Notes on Stochastic Simulations

MATLAB Codes for
Simulation of Stochastic Processes
2021

by
Aanchit Nayak

Preface

This notebook started in August 2021 after a devastating COVID-19 wave hit India. The goal was to use online notes made available by Dr. (Prof.) Karl Sigman and write comprehensive MATLAB code while staying home and recovering from debilitating pneumonia. A small success has been achieved, but there are miles to go. This notebook is a compilation of notes, codes, and results that will continue growing as time passes. This work does not shy away from going across multiple domains and tries to establish the versatility of mathematical modeling itself. However, it is a long way from becoming a complete resource.

The basic structure of each section is simple. Basic theory, proofs, and modeling information are provided wherever possible. The document first discusses random number generation techniques, Poisson processes, Markov Chains, and Brownian Motion. Later, some applications are provided viz-a-viz queueing models, inventory processes, and insurance credit risk models. MATLAB codes have been presented for each of these sections, and basic algorithms are provided. Each code tries to implement the algorithm, generates outputs including visualizations and estimates, and presents functions to generalize the program wherever possible.

Future work directions include writing rigorous theory based on proofs rooted in real analysis, deeper explorations in variations in the fundamental stochastic properties of some models, and using data for some of the applications. This document can be viewed as a journal of MATLAB programs in its current state.

Any and all feedback is welcome and readers can reach out to me on GitHub and LinkedIn.

Contents

1 Random Number Generators

One primary assumption, which holds across this document, is that the computer can, on demand, generate i.i.d. uniformly distributed random numbers. We use this to our advantage and generate other random variables using the uniform distribution. Based on this assumption, two different methods are presented which will be used to generate other random variables. They are:

1. Inverse Transform Method (ITM)
2. Acceptance-Rejection Method (ARM)

1.1 Inverse Transform Method (ITM)

Let $F(x)$, $x \in \mathbb{R}$, denote any cumulative distribution function. It can be noted that $F : \mathbb{R} \to [0, 1]$ is a non-negative and monotone function that is continuous from right and has left hand limits. Also, $F(\infty) = 1$ and $F(-\infty) = 0$. Our objective, in this method, is to generate a random variable X distributed as F such that $P(X \leq x) = F(x)$, $x \in \mathbb{R}$. As the name suggests ITM alludes towards the inverse of a function, which is the CDF in this case. We may define the generalized inverse of F as:

$$F^{-1}(y) = \min\{x : F(x) \geq y\}, y \in [0, 1] \tag{1}$$

Clearly, since F is continuous, F is also invertible. Because of the invertibility, we may directly state that $F(F^{-1}(y)) = y$. And, in general, it holds that $F^{-1}(F(x)) \leq x$ and $F(F^{-1}(y)) \geq y$. The inverse of F is a monotone function which we will use to simulate random variables.

The proof for the working of ITM is as follows:

Let F be a CDF whose inverse, F^{-1}, exists and is defined as (1). Now, define $X = F^{-1}(U)$ where U is a continuous uniformly distributed RV on $(0, 1)$. Then, we wish to prove that X is distributed as F. In other words, it is sufficient to show that $P(F^{-1}(U) \leq x) = F(x) \forall x \in \mathbb{R}$.

To do this, assume that F is continuous. Also, we need to essentially show an equality of events for the sets$\{F^{-1}(U) \leq x\}$ and $\{U \leq F(x)\}$. Let $a = F(x)$ in $P(U \leq a) = a$. This would give us $P(F^{-1}(U) \leq x) = P(U \leq F(x)) = F(x)$.

To this end, $F(F^{-1}(y) = y$ and if $F^{-1}(U) \leq x$ then $U = F(F^{-1}(U)) \leq F(x)$ or $U \leq F(x)$. And hence, $F^{-1}(F(x)) = x$ and so if $U \leq F(x)$, then $F^{-1}(U) \leq x$. This concludes the equality of the two events. The proof for this in the discrete domain is skipped.

1.1.1 Simulating Distributions using ITM

In this code, the simulation of Exponential, Bernoulli, Binomial, and Poisson Distributions is carried out. For saving space in the MATLAB workspace and keeping the code and related results modular, structures are used. Relevant algorithms are written as comments in the code wherever required.

```
%% Inverse Transform Method

clc
close all
clear all
% The inverse transform method states that if the cumulative distribution
% of a random variable is known, then the inverse of said cumulative
% distribution can, in tandem with the uniform random variable, be used to
% generate the distribution of X, whose CDF was described.


% This code creates structures to save generated distributions and their
% data

%% Simulating the Exponential Distribtuion

% Algorithm:
%   1 - Generate U~unif(0,1)
%   2 - Set X = (-1/Lm)*ln(U)

% Let N be the length of the random vector which is exponentially
% distributed.
expo.N = 1000;
% Let the rate of exponential distribution be Lm.
expo.Lm = 1.5;
for i = 1:expo.N
   U = rand();
   expo.X(i) = (-1/expo.Lm)*log(U);
end
clear i N Lm U X

figure
histogram(expo.X,'Normalization','probability')
grid on
title('ITM Generated Exponential Distribution')
xlabel('$x$','Interpreter','latex')
ylabel('$P(X = x)$','Interpreter','latex')
legend('\lambda = 1.5')

%% Simulating the Bernoulli(p) and Binomial(n,p) Distributions

% For the Bernoulli Distribution with success probability parameter 'p',
% we assume X follows that distribution. Hence, P(X=0) = 1-p and P(X=1)=p.

% Algorithm:
% 1 - Generate U~unif(0,1)
% 2 - Set X = 0 if U <= 1-p, X = 1 otherwise

% Let N be the length of the random vector
bern.N = 1000;
bern.p = 0.5;

for i = 1:bern.N
   U = rand();
   if U<=bern.p
       bern.X(i) = 0;
```

```
   else
       bern.X(i) = 1;
   end
end
clear i U

figure
histogram(bern.X,'Normalization','probability')
grid on
title('ITM Generated Bernoulli Distribution')
xlabel('$x$','Interpreter','latex')
ylabel('$P(X = x)$','Interpreter','latex')
legend('p = 0.5')
% For the Binomial Distribution, one can easily observe that a binomial
% distribution is the sum of n i.i.d. Bernoulli(p) RVs.
bin.N = bern.N;
bin.n = 500;
U = rand(bin.N,bin.n);
bin.p = bern.p;
for i = 1:bin.n
    for j = 1:bin.N
        if U(j,i) <= 1-bin.p
            bin.Y(j,i) = 0;
        else
            bin.Y(j,i) = 1;
        end
    end
        bin.X(i) = sum(bin.Y(:,i));
end

figure
histogram(bin.X,'Normalization','probability')
grid on
title('ITM Generated Binomial Distribution')
xlabel('$x$','Interpreter','latex')
ylabel('$P(X = x)$','Interpreter','latex')
legend('N = 500, p = 0.5')
clear i j U
%% Simulating the Poisson Distribution

% Algorithm is as follows:

pois.Lm = 3.5;
for i = 1:1000
    pois.n = 1;
    pois.a = 1;
    while pois.a>=exp(-pois.Lm)
        U = rand();
        pois.a = pois.a.*U;
        pois.n = pois.n + 1;
    end
    pois.X(i) = pois.n - 1;
end
clear U i
figure
histogram(pois.X,'Normalization','probability')
grid on
```

```
title('ITM Generated Poisson Distribution')
xlabel('$x$','Interpreter','latex')
ylabel('$P(X = x)$','Interpreter','latex')
legend('\lambda = 3.5')
```

The output of the above code is extracted as images from the respective structures. The outputs can be found in fig 1.

The evaluation of the inverse of these CDFs is an easy exercise. The exponential distribution given by the parameter λ is has CDF of the form $1 - e^{-\lambda x}$. In order to simulate the exponential distribution, we can trivially solve $y = 1 - e^{-\lambda x}$ for x. This is a luxury not often available to the investigator, which also constitutes a drawback of the ITM Algorithm.

On solving for x, we get $x = -\frac{ln(1-y)}{\lambda}$. Remember that y is simply U, a uniform random variable. Since U is uniformly distributed, we can simply ignore the effect of $1 - U$ since that distribution would be exactly the same as the distribution of U. Hence, a closed form solution to simulate a exponential distribution is using the following:

$$x = -\frac{ln(U)}{\lambda} \tag{2}$$

The simulation of the Bernoulli and the Binomial RVs is done by assigning $P(X = 0) = 1 - p$ and $P(X = 1) = p$ for some $p \in (0, 1)$. Simply, assigning $X = 0$ for $U \leq 1 - p$ and $X = 1$ for $U > 1 - p$. This gives us our Bernoulli Distribution. Repeating this exercise for N i.i.d. U_i $(1 \leq i \leq n)$, assigning $Y_i = 1$ and $Y_i = 0$ based on the same principles as X in Bernoulli will give us a set of Bernoulli distributed Y_i. Setting $X = \sum_{i=1}^{N} Y_i$ yields a Binomially distributed RV.

Finally, the simulation of Poisson Distributed RV is done using a novel method instead of simply taking the inverse of the CDF of a Poisson RV. This algorithm takes advantage of the properties of the Poisson Process with some rate λ. Let $\{N(t) : t \geq 0\}$ be a counting process with rate λ. Thus if we can simulate $N(1)$, then we can set some $X = N(1)$. Let $Y = N(1) + 1$, and let some $t_n = X_1 + ... + X_n$ denote the n^{th} point of the Poisson Process; the X_i are i.i.d. with an exponential distribution with rate λ. Hence, $Y = \min\{n \geq 1 : X_1 + ... + X_n\}$. Since we established that $X = -\frac{U}{\lambda}$, we can simply reach the following:

$$Y = \min\{n \geq 1 : U_1 ... U_n < e^{-\lambda}\} \tag{3}$$

Y is hence a Poisson distributed RV.

1.2 Acceptance-Rejection Method (ARM)

It has been established it is required to know the closed functional form of the CDF of a distribution of some RV to use ITM. The ARM presents an alternative to ITM. ARM works on a clever little trick. Say we wish to simulate a RV with a CDF F and a PDF f. Our aim is to find an alternative distribution G with density g for which we already have an efficient algorithm. Also, g must be

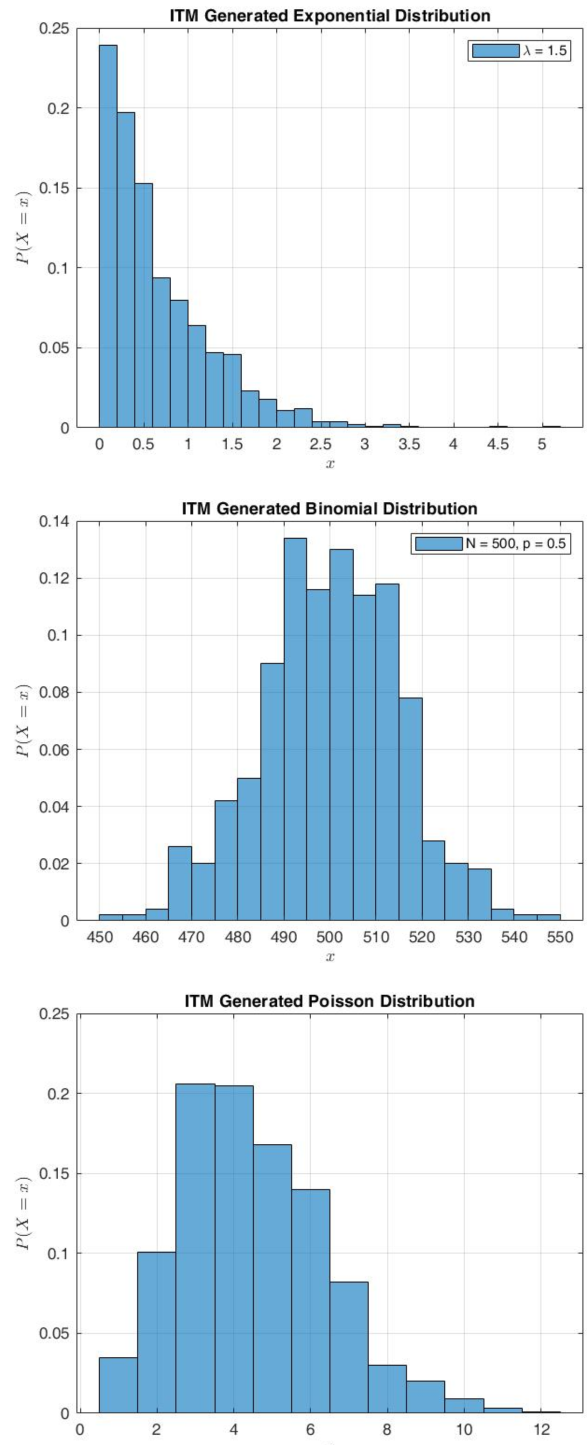

Figure 1: Output of ITM-based Random Number Generators

"closer" to f, or, the ratio $\frac{f(x)}{g(x)}$ is bounded by some constant c such that $c > 0$. Ideally, we want c to be as close to one as possible.

In general, we want to run the following algorithm:

1. Generate some Y distributed as G.
2. Generate U
3. If
$$U \leq \frac{f(Y)}{cg(Y)}$$
then we set $X = Y$, else we go back to step 1.

There are many things to note here. First we prove that this algorithm works and helps us generate X distributed as F. To prove this, we must show that the conditional distribution of Y given that $U \leq \frac{f(Y)}{cg(Y)}$ is F. Essentially, we intend to show that $P(Y \leq y | U \leq \frac{f(Y)}{cg(Y)}) = F(y)$. We can simply prove that $P(Y = y | U \leq \frac{f(Y)}{cg(Y)}) = f(y)$. Notice that the generation of Y and U is independent of each other. We use this to our advantage using the result $P(A|B) = P(AB)/P(B)$. However, to do this, we must know the probability $P(U \leq \frac{f(Y)}{cg(Y)})$.

Notice that the algorithm presented above has multiple RVs embedded in it. The distributions of f are g explicitly visible. But, clearly, $\frac{f(Y)}{cg(Y)}$ must also be a RV. Moreover, the number of times, say N, the iterations on steps 1 and 2 successfully generates the required RV is also a RV! The latter is a Geometrically distributed RV where the probability of success p, say, is defined as $p = P(U \leq \frac{f(Y)}{cg(Y)})$, and the mass function is $P(N = n) = p(1-p)^{n-1} \forall n \geq 1$. It is known that the average of this mass function is $E(N) = 1/p$.

We wish to evaluate p to return to the proof of the working of the algorithm. Notice that $P(U \leq \frac{f(Y)}{cg(Y)} | Y = y) = \frac{f(Y)}{cg(Y)}$, and thus unconditioning and recalling that Y has density $g(y)$ yields

$$p = \int_{-\infty}^{\infty} \frac{f(Y)}{cg(Y)} g(y) dy$$
$$p = \frac{1}{c}$$

Interestingly, we may say that *the expected number of iterations of the algorithm required until an X is succesfully generated is exactly the bounding constant $c = \sup_x \{f(x)/g(x)\}$.*

Now that we know that $p = 1/c$, we may return to the proof of the algorithm.

Recall that
$$P(Y = y | U \leq \frac{f(Y)}{cg(Y)}) = f(y)$$
must be established. Using the simple probability rule stating $P(A|B) = P(AB)/P(B)$, we may write:
$$P(Y = y | U \leq \frac{f(Y)}{cg(Y)}) = cP(Y = y, U \leq \frac{f(Y)}{cg(Y)})$$

Since Y and U are independent the joint distribution decomposes into a product. We now have:

$$P(Y = y|U \leq \frac{f(Y)}{cg(Y)}) = cg(y)P(U \leq \frac{f(Y)}{cg(Y)})$$

$$P(Y = y|U \leq \frac{f(Y)}{cg(Y)}) = cg(y)\frac{f(Y)}{cg(Y)}$$

$$P(Y = y|U \leq \frac{f(Y)}{cg(Y)}) = f(y)$$

Hence, proved.

1.2.1 Simulating the Normal Distribution

The Normal distribution with mean μ and variance σ^2 can be written as a linear combination as $X = \mu + \sigma Z$ where $Z \sim N(0, 1)$. Also, owing to its symmetry, $|Z|$ can be used to simulate Z using another independent RV for the sign, say S.

$|Z|$ is non-negative with density

$$f(x) = \frac{2}{\sqrt{2\pi}}e^{-\frac{x^2}{2}}, x \geq 0$$

ARM requires an alternative distribution. For this, we chose the exponential distribution with rate 1. Hence, $g(x) = e^{-x}$. Obviously, exponential distribution is easy to generate using ITM and is close enough to the Normal distribution. Hence the ratio of $f(x)/g(x)$ is the function

$$h(x) = \sqrt{\frac{2}{\pi}}e^{x-\frac{x^2}{2}}$$

The bounding constant c occurs when we maximize x. This is a trivial exercise. Letting $h'(x) = 0$, we realize that $h(x)$ hits its maximum at $x = 1$. Putting $x = 1$ in the original form of h, we get $c = \sqrt{2e/\pi} \approx 1.32$. Also, $f(y)/cg(y) = e^{-\frac{(y-1)^2}{2}}$. Now, we may lay down the algorithm for generating the Normal Distribution.

First, we generate two independent exponentials at rate 1, say $Y_1 = -ln(U_1)$ and $Y_2 = -ln(U_2)$. We now ask if $Y_2 \geq (Y_1 - 1)^2/2$, then set $|Z| = Y_1$, else generate the two exponentials again. If the above is true, generate U, then set $Z = |Z|$ if $U \leq 0.5$, set $Z = -|Z|$ if $U > 0.5$. Z is our standard normal distribution which can be used to generate any general normal distribution.

The code for this as follows:

```
%% Using ARM - Acceptance Rejection Method for simulating RVs

clc
clear all
close all
```

```matlab

%% Simulating the Normal Distribution

% We desire to generate X~N(mu,sigma). We know, X = mu*Z + sigma
% where Z~N(0,1). It suffices to find an algorithm for generating Z ~
% N(0,1).

% for the function close to normal,we choose the exponential

% Algorithm is as follows:
% 1 - Generate two independent exponentials Y1 and Y2 with rate 1
% 2 - if Y2 <= (Y1 - 1)^2/2, set modZ = Y1 else, go back to 1
% 3 - Generate U. Set Z = modZ if U<= 0.5. Set Z = -modZ if U>0.5
iter = 0;

DistLength = 10000;
for i = 1:DistLength
    Y1(i) = 0;
    Y2(i) = 0;
    while Y2(i) < (Y1(i) - 1)^2/2
        Y1(i) = -log(rand());
        Y2(i) = -log(rand());
        iter = iter + 1;
    end
    epoch(i) = iter;
    modZ(i) = Y1(i);
    if rand() <= 0.5
        Z(i) = modZ(i);
    else
        Z(i) = - modZ(i);
    end
end

tempdist = makedist('Normal',0,1);

figure
% subplot(211)
histogram(Z,'Normalization','pdf','DisplayName','Generated Data')
hold on
plot(-10:0.01:10,pdf(tempdist,-10:0.01:10),'DisplayName','In-Built MATLAB pdf','
    LineWidth',2)
grid on
title('Standard Normal Distribution','Interpreter','latex')
xlabel('$x$','Interpreter','latex')
ylabel('$P(X = x)$','Interpreter','latex')
hl = legend('show');
set(hl, 'Interpreter','latex')

% subplot(212)
% histogram(epoch,'Normalization','pdf','DisplayName','Geometric Distribution')
% grid on
% title('Distribution of Epochs Required to Generate a Gaussian','Interpreter','latex
    ')
% xlabel('$x$','Interpreter','latex')
% ylabel('$P(X=x)$','Interpreter','latex')
% hl = legend('show');
% set(hl, 'Interpreter','latex')
```

```

clear h1 tempdist modZ Y1 Y2 i iter
```

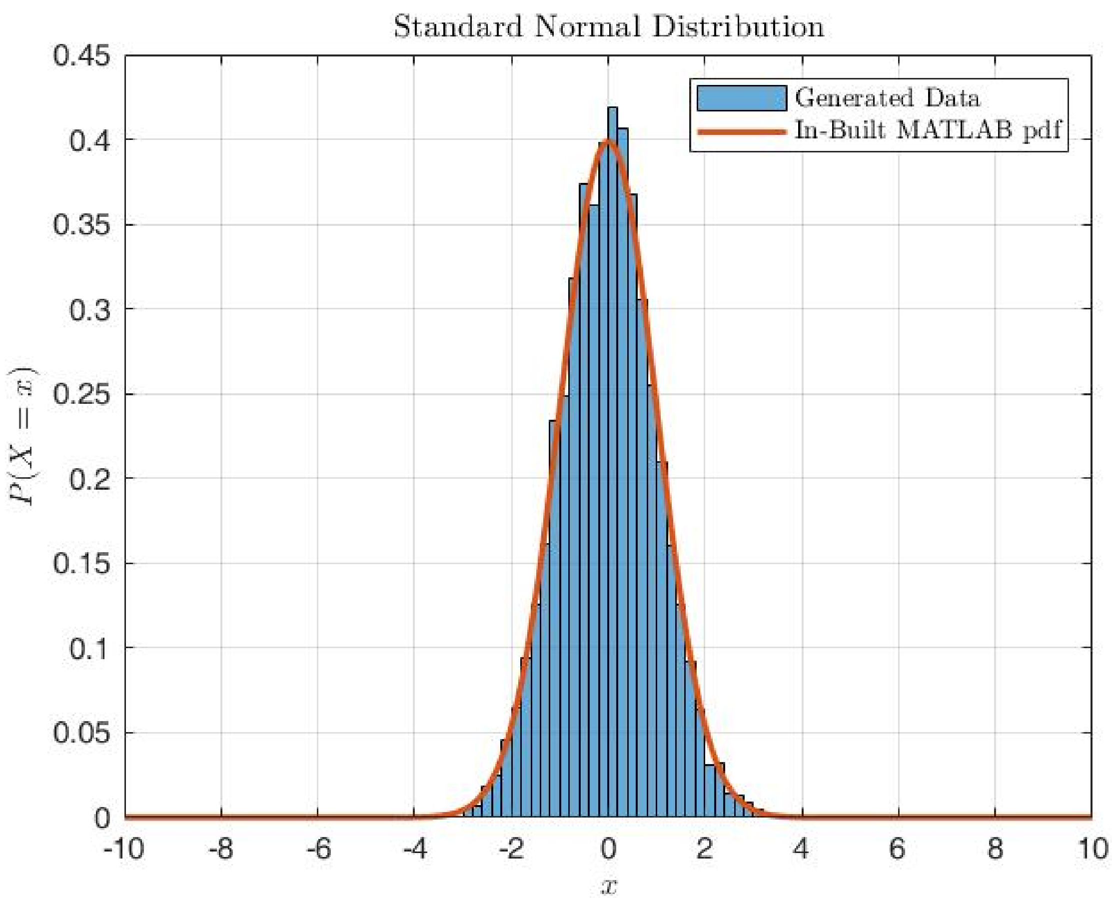

Figure 2: Acceptance-Rejection Method for generating Normal Distribution $Z \sim N(0, 1)$

2 The Poisson Process

The Poisson Process finds a wide array of applications in many applications. In order to build further on this, we define the *Point Process.* A simple point process $\psi = \{t_n : n \geq 1\}$ is a sequence of strictly increasing points $0 < t_1 < t_2 ...$ with $t_n \to \infty$ as $n \to \infty$. With $N(0) = 0$, we let $N(t)$ denote the number of points that fall in the interval $(0, t]$; $N(t) = \max\{n : t_n \leq t\}$. This process is called a **counting process** for ψ. If the t_n are random variables then ψ is called a **random point process**. Usually, and in this notebook, $t_0 = 0$ and $X_n = t_n - t_{n-1}$ is called the n^{th} interarrival time. In general, t is a time while t_n is the n^{th} arrival time. The process $\{t_n\}$ is often used to model the arrival of customers, phone calls, etc to a system. Note that the use of the word *simple* alludes to the fact that we are allowing only one arrival at a time.

Now, we define a renewal process. A random point process ψ for which the interarrival times X_n form an i.i.d. sequence is called a renewal process. Leveraging this definition, we move to the definintion of a Poisson process.

2.1 Defining and Simulating a Poisson Process

A Poisson Process atrate λ is a renewal point process in which the interarrival times are exponentially distributed with rate λ.

Using this information, an algorithm for simulation of a Poisson Process is as follows:

1. Set $t = 0, N = 0$
2. Generate U
3. Modify $t = t + (-1/\lambda)\ln(U)$. If t exceeds the intended length of the simulation, say T, then stop. Else continue.
4. Set $N = N + 1$ and set $t_N = t$.
5. Go back to step - 2.

This algorithm helps us simulate the arrival times and store them in an array called t_N and allows us to count with the variable N. Notice that the use of ITM extends its application here making it easy for us to simulate a Poisson Process.

2.1.1 Some Properties of a Poisson Process

1. For each fixed $t > 0$, the distribution of $N(t)$ is Poisson Distributed with rate λt. Hence, $E(N(t)) = \lambda t$ and the variance $Var(N(t)) = \lambda t \forall t \geq 0$. In essence, for any $s > 0$ the increment $N(s + t) - N(s)$ are Poisson Distributed and the distribution only depends on the length of the increment. Such increments are called **Stationary Increments**.

2. Aforementioned increments, for the Poisson Process, are also independent of each other. Hence, non-overlapping increments are independent random variables.

3. It turns out that the Poisson Process is completely characterized by stationary and independent increments. Hence, *if you have a point process ψ with stationary and independent increments, then ψ is a Poisson Process.*

4. One can also view the Poisson Process at rate λ as performing independent Bernoulli trials with probability of success given by $p = \lambda dt$ in each infinitesimal time interval of length dt.

2.1.2 MATLAB Code

```
%% Simulation of a Simple Poisson Process
clc
clear all
close all

T = 2000; %Length of the Simulation
t = 0;
N(1) = 0;
i = 2;
Lm = 1;
tN(1) = 0;
% Loop for Simulation
while t<T
    U = rand();
    t = t + (-1/Lm)*log(U);
    if t>T
        break
    end
    N(i) = N(i-1) + 1;
    tN(i) = t;
    i = i +1;
end

%% Plotting the Inter-arrival time distribution
X(1) = tN(1);
for i = 2:length(tN)
   X(i) = tN(i) - tN(i-1) ;
end

tempdist = makedist('Exponential',Lm);
n = length(X) - 1;
figure
% subplot(211)
histogram(X,'Normalization','pdf','DisplayName','Inter-Arrival Times')
hold on
plot(0:n,pdf(tempdist,0:n),'DisplayName','True Exponential PDF','LineWidth',2)
xlim([0 8])
grid on
title('Distribution of Inter-Arrival Times with $\lambda = 1$','Interpreter','latex')
xlabel('$x$','Interpreter','latex')
ylabel('$P(X = x)$','Interpreter','latex')
hl = legend('show');
```

```
set(hl, 'Interpreter','latex')

%% Plotting the Counting Process
figure
stairs(tN,N, 'LineWidth',1.75, 'DisplayName','Sample Path')
hold on
plot(0:15,0:15,'LineWidth',2,'DisplayName','$x = y$')
xlim([0 15])
grid on
title('Stair Step Plot of a Sample Poisson Process with $\lambda = 1$', 'Interpreter',
    'latex')
h2 = legend('show');
set(h2,'Interpreter','latex')
xlabel('t','Interpreter','latex')
ylabel('$N(t)$','Interpreter','latex')

%% Change in Sample Path Trajectory as Lm increases
%Case - 1: Lm = 1
LmTst = [1 2 0.5];
figure
subplot(131)
plot(0:T/1e+2,0:T/1e+2,'LineWidth',3,'DisplayName','$x = y$')
hold on
h2 = legend('show');
set(h2,'Interpreter','latex','AutoUpdate','Off')
xlabel('t','Interpreter','latex')
ylabel('$N(t)$','Interpreter','latex')
for i = 1:100
    [tN,N] = poisson_sim(LmTst(1),T);
    stairs(tN,N)
    hold on
end
xlim([0 T/1e+2])
grid on
title('$\lambda = 1$', 'Interpreter','latex')

subplot(132)
plot(0:T/1e+2,0:T/1e+2,'LineWidth',3,'DisplayName','$x = y$')
hold on
h2 = legend('show');
set(h2,'Interpreter','latex','AutoUpdate','Off')
xlabel('t','Interpreter','latex')
ylabel('$N(t)$','Interpreter','latex')
for i = 1:100
    [tN,N] = poisson_sim(LmTst(2),T);
    stairs(tN,N)
    hold on
end
xlim([0 T/1e+2])
grid on
title('$\lambda = 2$', 'Interpreter','latex')


subplot(133)
plot(0:T/1e+2,0:T/1e+2,'LineWidth',3,'DisplayName','$x = y$')
hold on
h2 = legend('show');
```

```
set(h2,'Interpreter','latex','AutoUpdate','Off')
xlabel('t','Interpreter','latex')
ylabel('$N(t)$','Interpreter','latex')
for i = 1:100
    [tN,N] = poisson_sim(LmTst(3),T);
    stairs(tN,N)
    hold on
end
xlim([0 T/1e+2])
grid on
title('$\lambda = \frac{1}{2}$', 'Interpreter','latex')
suptitle('Stair Step Plot of 100 Sample Poisson Processes with various \lambda');


%% Function to simulate a Poisson Process
function [tN,N] = poisson_sim(Lm,T)
    t = 0;
    N(1) = 0;
    i = 2;
    tN(1) = 0;
    while t<T
        U = rand();
        t = t + (-1/Lm)*log(U);
        N(i) = N(i-1) + 1;
        tN(i) = t;
        i = i +1;
    end
end
```

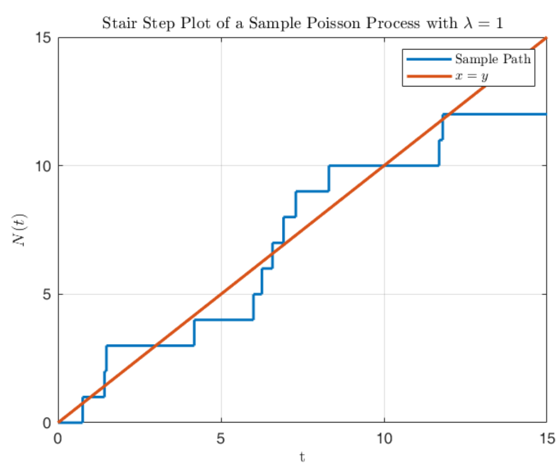

Figure 3: Here, a single trajectory of a Poisson Process is shown, whose rate is $\lambda = 1$.

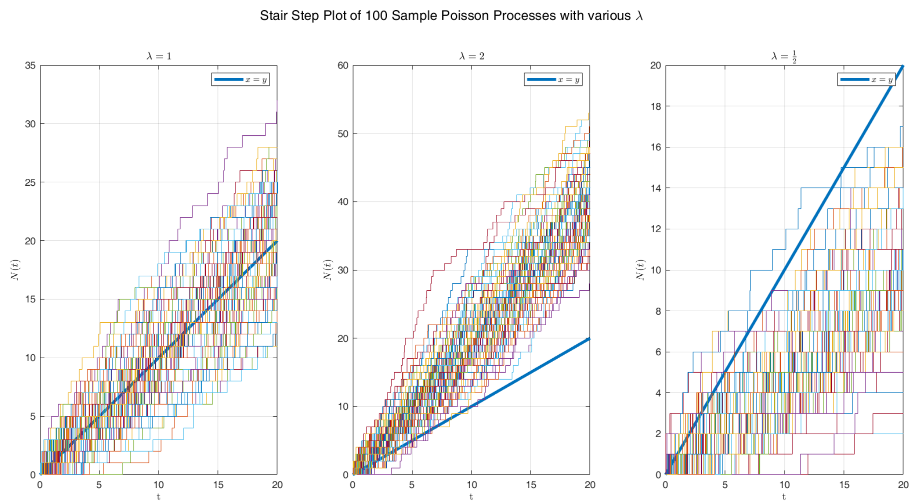

Figure 4: This plot shows how the sample paths shift with changes in λ

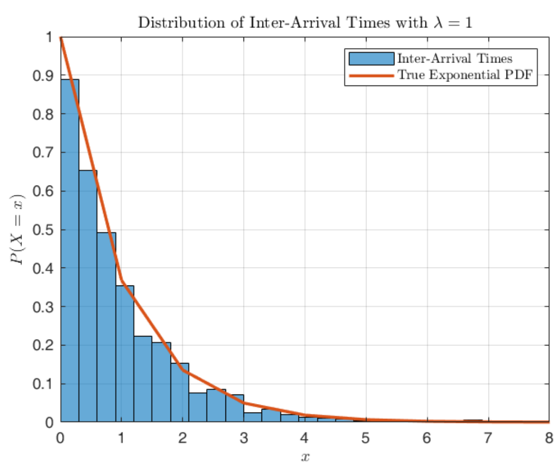

Figure 5: Distribution of the inter-arrival times is Exponential with rate $\lambda = 1$

2.2 Partitioning a Poisson Process

In this section we try to simulate a Poisson Process which can be separated based on another distribution. Here, we will look at $X \sim \text{Pois}(\lambda)$ where each X, upon arrival (say) is separated into two groups each with probability p and $1-p$. Hence, for some $X = x$, then the number x to be type-1 is $\text{Bin}(x, p)$ and type-2 as $\text{Bin}(x, 1-p)$. Let each type be denoted as X_1 and X_2 such that $X = X_1 + X_2$. If we do this, then each Poisson Process X_i is an independent Poisson Process with rate $p\lambda$ or $(1-p)\lambda$.

This can be easily shown by reducing the joint probability $P(X_1 = k, X_2 = m)$ like so:

$$P(X_1 = k, X = k+m) = P(X_1 = k|X = k+m)P(X = k+m)$$

But given $X = k+m$ and $X_1 \sim \text{Bin}(k+m, p)$,

$$P(X_1 = k|X = k+m)P(X = k+m) = \frac{(k+m)!}{k!m!}p^k(1-p)^m e^{\lambda}\frac{\lambda^{k+m}}{(k+m)!}$$

The above trivially decomposes to

$$P(X_1 = k, X_2 = m) = e^{-p\lambda}\frac{(p\lambda)^k}{k!}e^{-(1-p)\lambda}\frac{((1-p)\lambda)^m}{m!} \tag{4}$$

To simulate this behaviour, we have the following algorithm:

1. Initiate $t = 0, t1 = 0, N1 = 0, N2 = 0$ and set $\lambda = \lambda_1 + \lambda_2$ and $p = \lambda_1/\lambda$.
2. Generate U.
3. $t = t + (1 - /Lm)\ln(U)$. If $t > T$, then stop. Else, continue.
4. Generate U. If $U \leq p$, then set $N1 = N1 + 1$ and set $t_{N2} = t$; otherwise set $N2 = N2 + 1$ and set $t_{N2} = t$.
5. Go back to 2.

This can also be extended to handle more than two independent partitions. Using this, we can generate two different Poisson Processes using one.

2.2.1 MATLAB Code

```
%% Partitioning a Poisson Process
% Here we assume that a Poisson Process can be divided into two types of
% states both of which can occur with probability p and 1-p. We would like
% to simulate such a proces.
clc
```

```matlab
clear all
close all
% Initiating the values
T = 50;
t = 0;
t1 = 0;
t2 = 0;
N1(1) = 0;
N2(1) = 0;
Lm1 = 1;
Lm2 = 0.5;
Lm = Lm1 + Lm2;
p = Lm1/Lm;
i = 2;
j = 2;
tN(1) = 0;
tN(1) = 0;
%Loop for Simulation
while t<T
    U = rand();
    t = t + (-1/Lm)*log(U);
    if rand()<=p
        N1(i) = N1(i-1) + 1;
        tN1(i) = t;
        i = i + 1;
    else
        N2(j) = N2(j-1) + 1;
        tN2(j) = t;
        j = j + 1;
    end
end
%Plotting One Sample Path Each
stairs(tN1,N1,'color','red','LineWidth',1.5);
hold on
stairs(tN2,N2,'color','green','LineWidth',1.5);
hold on
plot(0:1:T,0:1:T,'LineWidth',2,'color','blue')
title("Sample Path for Partitioned Poisson processes with $\lambda_1$ and $\lambda_2$", 'Interpreter','latex');
grid on
legend('PP(\lambda_1)','PP(\lambda_2)');
xlim([0 25])

%% Plotting Multiple Sample Paths
figure
T = 1000;
plot(0:T/1e+2,0:T/1e+2,'LineWidth',3,'DisplayName','$x = y$','color','blue')
hold on
stairs(tN1,N1,'DisplayName','$\lambda = 1$','color','red');
hold on
stairs(tN2,N2,'DisplayName','$\lambda = \frac{1}{2}$','color','green');
hold on
h2 = legend('show');
set(h2,'Interpreter','latex','AutoUpdate','Off')
xlabel('t','Interpreter','latex')
ylabel('$N_1(t)$, $N_2(t)$','Interpreter','latex')
for i = 1:100
```

```
    [tN1,N1,tN2,N2] = partPoisson(T, Lm1, Lm2);
    stairs(tN1,N1,'color','red')
    hold on
    stairs(tN2, N2,'color','green')
    hold on
end
plot(0:T/1e+2,0:T/1e+2,'LineWidth',3,'DisplayName','$x = y$','color','blue')
hold on
xlim([0 T/1e+2])
grid on
title('Sample Paths for Partitioned Poisson processes with $\lambda_1$ and $\lambda_2$
    ', 'Interpreter','latex')
% suptitle('Stair Step Plot of 100 Sample Poisson Processes with various \lambda');



%% Function for 2-partition Poisson Process
function [N1,tN1,N2,tN2] = partPoisson(T, Lm1, Lm2)
    t = 0;
    t1 = 0;
    t2 = 0;
    N1(1) = 0;
    N2(1) = 0;
    Lm = Lm1 + Lm2;
    p = Lm1/Lm;
    i = 2;
    j = 2;
    tN(1) = 0;
    tN(1) = 0;
    while t<T
    U = rand();
    t = t + (-1/Lm)*log(U);
        if rand()<=p
            N1(i) = N1(i-1) + 1;
            tN1(i) = t;
            i = i + 1;
        else
            N2(j) = N2(j-1) + 1;
            tN2(j) = t;
            j = j + 1;
        end
    end
end
```

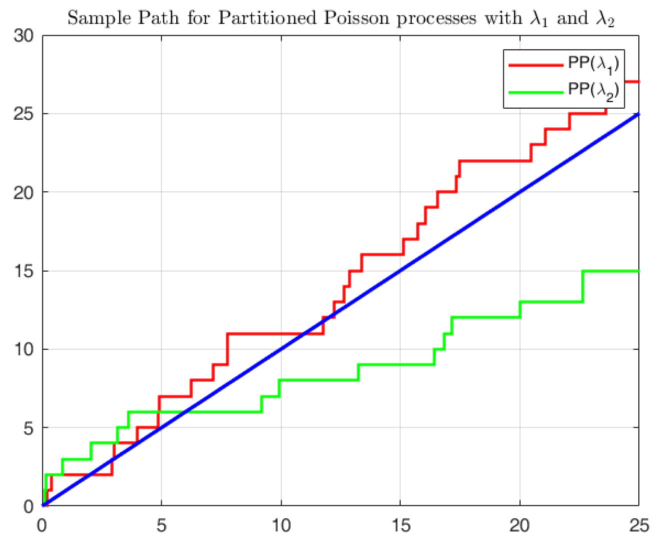

Figure 6: Sample Path for a Partitioned Poisson process

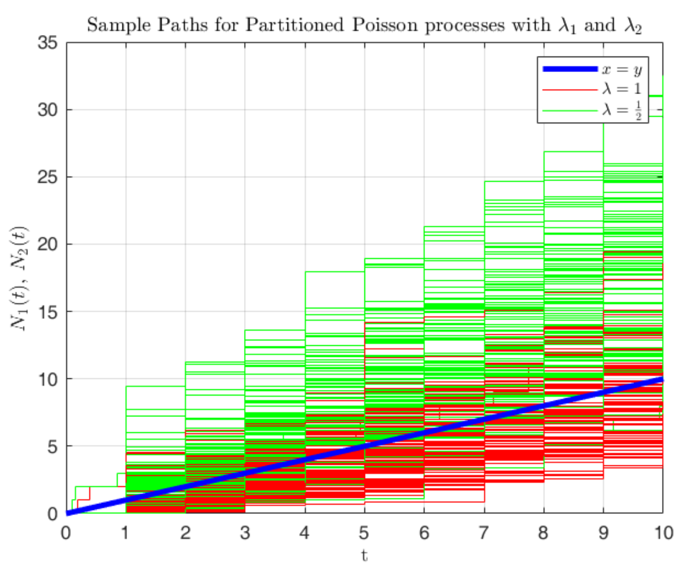

Figure 7: Multiple Sample Paths for Partitioned Poisson Process

2.3 Non-Stationary Poisson Processes

To extend the applications of Poisson processes, we introduce the notion of time-varying $\lambda(t)$ in a Poisson Process. This leads us to **Non-Stationary Poisson Processes**. For a given rate $\lambda(t)$, the expected number of arrivals by time t is given by:

$$m(t) = E(N(t)) = \int_0^t \lambda(s)ds \tag{5}$$

The general function $\lambda(t)$ is called the **Intensity** of the Poisson Process. The definition of a Non-Stationary Process is characterized by the following two conditions:

1. For each $t > 0$ the counting random variable N is Poisson Distributed with mean $m(t) = E(N(t)) = \int_0^t \lambda(s)ds$. More generally, we may state that the increment $N(t+h) - N(t)$ for $h > 0$ is Poisson Distributed with mean $m(t)$.
2. $\{N(t)\}$ has independent increments.

These points characterize (and hence, define) a Non-Stationary Poisson Distribution. In order to simulate these processes we assume the existence of λ^* such that

$$\lambda(t) \leq \lambda^*, t \geq 0$$

Practically it is important to use the smallest possible upper bound. Using this λ^* we use an algorithm called **Thinning** to simulate a Non-Stationary Poisson Process. In essence this algorithm asks the user to simulate a stationary Poisson Process with rate λ^* whose arrival times are denoted by $\{v_n\}$. When sampling these inter-arrival times, we know that the rate λ^* is larger than the intended rate $\lambda(t)$. So, we conduct a Bernoulli trial with a probability of success given by $\frac{\lambda(v_n)}{\lambda^*}$. Based on this trial, we decide to keep the value or move on. If $\{t_n\}$ is the sequence of accepted times, then this $\{t_n\}$ is our desired Non-Stationary Poisson Process. The algorithm is as follows:

1. Initiate $t = 0, N = 0$.
2. Generate a U.
3. $t = t + (-1/\lambda)\ln(U)$. If $t > T$, then stop.
4. Generate a U
5. If $U \leq \lambda(t)/\lambda^*$, then set $N = N + 1$ and set $t_N = t$.
6. Go back to 2.

2.3.1 MATLAB Code

```
%% Non-Stationary Poisson Process
% This is a case when Lm is a function of time. Such a Poisson Process is
% called a Non-Stationary Poisson Process (NSPP).
clc
close all
clear all

t = 0;
N(1) = 0;
LmStar = 6;
T = 20;
tN(1) = 0;
i = 2;
while t<T
U = rand();
t = t + (-1/LmStar)*log(U);
U = rand();
if U <= Lmfxn(t,T)/LmStar
    N(i) = N(i-1)+1;
    tN(i) = t;
    i = i + 1;
end

end


figure

    subplot(211)
    for i = 1:100
        [tN,N] = nsppSim(LmStar,T);
        stairs(tN,N, 'color','red')
        hold on
    end
    grid on
    title('NSPP Sample Paths','Interpreter','latex')
    xlabel('$t$','Interpreter','latex')
    ylabel('$N(t)$','Interpreter','latex')
    xlim([2,14])

    subplot(212)
    for i = 1:length(N)
        x(i) = Lmfxn(i,T);
    end
    plot(1:length(N),x,'LineWidth',1.5,'color','green')
    xlim([2,14])
    ylim([0,6])
    grid on
    title('Deterministic Poisson Rate Varying with Time','Interpreter','latex')
    xlabel('$t$','Interpreter','latex')
    ylabel('$N(t)$','Interpreter','latex')

%% Function for Non-Stationary Poisson Process Rate
function y = Lmfxn(t,T)

    if t<=T/2
```

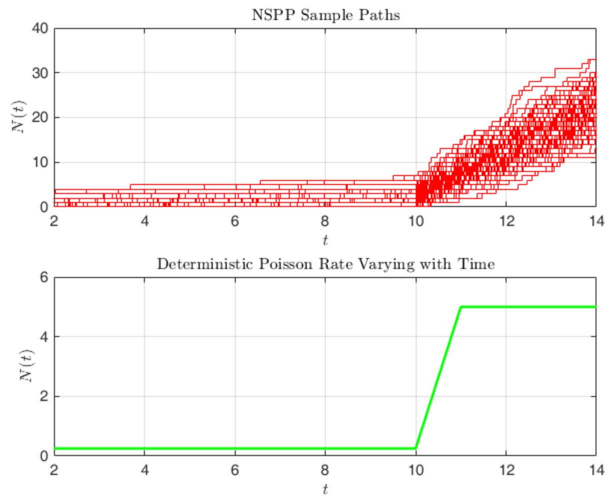

Figure 8: In this graph, we have used a Step function for the change in $\lambda(t)$. The corresponding change in the Poisson Process can easily be observed.

```
        y = 0.25;
    else
        y = 5;
    end

end

%% Function for NSPP Simulation
function [tN,N] = nsppSim(LmStar,T)
    t = 0;
    N(1) = 0;
    tN(1) = 0;
    i = 2;
    while t<T
        U = rand();
        t = t + (-1/LmStar)*log(U);
        U = rand();
        if U <= Lmfxn(t,T)/LmStar
            N(i) = N(i-1)+1;
            tN(i) = t;
            i = i + 1;
        end

    end

end
```

2.4 Compound Poisson Processes

In many applications, arrivals don't simply occur one after the other. We often see the arrival in batches, where the size of each batch is itself governed by a random variable. Such Poisson Processes are called **Compound Poisson Processes**.

Let the batch strength be denoted by B which is distributed by some general distribution G. The arrival of each such batch is tracked by a counting process $N(t)$. We define X such that:

$$X(t) = \sum_{n=1}^{N(t)} B_n \tag{6}$$

By Wald's Equation, we know that $E(X(t)) = E(N(t))E(B) = \lambda t E(B)$. If the Poisson process was non-stationary, we could write $E(X(t)) = m(t)E(B)$ and proceed accordingly. The simulation of this process is fairly simple. The algorithm is as follows:

1. Set $t = 0, N = 0, X = 0$.
2. Generate U.
3. Set $t = (-1/\lambda)\ln(U)$. If $t > T$, then stop.
4. Generate B distributed as G.
5. Set $N = N + 1$, $X = X + B$, $t_N = t$.
6. Go back to 2.

2.4.1 MATLAB Code

In this code, a Gamma Distribution was used to model the variation in batch strength. The process is stationary.

```
%% Compound Poisson Processes
clc
clear all
close all

[tN,N,B,X] = compoundPois();
figure
plot(1:0.1:10,1:0.1:10,'color','blue','LineWidth',2,'DisplayName','$x=y$')
hold on
stairs(tN,X,'DisplayName','$\lambda = 1$','color','red','DisplayName','Sample Path');
hold on
h2 = legend('show');
set(h2,'Interpreter','latex','AutoUpdate','Off')
for i = 1:100
```

```
    [tN,N,B,X] = compoundPois();
    stairs(tN, X,'color','red','LineWidth',0.5)
    hold on
end
grid on
title('Sample Paths for Compound Poisson Distribution ($\lambda = 1$)','Interpreter','latex')
xlabel('$t$','Interpreter','latex')
ylabel('$X(t)$','Interpreter','latex')
%% Function for Simulating a Compound Poisson Process
function [tN,N,B,X] = compoundPois()
    t = 0;
    N(1) = 0;
    X(1) = 0;
    G = makedist('Gamma','a',7,'b',1);
    i = 2;
    B(1) = random(G);
    T = 10;
    Lm = 1;
    tN(1) = 0;
    while t<=T
        U = rand();
        t = t + (-1/Lm)*log(U);
        if t>T
            break
        end
        B(i) = random(G);
        N(i) = N(i-1) + 1;
        X(i) = X(i-1) + B(i);
        tN(i) = t;
        i = i+1;
    end
end
```

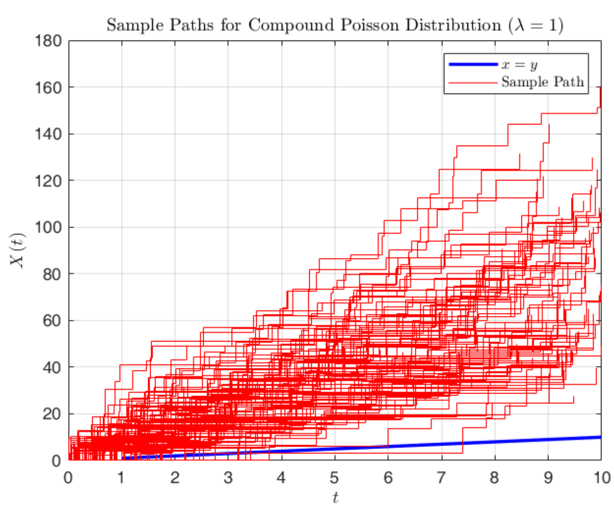

Figure 9: Using a Gamma Distribution, we can see how the process behaves much differently compared to a simple Poisson Process with the same rate.

3 Simulating Markov Chains

Markov Chains are very important class of Stochastic Processes because of their wide applicability and some inherent properties.

A stochastic process $\{X_n : n \geq 0\}$ *is called a* ***Markov Chain*** *if for all times* $n \geq 0$ *and all states* $i_0, ..., i, j \in \mathcal{S}$

$$P(X_{n+1} = j | X_n = i_n, X_{n-1} = i_{n-1}, ..., X_0 = i_0) = P(X_{n+1} = j | X_n = i) = P_{ij} \tag{7}$$

For each state that a Markov Chain can take, we may assign a probability of transition, P_{ij}, all of which can be arranged in a matrix. This matrix is a **Transition Probability Matrix** (TPM), $\mathbf{P}$. One property of this matrix is that the rows sum up to 1, i.e.,

$$\sum_{j \in \mathcal{S}} P_{ij} = 1$$

The most fundamental property that a Markov Chain holds is about being memory less, i.e., the probability of the next state only depends on the state the process is right now. This memorylessness is observed across domains in many applications especially in physics, operations research and finance. In order to simulate a Markov Chain we use the rows of a TPM as a probability mass function (PMF), i.e.,

$$P(Y_i = j) = P_{i,j},\ j \in \mathcal{S}$$

In this text, we use a Multinomial distribution with probabilities given by that row of the TPM. This enables us to draw randomly from each PMF and simulate the Markov Chain. The code for this is in section 3.1.

As an example for a Markov Chain, we simulate a Random Walk. Let $\{\delta_n : n \geq 1\}$ is any iid sequence of increments and

$$X_n = \delta_1 + ... + \delta_n \ , X_0 = 0 \tag{8}$$

Clearly, from the Markov property, we may write $X_{n+1} = X_n + \delta_{n+1}$, $n \geq 0$. Here, we simulate a *simple* random walk, such that $P(\delta = 1) = p$, $P(\delta = -1) = 1 - p$ we use $p = 1/2$. The code for this can also be found in 3.1. In this text, we refer to other methods of simulating Markov Chains with relevant application areas later on as well.

3.1 MATLAB Code

```
%% Markov Chain Simulations
clc
clear all
close all
%% Discrete Time Markov Chain
```

```matlab

P = [0.2, 0.7, 0.1;
    0.25,0.35,0.4;
    0.1,0.8,0.1]; %Probability Transition Matrix
initState = 1;
simLength = 100;
X = markovChain(P,initState,simLength);
markovChainPlotter(P,X)

%% Random Walk as a Markov Process
simLength = 10000;
T = 10;
del = randn(simLength,1); % an iid sequence
R(1,1) = 0;
for i = 2:simLength
   R(i,1) = R(i-1,1) + del(i);
%    Rr(i,1) = max(R(i-1,1) + del(i),0);
end
figure
plot(R)
% hold on
% plot(Rr)
grid on
xlabel('$n$','Interpreter','latex')
ylabel('$X_n$','Interpreter','latex')
title('Simple Random Walk','Interpreter','latex')

%% Function For making a DTMC
function chain = markovChain(P,initState,simLength)
    X(1) = initState;
    for i = 1:simLength
       Y = makedist('Multinomial','probabilities',P(X(i),:));
       X(i+1) = random(Y);
    end
    chain = X;
end

function mcPlot = markovChainPlotter(P,chain)
    figure
    subplot(2,1,1)
    plot(1:length(chain),chain,'o-');
    grid on
    xlabel('Time','Interpreter','latex')
    ylabel('State','Interpreter','latex')
    ylim([0,max(chain)+1])
    xlim([0,length(chain)])
    title('Markov Chain Behaviour','Interpreter','latex')
    subplot(2,1,2)
    graphplot(dtmc(P),'LabelEdges',true);
    title('Markov Chain State Transition Diagram','Interpreter','latex')
%     legend(['P])
end
```

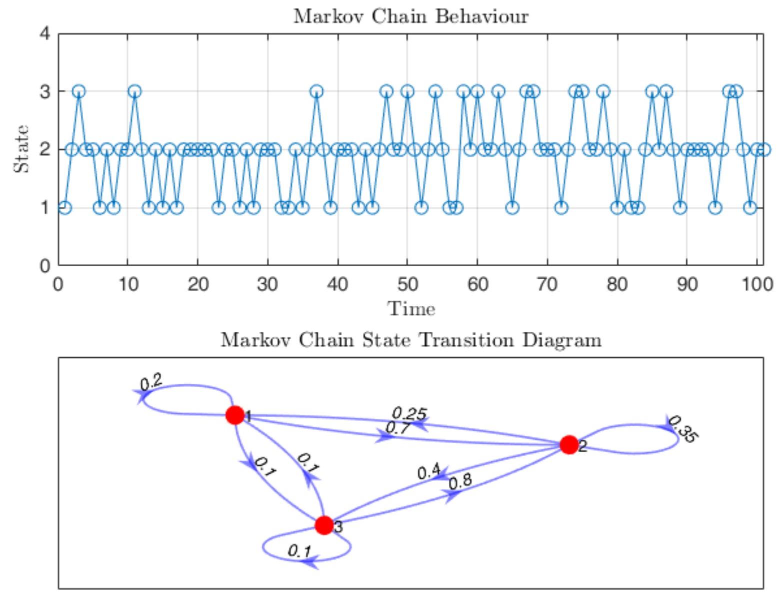

Figure 10: State Transition Diagram for a Discrete State Markov Chain and a sample path

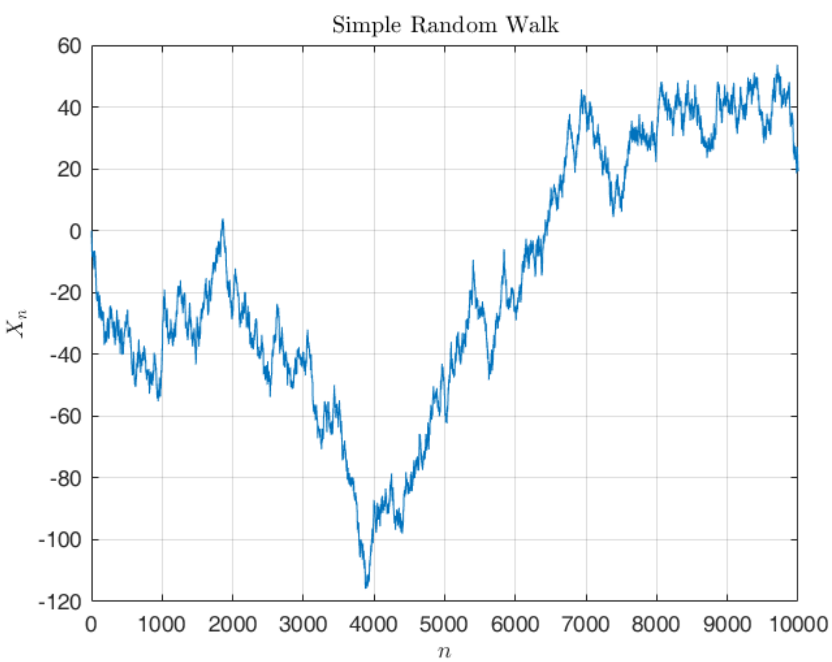

Figure 11: Simple Random Walk

4 Single Server Queueing Model

In this section we simulate a first-in-first-out Single Server Queueing Model. Here we are trying to simulate a system which only has one server, and customers arrive in exponentially distributed inter-arrival times $\{T_n\}$ (here, we have taken the rate to be $\lambda = 1$) and the time taken to service one individual arrival $\{S_n\}$ is uniformly distributed (taking parameters as $a = 1$ and $b = 3$).

Here, we are trying to estimate the delay faced by the n^{th} arrival. We define the delay as the time the arrival has to wait in queue before being serviced. Hence, it should be a sum of the service time of the current customer being serviced, and the time the customer has to wait in the queue. The queueing process is a Markov Process and hence it makes it easy for us simulate the behaviour.

We propose a convenient recursion which will enable us to simulate the delay times for the n^{th} customer recursively.

$$D_{n+1} = \max\{D_n + S_n - T_n, 0\} \tag{9}$$

Notice how the delay faced by the $(n+1)^{th}$ customer is solely a function of the delay for the n^{th} customer. This basic property signals a memorylessness in the process and hence the Queueing Model is a Markov Chain.

This queue is called a FIFO G—G—1 queue because we have two generalized distributions for the arrival and service times. Other variations exist which have not been explored in this text. The algorithm is as follows:

1. Set $t = 0, N = 0, D(1) = 0$
2. Generate U
3. Set $t = t + (-1/\lambda)\ln(U)$. If $t > T^*$, break.
4. Sample S_n (for service time) and set $T_n = t_n - t_{n-1}$
5. Evaluate $D_{n+1} = \max\{D_n + S_n - T_n, 0\}$
6. Go back to 2.

We are interested in evaluating average delay faced over all n customers until a time T^*. So we simulate multiple instances of this queue and estimate the delay by taking the average of delays. Let d_n be the estimate. Then,

$$d_n = \frac{1}{n}\sum_{j=1}^{n} D_j$$

Also, we may be interested in evaluating the number of consumers that have a delay greater than say some x. This is done by the estimate given by:

$$d_n^{(x)} = \frac{1}{n}\sum_{j=1}^{n} I\{D_j > x\}$$

where $I(.)$ is the indicator function. It returns 1 if the argument is true, else false.

4.1 MATLAB Code

```
%% Queueing Model

% Distribution of inter-arrival times (Tia)-> Exponential(Lm)
% Distribution of service times (S)       -> Uniform(a,b)

% Recursion for Delay-times is known. Simulate S and Tia to find D.

clc;
clear all;
close all;

T = 10;
Lm = 1;
%% Plotting one Sample Path for a FIFO Queue
% Plot of Delays
figure
% subplot(
plot(0:100,0:100,'color','red','DisplayName','$x = y$')
hold on
[tN,Tia,N,S,D]=fifoQ(Lm,T);
stairs(tN,D,'--','color','blue','DisplayName','Sample Path')
xlim([0,T])
h = legend('show');
set(h,'Interpreter','latex','AutoUpdate','Off')
hold on
for i = 1:100
    [tN,Tia,N,S,D]=fifoQ(Lm,T);
    stairs(tN,D,'--','color','blue')
    hold on
end
plot(0:100,0:100,'color','red','LineWidth',3.5)
grid on
title('$G|G|1$ Queue Simulation - Delay Times','Interpreter','latex')
xlabel('$t$','Interpreter','latex')
ylabel('$\{D_n\}$','Interpreter','latex')

figure
plot(0:100,0:100,'color','red','DisplayName','$x = y$')
hold on
[tN,Tia,N,S,D]=fifoQ(Lm,T);
stairs(tN,N,'--','color','blue','DisplayName','Sample Path')
xlim([0,T])
h = legend('show');
set(h,'Interpreter','latex','AutoUpdate','Off')
hold on
for i = 1:100
    [tN,Tia,N,S,D]=fifoQ(Lm,T);
    stairs(tN,N,'--','color','blue')
    hold on
end
plot(0:100,0:100,'color','red','LineWidth',3.5)
```

```
grid on
title('$G|G|1$ Queue Simulation - Incoming Customers','Interpreter','latex')
xlabel('$t$','Interpreter','latex')
ylabel('$\{D_n\}$','Interpreter','latex')

figure
hist_S = [];
for i = 1:1000
    [tN,Tia,N,S,D] = fifoQ(Lm,T);
    hist_S = [hist_S;S'];
end
histogram(hist_S,'Normalization','pdf')
grid on
title('Distribution of Service Times', 'Interpreter','latex')
xlabel('$s$','Interpreter','latex');
ylabel('$P(S = s)$','Interpreter','latex');
figure
hist_Tia = [];
for i = 1:1000
    [tN,Tia,N,S,D] = fifoQ(Lm,T);
    hist_Tia = [hist_Tia;Tia'];
end
histogram(hist_Tia,'Normalization','pdf')
grid on
title('Distribution of Inter-Arrival Times','Interpreter','latex');
xlabel('$t_{ia}$','Interpreter','latex');
ylabel('$P(T = t_ia)$','Interpreter','latex');
%% Long Run Simulation - Estimating Long Run Delays
% We use this method to estimate the average delay faced by all customers.
MCD = [];
for i = 1:1000
    [tN,Tia,N,S,D] = fifoQ(Lm,T);
    MCD = [MCD;D'];
end
average_delay_allCustomers = mean(MCD);

% Now evaluate the proportion of customers facing this average delay
CustProportion_for_x_delay = mean(MCD(MCD>mean(MCD)));

%% Function for FIFO G|G|1 Queue
function [tN,Tia,N,S,D]=fifoQ(Lm,T)

    t = 0;
    N(1) = 1;
    tN(1) = (-1/Lm)*log(rand());
    S(1) = 1 + 2*rand();
    D(1) = 0;

    i = 2;
    while t<T
        U = rand();
        t = t + (-1/Lm)*log(U);
        if t>T
            break
        end
        tN(i) = t;
        Tia(i) = (-1/Lm)*log(U);
```

```
        N(i) = N(i-1) + 1;
        S(i) = 1 + 2*rand();
        D(i) = round(max(D(i-1)+S(i-1)-Tia(i-1),0));
        i = i + 1;
    end
end
```

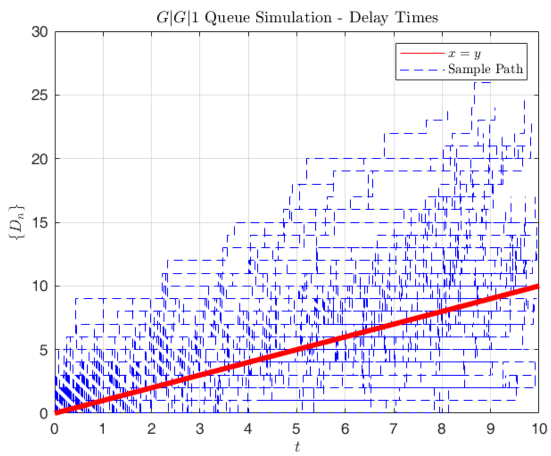

Figure 12: Delay Time Sample Paths

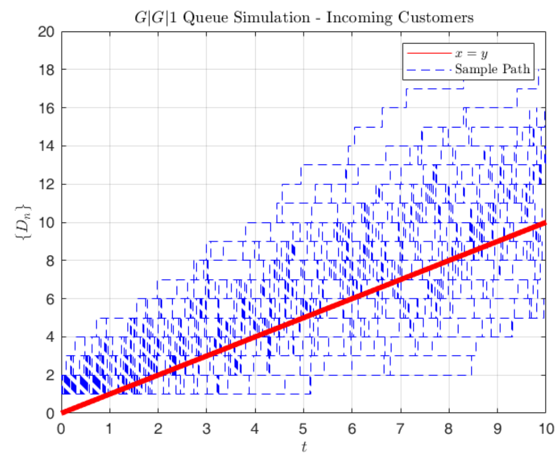

Figure 13: Incoming Customers Sample Paths

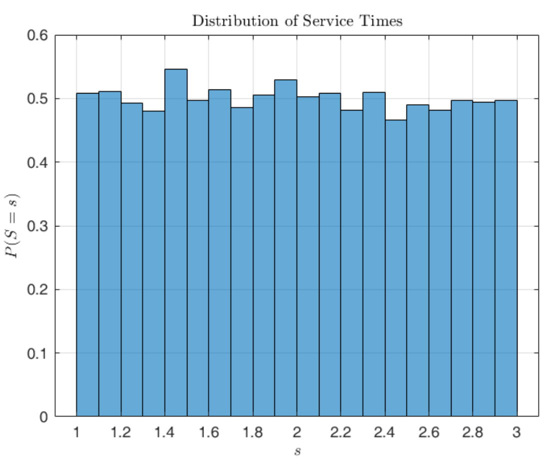

Figure 14: Distribution of Services

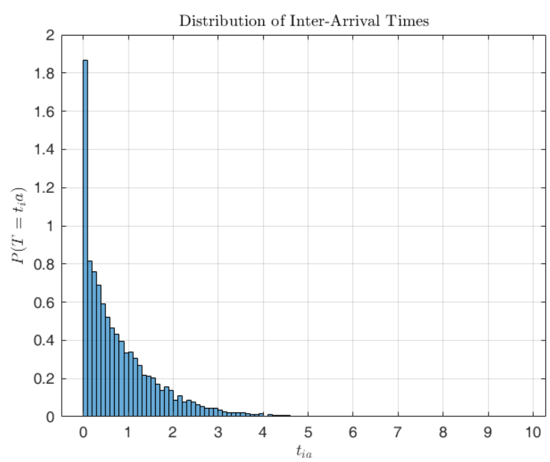

Figure 15: Distribution of Arrivals

5 Brownian Motion and its Variants

Although the journey of Brownian Motion began in physics and the movements of particles in a suspended medium, we here are exploring the mathematical formulation and simulation of Brownian Motion.

A stochastic process $\mathbf{B} = \{B(t) : t \geq 0\}$ possessing, with probability 1, continuous sample paths is called the **standard Brownian Motion** if:

1. $B(0) =$).
2. $\mathbf{B}$ has both stationary and independent increments
3. $B(t) - B(s)$ has a normal distribution with mean 0 and variance $t - s$ for all $0 \leq s < t$.

Let $\{Z_t\}$ be standard normals. The algorithm to generate this process by using a simple recursion given by

$$B(t_k) = \sum_{i=1}^{k} \sqrt{t_i - t_{i-1}} Z_i \tag{10}$$

Since we already know how to simulate a unit normal from the Acceptance-Rejection method, we can use that here to simulate a Brownian Motion.

Often times it is important to simulate a Brownian motion with an inherent drift and volatility present in it. BM with drift finds many applications in physics. Let $X(t) = \sigma B(t) + \mu t$ be the stochastic process where $\mu \in \mathbb{R}$ and $\sigma > 0$. It has continuous paths wp1 and is defined by the same criteria as in BM but the distribution of increments is a normal with mean $\mu(t - s)$ and variance $\sigma^2(t - s)$. We can easily simulate this using the recursion

$$X(t_k) = \sum_{i=1}^{k} \sigma \sqrt{t_i - t_{i-1}} Z_i + \mu(t_i - t_{i-1}) \tag{11}$$

Finally, another variant of BM with drift is a BM which is exponentiated. This is called a **Geometric Brownian Motion**.

$$S(t) = S(0)e^{X(t)}, \ t \geq 0 \tag{12}$$

$e^{X(t)}$ has a log-normal distribution for each $t > 0$. We can simulate this using the following recursion:

$$S(t_k) = S(0) \prod_{i=1}^{k} e^{\sigma \sqrt{t_i - t_{i-1}} Z_i + \mu(t_i - t_{i-1})} \tag{13}$$

The applications for GBM are best observed in Finance where the trajectories of risky securities are modeled using GBM. Hence the area of Derivative Pricing for various options uses GBM.

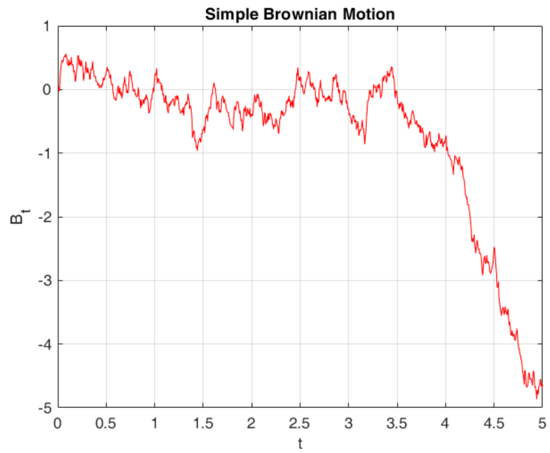

Figure 16: Standard Brownian Motion

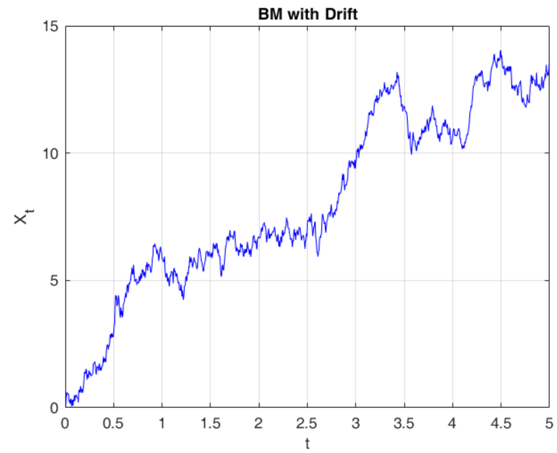

Figure 17: Brownian Motion with Drift

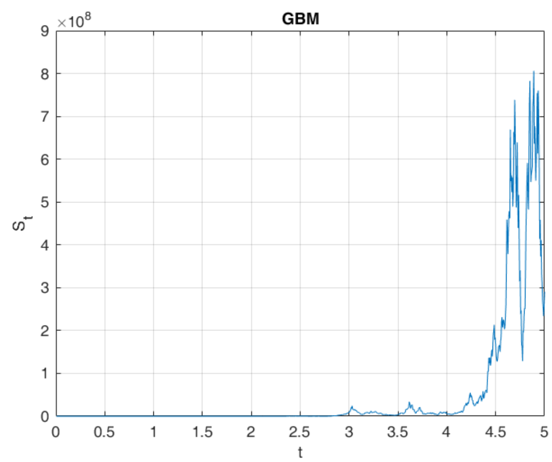

Figure 18: Geometric Brownian Motion

5.1 MATLAB Code

```
clc
clear all
close all
t = 5;
n = 1000;
% rng('default')
% randn('state',100);
dt = t/n;
w = zeros(1,n);
dw = zeros(1,n);
w(1) = 0;
for j = 2:n
   w(j) = w(j-1)+sqrt(dt)*randn;
end

```

```
x(1) = 0;
mu = 2;
sig = 2.5;
for j = 2:n
    x(j) = x(j-1) + sig*sqrt(dt)*randn + mu*dt;
end

s(1) = 1;
% for j = 2:n
%    s(j) = s(j-1)*exp(x(j)-x(j-1));
% end
for j = 1:n
   y(j) = exp(sig*sqrt(dt)*normrnd(0,1) + mu*dt);
end
for j = 2:n
   s(j) = s(j-1)*y(j);
end

T = [dt:dt:t];
test = exp(mu.*T);
figure
plot([dt:dt:t],w,'r-')
grid on
title('Simple Brownian Motion')
xlabel('t')
ylabel('B_t')
figure
plot([dt:dt:t],x,'b-')
grid on
title('BM with Drift')
xlabel('t')
ylabel('X_t')
figure
plot([dt:dt:t],s)
grid on
title('GBM')
xlabel('t')
ylabel('S_t')
```

6 Classic (S, s) Policy Inventory Model

Here we attempt to model a company which sells a product which is stored at some location. Said location has a rent or storage cost associated to it and a cost for re-supplying the location. Uncertainty manifests itself here in two aspects of the business. The first is the uncertainty in demand. Customers file orders for buying the product following some stochastic process. In this text we use a Poisson Process to model incoming orders. Also, the time taken for a re-supply request to be fulfilled, often referred to as a *lead time*, is uncertain. Here we assume it is normally distributed for convenience. The algorithm allows for any general distribution for the lead times.

We define some constants such as price of item bought by the customer as r, cost of delivery as some c, and cost of storage per item as h. Note that these constants can be deterministic or stochastic variables. For this simulation, we assume their behaviour to be constant scalar values. The Classic (S, s) policy is a simple inventory management strategy. It simply asks the manager to empty out the inventory based on the amount of order received. So, the number of items sent out is a function only of incoming order and the current level of the inventory. This is the simplest model possible for an inventory process.

Let I be the current inventory level, B be the order amount from the consumer. Then, we define m as

$$m = \min\{I, B\}$$

Now, to model the costs based on the assumed constants, we let $C_o(t)$ denote the ordering costs upto time t, $C_h(t)$ as total holding costs up to time t, and $R(t)$ as the total revenue upto time t. We wish to know the process

$$X = X(T) = \frac{R(T) - C_o(t) - C_h(t)}{T} \tag{14}$$

where T is the time till we run the simulation. The $E(X)$ will tell us the average money made by the business over multiple instances of simulation.

The algorithm for this is based on the assumption that there are only two discrete events occuring at once - customer request and re-supply. Let t_A be the time for the arrival of the next consumer order and t_o be the time for the arrival of the re-supply request. Once the re-supply order is received, we simply set t_o to ∞. The amount to be order for every re-supply request, say Y, is set to zero at the end of every re-supply request.

Algorithm is as follows:

1. Set $t = C_o = C_h = R = Y = 0$. Set $t_o = \infty$. Set $t_A = \frac{-1}{\lambda}\ln(U)$.
2. **Case - I**: Customer Request is next event, i.e., $t_A = \min\{t_A, t_o\}$:
   (a) If $t_A \geq T$, reset $C_h = C_h + (T - t)hI$ and give output as $X = (R - C_o - C_h)/T$ and break. Else, continue.
   (b) Reset $C_h = C_h + (t_A - t)hI$, $t_A = t - (1/\lambda)\ln(U)$.
   (c) Generate $B \sim G$.

(d) Set $m = \min\{I, B\}$.
(e) Reset $R = R + rm$, $I = I - m$
(f) If $I < s$ and $Y = 0$, then reset $Y = S - I$. Generate $L \sim H$. Reset $t_o = t + L$.

3. **Case - II**: Re-supply delivery is next event, i.e., $t_o = \min\{t_A, t_o\}$:

   (a) If $t_o \geq T$, then reset $C_h = C_h + (T - t)hI$ and give output as $X = (R - C_o - C_h)/T$ and break. Else, continue.

   (b) Reset $C_h = C_h + (t_o - t)hI$, $C_o = C_o + c$, $I = I + Y$, $t = t_o$, $t_o = \infty$, $Y = 0$.

In the above code, H and G are general distributions for lead time and order quantity respectively. These can be any general distributions. In the MATLAB code associated with this simulation, we have used a normal distribution for both of these. Ideally, data would be provided to estimate the distributions and simulate a system accordingly.

6.1 MATLAB Code

```
clc
clear all
close all
%% Defining the model
Lm = 15;
T = 365-52;
S = 100;
s = 25;
storageCosts = 2;
rate = 2.5;

pdDemand = makedist('Normal',20,5);
pdLeadTime = makedist('Normal',2,0.75);

output = ClassicSsModel(rate,storageCosts,S,s,Lm,T,pdDemand,pdLeadTime);


clear ans
clc
%% Plotting
figure
plot(-50:50,pdf(pdDemand,-50:50),'Linewidth',2.5)
hold on
histogram(random(pdDemand,10000,1),'Normalization','pdf')
xlim([-10,50])
ylim([0,0.1])
grid on
title('Probability Distribution for Demand')
xlabel('Order Quantity')
ylabel('Probability')

figure
plot(-2:0.025:6,pdf(pdLeadTime,-2:0.025:6),'Linewidth',2.5)
hold on
```

```
histogram(random(pdLeadTime,5000,1),'Normalization','pdf')
% xlim([-10,50])
% ylim([0,0.1])
grid on
title('Probability Distribution for Lead Time')
xlabel('Time Taken')
ylabel('Probability')

ModelAnalysisPlotter(output)

%% Monte Carlo Simulation - Simulating Multiple Trajectories
% T = 365-52+1;
MCLength = 150;
MCX = zeros(T+1,MCLength);
MCCh = zeros(T+1,MCLength);
MCCo = zeros(T+1,MCLength);
MCR = zeros(T+1,MCLength);
MCI = zeros(T+1,MCLength);
for k = 1:MCLength
    output = ClassicSsModel(rate,storageCosts,S,s,Lm,T,pdDemand,pdLeadTime);
    MCX(:,k) = output(:,1);
    MCCh(:,k) = output(:,2);
    MCCo(:,k) = output(:,3);
    MCR(:,k) = output(:,4);
    MCI(:,k) = output(:,5);
end
T = 365-52+1;
figure
% subplot(331)
stairs(MCR)
xlim([0,T])
grid on
xlabel('Time')
title('Revenue Generated')
% subplot(333)
figure
stairs(MCCo)
xlim([0,T])
grid on
xlabel('Time')
title('Cost of Re-Supplying Inventory')
% subplot(335)
figure
stairs(MCX)
title('Net Profit')
xlim([0,T])
grid on
xlabel('Time')
% subplot(337)
figure
stairs(MCCh)
xlim([0,T])
grid on
xlabel('Time')
title('Cost of Storage')
% subplot(339)
figure
```

```
stairs(MCI)
xlim([0,T])
grid on
xlabel('Time')
title('Inventory Level')
clc


%% Function to simulate Classic Ss Model
function output = ClassicSsModel(rate,storageCosts,S,s,Lm,T,pdDemand,pdLeadTime)
    %%Use this function to create an (S,s) Model and simulate T days'
    %%inventory behaviour.

    % Input:
    % rate       -> selling price of each item in the inventory
    % S, s       -> Model Parameters, S and s are upper and lower bounds
    %               for theInventory's buy and sell mechanism
    % Lm         -> Poisson Process Rate for incoming buyer orders
    % T          -> Maximum Length of Simulation. If not specified,
    %               it will run for314 days
    % costFxn    -> Cost - Function for Storage
    % pdDemand   -> Distribution for Order Quantity coming from Buyer
    % pdLeadTime -> Lead Time for each re-supply order Distribution

    t = 0;
    Ch(1) = 0;
    Co(1) = 0;
    R(1) = 0;
    Y = 0;
    I(1) = S;
    lm = Lm;
    r = rate;
    h = storageCosts;
    to = inf;
    tA =  -(1/lm)*log(rand());

    for i = 1:T
        if min(tA,to)==tA
            if tA>=T
                Ch(i) = Ch(i-1) + (T-t(i-1))*h*I(i-1);
                X = (R(i)-Co(i)-Ch(i))/T;
                break
            else
                Ch(i+1) = Ch(i) + (tA - t(i))*h*I(i);
                t(i+1) = tA;
                tA = t(i+1) - (1/lm)*log(rand());
                B(i) = random(pdDemand);
                m = min(I(i),B(i));
                R(i+1) = R(i) + r*m;
                I(i+1) = I(i) - m;
                if I(i+1)<s && Y == 0
                    Y = S - I(i+1);
                    L(i) = random(pdLeadTime);
                    to = t(i+1) + L(i);
                end
                Co(i+1) = Co(i) + 0;
            end
```

```
        else
            if min(tA,to) == to
                if to>=T
                    Ch(i) = Ch(i-1) + (T-t(i-1))*h*I(i-1);
                    X = (R(i)-Co(i)-Ch(i))/T;
                    break
                else
                    Ch(i+1) = Ch(i) + (to - t(i))*h*I(i);
                    Co(i+1) = Co(i) + costFxn(Y);
                    I(i+1) = I(i) + Y;
                    t(i+1) = to;
                    to = inf;
                    Y = 0;
                    R(i+1) = R(i) + 0;
                end
            end
        end
    end
    X = (R-Co-Ch);
    output = [X; Co; Ch; R; I]';
    function NetCost = costFxn(Y)
        c = 1.5;
        NetCost = Y*c;
    end
end
%% Function for Creating Plots
function [fig1,fig2,fig3,fig4,fig5]  = ModelAnalysisPlotter(output)
    T = length(output);
    X = output(:,1);
    Co = output(:,2);
    Ch = output(:,3);
    R = output(:,4);
    I = output(:,5);
    fig1 = figure

    stairs(R)
    title('Total Revenue Generated')
    grid on
    xlabel('Time')
    xlim([0,T])
    ylabel('R_t')

    fig2 = figure
    stairs(Co)
    title('Cost of Resupplying Inventory')
    grid on
    xlabel('Time')
    xlim([0,T])
    ylabel('C_{ot}')

    fig3 = figure
    stairs(X)
    title('Net Profit')
    grid on
    xlabel('Time')
    xlim([0,T])
    ylabel('X_{t}')
```

```
    fig4 = figure
    stairs(Ch)
    title('Cost of Storage')
    grid on
    xlabel('Time')
    xlim([0,T])
    ylabel('C_{ht}')

    fig5 = figure
    stairs(I)
    title('Inventory Level')
    grid on
    xlabel('Time')
    xlim([0,T])
    ylabel('I_{t}')
end
```

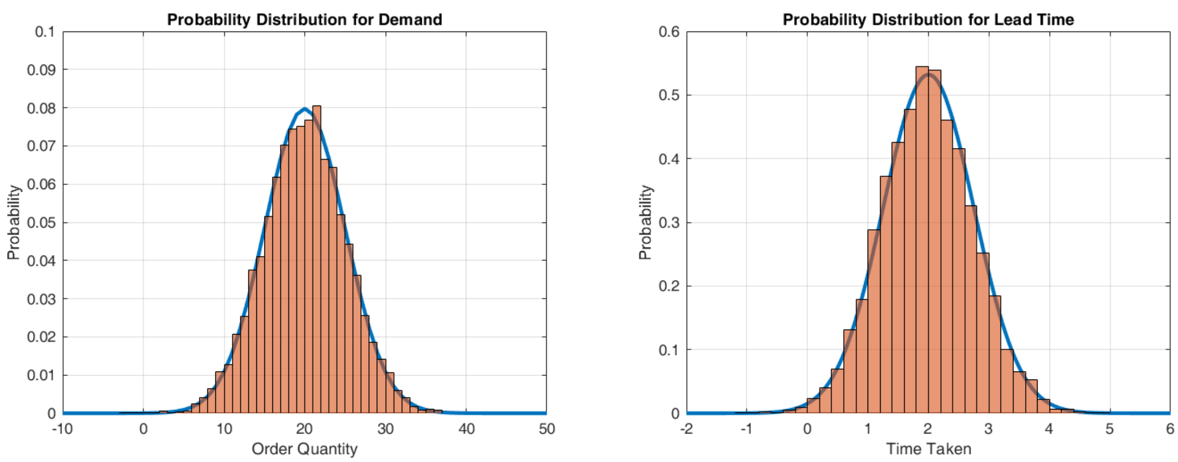

Figure 19: Distributions using in Model

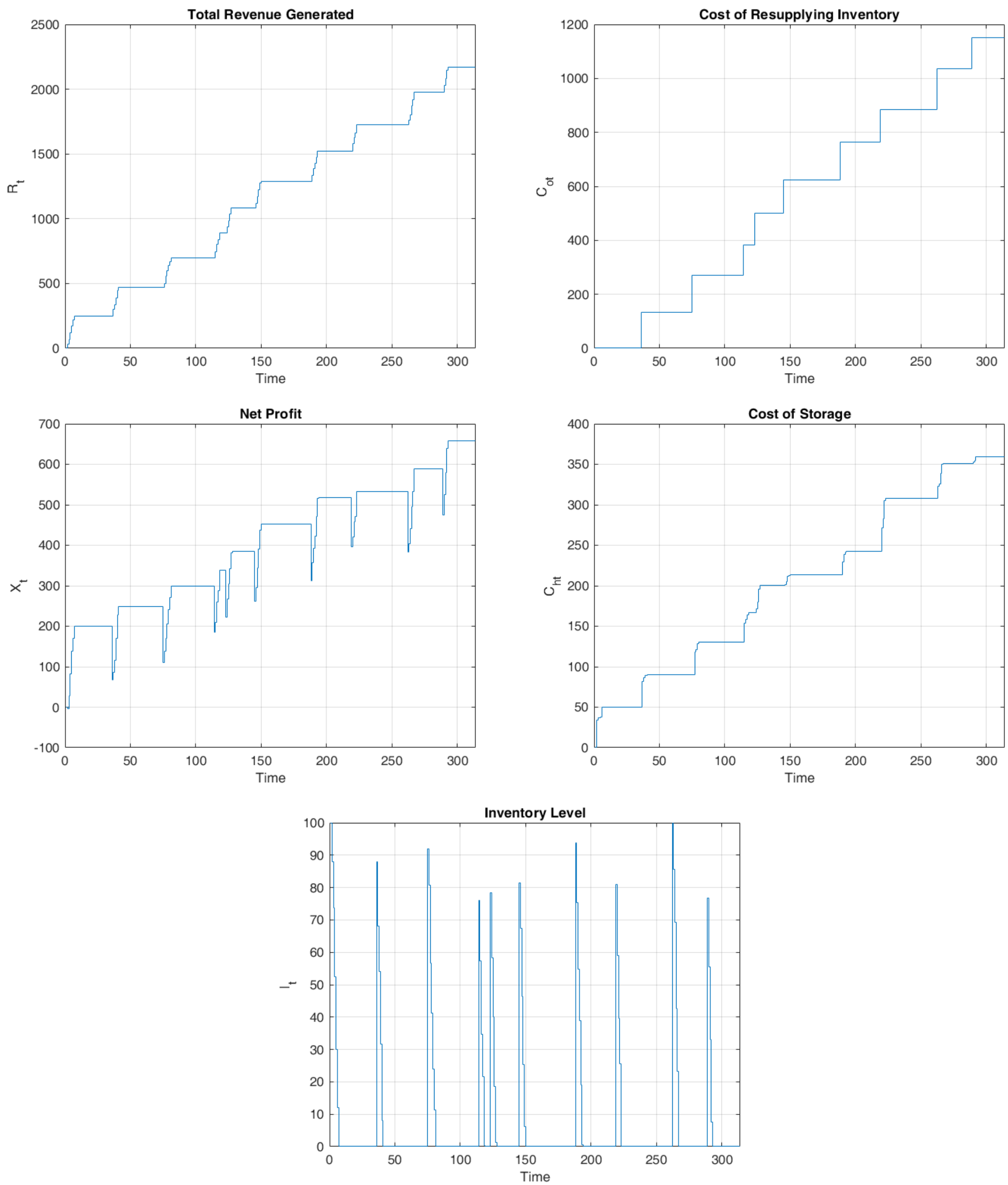

Figure 20: Model Parameters' Sample Paths

7 Classical Insurance Risk Model

In this simulation we simulate the working of an Insurance risk process which is a compound Poisson Process. For this we will define a few model parameters and declare variables as follows:

1. x: Initial reserve of finances with the business
2. c: Constant rate of premium for Insurance risk business
3. $\{t_n\}$: Point Process defining occurence of claims against the insurance risk business. Corresponding counting process is $N(t)$. This is a Poisson Process with rate λ.
4. B_n: Claim amount which is a random variable coming from a general distribution G.
5. X_n: The unrestricted reserve process. It is defined as follows:
$$X_n(x) = x + ct_n - \sum_{j=1}^{n} B_j,\ n \geq 1, \text{ and } X_0(x) = x \tag{15}$$
6. $tau(x)$: The stopping time for the business characterized by:
$$\tau(x) = \min\{t_n : X_n(x) \leq 0\} \tag{16}$$
7. $P(\tau(x) < \infty)$: **Probability of Ruin** which models the probability that the insurance business goes bankrupt.
8. M : the magnitude of ruin given that it occurred.
9. I : the indicator for the event { ruin by time T} $= \{\tau(x) \leq T\}$.
10. $R = R(x)$: the reserve level at that time given it started.

In general, we define an algorithm to simulate one run of an insurance risk process and repeat the runs for statistically large number of times to estimate the probability of ruin. The following algorithm generates one copy of the indicator $I\{\tau(x) \leq T\}$.

1. Set $t = 0, R = x, \tau = \infty, I = 0, M = 0$.
2. Generate U.
3. Set $t = t + (-(1/\lambda)\ln(U))$. If $t > T$ stop. $R = R + c(-(1/\lambda)\ln(U))$.
4. Generate $B \sim G$. Set $R = R - B$. If $R \leq 0$, then set $I = 1$, set $\tau = t$, set $M = |R|$ and stop.
5. Go back to 2.

For estimation of $\tau(x)$, we run this code multiple times and evaluate the mean of the indicator function's counts. Similarly, we can evaluate the mean of the magnitude of ruin M. The same is done in the code given.

7.1 MATLAB Code

```
%% MC Sim
clc
clear
close all
MCLength = 1000;
T = 45;
x = 10;
Lm = 1;
c = 1;

%% Case - 1: Distribution of Claims - Uniform

G1 = makedist('Uniform',1,2);
[R1, tau1, M1, fig1] = ruinSim(x,T,Lm,c,G1,MCLength);
ruin1 = mean(tau1(isfinite(tau1)));

%% Case - 2: Distribution of Claims - Gamma

G3 = makedist('Gamma',1,3);
[R3, tau3, M3, fig3] = ruinSim(x,T,Lm,c,G3,MCLength);
ruin3 = mean(tau3(isfinite(tau3)));



%% Function for Simulating Ruin
function [R, tau, M, fig] = ruinSim(x,T,Lm,c,G,MCLength)
    R = zeros(T,MCLength);
    R(1,:) = x;
    I = zeros(1,MCLength);
    M = zeros(1,MCLength);

    for j = 1:MCLength
        i = 1;
        t = 0;
        tau(j) = inf; %Time of Ruin
        while t<T
            i = i + 1;
            U = rand();
            t = t + ((-1/Lm)*log(U));
            if t>T
                break
            end
            R(i,j) = R(i-1,j) + c*((-1/Lm)*log(U));
            B = random(G);
            R(i,j) = R(i,j) - B;
            if R(i,j)<0
                I(j) = 1;
                tau(j) = i;
                M(j) = abs(R(i,j));
                break
            end
        end
    end

    fig = figure;
```

```
    subplot(2,2,1)
    plot(R)
    title('Reserve Level with Time')
    grid on
    xlabel('Time')
    ylabel('Reserve Level in USD')
    subplot(2,2,2)
    histogram(tau,'Normalization','probability')
    title('Distribution of Time to Ruin')
    grid on
    xlabel('Time in Days')
    ylabel('Probability')
    subplot(2,2,3)
    histogram(random(G,10000,1),'Normalization','probability')
    title('Distribution of Claim Amount')
    grid on
    xlabel('Claim Amount in USD')
    ylabel('Probability')
    subplot(2,2,4)
    histogram(M,'Normalization','probability')
    title('Distribution of Magnitude of Ruin')
    grid on
    xlabel('Amount in USD')
    ylabel('Probability')
end
```

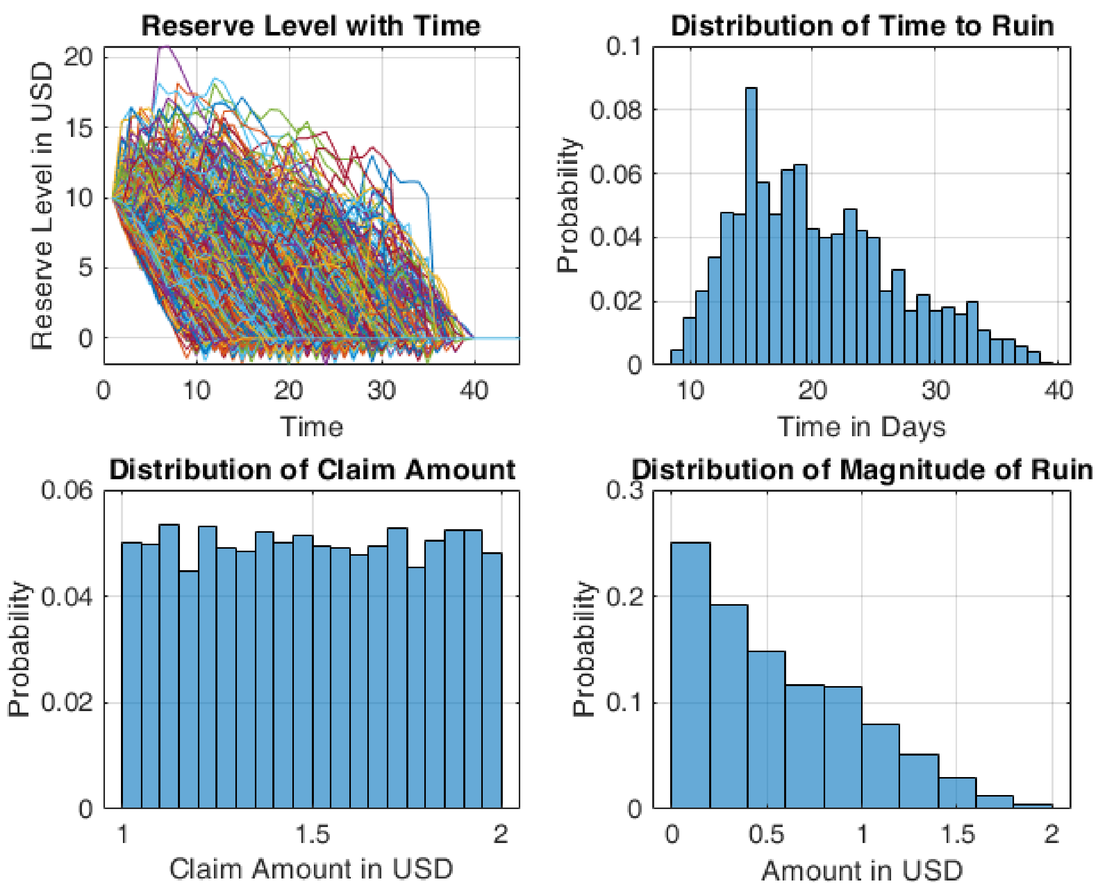

Figure 21: Model Parameters with Uniform Distribution of Claims

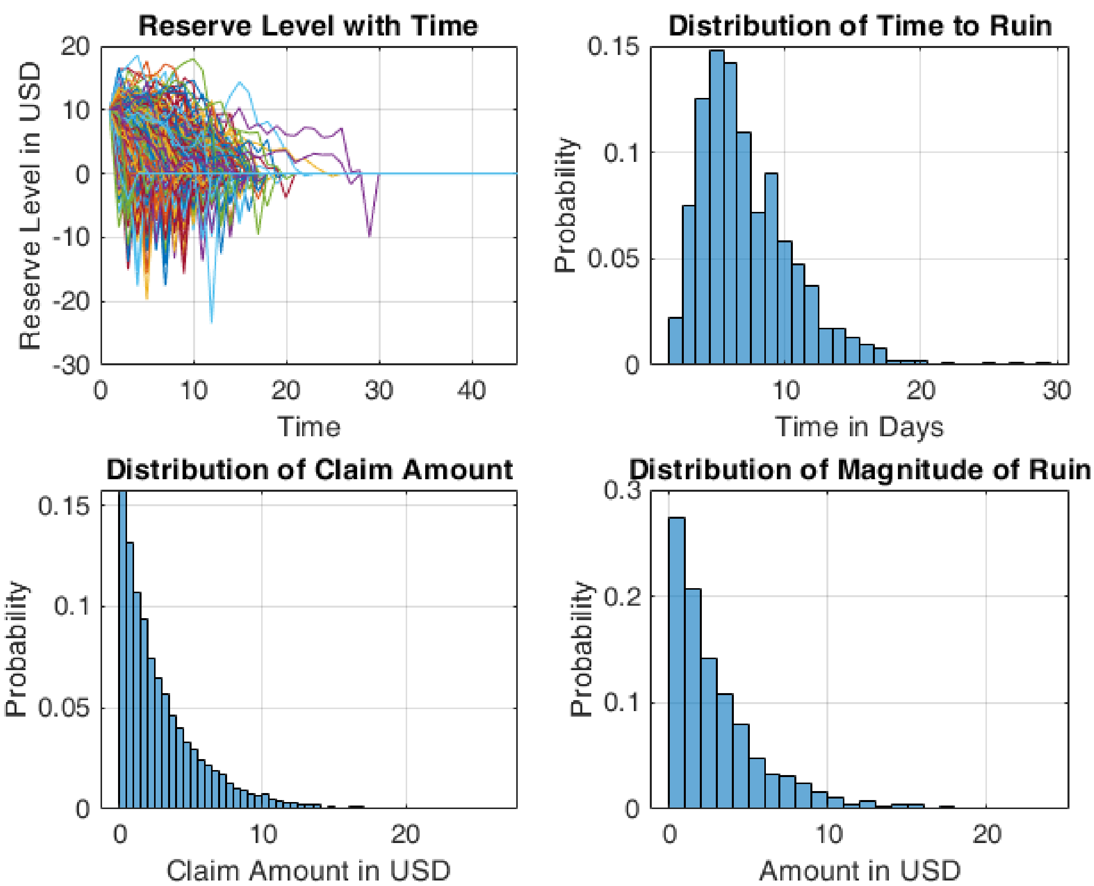

Figure 22: Model Parameters with Gamma Distribution of Claims