Random Number Generators

Jozo Dujmović

Contents

- Linear congruential generator
- Static variables and their applications
- Function rand( ) and constant RAND_MAX
- Generators of uniform and nonuniform distributions
- Machine dependent generators
- Machine independent generators
- Shuffler
- Monte Carlo integration

Applications of Random Numbers

- Simulation models of computer systems
- Monte Carlo methods for numerical computations
  - Based on high quality uniform generators
- File generation for benchmarking
  - Quality of randomness is not important (usually speed is important)
- Cryptography

Linear congruential generator

- Algorithm for computing pseudorandom numbers N_0, N_1, N_2, … :

```
N = seed;
while(1)
{
    N = (a*N + c) % M;
    use N ;  if (done) break;
}
```

Maximum sequence

- Linear congruential generators create a long sequence of pseudorandom numbers. The sequence permanently repeats.
- The maximum length of the sequence is M. The minimum random value is 0 and the maximum random value is M-1
- The numbers are called pseudorandom because they are generated by a known deterministic algorithm

Uniform distribution

- If the sequence of pseudorandom integers has the maximum length M then the sequence is a permutation of the regular sequence 0,1,2,…,M-1
- If each integer occurs only once then the distribution is uniform

```
int rng(void)
{
    static int N = 0;
    return N = (5*N + 3) % 8;  // Max length = 8
}
int rng(int M)          // M is supposed to be 2^b
{
    static int N = 0;
    return N = (5*N + 3) % M;
}
// main
for(i=0; i<32; i++) cout << setw(3) << rng() << ((i+1)%16 ? ' ' : '\n');
for(cout<<endl, i=0; i<32; i++)
    cout << setw(3) << rng(16) << ((i+1)%16 ? ' ' : '\n');
```

```
C:\CentralFiles\Jozo\M Y   D O C U M E N T S\C L A S S E S\840 SW Metri...
  3   2   5   4   7   6   1   0   3   2   5   4   7   6   1   0
  3   2   5   4   7   6   1   0   3   2   5   4   7   6   1   0

  3   2  13   4   7   6   1   8  11  10   5  12  15  14   9   0
  3   2  13   4   7   6   1   8  11  10   5  12  15  14   9   0
Press any key to continue . . .
```

Static variables and their use in random number generators

The concept of static objects

- Static objects have static (permanent) lifetime
- Static objects within a code block are constructed only once, initialized only once, and are not destroyed on leaving the block
- On re-entering the block the static object is available and has its previous value

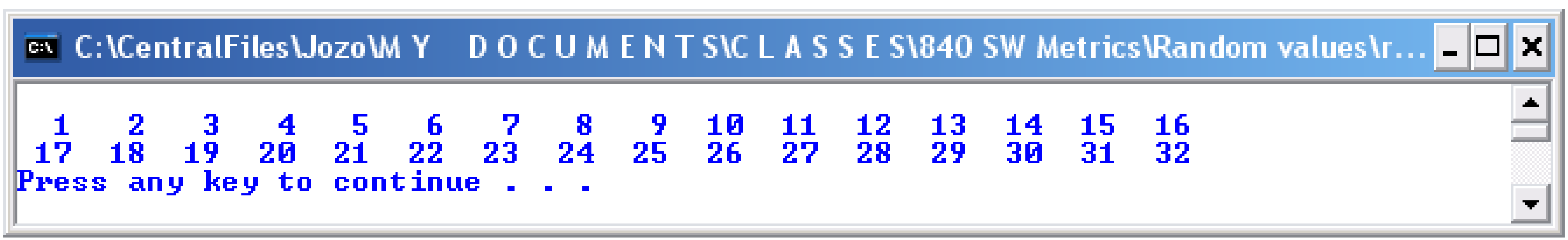
Static variables in functions

- Static local variables of function f are initialized only the first time the function is called.
- The values of local static variables are saved between function calls
- Syntax: static variables are defined using the prefix **static**:

```
static  int  a, b, c, N[200];
static  double x, y, z, A[20][20];
```

An example of static variable

```
int f(void)
{
  static int s=0;
  return ++s;
}

// main
for(cout<<endl, i=0; i<32; i++)
    cout << setw(3) << f( ) << ((i+1)%16 ? ' ' : '\n');
```



Use of static variables

- **Initialization of variables** (variables are given initial values, and these values are then modified in each call)
- **First call processing** (during the first call a set of parameters is computed and the parameters are then used in subsequent function calls)
- **Binary flags** (two alternative different processing in a sequence of calls)

Initialization of variables

```
int f(void)
{
    static int S = <initial value>;
    ………………
    S = <next value> ; // This value will
    return S;          // be available during
}                      // the next call of
                       // function f( )
```

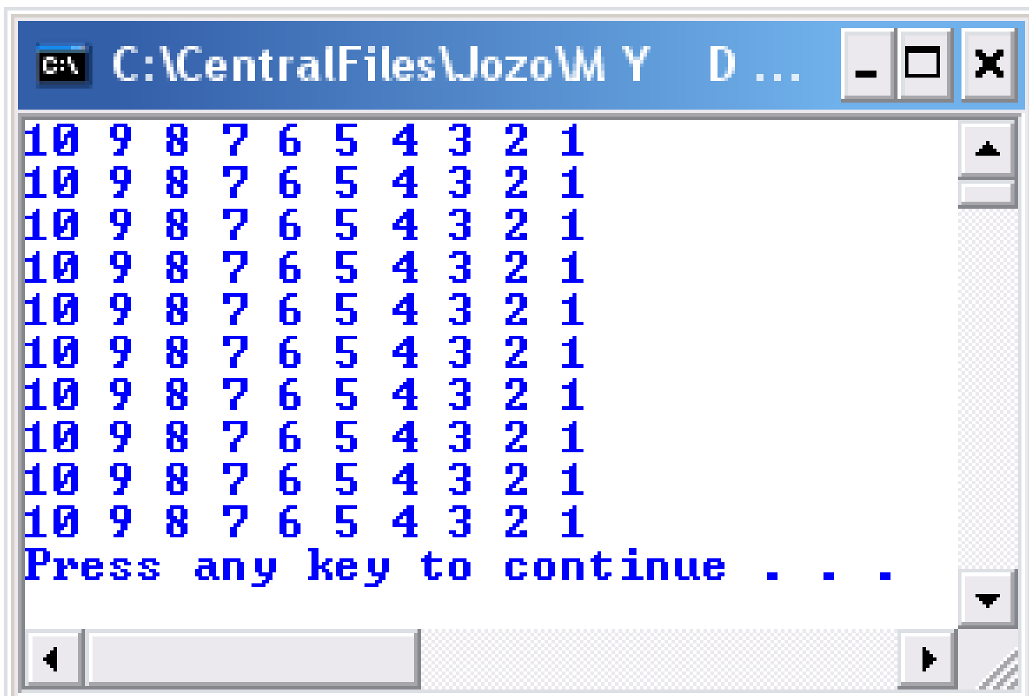
First call processing

```
int f(void)
{  static int firstall = 1;
   int a[200], result;
   if(firstcall)
   { compute array a[ ];
     firstcall = 0;
   }
    use array a[ ] to compute result;
    return result;
}
```

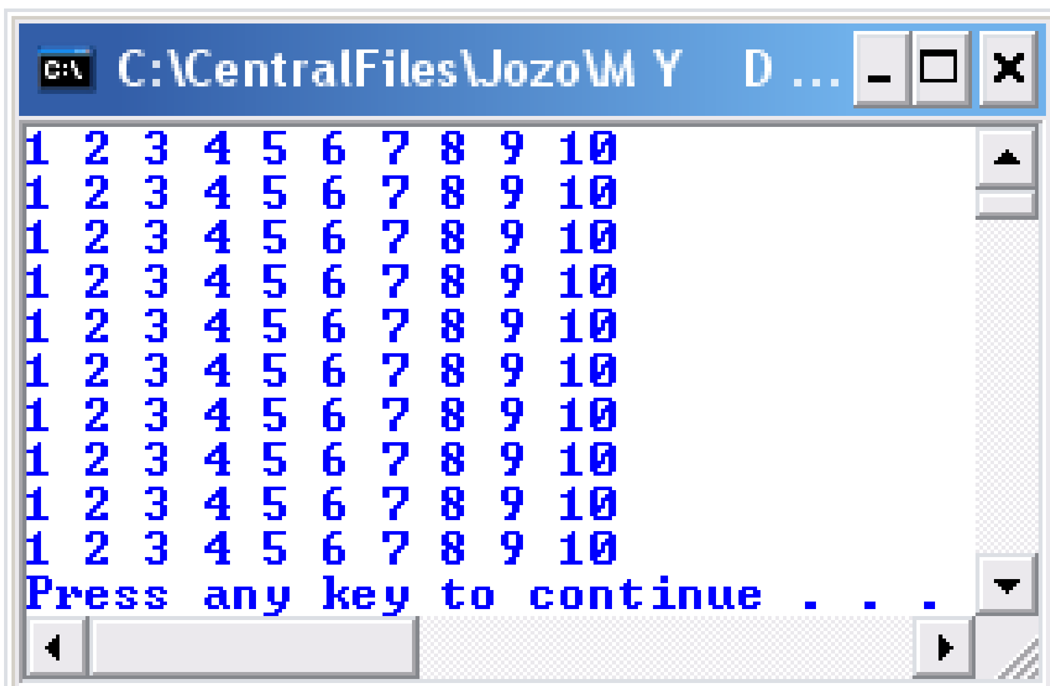
Static binary flags

```
int f(void)
{  static int flag = 1, x, y;
   if(flag)
   { compute the pair x and y;
     flag = 0;  return x;
   }
   else
   { flag = 1; return y;
   }
}
```

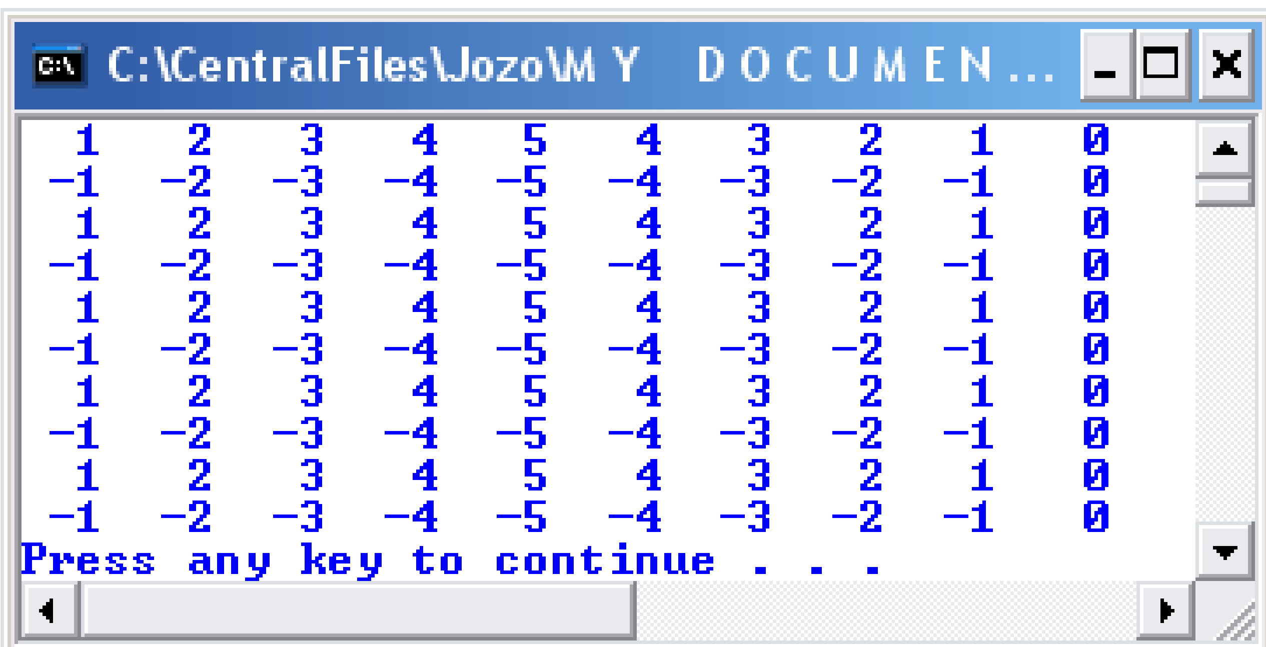
```
int saw(int nmax)
{
      static int n=0;
      if (n>1) return n=n-1;
      return n=nmax;
}

// main
for(i=0; i<100; i++) cout << saw(10) << ((i+1)%10 ? ' ' : '\n');
```



```
int saw1(int Nmax)
{
    static int N=0;
    N = N+1 - N/Nmax*Nmax;
    return N;
}

// main
for(i=0; i<100; i++) cout << saw1(10) << ((i+1)%10 ? ' ' : '\n');
```



```
int up_down(int Nmax)
{     static int N=0, one=1;
      N += one;
      if(N>=Nmax || N<=-Nmax) one=-one;
      return N;
}
// main
for(i=0; i<100; i++)
    cout << setw(3) << up_down(5) << ((i+1)%10 ? ' ' : '\n');
```



Function rand( ) and constant RAND_MAX

C/C++ generator rand( )

- The **rand()** function returns an unsigned pseudorandom integer in the range 0 to **RAND_MAX.** Usually RAND_MAX= 32767 (the case16 bits) or 2147483647 (32 bits)
- #include<stdlib.h> // in the case of C
- #include<cstdlib> // in the case of C++
- Function **srand( )** can be used to seed the pseudorandom-number generator before calling **rand()**.

Selecting the sequence of RN's

- Default seed (built in the generator)
- Selecting a seed as a given constant value:

```
srand(13);
```

- Randomizing:

```
srand( (unsigned) time( NULL ) );
```

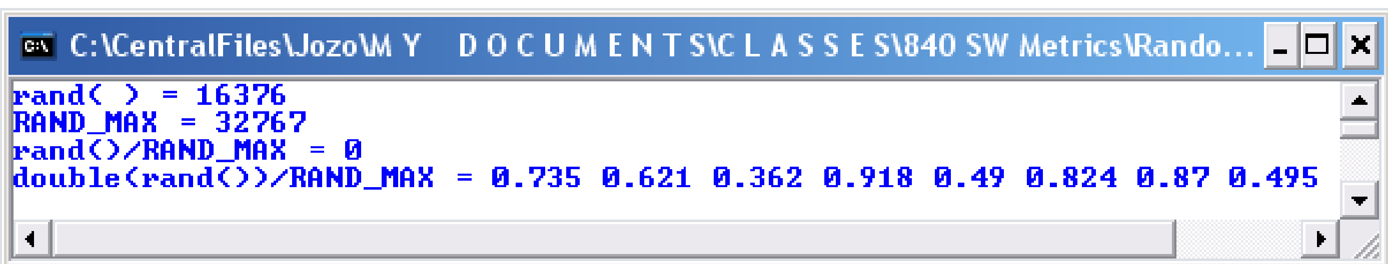
The concept of default seed

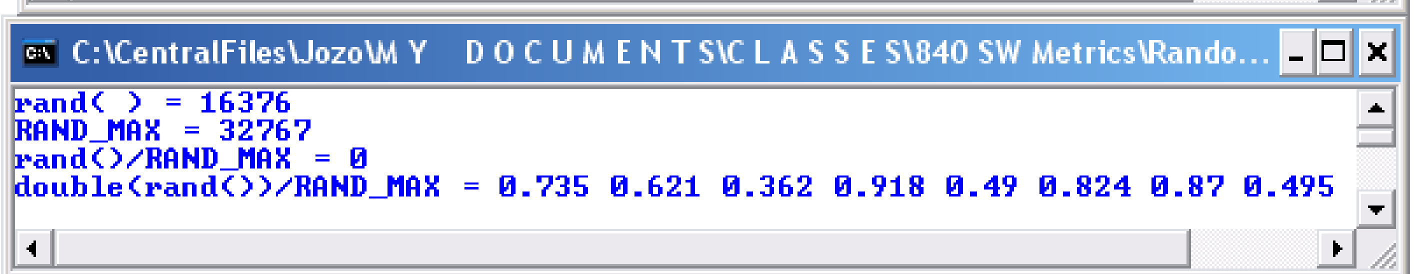
```
unsigned int rng(void)
{
   static unsigned int random = 13;
   return (random = f(random));
}
```

Some applications need the same sequence of random numbers. E.g. if we measure the speed of sort benchmark it is necessary that each competitor gets the same sequence of random numbers.

An example of default seed

```
int main(void)
{
    srand(13); // Constant seed
    cout << "rand( ) = " << rand()
         << "\nRAND_MAX = " << RAND_MAX
         << "\nrand()/RAND_MAX = " << rand()/RAND_MAX
         << "\ndouble(rand())/RAND_MAX = " << setprecision(3);
    for(int i=0; i<8; i++) cout << double(rand())/RAND_MAX <<' ';
    cout << "\n\n";
    system("pause");
    return 0;
}
```





Each time rand( ) generates the same sequence of random values

The concept of randomized seed

```
unsigned int rng(void)
{
   static unsigned int random =
                        (unsigned) time( NULL );
   return (random = f(random));
}
```

Some applications need always a different sequence of random numbers. E.g. all games need different sequences of random numbers to avoid repeating exactly the same game over and over again. Simulators of queuing systems also need different random sequences.

An example of randomized seed

```
int main(void)
{
    srand( (unsigned) time( NULL ) ); // Randomize
    cout << "rand( ) = " << rand()
         << "\nRAND_MAX = " << RAND_MAX
         << "\nrand()/RAND_MAX = " << rand()/RAND_MAX
         << "\ndouble(rand())/RAND_MAX = " << setprecision(3);
    for(int i=0; i<8; i++) cout << double(rand())/RAND_MAX <<' ';
    cout << "\n\n";
    system("pause");
       return 0;
}
```

C:\CentralFiles\Jozo\M Y D O C U M E N T S\C L A S S E S\840 SW Metrics\Rando...

```
rand( ) = 11092
RAND_MAX = 32767
rand()/RAND_MAX = 0
double(rand())/RAND_MAX = 0.601 0.61 0.95 0.0607 0.134 0.802 0.317 0.162
```

C:\CentralFiles\Jozo\M Y D O C U M E N T S\C L A S S E S\840 SW Metrics\Random v...

```
rand( ) = 1920
RAND_MAX = 32767
rand()/RAND_MAX = 0
double(rand())/RAND_MAX = 0.0921 0.717 0.276 0.964 0.305 0.0772 0.85 0.331
```

Each time rand( ) generates a different sequence of random values

```cpp
unsigned int rng(void) // linear congruential random number generator
{
   static unsigned int random = (unsigned) time( NULL );
   return random = (29*random + 13)%64;
}

int main(void)
{
    for(int i=0; i<80; i++) cout << setw(3) << rng()
                                 << ((i+1)%16 ? ' ':'\n');
    system("pause");
    return 0;
}
```

```
C:\CentralFiles\Jozo\MY DOCUMENTS\CLASSES\840 SW Met...
 11  12  41  50  55   8  53  14  35   4   1  42  15   0  13   6
 59  60  25  34  39  56  37  62  19  52  49  26  63  48  61  54
 43  44   9  18  23  40  21  46   3  36  33  10  47  32  45  38
 27  28  57   2   7  24   5  30  51  20  17  58  31  16  29  22
 11  12  41  50  55   8  53  14  35   4   1  42  15   0  13   6
```

```
C:\CentralFiles\Jozo\MY DOCUMENTS\CLASSES\840 SW Met...
 34  39  56  37  62  19  52  49  26  63  48  61  54  43  44   9
 18  23  40  21  46   3  36  33  10  47  32  45  38  27  28  57
  2   7  24   5  30  51  20  17  58  31  16  29  22  11  12  41
 50  55   8  53  14  35   4   1  42  15   0  13   6  59  60  25
 34  39  56  37  62  19  52  49  26  63  48  61  54  43  44   9
```

```
C:\CentralFiles\Jozo\MY DOCUMENTS\CLASSES\840 SW Met...
 30  51  20  17  58  31  16  29  22  11  12  41  50  55   8  53
 14  35   4   1  42  15   0  13   6  59  60  25  34  39  56  37
 62  19  52  49  26  63  48  61  54  43  44   9  18  23  40  21
 46   3  36  33  10  47  32  45  38  27  28  57   2   7  24   5
 30  51  20  17  58  31  16  29  22  11  12  41  50  55   8  53
```

Each execution of program generates a different sequence of 64 random numbers (maximum length)

Generating random numbers with uniform and nonuniform distributions

Distributions

- Uniform distribution in the interval [a,b]
- Standard uniform distribution: the interval is [0,1]
- Nonuniform distributions:
  - Exponential
  - Normal
  - Arbitrary

Uniform distribution

- N = [a*N + c] mod M
- M = 2^b;   b=number of bits in a word
- Conditions for the sequence of max length:

  c = odd integer ≈ 0.211*M

  a-1 must be multiple of 4
- If the random sequence has the max length each value occurs only once and the distribution is uniform
- Example: N = (5*N + 3) mod 8

  N = 0, 3, 2, 5, 4, 7, 6, 1, 0,…

```
/*****************************************************************\
|   Program:   RNG.cpp - uniform random number generator          |
|   Problem:   Generate a max length sequence for the 8-bit word  |
|   Purpose:   Demo program for static variables                  |
|   Author :   Jozo J. Dujmovic                                   |
|   Date   :   4/20/2012                                          |
\*****************************************************************/
#include<iostream>
#include<iomanip>
using namespace std;

unsigned char random(void)    // Uniform random number generator
{   static unsigned char c=1; // Initial value
    return 13*c + 1;          // 0 <= c <= 255
}                             // Maximum length of sequence = 256

int main(void)
{int r, i=0;
 do
  cout << setw(4) << (r=random()) << (++i%10 ? ' ' : '\n');
 while(r != 1);
 cout << "\n\n";
 system("pause");
 return 0;
}
```

Cycle length = 256 =2 ^ 8

This is the max length for the 8 bit word

C:\CentralFiles\Jozo\M Y D O C U M E N T S\C L A S...

```
 14  183   76  221   58  243   88  121   38  239
 36  213  210  171  176  241   62   39  252  205
106   99    8  105   86   95  212  197    2   27
 96  225  110  151  172  189  154  211  184   89
134  207  132  181   50  139   16  209  158    7
 92  173  202   67  104   73  182   63   52  165
 98  251  192  193  206  119   12  157  250  179
 24   57  230  175  228  149  146  107  112  177
254  231  188  141   42   35  200   41   22   31
148  133  194  219   32  161   46   87  108  125
 90  147  120   25   70  143   68  117  242   75
208  145   94  199   28  109  138    3   40    9
118  255  244  101   34  187  128  129  142   55
204   93  186  115  216  249  166  111  164   85
 82   43   48  113  190  167  124   77  234  227
136  233  214  223   84   69  130  155  224   97
238   23   44   61   26   83   56  217    6   79
  4   53  178   11  144   81   30  135  220   45
 74  195  232  201   54  191  180   37  226  123
 64   65   78  247  140   29  122   51  152  185
102   47  100   21   18  235  240   49  126  103
 60   13  170  163   72  169  150  159   20    5
 66   91  160   33  174  215  236  253  218   19
248  153  198   15  196  245  114  203   80   17
222   71  156  237   10  131  168  137  246  127
116  229  162   59    0    1

Press any key to continue . . .
```

The role of uniform distribution

- Standard uniform distribution is the most important of all distributions
- Why? Because other distributions can be obtained as functions or uniform random numbers:

random_number = function(uniform( ))

Nonuniform distributions

- Let urn( ) be a standard uniform random number generator: **P[urn( ) < x] = x**
- Desired probability distribution of random numbers r: **F(x) = P[r ≤ x]**
- Method:

$$\left.\begin{aligned} r &= F^{-1}(u) \\ r &= F^{-1}(1-u) \end{aligned}\right\} \quad \text{equivalent}$$

Derivation

$$u = urn(); \quad 0 \le u \le 1, \ \text{(standard uniform)}$$

$$\mathrm{P}[\mathrm{u} \le x] = x$$

$r = g(u)$ Select g for a desired distribution of r

$$F(x) = P[r \le x] = P[g(u) \le x] = P[u \le g^{-1}(x)]$$

$$= g^{-1}(x) \qquad \therefore \ g(x) = F^{-1}(x)$$

$$r = F^{-1}(u) \qquad \Rightarrow \qquad u = f(r)$$

$$r = F^{-1}(1-u) \qquad \Rightarrow \qquad 1-u = f(r)$$

Exponential distribution

$$F(x) = P[r \le x] = 1 - e^{-x/\overline{r}}$$

$$u = F(r) \quad \text{or} \quad 1 - u = F(r)$$

$$1 - u = 1 - e^{-r/\overline{r}}$$

$$u = e^{-r/\overline{r}}$$

$$-r / \overline{r} = \ln(u)$$

$$r = -\overline{r}\ln(u) = \overline{r}\ln(1/u)$$

```
double expo(double rmean)
{ return rmean*log(RAND_MAX/double(rand( ))); }
```

Types of RNG

- Machine-dependent RNG – not portable (can generate different sequences on different machines; e.g. rand( ))
- Machine-independent RNG – portable (generate the same sequence or RN's on any machine)

Iterative concept of RN generation

- **`R[i]=f(R[i-1]),  R[0]=seed`**
- **`R[i]=f(R[i-1],R[i-2]), R[0]=seed, R[1]=seed`**
- **`R[i]=f(R[i-1],R[i-n]), R[0..n-1]=seeds`**

Machine-dependent and machine-independent RNG's

Machine dependent generator

```
#include<cstdlib>
//------------------------------------------------------------
// Machine-dependent standard uniform random number
// generator from the standard library
//------------------------------------------------------------
 double urn(void)
 {
    return double(rand())/double(RAND_MAX);
 }
```

Generating 1 .. 6 with equal probability (six-sided die simulator)

```
int rng123456(void)
{
   return 1 + rand( )%6;
}
```

Generating random binary sequences: **rand( )%2**

Generating 1,2,3,4 with probabilities 10%,20%,30%,40%

```
int RandomDigit1234(void)
{
  double u = double(rand( ))/RAND_MAX;
  if(u < 0.1) return 1;
  if(u < 0.3) return 2;
  if(u < 0.6) return 3;
  return 4;
}
```

Fast Fibonacci Generator

```
//---------------------------------------------------------
//  Fast Fibonacci-style machine-independent standard
//  uniform random number generator. The quality of
//  randomness is not tested.
//  Jozo Dujmovic, Dec 2002
//---------------------------------------------------------
double FiboURN(void)
{
        static double p=sqrt(2.)-1., q=sqrt(31.)-5.;
        static int flag = 1;

        if(flag)
        {
                p += q;
                if(p>1.) p -= 1.;
                flag = 0;
                return p;
        }
        else
        {
                q += p;
                if(q>1.) q -= 1.;
                flag = 1;
                return q;
        }
}
```

Machine independent generator

```
//----------------------------------------------------------------------
// Machine-independent additive generator of standard uniform random
// numbers. Good quality of randomness.
//                r(i) := (r(i-1) + r(i-17)) mod xmod
// Jozo Dujmovic, 1998
//----------------------------------------------------------------------
double uniform(void)
{
      static unsigned long r[18]={131071,43691,262657,649657,274177,
                                   524287,121369,61681,179951,513239,
                                   333667,909091,1777,8617,87211,
                                   174763,65537,0},
                            xmod = 1048573;
      static double rnmax=xmod;
      static int i=17,j=16,k=0,n=18;
      double rn;

      r[i] = (r[j] + r[k]) % xmod;
      rn   = r[i]/rnmax;
      i    = (i+1)%n;
      j    = (j+1)%n;
      k    = (k+1)%n;
      return rn;
 }
```

The concept of shuffler

- Fill an array of 100-200 components with (arbitrarily distributed) random numbers
- Randomly (uniformly) select a number from the array and deliver to the user
- Restore the array: generate a new random number and replace the number that was delivered to the user

Shuffler

```
//------------------------------------------------------------------------
//      MacLaren - Marsaglia Shuffler for a random number generator
//      definied as urn( ).
//      Jozo Dujmovic, 1998
//------------------------------------------------------------------------
double rng(void)
{
      static int n=200, firstcall=1;
      static double table[200];
      double rnumber;
      int i, itable;

      if (firstcall)
      {
         for(i=0; i<n; i++) table[i]=urn( ); // Any desired distribution
         firstcall = 0;
      }

      itable        = int(n * urn( ));       // Uniform selection from table
      rnumber       = table[itable];
      table[itable] = urn( );                // Any desired distribution
      return rnumber;
}
```

Approximate normal distribution

```
C------------------------------------------------------------
C     Normal random number generator based on Central Limit Theorem
C------------------------------------------------------------
      function gauss( )
      external urn
      r = 0.
      do i=1,12
        call rng(urn, u)
        r = r + u
      end do
      gauss = r-6.
      end
```

U = standard uniform random number

Z = U1+…+U12 – 6   (≈ normal)

R = Mean + Sigma * Z

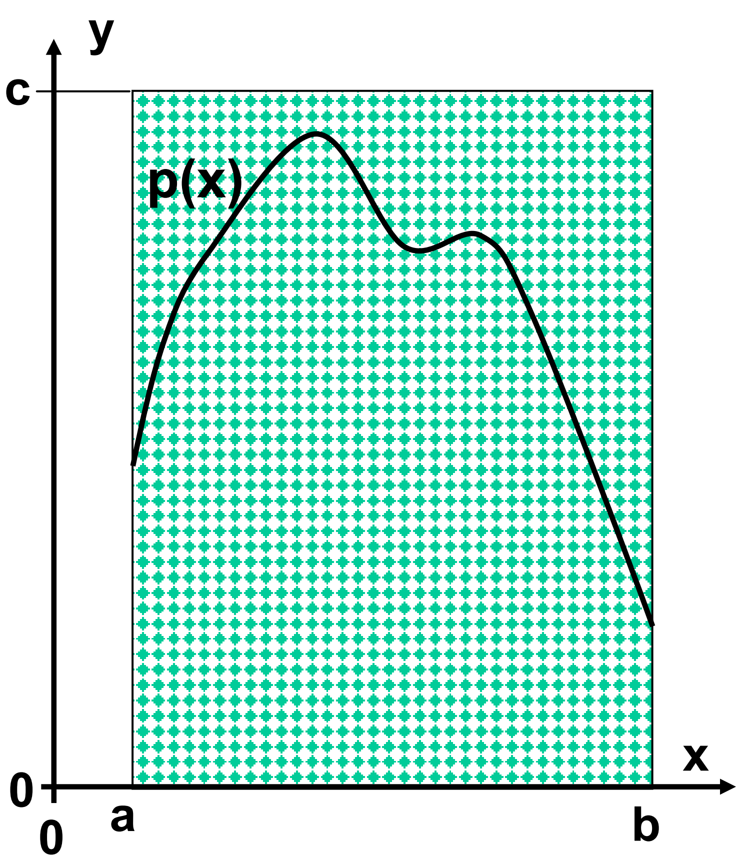
Mean = mean value

Sigma = standard deviation of R

Normal distribution

```
C=========================================================================
C     A NORMAL RANDOM NUMBER GENERATOR BASED ON THE POLAR METHOD        |
C-------------------------------------------------------------------------
C     Jozo J. Dujmovic                                                  |
C=========================================================================
      SUBROUTINE POLAR(RN,AVERAGE,SIGMA)
      LOGICAL flag                             ! A flip-flop flag
      DATA flag,saved_rn /.FALSE., 0./
      SAVE
      IF (flag) THEN                           ! Use the saved random number
         RN = AVERAGE + SIGMA*saved_rn
      ELSE                                     ! Generate a new pair of
   10    u1 = 2.*URN( ) - 1.                   ! normally distributed random
         u2 = 2.*URN( ) - 1.                   ! numbers
         a  = u1*u1 + u2*u2
         IF (a .GE. 1.) GO TO 10
         a  = SQRT(-2.*LOG(a)/a)
         RN = AVERAGE + SIGMA*u1*a             ! Resulting random number
         saved_rn  = u2*a                      ! Next random number
      END IF
      flag = .NOT. flag                        ! Flip the flag
      RETURN
      END
```

Acceptance-rejection RNG method

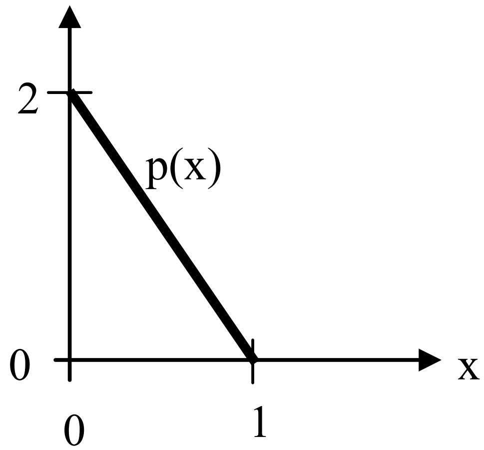


Generate a random point x,y and if the point is under the probability density function, return x. There will be more points where the density function has high values

1. p(x) = a desired probability density function
2. u() = double(rand())/RAND_MAX
3. Generate uniform RN x using u()
4. Generate uniform RN y using u()
5. If y < p(x) return x else go to step 3

Advantages/disadvantages of the A/R method

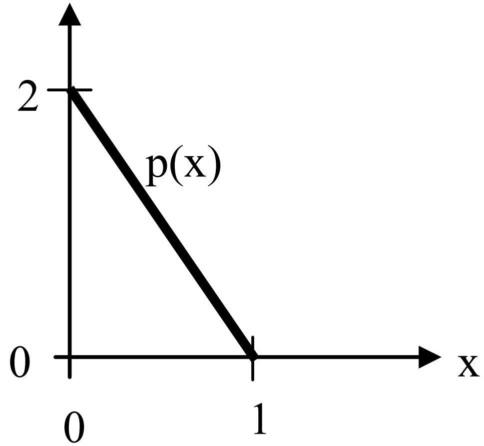
- Advantage: simplicity
- Disadvantages:
  - If the distribution has inconvenient shape (with a high peak) the number of rejected numbers can be very high and the algorithm can be very inefficient.
  - The method is restricted to distributions that are finite, i.e. do not have a long tail (e.g. you cannot use it for the exponential distribution)
  - We have a better method $r = F^{-1}(u())$



Write a function **lin( )** that returns a random number having the presented probability density function $p(x) = 2(x\text{-}1)$, $0 \le x \le 1$. You may use the library function rand( ).

Show a complete derivation of all results.

```
double u() {return double(rand())/RAND_MAX;}

double p(double x) {return 2.*(1.-x);}

double lin(void)
{  double x;
   do x=u(); while(2*u() > p(x));
   return x;
}
```



Write a function **lin( )** that returns a random number having the presented probability density function p(x) = 2(x-1) , $0 \le x \le 1$. You may use the library function rand( ).

Show a complete derivation of all results.

$$p(x) = 2(x-1), \quad 0 \le x \le 1 \qquad\qquad F(r) = u \ \text{ or } \ F(r) = 1-u$$

$$F(r) = \int_0^r p(x)dx = 2\int_0^r (1-x)dx = 2(r - r^2/2) = 2r - r^2 = 1-u$$

$$r^2 - 2r + 1 - u = 0 \quad \to \quad r_{1,2} = \frac{2 \pm \sqrt{4 - 4(1-u)}}{2} = 1 \pm \sqrt{u}$$

$$r \le 1 \quad \to \quad \underline{r = 1 - \sqrt{u}}$$

```
double lin(void)
{
  return 1. - sqrt(double(rand())/RAND_MAX);
}
```

```
#include<iostream>
#include<iomanip>
#include<cmath>
using namespace std;

double u(void) {return double(rand())/RAND_MAX;}

double p(double x) {return 2.*(1.-x);}

double lin(void)
{
      double x;
      do
        x=u();
      while(2*u()>p(x));
      return x;
}

double LIN(void)
{
  return 1.- sqrt(double(rand())/RAND_MAX);
}
```

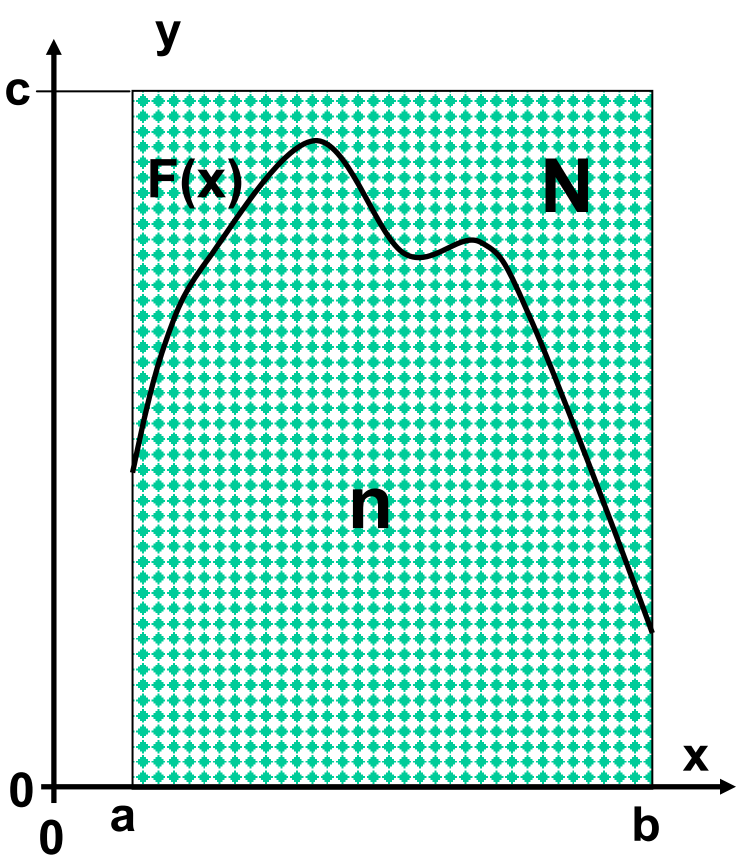
```
int main(void)
{
 int i;
 double f[21], F[21];
 for(i=0; i<21; i++) f[i]=F[i]=0.;
 for(i=0; i<=100000; i++)
 {
      f[int(20.*lin())]++;
      F[int(20.*LIN())]++;
 }

 for(i=0; i<20; i++) cout<<setw(5) << f[i]; cout<<"\n\n";
 for(i=0; i<20; i++) cout<<setw(5) << F[i]; cout<<"\n\n";
 cout << setprecision(1);
 for(i=0; i<20; i++)
    cout << setw(5) << 200.*fabs(f[i]-F[i])/(f[i]+F[i]);
 cout<< "\n\n";

 system("pause");
 return 0;
}
```

Results

- Both LIN and lin generate the same linear probability density function
- The difference between the two distribution is rather small (0.3-9%)

```
C:\CentralFiles\Jozo\M Y   D O C U M E N T S\C L A S S E S\840 SW Metrics\Random values\AcceptReject.exe
10002 9134 8669 8131 7765 7233 6815 6321 5765 5297 4682 4318 3680 3195 2772 2214 1729 1261  764  254
 9645 9229 8696 8354 7841 7208 6632 6230 5731 5332 4767 4229 3827 3251 2784 2231 1837 1230  697  246
    4    1  0.3    3    1  0.3    3    1  0.6  0.7    2    2    4    2  0.4  0.8    6    2    9    3
Press any key to continue . . .
```

Monte Carlo integration



A = c(b-a) = area or rectangle

N = total number of uniformly distributed random points generated inside the rectangle (a,b)

n = number of points that are located under the curve y = F(x)

n/N = fraction of points that are located under the curve y = F(x)

$$\int_a^b F(x)dx = A \lim_{N\to\infty} \frac{n}{N}$$

```
double urn(void) {return double(rand())/double(RAND_MAX);}

double f(double x) { return x*x*x; } // Integral = (x^4)/4

int main(void)
{
    int i, N, k, n=0 ;
    double x,y, Area;
    srand(time(NULL));

    for(k=0; k<8; k++, cout <<"\n\n")
    for(n=0, N=100; N<=1000000; N*=100, n=0)
    {
        for(i=0; i<N; i++)
        {
            x = urn();
            y = urn();
                if(f(x) > y) ++n;
        }
        Area = double(n)/double(N); // Analytic result = 0.25
        cout << N << "  Integral = " << Area << endl;
    }

    system("pause");
    return 0;
}
```

```
C:\CentralFiles\JozoW Y ...
100   Integral = 0.22
10000   Integral = 0.2414
1000000   Integral = 0.249927

100   Integral = 0.23
10000   Integral = 0.2469
1000000   Integral = 0.24989

100   Integral = 0.21
10000   Integral = 0.2469
1000000   Integral = 0.250005

100   Integral = 0.27
10000   Integral = 0.2563
1000000   Integral = 0.250063

100   Integral = 0.24
10000   Integral = 0.2508
1000000   Integral = 0.249877

100   Integral = 0.26
10000   Integral = 0.2452
1000000   Integral = 0.250609

100   Integral = 0.19
10000   Integral = 0.2462
1000000   Integral = 0.249736

100   Integral = 0.25
10000   Integral = 0.2507
1000000   Integral = 0.250061
```

APPROXIMATE ANALYSIS OF ACCURACY OF MONTE CARLO INTEGRATION

Number of random x,y points used for integration	Number of correct decimal digits
100	**1**
10000	**2**
1000000	**3**

Multidimensional MC integration

$$R = \prod_{i=1}^{m} [a_i , b_i] = \text{ region containing F}$$

$$V = \prod_{i=1}^{m} (b_i - a_i) = \text{ volume of region containing F}$$

$$\int_{a_1}^{b_1} dx_1 \int_{a_2}^{b_2} dx_2 \ldots \int_{a_m}^{b_m} dx_m F(x_1, x_2, \ldots, x_m) = V \lim_{N\to\infty} \frac{n}{N}$$

$$N : (u_1, u_2, \ldots, u_m) \in R, \quad u_i \Big|_{uniformm\ RN} \in [a_i , b_i]$$

n : number of points in subregion defined by $F(\ )$

Example

$$I = \int_0^1 \int_0^1 \sqrt{xy}\,dx\,dy = \frac{2}{3} x^{3/2} \Big|_0^1 \times \frac{2}{3} y^{3/2} \Big|_0^1$$

$$= \frac{4}{9} = 0.444444$$

```
int i, n = 1000000, N=0;
double x, y, z, integral, analytic = 4./9., error;
for(i=0; i<n; i++)
{
   x=urn(); y=urn(); z=urn(); // Random point inside the unit cube
   if(z <= sqrt(x*y)) N++;
}
integral = double(N)/double(n);
error = 100.*(integral - analytic)/analytic;
cout << "\nIntegral of sqrt(x*y) inside [0,1]^2\n\n"
     << "Analytic result     = " << analytic
     << "\nMonte Carlo result = " << integral
     << "\nRelative error [%] = " << error
     << "\n\nERROR ANALYSIS\n\n"
     << setw(10) << "N" << setw(11) << "Integral"
     << setw(12) << "Error [%]"
     << "\n---------------------------------\n";

for(n=100; n<100000000; n*=100)
{
  N=0; srand(time(NULL));
  for(i=0; i<n; i++)
  {
     x=urn(); y=urn(); z=urn();
     if(z <= sqrt(x*y)) N++;
  }
  integral = double(N)/double(n);
  error = 100.*(integral - analytic)/analytic;
  cout << setw(10) << n << setw(11) << integral << setw(12) << error << "\n";
}
```

ACCURACY OF MONTE CARLO INTEGRATION

C:\CentralFiles\Jozo\M Y D O C U M E N...

```
Integral of sqrt(x*y) inside [0,1]^2

Analytic result     = 0.444444
Monte Carlo result  = 0.444397
Relative error [%]  = -0.010675

ERROR ANALYSIS

        N      Integral      Error [%]
----------------------------------------
      100          0.48              8
    10000        0.4455         0.2375
  1000000      0.444051      -0.088525
```

Number of significant decimal digits ≈ 0.5 $\log_{10}(N)$

Accuracy is modest: typically 2-3 significant dec. dig.