Generating Pseudorandom Numbers From Various Distributions Using C++

by Robert J. Yager

Approved for public release; distribution is unlimited.
NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.
Generating Pseudorandom Numbers From Various Distributions Using C++

by Robert J. Yager
Weapons and Materials Research Directorate, ARL
Generating Pseudorandom Numbers From Various Distributions Using C++

This report documents a set of functions, written in C++, that can be used to generate pseudorandom numbers that have either uniform or normal distributions and pseudorandom integers that have either uniform or Poisson distributions. An implementation of the Mersenne twister algorithm, developed by Matsumoto and Nishimura, is included. The output from the Mersenne twister is used to generate the various distributions through the use of assorted transformation algorithms.

Robert J. Yager

Approved for public release; distribution is unlimited.

pseudorandom, algorithm, Mersenne twister, C++, normal, uniform, Poisson

ii
# Contents

## Acknowledgments

v

1. **Introduction**  

2. **Generating Pseudorandom Integers Using the Mersenne Twister 19937 Algorithm – The Rand() Function**  
   2.1 Rand() Code ................................................................................................... 2  
   2.2 Rand() Template Classes .................................................................................. 2  
   2.3 Rand() Parameters .......................................................................................... 2  
   2.4 Rand() Return Value ....................................................................................... 2  
   2.5 Rand() Example ............................................................................................ 2  
   2.6 Comparison of the Rand() Function to Other Mersenne Twister Implementations...... 3

3. **Initializing the Mersenne Twister – The Initialize() Function**  
   3.1 Initialize() Code ........................................................................................... 4  
   3.2 Initialize() Template Classes ......................................................................... 4  
   3.3 Initialize() Parameters .................................................................................. 5

4. **Generating Uniformly Distributed Pseudorandom Numbers – The RandU() Function**  
   4.1 RandU() Code ............................................................................................... 5  
   4.2 RandU() Template Classes ............................................................................. 6  
   4.3 RandU() Parameters ...................................................................................... 6  
   4.4 RandU() Return Value ................................................................................... 6  
   4.5 RandU() Simple Example ............................................................................. 6  
   4.6 RandU() Binning Example ............................................................................. 6

5. **Generating Normally Distributed Pseudorandom Numbers – The RandN() Function**  
   5.1 RandN() Code ............................................................................................... 8  
   5.2 RandN() Template Classes ............................................................................. 8  
   5.3 RandN() Parameters ...................................................................................... 8  
   5.4 RandN() Return Value ................................................................................... 9
6. Generating Uniformly Distributed Pseudorandom Integers – The RandI() Function 11
   6.1 RandI() Code .................................................................11
   6.2 RandI() Template Classes ..................................................11
   6.3 RandI() Parameters ...........................................................11
   6.4 RandI() Return Value ..........................................................11
   6.5 RandI() Simple Example .......................................................11
   6.6 RandI() Binning Example ......................................................12

7. Generating Poisson-Distributed Pseudorandom Integers – The RandP() Function 13
   7.1 RandP() Code .................................................................14
   7.2 RandP() Template Classes .....................................................14
   7.3 RandP() Parameters ...........................................................14
   7.4 RandP() Return Value ..........................................................14
   7.5 RandP() Simple Example .......................................................14
   7.6 RandP() Binning Example ......................................................15

8. Code Summary 16

9. References 18

Distribution List 19
Acknowledgments

I would like to thank Mr. Benjamin Flanders and Dr. Benjamin Breech of the U.S. Army Research Laboratory’s Weapons and Materials Research Directorate. Mr. Flanders and Dr. Breech provided technical and editorial recommendations that improved the quality of this report.
INTENTIONALLY LEFT BLANK.
1. Introduction

This report documents a set of functions, written in C++, that can be used to generate pseudorandom numbers that have either uniform or normal distributions and pseudorandom integers that have either uniform or Poisson distributions. An implementation of the Mersenne twister algorithm, developed by Matsumoto and Nishimura (1), is included. The output from the Mersenne twister is used to generate the various distributions through the use of assorted transformation algorithms.

The functions presented here are offered as compiler-independent alternatives to functions defined by the new C++11 standard (2). Although the new standard defines functions for generating pseudorandom numbers with uniform distributions, normal distributions, Poisson distributions, etc., it does not specify which algorithms will be used to generate those pseudorandom numbers. Thus, compilers that conform to the new standard will all generate pseudorandom numbers from the various distributions, but the actual numbers that are generated may differ from compiler to compiler.

The functions presented in this report have been grouped into the yRandom namespace, which is summarized at the end of this report.

2. Generating Pseudorandom Integers Using the Mersenne Twister 19937 Algorithm – The Rand() Function

The Rand() function uses the Mersenne twister 19937 algorithm to generate uniformly distributed pseudorandom integers in the interval \([0,2^{32})\). According to Matsumoto and Nishimura, the algorithm has a period of \(2^{19937}–1\) and passes the Diehard tests for statistical randomness.

The state of the Mersenne twister is stored in an array of 625 32-bit unsigned integers, which is passed as an argument to the Rand() function. The initial state of the Mersenne twister can be set using the Initialize() function (see section 3). The Initialize() function uses a user-supplied 32-bit integer to seed a simple pseudorandom number generator that generates the initial state integers. This effectively allows for the creation of multiple pseudorandom number generators (as many as \(2^{32}\)) that can each produce independent, reproducible sequences of pseudorandom integers.
The code contained in the Rand() function differs from the code that is presented by Matsumoto and Nishimura. To avoid using the modulo operator, their code calculates all of the algorithm’s 624 unsigned state integers once out of every 624 function calls. In contrast, the Rand() function uses the ternary operator to avoid using the modulo operator.

2.1 Rand() Code

```cpp
template<typename T>
T Rand(){
    // STATE OF MERSENNE TWISTER (FOR T, USE A 32-BIT UNSIGNED INT)
    C T i=I[624], j=i<623?i+1:0, y=i[i] & 0x80000000 | I[j] & 0x7fffffff;
    y=i[i]=I[i<227?i+397:i-227]^y>>1^(y&1)*0x9908b0df, I[624]=j;
    return (y^=(y^=(y^=y>>11)<<7&0x9d2c5680)<<15&0xefc60000)>>18;
}
```

2.2 Rand() Template Classes

T should be a 32-bit unsigned integer type. Greater than 32-bit unsigned integer types can be used, but performance may suffer.

2.3 Rand() Parameters

I points to a 625-element array that contains the state of the Mersenne twister algorithm. Each time the Rand() function is called, the array pointed to by I is modified. The initial state of the array should be set using the Initialize() function (see section 3).

2.4 Rand() Return Value

The Rand() function returns uniformly distributed pseudorandom integers in the interval \([0, 2^{32})\).

2.5 Rand() Example

The following example uses the Rand() function to generate one billion pseudorandom integers. The Initialize() function, introduced in section 3, is used to set the initial state of the Mersenne twister.

```cpp
#include <cstdio>
#include <ctime>
#include "y_random.h"

int main(){
    unsigned I[625];
    yRandom::Initialize(I, 1);
    for(int i=1; i<1000000000; ++i) yRandom::Rand(I);
    printf(“10^9 pseudorandom numbers generated in %.3f seconds. \nThe 10^9th number is %u.\n”, double(clock())/CLOCKS_PER_SEC, yRandom::Rand(I));
}
```
2.6 Comparison of the Rand() Function to Other Mersenne Twister Implementations

The following example compares the Rand() function to Matsumoto and Nishimura’s genrand() function, as well as to the C++11 built-in Mersenne twister implementation. The example code demonstrates that the three implementations produce identical output (at least for the first $10^9$ values and with a seed value of 1). Note that time values will vary based on computer specifications, compiler, compiler settings, etc.

```c
#include <stdio>
#include <ctime>
#include <random>
#include "matsumoto_nishimura.h"
#include "y_random.h"

int main(){
    double t=0; //elapsed time
    init_genrand(1); //initialize the Matsumoto-Nishimura implementation
    for(int i=1;i<1000000000;++i)genrand();
    printf("Using Matsumoto and Nishimura's genrand():\n");
    printf("  10^9 pseudorandom integers generated in %.3f seconds.\n  The 10^9th\n  number is %u.\n",(clock()-t)/CLOCKS_PER_SEC,genrand());

    std::mt19937 g(1); //initialize the C++11 built-in implementation
    for(int i=1;i<1000000000;++i)g();
    printf("Using std::mt19937: \n");
    printf("  10^9 pseudorandom integers generated in %.3f seconds.\n  The 10^9th\n  number is %u.\n",(clock()-t)/CLOCKS_PER_SEC,g());

    unsigned I[625]; /*yRandom::Initialize(I,1); init. yRandom implementation
    for(int i=1;i<1000000000;++i)yRandom::Rand(I);
    printf("Using yRandom::Rand():\n");
    printf("  10^9 pseudorandom integers generated in %.3f seconds.\n  The 10^9th\n  number is %u.\n",(clock()-t)/CLOCKS_PER_SEC,yRandom::Rand(I));

    yRandom::Initialize(I,1),init_genrand(1),g.seed(1);
    bool check=true;
    for(int i=0;i<1000000000;++i){
        unsigned x=yRandom::Rand(I);
        if(x!=genrand()||x!=g())check=false;
        printf("\nAre the first 10^9 pseudorandom integers generated by\n  Matsumoto and Nishimura's genrand(), std::mt19937, and yRandom::Rand() identical? %s\n",check?"YES":"NO");
    }
}
```

OUTPUT:

```
10^9 pseudorandom numbers generated in 1.778 seconds.
The 10^9th number is 2716480233.
```
OUTPUT:

Using Matsumoto and Nishimura's genrand():
   10^9 pseudorandom integers generated in 3.400 seconds.
   The 10^9th number is 2716480233.

Using std::mt19937:
   10^9 pseudorandom integers generated in 2.231 seconds.
   The 10^9th number is 2716480233.

Using yRandom::Rand():
   10^9 pseudorandom integers generated in 1.857 seconds.
   The 10^9th number is 2716480233.

Are the first 10^9 pseudorandom integers generated by
   Matsumoto and Nishimura's genrand(), std::mt19937,
   and yRandom::Rand() identical?  YES

3. Initializing the Mersenne Twister – The Initialize() Function

The Initialize() function uses an algorithm presented by Nishimura and Matsumoto (3) to
initialize the 625-element array that is used to store the state of the Mersenne twister algorithm.

3.1 Initialize() Code

```cpp
#include <iostream>

template<class T>
void Initialize()
{
    // INITIALIZE STATE OF MERSENNE TWISTER
    T I[625], // STATE OF MERSENNE TWISTER (FOR T, USE A 32-BIT UNSIGNED INT)
    unsigned long s;

    s=1812433253;
    I[0]=s&0xffffffff, I[624]=0;
    for(int i=1;i<624;++i)
    {
        I[i]=(1812433253*(I[i-1]^I[i-1]>>30)+i)&0xffffffff;
    }

    // YAGENAUT@GMAIL.COM
    // LAST UPDATED: 21 JAN 2014
}
```

3.2 Initialize() Template Classes

T  T should be a 32-bit unsigned integer type. Greater than 32-bit types can be used,
   but performance may suffer.
3.3 Initialize() Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>Points to storage for a 625-element array that is used to store the state of the Mersenne twister algorithm.</td>
</tr>
<tr>
<td>s</td>
<td>Specifies the seed for the algorithm that is used to generate the initial state of the Mersenne twister algorithm. s can be any value in the interval ([0,2^{32})).</td>
</tr>
</tbody>
</table>

4. Generating Uniformly Distributed Pseudorandom Numbers – The RandU() Function

The RandU() function can be used to generate uniformly distributed pseudorandom numbers that conform to the probability density function given by equation 1.

\[
f(x) = \begin{cases} \frac{1}{b-a}, & \text{for } a < x \leq b \\ 0, & \text{otherwise} \end{cases}
\]

(1)

Note that the C++ standards document defines a slightly different probability density function for uniformly distributed pseudorandom numbers \((a \leq x < b)\) rather than \(a < x \leq b\). The decision to exclude the lower bound from the distribution rather than the upper bound was based on the belief that it would be more likely to help the user to avoid a divide-by-zero error.

The RandU() function uses the transformation presented in equation 2, where \(r\) is a uniformly distributed pseudorandom number in the interval \((0,1]\).

\[
x = a + (b - a)r 
\]

(2)

Equation 3 can be used to generate \(r\) given \(q\), a uniformly distributed pseudorandom integer in the interval \([0,2^{32})\).

\[
r = \frac{q + 1}{2^{32}} 
\]

(3)

4.1 RandU() Code

```cpp
template<class T> double RandU(T I[625], double a, double b){
    return a+(b-a)*(Rand(I)+1.)/4294967296; // for a=0 and b=1, (0,1]
}
```

---

5
4.2 RandU() Template Classes

T should be a 32-bit unsigned integer type. Greater than 32-bit types can be used, but performance may suffer.

4.3 RandU() Parameters

I points to an array that contains the state of the Mersenne twister algorithm.

a specifies $a$, the lower bound for the distribution.

b specifies $b$, the upper bound for the distribution.

4.4 RandU() Return Value

The RandU() function returns uniformly distributed pseudorandom numbers in the interval $(a,b]$. Note that, due to limitations inherent in the storage of double precision numbers, the return value is not guaranteed to be distinct from $a$ in all cases (such as when $|a| >> |a-b|$).

4.5 RandU() Simple Example

The following example uses the RandU() function to generate and sum one billion uniformly distributed pseudorandom numbers in the interval $(2,6]$.

```c
#include <cstdio>
#include <ctime>
#include "y_random.h"

int main(){
    unsigned I[625];/*<
    yRandom::Initialize(I,1); //....state of Mersenne twister
    double s=0; /*<
    for(int i=0;i<100000000;++i)s+=yRandom::RandU(I,2,6);
    printf("%d\^9 pseudorandom numbers generated and summed in %.3f seconds.\n" "Their average is %f.\n\n",double(clock())**CLOCKS_PER_SEC,s/100000000);
}
```

OUTPUT:

```
10^9 pseudorandom numbers generated and summed in 8.255 seconds.
Their average is 3.999977.
```

4.6 RandU() Binning Example

The following example uses the RandU() function to generate and bin one billion uniformly distributed pseudorandom numbers in the interval $(2,6]$.
```c
#include <cstdio>
#include "y_random.h"

yRandom::Initialize(I,1);

int main(){
  unsigned I[625];
  yRandom::RandU(I,2,6);
  for(double x=yRandom::RandU(I,2,6);x>B[j]&&j<N;++j);
}
```

```plaintext
OUTPUT:

<table>
<thead>
<tr>
<th>BIN</th>
<th>COUNT (RAW)</th>
<th>COUNT (EXPECTED)</th>
<th>%DIFF</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;= 2.0</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>(2.0 to 2.4]</td>
<td>99999019</td>
<td>100000000.0</td>
<td>0.010%</td>
</tr>
<tr>
<td>(2.4 to 2.8]</td>
<td>100010759</td>
<td>100000000.0</td>
<td>0.0108%</td>
</tr>
<tr>
<td>(2.8 to 3.2]</td>
<td>99997646</td>
<td>100000000.0</td>
<td>0.0024%</td>
</tr>
<tr>
<td>(3.2 to 3.6]</td>
<td>100006913</td>
<td>100000000.0</td>
<td>0.0069%</td>
</tr>
<tr>
<td>(3.6 to 4.0]</td>
<td>100010417</td>
<td>100000000.0</td>
<td>0.0104%</td>
</tr>
<tr>
<td>(4.0 to 4.4]</td>
<td>999979386</td>
<td>100000000.0</td>
<td>0.0206%</td>
</tr>
<tr>
<td>(4.4 to 4.8]</td>
<td>99993880</td>
<td>100000000.0</td>
<td>0.0061%</td>
</tr>
<tr>
<td>(4.8 to 5.2]</td>
<td>99999652</td>
<td>100000000.0</td>
<td>0.003%</td>
</tr>
<tr>
<td>(5.2 to 5.6]</td>
<td>100004658</td>
<td>100000000.0</td>
<td>0.0047%</td>
</tr>
<tr>
<td>(5.6 to 6.0]</td>
<td>99997670</td>
<td>100000000.0</td>
<td>0.0023%</td>
</tr>
<tr>
<td>&gt; 6.0</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
</tr>
</tbody>
</table>
```
5. Generating Normally Distributed Pseudorandom Numbers – The RandN() Function

The RandN() function can be used to generate normally distributed pseudorandom numbers that conform to the probability density function given by equation 4, where $\mu$ is the mean of the distribution and $\sigma$ is the standard deviation.

$$f(x) = \frac{1}{\sigma \sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$  \hspace{1cm} (4)

The RandN() function uses the Box-Muller transform (4) to generate normally distributed pseudorandom numbers:

$$x = \mu + \sigma \sqrt{-2\ln(r_a)} \cos(2\pi r_b)$$  \hspace{1cm} (5)

where $r_a$ and $r_b$ are uniformly distributed pseudorandom numbers in the interval (0,1]. Equation 3 can be used to generate $r_a$ and $r_b$ given $q_a$ and $q_b$, two uniformly distributed pseudorandom integers in the interval [0,2^{32}).

5.1 RandN() Code

```cpp
template<class T> double RandN(/\*=====NORMALLY DISTRIBUTED PSEUDORANDOM DOUBLE I[625],/\*--STATE OF MERSENNE TWISTER (FOR T, USE A 32-BIT UNSIGNED INT) double m,double s{}/\*----------MEAN & STANDARD DEVIATION OF DISTRIBUTION return m+s*sqrt(-2*log((Rand(I)+1.)/4294967296)) *cos(1.4629180792671596E-9*(Rand(I)+1.));\*/~~~YAGENAUT@GMAIL.COM~~~~~~~~~~~~~~~~~~~~~~~~~LAST~UPDATED~21JAN2014~~~~~~}
```

5.2 RandN() Template Classes

T T should be a 32-bit unsigned integer type. Greater than 32-bit types can be used, but performance may suffer.

5.3 RandN() Parameters

I I points to an array that contains the state of the Mersenne twister algorithm.

m m specifies $\mu$, the mean of the distribution.

s s specifies $\sigma$, the standard deviation of the distribution.
5.4 RandN() Return Value
The RandN() function returns normally distributed pseudorandom numbers.

5.5 RandN() Simple Example
The following example uses the RandN() function to generate and sum one billion normally distributed pseudorandom numbers with $\mu = 4$ and $\sigma = 0.5$.

```c
#include <cstdio>
#include <ctime>
#include "y_random.h"

int main(){
//<==================SIMPLE EXAMPLE FOR THE yRandom::RandN() FUNCTION
unsigned I[625];/*<*/yRandom::Initialize(I,1);///<.state of Mersenne twister
double s=0;///<*/for(int i=0;i<1000000000;++i)s+=yRandom::RandN(I,4,.5);
printf("10^9 pseudorandom numbers generated and summed in %.3f seconds.\n" "Their average is %.f.\n\n",(double)(clock())/CLOCKS_PER_SEC,s/1000000000);
}///~~~~~~YAGENAUT@GMAIL.COM~~~~~~~~~~~~~~~~~~~~~~~~LAST~UPDATED~21JAN2014~~~~~~
```

OUTPUT:

```
10^9 pseudorandom numbers generated and summed in 46.213 seconds.
Their average is 3.999998.
```

5.6 RandN() Binning Example
The following example uses the RandN() function to generate and bin one billion normally distributed pseudorandom numbers with $\mu = 4$ and $\sigma = 0.5$. The Erf() function is an implementation of an algorithm presented by Abramowitz and Stegun (5).


```c
#include <cstdio>
#include "y_random.h"

#include "y_random.h"

inline double Erf(double x){
    double t=1/(1+.3275911*fabs(x));
    double a[]={.254829592,-.284496736,1.421413741,-1.453152027,1.061405429};
    double s=0;
    for (int i=0;i<5;++i)s+=a[i]*pow(t,i+1);
    return (x<0?-1:1)*(1-s*exp(-x*x));
}

int main(){
    unsigned I[625];
    yRandom::Initialize(I,1);
    int M=100000000;
    const int N=11;
    double B[N]=
        {4.0/M,8.0/M,12.0/M,16.0/M,20.0/M,24.0/M,28.0/M,32.0/M,36.0/M,40.0/M,44.0/M};
    double E[N+1]={};
    E[0]=E[N]=.5*(1+Erf((B[0]-4)/sqrt(2*.5*.5)))*M;
    for (int i=1;i<N;++i)
        E[i]=(.5*(1+Erf((B[i]-4)/sqrt(2*.5*.5)))-.5*(1+Erf((B[i-1]-4)/sqrt(2*.5*.5))))*M;
    int C[N+1];
    for (int i=0;i<N+1;++i)C[i]=0;
    for (double x=yRandom::RandN(I,4,.5);x>B[0];++i)++C[0];
    for (double x=yRandom::RandN(I,4,.5);x>B[0];++i,++C[i],i=0)
        for (double x=yRandom::RandN(I,4,.5);x>B[0];++i,++C[i],i=0)
            if (i==0)printf("%d
",C[0]);
            else if (i==N)printf("%d
",C[N-1]);
            else printf("%d
",C[i]);
    printf("%d
",E[0]-C[0]);
}
```

OUTPUT:

```

<table>
<thead>
<tr>
<th>BIN</th>
<th>COUNT (RAW)</th>
<th>COUNT (EXPECTED)</th>
<th>%DIFF</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;= 2.0</td>
<td>31839</td>
<td>31686.0</td>
<td>0.4827%</td>
</tr>
<tr>
<td>(2.0 to 2.4)</td>
<td>655830</td>
<td>655516.1</td>
<td>0.0479%</td>
</tr>
<tr>
<td>(2.4 to 2.8)</td>
<td>7507620</td>
<td>7510327.1</td>
<td>0.0360%</td>
</tr>
<tr>
<td>(2.8 to 3.2)</td>
<td>46597020</td>
<td>46601762.1</td>
<td>0.0102%</td>
</tr>
<tr>
<td>(3.2 to 3.6)</td>
<td>157066112</td>
<td>157056047.3</td>
<td>0.0064%</td>
</tr>
<tr>
<td>(3.6 to 4.0)</td>
<td>288150018</td>
<td>288144661.9</td>
<td>0.0019%</td>
</tr>
<tr>
<td>(4.0 to 4.4)</td>
<td>288145262</td>
<td>288144660.9</td>
<td>0.0002%</td>
</tr>
<tr>
<td>(4.4 to 4.8)</td>
<td>157043835</td>
<td>157056047.3</td>
<td>0.0078%</td>
</tr>
<tr>
<td>(4.8 to 5.2)</td>
<td>46607503</td>
<td>46601762.1</td>
<td>0.0123%</td>
</tr>
<tr>
<td>(5.2 to 5.6)</td>
<td>7508108</td>
<td>7510327.1</td>
<td>0.0295%</td>
</tr>
<tr>
<td>(5.6 to 6.0)</td>
<td>655295</td>
<td>655516.1</td>
<td>0.0337%</td>
</tr>
<tr>
<td>&gt; 6.0</td>
<td>31558</td>
<td>31686.0</td>
<td>0.4041%</td>
</tr>
</tbody>
</table>
```
6. Generating Uniformly Distributed Pseudorandom Integers – The RandI() Function

The RandI() function can be used to generate uniformly distributed pseudorandom integers that conform to the probability density function given by equation 6.

\[
f(k) = \begin{cases} 
\frac{1}{b-a+1}, & \text{for } a \leq k \leq b \\
0, & \text{otherwise} 
\end{cases} \quad (6)
\]

The RandI() function uses the transformation presented in equation 7, where \( q \) is a uniformly distributed pseudorandom integer in the interval \([0,2^{32})\).

\[k = a + \text{trunc}\left(\frac{q}{2^{32}}(b-a+1)\right)\]  \quad (7)

6.1 RandI() Code

```cpp
template<class T> long RandI() //<==========UNIFORMLY DISTRIBUTED PSEUDORANDOM INT T I[625], //<-STATE OF MERSENNE TWISTER (FOR T, USE A 32-BIT UNSIGNED INT) long a, long b) { //<-LOWER & UPPER BOUNDARIES OF DISTRIBUTION return a + T(Rand(I)/4294967296.*(b-a+1)); //..............................[a,b] } //~~~YAGENAUT@GMAIL.COM~~~~~~~~~~~~~~~~~~~~~~~~~LAST~UPDATED~21JAN2014~~~~~~
```

6.2 RandI() Template Classes

- \( T \) should be a 32-bit unsigned integer type. Greater than 32-bit types can be used, but performance may suffer.

6.3 RandI() Parameters

- \( I \) points to an array that contains the state of the Mersenne twister algorithm.
- \( a \) specifies \( a \), the lower bound for the distribution.
- \( b \) specifies \( b \), the upper bound for the distribution.

6.4 RandI() Return Value

The RandI() function returns a uniformly distributed pseudorandom integer in the interval \([a,b]\).

6.5 RandI() Simple Example

The following example uses the RandI() function to generate and sum one billion uniformly distributed pseudorandom integers in the interval \([-5,4]\).
#include <cstdio>
#include <ctime>
#include "y_random.h"

int main(){
  // SIMPLE EXAMPLE FOR THE yRandom::RandI() FUNCTION
  unsigned I[625];
  yRandom::Initialize(I,1);
  // state of Mersenne twister
  double s=0;
  for(int i=0;i<1000000000;++i)s+=yRandom::RandI(I,-5,4);
  printf("10^9 pseudorandom numbers generated and summed in %.3f seconds.\n" "Their average is %f.\n\n",double(clock())/CLOCKS_PER_SEC,s/1000000000);
}

10^9 pseudorandom numbers generated and summed in 10.140 seconds.
Their average is -0.500057.

### 6.6 RandI() Binning Example

The following example uses the RandI() function to generate and bin one billion uniformly distributed pseudorandom integers in the interval [-5,4].

```c
#include <cstdio>
#include "y_random.h"

int main(){
  // BINNING EXAMPLE FOR THE yRandom::RandI() FUNCTION
  int M=1000000000;
  const int N=11;
  double B[N];
  for(int i=0;i<N;++i)B[i]=i-6;
  double E[N+1]={0};
  for(int i=1;i<N;++i)E[i]=M/(N-1.);
  int C[N+1];
  for(int i=0;i<N+1;++i)C[i]=0;
  for(int i=0;i<M;++i,++C[j],j=0)
    for(double x=yRandom::RandI(I,-5,4);x>B[j]&&j<N;++j);
  printf(" COUNT COUNT
      BIN       (RAW)    (EXPECTED)  %%DIFF \
      \-----------------------------------------\n      \n");
  for(int i=0;i<N+1;++i){
    if(i==0)printf(" <= %4.1f     ,",B[0]);
    else if(i==N)printf(" > %4.1f     ,",B[N-1]);
    else printf(" (%4.1f to %4.1f]  ,",B[i-1],B[i]);
    printf("%11d  ,%13.1f",C[i],E[i]);
    E[i]>1?printf(" ,%9.4f%%\n",fabs(E[i]-C[i])/E[i]*100):printf("\n");
  }
}
```

//~~~~~~YAGENAULT@GMAIL.COM~~~~~~~~~~~~~~~~~~~~~~~~~LAST~UPDATED~21JAN2014~~~~~~
7. Generating Poisson-Distributed Pseudorandom Integers – The RandP() Function

The RandP() function can be used to generate Poisson distributed pseudorandom integers that conform to the probability density function given by equation 8, where \( \mu \) is the mean of the distribution.

\[
f(k) = \frac{\mu^k e^{-\mu}}{k!}
\]  

The RandP() function uses a transformation described by Knuth (6) to generate Poisson-distributed pseudorandom integers by finding the largest \( k \) that satisfies equation 9.

\[
\prod_{i=0}^{k} r_i > e^{-\mu}
\]  

where \( r_i \) is a uniformly distributed pseudorandom number in the interval (0,1]. Equation 3 can be used to generate each \( r_i \) given \( q_i \), a uniformly distributed pseudorandom integer in the interval [0,2^{32}).

Note that the for large \( \mu \), the RandP() function is slow. This problem can be overcome by making use of the fact that, for large \( \mu \), Poisson distributions are approximately normal.

<table>
<thead>
<tr>
<th>BIN</th>
<th>COUNT (RAW)</th>
<th>COUNT (EXPECTED)</th>
<th>%DIFF</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;= 6.0</td>
<td>0</td>
<td>0.0</td>
<td>0.0010%</td>
</tr>
<tr>
<td>(-6.0 to -5.0]</td>
<td>99999019</td>
<td>100000000.0</td>
<td>0.0108%</td>
</tr>
<tr>
<td>(-5.0 to -4.0]</td>
<td>100010759</td>
<td>100000000.0</td>
<td>0.0024%</td>
</tr>
<tr>
<td>(-4.0 to -3.0]</td>
<td>99997646</td>
<td>100000000.0</td>
<td>0.0069%</td>
</tr>
<tr>
<td>(-3.0 to -2.0]</td>
<td>100006913</td>
<td>100000000.0</td>
<td>0.0104%</td>
</tr>
<tr>
<td>(-2.0 to -1.0]</td>
<td>100010417</td>
<td>100000000.0</td>
<td>0.0206%</td>
</tr>
<tr>
<td>(-1.0 to 0.0]</td>
<td>99979386</td>
<td>100000000.0</td>
<td>0.0061%</td>
</tr>
<tr>
<td>( 0.0 to 1.0]</td>
<td>99993881</td>
<td>100000000.0</td>
<td>0.0033%</td>
</tr>
<tr>
<td>( 1.0 to 2.0]</td>
<td>9999651</td>
<td>100000000.0</td>
<td>0.0047%</td>
</tr>
<tr>
<td>( 2.0 to 3.0]</td>
<td>10004658</td>
<td>100000000.0</td>
<td>0.0023%</td>
</tr>
<tr>
<td>( 3.0 to 4.0]</td>
<td>9997670</td>
<td>100000000.0</td>
<td>0.0023%</td>
</tr>
<tr>
<td>&gt; 4.0</td>
<td>0</td>
<td>0.0</td>
<td>0.0023%</td>
</tr>
</tbody>
</table>
7.1 RandP() Code

```cpp
template<class T> T RandP() {  // POISSON-DISTRIBUTED PSEUDORANDOM UNSIGNED INT
  T I[625];  // STATE OF MERSENNE TWISTER (FOR T, USE A 32-BIT UNSIGNED INT)
  double m;  // MEAN (MUST BE GREATER THAN ZERO)
  T k = 0;  // STATE OF MERSENNE TWISTER (FOR T, USE A 32-BIT UNSIGNED INT)
  for (double P = 1, E = exp(-m); P > E; P *= (Rand(I) + 1.)/4294967296) ++k;
  return k - 1;
};//~~~~~YAGENAUT@GMAIL.COM~~~~~~~~~~~~~~~~~~~~~~~~~LAST~UPDATED~21JAN2014~~~~~~
```

7.2 RandP() Template Classes

T

T should be a 32-bit unsigned integer type. Greater than 32-bit types can be used, but performance may suffer.

7.3 RandP() Parameters

I

I points to an array that contains the state of the Mersenne twister algorithm.

m

m specifies \( \mu \), the mean of a Poisson distribution.

7.4 RandP() Return Value

The RandP() function returns a Poisson-distributed pseudorandom unsigned integer.

7.5 RandP() Simple Example

The following example uses the RandP() function to generate and sum one billion Poisson-distributed pseudorandom integers with \( \mu = 1 \).

```cpp
#include <cstdio>
#include <cstdlib>
#include "y_random.h"

int main() {
  unsigned I[625];
  yRandom::Initialize(I, 1);
  double s = 0;
  for (int i = 0; i < 1000000000; ++i) s += yRandom::RandP(I, 1);
  printf("10^9 pseudorandom numbers generated and summed in %.3f seconds. \n" "Their average is %.f.\n", double(clock()) / CLOCKS_PER_SEC, s / 1000000000);
};//~~~~~~YAGENAUT@GMAIL.COM~~~~~~~~~~~~~~~~~~~~~~~~~LAST~UPDATED~21JAN2014~~~~~~
```

OUTPUT:

10^9 pseudorandom numbers generated and summed in 35.708 seconds.
Their average is 1.000013.
7.6 RandP() Binning Example

The following example uses the RandP() function to generate and bin one billion Poisson distributed pseudorandom integers with $\mu = 1$.

```c
#include <cstdio>
#include "y_random.h"
int main(){
  // <================= BINNING EXAMPLE FOR THE yRandom::RandP() FUNCTION
  unsigned I[625];
  //......state of Mersenne twister
  int M=1000000000;
  const int N=11;
  double B[N];
  double E[N+1]={0};
  int C[N+1];
  int i=0,j=0;
  for (i=0;i<M;++i,++C[j],j=0)
    for (double x = RandP(I,1); x > B[j] && j < N; ++j);

  printf("    COUNT    COUNT
    BIN    ,    (RAW)    ,   (EXPECTED)  ,   %DIFF
      -----------------------------------------
      <=  0.0     ,  367886090  ,  367879441.2  ,   0.0018%
      ( 0.0 to  1.0]  ,  367871283  ,  367879441.2  ,   0.0022%
      ( 1.0 to  2.0]  ,  183927644  ,  183939720.6  ,   0.0066%
      ( 2.0 to  3.0]  ,   61321837  ,   61313240.2  ,   0.0140%
      ( 3.0 to  4.0]  ,   15332140  ,   15328310.2  ,   0.0112%
      ( 4.0 to  5.0]  ,     510230  ,     510943.7  ,   0.1402%
      ( 5.0 to  6.0]  ,      7287  ,      72992.0  ,   0.1657%
      ( 6.0 to  7.0]  ,       9053  ,       1014.8  ,  10.4779%
      ( 7.0 to  8.0]  ,         999  ,          999  ,  100.0000%
      ( 8.0 to  9.0]  ,        112  ,        101.4  ,   10.4779%
      ( 9.0 to 10.0]  ,         11  ,           9.2  ,   10.4779%
      > 10.0     ,         12  ,          9.2  ,  30.2061%
    
```
8. Code Summary

A summary sheet is provided at the end of this report. It presents the yRandom namespace, which contains the six functions that are described in this report.
yRandom Summary

```c
#include <y_random.h>

namespace yrandom { // Initialize state of Mersenne Twister (for T, use a 32-bit
// unsigned int)
T I[625], T I[625];

T> T = yRandom::RandN(I); // STATE OF MERSSENNE TWISTER (FOR T, USE A 32-BIT UNIGNED INT)
unsigned y = I[i] = I[i<227?i+397:i

yRandom::Rand(I)

yRandom::RandP() Probability Density Function and Simple Example

```

### RandU() Probability Density Function and Simple Example

```c
#include <y_random.h>

namespace yrandom {

```

```
```
9. References


INTENTIONALLY LEFT BLANK.