RAMAIAH UNIVERSITY OF APPLIED SCIENCES

Assignment

Course Code	CSC304A
Course Name	Computer Simulation
Programme	B.Tech
Department	CSE
Faculty	FET

Name of the Student	Prachi Poddar
Reg. No.	17ETCS002122
Semester/Year	05/2019
Course Leader(s)	Nithin Rao

Declaration Sheet

Student Name	Prachi Poddar		
Reg. No	17ETCS002122		
Programme	B.Tech	Semester/Year	05/2019
Course Code	CSC304A		
Course Title	Computer Simulation		
Course Date		to	
Course Leader	Nithin Rao		

Declaration

The assignment submitted herewith is a result of my own investigations and that I have conformed to the guidelines against plagiarism as laid out in the Student Handbook. All sections of the text and results, which have been obtained from other sources, are fully referenced. I understand that cheating and plagiarism constitute a breach of University regulations and will be dealt with accordingly.

Signature of the Student		Date	
Submission date stamp (by Examination & Assessment Section)			

Signature of the Course Leader and date	Signature of the Reviewer and date

1 Question 1

Solution to Question No. 1 Part A

1.1 Introduction to discrete-time and continuous-time simulation

Continuous-Time Models:

The basic approach in a continuous-time even-driven model is to simulate the event history for each micro-unit – that is, the timing of different types of events. Additional explanatory variables can be used to model the effects of the environment on the micro-unit. These dependencies can be modelled conveniently using hazard rates that give the instantaneous conditional probability density for experiencing the event conditional on the values of the explanatory variables given that the event has not yet been observed. Makes use of "Difference Equations".

Discrete-Time Models:

In discrete-time models, only the outcomes for discrete time periods are considered and no reference is made to the timing of events within a period. Thus, an aggregation over time is applied, resulting in a loss of information about the event history within the time period. However, the number of events of a given type that a micro-unit experiences within a time period may still be recorded. **(NATSEM, 1997)** Makes use of "Differential Equations".

1.2 Identify and explain advantage and disadvantages of discrete event simulation and continuous event simulation of a system by taking suitable example

Consider a 2-species Nicholson Bailey difference equation modified to include host density dependence.

$$x\ n+1\ = x\ n\ e^{r\times\left(1-\frac{x\ n}{K}\right)-a\times y\ n}$$
$$y\ n+1\ = x\ n\ \times(1-e^{-a\times y(n)})$$

We already know that discrete events are simulated using difference equations, here we have considered two equation system of difference equations, which we simulate in MATLAB and obtain the simulation graphs.

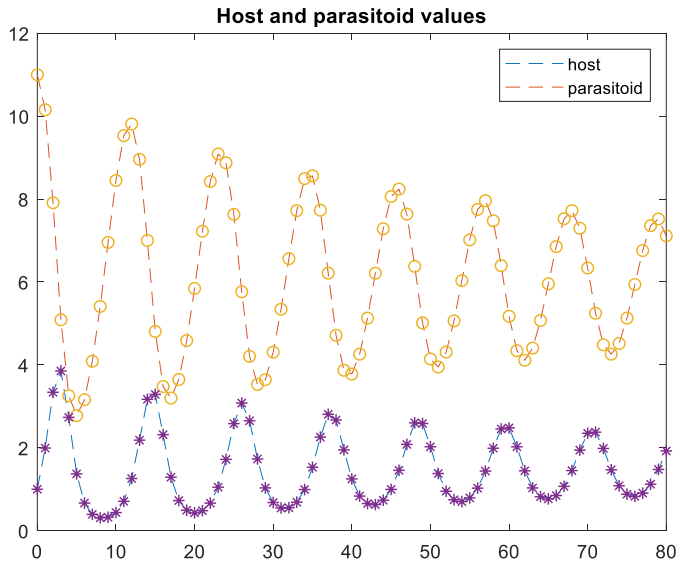

Figure 1-1 Nicholson Bailey Simulation

Consider the pair of first-order ordinary differential equations known as the Lotka-Volterra equations, or predator-prey model:

$$\frac{dx}{dt} = x - \alpha xy$$
$$\frac{dy}{dt} = -y + \beta xy$$

Continuous Simulation is done for differential equations, here we have considered a system of differential equations for which we have simulated the model.

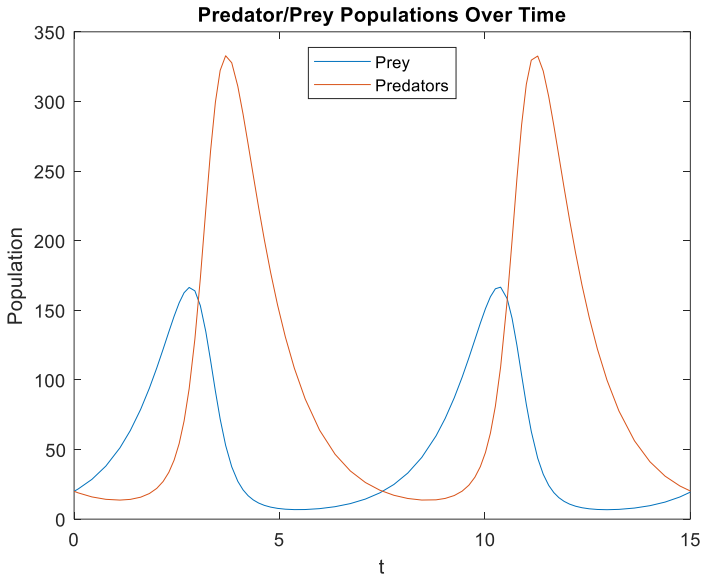

Figure 1-2 Lotka-Volterra Simulation

MATLAB codes for the respective graphs have been attached in Appendix A

1.2.1 Comparison of the two models

Both these models are some standard models for population simulation, the only difference is that one of them is for discrete events and the other is a continuous simulation.

Now we try to analyze the two models, the first obvious thing to do will be to determine the average population difference of the two populations, in the Nicholson-Bailey Model we can simply do so by just subtracting the discrete values at the different times over the complete period of time for example ye can compute x(0), y(0) and then obtain |x(0)-y(0)|.

While in the Lotka-Volterra Model, this cannot be done so, since they are differential equations, we need to integrate the two and then subtract them, this is much more tedious work and too much numeric precision than we need, making the statistical analysis of the model difficult, there are numerical methods that make it a little simpler, but most of the time the statistical parameters that we are after, it's not worth it. This is one of the reasons we prefer discrete models over the continuous ones. In practice the Lotka-Volterra model is also simulated discretely by using some numerical solvers, so as to make the analysis a little easier.

The Discrete Model is also more flexible in the sense that new parameters can be added to the difference equation easily without causing more computations to be made, but if new parameters are added to the Differential Equations, it might complicate the equation, converting it to Partial Differential Equations, making it even more difficult to solve and simulate.

1.3 Stance taken, justification and conclusion

Since only some theoretical and statistical properties have been discussed in this assignment, no general conclusions should be drawn about the relative merits of the continuous-time and the discrete-time approaches to dynamic microsimulation modelling. However, some of the arguments make a strong case in favour of a discrete-time modelling approach.

A first general result is that the conceptual simplicity of continuous time model is lost to a large degree if unobserved heterogeneity and other factors between partial processes are taken into account, the parameter estimation becomes even more complicated, thus for continuous time model to be operational, rather restrictive assumptions are required, such as the conditional independence of partial processes and specific functional form of duration dependence, like the Weibull specification. On the other hand, these problems can be dealt with in a multivariate discrete-time framework.

If the properties of competing-risk models are compared with the properties of discrete-time models, a clear advantage of the latter becomes obvious as far as the modelling of stochastic dependencies between partial processes is concerned. It is much simpler to account for such dependencies in a discrete-time approach. **(Heinz P. Galler, 1997)** If the continuous-time and the discrete-time approaches are compared with regard to the implications for modelling causal relationships, the basic continuous-time approach appears to be much simpler but also much less flexible than discrete-time models. Discrete-time models, on the other hand, do not require the assumption of conditional independence

Question 2

Solution to Question 1 Part B

1.4 Introduction

The given problem is a Monte-Carlo Simulation Method. The 'Monte Carlo' simulation technique involves conducting repetitive experiments on the model of the system under study, with some known probability distribution to draw random samples (observations) using random numbers. If a system cannot be described by a standard probability distribution such as normal, poisson, exponential, etc., an empirical probability distribution can be constructed.

The Probability Distribution for the rainfall when there was no rain the previous day, and also when there was rain on the previous day is given. The cumulative probability and the random digit assignment are determined for each of the table.

The Sequence of Random Numbers chosen for this 10-day simulation is:

{24, 59, 87, 22, 89, 96, 52, 24, 61, 21}

1.5 Simulation Table

Table 1 Distribution if it rained on previous day

Event	Probability	Cumulative Probability	Random Digit Assignment
No Rain	0.50	0.50	01-50
1cm Rain	0.25	0.75	51-75
2cm Rain	0.15	0.90	76-90
3cm Rain	0.05	0.95	91-95
4cm Rain	0.03	0.98	96-98
5cm Rain	0.02	1.00	99-100

Table 2 Distribution if it did not rain on previous day

Event	Probability	Cumulative Probability	Random Digit Assignment
No Rain	0.75	0.75	01-75
1cm Rain	0.15	0.90	76-90
2cm Rain	0.06	0.96	91-96
3cm Rain	0.04	1.00	97-100

```
run:
  day |    random number |    rainfall
    1 |                24 |           0
    2 |                59 |           0
    3 |                87 |           1
    4 |                22 |           0
    5 |                89 |           1
    6 |                96 |           4
    7 |                52 |           1
    8 |                24 |           0
    9 |                61 |           0
   10 |                21 |           0
Total Rainfall = 7
Number of Days without rain = 6
BUILD SUCCESSFUL (total time: 0 seconds)
```

Figure 1-3 Simulation Table for Rainfall

1.5.1 Simulation Program in Java

RAIN_SIM.java

```java
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package rain_sim;

import java.util.Random;

/**
 *
 * @author prachi
 */
public class RAIN_SIM {

    public static Integer[] gen_rn(Double[] distr) {
        Integer[] rn = new Integer[distr.length];
        int temp = 0;
        for (int i = 0 ; i < distr.length; i++) {
            temp += distr[i]*100;
            rn[i] = temp-1;
        }

        return rn;
    }
```

```
    public static void print_arr(Integer[] arr) {
        for (Integer i : arr) {
            System.out.print(i+" ");
        }
        System.out.println();
    }

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {

        Random rg = new Random();

        // if it rained the previous day
        Double prev_rain_distr[] = {0.50, 0.25, 0.15, 0.05, 0.03, 0.02};

        // if it did not rain on previous day
        Double no_prev_rain_distr[] = {0.75, 0.15, 0.06, 0.04};

        // create the random_no upper limit

        Integer[] prn_rn = gen_rn(prev_rain_distr);
        Integer[] nprn_rn = gen_rn(no_prev_rain_distr);

//        print_arr(prn_rn);
//        print_arr(nprn_rn);

        // generate 10 random numbers from 10 to 100
        // testing Integer rn[] = {67, 63, 39, 55, 29, 78, 70, 6, 78, 76};
        Integer rn[] = new Integer[10];
        for (int i = 0 ; i < rn.length ; i++) {
            rn[i] = rg.nextInt(89) + 10;
        }

        // assuming it has not rained on previous day
        Boolean has_rained_prev = false;

        Integer[] rainfall = new Integer[rn.length];
        Integer n_days_without_rain = 0;
        Integer total_rain = 0;

        for (int i = 0 ; i < rn.length ; i++) {
            if (!has_rained_prev) {
                for (int j = 0 ; j < nprn_rn.length ; j++) {
                    if (rn[i] <= nprn_rn[j]) {
                        rainfall[i] = j;
                        break;
```

```
                    }
                }
            } else {
                for (int j = 0 ; j < prn_rn.length ; j++) {
                    if (rn[i] <= prn_rn[j]) {
                        rainfall[i] = j;
                        break;
                    }
                }
            }

            has_rained_prev = rainfall[i] != 0;

            if (!has_rained_prev) {
                n_days_without_rain++;
            } else {
                total_rain += rainfall[i];
            }

        }

//        print_arr(rainfall);
        System.out.printf("%5s | %15s | %10s\n", "day", "random number", "rainfall");
        for (int i = 0 ; i < rainfall.length ; i++) {
            System.out.printf("%5d | %15d | %10d\n", i+1, rn[i], rainfall[i]);
        }

        System.out.printf("Total Rainfall = %d\nNumber of Days without rain = %d\n", total
_rain, n_days_without_rain);

    }

}
```

1.6 Result and Analysis

The Monte Carlo simulation technique consists of the following steps:

(1) Setting up a probability distribution for variables to be analyzed.

(2) Building a cumulative probability distribution for each random variable.

(3) Generating random numbers and then assigning an appropriate set of random numbers to represent value or range (interval) of values for each random variable.

(4) Conducting the simulation experiment using random sampling.

(5) Repeating Step – 4 until the required number of simulation runs has been generated.

(6) Designing and implementing a course of action and maintaining control.

From the Simulation Table constructed using the Monte-Carlo Method for the given problem we conclude that for the assumed Random Numbers in 2.1 the Total Rainfall is 7cm and the number of days without rain is 6, in total the simulation was performed for 10 days.

The highest rainfall was on Day 6 which was of 4cm.

Rainfall prediction done using this method is not so accurate since natural rain is so much dependent on many other factors such as geographical, ecological, global and local. Hence, we will be hardly successful in predicting the rainfall beyond a short term, such as predicting the rainfall for the next 50 years, the distribution is not complex enough to factor in all the variables that affect the rainfall.

The Random Numbers used in the simulation are generated using a LCG, which is pseudo-random number generator, the results for long term simulation hence will not be of any significance.

2 Question 3

Solution to Question 2 Part B

2.1 Introduction to problem solving approach

The problem is to implement an ATM Simulation System in Java for 12 hours, where the Service Time and Arrival Times are Random Integers from 1 to 6 and 1 to 4 respectively. The question specifies that one customer can take the service from the ATM, hence this is a Single Server Problem, and if another customer arrives, the customer is enqueued to the queue. When the current customer taking the service finishes the next customer is the front of the queue takes service.

This is a Single Server problem, with the Inter-Arrival-Time and the Service Time are generated Randomly, the total number of customers served is unknown before simulation, which can only be known after the simulation ends, which is determined by the total simulation time which is taken as input from the user before the simulation begins.

2.2 Implementation

ATM_SIM.java

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package atm_sim;

import java.util.ArrayList;
import java.util.Collections;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.OptionalDouble;
import java.util.Scanner;

/**
 *
 * @author shadowleaf
 */
public class ATM_SIM {



    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
```

```
        Scanner input = new Scanner(System.in);
        System.out.print("Enter the total time to simulate for (minutes) : ");
        Integer T = input.nextInt();

        // seed the random number generator with the current system time in milli-seconds
        Random rand = new Random(System.currentTimeMillis());

        // map to store the simulation table
        Map<String, List<Integer>> SIM_TAB = new HashMap<>();

        List<Integer> IAT = new ArrayList<>();
        List<Integer> ST = new ArrayList<>();

        // initialize the simulation table
        SIM_TAB.put("IAT", IAT);
        SIM_TAB.put("ST", ST);
        SIM_TAB.put("SS", new ArrayList<>(Collections.nCopies(1, 0)));
        SIM_TAB.put("AT", new ArrayList<>(Collections.nCopies(1, 0)));
        SIM_TAB.put("SE", new ArrayList<>(Collections.nCopies(1, 0)));
        SIM_TAB.put("WAIT", new ArrayList<>(Collections.nCopies(1, 0)));
        SIM_TAB.put("IDLE", new ArrayList<>(Collections.nCopies(1, 0)));

        // start the simulatioin
        Integer SYS_CLOCK = 0;
        SIM_TAB.get("IAT").add(rand.nextInt(4)+1);
        SIM_TAB.get("ST").add(rand.nextInt(6)+1);

        SIM_TAB.get("AT").set(0, 0);
        for (int i = 1; SYS_CLOCK <= T ; i++) {
            // IAT b/w 1-4 mins
            SIM_TAB.get("IAT").add(rand.nextInt(4)+1);
            // ST b/w 1-6 mins
            SIM_TAB.get("ST").add(rand.nextInt(6)+1);

            SIM_TAB.get("AT").add(SIM_TAB.get("AT").get(i - 1) + SIM_TAB.get("IAT").get(i)
);
            if (SIM_TAB.get("AT").get(i) >= SIM_TAB.get("SE").get(i - 1)) {
                SIM_TAB.get("SS").add(SIM_TAB.get("AT").get(i));
            } else {
                SIM_TAB.get("SS").add(SIM_TAB.get("SE").get(i - 1));
            }

            SIM_TAB.get("SE").add(SIM_TAB.get("SS").get(i) + SIM_TAB.get("ST").get(i));

            SIM_TAB.get("WAIT").add(SIM_TAB.get("SS").get(i) - SIM_TAB.get("AT").get(i));

            SIM_TAB.get("IDLE").add(SIM_TAB.get("AT").get(i) - SIM_TAB.get("SE").get(i-
1) >= 0 ? SIM_TAB.get("AT").get(i) - SIM_TAB.get("SE").get(i-1) : 0);
```

```
            // set the sys_time
            SYS_CLOCK = SIM_TAB.get("SE").get(i) + SIM_TAB.get("ST").get(i);

        }

        // Print the simulation table
        System.out.println("REQNO\tIAT\tAT\tSS\tSE\tST\tWAIT\tIDLE");
        for (int i = 1; i < SIM_TAB.get("IAT").size(); i++) {
            String out = i + "\t"
                    + SIM_TAB.get("IAT").get(i) + "\t"
                    + SIM_TAB.get("AT").get(i) + "\t"
                    + SIM_TAB.get("SS").get(i) + "\t"
                    + SIM_TAB.get("SE").get(i) + "\t"
                    + SIM_TAB.get("ST").get(i) + "\t"
                    + SIM_TAB.get("WAIT").get(i) + "\t"
                    + SIM_TAB.get("IDLE").get(i);
            System.out.println(out);
        }

        // Avg WAIT, Avg. Ser, Avg. IAT
        OptionalDouble avgWAIT = SIM_TAB.get("WAIT").stream().mapToDouble(a -
> a).average();
        OptionalDouble avgService = SIM_TAB.get("ST").stream().mapToDouble(a -
> a).average();
        OptionalDouble avgIAT = SIM_TAB.get("IAT").stream().mapToDouble(e -
> e).average();
        System.out.println("Total Requests Processed : " + (SIM_TAB.get("IAT").size()-1));
        System.out.println("Average WAIT : " + avgWAIT.getAsDouble());
        System.out.println("Averate Service Time : " + avgService.getAsDouble());
        System.out.println("Averate IAT : " + avgIAT.getAsDouble());
    }
```

2.3 Result and Analysis

OUTPUT for the ATM Simulation

```
run:
Enter the total time to simulate for (minutes): 720
REQNO   IAT     AT      SS      SE      ST      WAIT    IDLE
1       4       4       4       9       5       0       4
2       2       6       9       15      6       3       0
3       3       9       15      18      3       6       0
4       3       12      18      23      5       6       0
5       4       16      23      26      3       7       0
6       2       18      26      27      1       8       0
7       4       22      27      33      6       5       0
8       4       26      33      35      2       7       0
9       3       29      35      36      1       6       0
10      2       31      36      39      3       5       0
```

```
11      4       35      39      41      2       4       0
12      1       36      41      42      1       5       0
13      3       39      42      43      1       3       0
14      3       42      43      47      4       1       0
15      3       45      47      52      5       2       0
.
.
.
.
190     3       474     641     647     6       167     0
191     4       478     647     652     5       169     0
192     1       479     652     655     3       173     0
193     4       483     655     657     2       172     0
194     4       487     657     659     2       170     0
195     3       490     659     660     1       169     0
196     1       491     660     662     2       169     0
197     4       495     662     665     3       167     0
198     1       496     665     670     5       169     0
199     1       497     670     676     6       173     0
200     2       499     676     678     2       177     0
201     4       503     678     680     2       175     0
202     3       506     680     682     2       174     0
203     3       509     682     683     1       173     0
204     3       512     683     687     4       171     0
205     4       516     687     692     5       171     0
206     2       518     692     695     3       174     0
207     1       519     695     697     2       176     0
208     1       520     697     699     2       177     0
209     1       521     699     703     4       178     0
210     1       522     703     705     2       181     0
211     3       525     705     706     1       180     0
212     4       529     706     710     4       177     0
213     4       533     710     713     3       177     0
214     1       534     713     717     4       179     0

Total Requests Processed: 214
Average WAIT: 83.02325581395348 minutes
Average Service Time: 3.3209302325581396 minutes
Average IAT: 2.4976744186046513 minutes
BUILD SUCCESSFUL (total time: 2 seconds)
```

Since the Average Service Time is greater than the Average Arrival Time in general, our server is busy most of the time, but this also means that the requests have to wait for a considerable amount of time before getting serviced, from the statistics we can see that the average wait time is 83 minutes, which is much higher, hence to reduce this multi-servers must be used.

A Total of 214 requests were processed in 720 minutes, so an average of 3.3644 minutes of system time was given to each of the request and the server processed, 0.297 requests were processed by the server every minute.

Some Analysis from the Output

$$Average\ Arrival\ Rate = \lambda = \frac{1}{2.4976} = 0.4$$

$$Averate\ Service\ Rate = \mu = \frac{1}{3.3209} = 0.3$$

Algorithms Used:

```
// Generates the next pseudo-random number.
```

next(bits):

```
1. do {
2.     oldseed = seed.get()
3.     nextseed = (oldseed * MULTIPLIER + ADDEND) & MASK;
4. } while (!seed.compareAndSet(oldseed, nextseed))
5. return nextseed >>> (48-bits)
```

What this basically does is atomically update seed as

```
(seed * 0x5DEECE66DL + 0xBL) & ((1L << 48) - 1)
```

This is a Linear Congruential Generator as defined by D. H. Lehmer and described by Donald E. Knuth in The Art of Computer Programming, Volume 3: Seminumerical Algorithms, section 3.2.1.

```
// Returns a pseduo-random number distributed int value between 0 (inclusive) and
the specified value (exclusive)
```

nextInt(bound):

```
1. if (bound & -bound) == bound // bound is a power of 2
2.     return (bound * next(31)) >> 31
3. else
4.     do {
5.           bits = next(31)
6.           val = bits % bound
7.     } while (bits - val + bound - 1 < 0)
8. return val
```

The hedge "approximately" is used in the foregoing description only because the next method is only approximately an unbiased source of independently chosen bits. If it were a perfect source of randomly chosen bits, then the algorithm shown would choose int values from the stated range with perfect uniformity.

The algorithm is slightly tricky. It rejects values that would result in an uneven distribution (due to the fact that 2^31 is not divisible by n). The probability of a value being rejected depends on n. The worst case is n=2^30+1, for which the probability of a reject is 1/2, and the expected number of iterations before the loop terminates is 2.

The algorithm treats the case where n is a power of two specially: it returns the correct number of high-order bits from the underlying pseudo-random number generator. In the absence of special treatment, the correct number of low-order bits would be returned. Linear congruential pseudo-random number generators such as the one implemented by this class are known to have short periods in the sequence of values of their low-order bits. Thus, this special case greatly increases the length of the sequence of values returned by successive calls to this method if n is a small power of two.