Trabajo Práctico 1
Simulación 75.26 / 95.19

Facultad de Ingeniería,
Universidad de Buenos Aires

Grupo 4

Generador LXM

Integrantes:

- Gestoso, Ramiro — Padrón 105950
- Brizuela Lopez, Mariano — Padrón 105079
- Markarian, Darío Hernan — Padrón 98684
- Ripetour, Diego — Padrón 86601

Generador de números pseudoaleatorios LXM

Un generador de la familia LXM consiste en cuatro componentes:

1. L: un generador de números pseudoaleatorios congruencial lineal (LCG)
2. X: un generador de números pseudoaleatorios F2-linear. En este caso se usa el XGB “xor-based generator”.
3. Una combinación en dos operandos de 2-bits que producen un resultado de w-bits.
4. M: una función mezcladora biyectiva que mapee un argumento de w-bits a otro de w-bits.

Pseudocódigo del generador

generar():

z ← mezclar con s

s ← actualizar LCG con s

t ← actualizar XBG con t

devolver z

Si el algoritmo LCG tiene un periodo de 2^{64} y el XGB de 2^{128} - 1 (como es descrito en el paper), entonces este algoritmo tendrá un período de: $2^{192} - 2^{64} \approx 6.23*10^{57}$

Código en Python para números de 64 bits

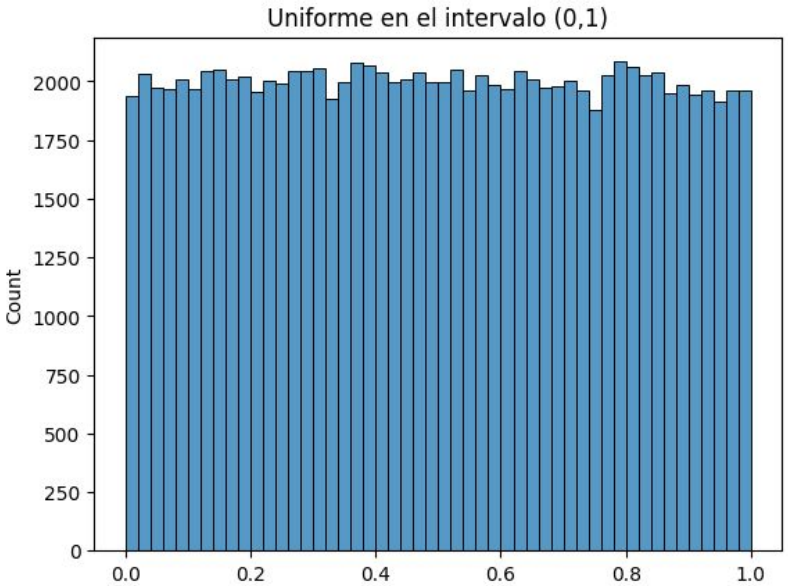
```python
def nextRand(self, n=1):
    if n < 1:
        raise Exception("'n' must be 1 or greater")

    numbers = []
    for i in range(n):
        # Combining operation
        z = self.s + self.x0
        # Mixing function (Lea64)
        z = (z ^ (shiftRight(z, 32))) * self.c
        z = (z ^ (shiftRight(z, 32))) * self.c
        z = (z ^ (shiftRight(z, 32)))
        # Update the LCG subgenerator
        self.s = self.M * self.s + self.a
        # Update the XBG subgenerator (xoroshiro128v1_0)
        q0, q1 = self.x0, self.x1
        q1 ^= q0
        q0 = rotateLeft(q0, 24)
        q1 = q0 ^ q1 ^ (shiftLeft(q1, 24))
        q1 = rotateLeft(q1, 37)
        self.x0, self.x1 = q0, q1
        #result
        numbers.append(z / uint64(-1))

    return numbers[0] if n==1 else np.array(numbers)
```

Donde “nextRand” devuelve ‘n’ números pseudoaleatorios en el intervalo (0,1).

Lo central del algoritmo puede verse en el cuadro rojo.

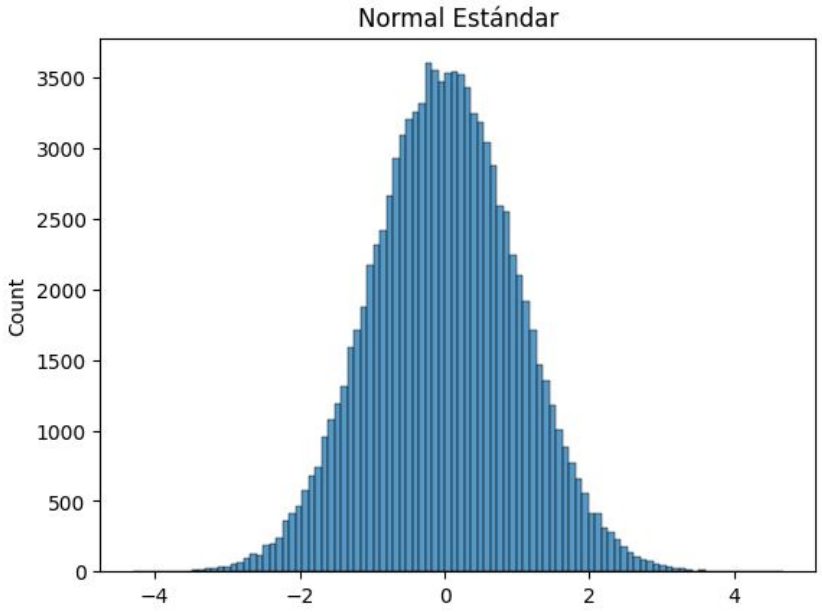
El código presente fue basado en la versión de Java presentada en el paper.

Generamos 100 mil números y observamos la distribución



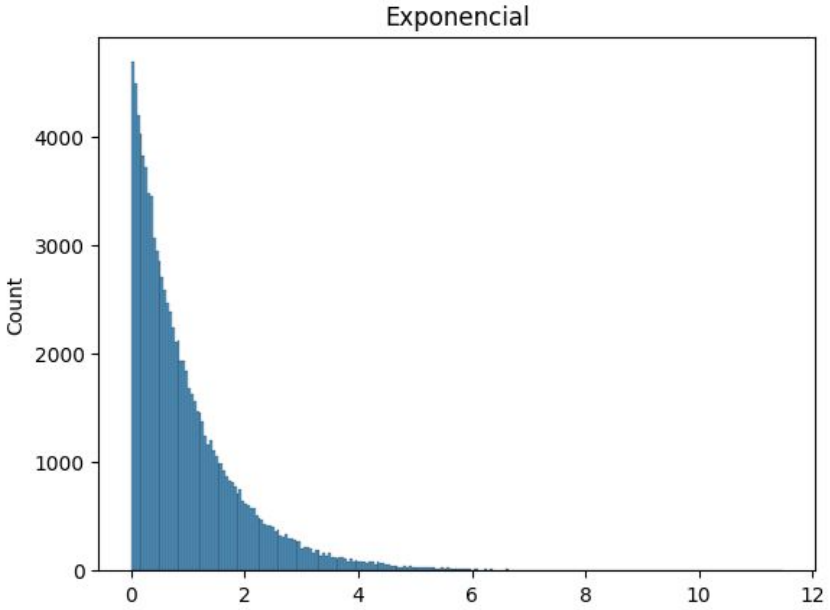
Generación de distribución normal

Para la generación de números pseudoaleatorios que sigan la distribución normal aplicamos el método de Box Muller de forma que no rechazamos ningún valor generado.



Generación de distribución exponencial

Para la generación de números pseudoaleatorios que sigan la distribución exponencial usamos el método de transformada inversa tal que no rechazamos ningún valor generado.



Realizamos distintos tests para verificar uniformidad de los datos para distintos valores de n

H0: la muestra sigue la distribución esperada
H1: la muestra no sigue la distribución esperada

En nuestro caso, las distribuciones esperadas serán:

- uniforme
- normal

Tests de uniformidad

Test Chi cuadrado

```
Test n=10
Datos uniformes (No se puede rechazar H0)

Test n=10000
Datos uniformes (No se puede rechazar H0)

Test n=100000
Datos uniformes (No se puede rechazar H0)
```

Ejemplo de código para Chi cuadrado

```python
for n in [10, 10_000, 100_000]:

    data = gen.nextRand(n)

    bins=50

    frecuencias = np.histogram(data,bins=bins,density=False)[0]

    cantidad = len(data)

    limiteSuperior = chi2.ppf(0.95, df=bins-1)

    Ei=cantidad/bins

    D2 = sum([(Oi - Ei)**2 for Oi in frecuencias])/Ei

    if D2 <= limiteSuperior:
        print("El test acepta la hipotesis nula.")
    else:
        print("El test rechaza la hipótesis nula")
```

Test Kolmogorov Smirnov

```
Test n=10
Datos uniformes (No se puede rechazar H0)

Test n=10000
Datos uniformes (No se puede rechazar H0)

Test n=100000
Datos uniformes (No se puede rechazar H0)
```

Gap Test

```
Test n=10
Datos uniformes (No se puede rechazar H0)

Test n=20
Datos uniformes (No se puede rechazar H0)

Test n=25
Datos uniformes (No se puede rechazar H0)
```

Tests de normalidad

Test Anderson-Darling

```
Test n = 10
Estadístico: 0.252
0.150: 0.501, Datos normales (No se puede rechazar H0)
0.100: 0.570, Datos normales (No se puede rechazar H0)
0.050: 0.684, Datos normales (No se puede rechazar H0)
0.025: 0.798, Datos normales (No se puede rechazar H0)
0.010: 0.950, Datos normales (No se puede rechazar H0)

Test n = 10000
Estadístico: 0.277
0.150: 0.576, Datos normales (No se puede rechazar H0)
0.100: 0.656, Datos normales (No se puede rechazar H0)
0.050: 0.787, Datos normales (No se puede rechazar H0)
0.025: 0.918, Datos normales (No se puede rechazar H0)
0.010: 1.092, Datos normales (No se puede rechazar H0)

Test n = 50000
Estadístico: 0.369
0.150: 0.576, Datos normales (No se puede rechazar H0)
0.100: 0.656, Datos normales (No se puede rechazar H0)
0.050: 0.787, Datos normales (No se puede rechazar H0)
0.025: 0.918, Datos normales (No se puede rechazar H0)
0.010: 1.092, Datos normales (No se puede rechazar H0)
```

Test Saphiro-Wilk

```
Test n=10
Datos normales (No se puede rechazar H0)

Test n=10000
Datos normales (No se puede rechazar H0)

Test n=100000
Datos normales (No se puede rechazar H0)
```

Código de ejemplo para test de Saphiro

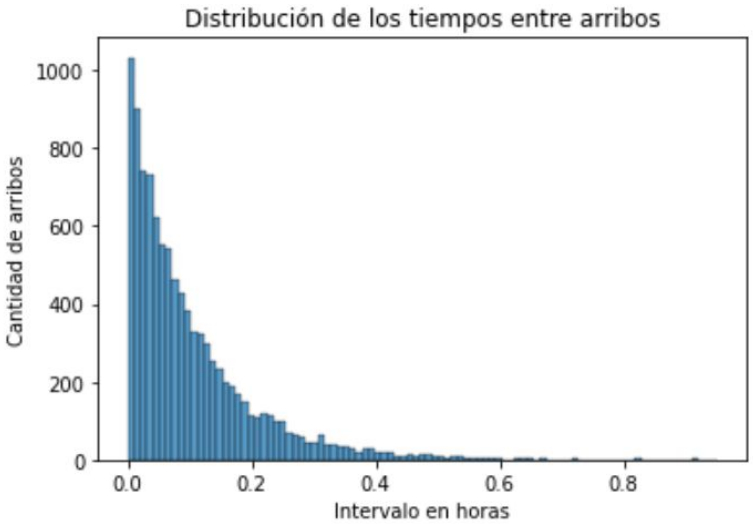
```python
for n in [10, 10_000, 100_000]:
    data = gen.generate_normal(2, 10, n)
    shapiro_test = stats.shapiro(data)
    if shapiro_test.pvalue > 0.05:
        print('Datos normales (No se puede rechazar H0)')
    else:
        print('Datos no normales (Se rechaza H0)')
```

Proceso de Poisson

Se modela la llegada de vehículos a un estacionamiento como un proceso de Poisson de tasa λ vehículos/hora.

Necesitamos estimar el valor de λ, se espera que el tiempo entre arribos tengan distribución exp (λ)

Datos Provistos



Se observa que los tiempos entre arribos efectivamente siguen una distribución exponencial.

Si Tn es la variable aleatoria que modela el tiempo entre arribos, entonces Tn tiene distribución exp(λ).

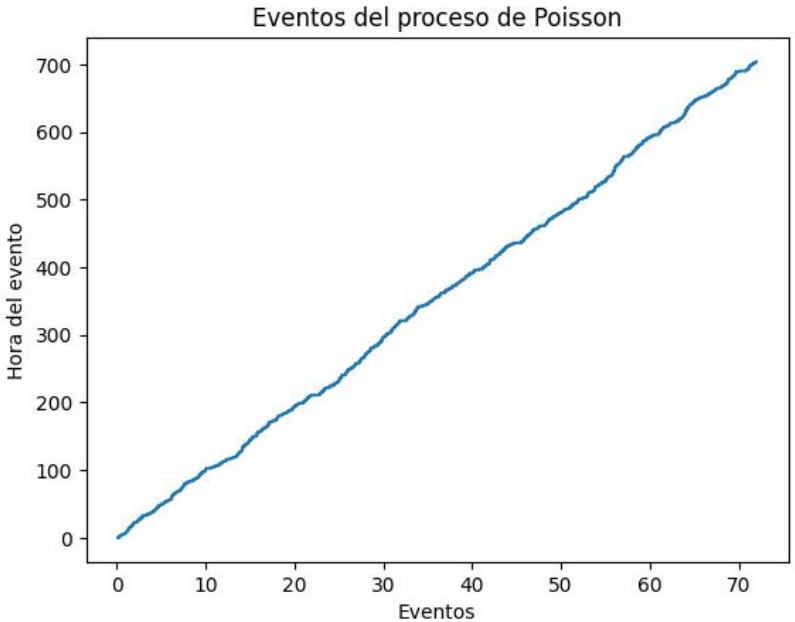
Luego E [Tn] = 1 / λ.

Finalmente λ = 1/E[Tn]

Si estimamos la E [Tn] como el promedio de los valores provistos entonces

λ = 9.89

Graficamos los eventos de forma acumulativa



Test de Kolmogorov Smirnov

Realizamos un Test de Kolmogorov Smirnov

- H0: La muestra sigue una distribucion exponencial
- H1: La muestra no sigue una distribucio exponencial

Vemos si podemos rechazar (o no) H0.

```
# Se repite varias veces el test
for i in range(5):
    # creo muestra exponencial con mismos parametros que nuestros datos
    exp = stats.expon.rvs(scale=1/lam, size=len(tiempos_entre_arribos))

    kstest_test = stats.kstest(tiempos_entre_arribos, exp)

    if kstest_test.pvalue > 0.05:
        print('Datos exponenciales (No se puede rechazar H0)')
    else:
        print('Datos no exponenciales (Se rechaza H0)')
```

```
Datos exponenciales (No se puede rechazar H0)
Datos exponenciales (No se puede rechazar H0)
Datos exponenciales (No se puede rechazar H0)
Datos exponenciales (No se puede rechazar H0)
Datos exponenciales (No se puede rechazar H0)
```

Simulación del Proceso utilizando el generador LXM

Necesitamos que el generador nos de valores que sigan una distribución exp(λ) los cuales representarán el tiempo entre arribos

Se aplicará el método de la transformada inversa para, para ello:
Calculamos la transformada inversa de la función exponencial

Función de densidad de probabilidad $p(x)$

$$f(x;\lambda) = \begin{cases} \lambda e^{-\lambda x} & x \geq 0, \\ 0 & x < 0. \end{cases}$$

Función acumulativa de probabilidad $F_X(x)$

$$F(x;\lambda) = \begin{cases} 1 - e^{-\lambda x} & x \geq 0, \\ 0 & x < 0. \end{cases}$$

Transformada inversa:

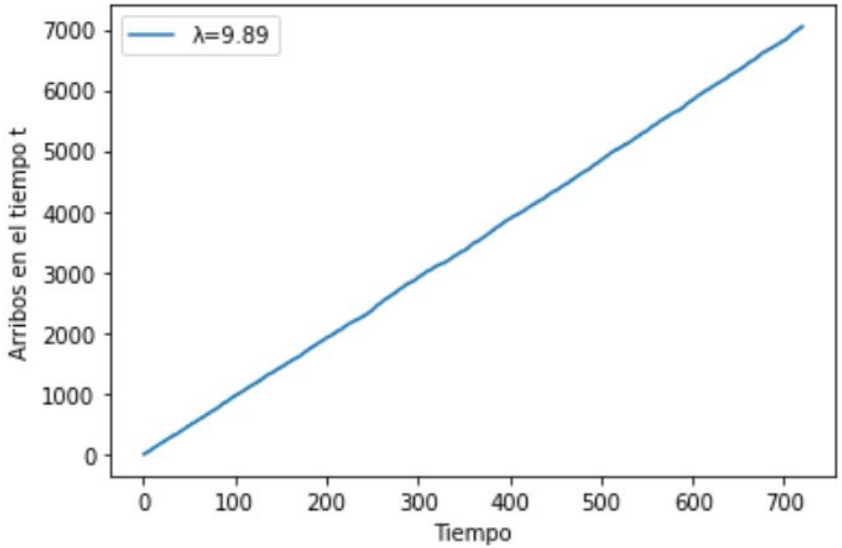
$$X = F_X^{-1}(U) = -\frac{1}{\lambda}\ln(1-U)$$

Generamos valores para el tiempo entre arribos utilizando el generador implementado anteriormente.

Nuestro generador genera números Ui con distribución uniforme [0,1]

Aplicamos $F_X^{-1}(U)$ sobre cada valor generado por nuestro generador para obtener un valor que sigue una distribución exp(λ)

Simulación del Proceso utilizando el generador LXM



Probabilidad que el primer vehículo arribe antes de los 10 minutos

Resultado teórico:

Opción 1

Se interpreta como que, dentro de del intervalo de tiempo de 10 minutos, al menos un vehículo arribe.

$$\lambda = 9.89 \cdot \frac{vechiculos}{hora}$$

$$N(t) \sim Poisson(\lambda t)$$

$$P(N(\tfrac{1}{6}) \geq 1) = 1 - P(N(\tfrac{1}{6}) < 1) = 1 - P(N(\tfrac{1}{6}) = 0) = 1 - \frac{(9.89 \cdot \frac{1}{6})^0}{0!} \cdot e^{-9.89 \cdot \frac{1}{6}} = 1 - e^{-9.89 \cdot \frac{1}{6}} \approx 0.81$$

Opción 2

Se interpreta como la probabilidad de que el tiempo entre arribos sea menor a 10 minutos. Esto es posible ya que se trata del primer arribo

$$T \sim \varepsilon(\lambda)$$

$$P(T \leq \tfrac{1}{6}) = 1 - P(T > \tfrac{1}{6}) = 1 - e^{-9.89 \cdot \frac{1}{6}} \approx 0.81$$

Simulación:

Se corrió el proceso 1000 veces y se calculó cuántas de ellas el primer vehículo arribó antes de los 10 minutos

Resultado = # favorables / 1000 = 0.79

Probabilidad que el undécimo vehículo arribe después de los 60 minutos

Resultado teórico:

Para este caso tenemos dos interpretaciones del mismo problema, que llevan al mismo resultado:

1. Calcular que el tiempo de llegada del onceavo vehículo sea despues de los 60 minutos
2. Calcular la probabilidad de que en 60 minutos no lleguen mas de 10 vehiculos

Para este caso, optamos por la opcion 2.

$$\lambda = 9.89 \cdot \frac{vechiculos}{hora}$$

$$N(t) \sim Poisson(\lambda t)$$

$$P(N(1) \leq 10) = \sum_{n=0}^{10} \frac{(9.89 \cdot 1)^n}{n!} \cdot e^{-9.89 \cdot 1} \approx 0.60$$

Simulación:

Se corrió el proceso 1000 veces y se calculó cuántas de ellas undécimo vehículo arribó después de los 60 minutos

Resultado = # favorables / 1000 = 0.63

Probabilidad que arriben al menos 750 vehículos antes de las 72 horas.

Resultado teórico:

Se interpreta como que, dentro de las primeras 72 horas, arriben al menos 750 vehículos.

$$\lambda = 9.89 \cdot \frac{vechiculos}{hora}$$

$$N(t) \sim Poisson(\lambda t)$$

$$P(N(72) \geq 750) = 1 - P(N(72) < 750) = 1 - \sum_{n=0}^{749} \frac{(9.89 \cdot 72)^n}{n!} \cdot e^{-9.89 \cdot 72} \approx 0.08$$

Simulación:

Se corrió el proceso 1000 veces y se calculó cuántas de ellas arribaron 750 vehículos o más antes de las 72 hs

Resultado = # favorables / 1000 = 0.07

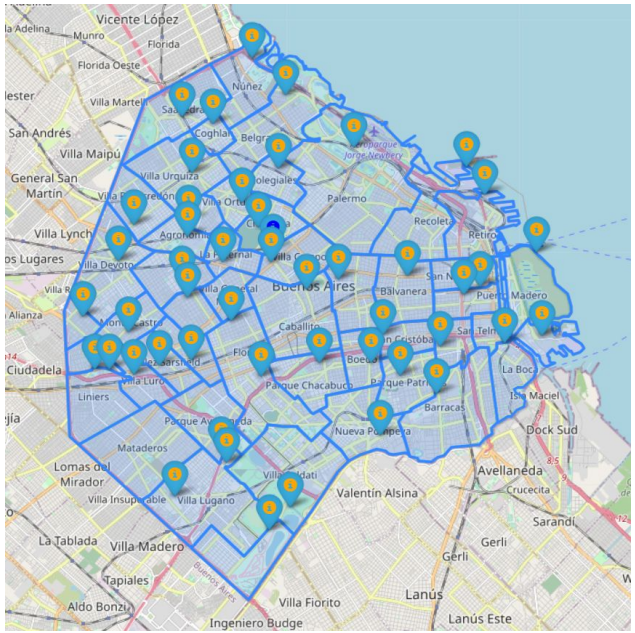
Generación de puntos en la ciudad mediante el generador LMX

Como el titulo lo indica vamos a generar puntos dentro de los distintos barrios de la ciudad de Buenos Aires mediante el generador LMX, y para eso crearemos los vectores lons[] y lats[] que representan la longitud y latitud respectivamente para luego extraer los valores máximos y mínimos de la latitud y longitud de cada uno de los 48 barrios de la capital. Mediante estos valores, podremos hacer funcionar al generador para que nos generen la latitud y la longitud donde irá el punto a graficar

Generación de puntos en la ciudad mediante el generador LMX

Una vez generadas las coordenadas, primero deberemos generar el mapa y lo haremos mediante la función folium.Map() con un origen y zoom arbitrarios. Una vez generado el mapa, generamos el punto con la función folium.Marker(), cuya ubicación será la que nosotros hemos simulado, es decir en el parámetro de la posición, nosotros pondremos [random_lat,random_lon] siendo respectivamente la latitud y la longitud que se ha simulado anteriormente

Mapa de Buenos Aires generados con el generador



Código

```python
import folium
import geopandas as gp
#Aqui declaramos los vectores de las latitudes y longitudes de los barrios de la ciudad
lats=[]
lons=[]
#Aqui plotearemos los limites de los barrios de la ciudad
info_barrios=gp.read_file("caba_barrios.json")

geoPath=info_barrios.geometry.to_json()
poligons=folium.features.GeoJson(geoPath)
m.add_child(poligons)

#Aqui ubicaremos un punto por cada barrio de la ciudad
for k in range(0,len(barrios)):
    barrios=lluvia[k]['geometry']['coordinates'][0]
    for i in barrios:
        for j in i:

            lons.append(j[0])
            lats.append(j[1])
    prim_lat=lats[0]
    b=len(lats)
    c=int(b*0.9)
    ult_lat=lats[c]
    prim_lon=lons[0]
    ult_lon=lons[c]
    random_lat = gen.nextRand(1)
    random_lat=random_lat*(ult_lat-prim_lat)+prim_lat
    random_lon = gen.nextRand(1)
    random_lon=random_lon*(ult_lon-prim_lon)+prim_lon
    folium.Marker([random_lat,random_lon],popup=lluvia[k]['properties']['BARRIO'],icon=folium.Icon(icon_color='orange')).add_to(m)
    lats=[]
    lons=[]
```