Física Numérica

Julio César Avila Torreblanca

23 de septiembre del 2021

Tarea 3

1. **Generador de números aleatorios**

   (a) Escriba un programa que genere números pseudo-aleatorios utilizando el método de congruencias lineales.

   (b) Con un objetivo pedagógico , pruebe su programa con $(a, c, M, x_0) = (57, 1, 256, 10)$. Determine el *periodo*, es decir,cuántos números deben generarse para que la sucesión se repita.

   (c) Tome la sucesión del inciso anterior, gracando los pares $(x_{2i-1}, x_{2i}), i = 1, 2, \ldots$

   (d) Grafique x_i *vs* i.

   (e) Existen diversos métodos para estudiar si un conjunto de números tiene o no una distribución uniforme. Investigue una de ellas,expliquela y aplique dicha prueba a los números generados con su programa.

   (f) Utilice la misma prueba para un conjunto de números generados con al función *random* de Pyhton.

- *Solución*:

(a) Para este ejercicio haremos uso del método de congruencias lineales. Este método establece que para dada una semilla r_0 y a, c, M enteros positivos, es posible generar una secuencia pseudo-aleatoria de números $0 \leq r_i \leq M - 1$ sobre el intervalo $[0, M - 1]$.
La expresión para generar esta secuencia de números es la siguiente:

$$r_{i+1} = ar_i + c \mod M \tag{1}$$

Para crear un programa que genere estos números pseudo-aleatorios crearemos una función llamada `congruencia_lineal` la cual tendrá como parámentros los valores de r_0, a, c, M y guardará la secuencia de números generada por (1) en una lista.
La función es la siguiente:

[32]:
```
def congruencia_lineal(a:int , c: int, M: int, r0)->list:
    num = [r0]
    for i in range(0, M):
        ri = (a* num[i] + c)%M
        num.append(ri)
    return num
```

(b) Para este inciso pondremos a prueba la función definida anteriormente. Notemos que dentro de la función se ha usado un ciclo `for` con un rango de $(0, M)$. La razón de haber colocado el entero M en ese rango es que después de repetir $M-1$ la fórmula de recursión (1) para generar el número r_M, obtenemos el mismo valor inicial r_0. Es decir:

$$r_M = r_0$$

Esto quiere decir que el periodo del método de cogruencias lineales está dado por el módulo M. Veamos lo anterior con un ejemplo pequeño en el que tomaremos $(a, c, M, r_0) = (1, 1, 9, 5)$:

[33]:
```
ejemplo1 = congruencia_lineal(1,1,6,5)
print(ejemplo1)
```

```
[5, 0, 1, 2, 3, 4, 5]
```

Los elementos de la lista anterior son de la forma $[r_0, r_1, ..., r_6]$, por lo que es claro que $r_6 = r_0$. Esto comprueba que el periodo es el módulo, que en nuestro caso es $M = 6$.

Ahora hagamos el ejemplo que se nos pide en este inciso, donde podremos observar que $r_{256} = r_0$. Así el periodo es 256.

[34]:
```
ejemplo2 = congruencia_lineal(57,1,256,10)
print(ejemplo2)
```

```
[10, 59, 36, 5, 30, 175, 248, 57, 178, 163, 76, 237, 198, 23, 32, 33, 90, 11,
116, 213, 110, 127, 72, 9, 2, 115, 156, 189, 22, 231, 112, 241, 170, 219, 196,
165, 190, 79, 152, 217, 82, 67, 236, 141, 102, 183, 192, 193, 250, 171, 20, 117,
14, 31, 232, 169, 162, 19, 60, 93, 182, 135, 16, 145, 74, 123, 100, 69, 94, 239,
56, 121, 242, 227, 140, 45, 6, 87, 96, 97, 154, 75, 180, 21, 174, 191, 136, 73,
66, 179, 220, 253, 86, 39, 176, 49, 234, 27, 4, 229, 254, 143, 216, 25, 146,
131, 44, 205, 166, 247, 0, 1, 58, 235, 84, 181, 78, 95, 40, 233, 226, 83, 124,
157, 246, 199, 80, 209, 138, 187, 164, 133, 158, 47, 120, 185, 50, 35, 204, 109,
70, 151, 160, 161, 218, 139, 244, 85, 238, 255, 200, 137, 130, 243, 28, 61, 150,
103, 240, 113, 42, 91, 68, 37, 62, 207, 24, 89, 210, 195, 108, 13, 230, 55, 64,
65, 122, 43, 148, 245, 142, 159, 104, 41, 34, 147, 188, 221, 54, 7, 144, 17,
202, 251, 228, 197, 222, 111, 184, 249, 114, 99, 12, 173, 134, 215, 224, 225,
26, 203, 52, 149, 46, 63, 8, 201, 194, 51, 92, 125, 214, 167, 48, 177, 106, 155,
132, 101, 126, 15, 88, 153, 18, 3, 172, 77, 38, 119, 128, 129, 186, 107, 212,
53, 206, 223, 168, 105, 98, 211, 252, 29, 118, 71, 208, 81, 10]
```

(c) En esta parte graficaremos las parejas ordenadas (r_{2i-1}, r_{2i}) donde consideraremos los elementos de la lista anterior. Para ello importaremos la siguiente librearia:

[35]:
```
import matplotlib.pyplot as plt
```

Ahora debemos crear una una lista que contenga los elementos impares, y otra con los elementos pares.

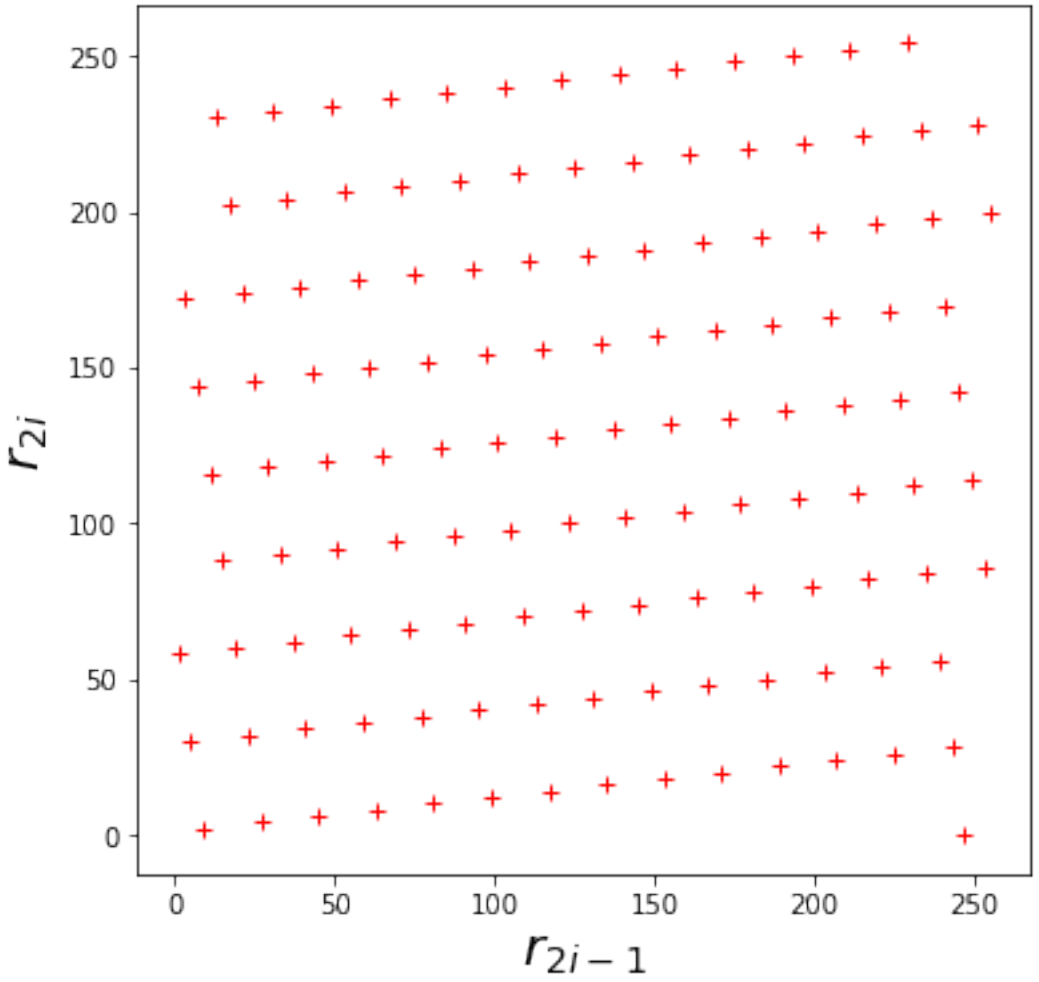
$$[r_1, r_3, ..., r_{253}, r_{255}] \quad ; \quad [r_2, r_4, ..., r_{254}, r_{256}]$$

Donde los elementos impares serán graficados en el eje x y los pares ene el eje y.

[36]:
```
#Lista con los elementos impares a partir de r_1 hasta r_255
x = [ejemplo2[i] for i in range(1,len(ejemplo2),2)]
#Lista con los elementos pares a partir de r_2 hasta r_256
y = [ejemplo2[i] for i in range(2,len(ejemplo2),2)]
```

Ya con las listas x y y graficaremos los puntos (r_{2i-1}, r_{2i}), con $i = 1, 2, ..., 256$.

[37]:
```
plt.figure(figsize=(6, 6))#Damos tamaño a la figura
plt.plot(x, y, 'r+') #Graficamos los puntos
plt.xlabel('$r_{2i - 1}$',fontsize=20) #nombre del eje x
plt.ylabel('$r_{2i}$',fontsize=20) #nombre del eje y
plt.show() #Mostramos la gráfica
```



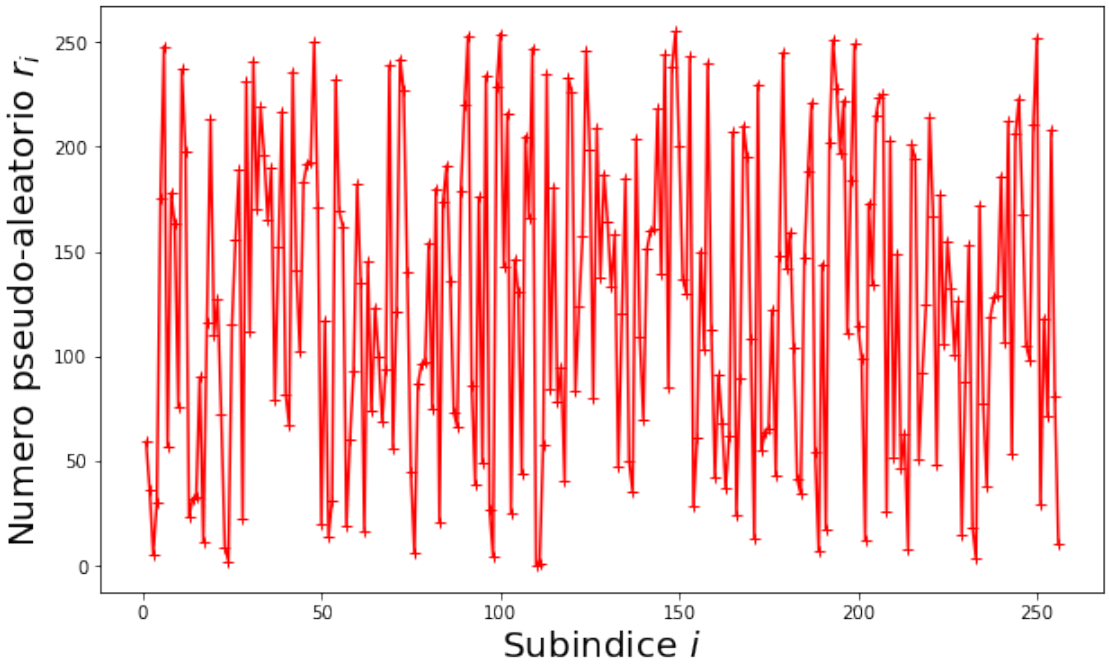
Con este gráfico se puede visualizar que la generación de los números no es aleatoria, ya que es posible encontrar un patrón entre las parejas ordenadas (r_{2i-1}, r_{2i}).

(d) Ahora graficaremos r_i *vs* i, con $i = 1, 2, ..., 256$. Para ello tomaremos los valores que hemos

guardado en la lista llamada `ejemplo2`.

[38]:
```
subindice = [i for i in range(1,257)] #Creamos la lista de subindices [1,2,...
 ↪,256] que se usará en el eje x

plt.figure(figsize=(10, 6))#Damos tamaño a la figura
plt.plot(subindice, ejemplo2[1:257], 'r-+') #Graficamos los puntos
plt.xlabel('Subindice $i$',fontsize=20) #nombre del eje x
plt.ylabel('Numero pseudo-aleatorio $r_{i}$',fontsize=20) #nombre del eje y
plt.show() #Mostramos la gráfica
```



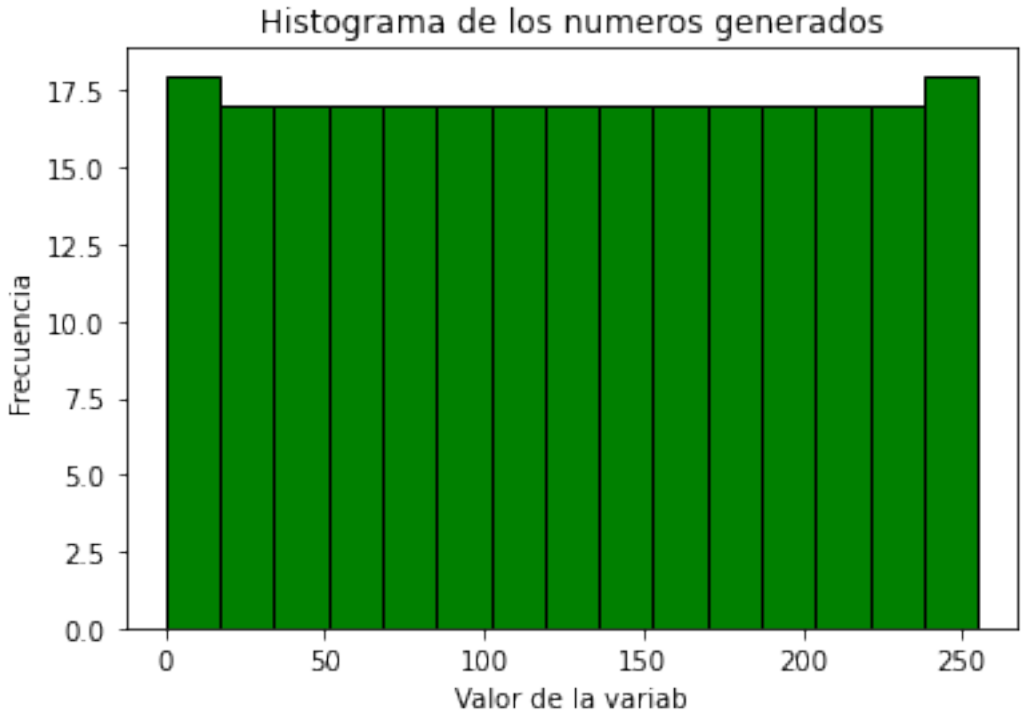
La gráfica anterior contiene las parejas (r_i, i) generadas en el inciso (b). Hemos conectado a los puntos a través de líneas para ver como fluctúan de uno a otro. No es sencillo ver la relaciión que exisrte de un puno a otro, pero esto no es una prueba de que sean aleatorios.

(e) Para ver si la distribución de los números pseudo-aleatorios generados tienen una distribución uniforme crearemos un histograma para ver la frecuencia de estos números y ver que tan bien están distribuidos. Además, evaluaremos si los datos tienen una distribución Gaussiana.
La distribución Gaussiana (o normal) es una de las distribuciones de probabilidad que aparece mucho en estadística. Para saber si un conjunto de datos tiene una distribución Gaussiana a través de un histograma, las distribución de los datos deben de tener forma de campana. Por otro lado, la librería `scipy` incluye una función que realiza un test de *Shapiro-Wilk* que permite estimar si una distribución es Gaussiana o no. La función es `scipy.stats.shapiro` y como parámetro hay que ingresar un arreglo de valores. Esta función regresa dos valores flotantes: `estadistico` y `p-valor`. Para evaluar si es Gaussiana la

distribución debe cumplirse que `p-valor > 0.05`. Si no se cumple, entonces no es Gaussiana.

Primeramente generemos el histograma para el conjunto de numeros que hemos generado por el método de congruencias lineales.

[39]:
```
#histograma
plt.hist(ejemplo2, 15, color = "green", ec = "black")
plt.title('Histograma de los numeros generados')
plt.xlabel('Valor de la variab')
plt.ylabel('Frecuencia')
plt.show()
```



En el histograma anterior hemos creado 15 clases, y se puede observar que no se tiene una distribución Gaussiana ya que no se genera la campana. Estos números generados están uniformemente distribuidos, lo cual tiene sentido, ya que solo aparecen una vez antes de acuerdo a su periodo estimado en el inciso (b).
Ahora apliquemos el test de *Shapiro-Wilk*.

[40]:
```
from scipy.stats import shapiro #importamos el test de shapiro-wilk

# Prueba de Shapiro-Wilk
estadistico, p_valor = shapiro(x)
print(f'Estadisticos = {estadistico:.3}')
print(f'p = {p_valor:.3}')
```

```
# Interpretación
alpha = 0.05
if p_valor > alpha:
   print('La muestra se aproxima a una distribución Gaussiana o Normal')
else:
   print('La muestra no es una distribución Gaussiana o Normal')
```

```
Estadisticos = 0.955
p = 0.000295
La muestra no es una distribución Gaussiana o Normal
```

De acuerdo a este test, nuestros datos no tienen una distribución Gaussiana.

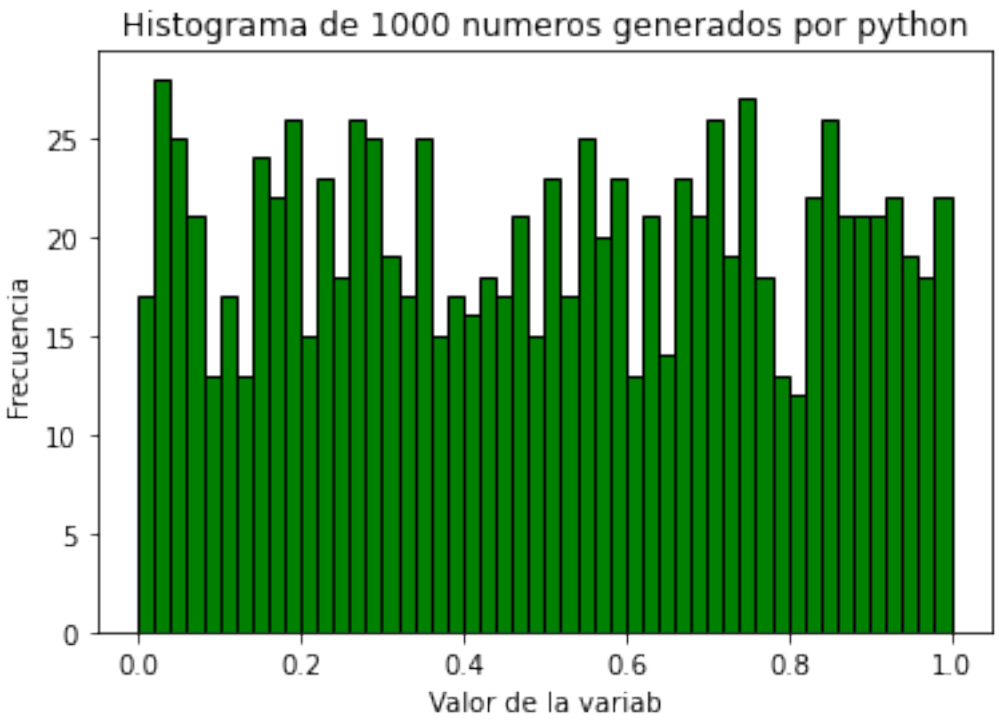
(e) En este inciso generaremos 1000 numeros aleatorios en el intervalo $[0, 1]$ para ver la distribución que tienen con un histograma y con el test de *Shapiro-Wilk*. El código es el siguiente:

[41]:
```
import numpy as np

#histograma
n=1000
x = np.random.rand(n) #generamos los n números aleatorios en el intervalo [0,1]
plt.hist(x, 50, color = "green", ec = "black")
plt.title(f'Histograma de {n} numeros generados por python')
plt.xlabel('Valor de la variab')
plt.ylabel('Frecuencia')
plt.show()

# Prueba de Shapiro-Wilk
estadistico, p_valor = shapiro(x)
print(f'Estadisticos = {estadistico:.3}')
print(f'p = {p_valor:.3}')

# Interpretación
alpha = 0.05
if p_valor > alpha:
   print('La muestra se aproxima a una distribución Gaussiana o Normal')
else:
   print('La muestra no es una distribución Gaussiana o Normal')
```




```
Estadisticos = 0.952
p = 1.6e-17
La muestra no es una distribución Gaussiana o Normal
```

Hemos generado 50 marcas de clase para el histograma. Se puede ver claramente que no es una distribución Gaussiana, y esto se puede corroborar con el test de *Shapiro-Wilk* ya que arroja el mismo resultado.

---

2. **Integración.** Elabore un programa que estime, utilizando el método de Montecarlo, las siguientes integrales:

   (a) $\int_0^1 (1-x^2)^{3/2} dx$

   (b) $\int_{-2}^2 e^{x+x^2} dx$

- *Solución*:

El algoritmo que desarrollaremos estimará las integrales en el intervalo $[0,1]$. Por lo que, para la segunda integral utilizaremos el siguiente cambio de variable que nos permitirá restringirnos a ese intervalo.

$$y = \frac{x+2}{4} \quad \Rightarrow dy = \frac{dx}{4}$$

Sustituyendo el valor de x encontramos que:

$$x + x^2 = 4y - 2 + (4y-2)^2 = 16y^2 - 12y + 2$$

Por lo que:

$$\int_{-2}^{2} e^{x+x^2} dx = \int_{0}^{1} 4e^{16y^2-12y+2} dy$$

De esa forma calcularemos el valor para las integrales en el intervalo $[0,1]$.

Lo primero que haremos será crear dos funciones que evaluen $f_1(x_0) = (1-x_0^2)^{3/2}$ y $f_2(x_0) = 4e^{16x_0^2-12x_0+2}$ para un x_0 dado como parámetro.
Las funciones son las siguientes:

[42]:
```
import numpy as np

def f1(x):
    return (1-x**2)**(3/2)

def f2(x):
    return 4*np.exp(16*x**2 - 12*x +2)
```

Ahora generaremos una función cuyo objetivo sea obtener una estimación del valor de la integral a través de la siguiente expresión:

$$\int_{0}^{1} g(x)dx \approx \frac{1}{N}\sum_{i=0}^{N} g(U_i) \tag{2}$$

Donde N es la cantidad de números aleatorios $U_1, U_2, ...U_N$ generados en el intervalo $[0,1]$ para obener la aproximación de la integral. La función es la siguiente:

[43]:
```
def experimento(f)->float:
    """Esta función genera n números aleatorios y los evalua
    con la función f para obtener la estimación de la integral en
    [0,1]

    Parametros
    -----------
    f: function -> es la funcion a ser integrada

    ------
    return: float -> es el valor de la integral de f en [0,1]
    """

    n = 100 #cantidad de numeros a generar en [0,1]
    numeros = np.random.rand(n)
    suma = 0.0 #aquí guardaremos las evaluaciones

    for x in numeros:
        suma += f(x)

    return suma/n
```

La función se llamo `experimento` y el único parámetro que tiene es la función f a ser integrada. Lo que se hace dentro de la función es generar 100 números aleatorios en el intervalo $[0, 1]$ y se utiliza (2) para obtener la aproximación deseada. Por último, la función regresa el valor númerico que se calculó para la integral. Cada vez que se calcule una integral se estará obteniendo una variable aleatoria para después obtener una mejor aproximación con el método visto en clase.

Lo siguiente que haremos será crear uns función llamada `refinamiento` cuyo objetivo será mejorar la estimación hecha para la integral. Los parámetros serán la función f a ser integrada y un valor d para el error aceptable. El código de la función es el siguiente:

[44]:
```
def refinamiento(f, d:float):
    """Esta función estima el valor de la integral de forma más precisa con
    un intervalo de confienza

    Parametros
    -----------
    f: function -> es la funcion a ser integrada
    d: float -> es el valor adecuado para nuestro error cuadratico medio

    ------
    return:  -> solo imprime el valor de la integral con el intervalo de
                confianza junto con el numero de iteraciones hechas
    """

    Xj = experimento(f) #obtenemos X1
    j = 1
    Sj = 0
    error_cuadratico = d + 1
    while error_cuadratico >= d:
        Xj1 = Xj + (experimento(f) - Xj)/(j+1) #obtenemos Xj+1

        Sj1 = (1 - 1/j)*Sj + (j+1)*(Xj1 - Xj)**2 #obtenemos Sj+1

        Xj = Xj1#guardamos el Xj+1 en Xj el anterior para el siguiente cálculo
        Sj = Sj1 #mismo que arriba
        #print(j) #numero de experimentos
        #print(Xj) #estimacion de la integral en el experimento j
        error_cuadratico = (Sj/j)**(1/2)
        j += 1

    intervalo = 1.96*error_cuadratico # calcula el intervalo de confianza
    print(f"Numero de experimentos realizados: {j}")
    print(f"Estimacion con intervalo de confianza: {Xj:} +- {intervalo:e}")
```

En el código anterior se usaron las siguientes fórmulas de recurrencia:

$$\bar{X}_{j+1} = \bar{X}_j + \frac{X_{j+1} - \bar{X}_j}{j+1} \tag{3}$$

$$S_{j+1}^2 = \left(1 - \frac{1}{j}\right) S_j^2 + (j+1)(\bar{X}_{j+1} - \bar{X}_j)^2 \tag{4}$$

Donde X_j son las variables aleatorias (en este caso cada experimento generado con la función `experimento`), $\bar{X}_j$ es la media muestral y S_j^2 la varianza muestral. Para poder haber usado (3) y (4) se supuso que $S_1^2 = 0$ y $\bar{X}_0 = 0$. Con estas suposiciones de (3) se obtiene que:

$$\bar{X}_1 = X_1$$

Donde X_1 será el primer experimento generado (el primer cálculo de la integral). Esto se puede ver en la variable `Xj` que se usó al inicio de la función definida. Luego se inicializó una variable `j` con valor 1, ya que esta será usada en la fórmula de recursión (4). Además se inicializío la variable `error_cuadratico` con un valor mayor a d para poder entrar en el ciclo while. Dentro del ciclo while se calcula el valor de $\bar{X}_{j+1}$ y S_{j+1}^2 a partir de (3) y (4). Después se reescriben las variables para la siguiente iteración. Luego se calcula el error cuadrático dado por $\frac{S_j}{\sqrt{j}}$ y se aumenta la variable `j` en 1 para la siguiente iteración del while. El cilo se repite hasta que se cumpla que:

$$\frac{S_j}{\sqrt{j}} < d$$

Cuando esto suceda, habremos terminado nuestra estimación para la integral y su valor estará dado por $\bar{X}_j$ con un intervalo de confianza dado por $1.96\frac{S_j}{\sqrt{j}}$. Es decir, después de un número j de iteraciones, se tendrá que:

$$\int_0^1 g(x)dx \approx \bar{X}_j \pm 1.96\frac{S_j}{\sqrt{j}}$$

Finalmente, la función imprime el número de iteraciones hechas, junto con la estimación obtenida.

Pongamos a prueba nuestra función para las integrales que se piden en el problema:

(a) Aquí propondremos que $d = 10^{-4}$. Calculando:

[45]:
```
print("\nIntegral 1:")
refinamiento(f1, 10**(-4))
```

```
Integral 1:
Numero de experimentos realizados: 111203
Estimacion con intervalo de confianza: 0.5890940104194493 +- 1.959988e-04
```

(b) En la segunda integral propondremos que $d = 10^{-1}$. Con este valor tarda unos minutos en generar el valor numérico, por lo que no es recomendable usar más pequeños que este.

[49]:
```
print("\nIntegral 2:")
refinamiento(f2,10**(-1))
```

```
Integral 2:
Numero de experimentos realizados: 59582
Estimacion con intervalo de confianza: 93.26780978233633 +- 1.959997e-01
```

Usando Mathematica para verificar el valor numérico de las integrales, se observa que las estimaciones obtenidas fueron muy precisas, y además el intervalo de confianza ofrecido es adecuado.

---

3. **Calculando π.** Elabore un programa que estime, utilizando el método de Montecarlo, el valor de π.

- *Solución*:

Lo primero que haremos será crear una función que estime el valor de π a partir de n parejas (x, y), $x, y \in [0, 1]$, generadas aleatoriamente. El proceso a seguir es evaluar cuantas de puntos caen dentro de la cuarta parte de la circunferenfia unitaria. Para que un punto esté dentro debe cumlplirse que:

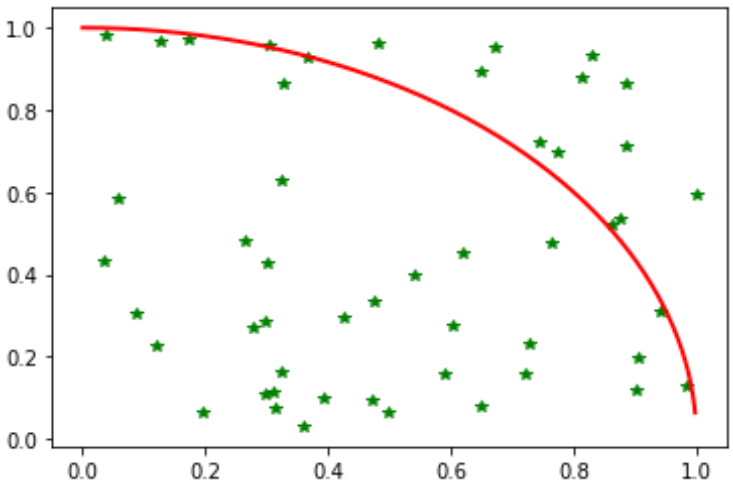
$$x^2 + y^2 \leq 1$$

Supongamos que m de los n puntos están dentro de la circunferencia, así podemos hacer la siguiente aproximación con el área de la cuarta parte de la circunferencia unitaria.

$$\frac{m}{n} \approx \frac{\pi}{4}$$

Así:

$$\pi \approx 4\frac{m}{n}$$



Ya con esto procedemos a generar la función que estimará pi. La función recibirá como parámetros la cantidad de puntos que se deseen generar y regresará una estimación para π. El código es el siguiente:

[55]:
```
import numpy as np


def experimento_pi(n:int)->float:

    """Esta función aproxima a pi usando una relación de proporción entre areas
    y n puntos generados aleatoriamente
```

```
    Parametros
    -----------
        n: int -> numero de puntos a generar aleatoriamente

    ----------
    return: float -> es la estimación de pi
    """
    m = 0 #numero de puntos dentro de la circunferencia
    x = np.random.rand(n)
    y = np.random.rand(n)

    for i in range(n):

        z = x[i]**2 + y[i]**2 #Condición a cumplirse para estar dentro de la␣
↪circunferencia
        if z <= 1:
            m += 1 #Si cae el punto aumentamos nuestro contador en 1

    return (4.0*float(m/n))
```

Cada que se llame a esta función estaremos llevando a cabo un experimento. Por lo que esta función es la análoga a la función `experimento` definida en el ejercicio 2 para poder lleavar a cabo el método de Montecarlo. Así que nuestras variables aleatorias estarán dadas por la función deinifida anteiriormente `experimento_pi`.
Lo siguiente es adecuar a la función `refinamiento` que se uso en el ejercicio 2. El ajuste queda como el siguiente:

[56]:
```
def refinamiento_pi(d: float, n = 1000):
    """Esta función estima el valor pi más precisa con un intervalo
    de confienza

    Parametros
    -----------
        n: int -> numero de puntos a generar aleatoriamente
        d: float -> es el valor adecuado para nuestro error cuadratico medio

    ------
    return:  -> solo imprime el valor pi con el intervalo de confianza,
                junto con el numero de experimentos hechos
    """

    Xj = experimento_pi(n) #obtenemos X1
    j = 1
    Sj = 0
    error_cuadratico = d + 1
    while error_cuadratico >= d:
```

```
        Xj1 = Xj + (experimento_pi(n) - Xj)/(j+1) #obtenemos Xj+1

        Sj1 = (1 - 1/j)*Sj + (j+1)*(Xj1 - Xj)**2 #obtenemos Sj+1

        Xj = Xj1#guardamos el Xj+1 en Xj el anterior para el siguiente cálculo
        Sj = Sj1 #mismo que arriba
        #print(j) #numero de experimentos
        #print(Xj) #estimacion de la integral en el experimento j
        error_cuadratico = (Sj/j)**(1/2)
        j += 1

    intervalo = 1.96*error_cuadratico # calcula el intervalo de confianza
    print(f"Numero de experimentos realizados: {j}")
    print(f"Estimacion con intervalo de confianza: {Xj:.8} +- {intervalo:e}")
```

Lo único que cambió fue que en la fórmula de recursión para $\bar{X}_{j+1}$ se llamó a la función `experimento_pi` y se colocó como parámetro $n = 1000$ para que cada experimento obtenga una aproximación de π con mil puntos. El resto de la función permanece igual como en el ejercicio 2, por lo que ya no hay necesidad de explicarlo.

Pongamos a prueba el algoritmo tomando $d = 10^{-3}$.

[57]:
```
refinamiento_pi(10**(-3))
```

```
Numero de experimentos realizados: 2643
Estimacion con intervalo de confianza: 3.1418388 +- 1.959960e-03
```

Como se puede apreciar, al igual aquí se obtuvo una aproximaicón muy cercana al valor de π.