ROLLBACK-ABLE RANDOM NUMBER GENERATORS FOR THE SYNCHRONOUS PARALLEL ENVIRONMENT FOR EMULATION AND DISCRETE-EVENT SIMULATION (SPEEDES)

By

R. KARTHIK NARAYANAN
B.E. P.C.E.A, Nagpur University, 2000

A thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science
in the Department of Modeling and Simulation
in the College of Engineering and Computer Science
at the University of Central Florida
Orlando, Florida

Fall Term
2004

ABSTRACT

Random Numbers form the heart and soul of a discrete-event simulation system. There are few situations where the actions of the entities in the process being simulated can be completely predicted in advance. The real world processes are more probabilistic than deterministic. Hence, such chances are represented in the system by using various statistical models, like random number generators. These random number generators can be used to represent a various number of factors, such as length of the queue. However, simulations have grown in size and are sometimes required to run on multiple machines, which share the various methods or events in the simulation among themselves. These Machines can be distributed across a LAN or even the internet. In such cases, to keep the validity of the simulation model, we need rollback-able random number generators. This thesis is an effort to develop such rollback able random number generators for the Synchronous Parallel Environment for Emulation and Discrete-Event Simulation (SPEEDES) environment developed by NASA. These rollback-able random number generators will also add several statistical distribution models to the already rich SPEEDES library.

ACKNOWLEDGEMENTS

I offer my most sincere and heartfelt thanks to my advisor Dr.Luis Rabelo, Dr. Jose. A Sepulveda and Dr. J. Peter Kincaid who have inspired, mentored, motivated and supported me through various educational, and personal triumphs and defeats. I am very thankful to Dr.Luis Rabelo for his careful reading of my thesis, helpful comments and inculcating the value of focused research in me. I take this opportunity to thank Dr J. Peter Kincaid for his sincere devotion towards his student's welfare. He is an example for other teachers and professors. I also thank Dr. Luis Rabelo, Dr. Jose A. Sepulveda and Dr. J. Peter Kincaid for agreeing to be on my thesis committee.

I dedicate this thesis to my parents and sandesha.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1: INTRODUCTION

In this chapter, the concepts of random numbers, distributions and parallel/distributed simulation will be explained. This chapter will provide the basic introduction to the research work done on Rollback-able random number generator.

1.1 Random Numbers

Random numbers form the most important components of any stochastic simulation model. The efficiency of the simulation is a direct function of the efficiency of its random generators. In a stochastic simulation model the random numbers are used to represent the occurrence of real world events which are naturally random or are simply not of a deterministic nature. Different sets of random numbers can be generated for representing different systems, each having its own set of characteristics. Random numbers can be generated by using various stochastic models for random number generations called the random distributions. These distributions plot a particular random number against the probability of its occurrence. The simulations use these random distributions to represent a particular event in the system. The modern simulations have grown larger and require large computer systems to handle them. A single simulation may run for several weeks on a normal desktop system. Thus, the concept of parallel/ distributed simulation was developed, which makes the simulation faster by distributing the task across various computers connected to a network

Synchronous Parallel Environment for Emulation and Discrete-Event Simulation (SPEEDES) is a general purpose discrete event distributed simulation environment designed to simulate a wide variety of process simulations. SPEEDES has been used with great success by the DoD (Department of Defense) and NASA for simulation of the shuttle model and other airspace simulations [10]. SPEEDES has support for different architectures ranging from a fast stand alone computer, to a large network of workstations (having varying speeds). In a nutshell, the SPEEDES simulation engine allows the simulation builder to perform parallel/distributed simulations. Parallel/Distributed simulation refers to the execution of a discrete event simulation program on a machine containing multiple CPU's. This includes execution on closely coupled multiprocessor machines (parallel simulation) as well as geographically distributed computers (distributed simulation). Simulations have increased tremendously in complexity and size and therefore require more CPU time. Thus, there is a need for powerful computers that can perform such a huge simulation in a very short time. Parallel and distributed simulations are of practical importance because it is more cost-effective to have parallel or distributed computers than it is to one super computer. The internet, spanning across millions of computers worldwide, is the fastest possible super computer. Parallel/Distributed simulations utilize this so called super computer to run large simulation experiments in times shorter than a life-span of a common house fly. SPEEDES supports the implementation of the High Level Architecture (HLA) federation of simulations.

A discrete-event simulation computes a sample path through the set of possible system states for a period of simulated time. A parallel/distributed simulation attempts to construct this sample path by utilizing many processors. There are two approaches for achieving this: The Space-Parallel approach, in this the simulation that is being run is partitioned in smaller entities called as logical processes (LP) and each LP is given a process id. These processes execute concurrently on the different processors. Each LP communicates with the others by using time stamped events or messages. Each LP computes the portion of the sample path pertaining to its subsystem. These LPs, after processing their part of the simulation code, either generate new events for themselves or schedule an event for a different LP running on a different processor. A key factor on the success of such a schema is of synchronization: how we make sure that the events are being processed in the correct order. There are two types of synchronization algorithms. Conservative algorithms make sure that dependent events are never processed out of order so that synchronization error never occurs. Optimistic algorithms allow synchronization errors to occur but provide a rollback/commit mechanism to recover from this error. Some applications that utilize parallel/distributed simulations are queuing, network simulation and simulation of digital logic circuits. These simulations represent real time processes with events that are more probabilistic than deterministic. Thus, the efficiency of the parallel/distributed simulation depends not only on the synchronization, but also on the random number generator used which determines how close it is to the real process. Random number generators determine the repeatability of a simulation model. When a repeatable simulation model is run with the same initial states and random number seeds, it will produce identical results both times. Random number generators are used to decide when an event occurs; different random numbers are used to generate various events. In parallel/distributed simulation, random number generators can be used for forecasting. They can be used to provide computational data for an event that is being processed out of order, or to find the probability of an event causing a synchronization error. Random number generators are the heart and soul of the parallel/distributed simulation, and the efficiency of the simulation is a direct function of the random number generators efficiency. SPEEDES can be used to developed complex, high end simulations. It is the capability of SPEEDES to maintain a short run time for complex simulations, which makes it a very powerful simulation tool.

1.2 Problem Statement

As explained above, the very efficiency of a simulation depends on the efficiency of the random number generators used to represent its events. An inefficient random number generator or a random number generator with no rollback ability can cause fatal synchronization errors. The results generated by this simulation would be erroneous and of no value. Thus, the problem addressed by this thesis project is a very important one for both the efficiency and the speed of the simulations. Unfortunately, SPEEDES random number generator library does not have classes for random number generators which are commonly used in operations. Therefore, the thesis also addresses this issue: development of these common random number generators, for the SPEEDES library. SPEEDES was developed using C++, therefore C++ is the chosen language for developing these random number generators.

1.3 Synopsis of the Thesis

This thesis has been divided into six chapters. The first chapter is a brief introduction to the problems and issues addressed by this thesis. In the second chapter, we will be introduced to the SPEEDES simulation environment. The third chapter discusses the rollback ability of SPEEDES. The fourth chapter explains the implementation and development of the random number generators. The fifth chapter validates the outputs of the random number generators. The concepts presented in this thesis are then ended in the sixth chapter with a conclusion of the thesis and the observations. It also gives and overview on the avenues for further research, on the issues addressed by this thesis topic.

CHAPTER 2: INTRODUCTION TO SPEEDES

In this chapter we will further introduce the advanced concepts, tools and functionality of the SPEEDES C++ library. SPEEDES is a Linux based GNU C++ compliant library with built-in templates and classes for developing parallel/distributed simulation. SPEEDES is currently being used by NASA for its Virtual Test Bed project.

2.1 SPEEDES

SPEEDES (Synchronous Parallel Environment for Emulation and Discrete-Event Simulation) is an object-oriented simulation environment that was developed at the Jet Propulsion Laboratory (Steinman 1991, 1992). Designed for distributed simulation, SPEEDES supports multiple synchronization strategies (including Time Warp, Breathing Time Buckets, and Breathing Time Warp [2][17]) that can be selected by the user at runtime. In addition, SPEEDES provides a sequential simulation mode (with most of the parallel overhead removed) so that a particular simulation model can be executed serially or in parallel, depending on a runtime flag. Developed using C++, SPEEDES uses an object oriented computational model. SPEEDES has the capability to run parallel/distributed discrete-event simulations. The various simulation objects are distributed by SPEEDES to different processors. Each processor is then responsible for executing all the events that are generated by the simulation object. One of the processors is called the SPEEDESSERVER and is responsible for the synchronizations; as each event may affect objects on other processors, it is necessary to form a gateway through which the processors can send event occurrence or completion notice to the other processors in the simulation. Though the distribution of the simulation objects reduces the time taken by the simulation to complete execution, it tremendously increases the complexity of the simulation, and the efficiency or correctness of the simulation is now is a direct function of the ability of the SPEEDESSERVER to maintain synchronization of various events. SPEEDES uses several synchronization algorithms for running complicated distributed simulations which include conservative algorithm and optimistic algorithm. Why are these schemes needed? When running such complicated simulations, each processor has no information about the status of the other processors in the system. Hence, it is a highly probable that one processor may execute its events faster than the other processor was able to; this lag could cause the simulation system to be in the wrong state; the faster processor might miss out on some valid data and trigger of a erroneous simulation results. To avoid such a happening, one of the schemes is chosen. The conservational scheme guarantees that no event will be erroneously processed. Programmatically, an LP cannot process an event with timestamp t until it can be sure that there is no event with timestamp smaller than t that may arrive later. In case that such an event exists the event with timestamp t is stalled. The Time Warp mechanism invented by Jefferson and Sowizral [Jefferson and Sowizral, 1982, Jefferson, 1985] is the most well known optimistic algorithm. When an LP receives an event with timestamp t, it assumes that it is safe to process this event, and it goes ahead and processes the event. If the LP then receives an event with timestamp smaller than t or any other event it has already processed, it rolls back to the event it is processing and processes the event in the order of their timestamp. Rolling back an event involves restoring the state of the LP to that which existed prior to processing the event and un-sending messages sent by the roll backed events. Once it has been made sure that the simulation is running synchronously at a point, SPEEDESSERVER commits the simulation at that state. The most important thing here is that once the rollback has occurred, it is necessary that the simulation follow the same path as it had before the rollback was issued. That means the same events will be triggered, the same data will be used, and if any random numbers were generated, the same random numbers will be generated during the retrace. Hence, in distributed simulation, it is very important to have a method for generating rollback able random numbers. Before we proceed further with the rollback feature of SPEEDES, we will give you a brief introduction the SPEEDES API. The major entities of a SPEEDES framework simulation are listed below. A brief overview of each entity will also be provided, taking in mind their relevant importance to the thesis topic.

2.2. Simulation Objects

2.3. Simulation Object State

2.4. Events and Event Handlers

2.5. The Process Model

2.6. Rollback Utilities

2.2 Simulation Objects

Simulation objects form the basic building block of the SPEEDES simulation framework. SPEEDES framework is based on the C++ programming language. The various classes and API (Application Programming Interface) that form the SPEEDES framework are programmed using the C++ programming language. Consequently SPEEDES also inherits the object oriented features of C++. The various entities in the simulation are seen as objects. These objects have properties and methods; they respond to various events and use their methods to process them. The methods either generate simulation data or trigger further events on other objects. A simulation object could be any physical object; it can be as large as a ship or as small as an electron. In a war game simulation, the enemy tanks are represented as objects; these have some properties, such as color, height, weight, location on map, name, capacity etc; and methods such as fire, dodge, move front, move back, turn right, et cetera. Like in real world these objects respond to events, such as ‘receiving fire’ i.e. if the tank is fired upon by an aircraft or the players tank. The occurrence of this event triggers a tanks method, perhaps in this case move back. It may even trigger a series of methods such as move back and fire. They can also generate events for other simulation objects, e.g. ‘a tanker hit by a missile, giving out rescue SOS to its friends tanks in the area’. So, a SPEEDES object is an entity that has some properties and responds to events, by triggering its corresponding method or by triggering new events. The following figure should help you get a better overview.

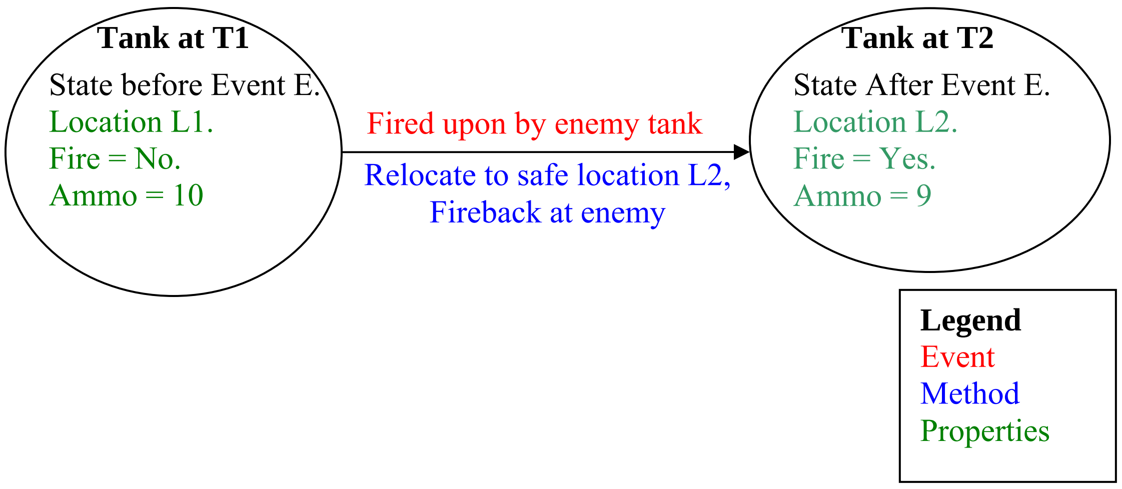


Figure 2.1 Explanation of the simulations objects properties, methods and events.

2.3 Simulation Object States

State refers to the values of the properties of a simulation object at a particular instance in time. In the above example, T1 and T2 represent the states of the tank at time T1 when it was fired upon and at the time T2 the time at which it fired back at the enemy. It can be easily noted from Figure 2.1 that the various properties of the tank, such as its location and ammo count have changed. Events cause the simulation objects to trigger their corresponding methods which in turn change their properties. The SPEEDESSERVER keeps track of the state of a simulation object, at each instance of time in the simulation. A case may occur that in a simulation that has been distributed over several processors where a slow processor may schedule an event, called a straggler event, on a second processor, which is ahead in time (future) with respect to the first processor. To process this event, scheduled in the faster processors past, the faster processor must rollback the state of its simulation object to a point in time before the event occurrence. Thus, simulation object state helps the SPEEDESSERVER to keep in sync the various simulation events and their corresponding affect on the simulation objects.

2.4 Events

Events are the entities that along with simulation objects form the core of the simulation. They form a communication link between the SPEEDES object and the simulation environment. A simulation environment represents a real world process which is dynamic and is constantly changing. The changes in the simulation environment are represented in the simulation as events. An optimist would say these events cause the simulation environment to change. E.g. in a queue an arrival of a customer into the queue changes the length of the queue. Hence, the simulation environment has now changed, that there is one more customer in the process now. The point that has to be noted here is that the queue itself is mostly represented as a simulation object. Thus, an event not only initiates a change of state of the object but of the whole simulation as well. In objects, there are event handlers which respond to an occurrence of an event. Similarly the simulation has is own event handlers. An event handler is a mechanism provided to dynamically add, subscribe, and remove any type of event. With traditional events which are also called point-to-point events, the caller must have knowledge about the event being called, and the object which is responsible for processing that event. Event handlers make all this unnecessary. Event handlers are of two types: direct and indirect. Direct handlers are handlers for which the scheduler schedules the handler on a specific simulation object. In some cases, the scheduler does not specify the handler. The handler for such types of events is decided during the run time, or based on some trigger string logic. The scheduler in this case has no idea which entity receives the handler event, or which handler events are active. In a queues simulation, when a person enters a queue, an arrival event is triggered, but only that queue to which the customer arrives is affected. Such events are handled by the scheduler as indirect handlers, because they need not know which queue received the event, and what method was triggered. The event will, however change the simulation state, e.g. there will be 1 customer added to the system. Hence, the simulation will implement a direct event handler for the object say ‘customer counter’, which increment the count of the customer by 1, every time a new customer enters a queue. Thus, the event handler enables the simulation to better utilize its resources and hardware.

2.5 The Process Model

An event in a simulation model occurs at a particular instance of time. A process in SPEEDES refers to a reentrant event. A reentrant event can take place over a finite interval of time. In other words, a reentrant event repeats itself after some interval of time. The point in time when this event is triggered is decided by the simulation. The major difference between a process and an event is that the process holds in its memory the values of the local variables of the simulation object. In the case of the simulation of queue, the process that keeps track of the number of customers in a queue is simple process which increments the previous value of the count by 1. The event is reentrant as it is triggered every time a customer enters the queue. Discrete-Event models written mainly using ‘processes’ are called process based simulation models. Processes help in writing codes that are reentrant, that is the program can leave control of a process at any time and exit to a different process. The program can return to this process at any instance in time, and start executing from the last exit point. The process has the ability to remember, during reentry, the state of all its local variables at the point of last exit. Writing a process based simulation is simpler, more intuitive, and easier to maintain than the event-based model.

2.6 Rollback Utilities

One of the most important features that the SPEEDES library provides is the ability to rollback. All attributes of a distributed simulation model must be rollback-able. The same is true of dynamically allocated memory. SPEEDES has built-in functionality to manage dynamic memory management in a rollback fashion to avoid memory leaks, and heap corruption. SPEEDES also has inbuilt rollback able random number generators that can generate random numbers with a uniform distribution. This enables programmers to generate simulation models that are repeatable; e.g. when run twice, with the same inputs, random seeds and initial states, the simulation will produce identical results. SPEEDES has a class which defines various rollback able data types like rollback-able integer, float and other normal C++ data types. The SPEEDES API also has rollback-able version of standard C++ functions like rollback-able asserts, rollback-able memory copy and string duplication. Apart, from this it also enables the programmer to create new rollback-able functions or objects from the pre defined types and objects. Next chapter will give a further detailed explanation of rollback ability of SPEEDES and its implementation. We will be discovering how SPEEDES handles various rollbacks and the rollback able function, objects and data types.

2.7 Summary

In this chapter we discussed the SPEEDES libraries architecture and how it handles the various events and objects in the simulation. We also were introduced to the various features of SPEEDES including the SPEEDES process model, data types and rollback utilities. SPEEDES also has various classes for generating random number with rollback capability. But, there are only limited distributions that are available in SPEEDES.

There are no available classes for generating numbers with the following probability distribution

.

- Lognormal.
- Gamma.
- Exponential.
- Triangular
- Weibull.
- Johnson System of Distribution.
- Discrete Empirical Distribution.
- Continuous Empirical Distribution.

The main objective of this thesis is to develop rollback able random number generators, using the SPEEDES development library for the above mentioned distributions.

CHAPTER 3: ROLLBACK ABILITY IN SPEEDES

In this chapter we introduce and explain the various rollback-able classes, data types and containers available in the SPEEDES development library. These tools at the disposal of a good programmer can generate simulations that are efficient, fast and can be executed in a parallel/distributed environment.

3.1 The Need for Rollback-able Random Number Generators

Simulation objects need to be made rollback-able because SPEEDES is an optimistic framework. Optimistic framework encourages faster distributed simulation. In optimistic framework, a simulation objects assumes that an event occurred successfully and proceeds with its simulation. If an event has failed SPEEDED does a rollback to a state before the occurrence of the event and retraces the simulation with the event data. This means that any change to the state of a simulation object must be recorded so that the state may be restored in case of an event failing. This goal is achieved with the help of the various rollback-able data types and objects that are included with the SPEEDES framework. The SPEEDES framework also allows the user to program new rollback-able data types, if the need arises. The most basic rule for developing rollback-able simulation is that "if the value of variable changes as the result of processing an event, then that variable must be made rollback-able". This rule applies to all the variables, classes, memory pointers and data structures being used in the simulation. The rollback-able used to build the contents (attributes) of a simulation object, record changes in the attributes of the simulation. When a rollback occurs, the attributes of the simulation objects are restored to their respective values at the state it was roll-backed to. SPEEDES also provides rollback able memory management; with so much data being tracked and saved, it is very important redundant or unneeded be disposed off and the memory reclaimed for storing useful data.

3.2 Basic Rollback-able data types

3.2.1. Rollback-able integers and floats.

Built in rollback-able data types RB_int and RB_float are similar in their usage and functionality to the normal C++ int and float variables. The prefix RB suggests that is a rollback-able data type. They are normally used to store variables that need to be rollback-able. RB_int will used the variable stores integer values (ex. 2,4,16 etc) and RB_float is used when the variable stores real numbers (ex. 1.5, 6.25 etc). You can do all the normal arithmetic and logical operations on these variables. The definition for these built in types is given in RB_int.H and RB_float.H respectively. You need to include the definitions file in your code before you can use these data types.

3.2.2. Rollback able-Strings and void pointers.

Classes RB_SpString and RB_VoidPtr provide the capability of having class attributes, which handle strings and pointers in a rollback able fashion. RB_SpString data type is used when u need a string variable whose value changes on an event. RB_VoidPtr can be used to create rollback able pointers for variables whose values change on an event. The definitions for these data types can be found in the file RB_SpString.H and RB_VoidPtr.H.

3.2.3. Rollback-able Boolean

RB_SpBool is a Boolean class that can only take values of either 0 or 1, i.e. either a value SpFalse or SpTrue. This class supports rollback-able types by restoring Boolean states of the various simulation objects. The definition for this class is provided in the definition file RB_SpBool.H.

3.2.4. Rollback-able Streams

Rollback able streams are rollback-able versions of standard C++ cout function. It is necessary to output information to the user of the simulation. Use of COUT may result in incorrect output, therefore, to output data in a simulation that has rollback ability; we must use the RB_cout function. Classes RB_ostream and RB_exostream provide rollback-able stream capability. The definitions for these classes are described in include files RB_ostream.H and RB_exostream.H.

3.3 Rollback-able Container Classes

Rollback-able data types satisfy all the data storage needs for a simple simulation model. But as the model gets more complex, the need for container classes arises. Container classes are special C++ built-in classes that give the user some specific data storage functionalities. They also let them dynamically declare data and allocate memory for them.

3.3.1. Rollback-able binary and hash trees

The RB_SpBinaryTree is a standard binary tree that can be used in either a normal, balanced, or splay modes. One of the major principals of a binary tree is that value or data at the parent is always greater than either of its child. Any binary node has two children, the left and the right child. The right child is always greater than the left child. The RB_SpHashTree supports a similar interface but differs in that the elements cannot be traversed in a sorted fashion. These classes support inserting, sorting and deleting of elements in the tree. The definition for these classes are described in the include file RB_SpBinaryTree.H and RB_SpHashTree.H

3.3.2. Rollback-able lists

The Lists data structure is normally used when items in a data structure do not need to be sorted. They store all data in memory as void pointers. The definitions for these classes are described in the include file RB_SpList.H.

3.3.3. Rollback-able priority trees.

Class RB_SpPriorityTree is a specialization of the binary tree. In a Priority tree, the element can be inserted in any order, but only removed from the front of the list or retracted through a retraction handle. The item at the front of the list is always of the highest priority. The container can be searched for an element that match a certain priority, duplicates are allowed and the elements retain the priority of their insertion order. The definitions of this class are defined in include file RB_SpPriorityTree.H.

3.3.4. Rollback-able dynamic pointer arrays.

RB_SpDynPtrArray behaves like an array of RB_VoidPtr. The data structure is dynamic in the sense that the array size can grow whenever an element beyond the limit of the array is accessed. The definitions for this class are defined in the include file RB_SpDynPtrArray.H.

3.4 Rollback-able Random Number Generator

SPEEDES has an inbuilt class, which provides a library of various statistical random number generators. The definition for this class is given in the include file, SpRandom.H. The classes defined in SpRandom.H are inherited by the include file RB_SpRandom.H, which has the class definition for the rollback able random number generator. RB_SpRandom.H implements the function "GenerateUniform()" which is a rollback able uniform random number generator. The GenerateUniform() function, maps the random seed to the timestamp at which the random number was generated. In case, of a rollback, GeneratedUniform() function uses the same seed and input values used at the time to which the simulation was roll backed. This ensures that there are no synchronization errors, or any erroneous events being generated.

3.5 Summary

This chapter went deeper into the SPEEDES development library investigating and explaining the various data types and classes. With this knowledge we will proceed further and get in to the details of developing random number generators using

SPPEDES. But, before that we will discuss the characteristics and functionalities of the distributions being developed in this project.

CHAPTER 4: EXPLANATION AND IMPLEMENTATION OF THE VARIOUS RANDOM NUMBER GENERATORS

This chapter gives a detailed insight into the various random distributions developed, their statistical and mathematical equations, and the representation of these distributions on a chart. Finally it gives you an overview of the algorithms used to generate these random distributions.

4.1 Random Number Generators Developed

A total of eight random number generators were developed using the SPEEDES simulation library during the course of this project. The random number generators were used in the simulation of the NASA shuttle model. This chapter will explain the various random number generators and their implementation (source code). Random number generators with the following distribution were generated.

- Lognormal.
- Gamma.
- Exponential.
- Triangular
- Weibull.
- Johnson System of Distribution.
- Discrete Empirical Distribution.
- Continuous Empirical Distribution.

The Most important thing about all the above mentioned distributions is that they are either directly derived from uniform distribution; which is the most basic distribution or they are based upon the normal distribution, which is again based on uniform distribution. The fact that SPEEDES had a built in rollback able random number generator, proved very useful in making the above random number generators rollback-able. Before going ahead with the explanation the random distributions developed for the project we will take a small look at the uniform.

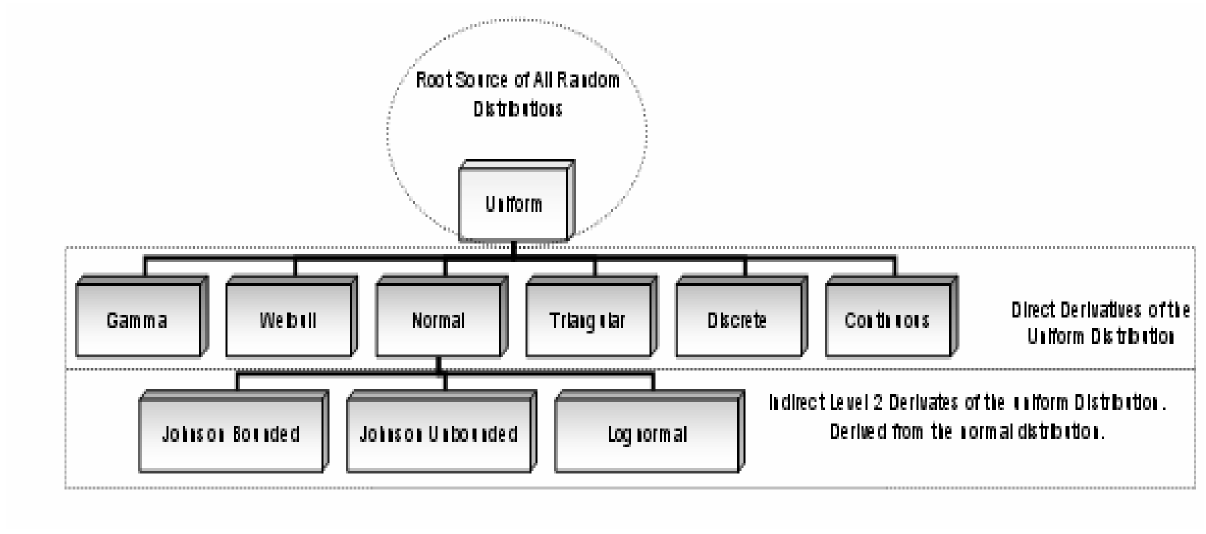

Root Source of All Random Distributions
Uniform
Gamma
Weibull
Normal
Triangular
Discrete
Continuous
Direct Derivatives of the Uniform Distribution
Johnson Bounded
Johnson Unbounded
Lognormal
Indirect Level 2 Derivates of the uniform Distribution. Derived from the normal distribution.

Figure 4.1: Distribution Derived from uniform distribution.

4.2 Uniform Distribution

In a uniform distribution, all numbers within a given range have equal or uniform probability of occurrence hence, the name uniform distribution. The Probability Density Function for uniform distribution is given by the following equation:

$$F(x) = \begin{cases} 0 & \text{for } x < a \\ \frac{x-a}{b-a} & \text{for } a \le x < b \\ 1 & \text{for } x \ge b \end{cases}$$

Figure 4.2: PDF for uniform Distribution.

To generate a uniform random number we choose a random probability R between 0 and 1. This probability is the probability P(x) of a number x being generated. The number X is the random number that has to be returned as an output. To calculate x we apply the inverse transform technique and since 0<R<1, from the PDF for uniform distribution we have:

$$X= a + (b-a)* R$$

The uniform random number developed by the above algorithm will always return a number x which is greater than a, the minimum and less than b, the maximum. In case, of "GenerateUniforn()" SPEEDES library function, the value of a is always 0 and b is always 1. The standard call for the function is "X = GenerateUniform();" The source code for the above distribution is given in table 4.1. The probability distribution curve for the uniform distribution is given in Figure 4.3.

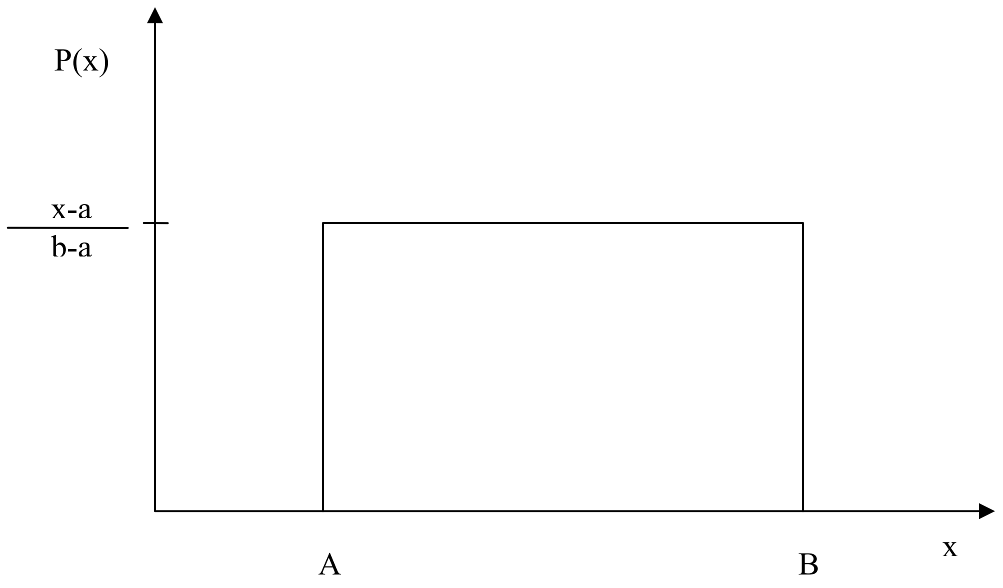


Figure 4.3: Probability Distribution curve for Uniform Distribution [1]

4.3 Gamma Distribution

Gamma distribution is directly derived from the uniform distribution, and is one of the toughest random number generators to program. The PDF for the gamma distribution is given by the equation below:

$$f(x) = \frac{\left(\frac{x-\mu}{\beta}\right)^{\gamma-1} \exp\left(-\frac{x-\mu}{\beta}\right)}{\beta\Gamma(\gamma)} \qquad x \geq \mu; \gamma, \beta > 0$$

Figure 4.4: PDF for Gamma Distribution

β is the scale parameter of the distribution and is an integer. γ is the shape parameter of the distribution and μ is the location parameter. To generate a random variate with a gamma distribution the following steps are exectued. The source code for the gamma distribution function can be found in appendix[b]

1. Compute $a = (2\beta - 1)^{1/2}$, $b = 2\beta - \ln 4 + 1/\alpha$.
2. Generate two uniform variates R_1 and R_2.
3. Compute $X = \beta[R_1 / (1 - R_1)]^{a.}$
4. If $X > b - \ln(R_1^2 R_2)$, reject X and go to 2.
5. if $X <= b - \ln(R_1^2 R_2)$, use X as the variate
6. Replace X by $X/\beta\gamma$

The Source code for the gamma distribution can be found in table 4.1.

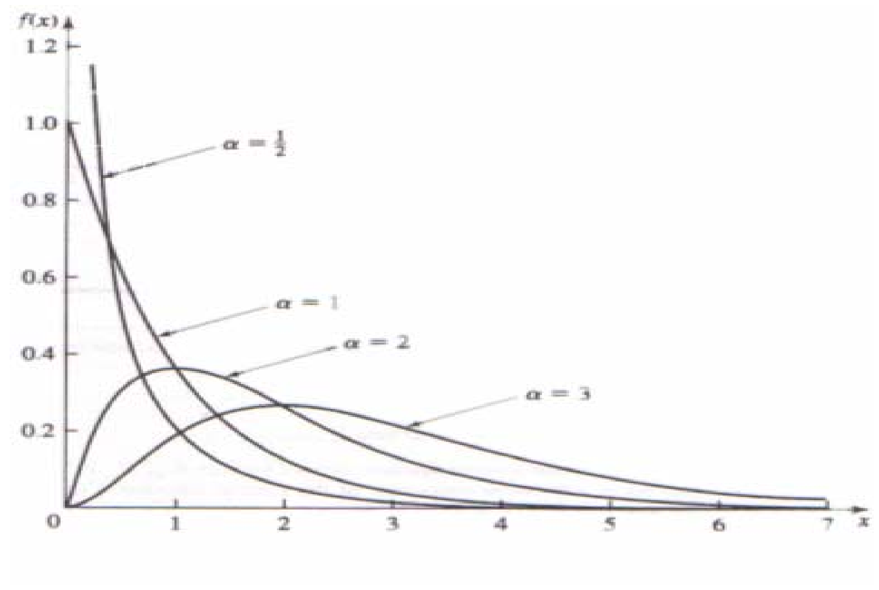


Figure 4.5: Probability Distribution curve for Gamma Distribution.[1]

Table 4.1 Gamma Distribution Source Code

```
#include<iostream.h>
#include<fstream.h>
#include<stdlib.h>
#include<math.h>
#include<time.h>
double gamma(double,double,double);
int main()
{
 double gam,beta,offset,retval;
 int i;
 ofstream ofile("gamma.txt");
 cout<<"Enter the Value for "<<endl<<"gamma :- ";
 cin>>gam;
 cout<<endl<<"Beta :- ";
 cin>>beta;
 cout<<endl<<"offset :- ";
 cin>>offset;
 srand(unsigned(time(NULL)));
 for(i=0;i<50000;i++)
{
    retval=gamma(gam,beta,offset);
      if(isinf(retval)||isnan(retval))
        i--;
      else
    ofile<<retval<<endl;
  }
return 0;
}

double gamma(double gam,double beta,double offset)
{
 double x,a,b,c,d,v,y,z,w,p,theta,randnum1,randnum;
 theta=4.5;
 if(beta > 1)
 {
 a=1/(sqrt(2*beta - 1));
 b=beta - log(4);
 c=beta + 1/a;
 d=1+log(theta);
 randnum=rand()%10001;
 randnum=randnum/10000;
 randnum1=rand()%10001;
 randnum1=randnum1/10000;
 v=a*log(randnum/(1-randnum));
 y=beta*exp(v);
 z=randnum*randnum*randnum1;
```

(contd)

```
w=b+c*v-y;
 if(isinf(y)||isnan(y)) {
   x=gamma(gam,beta,offset);
        return x;
 }
 else {
  if((w+d-theta*z)>=0)
        return y;
  else if(w>=log(z))
        return y;
  else  {
        x=gamma(gam,beta,offset);
        return x;
  } } }
 else if(beta==1) {
  randnum=rand()%10001;
  randnum=randnum/10000;
  return (offset-gam*log(randnum));
 }
 else {
       b= 1 + beta/2.71828;
       randnum=rand()%10001;
       randnum=randnum/10000;
       p=b*randnum;
       if(p > 1)   {
             y=0-log((b-p)/beta);
             randnum1=rand()%10001;
         randnum1=randnum1/10000;
             if(randnum1<pow(y,(beta-1)))
                   return y+gam;
             else             {
              x=gamma(gam,beta,offset);
                    return x;
             }     }
       else {
             y=pow(p,1/beta);
             randnum1=rand()%10001;
                 randnum1=randnum1/10000;
             if(randnum1<exp(-y))
              return y+gam;
            else         {
                  x=gamma(gam,beta,offset);
                 return x;
             } }}
while(x >(b-log(randnum*randnum*randnum1))){
  cout<<x<<endl;
  randnum=rand()%10001;
  randnum=randnum/10000;
  x=beta*(exp(a*log(randnum/(1-randnum))));
  randnum1=rand()%10001;
  randnum1=randnum1/10000;
}
return ((x+offset)/(beta*theta));}
```

4.4 Exponential Distribution

The Exponential Distribution is a special case of the gamma distribution. In exponential distribution, the value of $\gamma = 1$. The probability distribution curve for the exponential distribution is the curve in figure 4.5, where $\gamma= 1$. The pdf and the procedure for generating a random number is same as that for gamm distribution.

The source code for this distribution can be found in table 4.2.

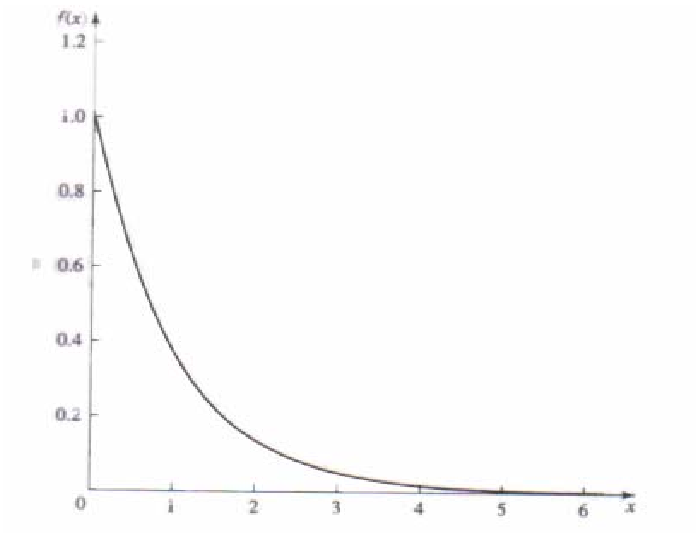


Figure 4.6 Probability Distribution curve for exponential distribution [1].

Table 4.2 Exponential Distribution Source Code

```
double exponential(double beta, double offset){
      double x = gamma(1,beta,offset); //call gamma()with gamma = 1.
}
```

4.5 Log-Normal Distribution

Lognormal distribution is derived from the normal distribution, which is in turn derived from the uniform distribution. It has three parameters; μ the scale parameter, σ the shape parameter and θ the location parameter. The PDF for the lognormal distribution is given by:

$$f(x) = \frac{e^{-((\ln((x-\theta)/m))^2/(2\sigma^2))}}{(x-\theta)\sigma\sqrt{2\pi}} \qquad x \geq \theta; m, \sigma > 0$$

Figure 4.7: PDF for the Log-Normal Distribution.

To derive a random number with lognormal distribution we first generate random normal variates with parameters μ, σ, θ. The random number that is generated is now exponentiated (anti-log) to get the corresponding lognormal random number.

1. Generate X = Normal (μ_n, σ_n, θ_n).
2. Lognormal = e^X.
3. The values μ_n, σ_n, θ_n are the normalized values of μ, σ, θ.

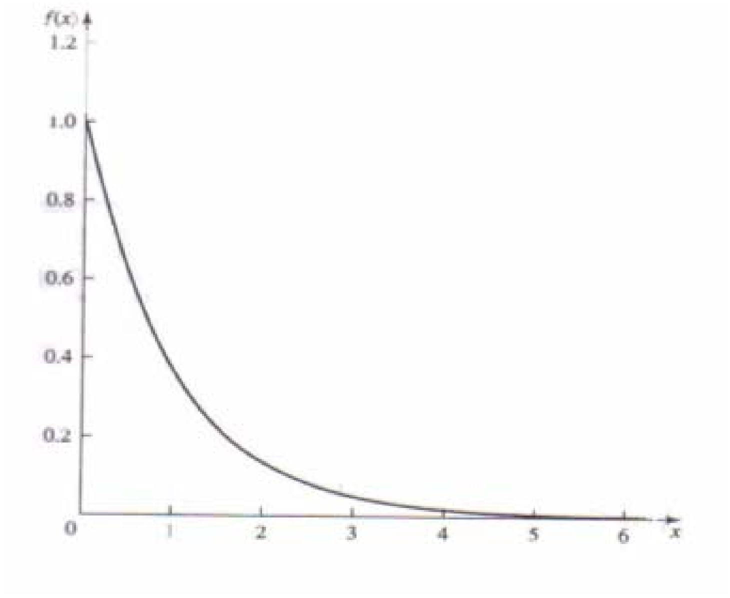


Figure 4.8 Probability Distribution Curve for Log-Normal Distribution [1].

The source code for this distribution can be found in table 4.3.

Table 4.3 Log-Normal Distribution source code

```
#include <iostream.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <fstream.h>
double lognormal(double,double,double);
int main(void)
{
  double offset,mean,std1;
  ofstream outfile;
  double number;
  srand(unsigned(time(NULL)));
  outfile.open("random.txt");
  if(!outfile)
  {
        cout<<"cannot open file";
        exit(1);
  }
  cout<<"Enter value for "<<endl;
  cout<<"Offset:- ";
  cin>>offset;
  cout<<"Mean:- ";
  cin>>mean;
  cout<<"Standard Deviation:- ";
  cin>>std1;
  for(int i=0;i<50000;++i)
  {
    number=lognormal(offset,log(mean),log(std1));
    cout << number << endl;
      outfile<<number<<"\n";
  }
  return 0;
}
double lognormal(double offset,double mean,double std1)
{
  double r1,r2,r3=2,number;
  double const pii=3.14152;
  while(r3>=1)
  {
  r2=sin(rand()%361);
  r1=sin(rand()%361);
  r3=r1*r1+r2*r2;
  }
  number = exp(std1*r1*sqrt(-2*(log(r3)/r3)*sin(2*pii*r2))+mean) +
offset;
cout<<number<<endl;
if(isnan(number))
 number=lognormal(offset,mean,std1);
if(isinf(number))
 number=lognormal(offset,mean,std1);
 return number;
}
```

4.6 Weibull Distribution

Weibull distribution is derived from the uniform distribution. It has three input parameters γ is the shape parameter, μ is the location parameter and α is the scale parameter. The PDF for the Weibull distribution is given by the equation:

$$f(x) = \frac{\gamma}{\alpha}\left(\frac{x-\mu}{\alpha}\right)^{(\gamma-1)} \exp(-((x-\mu)/\alpha)^{\gamma})\ldots x > \mu, \gamma, \alpha > 0$$

Figure 4.9 PDF for Weibull Distribution

To generate a random number with the Weibull distribution we start from the CDF of Weibull distribution. The CDF is the cumulative distribution factor.

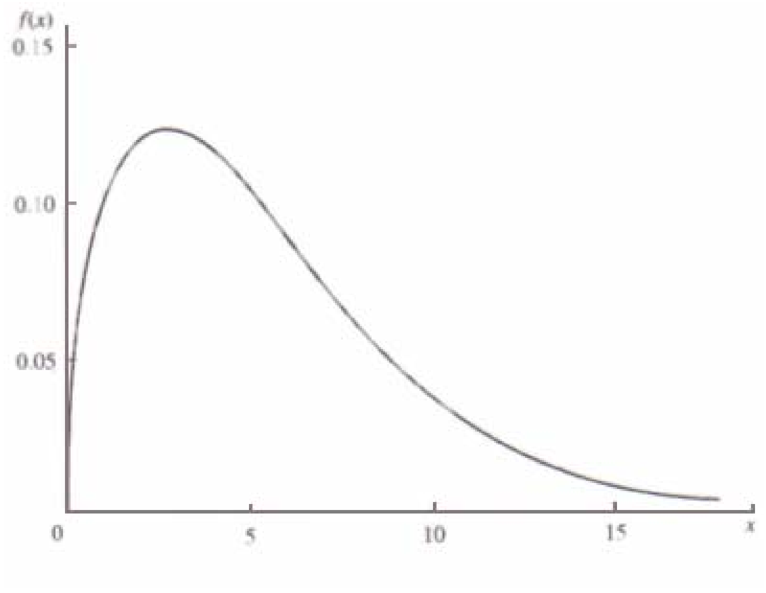


Figure 4.10 probability distribution curves for Weibull distribution [1].

1. The CDF of Weibull Distribution is given by $f(x) = 1 - e^{-(x/\alpha)^\wedge\gamma}$.
2. Let $f(x) = 1 - e^{-(x/\alpha)^\wedge\gamma} = R$.
3. Solving X we get $X = [-\ln(1 - R)]^{1/\gamma}$…… (1)
4. Generate a uniform random number R and put in equation 1. The resulting value of X is a random variate with Weibull distribution.

The source code for this distribution can be found in table 4.4.

Table 4.4 Weibull Distribution Source Code.

```
#include<iostream.h>
#include<fstream.h>
#include<math.h>
#include<time.h>
float weibull(float,float);
int main()
{
 float alpha,beta,retval;
 int i;
 ofstream ofile("weibull.txt");
 cout<<"Enter the Value for "<<endl<<"Beta :- ";
 cin>>beta;
 cout<<endl<<"alpha :- ";
 cin>>alpha;
 srand(unsigned(time(NULL)));
 for(i=0;i<50000;i++){
    retval=weibull(beta,alpha);
      if(isinf(retval)||isnan(retval))
        i--;
      else
    ofile<<retval<<endl;
  }
return 0;
}
float weibull(float beta, float alpha)}
      float x, randnum1;
      randnum1 = GenerateUniform(); // uniform random number
      x = 0 - log(1 - randnnum1);
      x = pow(x,(1/beta));
      if(x < 0) x = weibull(beta,alpha);
      return x;
}
```

4.7 Triangular Distribution

The triangular distribution is also based on the uniform distribution. It has three parameters as its input a min, a max and a mode. The mode is greater than the min value but is less than the max value. The PDF for the triangular distribution is given by the equations below, where a, b, c are the min, mode and the max respectively.

$$f(x)=\frac{2(x-a)}{(x-a)(x-b)},a\leq x\leq b$$

$$f(x)=\frac{2(c-x)}{(x-a)(x-b)},b<x\leq c$$

$$f(x)=0, \text{ elsewhere}$$

Figure 4.11 PDF for Triangular Distribution [1].

Following are the steps required to generate a random triangular variate from a uniform distribution.

1 Generate a random number R from the uniform distribution.

2 Compute $x = a + \sqrt{(b-a)(c-a)R}$.

3 if a<= X <=b then X is the random variate you need, else go to 4.

4 Do $x = c - \sqrt{(1-R)(c-a)(c-b)}$,

5 if b < x <= c then x is the random variate with triangular distribution, else repeat step 1.

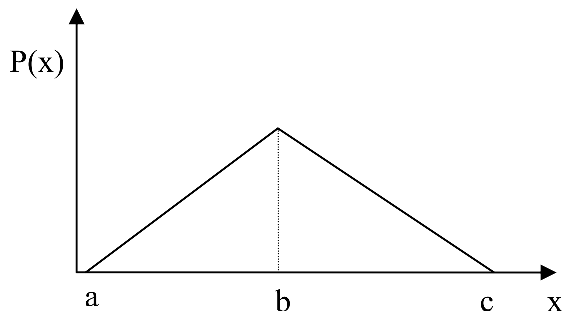


Figure 4.12 Probability Distribution curve for Triangular Distribution [1].

The source code for this distribution can be found in table 4.5

Table 4.5 Triangular Distribution Source Code

```
#include<iostream.h>
#include<fstream.h>
#include<stdlib.h>
#include<time.h>
#include<math.h>
float triangular(int,int,int,double);
int main(){
 ofstream ofile;
 int a,b,c;
 int i;
 double randnum;
 srand(unsigned(time(NULL)));
 ofile.open("triang.txt");
 cout<<"Enter a,b and c"<<endl;
 cin>>a>>b>>c;
 for(i=0;i<=50000;i++) {
 randnum=rand()%1001;
 randnum=randnum/1000;
 randnum=triangular(a,b,c,randnum);
 ofile<<randnum<<"\n";
 cout<<randnum<<endl;
 }
 ofile.close();
 return 0;
}
float triangular(int a,int b, int c, double randnum){
      double x;
      x = a + (sqrt((b-a)*(c-a)*randnum));
      cout<<randnum<<x<<"\n";
      if(x>=a && x<=b)
            return x;
      if(x>=b && x<=c)
      {
            x = (c-sqrt((1-randnum)*(c-a)*(c-b)));
            return x;
            cout<<randnum<<x<<"\n";
      }
}
```

4.8 Johnson Distribution

The Johnson distribution is a system of distribution and not a single distribution. They are categorized into the Johnson bounded and the Johnson unbounded distribution. Both have four input parameters α_1, α_2, γ, β. Where, β is the scale parameter, γ is the location parameter and α_1 and α_2 are the shape parameters. The value of $\beta > 0$ in case of the

Johnson bounded system and (β – γ) > 0 in case of the Johnson unbounded system of distribution. The PDF for the Johnson system of distribution is given by the equation below.

$$F(x) = \phi\left[\alpha_1 + \alpha_2 \ln\left(\frac{x-\gamma}{\beta - x}\right)\right] \text{ if } \gamma < x < \beta$$

$$F(x) = 0 \quad \text{Otherwise.}$$

Figure 4.13 PDF for Johnson bounded system

$$F(x) = \phi\left[\alpha_1 + \alpha_2 \ln\left[\frac{x-\gamma}{\beta} + \sqrt{\left(\frac{x-\gamma}{\beta}\right)^2 + 1}\right]\right] \cdots for \cdots -\infty < x < \infty$$

Figure 4.14 PDF for Johnson unbounded system

In the above equation the value Φ(x), represents a normal distribution with a mean $\mu = 0$ and $\sigma^2 = 1$. Given below are the mathematical steps required to generate a random variate with Johnson distribution.

1. Generate a random variate Z with distribution N (0, 1), where N is normal, with 0 and 1 as the parameters.
2. Compute $Y = EXP [(Z - \alpha_1) / \alpha_2$.
3. In case of Johnson bounded compute **a**, and in case of Johnson unbounded do **b**.

    3a. $X = (\gamma + \beta Y)(Y+1)$.

    3b. $X = \gamma + (\beta/2)(Y – 1/Y)$.

4. Return X as the random number.

Case $\alpha_2 < 1$

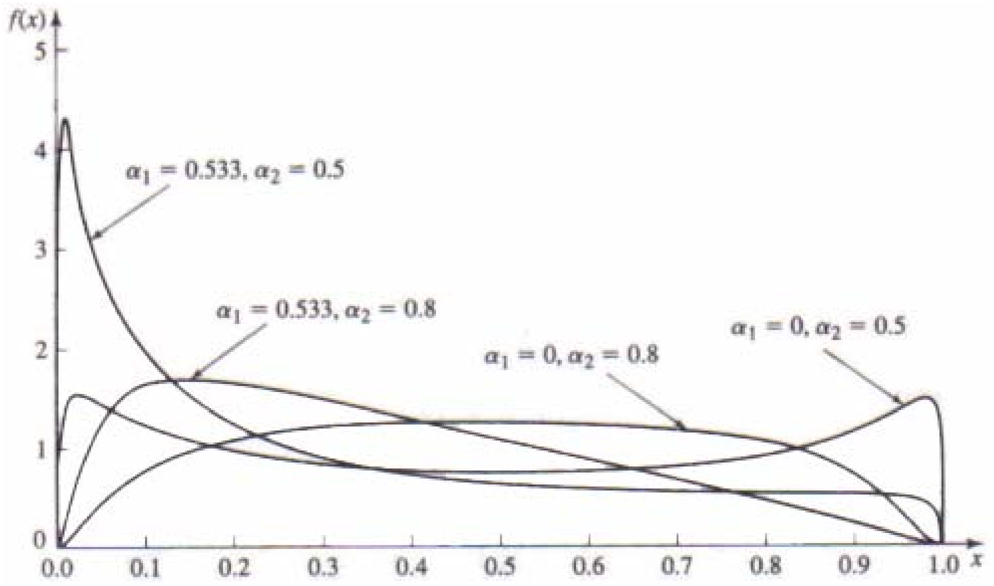


Figure 4.15 (a): Probability distribution curve for Johnson bounded system [1].

Case $\alpha_2 > 1$

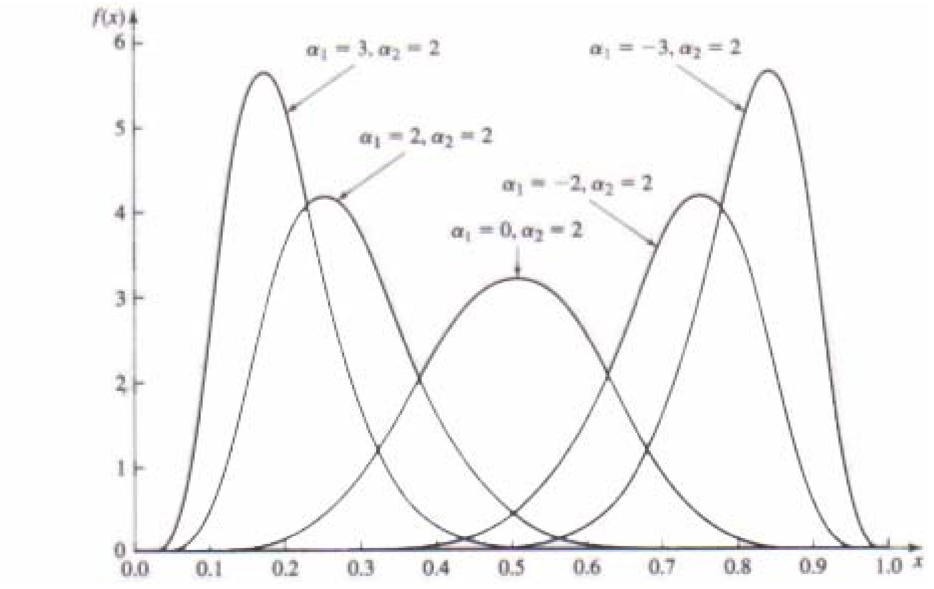


Figure 4.15 (b): Probability distribution curve for Johnson bounded system [1].

Case $\alpha_2 > 1$

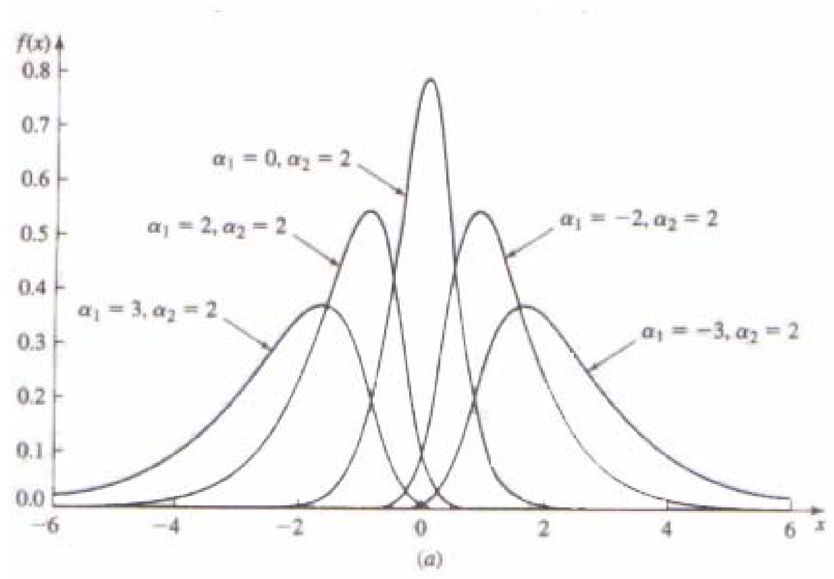


Figure 4.15 (c): Probability distribution curve for Johnson unbounded system [1].

Case $\alpha_2 < 1$

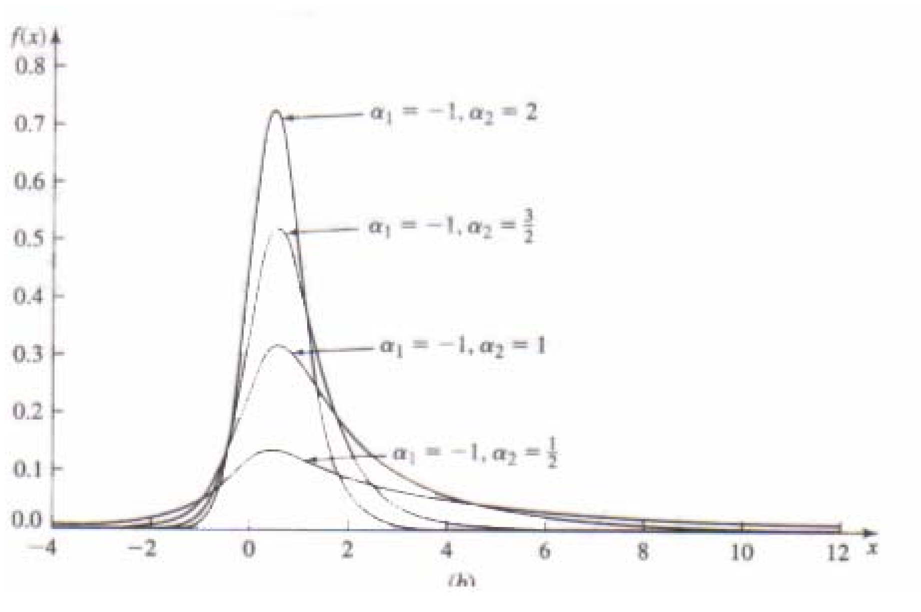


Figure 4.15 (d): Probability distribution curve for Johnson unbounded system [1].

The source code for the Johnson distribution can be found in table 4.6.

Table 4.6 Johnson Distribution Source Code

```
#include <iostream.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <fstream.h>
double johnson(double,double,double,double,bool);
int main(void)
{
  double alpha1,alpha2,beta,gamma;
  ofstream outfile;
  double number;
  srand(unsigned(time(NULL)));
  outfile.open("johnson.txt");
  if(!outfile)
  {
        cout<<"cannot open file";
        exit(1);
  }
  cout<<"Enter value for "<<endl;
  cout<<"alpha1:- ";
  cin>>alpha1;
  cout<<"alpha2:- ";
  cin>>alpha2;
  cout<<"Beta:- ";
  cin>>beta;
  cout<<"Gamma:- ";
  cin>>gamma;
  for(int i=0;i<50000;++i)  {
   number=johnson(alpha1,alpha2,beta,gamma,true);
// in the above function
//true will return bounded johnson
//while false will return unbounded
    cout << number << endl;
      outfile<<number<<"\n";  }
  return 0;}

double johnson(double alpha1, double alpha2, double beta, double gamma,
bool bounded)
{
      float y, randnum1;
      randnum1 = normal(0,1); // normal random number
      y = exp((randnum1-alpha1)/alpha2);
      if(bounded)
      y = (gamma + beta * y)(y + 1);
      else
      y = gamma + (beta/2)(y - 1/y);
      return y;
}
```

4.9 Discrete Distribution

Discrete distribution is based on empirical data. Discrete distribution take pairs of numbers and their probability value as input. The empirical data are provided as follows. P(0) = 0.2, p(1)=0.3, p(2) = 0.5. Here the value p(x) represents the probability value for x. The probability distribution of the discrete distribution for the above empirical data is

$$\begin{aligned} f(x) &= 0,\ldots x < 0 \\ f(x) &= 0.2,..0 \le x < 1 \\ f(x) &= 0.5,\ldots 1 \le x < 2 \\ f(x) &= 1,\ldots.2 \le x \end{aligned}$$

Figure 4.16 PDF for discrete distribution

To generate a random number, with the above PDF, we generate a uniform random number between 0 and 1. This number is the value for f(x). Depending on the value of f(x) the corresponding value of x is returned. For example, if f(x) = 0.4 then x=2 is returned. The Discrete empirical distribution are mostly used in simulation of networking concepts to represent events like “if a signal transmitted was received correctly”, or “if there was any collision”. They are normally used at places that can have only a particular set of values as their result, which is predefined at the time simulation was designed.

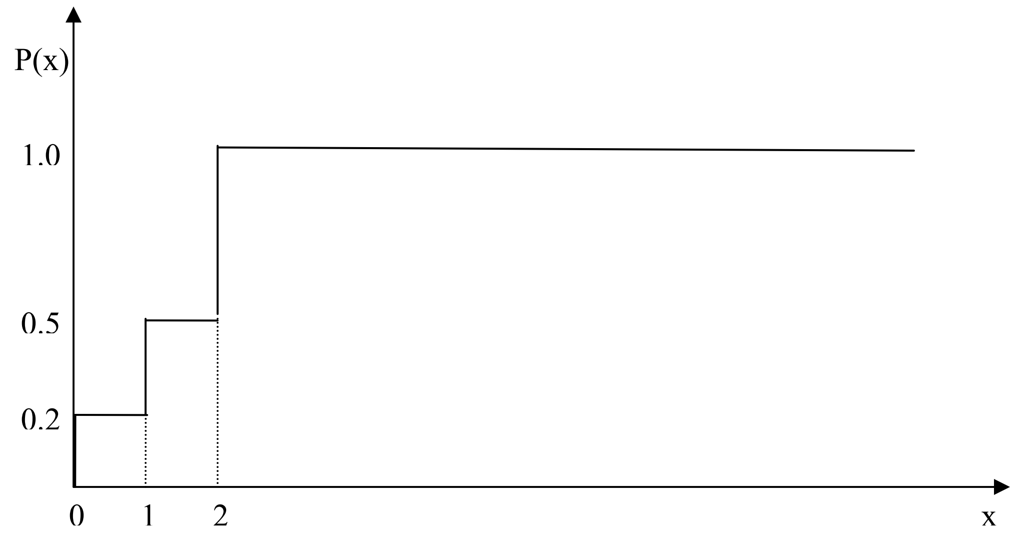


Figure 4.17 Probability distribution curves for Discrete Distribution [1].

The source code for this distribution can be found in Table 4.7.

Table 4.7 Discrete Distribution Source Code

```
double discrete ( double v[], int vlen, double p[], int plen)
{
 int i,j, escape =1;
 double x;
if(vlen == plen)
{
 //code for sorting the inputs v[] and p[] in ascending order
 for(i=0;i<vlen-1;i++)
 {
      for(j=1;vlen-i;j++)
        {
            if(v[i]<v[i+j])
               //swap v[i] and v[i+j]
            v[i]=v[i]+v[i+j];
            v[i+j]=v[i]-v[i+j];
            v[i]=v[i]-v[i+j];
            p[i]=p[i]+p[i+j];
            p[i+j]=p[i]-p[i+j];
            p[i]=p[i]-p[i+j];
            }
 }
x = GenerateUniform();
i=0;
while(escape == 1)
{
 if(x < p[i])
    {
      if(i==0)
        x= v[0];
      else
       x = v[i-1]
     return x;
    }
 i++;
 if(i == vlen && escape ==1)
     return 0;
}
}
else
 printf("error lengths of p[] and v[] are not matching");
}
```

4.10 Continuous Distribution

Continuous distribution is also based on empirical data. The empirical input is again pairs of numbers and their probability values. The only difference between discrete and continuous distribution would be that, in case of continuous the random numbers are calculated by interpolating. Using the same empirical data as the one used for discrete distribution and if we generate a value f(x) = 0.3, the corresponding value of x generated is given by;

$$\left(\frac{0.4-0.2}{x-1}\right)=\left(\frac{0.5-0.2}{2-1}\right)$$

Solving the above equation we get:

$$3x-3=2 \rightarrow x=\frac{5}{3} \rightarrow x=1.666$$

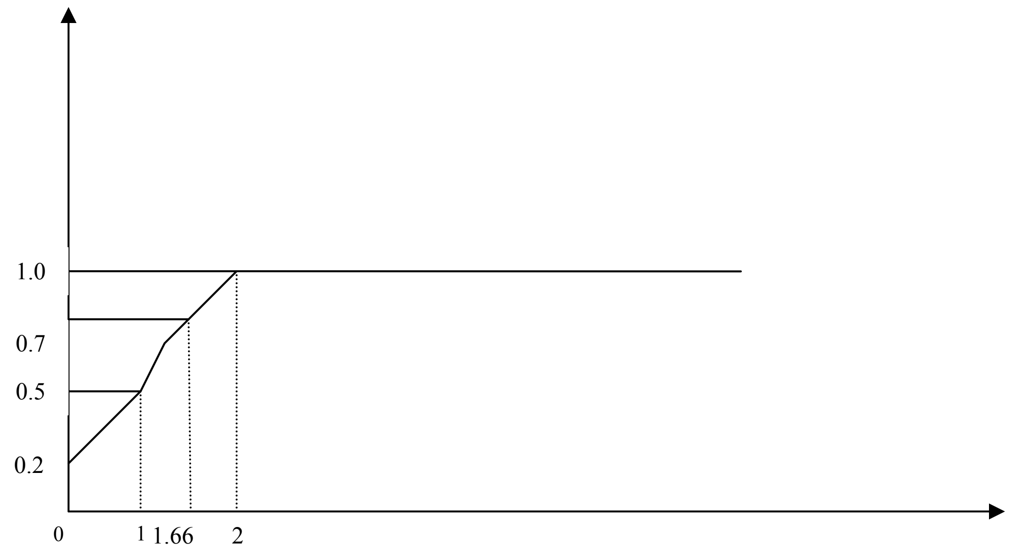


Figure 4.18 Probability distribution curves for continuous distribution [1].

The source code for this distribution is given in Table 4.8.

Table 4.8 Continuous Distribution Source Code

```
double continuous ( double v[], int vlen, double p[], int plen)
{
 int i,j, escape =1;
 double x,b,c,rhs,lhs;
if(vlen == plen)
{
 //code for sorting the inputs v[] and p[] in ascending order
 for(i=0;i<vlen-1;i++)
 {
      for(j=1;vlen-i;j++)
        {
            if(v[i]<v[i+j])
               //swap v[i] and v[i+j]
            v[i]=v[i]+v[i+j];
            v[i+j]=v[i]-v[i+j];
            v[i]=v[i]-v[i+j];
            p[i]=p[i]+p[i+j];
            p[i+j]=p[i]-p[i+j];
            p[i]=p[i]-p[i+j];
            }
 }
x = GenerateUniform();
i=0;
while(escape == 1)
{
 if(x < p[i])
    {
      if( i == 0)
      {
            b=0;
            c=0;
      }
      else
      {
            b=v[i-1];
            c=p[i-1];
      }
      rhs = ((v[i] - x)/( x - c));
      y = ((v[i] + rhs*b)/(rhs +1));
     return y;
    }
 i++;
 if(i == vlen && escape ==1)
     return 0;
}
}
else
 printf("error lengths of p[] and v[] are not matching");
}
```

4.11 Implementation in SPEEDES

Once all the above distribution were developed and packaged into individual functions, these function were added to the random number generator class library of SPEEDES “SpRandom.H”. The Source Code for “SpRandom.H” is given in table 4.9.

Table 4.9 SpRandom.H

```
#ifndef SpRandom_H
#define SpRandom_H

#include "SpPersistenceBaseClass.H"
#include "SpSystemConsts.H"

#include "SpMath.H"
//----------------------------------------------------------------------
-------
// Class: SpDensityFunction
//
// This base class defines the virtual functions required for the
rejection
// method used for generating some random numbers.
//----------------------------------------------------------------------
-------
class SpDensityFunction {
  public:
    virtual double f(double x) = 0;
    virtual double F(double x) = 0;
    virtual double operator() (double x) {return f(x);}
    virtual double GetMaxAmplitude() = 0;
    virtual double GetLoLimit() = 0;
    virtual double GetHiLimit() = 0;
  protected:
  private:
};

class SpRayleighDensityFunction : public SpDensityFunction {
  public:

    SpRayleighDensityFunction(double alpha = 1) {Alpha = alpha;}
    void SetAlpha(double alpha) {Alpha = alpha;}
    void SetHiLimit(double hiLimit) {HiLimit = hiLimit;}

    virtual double f(double x) {
      double value = 0;
      if (0 < x) {
        double alpha2 = Alpha * Alpha;
        value = (x / alpha2) * exp(-(x * x) / ( 2 * alpha2));
      }
      return value;
    }

    virtual double F(double) {return 0;}

    virtual double GetMaxAmplitude() {return 1.0 / (Alpha *
SP_SQRT_E);}
    virtual double GetLoLimit() {return 0;}
    virtual double GetHiLimit() {return HiLimit;}

  protected:

  private:
    double Alpha;
    double HiLimit;
    static double SP_SQRT_E;
};

//---------------------------------------------------------------------
-------
// Class: SpBaseRandom
//
// SpBaseRandom class provides all of the APIs for generating various
random
// numbers. The method for generating uniform random numbers between
[0, 1]
// is supported through virtual functions in child classes. Also, the
getting
// and setting of seeds is provided through virtual functions to
support
// rollbackable versions of the random number generator. There is no
state
// associated with SpBaseRandom, although it does support the
generation of
// random bits.
//---------------------------------------------------------------------
-------
class SpBaseRandom : private SpPersistenceBaseClass {
  public:

    virtual int GetSeed() = 0;
    virtual void SetSeed(int seed) = 0;
    virtual void ExchangeSeed(int &seed) = 0;

    virtual operator int() = 0;

    int operator =(int seed) {
      SetSeed(seed);
      return seed;
    }

    int operator ++() {
      int s = GetSeed() + 1;
      SetSeed(s);
      return s;
    }

    int operator ++(int) {
```

```
    int s = GetSeed();
    SetSeed(s + 1);
    return s;
  }

  int operator --() {
    int s = GetSeed() - 1;
    SetSeed(s);
    return s;
  }

  int operator --(int) {
    int s = GetSeed();
    SetSeed(s - 1);
    return s;
  }

  int operator += (int seed) {
    int s = GetSeed() + seed;
    SetSeed(s);
    return s;
  }

  int operator -= (int seed) {
    int s = GetSeed() - seed;
    SetSeed(s);
    return s;
  }

  int operator |= (int seed) {
    int s = GetSeed() | seed;
    SetSeed(s);
    return s;
  }

  int operator &= (int seed) {
    int s = GetSeed() & seed;
    SetSeed(s);
    return s;
  }

  int operator ^= (int seed) {
    int s = GetSeed() ^ seed;
    SetSeed(s);
    return s;
  }

  int operator *= (int seed) {
    int s = GetSeed() * seed;
    SetSeed(s);
    return s;
  }

  int operator /= (int seed) {
    int s = GetSeed() / seed;
    SetSeed(s);
    return s;
    }

    int operator <<= (int seed) {
      int s = GetSeed() << seed;
      SetSeed(s);
      return s;
    }

    int operator >>= (int seed) {
      int s = GetSeed() >> seed;
      SetSeed(s);
      return s;
    }

// Flat between [0, 1]
    virtual double GenerateUniform() = 0;

// Random Bit methods
    int GenerateBit();
    int GenerateBits(int numBits);

// Random flat integer between [lo, hi]
    int GenerateInt(int lo, int hi) {
      int r = (int) ((hi - lo + 1) * GenerateUniform() + lo);
      if (r > hi) {
        r = hi;
      }
      return r;
    }

// Random flat double between [lo, hi)
    double GenerateDouble(double lo = 0.0, double hi = 1.0) {
      return((hi - lo) * GenerateUniform() + lo);
    }

// Random exponential distribution f(t) = timeConstant * exp(-t /
timeConstant)
    double GenerateExponential(double timeConstant) {
      return - timeConstant * log(GenerateUniform());
    }

// Random Laplace distribution is same as exponential distribution but
can be
// negative or positive
    double GenerateLaplace(double timeConstant) {
      double rv = GenerateDouble(-1.0, 1.0);
      double sign = 1.0;
      if (rv < 0) {
        sign = -1.0;
      }
      rv = rv * sign;
      return -sign * timeConstant * log(rv);
    }

//f(x) = (x/alpha^2)exp(-x^2/(2*alpha^2) where 0<= x
    double GenerateRayleigh(double alpha) {
      return(sqrt(-2.0 * alpha * alpha * log(GenerateUniform())));
    }

// f(x) = (power+1)*x^power where 0<= x<= 1
    double GeneratePower(double power) {
      return pow(GenerateUniform(), 1.0 / (power + 1));
    }

// f(x) = (power+1)*(1-x)^power where 0<= x<= 1
    double GenerateReversePower(double power) {
      return(1 - pow(GenerateUniform(), 1.0 / (power+1)));
    }

// f(x) = 2x where 0<= x<= 1
    double GenerateTriangleUp() {
      return GeneratePower(1);
    }
//weibull random number generator
double weibull(double alpha, double beta,double offset)
 {
double x;//variable to store intermediate results
 x=exp(log(0-log(1-GenerateUniform()))/beta);
 //cout<<x<<endl;
 x=(alpha*x)+offset;
 return x;
 }
//log normal random number generator
/*double lognormal(double offset,double mean,double std1)
{
  double r1,r2,r3=2;
  double const pii=3.14152;
  double number;
  while(r3>=1)
  {
  r1=GenerateUniform();
  r2=GenerateUniform();
  r3=r1*r1+r2*r2;
  }
  number = exp(std1*r1*sqrt(-2*(log(r3)/r3)*sin(2*pii*r2))+mean) +
offset;
  if(isnan(number))
    number=lognormal(offset,mean,std1);
  if(isinf(number))
    number=lognormal(offset,mean,std1);
  return number;
}*/
//gamma random number generator
double gamma(double gam,double beta,double offset)
{
 double x,a,b,c,d,v,y,z,w,p,theta,randnum,randnum1;
 theta=4.5;
 if(beta > 1)
 {
 a=1/(sqrt(2*beta - 1));
 b=beta - log(4.0);
 c=beta + 1/a;
 d=1+log(theta);
 randnum=GenerateUniform();
```

```
randnum1=GenerateUniform();
v=a*log(randnum/(1-randnum));
y=beta*exp(v);
z=randnum*randnum*randnum1;
w=b+c*v-y;
if(isinf(y)||isnan(y))
{
  x=gamma(gam,beta,offset);
         return x;
}
else
{
 if((w+d-theta*z)>=0)
         return y;
 else if(w>=log(z))
         return y;
 else
 {
         x=gamma(gam,beta,offset);
         return x;
 }
}
}
else if(beta==1)
{
 randnum=GenerateUniform();
 return (offset-gam*log(randnum));
}
else
{
        b= 1 + beta/2.71828;
        randnum=GenerateUniform();
        p=b*randnum;
        if(p > 1)
        {
                y=0-log((b-p)/beta);
                randnum1=GenerateUniform();
                if(randnum1<pow(y,(beta-1)))
                        return y+gam;
                else
                       {
             x=gamma(gam,beta,offset);
                         return x;
                }
        }
        else
        {
                y=pow(p,1/beta);
                randnum1=GenerateUniform();
                if(randnum1<exp(-y))
                 return y+gam;
                else
                {
                     x=gamma(gam,beta,offset);
                    return x;
                }
}
```

```
}
{
  randnum=GenerateUniform();
  x=beta*(exp(a*log(randnum/(1-randnum))));
  randnum1=GenerateUniform();
}
return ((x+offset)/(beta*theta));
}
//Exponential Random Number Generator
double expo(double gam,double offset)
{
 double x,a,b,c,d,v,y,z,w,p,theta,randnum,randnum1;
 double beta=1.0;
 theta=4.5;
 if(beta > 1)
 {
 a=1/(sqrt(2*beta - 1));
 b=beta - log(4.0);
 c=beta + 1/a;
 d=1+log(theta);
 randnum=GenerateUniform();
 randnum1=GenerateUniform();
 v=a*log(randnum/(1-randnum));
 y=beta*exp(v);
 z=randnum*randnum*randnum1;
 w=b+c*v-y;
 if(isinf(y)||isnan(y))
 {
   x=gamma(gam,beta,offset);
          return x;
 }
 else
 {
  if((w+d-theta*z)>=0)
          return y;
  else if(w>=log(z))
          return y;
  else
  {
          x=gamma(gam,beta,offset);
          return x;
  }
 }
 }
 else if(beta==1)
 {
  randnum=GenerateUniform();
  return (offset-gam*log(randnum));
 }
 else
 {
        b= 1 + beta/2.71828;
        randnum=GenerateUniform();
        p=b*randnum;
        if(p > 1)
        {
                y=0-log((b-p)/beta);
```

```
                 randnum1=GenerateUniform();
                 if(randnum1<pow(y,(beta-1)))
                         return y+gam;
                 else
                        {
              x=gamma(gam,beta,offset);
                          return x;
                 }
         }
         else
         {
                 y=pow(p,1/beta);
                 randnum1=GenerateUniform();
                 if(randnum1<exp(-y))
                  return y+gam;
                 else
                 {
                      x=gamma(gam,beta,offset);
                     return x;
                 }
 }
}
{
  randnum=GenerateUniform();
  x=beta*(exp(a*log(randnum/(1-randnum))));
  randnum1=GenerateUniform();
}
return ((x+offset)/(beta*theta));
}
//Triangular Distribution RNG
double triangular(int a,int b, int c)
{
        double x;
      double randnum;
      randnum=GenerateUniform();
        x = a + (sqrt((b-a)*(c-a)*randnum));
        //cout<<randnum<<x<<"\n";
        if(x>=a && x<=b)
                return x;
        else if(x>=b && x<=c)
        {
                x = (c-sqrt((1-randnum)*(c-a)*(c-b)));
                return x;
                //cout<<randnum<<x<<"\n";
        }
      else
        {
         x = triangular(a,b,c);
         return x;
        }
}
//-----------------------------------

double lognormal(double mean,double std)
{
  double r1,r2,r3=2;
  double pii=3.14152;
  double mean1,std1;
  double number;
  mean1=log((mean*mean)/(sqrt(mean*mean + std)));
  std1=log((std+(mean*mean))/(mean*mean));
  while(r3 >= 1)
  {
  r1=GenerateUniform();
  r2=GenerateUniform();
  r3=r1*r1+r2*r2;
  }
  number = exp(std1*r1*sqrt(-2*(log(r3)/r3)*sin(2*pii*r2))+mean1+mean);
  if(isnan(number)||number <(std1/10))
    number=lognormal(mean,std1);
  if(isinf(number)||number >(mean*10))
    number=lognormal(mean,std1);
  return number;
}
//-------------------------------------------------------------


// Poisson Discrete Distribution
int discrete(double mean)
{
 double a =exp(-mean);
 double r = 1;
int x=-1;
while(r>a)
{
 r=r*GenerateUniform();
 x=x+1;
}
return x;
}

// f(x) = 2(1-x)
    double GenerateTriangleDown() {
      return(1.0 - GeneratePower(1));
    }

// f(x) = (n1+n2+1)!/(n1!*n2!) * [t^n1 *(1-t^n2)]
    double GenerateBeta(int n1, int n2);

// f(x) = 1/(sqrt(2*pi) * sigma) * [exp(-(x-mean)^2/(2*sigma^2))]
    double GenerateGaussian(double mean, double sigma);

// f(x) is provided by the user(rejection method used)
    double GenerateDensityFunction(SpDensityFunction *
densityFunction);

// Random vector with given magnitude(default, random unit vector)
    void GenerateVector(double vector[3], double magnitude = 1.0);

// random vector with random magnitude given by mean and sigma
magnitudes
    void GenerateVector(double vector[3],
                        double meanMagnitude,
                        double sigmaMagnitude) {
```

```
      GenerateVector(vector, GenerateGaussian(meanMagnitude,
sigmaMagnitude));
    }
    double GenerateCauchy(double alpha);

// Error function = area under gaussian curve from 0 to x
// note, Erf(infinity = 0.5
    double Erf(double x);

// Inverse error function determines x given Erf(x)
    double InvErf(double erfx);

// Factorial = n *(n-1) *(*n-2) * ... * 1
    double Factorial(int n);

  protected:

  private:
    void ComputeBetatfF(double time,
                       double &f,
                       double &F,
                       double *beta_coeff_f,
                       double *beta_coeff_F,
                       double beta_n1,
                       double beta_n2);

};

//-----------------------------------------------------------------------
-------
// Class: SpUniformLCG
//
// Linear Congruential Generator provides very fast random number
generation
// but not of high quality. Applications should never directly use this
class
// since it requires a child to provide the representation of the
seed(which
// may be a rollbackable integer or a normal integer).
//-----------------------------------------------------------------------
-------
class SpUniformLCG : public SpBaseRandom {
  public:

    virtual double GenerateUniform() {
      int seed = GetSeed();
      seed = (seed * SP_IA + SP_IC) % SP_IM;
      SetSeed(seed);
      return fabs(double(seed) / double(SP_IM));
    }

  protected:

  private:
    static int SP_IA;
    static int SP_IC;
    static int SP_IM;
```

```
};

//----------------------------------------------------------------------
-------
// Class: SpUniformRandBits
//
// Random Bit Generator provides very high quality random numbers, but
it
// is slow. Applications should never directly use this class since it
// requires a child to provide the representation of the seed(which may
be
// a rollbackable integer or a normal integer)
//----------------------------------------------------------------------
-------
class SpUniformRandBits : public SpBaseRandom {
  public:
    virtual double GenerateUniform() {
      int ranBits = GenerateBits(SP_NBITS);
      return(((double) ranBits) / (1.0 + (double) SP_MAX_32_BIT_INT));
    }

  protected:

  private:
    enum {
      SP_NBITS = 31,
      SP_MAX_32_BIT_INT = 0x7fffffff
    };

};

// SpRandom is the normal way people should use the random number
generator
// when rollback support is not required. It is slow, but very good. If
// rollback support is needed, you should use the RB_SpRandom class
which is
// identical to this class except that it uses an RB_int instead of an
int
// to represent its seed.

class SpRandom : public SpUniformRandBits {
  public:
    SpRandom(int seed = SpSeedInit) {SetSeed(seed);}

    virtual int GetSeed() {return Seed;}
    virtual void SetSeed(int seed);
    virtual void ExchangeSeed(int &seed) {
      int i = Seed;
      SetSeed(seed);
      seed = i;
    }
    virtual operator int() {return Seed;}

  protected:

  private:
    int Seed;
```

```
};

// SpFastRandom is a very fast random number generator but it is not of
// high quality. Use RB_FastRandom when rollback support is needed.

class SpFastRandom : public SpUniformLCG {
  public:
    SpFastRandom(int seed = 1) {Seed = seed;}

    virtual int GetSeed() {return Seed;}
    virtual void SetSeed(int seed) {Seed = seed;}
    virtual void ExchangeSeed(int &seed) {
      int i = Seed;
      Seed = seed;
      seed = i;
    }
    virtual operator int() {return Seed;}

  protected:

  private:
    int Seed;
};
#endif
```

CHAPTER 5: VALIDATION

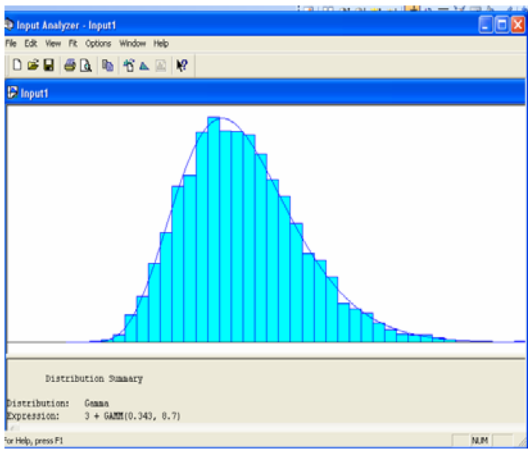
This chapter presents the validation of the random generators. The random generators were validated both with arena and SPEEDES.

5.1 Validation with Arena

Arena 7.01 was used to validate the random number generated by the random distribution functions given in chapter 4. The random numbers were run for a specific number of iteration during which they generated a set of random values. These random values were then stored in a text file by the C++ program. The text file was then opened in Arena's Input analyzer and the distribution curve was studied. Arena input analyzer has a function "Fit All" which tries to validate the current distribution to any of the distribution in its library. The Random function was considered validated if the curve obtained in arena was similar to the theoretical curve given in [1].

The random functions generated valid random numbers with respect to the distribution used to generate them. Also, due the fact that these random generators were based on a rollback-able random number generator, the random numbers generated by them were also rollback-able. The rollback ability was validated by using these functions to generate events for the SPEEDES shuttle model running on computers distributed across a network. The following diagrams will provide a detailed validation for the correctness of the random number generators developed.

Curve obtained in Arena Input Analyzer



```
        Distribution Summary

Distribution:   Gamma
Expression:     3 + GAMM(0.343, 8.7)
```

Theoretical Curve from Reference 2

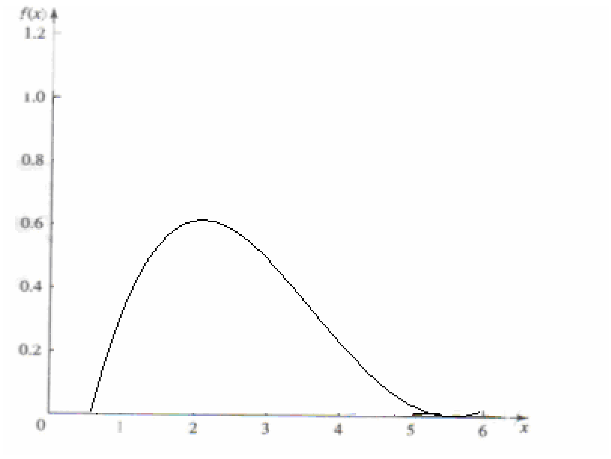


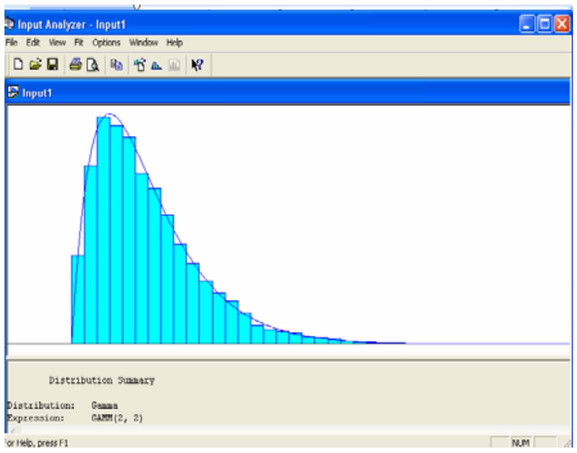
Figure 5.1 Lognormal distribution validations

5.1.1. Lognormal Distribution Validation Procedure

- The Program source code for the random number generators was compiled and executed, with an input set I.
- The number of iterations, i.e., the length of time for which each program ran was 50,000 iterations, producing 50,000 random numbers
- The output was stored in a ".txt "file in the program's directory.
- The output file was transferred via FTP to a windows machine, where it was opened using "Arena Input Analyzer".
- The Input analyzer plotted the curve for the output file; this curve was the fitted to the corresponding random number generator using the "fit all" function. The fitted curve was then visually compared to the theoretical curve from reference [1] for the same set of inputs I. The two curves are shown in Figure 5.1.

Curve obtained in Input Analyzer α = 2

Theoretical Curve from Reference 2



```
Distribution Summary

Distribution:   Gamma
Expression:     GAMM(2, 2)
```

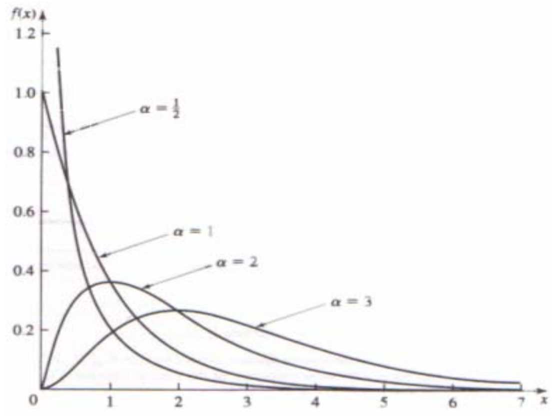
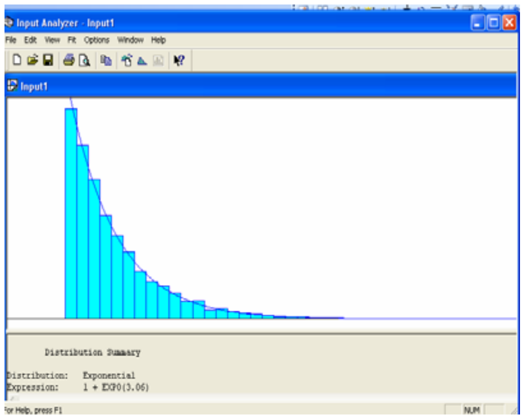


Figure 5.2: Gamma Distribution Validations

5.1.2. Gamma Distribution Validation Procedure

- The Program source code for the random number generators was compiled and executed, with an input set I.
- The number of iterations, i.e., the length of time for which each program ran was 50,000 iterations, producing 50,000 random numbers
- The output was stored in a “.txt “file in the program’s directory.
- The output file was transferred via FTP to a windows machine, where it was opened using “Arena Input Analyzer”.
- The Input analyzer plotted the curve for the output file; this curve was the fitted to the corresponding random number generator using the “fit all” function. The fitted curve was then visually compared to the theoretical curve from reference [1] for the same set of inputs I. The two curves are shown in Figure 5.2

Curve obtained in Arena Input Analyzer



```
        Distribution Summary

Distribution:   Exponential
Expression:     1 + EXPO(3.06)
```

Theoretical Curve from Reference 2

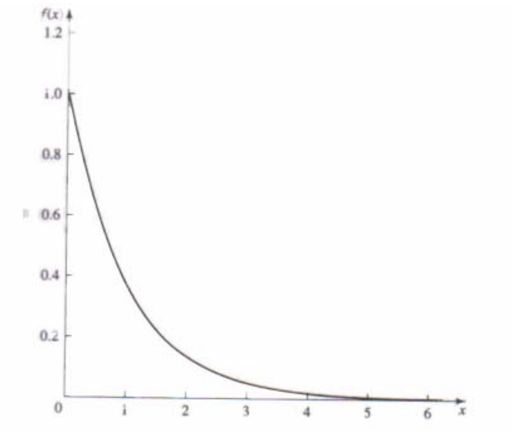
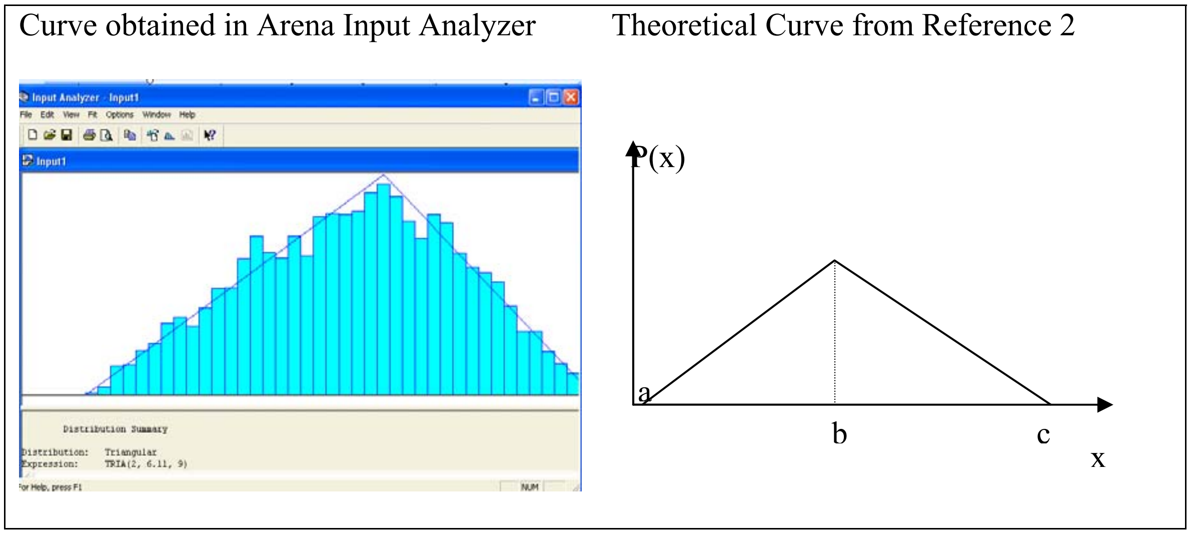


Figure 5.3 Exponential Distribution Validations

5.1.3. Exponential Distribution Validation Procedure

- The Program source code for the random number generators was compiled and executed, with an input set I.
- The number of iterations, i.e., the length of time for which each program ran was 50,000 iterations, producing 50,000 random numbers
- The output was stored in a ".txt "file in the program's directory.
- The output file was transferred via FTP to a windows machine, where it was opened using "Arena Input Analyzer".
- The Input analyzer plotted the curve for the output file; this curve was the fitted to the corresponding random number generator using the "fit all" function. The fitted curve was then visually compared to the theoretical curve from reference [1] for the same set of inputs I. The two curves are shown in Figure 5.3.



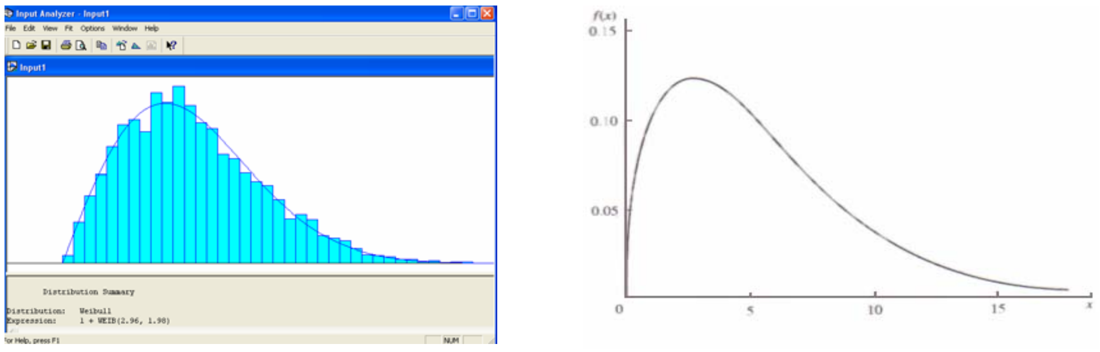
```
Distribution Summary

Distribution:   Triangular
Expression:     TRIA(2, 6.11, 9)
```

Figure 5.4 Triangular distribution validations

5.1.4. Triangular Distribution Validation Procedure

- The Program source code for the random number generators was compiled and executed, with an input set I.
- The number of iterations, i.e., the length of time for which each program ran was 50,000 iterations, producing 50,000 random numbers
- The output was stored in a ".txt "file in the program's directory.
- The output file was transferred via FTP to a windows machine, where it was opened using "Arena Input Analyzer".
- The Input analyzer plotted the curve for the output file; this curve was the fitted to the corresponding random number generator using the "fit all" function. The fitted curve was then visually compared to the theoretical curve from reference [1] for the same set of inputs I. The two curves are shown in Figure 5.4.

Curve obtained in Arena Input Analyzer   Theoretical Curve from Reference 2



```
Distribution Summary

Distribution:   Weibull
Expression:     1 + WEIB(2.96, 1.98)
```

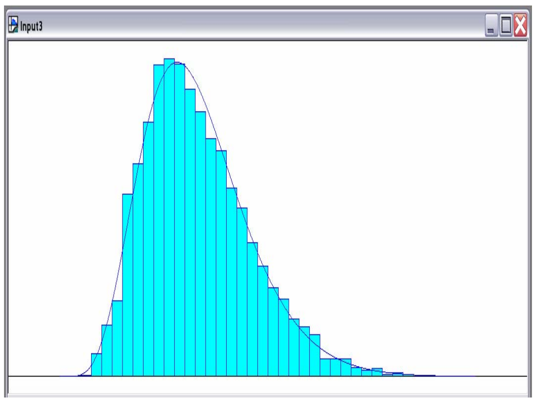
Figure 5.5 Weibull Distribution validations

5.1.5. Weibull Distribution Validation Procedure

- The Program source code for the random number generators was compiled and executed, with an input set I.
- The number of iterations, i.e., the length of time for which each program ran was 50,000 iterations, producing 50,000 random numbers
- The output was stored in a ".txt "file in the program's directory.
- The output file was transferred via FTP to a windows machine, where it was opened using "Arena Input Analyzer".
- The Input analyzer plotted the curve for the output file; this curve was the fitted to the corresponding random number generator using the "fit all" function. The fitted curve was then visually compared to the theoretical curve from reference [1] for the same set of inputs I. The two curves are shown in Figure 5.5.

Curve in Arena Input Analyzer α1 =3 , α2 = 2

Theoretical Curve from Reference 2.



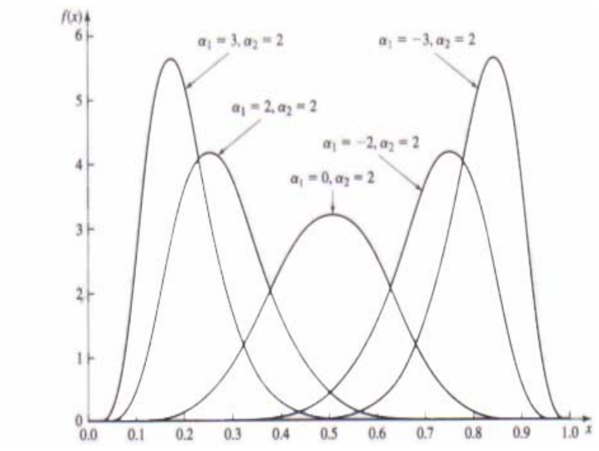
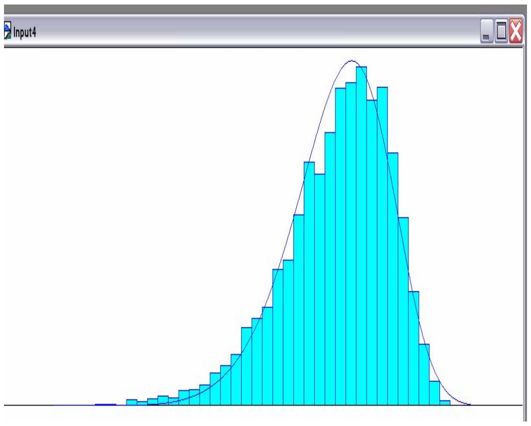


Figure 5.6.a: Johnson bounded Distribution Comparison

Curve in Arena Input Analyzer α1 =3 , α2 = 2

Theoretical Curve From Reference 2.



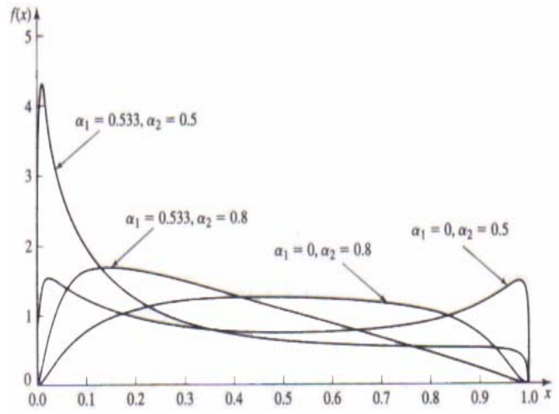
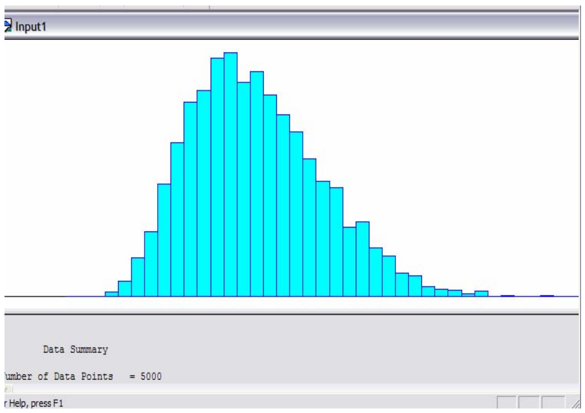


Figure 5.6.b Johnson bounded Distribution

Curve obtained in Arena Input Analyzer

α1 =2 , α2 = 2



```
Data Summary
umber of Data Points   = 5000
```

Theoretical Curve from Reference 2

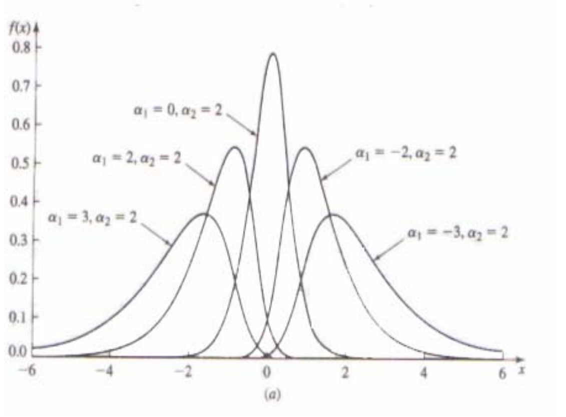
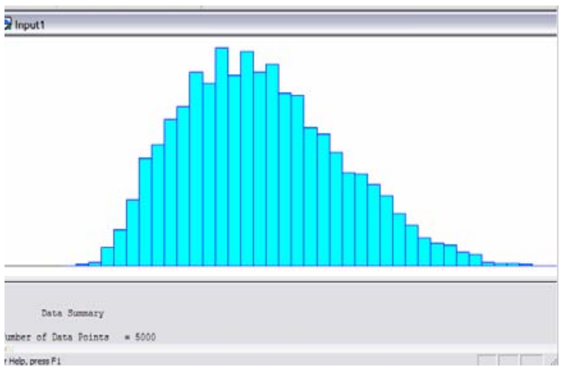


Figure 5.7.a Johnson Unbounded Distribution comparison

Curve obtained in Arena Input Analyzer

α1 = -1, α2 = 3/2



```
Data Summary
umber of Data Points   = 5000
```

Theoretical Curve From Reference 2.

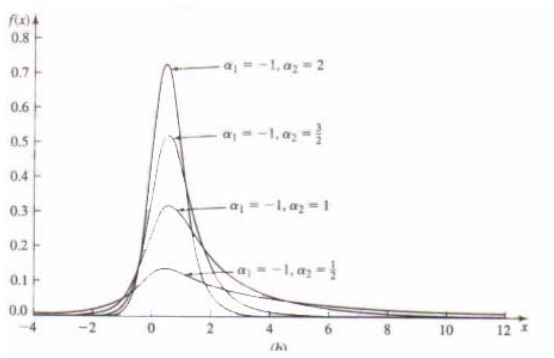


Figure 5.7.b Johnson unbounded Distribution

5.1.6. Johnson System of Distribution Validation Procedure

- The Program source code for the random number generators was compiled and executed, with an input set I.
- The number of iterations, i.e., the length of time for which each program ran was 50,000 iterations, producing 50,000 random numbers
- The output was stored in a ".txt "file in the program's directory.
- The output file was transferred via FTP to a windows machine, where it was opened using "Arena Input Analyzer".
- The Input analyzer plotted the curve for the output file; this curve was the fitted to the corresponding random number generator using the "fit all" function. The fitted curve was then visually compared to the theoretical curve from reference [1] for the same set of inputs I. The curves are shown in Figures 5.6(a), 5.6(b), 5.7(a) and 5.7(b).

5.2 Validation with SPEEDES.

The "NASA space shuttle model" previously developed in Arena was broken down into smaller modules to better understand the behavior of the various entities and processing objects. One of the problems faced in doing this was a lack of random number generators in SPEEDES. These random number generators represent various operation models in the ARENA simulation. After validating the random number generators "as stand alone functions", the next step was to validate them in the SPEEDES Shuttle Model. The

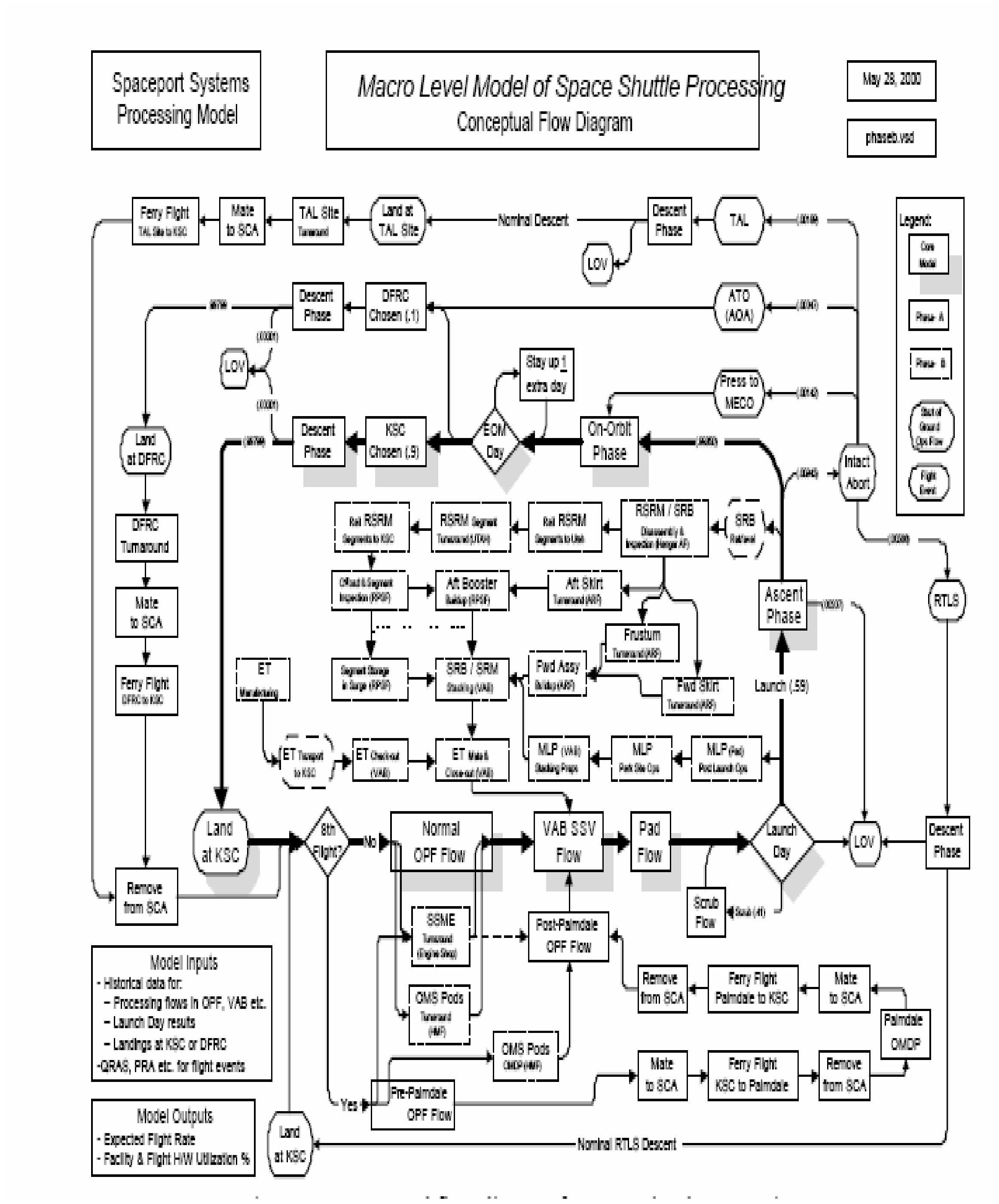


Figure 5.8 NASA Shuttle Model

NASA shuttle model is given in figure 5.8. The validation for SPEEDES was done by using these random number generators to represent the various operations like OPF, Launch, On Orbit, etc. The SPEEDES model was then executed. The output of this execution was then compared with the output obtained by running the arena model for the same set of inputs. The comparison is given in Table 5.1. The SPEEDES model takes the number of years the simulation has to be run as one of the inputs and the number of times this simulation has to be repeated.

Table 5.1 Result comparison ARENA and SPEEDES.

INPUTS	Result for ARENA	SPEEDES
Number of Repetition = 25 Time in Years = 10	Mean for Number of Flights = 69.48 Standard Deviation = 1.47535	Mean for Number of Flights = 68.8 Standard Deviation = 1.47196

CHAPTER 6: CONCLUSION AND FUTURE RESEARCH

This chapter provides a brief conclusion of the results that were obtained in the thesis performed and proposes some future research ideas that would help improve and widen the role of SPEEDES in the simulation industry.

6.1 Conclusions

The rollback-able random numbers were developed using the equation and theories of statistics, which define the various distributions. These random number generators were observed to be correct when their output were validated in Arena. They were also tested with SPEEDES by using these random number generators in the VTB shuttle model program. The output of the simulation was very similar to the output generated by the arena version of the simulation. The results of the comparison are given in Table 5.1.

Methodology utilized to Develop Random Number Generators

1. Random numbers are based on statistical distributions and equations.
2. The statistical equations corresponding to the random number generator being developed were studied. An algorithm was developed to generate random numbers based on these statistical equations.
3. These statistical equations basically give the probability P(x) of a number x being generated. Hence, given p(x) we can always generate the value of the corresponding x.
4. P(x) was generated by using the uniform number generator provided by the SPEEDES library. The P(x) generated was a valid value since the generateUniform() function return a float value in the range 0 to 1.

5. The output obtained from the algorithm was stored in a text file.
6. This output was then validated using Arena 7.01 Input Analyzer.

6.2 Contribution

6.2.1 Improvement to SPEEDES, Parallel/Distributed Simulation.

These random generators enhanced the capability of SPEEDES to perform operations in a simulation. The SPEEDES library lacked these random generators and hence, was not preferred for simulations involving operations. Adding these random generators to the SPEEDES library has enhanced and improved the SPEEDES environment to model operations efficiently

6.2.1 Utilization of statistics, computer architecture, operations, network and programming knowledge.

The concepts of statistics formed the foundation of this thesis. Statistical equations and concepts were utilized to develop the algorithms for the random generators. C++ was used to program these algorithms so that they can be used with the SPEEDES environment. Since SPEEDES is a parallel/distributed simulation environment the knowledge of computer architecture, networking and programming proved useful in understanding the SPEEDES environment, its working and its installation.

6.3 Further Research

Rollback-able random number generators are the key element in a successful parallel/distributed simulation. It provides a huge area for future study, research and development. The next generation of parallel distributed simulations is the HLA/RTI based simulation. The RTI software provides a set of services used by federates to coordinate their operations and data exchange during a runtime execution. Access to these services is defined by the HLA Interface Specification. RTI like SPEEDES uses the HLA specifications and a network connection to communicate with various simulations running on computers across the globe, making them faster. Thus, RTI provides for an area where there will be extensive research in developing efficient and accurate random number generators.

The other area of improvement that we observed during the thesis was the lack of a central repository for the SPEEDES code and a visual interface for writing SPEEDES programs. SPEEDES, though being open source software, has not gained much acceptance in the simulation industry in comparison to other tools. One reason for this is SPEEDES lack of a visual interface or IDE for programming. Also, it only runs on the Linux operating system, which considerably reduces the number of users with access to SPEEDES. We propose to develop a visual interface for SPEEDES capable of working both on Windows and Linux platforms. The code written using this software will be submitted to a central Linux server, where the user can compile/run/test his code. This makes sure that more people get access to the SPEEDES code, without really installing the SPEEDES library, which is a tough issue for a novice user. Also, since all the code is submitted to a central repository, a user can have access to codes already written by another user, encouraging reusability. Only users with the required access rights and permissions will be able to reuse another user’s codes. The interface will require a login and password before a user can start coding or even access code on it.

LIST OF REFERENCES

1. “Discrete Event System Simulation” Jerry Banks, John S. Carson, II Barry L. Nelson, David M. Nicol 3rd edition, Prentice Hall Publications, ISBN 0-13-088702-1.
2. “Parallel and distributed discrete event simulations: algorithms and application” Richard M. Fujimoto, December 2001, proceedings of the 25th conference on winter simulation.
3. “Parallel and Distributed Simulation, Richard M. Fujimoto” December 1999, Proceedings of the 31st conference on winter simulation---a bridge to the future – volume 1.
4. “Massively Parallel and distributed simulation of a class of discrete event systems: a different perspective” Pirooz, Vakli, July 1992, ACM transaction on modeling and computer simulations.
5. “Random numbers for simulation” Pierre L’Ecuyer, October 1990, communications of the ACM volume 33 issue 10.
6. “Efficient optimistic parallel simulations using reverse computation” Christopher D. Carothers, Kaylan S. Perumalla, Richard M. Fujimoto, May 1999 , proceedings of the 13th workshop on parallel and distributed simulation.
7. “An Analysis of a rollback-based simulation” Boris Lubachevsky, Adam Schwartz, Alan Weiss, April 1991 ACM transaction on modeling and computer simulation.
8. “Official SPEEDES Documentation and User Manual” Source: - http://www.speedes.com
9. Web reference” http://www.itl.nist.gov

10 Efficient and portable combined random number generators", P. L' Ecuyer, Jun 1988, communications of ACM, volume 31, Issue 6.

11 "Random number generators: good ones are hard to find", S.K. Park and K.Q.Miller, October 1988 , communications of the ACM volume 31 issue10.

12 "Uniform Random number generators", M. Donald MacLauren, George Masgalia, journal of the ACM, Volume 12 issue 1.

13 "Internet-based simulation using off=-the-shelf simulation tools and HLA", Steffen strabbugger, Thomas Schulze, Ulrich Klein and James Henriksen.

14. Linear and inversive pseudorandom numbers for parallel and distributed simulation", K. Entacher, A.Uhl, S. Wegenkittl, July 1998, Proceedings of the 12th workshop on parallel and distributed simulation, volume 28 issue 1.

15 "Speeding up distributed simulation using the time warp mechanism", Orna Berry, Greg Lomow, September 1986, Proceedings of the 2nd workshop on making distributed systems work.

16 "Distributed stochastic discrete-event simulation in parallel time streams", Krztsztof pawlikowski, Victor W.C, Yau and Don McNickel, Decemeber 1994. Proceedings of the 26th conference on winter simulation.

17 "Automatic load balancing in SPEEDES". Linda F. Wilson, David M. Nicol. Proceedings of the 27th conference on winter simulation.

18 "Supporting large-scale distributed simulation using HLA". Tainchi Lu, Chungan Lee, Wenyang Hsia, Mingtang Lin. July 2000, ACM Transactions on modeling and computer simulation.

19. “Scalable RTI-based parallelel simulation of networks”. Kalyan S. Perumalla, Alfred Park, Richard M. Fujimoto, George F. Riley. June 2003. Proceedings of the 17th workshop on parallel and distributed simulation.

20 “The Department of Defense High Level Architecture” Judith S. Dahmann, Richard M. Fujimoto, Richard M. Weatherly. December 1997. Proceeding of the 29th conference on winter simulation.

21 “The SPEEDES persistence framework and the standard simulation Architecture”. Dr. Jeffrey S. Steinman, Jennifer W. Wong. June 2003. Proceedings of the 17th workshop on parallel and dist4ributed simulations.