Assignment III (MA226)

Name: Sourav Bikash

Roll No.:11012338

Submission Date: 25/01/2013 Time:23:59 hrs.

Aim of the Problem:

The problem involves the use of a linear congruence generator. The basic aim lies in the fact that although it generates a large number of values, the values are not randomly distributed. And there exists a definition to these distributions.

Mathematical Analysis/Theory:

The problem uses the following linear congruence generator:

$$x_{i+1} = (ax_i + b) \mod m$$
$$u_{i+1} = x_{i+1}/m$$

It generates a sequence of x_i and a dependent sequence u_i.
For the second part of the problem we make use of the extended Fibonacci generator. This is given by:

$$U_i = (U_{i-17} + U_{i-5}) \mod 2^{31}$$

This requires buffering of 17 values.

Part I:

A sequence of x_i was generated using the linear congruence generator. Using this a sequence of U_i was obtained. The frequencies of the obtained U_i was plotted using a barplot.

C++ implementation:

```
#include<iostream>
#include<climits>
#define N 100000
using namespace std;
int axmodm(int m, int a, int x)
{
	int q = m/a;
	int r = m%a;
	int k = x/q;
	x = a * (x - k * q) - k * r;
	if(x < 0)
		x += m;
	return x;
}

void generate(int m, int a)
{
	//daclaring variables
	int x0 = 1, count, j;
	double *u;
```

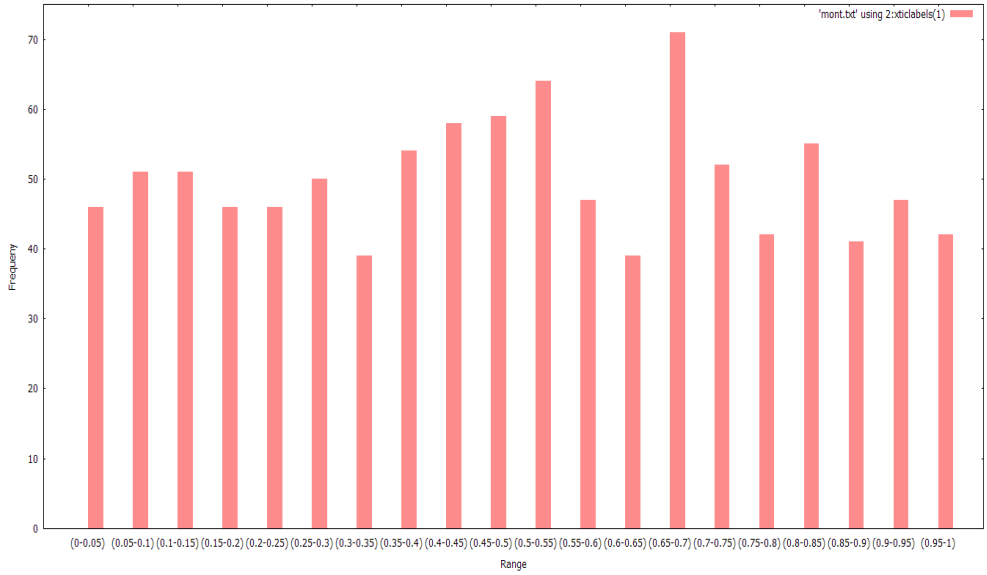
```
	int frequency[20][3];// three frequncies 1000,10000,100000
	u = new double[N];// store frequencies in this array
	int sum[3] = {0, 0, 0};
	//initialising frequency array
	for(count = 0; count < 20; count++)
	{
		frequency[count][0] = frequency[count][1] = frequency[count][2] = 0;
	}
	//calculating frequencies
	for(count = 0; count < N; count++)
	{
		u[count] = (x0*1.0)/m;
		if(count < 1000)
		{
			frequency[(int)(u[count]/0.05)][0]++;
			frequency[(int)(u[count]/0.05)][1]++;
			frequency[(int)(u[count]/0.05)][2]++;
		}
		else if(count < 10000)
		{
			frequency[(int)(u[count]/0.05)][1]++;
			frequency[(int)(u[count]/0.05)][2]++;
		}
		else if(count < N)
		{
			frequency[(int)(u[count]/0.05)][2]++;
		}
		x0 = axmodm(m, a, x0);
	}
	//display frequency
	for(count = 0; count < 20; count++)
	{

	cout<<frequency[count][0]<<'\t'<<frequency[count][1]<<'\t'<<frequency[count][2]
<<endl;
		sum[0]+=frequency[count][0];
		sum[1]+=frequency[count][1];
		sum[2]+=frequency[count][2];
	}
	cout<<sum[0]<<'\t'<<sum[1]<<'\t'<<sum[2]<<endl;
}

void coordinates(int m, int a)
{
	//declaring variables
	int x0 = 1, count, j;
	double *u;
	u = new double[N];
	//display coordinates
	for(count = 0; count < N; count++)
	{
		u[count] = (x0*1.0)/m;
		if(count == 0)
		{
			cout<<u[count]<<endl;
		}
		else
		{
			cout<<'\t'<<u[count]<<endl<<u[count];
		}
		x0 = axmodm(m, a, x0);
	}
```

```
}

void generate17(int m, int a)
{
	//declaring variables
	int x0 = 1, count;
	//displaying sequence
	for(count = 0; count < 17; count++)
	{
		cout<<x0<<endl;
		x0 = axmodm(m, a, x0);
	}
}

int main(int argc, char **argv)
{
	if(argc == 1)
	{
		generate(INT_MAX, 16807);//INT_MAX is the maximum int value of the
compiler(32-bit in this case)
		generate(2147483399, 40692);
		generate(2147483563, 40014);
	}
	else if(argv[1][0] == '1')
	{
		coordinates(INT_MAX, 16807);
	}
	else if(argv[1][0] == '2')
	{
		generate17(INT_MAX, 16807);
	}
	return 0;
}
```

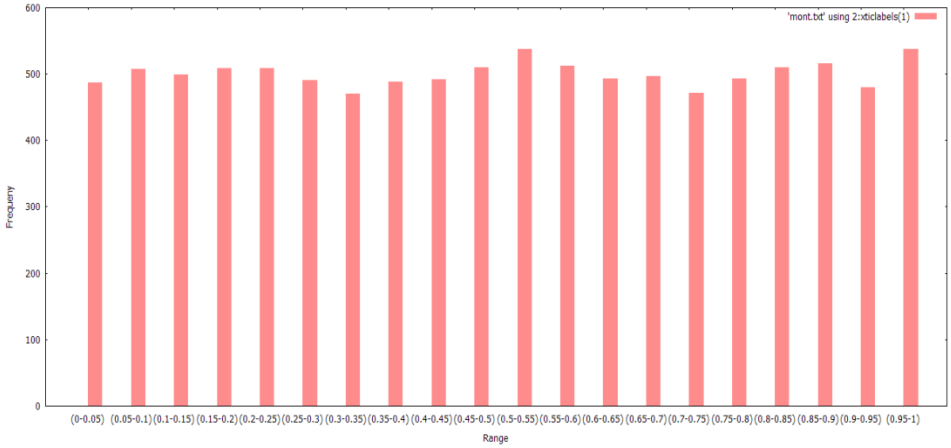
Using the output bar plot were generated:

Using a=16807,b=0, m=2^{31}-1

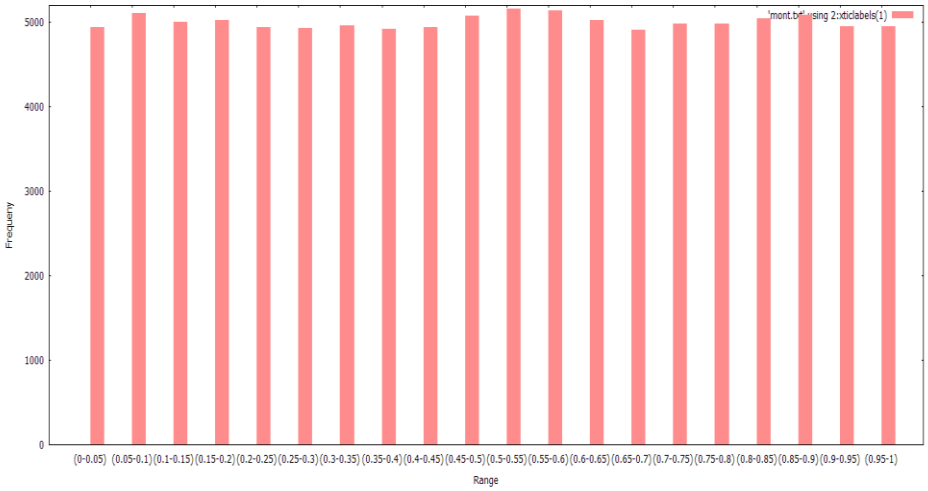
i) For N=1000



ii) For N=10000

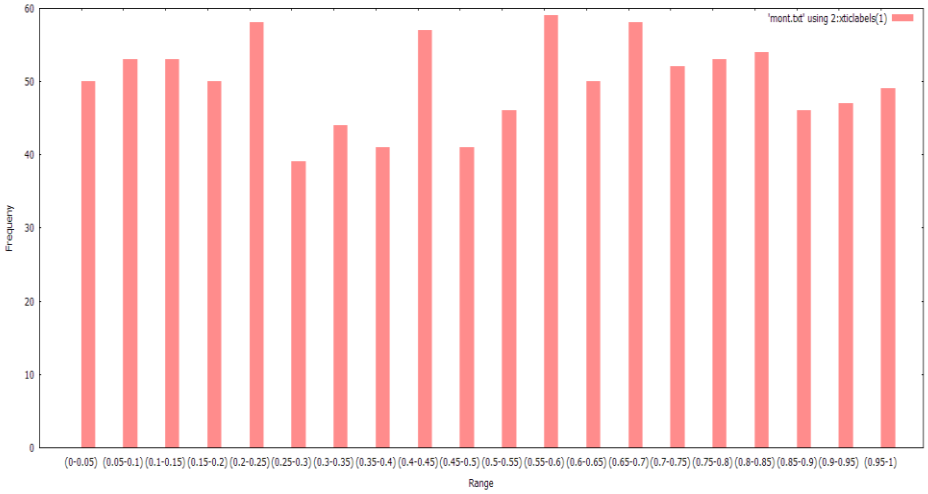


iii) For N=100000

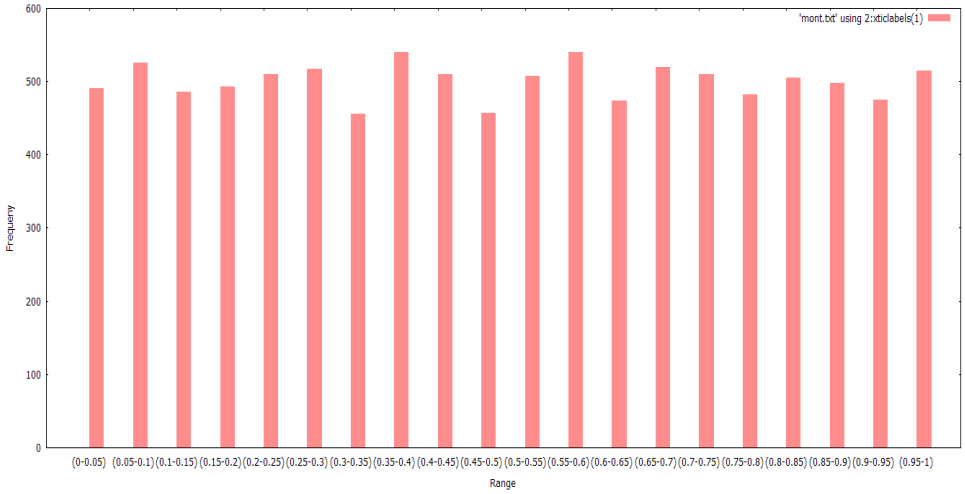


Using a=40692,b=0,m=2147483399.

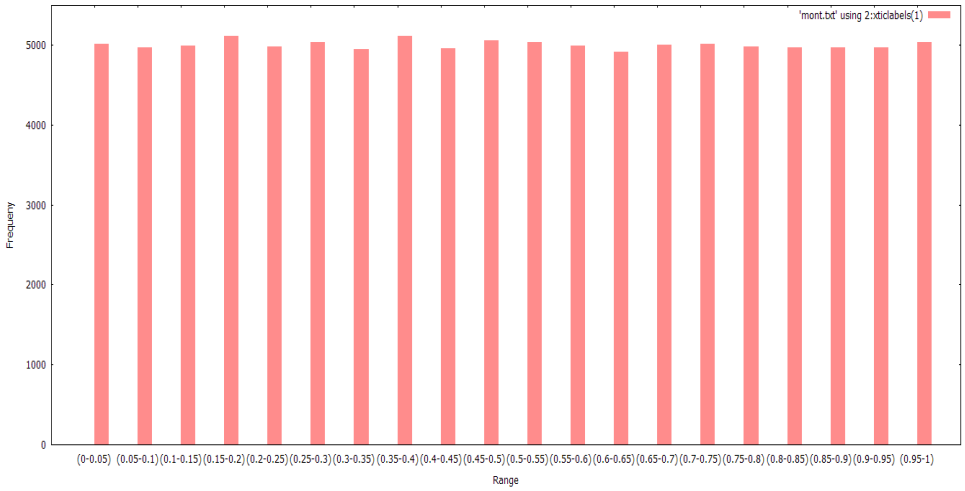
i) For N=1000



ii) For N=10000

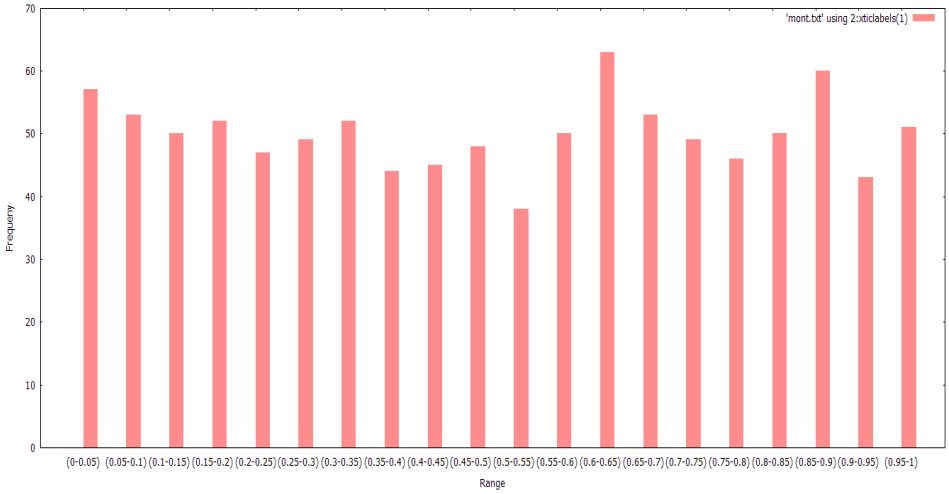


iii) For N=100000

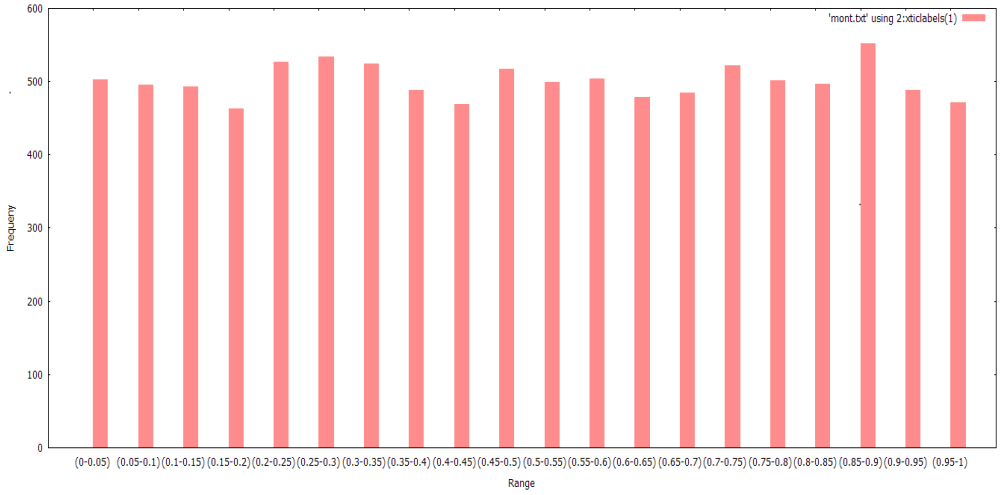


Using a=40014,b=0,m=2147483563

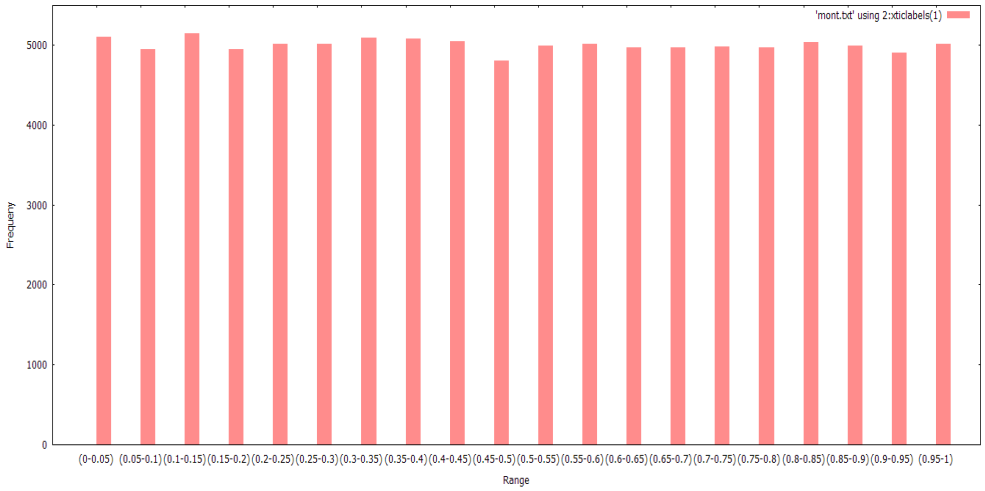
i) For N=1000



ii) For N=10000

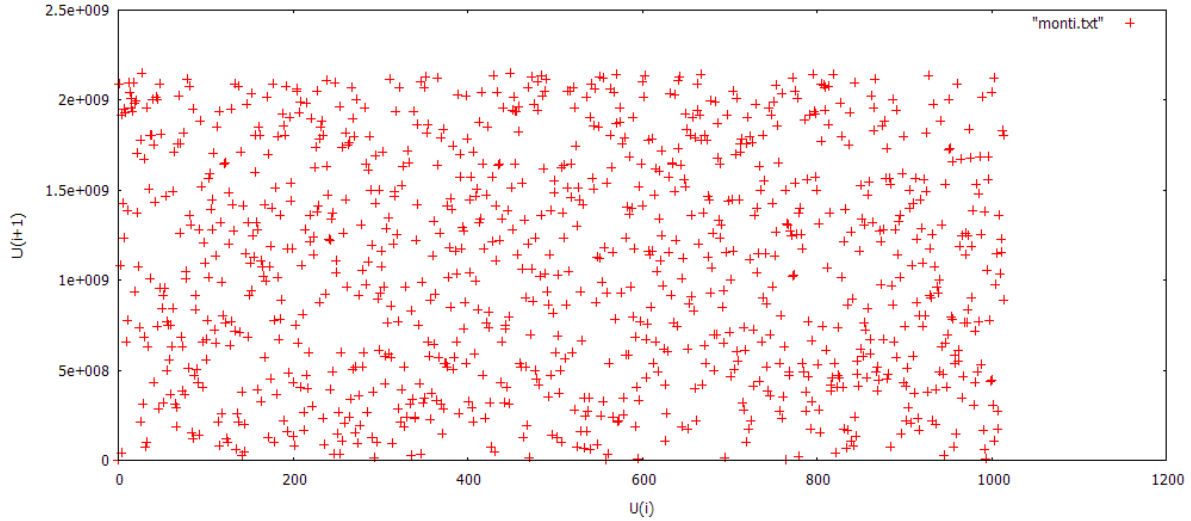


iii) For N=100000

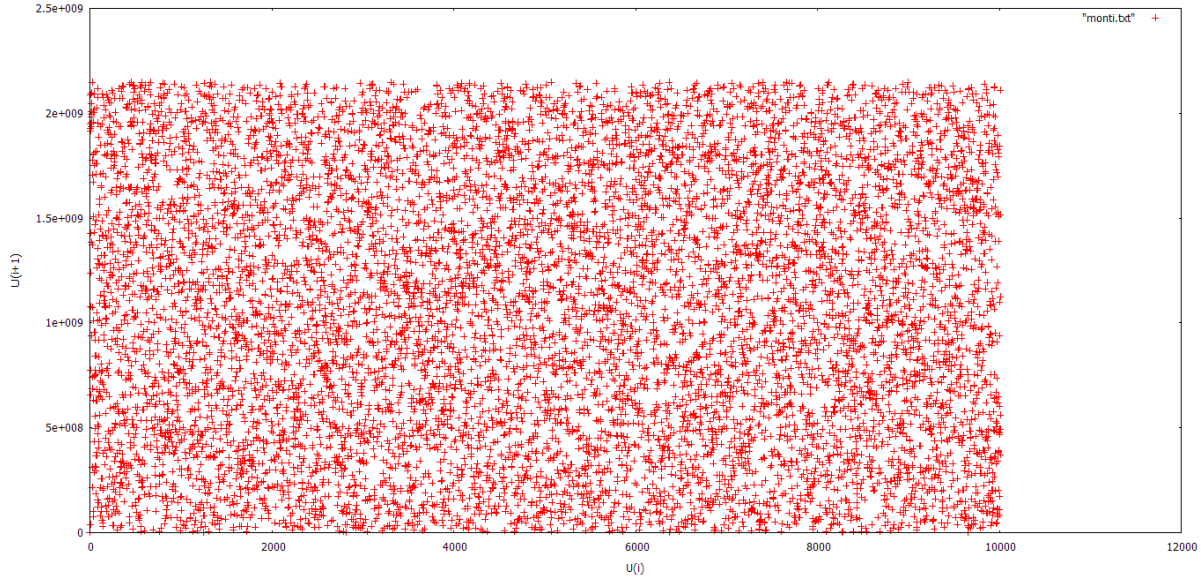


<u>The plot of U_i vs U_{i-1}:</u>

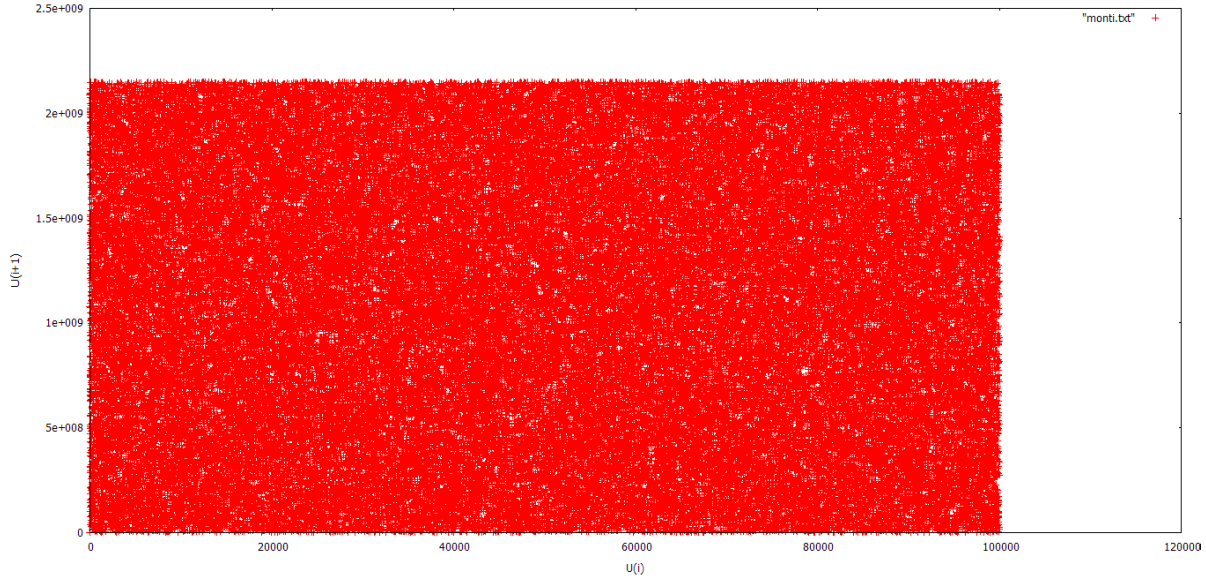
For N=1000



For N=10000



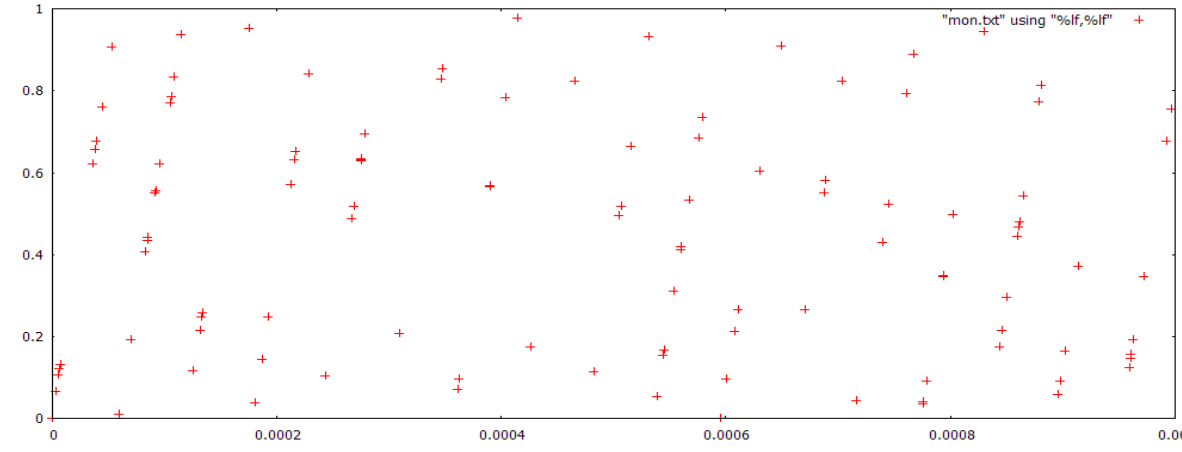
For N=100000



Observations:

From the bar plot we can see that the greater the value of N, the frequency tend to stabilise near a value (N/20).

Magnification of the plot:



Part II:

This part focuses on the use of extended Fibonacci generators to generate nos. First 17 buffer values was generated using the linear congruence generator of the previous problem.

C++ implementation:

```
#include<iostream>
#include<climits>
#include<cmath>
#define N 1000
using namespace std;
unsigned int *u;

unsigned int eFib(int index)
{
	unsigned int ui, temp;
	ui = (u[index-5] + u[index-17]) % (INT_MAX+1);
	return ui;
}

int main(int argc, char **argv)
{
	//declaring variables
	int count, max;
	double mean = 0;
	double variance = 0;
	double temp;
	int f[50];
	double p[50];
	u = new unsigned int[N];
	//taking the starting 17 values
	for(count = 0; count < 17; count++)
	{
		cin>>u[count];
	}
	//u[0] = 2147483647;
	count = 17;
	//generating sequence
	while(count < N)
	{
		u[count] = eFib(count);
		count++;
	}
	if(argv[1][0] == 'c')
	{
		switch(argv[1][1])
		{
			case '1':
				max = 1000;
				break;
			case '2':
				max = 10000;
				break;
			case '3':
				max = 100000;
				break;
		}
		for(count = 1; count < max; count++)
		{
```

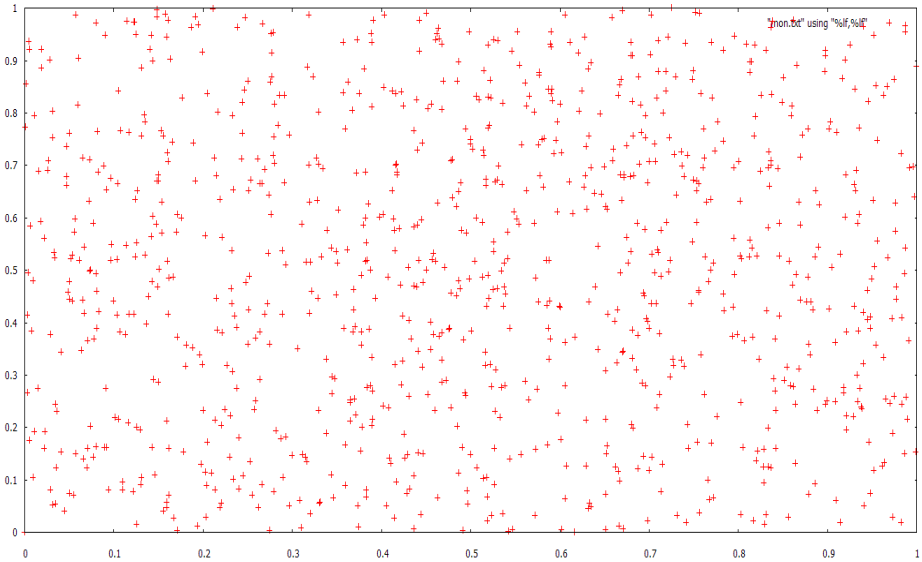
```
                cout<<u[count-1]<<'\t'<<u[count]<<endl;
        }
    }
    else if(argv[1][0] == 'd')
    {
        //calculate mean
        mean = u[0];
        for(count = 1; count < 1000; count++)
        {
            mean = mean + (u[count] - mean*1.0)/(count + 1);
        }
        //calculate variance
        for(count = 0; count < 1000; count++)
        {
            variance = pow(u[count]*1.0 - mean, 2);
        }
        variance = variance/1000.0;
        cout<<(unsigned int)mean<<endl;
        cout<<variance<<endl;
        //calculate probability distribution
        for(count = 0; count < 50; count++)
        {
            f[count] = 0;
        }
        for(count = 0; count < 1000; count++)
        {
            temp = (u[count]*1.0)/pow(2.0, 15);
            temp = temp/pow(2.0, 16);
            f[(int)(temp/0.02)]++;
        }
        double pd[50];
        pd[0] = 0;
        for(count = 0; count < 50; count++)
        {
            p[count] = f[count]/1000.0;
            if(count == 0)
            {
                pd[count] = p[count];
            }
            else
            {
                pd[count] = pd[count-1] + p[count];
            }
            cout<<pd[count]<<endl;
        }
    }
    else if(argv[1][0] == 'e')
    {
        //calculating autocorrelation values
        double temp1 = 0, temp2 = 0;
        double autoc[5];
        for(count = 0; count < 1000; count++)
        {
            temp2 += ((double)u[count] - (double)mean)*((double)u[count] -
(double)mean);
        }
        cout<<temp2<<endl;
        for(max = 0; max < 5; max++)
        {
            temp1 = 0;
            for(count = max+1; count < 1000; count++)
            {
```

```
                        temp1 += 1.0*((double)u[count] -
(double)mean)*((double)u[count-1] - (double)mean);
                }
                autoc[max] = temp1/temp2;
                cout<<temp1<<endl;
        }
        for(count = 0; count < 5; count++)
                cout<<autoc[count]<<endl;
    }
    return 0;
}
```

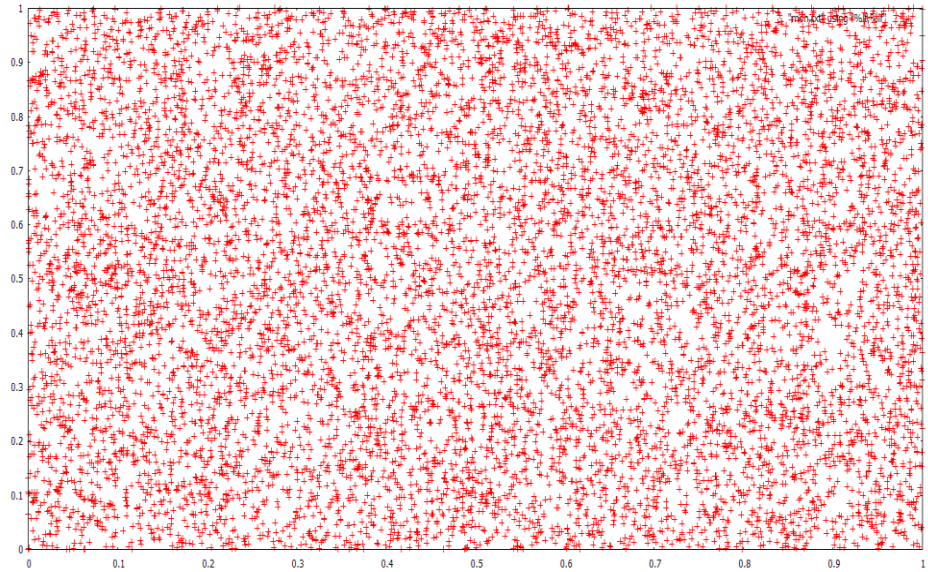
Using the data obtained:

The following plots were obtained;

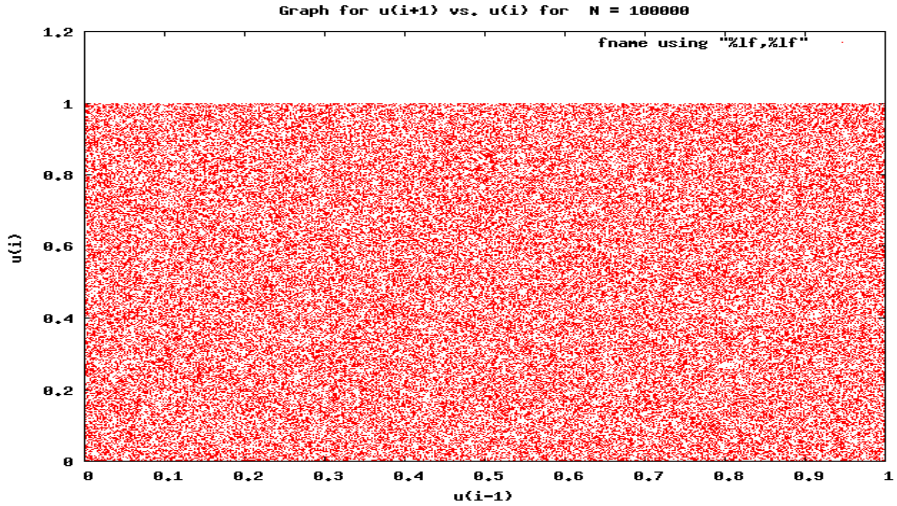
i) For N=1000



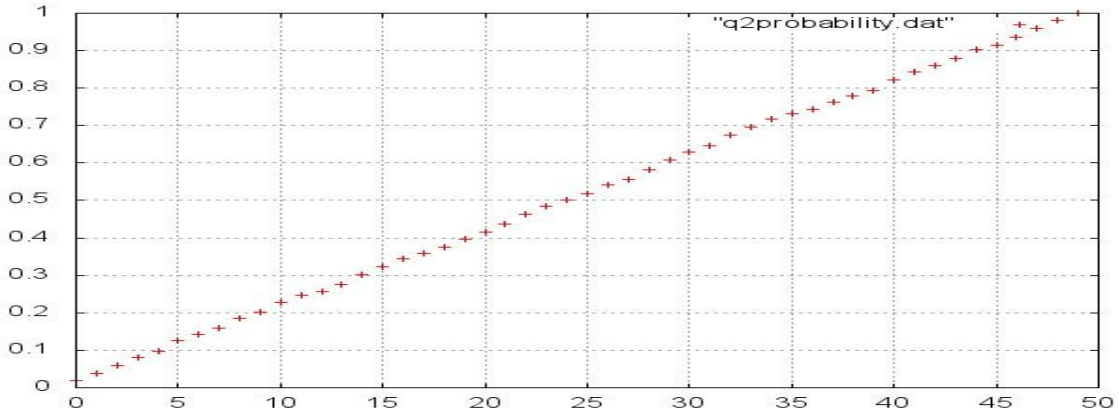
ii) For N=10000



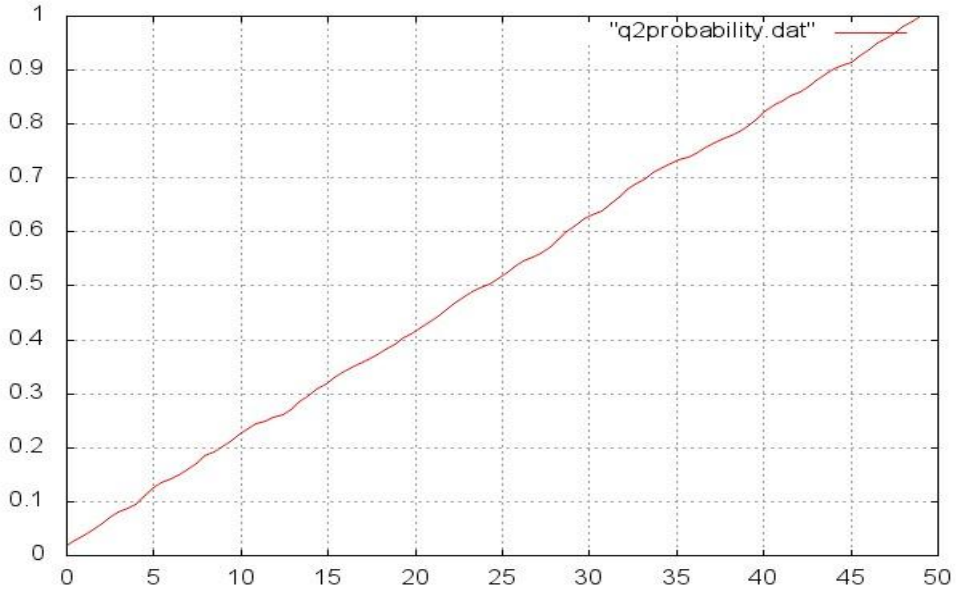
iii) For N=100000



The cumulative probability distribution plot so obtained:



The cumulative probability distribution curve so obtained:



Observations:

The autocorrelation lags calculated from the above program tends very close to zero. However if the numbers are independent it should simply be zero. The Fibonacci generator passes all statistical tests and is certainly a better generator than the LCG.