Generazione di numeri pseudocasuali

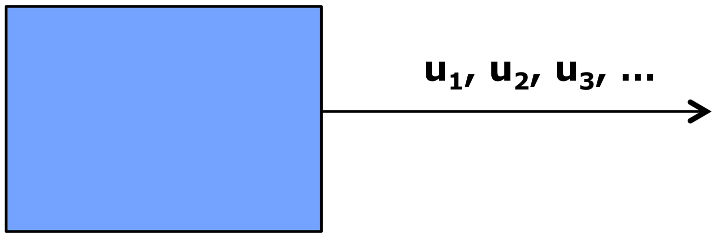
Obiettivo

- Si vogliono generare delle sequenze di numeri che si possano interpretare come realizzazioni di variabili aleatorie aventi una data distribuzione di probabilità.

- I meccanismi di generazione sono deterministici e replicabili, ma un osservatore esterno (che non abbia conoscenza del meccanismo di generazione) deve essere indotto a ritenere che la sequenza di numeri sia effettivamente costituita da realizzazioni di una variabile aleatoria.

Il punto di partenza

- *Numeri casuali (random numbers, RN)*: una successione di numeri



- che devono potersi interpretare come realizzazioni di una sequenza di v.a. U_1, U_2, U_3, … indipedenti e ~U(0,1).
- Il meccanismo è deterministico per garantire la riproducibilità → numeri pseudocasuali

La sequenza u_1, u_2, … deve soddisfare ...

- Partizionare [0,1] in *n* intervalli di uguale ampiezza.
- Proprietà di uniformità: numero di osservazioni in ogni sottointervallo $\rightarrow N/n$ per $N \rightarrow +\infty$ (legge dei grandi numeri) … test di adattamento
- Proprietà di indipendenza: la probabilità di ottenere il k-mo numero casuale in un particolare intervallo è indipendente dai valori precedentemente ottenuti … test di indipendenza

Metodo della congruenza lineare

- I generatori utilizzati oggigiorno sono un'evoluzione del metodo della congruenza lineare (Lehmer, 1951)

$$x_{k+1} = (a\, x_k + c) \bmod m$$

$$u_{k+1} = x_{k+1} / m$$

- La sequenza dipende dalla scelta del *seme*, x_0
- I metodi puramente moltiplicativi (c=0) sono da preferiti (un'operazione in meno), a parità di altre condizioni

Python naive implementation

```
# Naive implementation of the Lehmer LGM
def lehmer (n, a, c, x0, m):
    x = [x0]
    u = [x0/m]
    for i in range(1, n):
        x.append((a * x[i-1] + c) % m)
        u.append(x[i]/m)
    return(u)
```

Esempio

- *Si vuole generare una sequenza di numeri casuali utilizzando il metodo della c.l. con* a=1, c=5 e m=4, x_0 =2,.

$$x_{k+1} = (a \times x_k + c) \bmod m$$

- x_1 = (1x2+5) mod 4 = 3
- x_2 = (1x3+5) mod 4 = 0
- x_3 = (1x0+5) mod 4 = 1

...

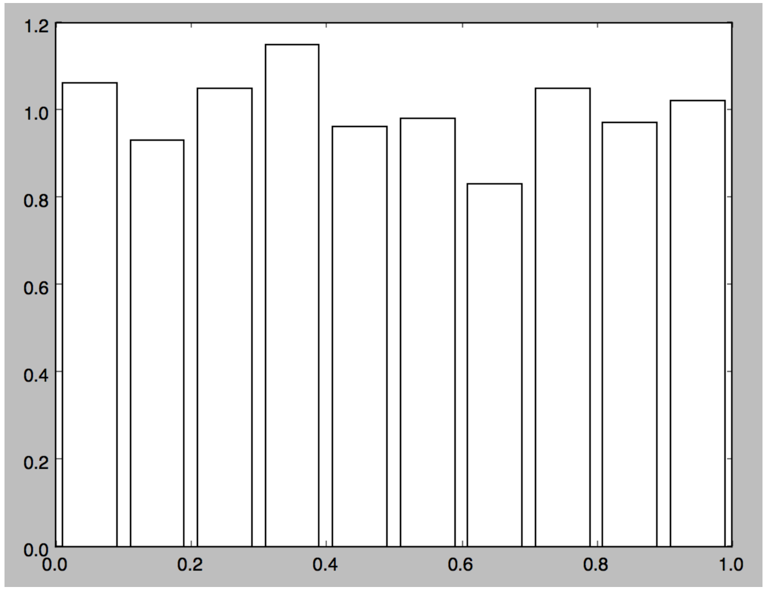
u_1 = 3/4 = 0.75
u_2 = 0/4 = 0
u_3 = 1/4 = 0.25
...

Python code:

```
u = lehmer(10, 1, 5, 2, 4)
print(u)

>>>
[0.5, 0.75, 0.0, 0.25, 0.5, 0.75, 0.0, 0.25, 0.5, 0.75]
>>>
```

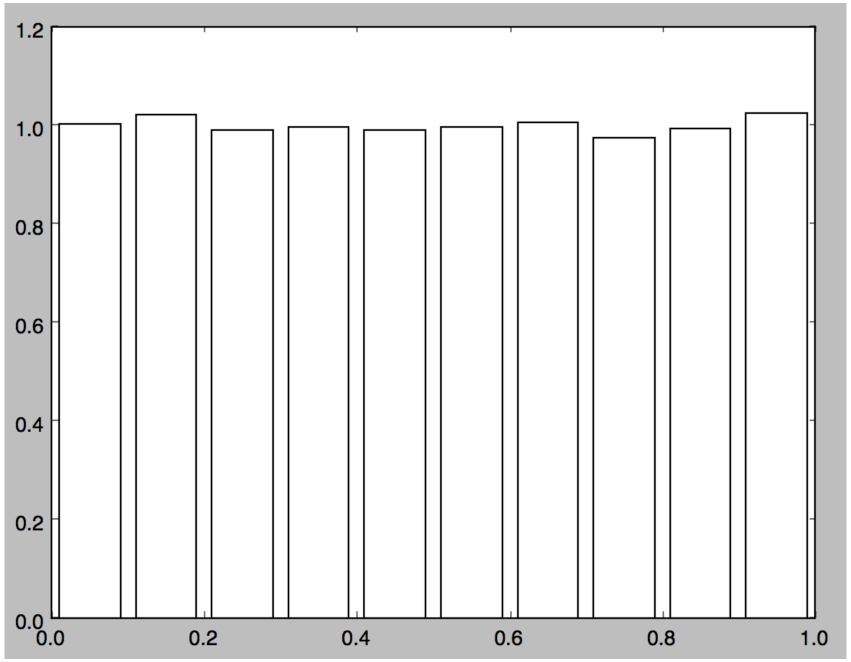
Metodo della congruenza lineare

```
u = lehmer(1000, 372, 551, 79, 44443)

# Creating a histogram with a fixed number (10) of bins
plt.hist(u, bins=10, histtype='bar', rwidth=0.8, color='white',
            label='Lehmer LGM algorithm', normed=True)
plt.show()
```



Metodo della congruenza lineare

```
u = lehmer(10000, 372, 551, 79, 44443)

# Creating a histogram with a fixed number (10) of bins
plt.hist(u, bins=10, histtype='bar', rwidth=0.8, color='white',
         label='Lehmer LGM algorithm', normed=True)
plt.show()
```



a=1, c=5 e m=4, x_0 =2

$$x_{k+1} = (a \times x_k + c) \bmod m$$

- x_1 = (1x2+5) mod 4 = 3 — $u_1 = 3/4 = 0.75$
- x_2 = (1x3+5) mod 4 = 0 — $u_2 = 0/4 = 0$
- x_3 = (1x0+5) mod 4 = 1 — $u_3 = 1/4 = 0.25$

- x_4 = (1x1+5) mod 4 = 2 — $u_4 = 2/4 = 0.5$
- x_5 = (1x2+5) mod 4 = 3 — $u_5 = 3/4 = 0.75$ Nuovamente 0.75 !
- ……

Python code:

```
u = lehmer(10, 1, 5, 2, 4)
print(u)

>>>
[0.5, 0.75, 0.0, 0.25, 0.5, 0.75, 0.0, 0.25, 0.5, 0.75]
>>>
```

Metodo della congruenza lineare

- Il metodo è in grado di "ingannare" un osservatore esterno?
- La sequenza u_k è:

1. ciclica (con periodo ≤ m)
2. le u_k assumono soltanto i valori 0, 1/m, 2/m, … *(m-1)/m*

- Condizione NECESSARIA il RNG sia "buono": il modulo m deve essere sufficientemente grande in rapporto al numero di numeri pseudocasuali richiesti.
- La condizione è SUFFICIENTE?

La condizione è sufficiente?

```
# Determining the cycle length of the Lehmer LGM
def lehmer_cycle (n, a, c, x0, m):
    x = [x0]
    u = [x0/m]
    for i in range(1, n):
        x.append((a * x[i-1] + c) % m)
        u.append(x[i]/m)
        for j in range(0,i-1):
            if x[i] == x[j]:
                return(j,i)
    return(0,0)
```

```
print(lehmer_cycle(50000, 1, 5, 2, 4444))
>>>
(0, 4444)
>>>
```

```
print(lehmer_cycle(100000, 372, 551, 78, 39872))
>>>
(3, 80)
>>>
```

Metodo della congruenza lineare

- L’algoritmo:
  - È Veloce (può essere richiamato molte volte nel corso della simulazione di un sistema complesso);
  - Genera sequenze replicabili

Proprietà di una classe di RNGs

- *m* numero primo
- *c=0*
- *a: il più piccolo numero intero k che rende a^k-1 divisibile per m è k=m-1*
- *→ massimo periodo ottenibile = m-1*

Esempio:

- *a=7 ^{5}=16807*
- *c=0*
- *m=2^{31}-1=2 147 483 647 (numero primo)*

RNG per simulazione di sistemi complessi

- *RNG con periodo 10^9 si rivelano inadatti a molte applicazioni*
- *Generatori più evoluti (disponibili in R):*
  - *Whichmann-Hill, ciclo $\approx 10^{12}$*
  - *Marsaglia-Multicarry, ciclo $\approx 2^{60}$*
  - *Super-Duper (anni '70), ciclo $\approx 10^{18}$*
  - *Knuth-TAOCP (1997, 2002), ciclo $\approx 10^{129}$*
  - *Mersenne-Twister (1998), ciclo $\approx 2^{19937}$*
- *Analisi approfondita: non rientra tra gli scopi del corso*

Python: the "random" library

```
from random import *

print('Random float x, 0.0 <= x < 1.0:', random())
print('Another random float x, 0.0 <= x < 1.0:', random())
print('One more random float x, 0.0 <= x < 1.0:', random())
```

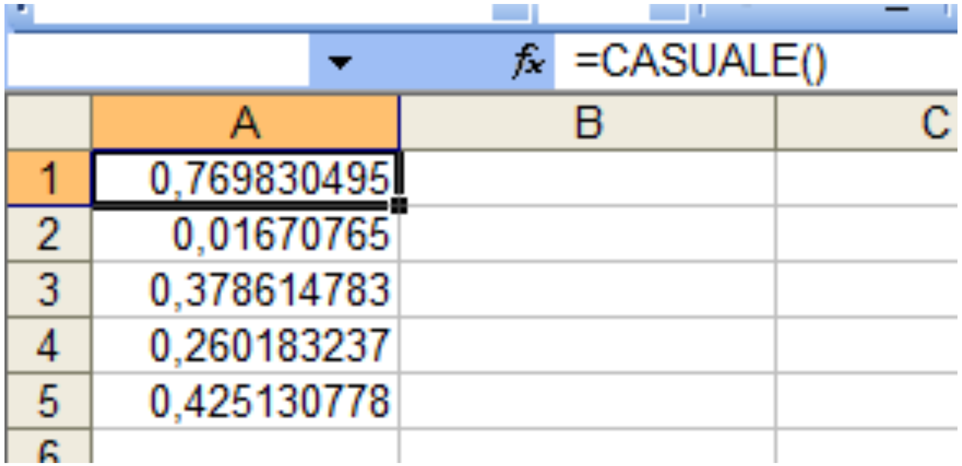
```
>>>
Random float x, 0.0 <= x < 1.0: 0.8399771486163523
Another random float x, 0.0 <= x < 1.0: 0.37538722044896966
One more random float x, 0.0 <= x < 1.0: 0.8731015377246271
>>>
```

Making pseudo-random number generation reproducible

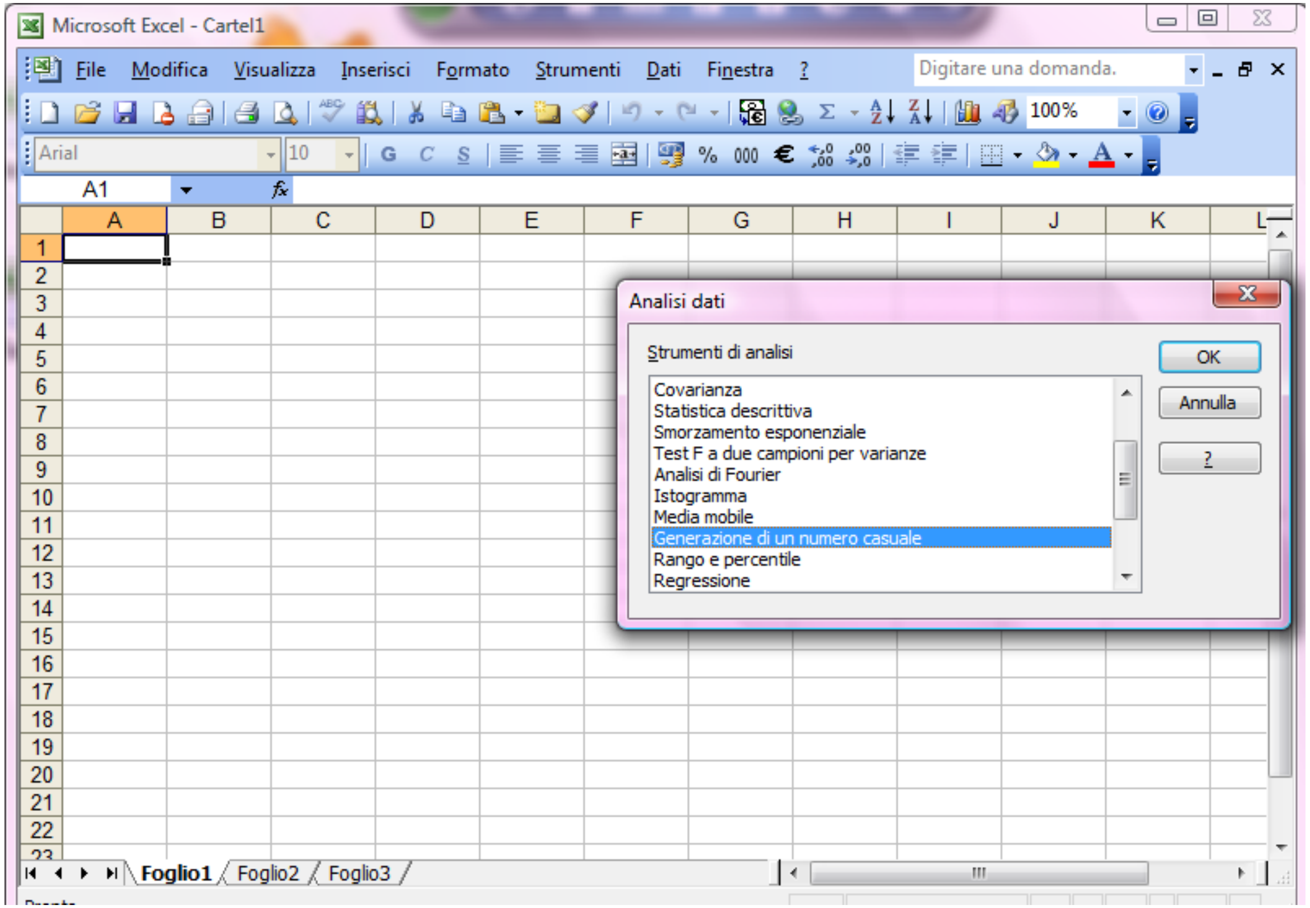
```
from random import *

print('Making a simulation reproducible\n')
state=getstate()
print('The state of the Mersenne Twister generator is', state)
print('Extracting three uniform random numbers: ', random(), random(), random())
print('Restoring the state of the generator')
setstate(state)
print('Extracting the same uniform random numbers as before: ',
      random(), random(), random())
```

```
Making a simulation reproducible

The state of the Mersenne Twister generator is (3, (2147483648, 2230661
500616, 682741392, 3451973918, 140455950, 956702289, 1984135242, 335218
826190, 118153875, 2574371250, 217263865, 2752193447, 129417357, 414349
1812153, 3132995736, 1671536053, 196209824, 1502684879, 3950558077, 272
230977648, 3458500039, 3338374512, 3821344721, 1712173518, 2710395641,

7931, 150117641, 1163255863, 3561668035, 424775999, 624), None)
Extracting three uniform random numbers:  0.138200220994264 0.871036181684126 0.
24792657024731823
Restoring the state of the generator
Extracting the same uniform random numbers as before:  0.138200220994264 0.87103
6181684126 0.24792657024731823
>>> 
```

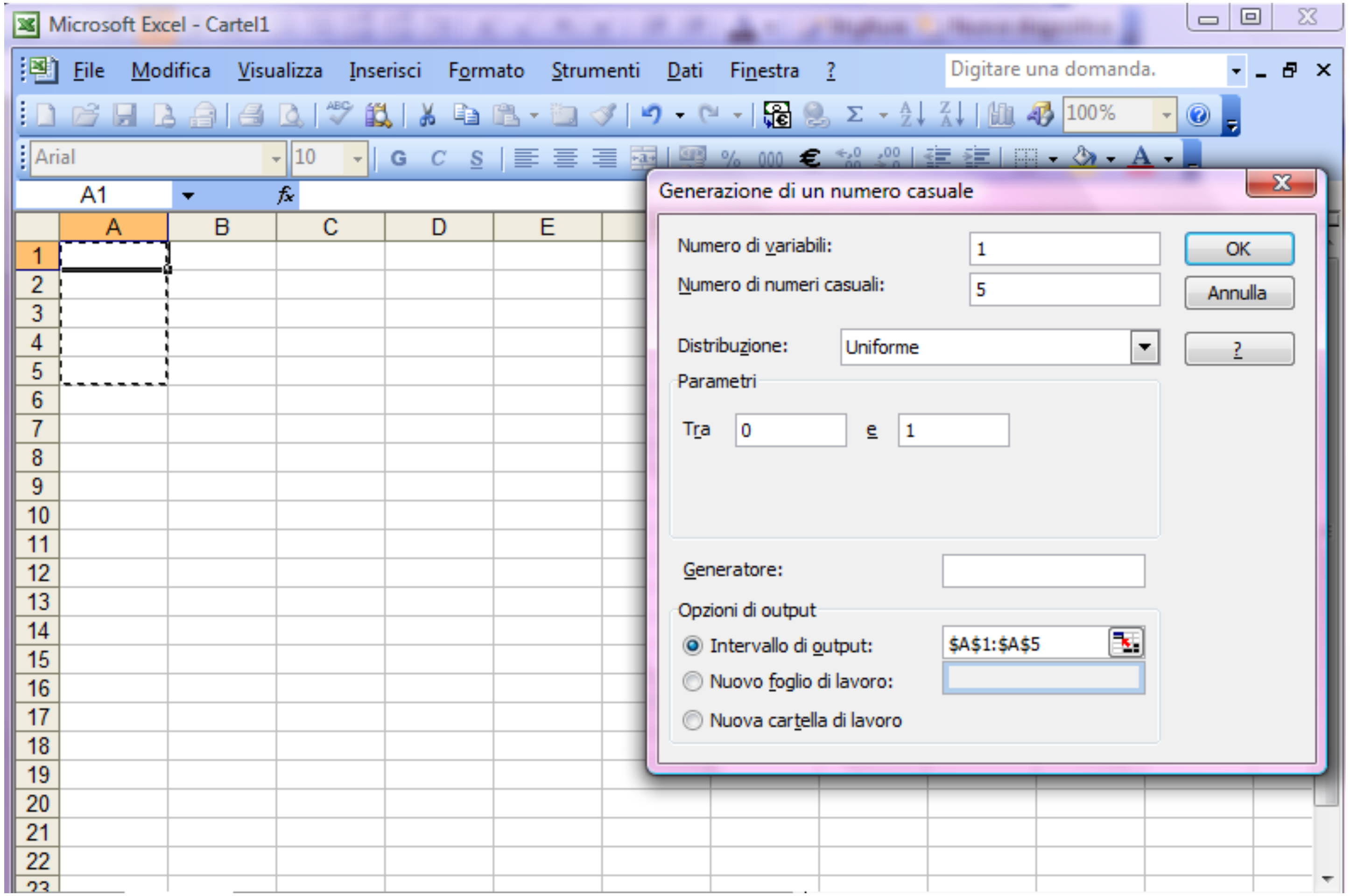
Generatore di Excel



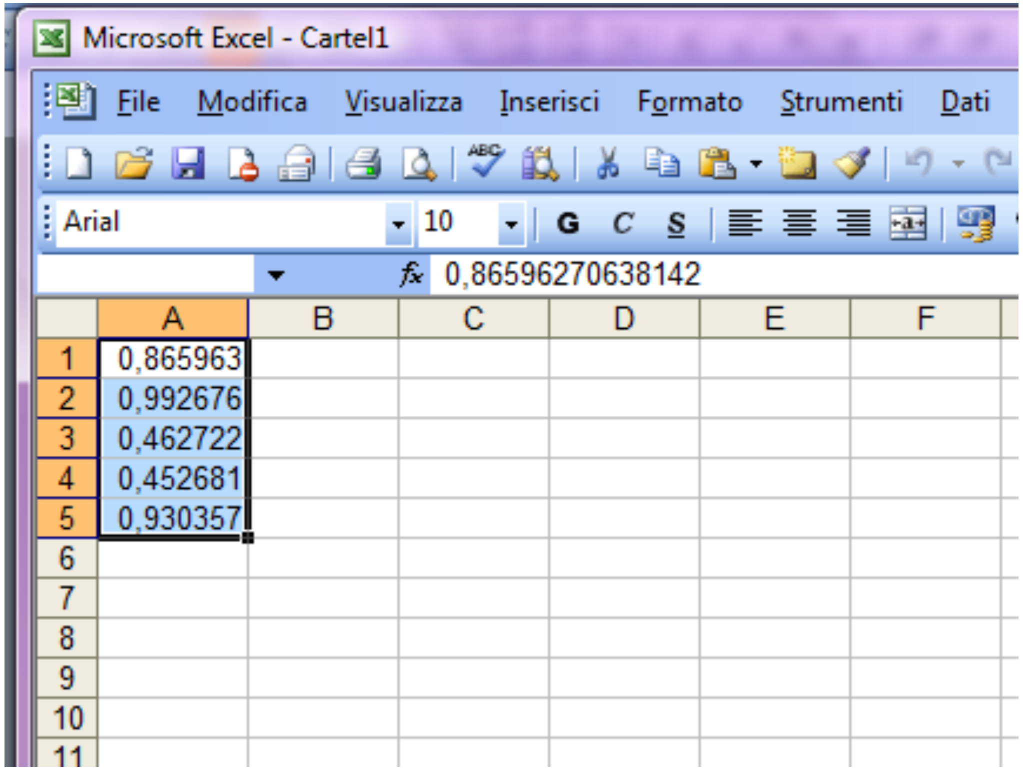
Generatore di Excel



Generatore di Excel



Generatore di Excel



Generazione di variabili aleatorie con distribuzione generica

Generazione di variabili aleatorie con distribuzione generica

- **A partire da una sequenza di numeri random (U(0,1))** opportunamente generati, i metodi per la generazione di variabili aleatorie con distribuzione generica sono:

  1. tecnica di *trasformata inversa*
  2. tecnica di *trasformazione diretta*
  3. metodo di *accettazione/rifiuto*

- E' evidente che la routine di generazione dei numeri casuali per essere veloce deve chiamare poche volte la routine di generazione dei numeri distribuiti in maniera uniforme.

Tecnica di trasformazione inversa

- Sia $F(x)=Pr(X \leq x)$ la CDF desiderata. $F(x)$ assume valori $\in[0,1]$. Supponiamo che sia strettamente crescente (e quindi invertibile).
- Sia $U \sim U(0,1)$. Si osserva che la v.a. definita da $X=F^{-1}(U)$ ha CDF pari proprio a $F(x)$. Infatti:

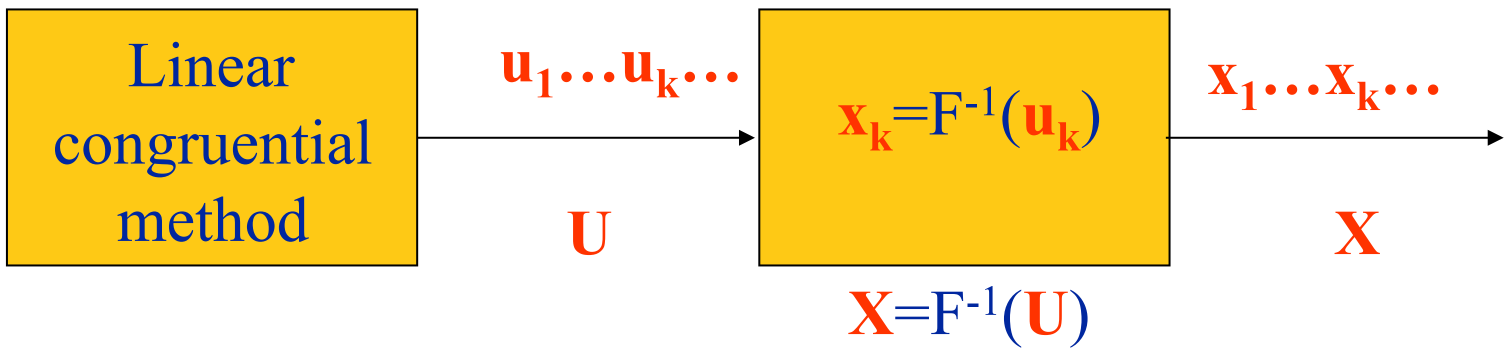
$$P(X \leq x)=$$

$$P(F^{-1}(U) \leq x)=$$

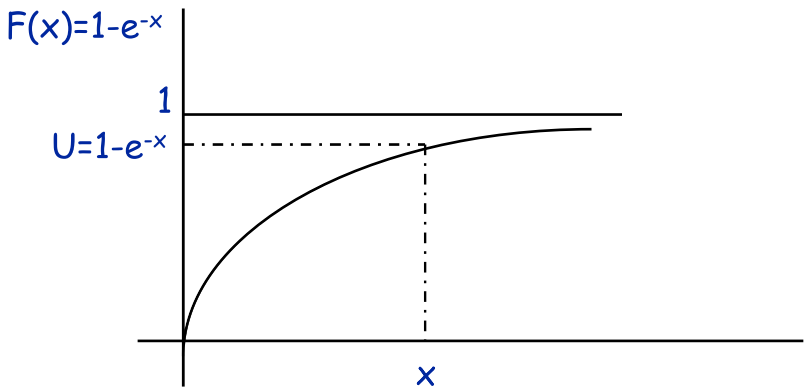
$$P(U \leq F(x))=$$

$$F(x)$$

poichè $P(U \leq u) = u$ per $0 \leq u \leq 1$.



Distribuzione esponenziale con parametro λ=1



Può essere utilizzata per ottenere campioni da molte tipologie di funzioni di distribuzione, come esponenziali, uniformi, triangolari.

Risulta essere il più intuitivo ma non il più efficace dal punto di vista computazionale.

Esempio: generare una variabile aleatoria X con funzione di distribuzione esponenziale con media 1/λ

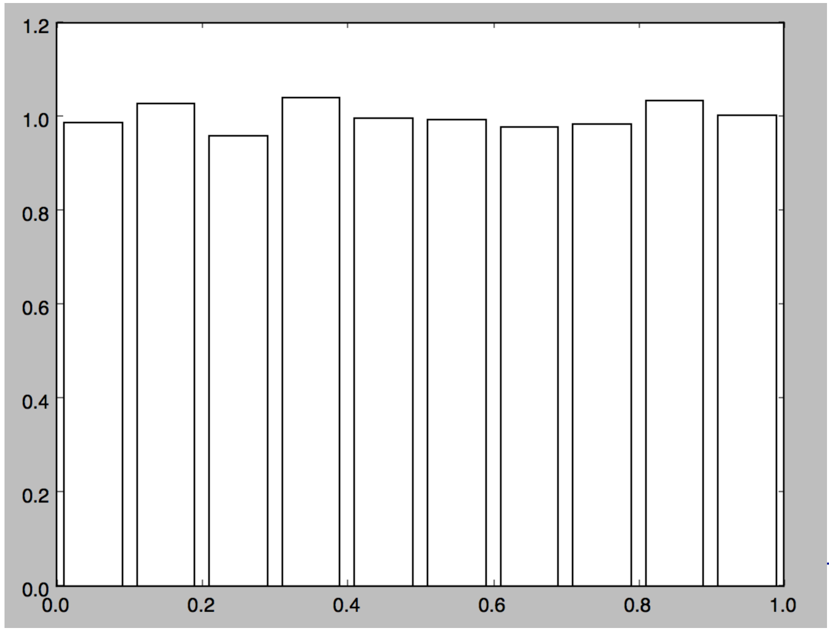
1. $F_x(x) = \int_0^x \lambda e^{-\lambda t} dt = 1 - e^{-\lambda x}$

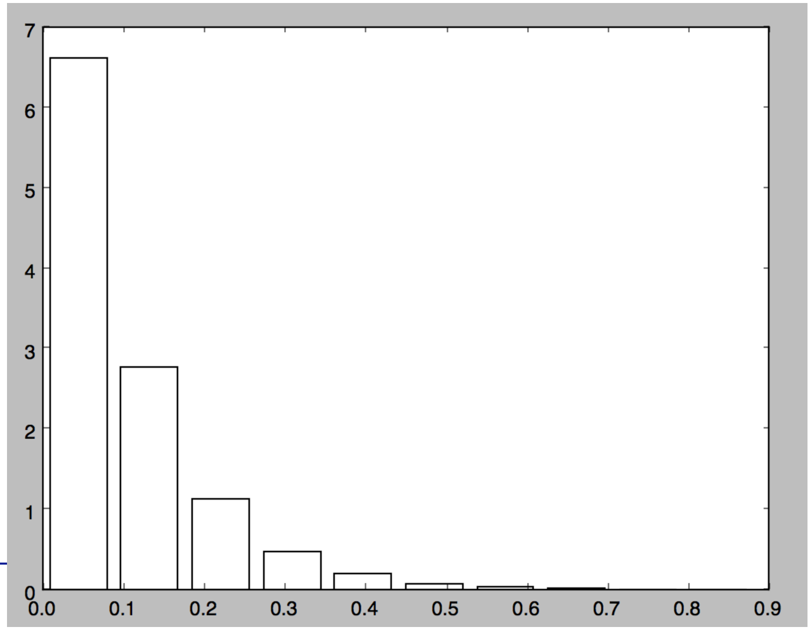
   $e^{-\lambda x} = 1 - u$

   $-\lambda x = \ln(1-u)$

2. u = 1 – e $^{-\lambda x}$, con u ~ U(0,1)
3. X = -ln(1-u)/λ, o, in modo equivalente, $X = -\frac{1}{\lambda}\ln(u)$

```
from random import *
import matplotlib.pyplot as plt
from math import *

n = 10000
u = []
for i in range(0, n-1):
    u.append(random())

# Creating a histogram with a fixed number (20) of bins
plt.hist(u, bins=10, histtype='bar', rwidth=0.8, color='white',label='', normed=True)
plt.show()

lbd = 10
x = []
for i in range(0,n-1):
    x.append(- (1/lbd) * log(u[i]))

# Creating a histogram with a fixed number (20) of bins
plt.hist(x, bins=10, histtype='bar', rwidth=0.8, color='white', label='', normed=True)
plt.show()
```





$\lambda=5$

=-0,2*LN(A6)

	A	B	C
1	u_k	x_k	
2	0,903963962	0,020193157	
3	0,296159982	0,243371098	
4	0,314425879	0,231401382	
5	0,410497932	0,178076878	
6	0,327441585	0,223289121	
7			

$\lambda=5$

=-0,2*LN(A6)

	A	B	C
1	u_k	x_k	
2	0,903963962	0,020193157	
3	0,296159982	0,243371098	
4	0,314425879	0,231401382	
5	0,410497932	0,178076878	
6	0,327441585	0,223289121	
7			

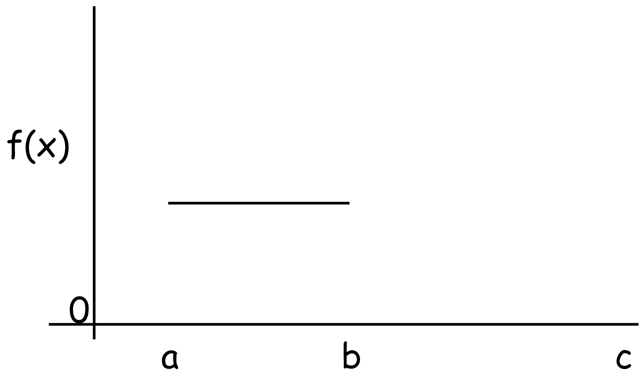
Osservazioni:

La routine di generazione di una variabile aleatoria X con funzione di distribuzione esponenziale:

1. chiama una sola volta il RNG, per ogni istanza di X
2. ha lo stesso ciclo di RNG
3. è replicabile se lo è RNG
4. genererebbe numeri con proprietà statistiche ideali se RNGgenerasse numeri random ideali

Esempio n.2: generare una variabile aleatoria X uniformemente distribuita in (a,b)

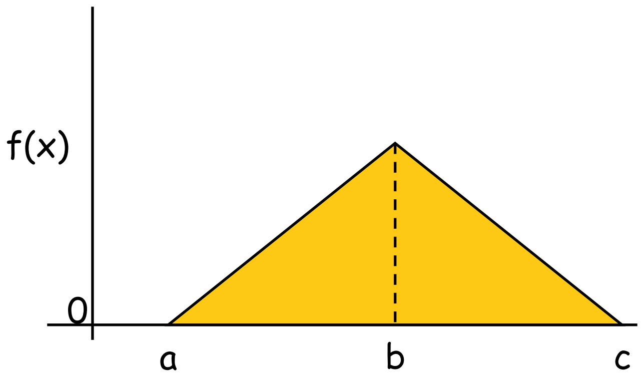
1. $F_x(x) = \int_a^x f(x) = \int_a^x \frac{1}{b-a} dt = \frac{x-a}{b-a}$
2. u = (x-a)/(b-a), con u ~ U(0,1)
3. X = a+u(b-a)



Esempio n.3: generare una variabile aleatoria X con distribuzione triangolare, ossia:

se $a \le x \le b$, allora $f(x)=[2(x-a)]/[(c-a)(b-a)]$;

altrimenti $f(x)=[2(c-x)]/[(c-a)(c-b)]$



1. $se\ \ \mathrm{x} \leq \mathrm{b},\ F_x(x) = \dfrac{(x-a)^2}{(c-a)(b-a)}$

   $\text{altrimenti } F_x(x) = 1 - \dfrac{(c-x)^2}{(c-a)(c-b)}$

2. $se\ \ \mathrm{x} \leq \mathrm{b},\ x = \mathrm{a} + \sqrt{u(c-a)(b-a)}$

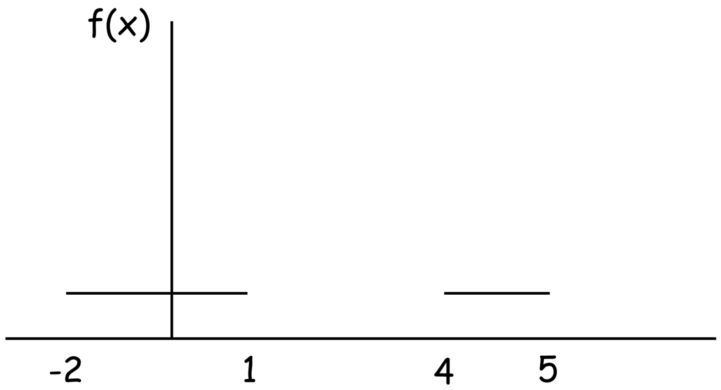
   $\text{altrimenti } x = \mathrm{c} - \sqrt{(1-u)(c-a)(c-b)}$

$$u = \frac{(x-a)^2}{(c-a)(b-a)}$$

$$u(c-a)(b-a) = (x-a)^2$$

$$x = a + \sqrt{u(c-a)(b-a)}$$

Esercizio n.4: generare una variabile aleatoria X con distribuzione uniforme in [-2,1]U[4,5]



Python

```
from random import *

# Uniform distribution
print('Random float x, -1.0 <= x < 7.5:', uniform(-1.0, 7.5))

# Triangular distribution
print('Random number from the triangular distribution TRIANG(1,3,2):',
      triangular(1,3,2))

# Exponential distribution
print('Random number from the exponential distribution with parameter Lambda=5',
      expovariate(5))

# Normal distribution
# We can use normalvariate() instead of gauss()
print('Random number from the normal distribution with mu=0 and sigma=1',
      gauss(0,1))

# More distribution are available. See the random.py documentation
```

Sampling a discrete distribution

Let

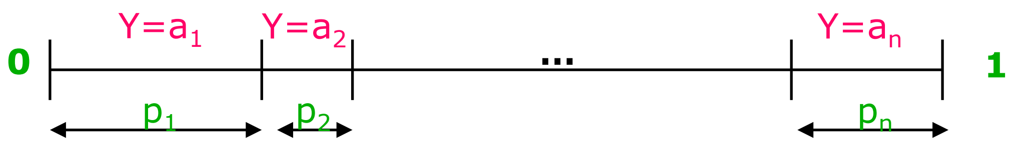
$$\Pr(Y = a_i) = \begin{cases} p_1 \\ p_2 \\ \dots \\ p_n \end{cases}$$

be the required mass probability function (of course $p_1+\ldots+p_n=1$).

Sampling a discrete distribution

By appling the inversion method, we get:

$$Y = \begin{cases} a_1 & if\, 0 \le u \le p_1 \\ a_2 & if\;\; p_1 \le u \le p_1 + p_2 \\ & \qquad\quad \ldots \\ a_n & if\;\; p_1 + \ldots + p_{n-1} \le u \le 1 \end{cases}$$



Python

```
from random import *

print('Uniform random integer from 0 to 7', randrange(8))

print('Single random element of list', ['dog', 'cat', 'crocodile', 'snake'],
      'is:', choice(['dog', 'cat', 'crocodile', 'snake']))

# Probability mass function
p={'dog': 0.1, 'cat': 0.3, 'crocodile': 0.5, 'snake': 0.1}

# The more general method in which probabilities are attached to elements
# has to be programmed
def discrete_random_variable (p):
    x = random()
    cp=0 # cumulative probability
    for value in p.keys():
        if x <= cp+p[value]:
            return(value)
        else:
            cp += p[value]
print('Single random element of list', ['dog', 'cat', 'crocodile', 'snake'],
      'according to distribution', p, 'is:', discrete_random_variable(p))

items=[1, 2, 3, 4, 5]
print('An uniform permutation of', items, 'is:')
shuffle(items)
print(items)
```

Example

Assume Pr(Y=2)=0.3; Pr(Y=5)=0.2; Pr(Y=10)=0.5

B2 =SE(A2<=0.3;2;0)+SE(E(A2>0.3;A2<=0.5);5;0)+SE(A2>0.5;10;0)

	A	B	C	D	E	F	G	H	I
1	u_k	y_k							
2	0.717112	10							
3	0.921037	10							
4	0.443001	5							
5	0.585504	10							
6	0.759126	10							
7	0.881229	10							
8	0.872688	10							
9	0.599395	10							
10	0.491334	5							
11	0.117939	2							
12	0.757856	10							
13	0.457647	5							
14	0.18799	2							
15	0.329903	5							
16	0.883832	10							
17	0.48281	5							
18	0.568911	10							