Random number generation (in C++)
- past, present and potential future

Pattabi Raman
pattabi@numericalsolution.co.uk
What is Random?

It represents an event or entity, which cannot be determined but only described probabilistically,

For example:

• Falling rain drops which hit the ground in random pattern,

• Distribution of stars within the universe is a random, and

• Babies cry in random and so on
What are the use of Random Numbers?

They are mainly used in:

• Simulations be it a numerical calculation or a cartoon game,
• Binning analog data in channel format,
• Testing a product and so on.
What is the plan of this presentation?

• Historic evolution of random numbers and their applications,

• Modern development and implementation in C++, and beyond.
How did random numbers and their applications evolve?
Since pre-historic period, random numbers were generated from dice for gambling
In bronze age gambling became unethical

Random numbers from dice

Made Kings to give up their crowns to opponents

http://www.numericalsolution.co.uk/random-numbers-and-simulations/
In 300 BC, Prof Chanakya of Takshashila University prescribed laws and taxations to regulate gambling.
In 1650 Blaise Pascal, who introduced the computer in the form of his mechanical calculator
Designed perpetual motion experiment
That lead the way to the Roulette
Casino de Monte-Carlo
Monaco - MC 98000
Casino de Monte-Carlo
Monaco - MC 98000
In 1930

FERMIAC The Monte Carlo trolley

statistical sampling techniques

Random number generator

http://www.numericalsolution.co.uk/random-numbers-and-simulations/
In 1940

While playing solitaire during his recovery from a surgery, he had thought about playing hundreds of games to estimate statistically the probability of a successful outcome.

This lead to the idea of Monte Carlo Method!
ENIAC - first electronic general-purpose computer

Used Monte Carlo Method and employed random numbers to solve complicated problems.

Obtained random numbers from:
- nuclear radioactivity;
- voltage fluctuation;
- Solar flare and
stored in punch cards.

But reading from punch card was very slow!
ENIAC - first electronic general-purpose computer

So, he developed pseudo-random numbers, using the 13\textsuperscript{th} century mid-square method

Let, Seed $x_0 = 0.7891$, then

\[ x_0^2 = 0.62 \, 2678 \, 81 \]
\[ \Rightarrow x_1 = 0.2678 \]

\[ x_1^2 = 0.07 \, 1716 \, 84 \]
\[ \Rightarrow x_2 = 0.1716 \]

\[ x_2^2 = 0.02 \, 9446 \, 56 \]
\[ \Rightarrow x_3 = 0.9446 \]

"Any one who considers arithmetical methods of producing random digits is, of course, in a state of sin."

John von Neumann

Used Monte Carlo Method and employed random numbers to solve complicated problems.
He applied Monte Carlo Methods and

John von Neumann
He applied Monte Carlo Methods and

John von Neumann

Fig. 2. Forecast of January 5, 1949, 0300 GMT: (a) observed $z$ and $\eta$ at $t = 0$; (b) observed $z$ and $\eta$ at $t = 24$ hours; (c) observed (continuous lines) and computed (broken lines) 24-hour height change; (d) computed $z$ and $\eta$ at $t = 24$ hours. The height unit is 100 ft and the unit of vorticity is $153 \times 10^{-4}$ sec$^{-1}$. 

http://www.numericalsolution.co.uk/random-numbers-and-simulations/
He applied Monte Carlo Methods and

John von Neumann
He applied Monte Carlo Methods and

John von Neumann

http://www.numericalsolution.co.uk/random-numbers-and-simulations/
He applied Monte Carlo Methods and

John von Neumann

replicators
In 1957

**Fortran (Formula Translating System)**

Multiplicative Congruential Generator (MCG)

\[ X_n = (aX_{n-1}) \mod m \]

```fortran
RANDOM = MOD(A*SEED,M)
PRINT*, RANDOM
SEED = RANDOM
```

Linear congruential generator

\[ X_n = (aX_{n-1} + b) \mod m \]
In 1972

static unsigned long int next = 1;

int rand(void) // RAND_MAX assumed to be 32767
{
    static const unsigned long int a = 1103515245;
    static const unsigned short b = 12345;
    next = next * a + b;
    return (unsigned int)(next/65536) % 32768;
}

void srand(unsigned int seed)
{
    next = seed;
}
In 1972

```c
static unsigned long int next = 1;

int rand(void) // RAND_MAX assumed to be 32767
{
    static const unsigned long int a = 1103515245;
    static const unsigned short b = 12345;
    next = next * a + b;
    return (unsigned int)(next/65536) % 32768;
}

void srand(unsigned int seed)
{
    next = seed;
}
```
```c
#include <cstdlib>
#include <cstdio>
#include <ctime>

int main() {
    srand(time(NULL));
    printf("Random numbers:\n");
    float random = 0;
    for (int i = 0; i < 10; ++i) {
        random = (float) rand()/RAND_MAX;
        printf("%f\n", random);
    }
}
```

Random numbers:

0.003143
0.569353
0.033021
0.545427
0.627430
0.035096
0.817377
0.055635
0.282266
0.187628
→ BCPL
→ 1972 C (proximity to hardware)

1967 Simula (OOP)
→ ALGOL
→ Fortran
→ Speedcoding

1983 C++
C with Classes (of Simula)

Machine code (100011 00011)
→ Assembly language (mov ax,@data)
int rand (void);

void srand (unsigned int seed);

#define RAND_MAX 32767

std::random_shuffle (data.begin(), data.end());

Initial data is {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}

Shuffled data is {6, 10, 5, 2, 3, 7, 9, 8, 1, 4}
Random Numbers - Numberphile
http://www.youtube.com/watch?v=SxP30euw3-0
Improvements to TRI’s Facility for Random Number Generation

Contents

1 Introduction 1
2 A brief history 2
3 General structure 2
4 New and changed requirements 2
5 Header <random> synopsis 5
6 The demise of variate_generator 5
7 Random number engine adaptor class templates 7
8 Engines with predefined parameters 7
9 Random number distribution class templates 7
10 Conclusion 8
11 Acknowledgments 8
Bibliography 8

If the numbers are not random, they are at least higgledy-piggledy.
— GEORGE MARSAGLIA

I cannot do it without compters.
— WILLIAM SHAKESPEARE
In 2011 C++ 11

\texttt{<random>} (26.5 Random number generation N3797)

\texttt{Distribution(Engine)}
In 2011

C++ 11

<random> (26.5 Random number generation N3797)

Random = Distribution(Engine)

Engines:
- linear_congruential_engine
- subtract_with_carry_engine
- mersenne_twister_engine

Engines Adaptors:
- discard_block_engine
- independent_bits_engine
- shuffle_order_engine

True random number generator:
- random_device

Distributions:
- Uniform distributions
- Normal distributions
- Bernoulli distributions
- Rate-based distributions
- Piecewise distributions
- Canonical numbers

http://www.numericalsolution.co.uk/random-numbers-and-simulations/
linear_congruential_engine

\[ X_n = (aX_{n-1} + b) \mod m \]

**minstd_rand0:** \( a = 16807; b = 0; m = 2, 147, 483, 647 \)

**minstd_rand:** \( a = 48271; b = 0; m = 2, 147, 483, 647 \)
linear_congruential_engine

\[ X_n = (aX_{n-1} + b) \mod m \]

\text{minstd\_rand0:} \quad a = 16807; \ b = 0; \ m = 2, 147, 483, 647

\text{minstd\_rand:} \quad a = 48271; \ b = 0; \ m = 2, 147, 483, 647
**subtract_with_carry_engine**

1991 George Marsaglia and Arif Zaman

Linear Congruential algorithm => $X_n = (aX_{n-1} + b) \mod m$

Lagged Fibonacci algorithm => $f(X_{n-1}, X_{n-2}) \Rightarrow f(X_{n-S}, X_{n-R})$; where, $S < R < 0$

$$X_n = (X_{n-S} - X_{n-R} - cy_{n-1}) \mod m$$

where, $cy_n = (X_{n-S} - X_{n-R} - cy_{n-1} < 0) \, ? \, 1 : 0$;

**ranlux24_base**: 24-bit number $S = 10; R = 24$;

**ranlux48_base**: 48-bit number $S = 5; R = 12$;
mersenne_twister_engine

1997 Makoto Matsumoto and Takuji Nishimura
“Twisted Generalized Feedback Shift Register”

Period length = Mersenne prime = $2^{19937} - 1$

MT19937 uses a 32-bit word length

MT19937-64 uses a 64-bit word length
random_device

Generates random numbers from hardware where available

```cpp
crtimp2_pure unsigned int __clrcall_pure_or_cdecl _Random_device();

unsigned int operator()()
{
    return (_Random_device());
}

std::random_device rd;

std::default_random_engine e1(rd());```
void my_random_generator::check_mt19937()
{
    mt19937 engine;
    static long long random_number = 0;

    for(int i =0; i != 9999; ++i) engine();
    random_number = engine();

    // If the implementation is correct then the 10000th consecutive invocation of a default-constructed object of type mt19937 shall produce the value 4,123,659,995, ref. C++11 ISO statement.
    if(random_number == 4123659995)
        cout << "\n Note:\n\t The pseudorandom number generator\n\t "
        << "Mersenne twister: MT19937 has been tested\n\t and "
        << "it shows it is implemented properly.\n"
    else
        cout << "\nWarning:\n\t "
        << "The pseudorandom number generator\n\t "
        << "Mersenne twister: MT19937 has been "
        << "tested\n\t and it shows it is NOT "
        << "implemented properly.\n";
}

Required behaviour: The 10000th consecutive invocation of a default-constructed object of type mt19937 shall produce the value 4123659995.
// A Uniform Distribution based on default_random_engine:

// Using Bind function:

auto dist = bind(uniform_real_distribution<double>{0.0, 1.0}, default_random_engine{});

Random = dist();
class uniform_dist {

public:

    double operator()() { return uniform(engine); }

uniform_dist():uniform(0.0, 1.0) {}

void discard(unsigned long long z) { engine.discard(z); }

void discard_distribution(unsigned long long z)
{
    for (auto i = z; i != 0; --i)
        uniform(engine);
}

uniform_dist(double low, double high):uniform(low, high) {}  

private:

    default_random_engine engine;

    uniform_real_distribution< double > uniform;
};
// A Normal distribution based on
// Mersenne Twister engine:

class normal_dist {
public:
  double operator()() { return normal(engine); }

  normal_dist() : normal {0.0, 1.0} {}

  void discard(unsigned long long z) { engine.discard(z); }

  void discard_distribution(unsigned long long z)
  {
    for (auto i = z; i != 0; --i)
      normal(engine);
  }

  normal_dist(double mean, double std_dev) : normal( mean, std_dev ) {}

private:

  mt19937 engine;

  normal_distribution<double> normal;
};
Normal distribution

\[ \phi(x) \]

\( \mu = 0 \)

\( \sigma^2 = 1 \)
Cumulative normal distribution

CDF

\( \mu = 0 \)

\( \sigma^2 = 1 \)

Probability normal distribution

\( \phi(X) \)

\( \mu = 0 \)

\( \sigma^2 = 1 \)
Cumulative normal distribution

CDF

μ = 0
σ² = 1

Probability normal distribution

μ = 0
σ² = 1

http://www.numericalsolution.co.uk/random-numbers-and-simulations/
Cumulative normal distribution

- CDF
- $\mu = 0$
- $\sigma^2 = 1$
- $\approx$ Uniform distribution

Probability normal distribution

- $\mu = 0$
- $\sigma^2 = 1$

http://www.numericalsolution.co.uk/random-numbers-and-simulations/
Pseudo-code algorithm for rational approximation

The algorithm below assumes \( p \) is the input and \( x \) is the output.

Coefficients in rational approximations.
\[
\begin{align*}
a(1) &< -3.969683028665376e+01 \\
a(2) &< -2.209460984245205e+02 \\
a(3) &< -2.759285104469687e+02 \\
a(4) &< -1.383577518672690e+02 \\
a(5) &< -3.066479806614716e+01 \\
a(6) &< -2.506628277459239e+00 \\
b(1) &< -5.447609879822406e+01 \\
b(2) &< 1.615858368580409e+02 \\
b(3) &< -1.556989798598866e+02 \\
b(4) &< 6.680131188771972e+01 \\
b(5) &< -1.328068155288572e+01 \\
c(1) &< -7.784894002430293e-03 \\
c(2) &< -3.223964580411365e-01 \\
c(3) &< -2.400758277161838e+00 \\
c(4) &< -2.549732539343734e+00 \\
c(5) &< 4.374664141464968e+00 \\
c(6) &< 2.938163982698738e+00 \\
d(1) &< 7.784695709041462e-03 \\
d(2) &< 3.224671290700398e-01 \\
d(3) &< 2.445134137142996e+00 \\
d(4) &< 3.754408661907416e+00
\end{align*}
\]

Define break-points.
\[
\begin{align*}
p\_low &< 0.02425 \\
p\_high &< 1 - p\_low
\end{align*}
\]

Rational approximation for lower region.
\[
\begin{align*}
\text{if } 0 < p < p\_low & \text{ then } \\
q &< \sqrt{-2\log(p)} \\
x &< (((((c(1)\cdot q+c(2))\cdot q+c(3))\cdot q+c(4))\cdot q+c(5))\cdot q+c(6)) / (((d(1)\cdot q+d(2))\cdot q+d(3))\cdot q+d(4))\cdot q+1)
\end{align*}
\]

Rational approximation for central region.
\[
\begin{align*}
\text{if } p\_low <= p <= p\_high & \text{ then } \\
q &< p - 0.5 \\
r &< q^2 \\
x &< (((((a(1)\cdot r+a(2))\cdot r+a(3))\cdot r+a(4))\cdot r+a(5))\cdot r+a(6)) \cdot q / (((b(1)\cdot r+b(2))\cdot r+b(3))\cdot r+b(4))\cdot r+b(5)\cdot r+1)
\end{align*}
\]

Rational approximation for upper region.
\[
\begin{align*}
\text{if } p\_high < p < 1 & \text{ then } \\
q &< \sqrt{2\log(1-p)} \\
x &< (((((c(1)\cdot q+c(2))\cdot q+c(3))\cdot q+c(4))\cdot q+c(5))\cdot q+c(6)) / (((d(1)\cdot q+d(2))\cdot q+d(3))\cdot q+d(4))\cdot q+1)
\end{align*}
\]

http://home.online.no/~pjacklam/notes/invnorm/index.html

http://www.numericalsolution.co.uk/random-numbers-and-simulations/
Please avoid coding random number generators or distributions!

Make use of your compliers and

http://www.numericalsolution.co.uk/random-numbers-and-simulations/
C++

:: Concise programs from basics to high performance computing

_Dozens of the new C++ 11 / 14 objects and easy to solve questions with answers for your fast and precise practice_

N.S.Pattnaik Raman

www.numericalsolution.co.uk
\[ V_t = V_{t-dt} (1 + r.dt + \text{sigma}.\phi().\sqrt{dt}), \]

```c
for(i = 1; i < T; ++i) {
    dX = phi()*sqrt_dt;
    dS = S*(r_dt + sigma*dX);
    S += dS;
    data[i] = S;
}
```
Your realistic expectation

Sellers' argument

Buyers' argument

House price (in hundreds)

Active hours (excludes holiday hours)

http://www.numericalsolution.co.uk/random-numbers-and-simulations/
Amdahl’s law $\Rightarrow P = \frac{(1-S)N}{S(1-N)}$
<table>
<thead>
<tr>
<th>&lt;random&gt; engine</th>
<th>uniform_int average</th>
<th>stddev</th>
<th>uniform_real average</th>
<th>stddev</th>
<th>normal_distribution average</th>
<th>stdev</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Linear congruential: minstd_rand0</td>
<td>88.00</td>
<td>2.67</td>
<td>91.89</td>
<td>2.97</td>
<td>79.28</td>
<td>3.60</td>
</tr>
<tr>
<td>2. Mersenne twister: mt19937</td>
<td>91.18</td>
<td>2.34</td>
<td>88.67</td>
<td>3.47</td>
<td>79.60</td>
<td>2.61</td>
</tr>
<tr>
<td>3. Mersenne twister: mt19937_64</td>
<td>90.68</td>
<td>2.27</td>
<td>87.73</td>
<td>3.10</td>
<td>76.38</td>
<td>3.04</td>
</tr>
<tr>
<td>4. Shuffle order: knuth_b</td>
<td>83.77</td>
<td>4.29</td>
<td>86.75</td>
<td>6.12</td>
<td>70.45</td>
<td>5.71</td>
</tr>
<tr>
<td>5. Subtract with carry: ranlux24_base</td>
<td>88.09</td>
<td>2.14</td>
<td>83.62</td>
<td>4.74</td>
<td>Dead Lock!</td>
<td></td>
</tr>
<tr>
<td>6. Discard block: ranlux24</td>
<td>Dead Lock!</td>
<td>Dead Lock!</td>
<td>Dead Lock!</td>
<td>Dead Lock!</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

![Graph showing % of Parallel code vs <random> engine number](http://www.numericalsolution.co.uk/random-numbers-and-simulations/)

<random> engine numbers:
1. minstd_rand0
2. mt19937
3. mt19937_64
4. knuth_b
5. ranlux24_base
6. ranlux24

(Dead Lock with Normal Distribution)
(Dead Lock with all!)
Spring 2013 JTC1/SC22/WG21
C++ Standards Committee Meeting

Bristol, UK, April 15 – 20, 2013
(note: this is 6 days: Mon - Sat)

The meeting venue and host hotel is the Marriott Hotel, Bristol City Centre.
2 Lower Castle Street, Old Market, Bristol, England BS1 3AD
http://www.marriott.co.uk/hotels/travel/brsdt-bristol-marriott-hotel-city-centre/
Thanks to:
Hans Boehm, Lawrence Crowl, Mike Giles, Peter Jäckel, Stephan T. Lavavej, Nick Maclaren
Alisdair Meredith, Roger Orr, I.M. Sobol’, Herb Sutter, Jonathan Wakely, Michael Wong and more

```cpp
template<class UIntType, size_t w, ...>
class mersenne_twister_engine
{
    public:
        ....
        ....
        ....
        explicit mersenne_twister_engine(result_type value = default_seed);
        template<class Sseq> explicit mersenne_twister_engine(Sseq& q);
        void seed(result_type value = default_seed);
        template<class Sseq> void seed(Sseq& q);

    // generating functions
    result_type operator()() const; // if const method then it would be safe for concurrency.
    void discard(unsigned long long z);
};
```
Thanks to:

Hans Boehm, Lawrence Crowl, Mike Giles, Peter Jäckel, Stephan T. Lavavej, Nick Maclaren, Alisdair Meredith, Roger Orr, I.M. Sobol, Herb Sutter, Jonathan Wakely, Michael Wong and more

```cpp

template<class UIntType, size_t w, ...>
class mersenne_twister_engine
{
    public:
        ....
        ....
        ....
        explicit mersenne_twister_engine(result_type value = default_seed);
        template<class Sseq> explicit mersenne_twister_engine(Sseq & q);
        void seed(result_type value = default_seed);
        template<class Sseq> void seed(Sseq & q);

    // generating functions
    result_type operator()() const; // but it is not const method.
    void discard(unsigned long long z);
};
```
Thanks to:
Hans Boehm, Lawrence Crowl, Mike Giles, Peter Jäckel, Stephan T. Lavavej, Nick Maclaren
Alisdair Meredith, Roger Orr, I.M. Sobol’, Herb Sutter, Jonathan Wakely, Michael Wong and more

```
template<class UIntType, size_t w, ...>
class mersenne_twister_engine
{
    public:
    ....
    ....
    ....
    explicit mersenne_twister_engine(result_type value = default_seed);
    template<class Sseq> explicit mersenne_twister_engine(Sseq & q);
    void seed(result_type value = default_seed);
    template<class Sseq> void seed(Sseq & q);

    // generating functions
    result_type operator()(); // As it is not const method, it is not safe for concurrency!
    void discard(unsigned long long z);
};
```
Thanks to:
Hans Boehm, Lawrence Crowl, Mike Giles, Peter Jäckel, Stephan T. Lavavej, Nick Maclaren
Alisdair Meredith, Roger Orr, I.M. Sobol, Herbs Sutter, Jonathan Wakely, Michael Wong and more

• Declare random number generator as thread_local, then each thread can have independent copy of the random object.
• But, it will provide same set of numbers in all threads, so that is multiplication of same data, therefore, that cannot add to statistics!
Thanks to:
Hans Boehm, Lawrence Crowl, Mike Giles, Peter Jäckel, Stephan T. Lavavej, Nick Maclaren Alisdair Meredith, Roger Orr, I.M. Sobol, Herb Sutter, Jonathan Wakely, Michael Wong and more

• Construct independent engines for each thread and set different range!
// A Normal distribution based on
// Mersenne Twister engine:

class normal_dist {
public:
    double operator()() { return normal(engine); }

    normal_dist() : normal {0.0, 1.0} {}  

    void discard(unsigned long long z) {
        engine.discard(z); }

    void discard_distribution(unsigned long long z) {
        for (auto i = z; i != 0; --i)  
            normal(engine);
    }

    normal_dist(double mean, double std_dev) : normal(mean, std_dev) {}  

private:
    mt19937 engine;  

    normal_distribution<double> normal;
};
Complexity of ‘discard(number)’, i.e., performance time should be reduced, so number theoreticians can help!

→ It multiplies the data not increasing the statistics
Quasi-Random Numbers

• What are they?
• How do they differ from Pseudo-random numbers?
• Where do they help?
Quasi-Random Numbers
Quasi-Random Numbers
Quasi-Random Numbers
Quasi-Random Numbers
Quasi-Random Numbers
Quasi-Random Numbers
Quasi-Random Numbers
Low-discrepancy sequence

Pseudo-Random Numbers

$\epsilon \propto c(d) \frac{(ln N)^d}{N}$

$d$ is dimension, i.e., $d$ numbers of random numbers were estimated per iteration.

Halton
Niederreiter-Xing
Sobol

Linear congruential
Subtract with carry
Mersenne twister

$\epsilon \propto \frac{1}{\sqrt{N}}$

N is number of iterations.

http://www.numericalsolution.co.uk/random-numbers-and-simulations/
Quasi-Random Numbers
Low-discrepancy sequence

Halton
Niederreiter-Xing
Sobol
–it uses primitive polynomial
based bitwise arithmetic
without carry - XOR

\[ \epsilon \propto c(d) \frac{(\ln N)^d}{N} \]

d is dimension, i.e.,
d numbers of random numbers were estimated
per iteration.

\[ y = f(x, \text{random}()) + g(x, \text{random}()); \rightarrow \text{equation 1} \]
\[ P = f(Q, \text{random}()) + g(R, \text{random}()) + h(T, \text{random}()); \rightarrow \text{equation 2} \]

Pseudo-Random Numbers

Linear congruential
Subtract with carry
Mersenne twister

\[ \epsilon \propto \frac{1}{\sqrt{N}} \]

N is number of iterations.

http://www.numericalsolution.co.uk/random-numbers-and-simulations/
“Most programmers will soon discover that the rand() function is completely inadequate because it can only generate a single stream of random numbers.

Most games need multiple discrete streams of random numbers”.
Potential candidates for C++17:

- Dimensionality to random number generations;
- Low-discrepancy sequence (Sobol numbers);

```cpp
int pick_a_number( int from, int thru )
{
    static std::uniform_int_distribution<> d{};
    using parm_t = decltype(d)::param_type;
    return d( global_urng(), parm_t{from, thru} );
}

double pick_a_number( double from, double upto )
{
    static std::uniform_real_distribution<> d{};
    using parm_t = decltype(d)::param_type;
    return d( global_urng(), parm_t{from, upto} );
}
```
## Hardware Random Number Generator

<table>
<thead>
<tr>
<th>Manufacturer</th>
<th>Throughput (Mbit/s)</th>
<th>Price</th>
<th>Intro Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comscire</td>
<td>32</td>
<td>$1,495</td>
<td>2013</td>
</tr>
<tr>
<td>ID Quantique SA</td>
<td>16</td>
<td>2,230.00 €</td>
<td>2006</td>
</tr>
<tr>
<td>Quant-Lab</td>
<td>12</td>
<td>2,700.00 €</td>
<td>2005</td>
</tr>
<tr>
<td>TectroLabs</td>
<td>1.4</td>
<td>$329</td>
<td>2013</td>
</tr>
<tr>
<td>Flying Stone Tech</td>
<td>0.6</td>
<td>JPY 4,000</td>
<td>2013</td>
</tr>
<tr>
<td>TRNG98</td>
<td>0.5</td>
<td>620.00 €</td>
<td>2009</td>
</tr>
<tr>
<td>ubld.it</td>
<td>0.4</td>
<td>$49.95</td>
<td>2014</td>
</tr>
<tr>
<td>Araneus</td>
<td>0.4</td>
<td>159 €</td>
<td>2003</td>
</tr>
</tbody>
</table>

The chart compares the throughput (Mbit/s) of different hardware random number generators.

cuRAND: Random Number Generation

• Generating high quality random numbers in parallel is hard
  • Don’t do it yourself, use a library!

• Pseudo- and Quasi-RNGs
• Supports several output distributions
• Statistical test results in documentation

• New in CUDA 5.0: Poisson distribution
cuRAND Engines

Normal Distributions (cuRAND Engines)

Double Precision Uniform Distribution

- CURAND XORWOW
- CURAND MRG32k3a
- CURAND MTGP32
- CURAND 32 Bit Sobol
- CURAND 64 Bit Sobol
- CURAND 64 bit Scrambled Sobol
- MKL MRG32k3a
- MKL 32 Bit Sobol

Double Precision Normal Distribution

- CURAND XORWOW
- CURAND MRG32k3a
- CURAND MTGP32
- CURAND 32 Bit Sobol
- CURAND 64 Bit Sobol
- CURAND 64 bit Scrambled Sobol
- MKL MRG32k3a
- MKL 32 Bit Sobol

• cuRAND 4.1 on Tesla M2090, ECC on
• MKL 10.2.3, TYAN FT72-B7015 Xeon x5680 Six-Core @ 3.33 GHz

Performance may vary based on OS ver. and motherboard config.
“Skip ahead or saving state to avoid overlap is the caller’s responsibility”

\[
\text{seed}[n] = \text{seed}[n-1] + 1000; \ (\text{.discard}(i \times \text{iterations});)
\]
Do you realise, our brain is a source of random numbers?!

- It accesses the past and present data, generates random scenarios and simulates future be it next minute or next decade;
- It is a sophisticated random generator, very smooth simulator, but it is bit of biased;
- Pessimistic brain skewed towards negative random;
- Optimistic brain skewed towards positive random;
Thank You!