|  |
| --- |
|  |
| Pseudorandom Number generation: |
| A brief demonstration |
|  |
| **Najee Walls** |
| **5/12/2014** |

|  |
| --- |
|  |

Contents

[Pseudorandom Number generation: A brief demonstration. 2](#_Toc387672190)

[Introduction – trying to create stream of randomness 2](#_Toc387672191)

[Linear congruential generator 2](#_Toc387672192)

[Demonstration – The Program 4](#_Toc387672193)

[Main Menu 4](#_Toc387672194)

[Form #1 Raw Numbers 5](#_Toc387672195)

[Form #2 – Dice Roll 7](#_Toc387672196)

[Code 8](#_Toc387672197)

[Modular Generation Class 8](#_Toc387672198)

[Raw Numbers Form 10](#_Toc387672199)

[Coin flip form 13](#_Toc387672200)

# Pseudorandom Number generation: A brief demonstration.

## Introduction – trying to create stream of randomness

When you see a dice roll on a computer game or a set of cards picked out in a virtual poker game, those numbers you see on the die was not truly chosen randomly. It is not to say that the odds are rigged against you (or maybe they are in a casino) but what I mean to say is that when a computer picks a number it *seems* random but that number was altered from another initial number. The function or algorithm that changes the initial number or “seed” into another number is meant to be hard to follow but at the same time be statistically fair. For example, if some function of f gives me f(2) = 4 and f(3) = 5 and f(4) = 6, then I might be able guess what the function is doing. But if some function of g gives me g(2) = 56 and g(3) = 2420 and g(4) = 3 then it would be hard to find out what the function is doing. With the first function one could see a pattern of the output being 2 more than the input but with the other function it’s very hard to see a pattern. There might actually be a pattern that the computer is using but to the human mind, this lacks a pattern and is seems random. That is why these algorithms that generate these numbers are called pseudorandom Number Generation.

## Linear congruential generator

This is the most popular and easiest to understand Pseudorandom Number so this will be the focus of this report.

With all numbers being integers and a given random number Xn, multiply by multiplier k, add an offset c and then the resulting number is taken to the modulus m. It only needs one initial number to continue to generate numbers indefinitely and the math operations are simple commands that a computer can calculate it in an instant. However the quality of the numbers generated is very dependent on the numbers chosen for k, c, or m. For example when k =2 , c = 1 and m = 10, and the initial number is 2, the sequence generated is,

2,5,1,3,7,5,1,3,7,5

The numbers are alternating in a predictable and unbalanced fashion for a set of numbers meant to alternate between 1 and 10. However when the numbers chosen are k = 11, c = 7, and m = 10 then the sequence generated is,

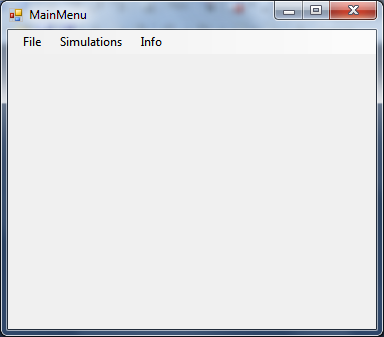
2,9,6,3,0,7,4,1,8,5,2,9

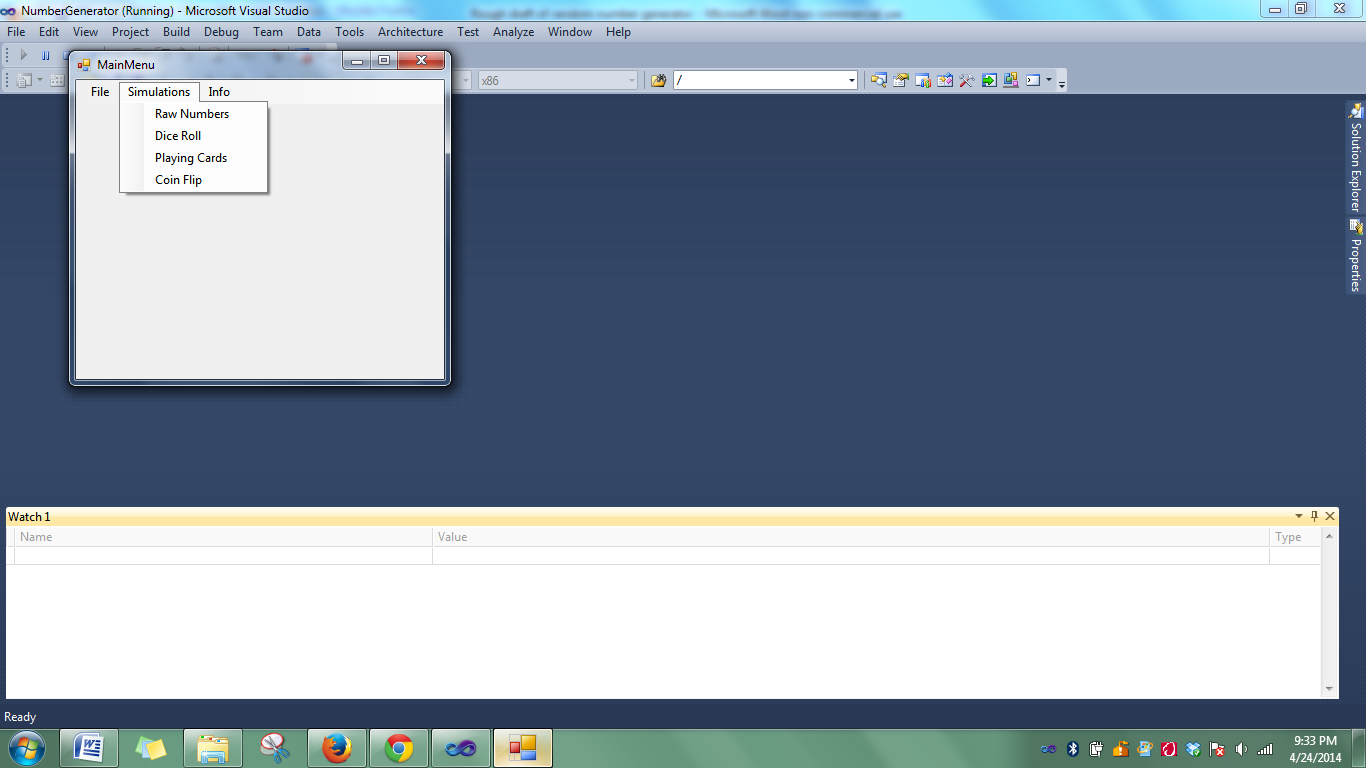
This is a bit harder to predict but is still possible with values this low. And statistically, it hits each number once in the period of 10 numbers (the modular is the limit to how long a chain of numbers can be before it repeats) and once the generator reaches the same number it started with, it repeats itself. This is one of the flaws with this method of generation but in practice the m or modular is a very large number so that a very long chain of numbers can be generated. Another technique to get past this flaw is to make the seed or initial number come from a source outside of the algorithm. For example, the initial number could be the internal clock of the computer. And in some applications the number generated is divided by the modular number itself. In that case the equation looks like this.

The number produced from that calculation would range from 0 to 1 and could be multiplied by any other number to get a range of numbers between that given number and 0. It would be chaotic and long series of numbers but still ultimately bound by a function of some sort.

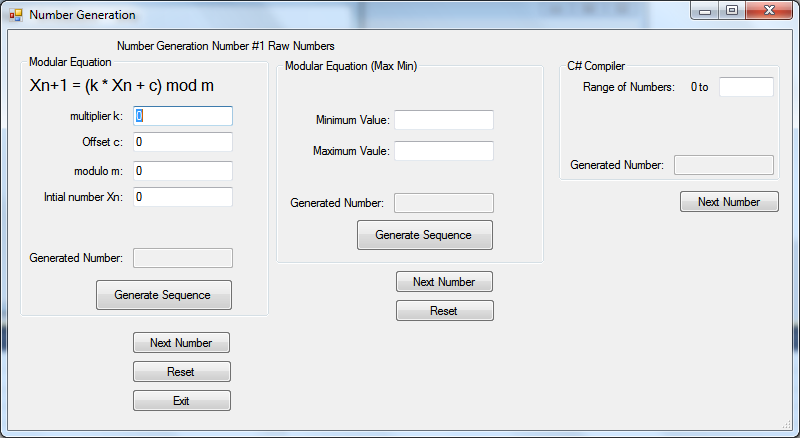
# Demonstration – The Program

## Main Menu

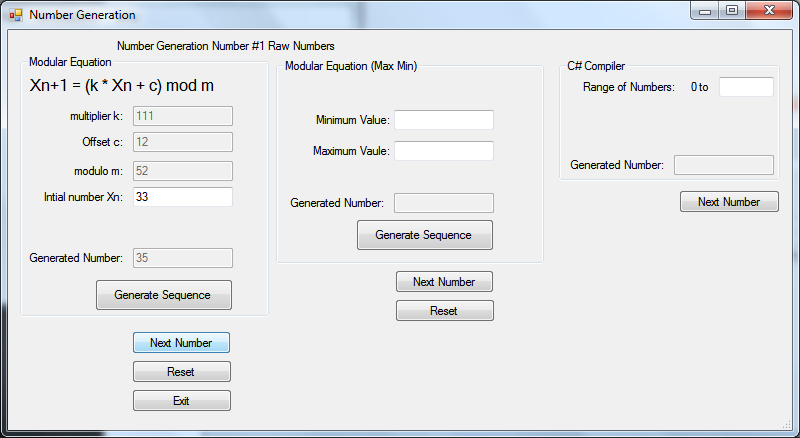


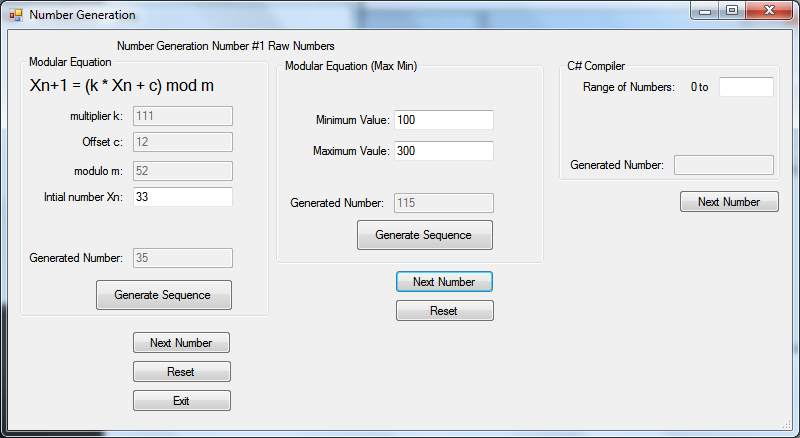


## Form #1 Raw Numbers

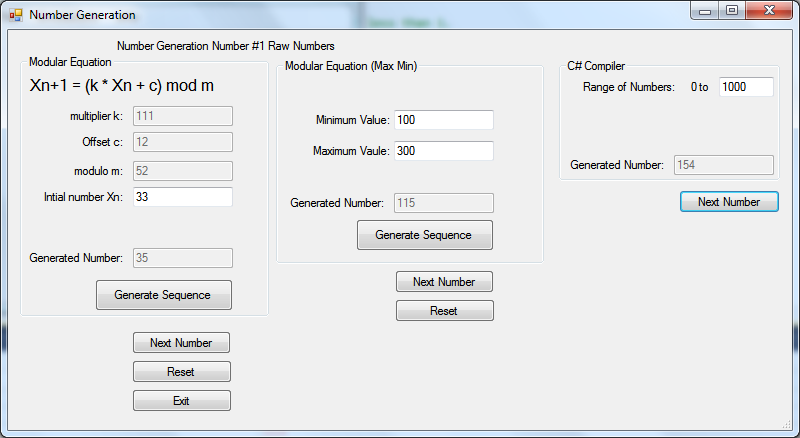


The left most section allows the user to enter in the numbers for the modular equation manually. One the numbers are set, the user can click the next number button to generate the next number in the sequence generated by the user given number.



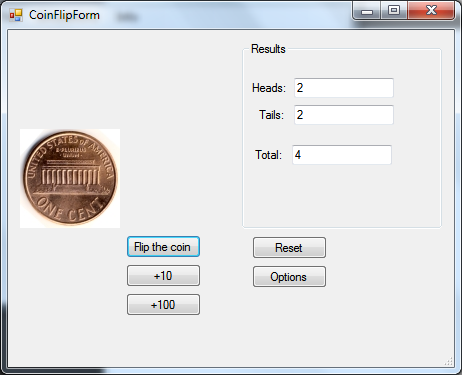


2nd section actually uses a counter that increments on a timer. From the counter, the offset, multiplier and initial number is set. A range of random number can be chosen here as well.



The 3rd section just uses the random function provided by computer itself. It is used as a control and as a comparison.

## Form #2 – Dice Roll



This is a simple application of a Pseudo Random Number Generator that generates a random decimal value between 1 and 0. If the number generated is below 0.5 then heads is shown. Otherwise tails is shown.

# Code

## Modular Generation Class

**using** System**;**

**using** System**.**Collections**.**Generic**;**

**using** System**.**Linq**;**

**using** System**.**Text**;**

**namespace** NumberGenerator

**{**

class Modgeneration

**{**

//Modular Pseudo number generation

//in the form Xn+1 = (k \* Xn + c) mod m

int k**;**

int x**;**

int c**;**

int m**;**

int**[]** numbers**;**

int seed**;**

**public** Modgeneration**()**

**{**

**}**

//

**public** Modgeneration**(**int seed**)**

**{**

/\* Things needed for a full period

\* c and m are relatively prime

\* k - 1 is divisible by all prime factors of m,

\* k - 1 is a multiple of 4 if m is a multiple of 4.

\*/

//2^8 or an "8-bit" number generator.`

m **=** 1024**;**

k **=** **(**seed **\*** 4**)** **+** 1**;**

//Since prime factorization of modular is just 2 eight times, any odd number should do.

c **=** **(**seed **\*** 2**)** **+** 1**;**

x **=** seed**;**

**}**

//Object Constructor. Creates sets varibles that this class will use.

**public** Modgeneration**(**int seed**,** int offset**,** int multiplyer**,** int modular**)**

**{**

//The variables

x **=** seed**;**

c **=** offset**;**

k **=** multiplyer**;**

m **=** modular**;**

**}**

**public** int NextNum**()**

**{**

//x becomes Xn+1

x **=** **((**k **\*** x**)** **+** c**)** **%** m**;**

**return** x**;**

**}**

**public** void Reseed**(**int seed**)**

**{**

//New initial value. Generator still works the same

x **=** seed**;**

**}**

//Should always return a number greater than or equal to 0 but less than 1.

**public** decimal NextDecimal**()**

**{**

decimal d**;**

//(k \* Xn + c) mod m divided by m.

x **=** **((**k **\*** x**)** **+** c**)** **%** m**;**

d = (decimal)x / (decimal)m;

return d;

}

//Property blocks may not be needed for the moment

public int CurrentVal

{

get { return x; }

set { x = value; }

}

public int Offset

{

get { return c; }

set { c = value; }

}

public int Multiplyer

{

get {return k;}

set {k = value;}

}

public int Mod

{

get { return m; }

set { m = value; }

}

}

}

## Raw Numbers Form

**using** System**;**

**using** System**.**Collections**.**Generic**;**

**using** System**.**ComponentModel**;**

**using** System**.**Data**;**

**using** System**.**Drawing**;**

**using** System**.**Linq**;**

**using** System**.**Text**;**

**using** System**.**Windows**.**Forms**;**

**namespace** NumberGenerator

**{**

**public** partial class RawForm **:** Form

**{**

//Declare variables

Random rand **=** **new** Random**();**

bool intialvalueset **=** **false;**

int k**;**

int c**;**

int m**;**

int x**;**

int xnext**;**

int xcurrent**;**

int**[]** lognumberlist**;**

int**[]** compliernumberlist**;**

int max**;**

int min**;**

int counter**;**

Modgeneration mod**;**

**public** RawForm**()**

**{**

InitializeComponent**();**

//start the timer.

timer1**.**Start**();**

**}**

**private** void generateButton\_Click**(object** sender**,** EventArgs e**)**

**{**

//Checks if text boxes are filled out. Give a warning message if they're not.

**if** **(**multiTextBox**.**Text**.**Trim**().**Length **==** 0 **||** offsetTextBox**.**Text**.**Trim**().**Length **==** 0 **||**

modTextBox**.**Text**.**Trim**().**Length **==** 0 **||** initialTextBox**.**Text**.**Trim**().**Length **==** 0**)**

**{**

MessageBox**.**Show**(**"Make sure all boxes are filled"**);**

**}**

**try**

**{**

//Parse text Box string to text

//multiplyer

k **=** int**.**Parse**(**multiTextBox**.**Text**);**

//offset/increment

c **=** int**.**Parse**(**offsetTextBox**.**Text**);**

//Modular

m **=** int**.**Parse**(**modTextBox**.**Text**);**

//Intial value

x **=** int**.**Parse**(**initialTextBox**.**Text**);**

mod **=** **new** Modgeneration**(**x**,**c**,**k**,**m**);**

//Make textboxes inactive until reset

multiTextBox**.**Enabled **=** **false;**

offsetTextBox**.**Enabled **=** **false;**

modTextBox**.**Enabled **=** **false;**

//initialTextBox.Enabled = false;

**}**

**catch**

**{**

MessageBox**.**Show**(**"Please only use integers for formula variables"**);**

**}**

**}**

**private** void resetButton\_Click**(object** sender**,** EventArgs e**)**

**{**

//Make textboxes inactive until reset

multiTextBox**.**Enabled **=** **true;**

offsetTextBox**.**Enabled **=** **true;**

modTextBox**.**Enabled **=** **true;**

**}**

//Generate Next Number in sequence.

**private** void nextButton1\_Click**(object** sender**,** EventArgs e**)**

**{**

**try**

**{**

generatedTextBox**.**Text **=** mod**.**NextNum**().**ToString**();**

**}**

**catch**

**{**

MessageBox**.**Show**(**"Sequence not generated"**);**

**}**

**}**

//Get compliler/computer to generate it's random number from it's pre-defined random function.

**private** void nextButton2\_Click**(object** sender**,** EventArgs e**)**

**{**

**try**

**{**

//Takes number from max number textbox,parses it to integer for random function,

//then spits it back out to string into another textbox.

complieTextBox**.**Text **=** rand**.**Next**(**int**.**Parse**(**maxTextBox**.**Text**)).**ToString**();**

**}**

**catch**

**{**

MessageBox**.**Show**(**"Integers only please"**);**

**}**

**}**

**private** void exitButton\_Click**(object** sender**,** EventArgs e**)**

**{**

**this.**Close**();**

**}**

**private** void generateButton2\_Click**(object** sender**,** EventArgs e**)**

**{**

int seed**;**

decimal y**;**

//Parse text Box string to text

max **=** int**.**Parse**(**modMaxTextBox**.**Text**);**

min **=** int**.**Parse**(**modMinTextBox**.**Text**);**

//Use timer as the seed value.

mod **=** **new** Modgeneration**(**counter**);**

**}**

**private** void nextButton3\_Click**(object** sender**,** EventArgs e**)**

**{**

decimal y**;**

//Multiply the output of the number generator by the min, then add the difference;

y **=** **(**mod**.**NextDecimal**()** **\*** **(**max **-** min**))** **+** min **;**

//round to nearest int

y **=** **(**int**)**y**;**

generatedTextBox2**.**Text **=** y**.**ToString**();**

**}**

**private** void timer1\_Tick**(object** sender**,** EventArgs e**)**

**{**

counter**++;**

**if** **(**counter **>** 1024**)**

**{**

counter **=** 0**;**

**}**

**}**

**}**

**}**

## Coin flip form

**using** System**;**

**using** System**.**Collections**.**Generic**;**

**using** System**.**ComponentModel**;**

**using** System**.**Data**;**

**using** System**.**Drawing**;**

**using** System**.**Linq**;**

**using** System**.**Text**;**

**using** System**.**Windows**.**Forms**;**

**namespace** NumberGenerator

**{**

**public** partial class CoinFlipForm **:** Form

**{**

**public** CoinFlipForm**()**

**{**

InitializeComponent**();**

**}**

**private** GifImage gifImage **=** **null;**

**private** string filePath **=** @"~/Flipping\_Coin.gif"**;**

int headcounter **=** 0**;**

int tailcounter **=** 0**;**

Random rand**;**

//When flip button is pressed, if a random number between 0 and 1 is less then .5 then heads shows

//Else tails shows. Counts of each is tracked.

**private** void flipButton\_Click**(object** sender**,** EventArgs e**)**

**{**

timer1**.**Start**();**

flipPictureBox**.**Visible **=** **true;**

pictureBoxheads**.**Visible **=** **false;**

pictureBoxtails**.**Visible **=** **false;**

rand **=** **new** Random**();**

double x **=** rand**.**NextDouble**();**

**if** **(**x **<** 0.5**)**

**{** pictureBoxheads.Visible = true;

headcounter++;

}

else

{

pictureBoxtails.Visible = true;

tailcounter++;

}

headTextBox.Text = headcounter.ToString();

tailTextBox.Text = tailcounter.ToString();

counterBox.Text = (headcounter + tailcounter).ToString();

}

private void timer1\_Tick(object sender, EventArgs e)

{ flipPictureBox.Visible = false;

timer1.Stop();

}

}

}