Introduction to CUDA Parallel Programming
CUDA平行計算導論

https://ceiba.ntu.edu.tw/1092Phys8061_CUDA

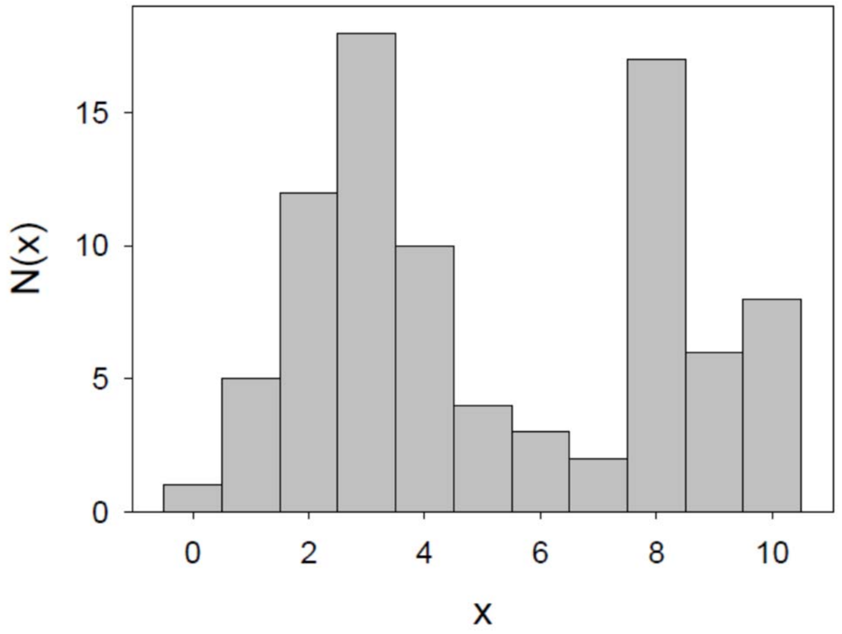
Professor Ting-Wai Chiu (趙挺偉)
Email: twchiu@phys.ntu.edu.tw
Physics Department
National Taiwan University

This lecture will cover:

- Histogram, Atomic Operations
- Histogram with Global Memory
- Histogram with Shared Memory
- Introduction to Pseudo Random Number Generators

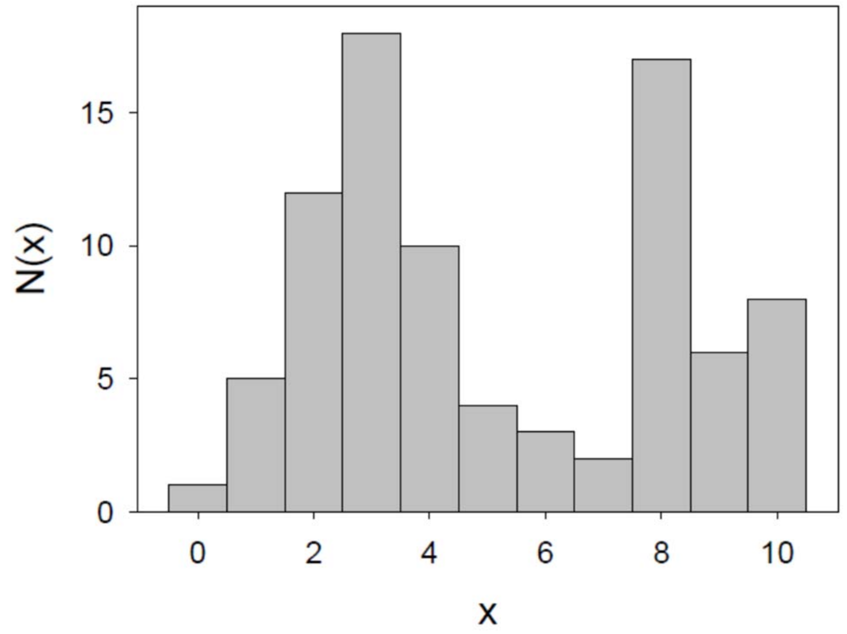
Histogram

Consider a data set of integers {0,1,1,8,6,7,8,7,9,3,6,10,...,1} which gives the following histogram:



Histogram

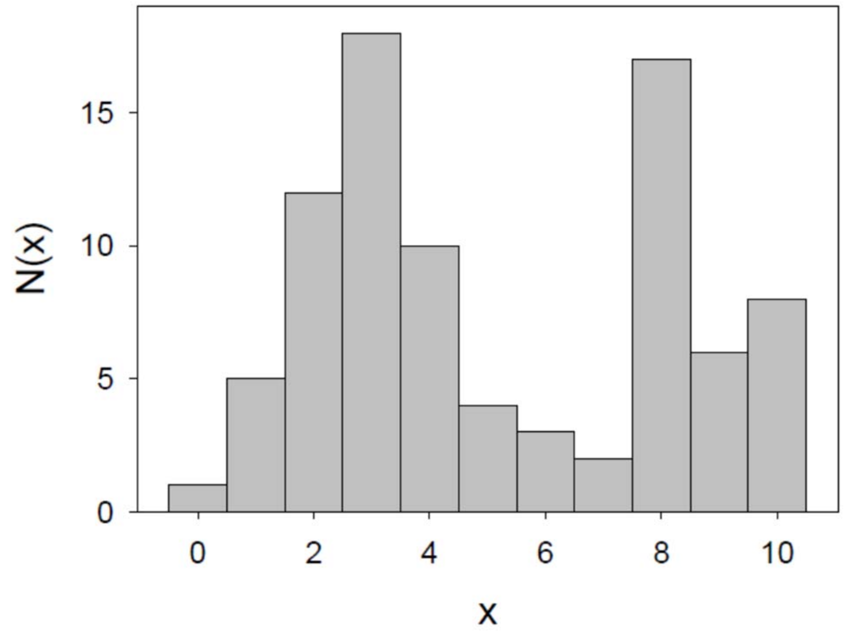
Consider a data set of integers {0,1,1,8,6,7,8,7,9,3,6,10,...,1} which gives the following histogram:



```
hist_c = (unsigned int*)malloc(bins*sizeof(int));
for(int i=0; i<bins; i++) hist_c[i]=0;
...
for(int i=0; i<N; i++) {
  index = (int)((data_h[i]-Rmin)/binsize);
  hist_c[index]++;
}
```

Histogram

Consider a data set of integers {0,1,1,8,6,7,8,7,9,3,6,10,...,1} which gives the following histogram:



Now the question is how to use GPU to compute a histogram.
Consider the data is decomposed into blocks each of 4 threads:

$$\{\boxed{0,1,1,8},\ \boxed{6,7,8,7},\ \boxed{9,3,6,10},\ ...,1\}$$

It must happen that multiple threads may want to increment the same bin of the histogram at the same time, thus leading to unpredictable results.

Histogram

To avoid the racing condition, Nvidia GPUs (since 2009, > sm1.3) can perform the so-called **atomic operations**, such that no other thread can read-modify-write (32/64 bits) data in global/shared memory until one thread has completed its operation on this data. (see Appendix B.12 in the CUDA C Progamming Guide)

Histogram

To avoid the racing condition, Nvidia GPUs (since 2009, > sm1.3) can perform the so-called **atomic operations**, such that no other thread can read-modify-write (32/64 bits) data in global/shared memory until one thread has completed its operation on this data. (see Appendix B.12 in the CUDA C Progamming Guide)

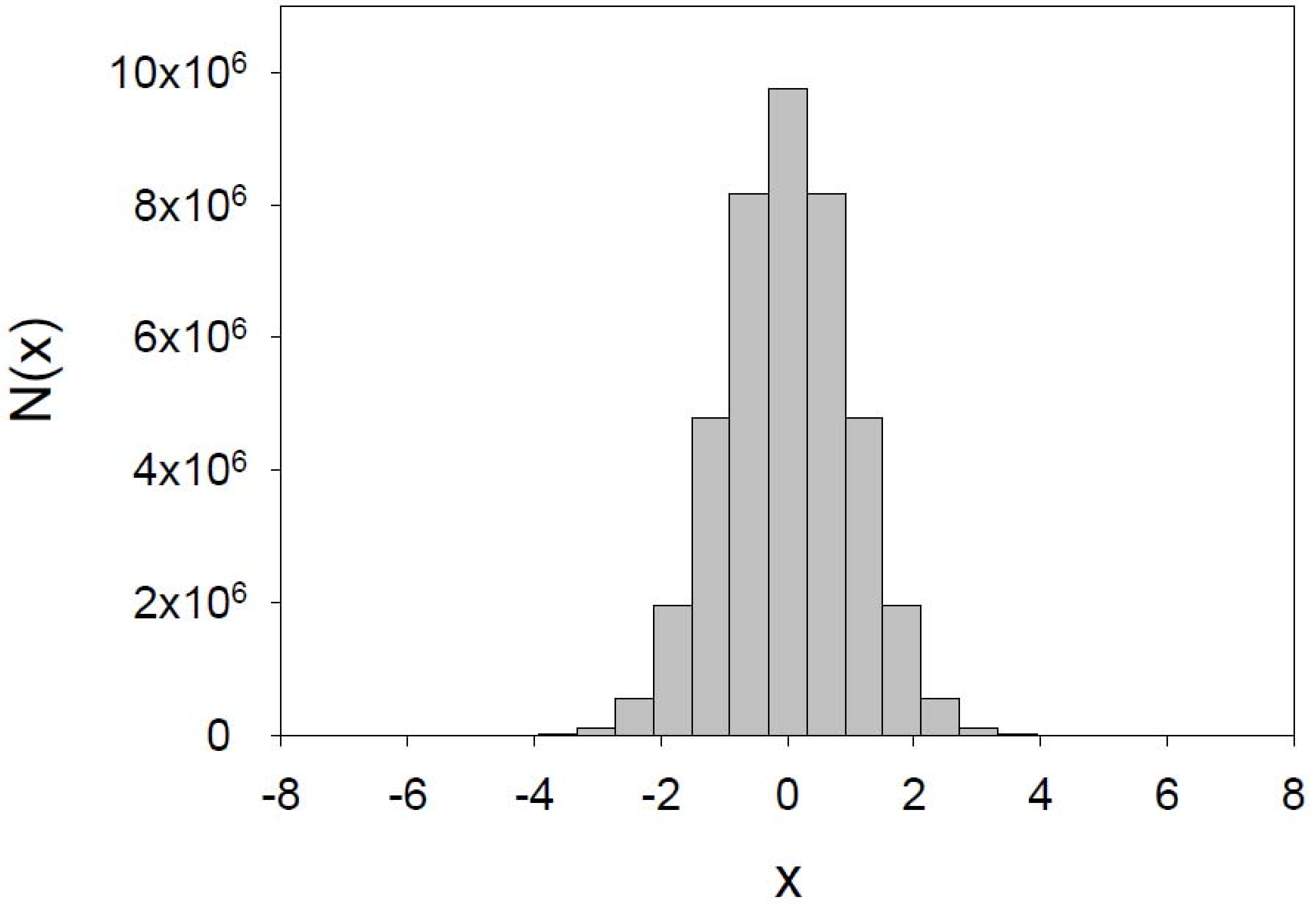
```
__global__ void hist_gmem (float *data, long N, unsigned int *hist,
int bins, float Rmin, float binsize)
{
    // use global memory and atomic addition

    long i = threadIdx.x + blockIdx.x * blockDim.x;
    long stride = blockDim.x * gridDim.x;

    while (i < N) {
      int index = (int)((data[i]-Rmin)/binsize);
      atomicAdd(&hist[index],1);      // atomic addition
      i += stride;       // goto the next grid
    }
    __syncthreads();
}
```

Compute Histogram with Shared Memory

```
__global__ void hist_shmem(float *data, long N, unsigned int *hist,
                           int bins, float Rmin, float binsize)
{   // use shared memory and atomic addition

    extern __shared__ unsigned int temp[];  // blocksize = # of bins
    temp[threadIdx.x] = 0;
    __syncthreads();

    long i = threadIdx.x + blockIdx.x * blockDim.x;
    long stride = blockDim.x * gridDim.x;

    while (i < N) {
      int index = (int)((data[i]-Rmin)/binsize);
      atomicAdd(&temp[index],1);
      i += stride;     // go to the next grid
    }

    __syncthreads();
    atomicAdd( &(hist[threadIdx.x]), temp[threadIdx.x] );

}  // The complete code is in twqcd80:/home/cuda_lecture_2021/histogram_1GPU
```

Histogram of the Test Run

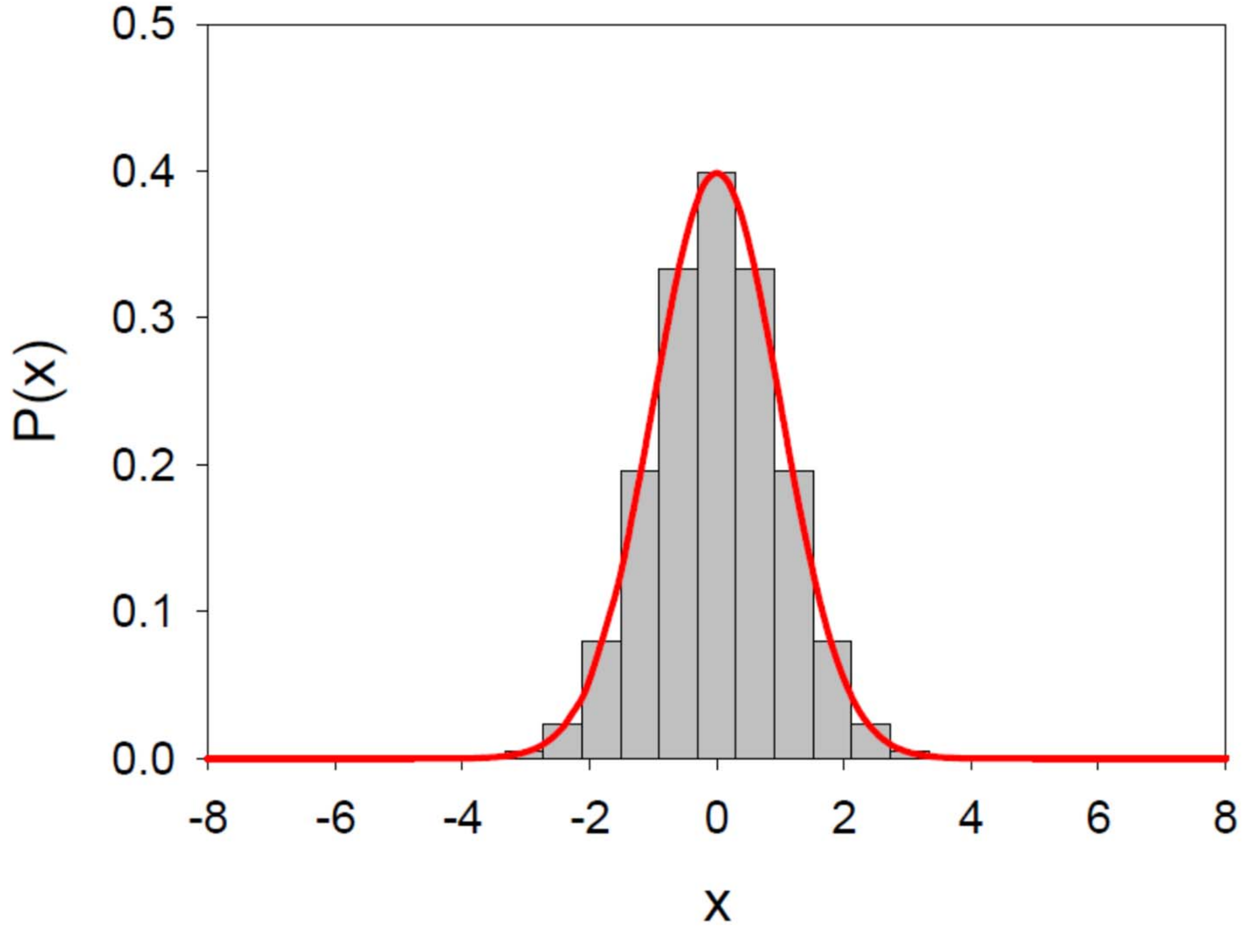
```
Histogram (GPU):

-9.696970 0
 . . .
-6.060606 0
-5.454545 5
-4.848485 99
-4.242424 1559
-3.636364 15729
-3.030303 113053
-2.424242 563761
-1.818182 1962843
-1.212121 4784050
-0.606060 8161313
0.000000 9758491
0.606061 8162147
1.212121 4779688
1.818182 1963511
2.424243 562914
3.030303 113454
3.636364 15710
4.242424 1561
4.848485 105
5.454546 7
6.060606 0
 . . .
9.696970 0
```



Normalized Histogram of the Test Run

```
Histogram (GPU):

-9.696970 0
 . . .
-6.060606 0
-5.454545 5
-4.848485 99
-4.242424 1559
-3.636364 15729
-3.030303 113053
-2.424242 563761
-1.818182 1962843
-1.212121 4784050
-0.606060 8161313
0.000000 9758491
0.606061 8162147
1.212121 4779688
1.818182 1963511
2.424243 562914
3.030303 113454
3.636364 15710
4.242424 1561
4.848485 105
5.454546 7
6.060606 0
 . . .
9.696970 0
```

$$p(x) = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{x^2}{2}\right)$$



CUDA C Programming Guide v10.1
https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

Arithmetic Functions：

atomicAdd()

int atomicAdd(int* address, int val);
unsigned int atomicAdd(unsigned int* address, unsigned int val);
unsigned long long int atomicAdd(unsigned long long int* address,
unsigned long long int val);
float atomicAdd(float* address, float val);
double atomicAdd(double* address, double val);
__half2 atomicAdd(__half2 * address, __half2 val);
__half atomicAdd(__half * address, __half val);

reads the 16-bit, 32-bit or 64-bit word old located at the address address in global or shared memory, computes (old + val), and stores the result back to memory at the same address. These three operations are performed in one atomic transaction. The function returns old.
The 64-bit floating-point version of atomicAdd() is only supported by devices of compute capability 6.x and higher. The 32-bit __half2 floating-point version of atomicAdd() is only supported by devices of compute capability 6.x and higher. The 16-bit __half floating-point version of atomicAdd() is only supported by devices of compute capability 7.x and higher.

atomicSub()

int atomicSub(int* address, int val);
unsigned int atomicSub(unsigned int* address, unsigned int val);

atomicExch()

int atomicExch(int* address, int val);
unsigned int atomicExch(unsigned int* address, unsigned int val);
unsigned long long int atomicExch(unsigned long long int* address,
unsigned long long int val);
float atomicExch(float* address, float val);

reads the 32-bit or 64-bit word old located at the address address in global or shared memory and stores val back to memory at the same address. These two operations are performed in one atomic transaction. The function returns old.

atomicMin() atomicMax() atomicInc() atomicDec() atomicCAS()

Bitwise Functions

atomicAnd()
atomicOr()
atomicXor()

Introduction to Pseudo Random Number Generators

There are many situations arisen in various fields for which the mathematical model calls for long sequences of random numbers. An important example in the field of the computational physics is the Monte Carlo simulations of spin models and lattice field theories.

However, a sequence of truly random numbers cannot be generated according to an algorithm. In practice, we use an algorithm to generate a sequence of pseudo random numbers which are only good approximation of the truly random numbers.

Introduction to PRNG (cont)

The quality of a pseudo random number generator can be measured in terms of the following properties:

1. RANDOMNESS
   good distribution, uniformity, and independence
2. LONG PERIOD
3. PORTABILITY and REPRODUCIBILTY
4. SPEED and EFFICIENCY

The results of Monte Carlo simulations depend crucially on the randomness of the sequence of pseudo-random numbers. Thus, independent simulations with different RNG's should be performed in order to check the validity of the results, if it can be carried out in practice.

Introduction to PRNG (cont)

In principle, there are infinite number of algorithms to generate a sequence of pseudorandom numbers.

For any one of these PRNGs, its period must be finite, no matter how long it is, and there must exist at least one test which can reveal its non-randomness.

In fact, there are infinite number of tests for the randomness. In other words, it is always open for a better PRNG on one hand, and a more stringent test for its randomness on the other. This makes the subject of PRNG alive, interesting, competitive, and challenging.

MULTIPLICATIVE LINEAR CONGRUENTIAL GENERATOR

$$x_{i+1} = A x_i \pmod{M}$$

which has *the maximum period M−1*, since the zero must be excluded from the sequence.

The maximum period *M – 1* can be achieved if *M is a prime and A is a positive prime root of M*,

$$A^M = 1 \pmod{M}$$
$$A^k \neq 1 \pmod{M} \quad \text{for } k < M$$

MULTIPLICATIVE LINEAR CONGRUENTIAL GENERATOR (cont)

The choice for the modulus M is usually dictated by the word length of the computer. For most machines which use 32 bits (4 bytes) to represent an integer, the most significant bit is reserved for denoting the sign, the remaining 31 bits gives the largest positive integer $2^{31}-1$

The largest prime number within this range is $2^{31}-1$

(This is a **Mersenne prime, a prime number of the form** 2^p-1 **where the exponent** $\boldsymbol{p}$ **is also a prime number**).

The positive prime root A of $M=2^{31}-1$ has the following two good choices:

$$A = 16807$$

$$A = 742938285$$

MULTIPLICATIVE LINEAR CONGRUENTIAL GENERATOR (cont)

The integer sequence can be converted to floating-point number sequence of uniform deviate in the interval (0, 1) by normalization $u_i = \frac{x_i}{M}, \quad u_i \in (0,1)$

```
    void RandomUniform(float* data, long n) {
      for(long i=0; i<n; i++)
        data[i] = rand()/(float)RAND_MAX;
    }
```

The drawbacks of MLCG are :

- The period $2^{31} - 2 \approx 10^9$ is too short for any serious Monte Carlo simulations of a system of moderate size. The sequence can be easily consumed by any PC in a few hours.
- Strong short-range correlations.

SHIFT-REGISTER SEQUENCE GENERATOR (Tausworthe 1965)

An algorithm for producing pseudo-random bits of effectively unlimited period was proposed by Tausworthe in 1965. Let $\{a_n\}$ be a sequence pseudo-random bits generated by the linear recursion formula

$$a_k = \sum_{i=1}^{p} c_i a_{k-i} \pmod{2}$$

for a given set of bits $\{c_i, i = 1, \cdots, p\}$. Since a_k is determined by the previous p bits, thus the maximum period is $2^p - 1$ (minus 1 is to subtract off the case when the initial p bits are all zero). The maximum period is attained if and only if the polynomial

$$f(y) = 1 + \sum_{i=1}^{p} c_i y^i$$

is irreducible over GF(2), the Galois finite field of 2 elements, {0, 1}, and p and $2^p - 1$ are both prime numbers.

Example $P(x) = 1 + x^2 + x^5$ is irreducible over GF(2)

1. x is not a factor of $P(x)$
2. $x+1$ is not a factor of $P(x)$ since $P(1)=1+1+1=3=1\neq 0$
3. x^2 is not a factor of $P(x)$
4. $x^2 + x = x(x + 1)$ is not a factor of $P(x)$
5. $x^2 + 1 = x^2 + 2x + 1 = (x + 1)^2$ is not a factor of $P(x)$
6. $x^2 + x + 1$ is not a factor of $P(x)$

Hence all factors of P(x) must have degrees ≥ 3. But the product of any two such factors would give degrees ≥ 6. We deduce that P(x) is irreducible over GF(2). Since $2^5-1 = 31$ is a Mersenne prime, we can generate a sequence of random bits of maximum period=31, starting with any initial five bits (except all zeros), according to the algorithm

$$a_k = a_{k-2} + a_{k-5} \pmod{2} \Leftrightarrow a_{k+p} = a_{k+p-q} + a_k \pmod{2}$$

To generate a sequence of random integers, it is natural to group consecutive r bits in the sequence of random bits to form the binary representation of a sequence of random integers (Tausworthe, 1965). However, such sequence of random integers has serious defects. In 1973, Lewis and Payne proposed to generate a sequence of r-bit integers from r independent sequences of random bits, each independent sequence of random bits is generated according to

$$a_k^m = \sum_{i=1}^{p} c_i a_{k-i}^m \pmod{2}, \quad m = 1, \cdots, r$$

Therefore the recursion formula is generalized to

$$x_k = c_1 x_{k\text{-}1} \oplus \cdots \oplus c_{p\text{-}1} x_{k\text{-}p+1} \oplus c_p x_{k\text{-}p}$$

where $\oplus$ is the bitwise exclusive or operation, and x_k is the sequence of random r-bit integer.

However, the conditions that f(y) is irreducible over GF(2), and both p and $2^p - 1$ are prime numbers can no longer ensure that the period is maximum, unless the initial p seeds of the r-bit integers are linearly independent.

The seeds can be visualized as a $r \times p$ matrix of elements of bits. If there is any “regular pattern” in this seeds matrix, the regularity will be manifested in the sequence and thus the integer sequence will not be satisfactorily random. Consider the trivial case that the i-th and j-th bits are identical in each of the first p seeds, then they will remain identical throughout.

Therefore one of the basic criteria of “good seeds” is that the row vectors of the seeds matrix must be linearly independent. It seems to us that the column vectors of the seeds matrix should also be linearly independent. In general the requirements of linear independence on the column vectors as well as the row vectors respectively, together with the absence of any ”visible pattern” in the graphic display of the seeds matrix would guarantee the sequence to be satisfactorily random. Of course, this is only our empirical experience and certainly not a rigorous mathematical proof.

In 1987, Chiu and Guu **[Comp. Phys. Comm. 47 (1987) 129]** implemented a shift register random number generator on the microcomputers with 8088/8086 and 8087, we used a new initialization procedure as follows.
The first p seeds were generated by the multiplicative linear congruential algorithm

$$x_{i+1} = 16807\, x_i \quad (\mathrm{mod}\ \ 2^{31} - 1)$$

The linear independence of the seeds is ensured by bitwise exclusive or with the system time in 4 bytes (hours, minutes, seconds, and milliseconds) from the internal clock of the microcomputer. We found that this procedure does give a satisfactory sequence of random numbers. In practice, we only employ the primitive trinomials $f(y) = 1 + y^q + y^p$, and the algorithm

$$x_k = x_{k\text{-}p} \oplus x_{k\text{-}q}$$

p	q
2	1
3	1
5	2
7	1, 3
17	3, 5, 6
31	3, 6, 7, 13
89	38
127	1, 7, 15, 30, 63
521	32, 48, 158, 168
607	105, 147, 273
1279	216, 418
2281	715, 915, 1029
3217	67, 576
4423	271, 369, 370, 649, 1393, 1419, 2098
9689	84, 471, 1836, 2444, 4187
. . .	. . .
19937	881, 7083, 9842
. . .	. . .

In principle, this table has no end and thus we have effectively unlimited period for a shift-register sequence of random umbers.
[https://maths-people.anu.edu.au/~brent/ftp/trinom/table.txt]

Mersenne Twister

Mersenne Twister generator of M. Matsumoto and T. Nishimura uses a variant of the twisted generalized feedback shift-register algorithm. It has a Mersenne prime period of 2^19937 - 1 (~ 10^6000) and is equi-distributed in 623 dimensions.
It has passed the diehard statistical tests.
It uses 624 words of state per generator and is comparable in speed to the other generators.
It is widely recognized as one of the best PRNGs.
More information can be found at
http://en.wikipedia.org/wiki/Mersenne_Twister

Mersenne Twister generator has been implemented in the GNU Scientific Library (gsl)

Using Mersenne Twister PRNG in gsl

```
//   To compile, type the following command.
//   gcc -o rng_mt rng_mt.c –lgsl –lgslcblas -lm

#include <gsl/gsl_rng.h>

int main(void){
   gsl_rng *rng;         // pointer to a random number generator
   double r;             // a random number

   rng = gsl_rng_alloc(gsl_rng_mt19937);  // allocate generator
   gsl_rng_set(rng, 12345);               // set the seed to 12345
   r = gsl_rng_uniform(rng);           // generate a random number
   printf("%f\n",r);
   return 0;
}
```

PRNG WITH A SPECIFIED DISTRIBUTION

Heat Bath Method

Given a probability distribution $f(x)$ such that $\int_a^b f(x)\,dx = 1$
we want to generate a sequence of x in the domain (a, b)
having the probability distribution $f(x)$.
Consider the integral

$$F(x) \equiv \int_a^x f(t)\,dt$$

satisfying $F(a) = 0$ and $F(b) = 1$.
Now define $y = F(x)$, $dy = f(x)dx$

Obviously, y has uniform probability distribution in [0, 1].
Thus x can be generated by solving the equation

$$x = F^{-1}(y)$$

In other words, if y is generated by a RNG of uniform deviate in (0, 1), then $x = F^{-1}(y)$ has the desired distribution f(x).

PRNG WITH EXPONENTIAL DISTRIBUTION

The simplest example is the exponential distribution

$$f(x) = \exp(-x) \qquad 0 < x < \infty$$

Then $$x = -\ln(1-y)$$

can be generated by sampling $y \in [0, 1]$ with uniform deviate.

In general, the integral F(x) as well as its inverse function cannot be done analytically, and numerical solutions will render this method impractical.

PRNG WITH GAUSSIAN DISTRIBUTION

Random numbers with Gaussian distribution

$$p(x) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left[-\frac{(x-\mu)^2}{2\sigma^2} \right]$$

are often used in Monte Carlo simulations
(e.g. , molecular dynamics).

Box-Muller Method

Consider points (x_1,x_2) in the plane where x_1 and x_2 both have Gaussian distribution

$$p(x_1) = \frac{1}{\sqrt{2\pi}} \exp\left[-\frac{x_1^2}{2}\right] \qquad p(x_2) = \frac{1}{\sqrt{2\pi}} \exp\left[-\frac{x_2^2}{2}\right]$$

Then the number of points in $dx_1 dx_2$ is proportional to

$$p(x_1)\,p(x_2)\,dx_1 dx_2 = \frac{1}{2\pi} e^{-\frac{x_1^2+x_2^2}{2}} dx_1 dx_2 = e^{-\frac{r^2}{2}} r dr d\theta = e^{-u} du d\theta$$

Hence, if we generate u ∈ (0,∞) with an exponential distribution , and θ uniformly between 0 and 2π , then

$$x_1 = \sqrt{2u}\cos\theta \qquad x_2 = \sqrt{2u}\sin\theta$$

both will have the desired Gaussian distribution.

Box-Muller Method (cont)

```
void RandomNormal(float* data, long n) {
    // RNG with Gaussian distribution, mu=0 and sigma=1

  const float Pi = acos(-1.0);

  for(long i=0; i< n; i++) {
    double y = (double) rand() / (float)RAND_MAX;
    double x = -log(1.0-y);
    double z = (double) rand() / (float)RAND_MAX;
    double theta = 2*Pi*z;
    data[i] = (float) (sqrt(2.0*x)*cos(theta));
  }
}
```