. S. (1997). An implementation of the batch means method. *INFORMS Journal on Computing*, 9.
- Gross, D. and Harris, C. M. (1998). *Fundamentals of Queueing Theory*. John Wiley & Sons, New York, 3rd edition.
- Hull, T. E. and Dobell, A. R. (1962). Random number generators. *SIAM Review*, 4:230–254.
- Kelton, W. D., Sadowski, R. P., and Sturrock, D. T. (2004). *Simulation with Arena*. McGraw-Hill, 3rd edition.
- Lada, E. K., Wilson, J. R., and Steiger, N. M. (2003). *A Wavelet-Based Spectral Method for Steady-State Simulation Analysis*. Proceedings of the 2003 Winter Simulation Conference.
- Law, A. (2007). *Simulation Modeling and Analysis*. McGraw-Hill, 4th edition.
- L'Ecuyer, P., Simard, R., and Kelton, W. D. (2002). An object-oriented random number package with many long streams and substreams. *Operations Research*, 50:1073–1075.
- Leemis, L. (1991). Nonparametric estimation of the cumulative intensity function for a nonhomogeneous poisson process. *Management Science*, 37(7):886–900.
- Leemis, L. M. and Park, S. K. (2006). *Discrete-Event Simulation: A First Course*. Prentice-Hall.
- Litton, J. R. and Harmonosky, C. H. (2002). *A Comparison of Selective Initialization Bias Elimination Methods*. Proceeding of the 2002 Winter Simulation Conference.
- Miller, L. H. (1956). Table of percentage points of kolmogorov statistics. *Journal of the American Statistical Association*, pages 111–121.
- Montgomery, D. C. and Runger, G. C. (2006). *Applied Statistics and Probability for Engineers*. John Wiley & Sons, 4th edition.
- Ravindran, A., Phillips, D., and Solberg, J. (1987). *Operations Research Principles and Practice*. John Wiley & Sons, 2nd edition.
- Ripley, B. D. (1987). *Stochastic Simulation*. John Wiley & Sons Inc.
- Robinson, S. (2005). Automated analysis of simulation output data. *Proceedings of the 2005 Winter Simulation Conference*, 763-770.
- Ross, S. (1997). *Introduction to Probability Models*. Academic Press, 6th edition.
- Rossetti, M. D. (2008). JSL: An open-source object-oriented framework for discrete-event simulation in Java. *International Journal of Simulation and Process Modeling*, 4(1):69–87.
- Rossetti, M. D. (2015). *Simulation Modeling and Arena*. John Wiley & Sons, New York, New York.

- Rossetti, M. D. and Delaney, P. J. (1995). *Control of initialization bias in queueing simulations using queueing approximations*. Institute of Electrical and Electronics Engineers, Piscataway, New Jersey.
- Schmeiser, B. W. (1982). Batch size effects in the analysis of simulation output. *Operations Research*, 30:556–568.
- Soto, J. (1999). Statistical testing of random numbers. In *Proceedings of the 22nd National Information Systems Security Conference*.
- Steiger, N. M. and Wilson, J. R. (2002). An improved batch means procedure for simulation output analysis. *Management Science*, 48.
- Welch, P. D. (1983). A graphical approach to the initial transient problem in steady state simulation. In *10th IMACS World Congress on System Simulation and Scientific Computation*, pages 219–221.
- White, K. P., Cobb, M. J., and Spratt, S. C. (2000). *A Comparison of Five Steady-State Truncation Heuristics for Simulation*. Proceeding of the 2000 Winter Simulation Conference.
- Whitt, W. (1989). Planning queueing simulations. *Management Science*, 35(11):1341–1366.
- Wilson, J. R. and Pritsker, A. A. B. (1978). A survey of research on the simulation startup problem. *Simulation*.