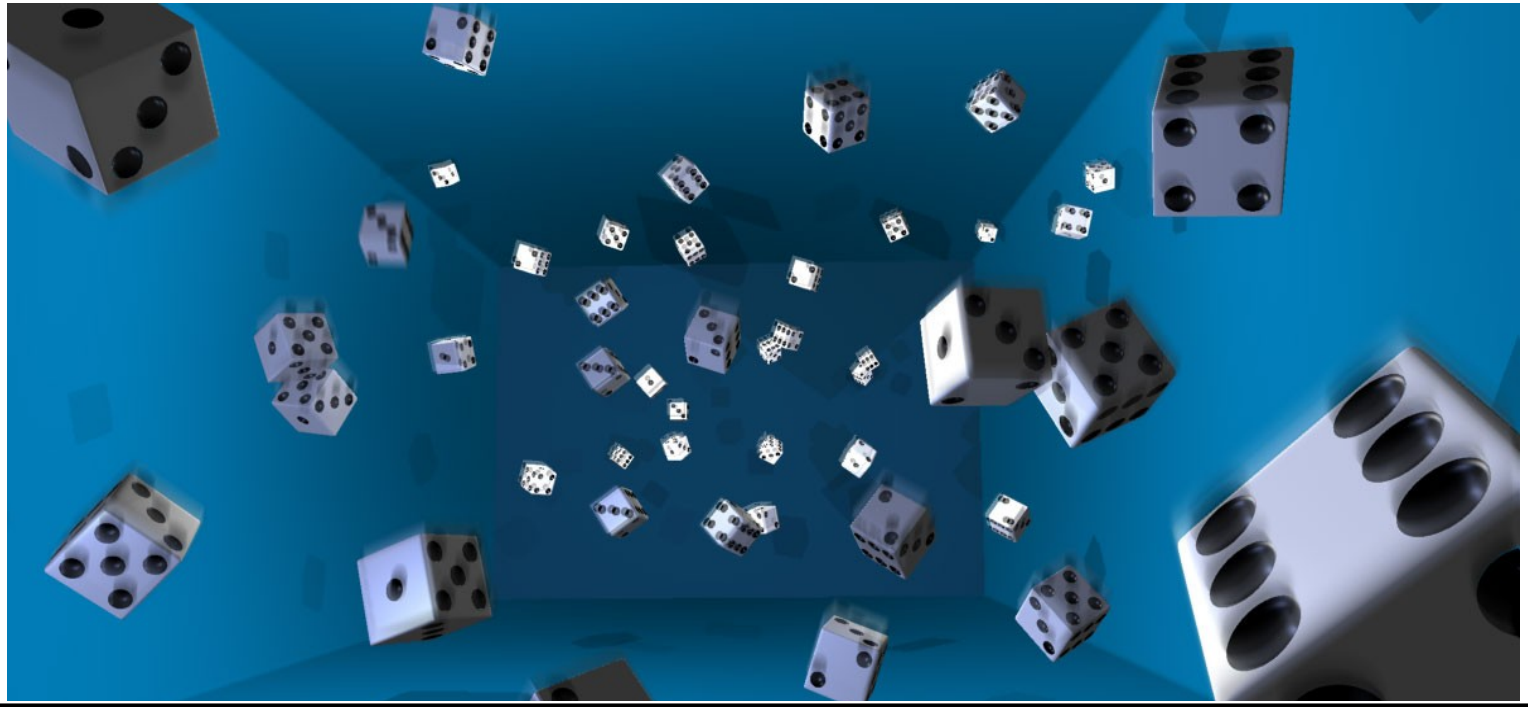
# 2. Método Monte Carlo

- Introducción al Método Monte Carlo.
- Secuencias de números aleatorios y pseudoaleatorios.
- Generadores de números pseudoaleatorios.
- Muestreo de funciones de probabilidad.
- Integración clásica y idoneidad del método Monte Carlo
- Integración mediante Monte Carlo.
- Monte Carlo de Metropolis.

# Introducción al método Monte Carlo

• Las **simulaciones por Monte Carlo** utilizan números generados **aleatoriamente** para calcular las propiedades de un modelo conforme a las **distribuciones de probabilidades asociadas a la respuesta de un sistema**. Vease **[TheBeginning]** sobre el origen del nombre y las técnicas.

• El método se utiliza en principio en cualquier rama de la ciencia, aunque se aplica de forma natural a aquellos procesos en los que se encuentran **distribuciones de probabilidad de sucesos o de ocurrencia de determinadas situaciones** y en los estas **probabilidades sean determinantes para el resultado**, independientemente de que sean básicas en su modelización (como en Mecánica Cuántica) o derivadas de la simplificaciones de una descripción mas compleja (por ejemplo, probabilidades de parámetro de impacto en Mecánica Clásica).



# Introducción al método Monte Carlo (2)

• Por ejemplo: una simulación siguiendo el método de Monte Carlo puede devolver una serie de **valores numéricos como solución a un problema de evolución temporal** de objetos (partículas clásicas o cuánticas) que **interactúan a través de leyes o distribuciones de probabilidad** (secciones eficaces, distribución de momentos o velocidades, ...).

• Estas interacciones se realizan **consecutivamente de forma repetida** y **según la combinación de sus probabilidades** y la **generación de números aleatorios que permita muestrear las distribuciones de probabilidad asociadas a cada interacción**.

• El proceso se sucede hasta obtener **convergencia de los resultados en los observables** que se pretenden determinar (momentos y posiciones de las partículas interaccionantes, energías depositadas, varianzas de cada distribución, ...).

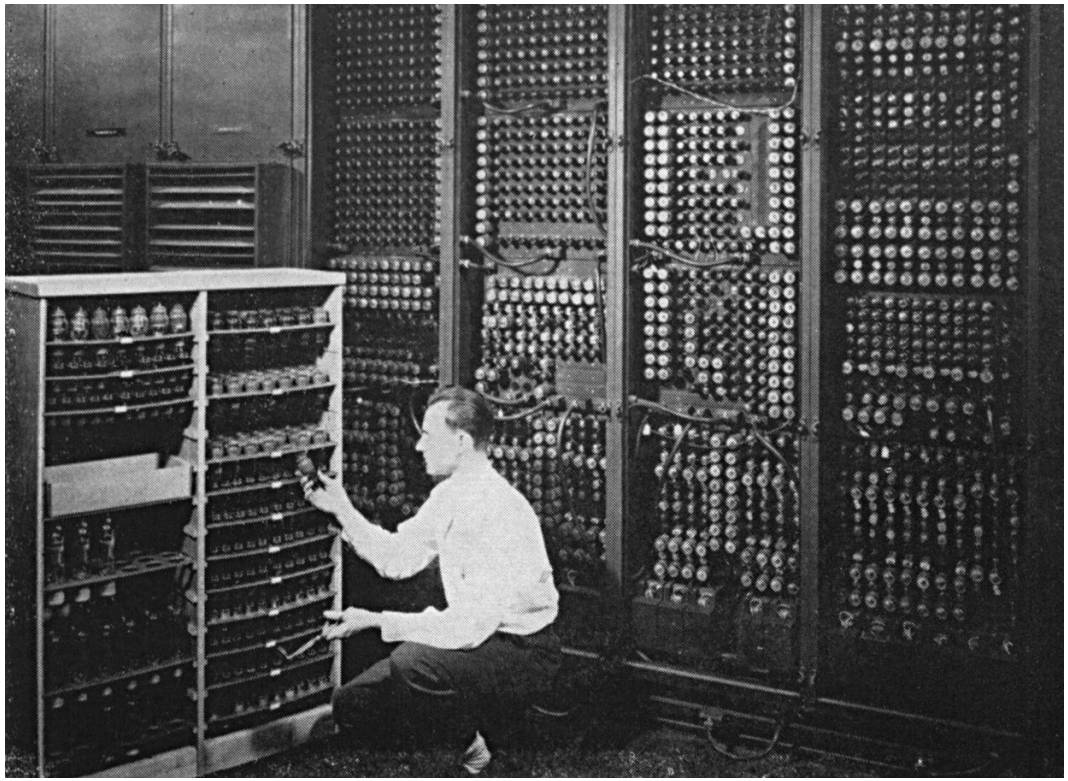
**El método de Monte Carlo pretende obtener una solución a un sistema macroscópico mediante la simulación de sus interacciones microscópicas [Bielajew12].**

# ¿Por que su utiliza el método Monte Carlo?

• **Las técnicas de Monte Carlo resultan adecuadas para tratar problemas complejos o de alta dimensionalidad** (con un espacio de variables de dimensión igual o mayor que 4 **[Bielajew12]**), como son, por ejemplo, los seguimientos de la posición y momento de partículas físicas.

• En función de la complejidad del problema, **los métodos de Monte Carlo tienen ventajas en tiempo computacional y convergencia (varianza) al resultado final frente a soluciones basadas en ecuaciones de transporte**.

• **¿Por qué?** Veremos una demostración matemática más adelante, de forma restringida, utilizando el problema de la varianza en el cálculo de integrales numéricas. Pero **también tiene importancia la forma en la que funcionan los ordenadores**.

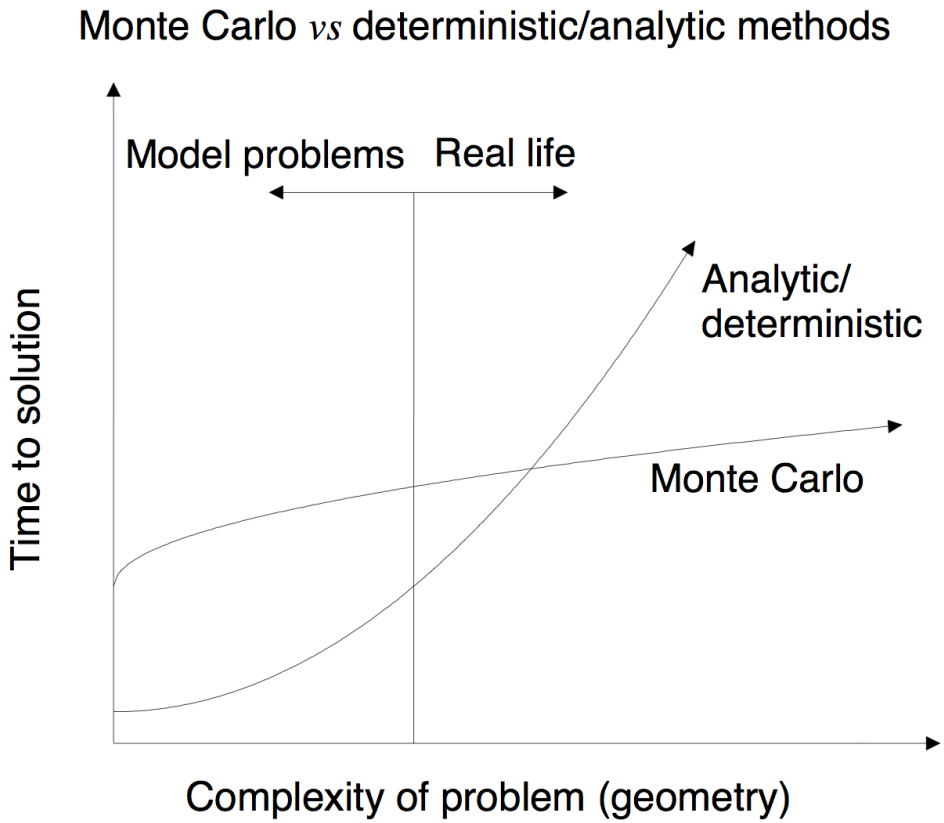


# ¿Por que su utiliza el método Monte Carlo?

• Los ordenadores han seguido en los últimos 40 años la "**ley de Moore**" creciendo su capacidad de computación de forma geométrica (duplicandose cada año y medio o dos años el número de transistores empleados en una CPU).

• Las técnicas Monte Carlo se basan en una **utilización baja (mínima) de datos y un alto uso de calculo en operaciones de punto flotante** (iteraciones, cálculo de aleatorios, transporte, ...), al contrario que en cálculos deterministas.

• Esto se ve **favorecido en velocidad por la estructura de los ordenadores modernos** (memoria cache de alto ancho de banda a la CPU frente al tiempo de acceso a disco).



# Secuencias de números aleatorios

• Una **secuencia de números aleatorios** es aquella que **no muestra un patrón o una regularidad reconocible**. Cada número escogido tiene **la misma probabilidad que cualquier otro** número (en una muestra homogénea) y **no depende de una elección anterior o posterior**.

• Se pueden generar secuencias de números aleatorios mediante **procesos físicos realmente estocásticos** como los procesos de **desintegración radiactiva**, el **ruido térmico**, **probabilidades cuánticas**, **juegos de azar bien diseñados**...

• Existen pruebas para determinar si una secuencia de números es estadísticamente aleatoria, basadas en su frecuencia (homogeneidad en el número de veces que sale una cifra concreta), frecuencia en series de números consecutivos o separación entre ellos, … (Los tests originales "Diehard" de Massaglia no están operativos, pero si "Dieharder" en **http://www.phy.duke.edu/~rgb/General/dieharder.php**, o para una explicación de los tests, **http://en.wikipedia.org/wiki/Diehard_tests**).

Is the number 7 random? If it is generated by a random process, it might be. If it is made up to serve the purpose of some argument (like this one) it is not. Perfect random number generators produce "unlikely" sequences of random numbers -- at exactly the right average rate. Testing a random number generator (rng) is therefore quite subtle.

Extraido de **http://www.phy.duke.edu/ ~rgb/General/dieharder.php**

Ver también la discusión en **https://www.random.org/analysis/**

# Secuencias de números aleatorios

• Una **secuencia de números aleatorios** es aquella que **no muestra un patrón o una regularidad reconocible**. Cada número escogido tiene **la misma probabilidad que cualquier otro** número (en una muestra homogénea) y **no depende de una elección anterior o posterior**.

• Se pueden generar secuencias de números aleatorios mediante **procesos físicos realmente estocásticos** como los procesos de **desintegración radiactiva**, el **ruido térmico**, **probabilidades cuánticas**, **juegos de azar bien diseñados**...

• Existen pruebas para determinar si una secuencia de números es estadísticamente aleatoria, basadas en su frecuencia (homogeneidad en el número de veces que sale una cifra concreta), frecuencia en series de números consecutivos o separación entre ellos, … (Los tests originales "Diehard" de Massaglia no están operativos, pero si "Dieharder" en **http://www.phy.duke.edu/~rgb/General/dieharder.php**, o para una explicación de los tests, **http://en.wikipedia.org/wiki/Diehard_tests**).

# Secuencias de números aleatorios

http://www.microsiervos.com/archivo/azar/random-sanity-project-aleatoriedad.html

AZAR

## Código para comprobar la aleatoriedad de secuencias de números aleatorios

POR @ALVY — 2 DE MAYO DE 2017

Random Sanity Project

### How it Works

1. Get an array of 64 bytes from whatever source of cryptographically-secure randomness you are using
2. Encode them as hex, and make a REST query to the randomsanity service
3. The service will return 'true' if it those bytes look random and nobody else has submitted the same stream of bytes; otherwise it will return 'false'.

**Random Sanity Project** es un servicio web con un único objetivo: comprobar y garantizar que las secuencias de números aleatorios que se generan con todo tipo de software son *realmente* aleatorias. La forma de hacerlo es mediante una API a la que se puede enviar una secuencia de números que se hayan generado y obtener una respuesta: *verdadero* si parecen realmente aleatorios, *falso* en caso contrario.

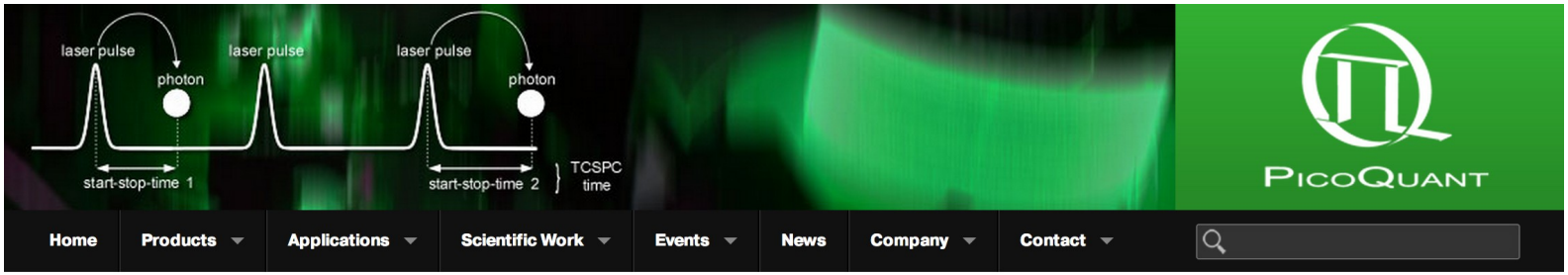
El objetivo es sencillamente **«evitar fallos catastróficos»** de algunas funciones de generación de números de este tipo, que por diversas circunstancias puedan fallar o estropearse (generando cosas secuencias como *ooooo*...). tiene una alta precisión y solo da un falso positivo cada $2^{60}$ casos, suficiente para casi cualquier aplicación.

Además de otros detalles respecto a la API en página del proyecto se puede descargar el código en Github con ejemplos.

Ver también **https://www.random.org/**

# Secuencias de números aleatorios (2)

Curiosamente, los **números aleatorios puros no son adecuados para su uso en técnicas de Monte Carlo**, puesto que **su secuencia no es repetible y los generadores son lentos** en la producción del resultado, necesitando almacenar en el ordenador las secuencias producidas por herramientas externas **[Bielajew93, p. 25]**. Además, en ciertos casos, su distribución, aunque aleatoria, puede no ser simétrica o homogéneamente distribuida **[Rummukainen, p. 20]**.




Home > Products > Photon Counting and Timing > Quantum Random Number Generator

**Products**

- **Pulsed Lasers and LEDs**
- **Photon Counting and Timing**
  - TCSPC and Time Tagging Electronics
  - Photon Counting Detectors
  - Software
  - Accessories
  - Quantum Random Number Generator
- **Fluorescence Spectrometers**
- **Fluorescence Microscopes**

## Quantum Random Number Generator

### PQRNG 150

**Quantum Random Number Generator**

- Quantum Random Number Generator based on photon arrival times
- Bit rate: 150 Mbits/s
- Interface: USB 2.0
- Software: driver and user library (DLL) for Windows XP, Vista and 7
- DLL demos and simple GUI for data retrieval to files

Description | Specifications | Images | Documents | Publications

Randomness is an invaluable resource in many areas of science and technology ranging from Monte Carlo simulations to secure encryption methods. While computer generated random numbers can be used for some applications, they remain fundamentally non-random in the sense that anything generated by an algorithm is at least in principle predictable. However, quantum physics provides randomness which is unpredictable based on the fundamental laws of nature.

# Números aleatorios y pseudoaleatorios

**Secuencias de números pseudoaleatorios:**

• Son los generados mediantes **algoritmos deterministas** que producen secuencias de números **aparentemente aleatorios pero que se repiten con una periodicidad determinada**.

• Estas secuencias no son realmente aleatorias, sino que **vienen determinadas por un subconjunto pequeño de los datos iniciales o una semilla aleatoria**, lo que implica que **no son totalmente independientes entre si**, característica esencial de las secuencias aleatorias.

• Se utilizan por la **rapidez de su generación** mediante los algoritmos deterministas y por la **reproductibilidad de la secuencia** de números con propiedades parecidas a los aleatorios una vez conocida la semilla.

• La **periodicidad** en los generadores normalmente utilizados va desde $2^{32}$ (esto es, 4294967296) a $2^{64}$ ($1.84467441 \times 10^{19}$) antes de una repetición de la secuencia, aunque se pueden encontrar en el mercado generadores de periodicidades mayores.

• Idealmente, el periodo y las propiedades de homogeneidad y aleatoridad no deberían depender de la semilla utilizada. ¡**En la realidad sí dependen de la semilla** y no se deben variar los parámetros sin un buen conocimiento de lo que se está haciendo!

# Generadores de números pseudoaleatorios

• Veremos varios generadores de números pseudoaleatorios de uso común:

- **Generador lineal congruente** (linear congruential generator (LCG ó LCRNG)).
- **Método de los números medios de los cuadrados** (midsquare method).
- **Generador de Fibonacci retardado** (lagged Fibonacci generator).
- **Mersenne twister**.

• En ocasiones se pueden combinar/alternar generadores para evitar los problemas asociados con cada uno de ellos por separado.

• En general merece la pena pasar un tiempo en el estudio de las propiedades de las secuencias generadas por los distintos generadores de números aleatorios antes de utilizarlos para aplicaciones científicas, por ejemplo su uso en simulaciones Monte Carlo.

• Las secuencias correctas de aleatorios deben pasar una serie de test estadísticos, alguno de los cuales están recogidos en los tests “Diehard”, “Dieharder”, test NIST (recogidos en https://csrc.nist.gov/projects/random-bit-generation) ... Veremos ejemplos en los ejercicios.

# Pseudoaleatorios: generador lineal congruente

**Generador lineal congruente (linear congruential generator (LCG ó LCRNG))**

• **Algoritmos que generan números pseudoaleatorios a partir de la expresión**:

$$X_{i+1} = \mathrm{Mod}_m(a \cdot X_i + c)$$

generando enteros de 0 a (m–1) si c ≠ 0, o reales en el rango [0,1) tras dividir estos por m, el módulo de la operación. Cuando c = 0 se denominan generadores multiplicativos congruentes.

• El **valor inicial $X_0$ se denomina semilla** de la generación de pseudoaleatorios.

• El **periodo del generador está limitado por el valor de m** (que puede tomar como máximo el rango del tipo básico: $2^{32} = 2x10^9$ , $2^{64} = 1.84x10^{19}$, ...), ya que cada valor de $X_i$ es (por definición) menor que m y la secuencia de resultados se repite a partir de un estado dado.

• Los valores de a y c se deben escoger cuidadosamente para que el periodo de la secuencia no quede reducido muy por debajo del módulo.

• En general es una mala opción jugar con los parámetros, pues una opción desafortunada puede reducir tremendamente el periodo del generador. Además, si se precisan enteros o bits aleatorios, no se debe truncar el resultado de los LCG, puesto que en muchas ocasiones repiten secuencias cíclicas por la forma en la que se realiza la operación de módulo.

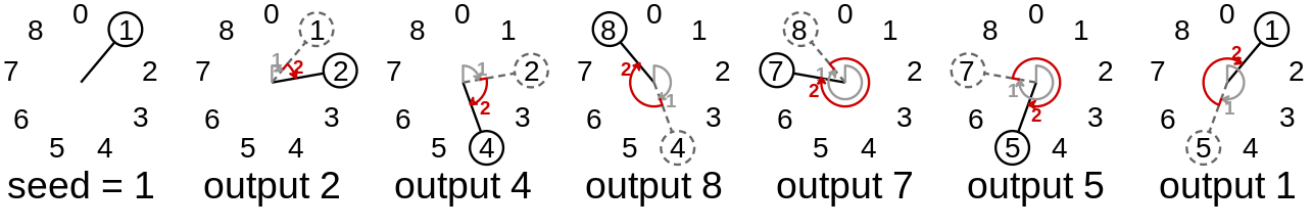
# Pseudoaleatorios: generador lineal congruente

**Ejemplos simples con el LCG:**

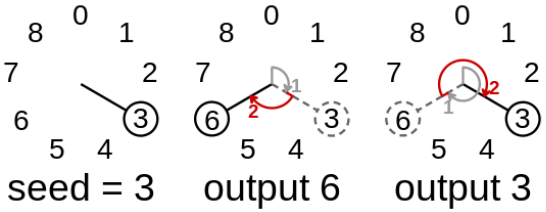
Tomando m = 9 (y por tanto un período máximo de 9), vemos como distintas semillas o elección de los parámetros a y c conducen a diferentes secuencias:

$$X_{i+1} = \mathrm{Mod}_m(a \cdot X_i + c)$$

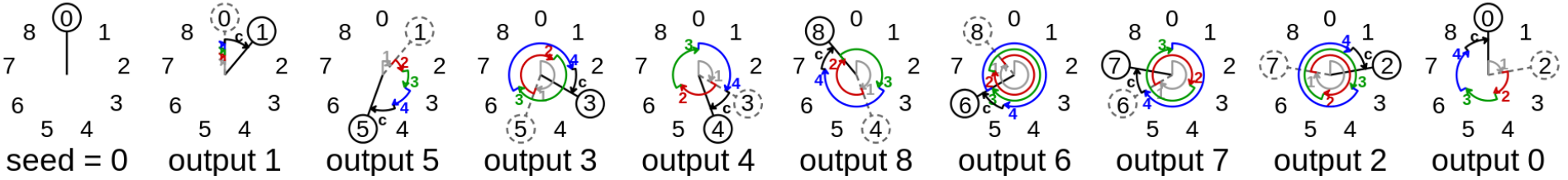
*m* = 9
*a* = 2
*c* = 0




*m* = 9
*a* = 2
*c* = 0




*m* = 9
*a* = 4
*c* = 1




https://en.wikipedia.org/wiki/Linear_congruential_generator

# Pseudoaleatorios: generador lineal congruente (2)

**Ejemplos de generador lineal congruente (linear congruential generator (LCG ó LDRNG))**

• El antiguo generador de aleatorios de **ANSI C, rand()**, es un LCG, con parámetros a=1103515245, c=12345 y $m = 2^{31} \equiv 2 \cdot 10^{9}$

• El **generador de MATLAB** utiliza los parámetros a=16807, c=0, $m=2^{31} - 1$.

• En **UNIX/Linux** se utilizan variaciones de LCG (drand48, erand48, jrand48, lcong48, lrand48, mrand48, nrand48, seed48, srand48, ver comando **man**) generalmente con parámetros a=0x5deece66d (25214903917), c=0xb (11) y $m=2^{48} \equiv 2.8 \times 10^{14}$

• Otros autores **[Bielajew93]** recomiendan a = 663608941 ó a = 69069 como buenas opciones en 32 bits y a=6364136223846793005 para 64 bits.

• Si c=0, el método se conoce como generador congruente multiplicativo, de menor periodicidad, pero más rápido en la generación de las secuencias pseudoaleatorias.

# Ejercicios: generador lineal congruente

• **[GEN1]** Realizar un código básico de un generador lineal congruente que acepte como parámetros los valores a, c, m y la semilla. Modifique los valores para obtener distintos comportamientos.

• **[GEN2]** Representar los valores en un histograma y su homogeneidad en el espacio.

• **[GEN3]** Comprobar que la distribución del espaciado (distancia)$^2$ entre los valores generados es (aproximadamente) exponencial (ver **https://en.wikipedia.org/wiki/Diehard_tests**). Este test también se puede realizar colocando un número de puntos en una malla n-dimensional y calculando la n-distancia entre los pares. El cuadrado de la n-distancia debe distribuirse de forma exponencial.

• **[GEN4]** Comprobar secuencias de cuatro numeros consecutivos. Estos se pueden ordenar de solo 4!=24 formas, en función de quien es mayor. La distribución de probabilidad para cada una de las 24 diversas posibilidades tiene que ser homogenea. Compruébese. Hagase también con 5 numeros consecutivos (5!=120).

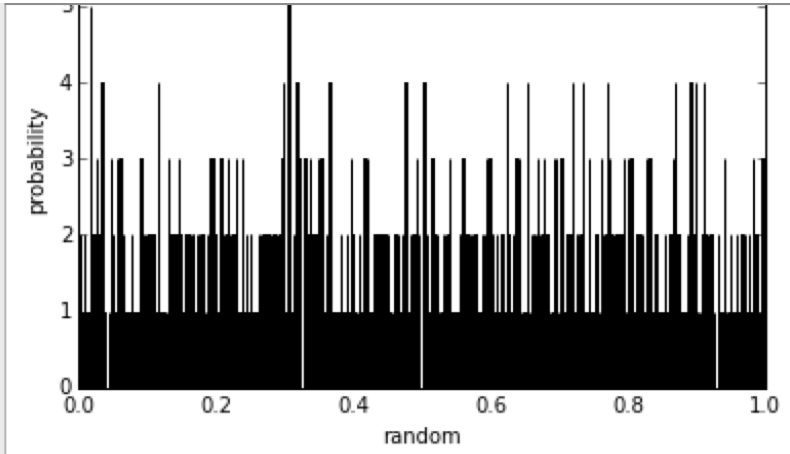
• **[GEN5]** Súmense cada 100 números generados de forma consecutiva. La distribución de la suma debe distribuirse de forma normal. Compruébese y caracterice la distribución resultante.

• **[GEN6]** (Avanzado) Construya una representación de planos de Marsaglia colocando los aleatorios consecutivamente generados en las posiciones x,y,z de puntos en el espacio. Represente esos puntos y compruebe si generan planos definidos en el espacio 3D. (Para más información, leer G. Marsaglia. Random numbers fall mainly in the planes. Nat. Acad. Sci., 61:25 – 28, 1968.)

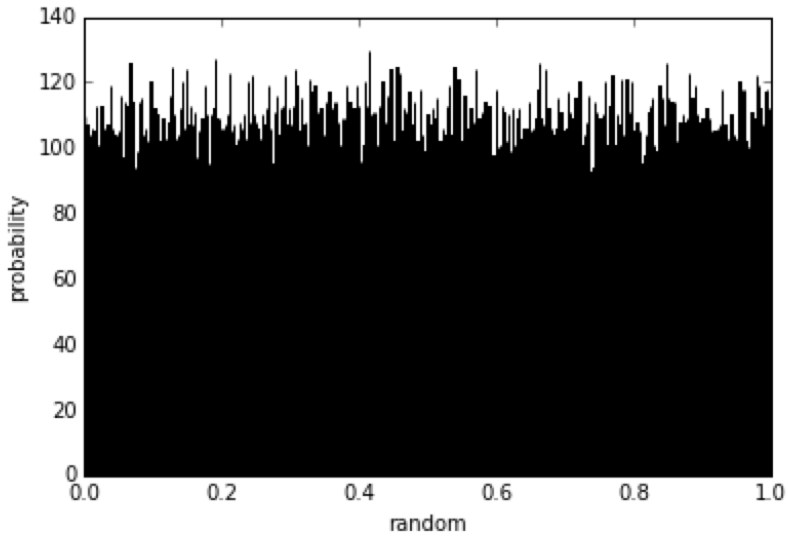
# Ejercicios: generador lineal congruente (2)

[GEN1]

```
# -*- coding: utf-8 -*-
"""
Hector Alvarez Pol (hector.alvarez@usc.es)
Last update: 07 Octubre 2014
A linear Congruential Generator
 Xi+1 = Modm(a · Xi + c)
"""
import array
import matplotlib.pyplot as plt

print "Linear Conguential Generator test system"
user=0
if user:
    print "Enter a modulo (m):",
    m=int(raw_input())
    print "Enter a multiplier (a):",
    a=int(raw_input())
    print "Enter a constant (c):",
    c=int(raw_input())
    print "Enter a seed:",
    x=float(raw_input())
    print "Enter the number of needed randoms:",
    size=int(raw_input())
else:
    m=2147483647
    a=16807
    c=0
    x=10
    size=100000

xArray = array.array('f',[])#storing results in array just for plotting
for i in range(0,size):
    x = (a*x+c) % m
    xArray.append(x/float(m))

plt.hist(xArray, bins=1000)
plt.xlabel('random')
plt.ylabel('probability')
plt.show()
```




```
In [20]:
runfile('/Users/hapol/DOCENCIA/Master/NuevoMaster_2014/Python/
LinearCongruentialGenerator_1.py',
wdir='/Users/hapol/DOCENCIA/Master/NuevoMaster_2014/Python')
Linear Conguential Generator test system
```




```
In [21]:
```

# Ejercicios: generador lineal congruente (3)

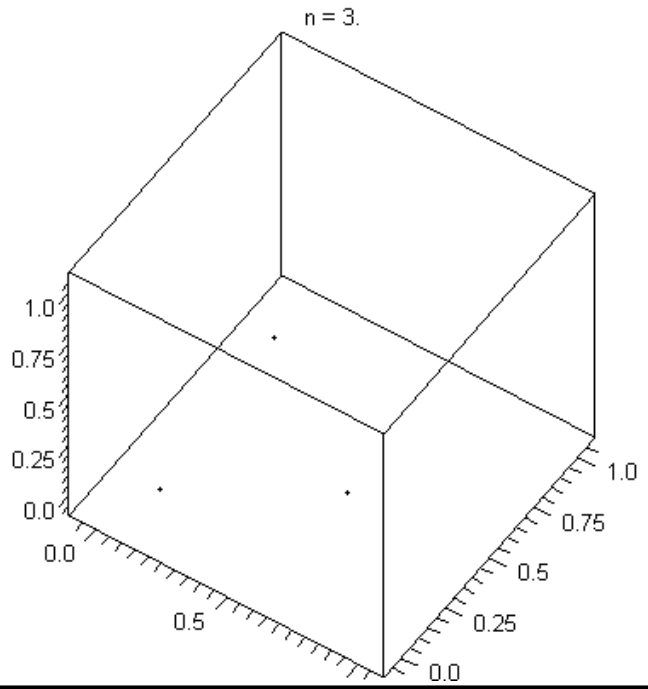
**[GEN1]**

```
    m=2147483647
    a=16807
    c=0
    x=10
    size=100000

xArray = array.array('f',[])#storing results in array just for plotting
for i in range(0,size):
    x = (a*x+c) % m
    xArray.append(x/float(m))

plt.hist(xArray, bins=1000)
plt.xlabel('random')
plt.ylabel('probability')
plt.show()
```

# Ejercicios: generador lineal congruente (3)

The purpose of this note is to point out that all multiplicative congruential random number generators have a defect—a defect that makes them unsuitable for many Monte Carlo problems and that cannot be removed by adjusting the starting value, multiplier, or modulus. The problem lies in the "crystalline" nature of multiplicative generators—if $n$-tuples $(u_1,u_2,\ldots,u_n)$, $(u_2,u_3,\ldots,u_{n+1}),\ldots$ of uniform variates produced by the generator are viewed as points in the unit cube of $n$ dimensions, then *all* the points will be found to lie in a relatively small number of parallel hyperplanes. Furthermore, there are many systems of parallel hyperplanes which contain all of the points; the points are about as randomly spaced in the unit $n$-cube as the atoms in a perfect crystal at absolute zero.

[Marsaglia83]



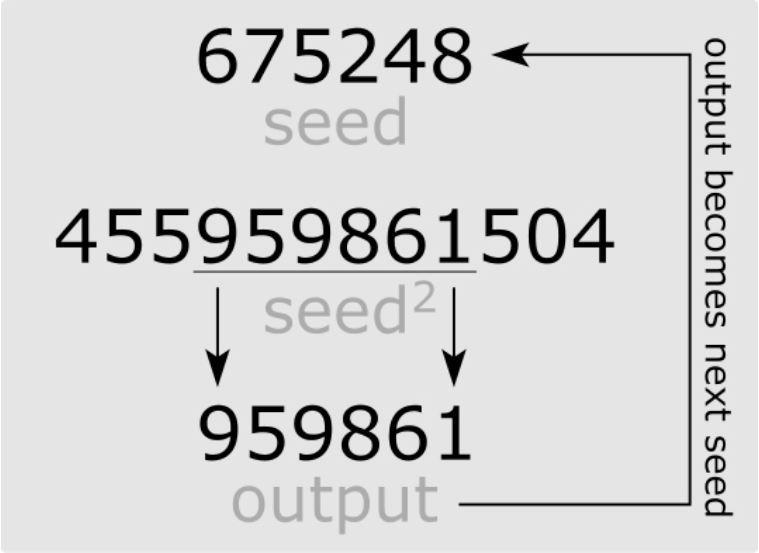
# Pseudoaleatorios: cifras medias de los cuadrados

**Método de las cifras intermedias de los cuadrados (midsquare method):**

• Tiene cierta importancia histórica, al ser el **primer generador de pseudoaleatorios para uso computacional**. Se describe por primera vez en el siglo XIII y se reutiliza durante el proyecto Manhattan (véase **[TheBeginning]** sobre su uso en el proyecto Manhattan).

• Se basa en hacer el **cuadrado de un número entero de n dígitos, y tomar los n dígitos centrales del resultado como nuevo valor aleatorio**.

• En la práctica no es un buen método, puesto que su periodo es muy corto y su secuencia puede converger a cero o quedarse bloqueada en ciertos valores.



# Pseudoaleatorios: Fibonacci retardado

**Generador de Fibonacci retardado (lagged Fibonacci generator):**

• Utiliza una **operación entre dos enteros tomados de la sucesión**, cada uno de ellos obtenido en un punto previo fijo de la sucesión:

$$S_n = S_{n-j} \circ S_{n-k} \qquad 0 < j < k$$

donde el círculo puede corresponder a cualquier operador aritmético valido entre los valores (suma, resta, multiplicación, un operador de bits, ...)

• Su nombre viene de una **generalización de la sucesión de Fibonacci**.

• El generador debe tener al menos k valores en memoria. El estándar random() de UNIX utiliza el operador XOR de bits (en este caso se denominan **GFSR, generalized feedback shift registers**), con un periodo de $16 \cdot (2^{31} - 1) = 3.43597 \times 10^{10}$.

• Como requieren un gran número de aleatorios en memoria para comenzar su funcionamiento, normalmente se rellenan inicialmente con otros generadores rápidos, como LCG.

• Depende mucho del tipo de operación realizada, de la inicialización, … Puede dar problemas en muchos casos, por lo que su uso es limitado y restringido a operaciones y valores iniciales seguros.

| *A* | *B* | *A* XOR *B* |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

# Pseudoaleatorios: Mersenne twister

**Generador Mersenne twister:**

• Se trata de un generador relativamente reciente, desarrollado en 1997 por M. Matsumoto y T. Nishimura (Universidad de Keio, Yokohama, Japón).

• Un número de Mersenne es el obtenido mediante la operación: $M_n = 2^n - 1$. Un número primo de Mersenne es un número de Mersenne que además es primo. Normalmente, **los números primos de Mersenne son los primos mas grandes que se conocen**.

• La **longitud del periodo del generador es un número primo de Mersenne**. El generador es una generalización del GFSR que usa 624 enteros de 32 bits en memoria. El más común, llamado **MT19937, está basado en el primo de Mersenne $2^{19937} - 1$** y tiene este número por periodo.

• Para más detalles sobre el algoritmo, ver **http://en.wikipedia.org/wiki/Mersenne_twister**

• **Python**: Almost all module functions depend on the basic function random(), which generates a random float uniformly in the semi-open range [0.0, 1.0). Python uses the Mersenne Twister as the core generator. It produces 53-bit precision floats and has a period of 2**19937-1. The underlying implementation in C is both fast and threadsafe. The Mersenne Twister is one of the most extensively tested random number generators in existence. However, being completely deterministic, it is not suitable for all purposes, and is **completely unsuitable for cryptographic purposes**.

**https://docs.python.org/2/library/random.html**

# Pseudoaleatorios: Mersenne twister

http://www.microsiervos.com/archivo/ordenadores/primo-mersenne-49-skylake-intel.html

microsiervos

PORTADA BUSCAR CONTACTAR HUMOR ¡SALTA!

ORDENADORES

## Descubierto el 49º primo de Mersenne (y un bug en un procesador Skylake de Intel)

POR @ALVY — 19 DE ENERO DE 2016

Los ordenadores del Proyecto GIMPS *(Great Internet Mersenne Prime Search)* han descubierto el mayor número primo conocido hasta la fecha, un primo de Mersenne de la forma $M = 2^n - 1$ en el que $n$ también es primo. El número en cuestión es el 49º de este tipo que se conoce; lo llaman cariñosamente M74207281 y es el

$$2^{74207281} - 1$$

Es decir 2 elevado a 74207281 menos 1. En total tiene 22 338 618 dígitos y ha batido el récord anterior por unos 5 millones de dígitos.

Para calcularlo y comprobar su primalidad se necesitaron **31 días de cálculos de un PC con una CPU Intel I7-4790,** además de otros tres o cuatro días para comprobarlo con una GPU NVidia Titan Black, una AMD Fury X y la EC2 de Amazon con procesadores Intel Xeon de 18 cores. Según parece estas comprobaciones no fueron baladí; precisamente buscando estos número primos de Mersenne se descubrió recientemente un bug en los procesadores Skylake de Intel gracias al proyecto GIMPS (también por la gente de hardwareluxx.de. Una demostración palmaria de que a veces calcular números de este tipo, dígitos de pi o alguna de esas *computaciones raras* sirve para algo práctico.

(Vía New Scientist.)

http://primes.utm.edu/largest.html

http://primes.utm.edu/mersenne/index.html

# Pseudoaleatorios: Mersenne twister

https://www.microsiervos.com/archivo/ciencia/primo-mersenne-51.html

Es decir 2 elevado a 82589933 menos 1. En total tiene **24.862.048 dígitos** [aquí en un archivo zip comprimidos a 11 MB] y ha batido el récord anterior por un millón y medio de dígitos.

Para calcularlo y comprobar su primalidad se necesitaron **12 días de cálculos de un PC con una CPU Intel i5-4590t,** además de otra semana para comprobarlo con otros procesadores «por si acaso», incluyendo una GPU NVidia, 16 cores de Amazon AWS y un Intel 7700K.

El miembro del proyecto GIMPS en cuyo ordenador se descubrió este número con el software Prime95 fue **Patrick Laroche** (Estados Unidos) y el Papá Noel de las matemáticas le deja 3.000 dólares por su contribución.

(Vía La ciencia de la Mula Francis.)

microsiervos

PORTADA BUSCAR CONTACTAR HUMOR ¡SALTA! TIENDA

CIENCIA

## Descubierto el 51º primo de Mersenne

POR @ALVY — 26 DE DICIEMBRE DE 2018

El Papá Noel de las matemáticas se ha adelantado este año unos días cuando el Proyecto GIMPS *(Great Internet Mersenne Prime Search)* ha anunciado el descubrimiento el 21 de diciembre del **mayor número primo conocido hasta la fecha**, un primo de Mersenne de la forma $M = 2^n - 1$ en el que *n* también es primo. El número en cuestión es el 51º de este tipo que se conoce:

$$2^{82589933} - 1$$

http://primes.utm.edu/largest.html

http://primes.utm.edu/mersenne/index.html

http://www.mersenne.org

# Ejercicios: generación de númeos de aleatorios

• **[GEN7]** Realizar un código de un generador de números pseudoaleatorios mediante selección de dígitos centrales de los cuadrados (¿Problemas? Puedes copiar una posible implementación en python de: **http://en.wikipedia.org/wiki/Middle-square_method**, pero ¡intentalo primero sin mirar!).

• **[GEN8]** Realícese un gráfico de evolución de los valores generados mediante el generador del ejercicio anterior. Comprobar si, para ciertos números iniciales, la serie tiende a cero o se queda estancada alrededor de un valor determinado.

• **[GEN9]** Compruebe, mediante alguno de los test realizados para los generadores LCG, que se obtiene en el caso del generador de números aleatorios integrado en Python.

• **[GEN10]** (Avanzado) Estudiar e implementar el algoritmo MT19937. Se puede partir del psuedocódigo que se ofrece en **http://en.wikipedia.org/wiki/Mersenne_twister** o del código en C en **http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html**

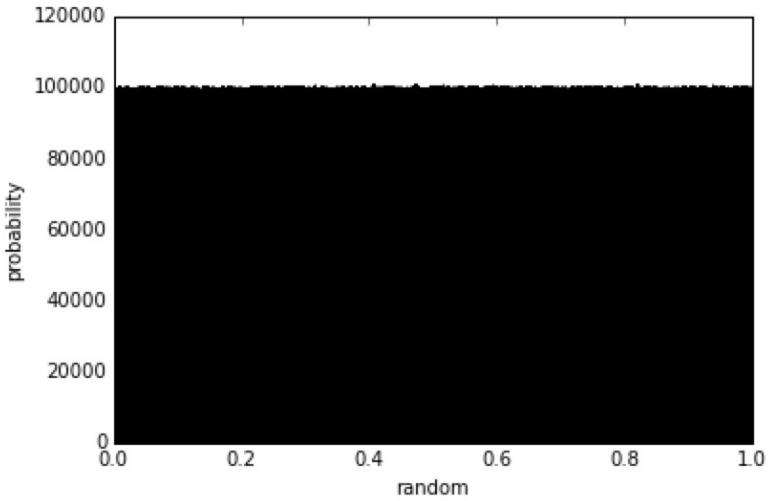
• **[GEN11]** (Avanzado) Para un generador simple de periodo corto, compruebese la tasa de repetición de aleatorios en la secuencia generada. La tasa de repetición puede aproximarse por una exponencial. Compruébese y estudiense los valores de los parámetros obtenidos. (Ver la explicación de la aproximación a exponencial en casos de calculo de probabilidades de repetición de sucesos al azar en la entrada **https://en.wikipedia.org/wiki/Birthday_problem** ).

# Ejercicios: generador lineal congruente (4)

[GEN11]

```python
# -*- coding: utf-8 -*-
"""
Hector Alvarez Pol (hector.alvarez@usc.es)
Last update: 07 Octubre 2014
A linear Congruential Generator
 Xi+1 = Modm(a · Xi + c)
"""
import matplotlib.pyplot as plt

#m=2147483647
m=214748
a=16807
c=0
x=10
size=100000000

ranList=[]
ranIntList=[]
for i in range(0,size):
    x = (a*x+c) % m
    ranList.append(x/float(m))
    ranIntList.append(x)

dup=0
dupInt=0
ranList.sort()
ranIntList.sort()
for j in range(1,size):
    if ranList[j-1] == ranList[j]:
        dup=dup+1;
    if ranIntList[j-1] == ranIntList[j]:
        dupInt=dupInt+1;

print dup
print dupInt

plt.hist(ranList, bins=1000)
plt.xlabel('random')
plt.ylabel('probability')
plt.show()
```

```
In [4]:
runfile('/Users/hapol/DOCENCIA/Master/NuevoMaster_2014/Python/LCG_BirthdayParadox_1.py',
wdir='/Users/hapol/DOCENCIA/Master/NuevoMaster_2014/Python')
0
0
```




```
In [5]:
runfile('/Users/hapol/DOCENCIA/Master/NuevoMaster_2014/Python/LCG_BirthdayParadox_1.py',
wdir='/Users/hapol/DOCENCIA/Master/NuevoMaster_2014/Python')
```

# Ejercicios: generador lineal congruente (5)

[GEN11]

```
ranList=[]
ranIntList=[]
for i in range(0,size):
    x = (a*x+c) % m
    ranList.append(x/float(m))
    ranIntList.append(x)

dup=0
dupInt=0
ranList.sort()
ranIntList.sort()
for j in range(1,size):
    if ranList[j-1] == ranList[j]:
        dup=dup+1;
    if ranIntList[j-1] == ranIntList[j]:
        dupInt=dupInt+1;

print dup
print dupInt
```

Nota: si la probabilidad de que aparezca un número aleatorio dentro del rango fuera siempre la misma, la probabilidad de repetición sería:

$P = 1 - P(N, n) \approx 1 - \exp(-n^2/2N)$

Donde N es el numero de posibles valores y n el número de muestras que comprobamos.

La última fórmula es una aproximación únicamente válida cuando n y N son muy grandes.

Para el Mersenne twister (random de python) tendremos $N = 2^{53}-1$ y $P \sim 0$ para $n<10^7$