PRÁCTICA 0 - PYTHON

Alumnos: Arturo Barbero Pérez y Adrián Ogáyar Sánchez

La práctica consiste en implementar un algoritmo de integración basado en el método de Monte Carlo. Dicho método, es un método de aproximación que calcula la integral definida en un intervalo [a, b] mediante métodos numéricos. La función utilizada debe ser positiva, real e integrable de una sola variable f(x), donde x ϵ [a, b].

El algoritmo permite aproximar el valor de la integral mediante el siguiente cálculo:

$$I \approx \frac{N_{debajo}}{N_{total}} * (b - a) * M$$

Donde (N_{debajo} / N_{total}) es el porcentaje de puntos (x, y) generados aleatoriamente que caen por debajo de la gráfica y donde *M* es el máximo de la función *f(x)* en el intervalo [a, b].

Para el desarrollo de la práctica se han realizado dos versiones del mismo algoritmo. Una implementada de manera iterativa y otra utilizando las operaciones entre vectores que nos proporciona la librería *NumPy*. Posteriormente se han evaluado los tiempos de ejecución de las dos versiones.

VERSIÓN ITERATIVA

```
#------------------------------------------------------------------------
#----------------------- LLAMADA A LA FUNCION ---------------------------
#------------------------------------------------------------------------

a = 0
b = np.math.pi
fun = lambda x: x**2

resIntegral, valor = integrate.quad(fun, a, b)

print("El resultado obtenido mediante 'integrate.quad' es: " + str(resIntegral))

#Llamada a integra con el segmento deseado
result, tiempo = integra_mc(fun, a, b, 10000)
print("El resultado de la funcion es: " + str(result))

#Muestra el tiempo de ejecucion
print("Tiempo de ejecucion: " + str((tiempo)))
```

Definimos unas variables que utilizamos para pasárselas al método *integra_mc(): 'a'* inicializada a 0, '*b'* inicializada a π y la *función lambda* que en este caso es $f(x) = x^2$

También realizamos los correspondientes *print()* para mostrar los resultados.

En esta versión comenzamos poniendo en marcha el contador de tiempo, para poder obtener unas conclusiones posteriormente. Creamos unas listas vacías, *pX* y *pY*, las cuales van a contener los puntos aleatorios (x, y) respectivamente. Acto seguido llamamos a la función *valorMax()* la cual se va a encargar de obtener el valor .

```
def integra_mc(fun, a, b, num_puntos=10000):
    #Tiempo inicial
    tic = time.process_time()

    #Array de puntos aleatorios
    pX = []
    pY = []

    #Calcula el maximo de la funcion
    max = valorMax(fun, a, b)

    nDebajo = 0
    for i in range(0, num_puntos):
        x = np.random.uniform(a, b)
        y = np.random.uniform(0, max)

        pX.append(x)
        pY.append(y)

        if fun(x) > y: nDebajo += 1

    #Dibuja los puntos creados aleatoriamente
    pX = np.array(pX)
    pY = np.array(pY)
    plt.plot(pX, pY, 'r.', label='Puntos aleatorios')

    #Establece la información de los ejes
    plt.xlabel('x')
    plt.ylabel('f(x)')
    plt.legend(loc=0) #El parámetro permite colocar la leyenda en la mejor posición posible.
    plt.show() #Muestra la grafica

    #Obtiene el tiempo final
    toc = time.process_time()

    return (((nDebajo / num_puntos) * ((b - a)*max)), toc-tic)
```

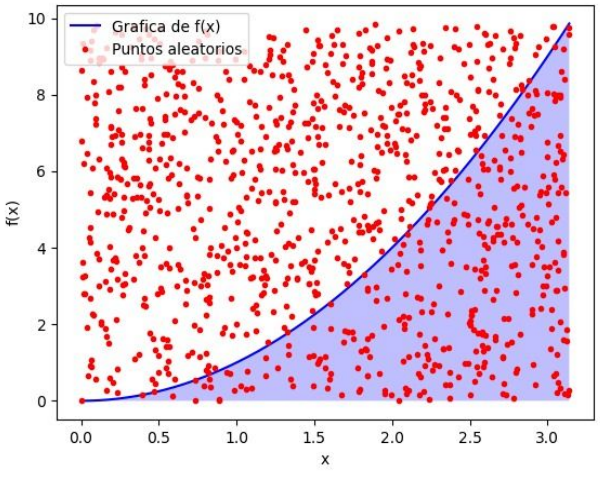
Iteramos con la variable *i* en un rango de [0, *num_puntos*), generando con cada iteración un punto *x,* y un punto *y* aleatorio con las debidas operaciones de *NumPy*.
Añadimos estos puntos a sus correspondientes vectores y realizamos una cuenta en una variable *nDebajo* en caso de que el punto generado aleatoriamente caiga por debajo de la gráfica.
Una vez realizado el bucle metemos los vectores en la función *plot()* y con la función *show()* mostramos la gráfica.
Por último, paramos el contador del tiempo y devolvemos: la operación previamente explicada para calcular el resultado de la integral y el tiempo que ha tardado en ejecutarse el código.

```
def valorMax(fun, a, b):
    puntosX = []
    puntosY = []

    max = -1

    for i in np.arange(a, b, 0.01):
        if(fun(i) > max):
            max = fun(i)

        puntosX.append(i)
        puntosY.append(fun(i))


    puntosX = np.array(puntosX)
    puntosY = np.array(puntosY)

    plt.plot(puntosX, puntosY, '-b', label='Grafica de f(x)')
    plt.fill_between(puntosX, puntosY, facecolor='blue', alpha=0.25)

    return max
```

En la función *valorMax()* se itera con la variable *i* en un rango de [a, b] utilizando el método que nos proporciona *NumPy*: *arange()*. Con este método podemos generar una progresión aritmética a hasta b con una distancia entre un valor y otro, en este caso, de 0.01. Se ha escogido esta distancia para poder dibujar posteriormente la gráfica de la función de una mejor manera.
En cada iteración del bucle se comprueba si la imagen de *i* supera al máximo que hay hasta el momento. Si es así se actualiza el máximo.
Al acabar el bucle se procede a llamar al método *plot()* de *NumPy* el cual va a permitir lineal la gráfica con un color azul y ponerle una etiqueta la cual se verá más tarde en una leyenda. Además, se rellena el área de la integral con la función *fill_between()*

Ejemplos de ejecución con a = 0, b = π y num_puntos = 1000:

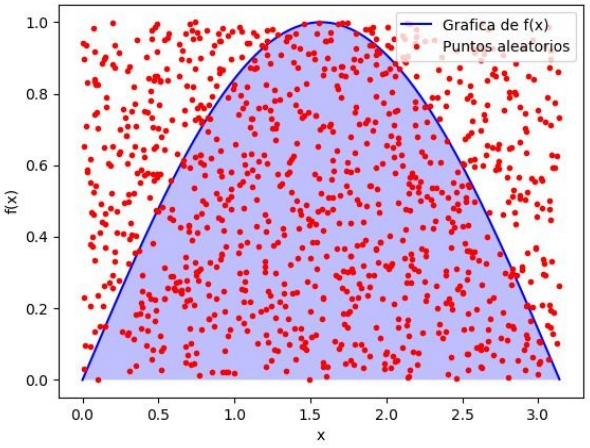
Gráfica: $f(x) = x^2$




```
n.python-2018.8.0/pythonFiles/experimental/ptvsd" /home/arturo/anaconda3/b:
erativo.py

El resultado obtenido mediante 'integrate.quad' es: 10.335425560099939
El resultado de la funcion es: 10.438523414511533
Tiempo de ejecucion: 0.9758602879999998
```

Gráfica: $f(x) = sen(x)$

```
n.python-2018.8.0/pythonFiles/experimental/ptvsd" /home/arturo/
erativo.py

El resultado obtenido mediante 'integrate.quad' es: 2.0
El resultado de la funcion es: 1.998052294164132
Tiempo de ejecucion: 0.9462187539999993
```



VERSIÓN CON VECTORES

En esta versión, la llamada la función es igual que la mencionada anteriormente, lo único que cambia es el método *integra_mc()* y *valorMax()*.

```
def integra_mc(fun, a, b, num_puntos=10000):
    #Tiempo inicial
    tic = time.process_time()

    #Calcula el maximo de la funcion
    max = valorMax(fun, a, b)

    #Array de puntos aleatorios
    pX = np.random.uniform(a, b, (1, num_puntos))
    pY = np.random.uniform(0, max, (1, num_puntos))

    #Obtenemos las imagenes de cada elemento de pX
    imagen_pX = fun(pX)

    #Obtenemos que elementos son mayores y cuales no
    resultResta = imagen_pX > pY

    #Obtenemos el número de elementos True de resultResta
    nDebajo = len(resultResta[resultResta == True])

    #Dibuja los puntos creados aleatoriamente
    plt.plot(pX[0], pY[0], 'r.', label='Puntos aleatorios')


    #Establece la información de los ejes
    plt.xlabel('x')
    plt.ylabel('f(x)')
    plt.legend(loc=0) #El parámetro permite colocar la leyenda en la mejor posición posible.
    plt.show() #Muestra la grafica

     #Obtiene el tiempo final
    toc = time.process_time()

    return (((nDebajo / num_puntos) * ((b - a)*max)), toc-tic)
```

Comenzamos de la misma manera, poniendo en marcha el contador de tiempo y llamando a la función *valorMax()* para obtener el valor *M* que actúa como cota superior en el eje 'y' en el momento de conseguir las coordenadas de los puntos aleatorios.

Para obtener estos puntos aleatorios, se ha optado por utilizar la función *uniform()* la cual devuelve una matriz de números aleatorios reales comprendidos entre [*a*, *b*) (en el caso de pX) en la forma en la que le indiques por el tercer parámetro. En este caso hemos utilizado la forma *(1, num_puntos)* ya que de esta manera la matriz tendría una sola fila con *num_puntos* elementos a la cual se podría acceder más tarde usando el operador de indexación.
Una vez hemos hecho esto, calculamos las imágenes de cada elemento de *pX* pasándoselo a nuestra *función lambda* previamente creada.
Al obtener esto, ya solo nos queda comparar estas imágenes con *pY* mediante el operador '*>'. NumPy* lo que hace cuando comparas dos estructuras, es ir comparando elemento a elemento si se cumple la propiedad y devuelve una estructura del mismo tipo que la que se están comparando, cuyos elementos son booleanos indicando si se ha cumplido esta propiedad o no.
Una vez tenemos esto solo nos queda obtener los elementos que son *True* en esa estructura (En el código: *resultResta[resultResta == True]*) y calcular la longitud de esta mediante la función *len()*.

Con esto nos aseguramos tener el número de puntos que caen por debajo de la curva de la función *f(x)* y podemos pasar a graficarlo todo mediante *plot()*

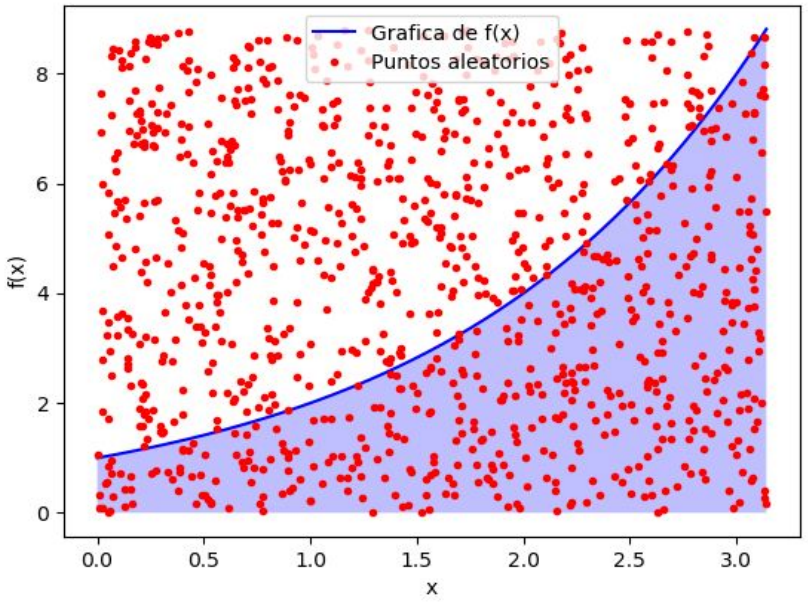
```
def valorMax(fun, a, b):
    puntosX = np.arange(a, b, 0.01)
    puntosY = fun(puntosX)

    max = puntosY.max()

    plt.plot(puntosX, puntosY, '-b', label='Grafica de f(x)')
    plt.fill_between(puntosX, puntosY, facecolor='blue', alpha=0.25)

    return max
```

En cuanto a la función *valorMax(),* el funcionamiento es parecido a lo ya explicado: Obtenemos una progresión aritmética comprendida entre *a* y *b* con una distancia de 0.01 entre cada elemento y le pasamos este resultado a la función lambda.
Con este array, *puntosY,* podemos obtener su máximo mediante la función *max()*.

Una vez hecho todo esto podemos pasar a graficar la función *f(x)* y rellenar el área de la integral calculada.

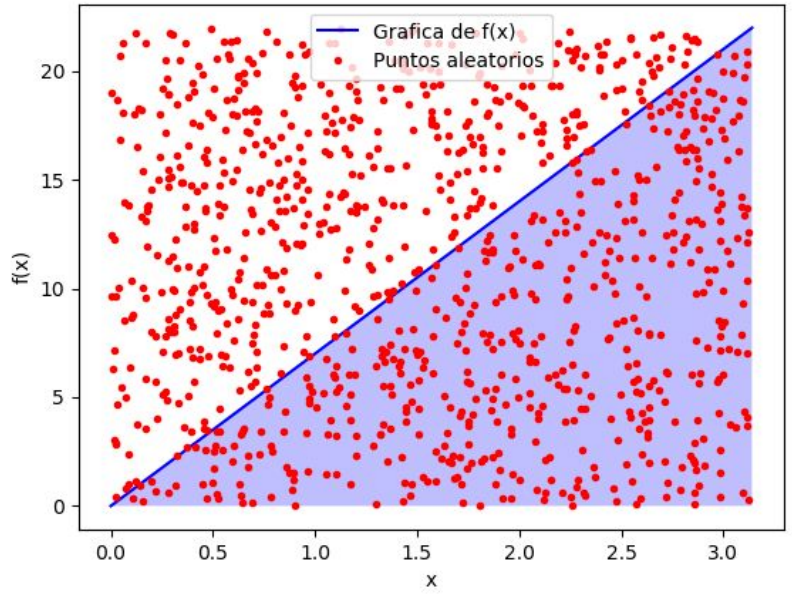
Ejemplos de ejecución con a = 0, b = π y num_puntos = 1000:

Gráfica: *f(x) = 2^x*




```
n.python-2018.8.0/pythonFiles/experimental/ptvsd" /home/arturo/anaconda3/
ctores.py

El resultado obtenido mediante 'integrate.quad' es: 11.289056706189056
El resultado de la funcion es: 11.742211961633371
Tiempo de ejecucion: 0.9016295280000004
```

Gráfica: *f(x) = 7x*



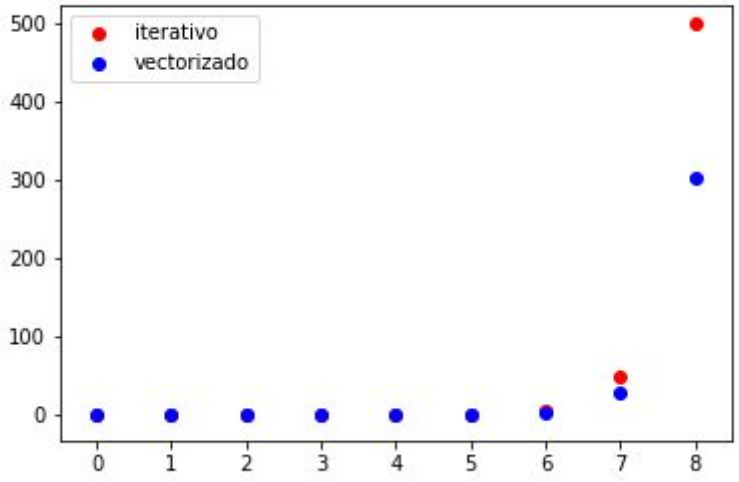
```
n.python-2018.8.0/pythonFiles/experimental/ptvsd" /home/arturo/anaconda
ctores.py

El resultado obtenido mediante 'integrate.quad' es: 34.54361540381276
El resultado de la funcion es: 34.18084223032231
Tiempo de ejecucion: 0.9024560790000002
```

COMPARACIÓN DE TIEMPOS

Para comparar los tiempos de ambas funciones se usa la integral de $f(x) = x^2$ en un intervalo entre 0 y π ejecutando las dos funciones repetidamente donde el número de puntos para generar la integral es igual a 10^i donde $i = (0, 9) \epsilon N$.

Los tiempos se han calculado teniendo en cuenta el tiempo de dibujar la gráfica de cada ejecución para resaltar más la diferencia entre la iteración y la vectorización.

Debido a que para una cantidad de puntos pequeña la ejecución es muy rápida, la diferencia de tiempos apenas es perceptible, sin embargo cuando nos acercamos a números muy grandes (del orden de 10^5) la diferencia es más notoria.



Como se puede apreciar en la gráfica a partir de una cantidad de puntos superior a 10^6 la diferencia empieza a ser perceptible (en las pruebas el iterativo rondaba los 5 segundos mientras que el vectorizado rondaba los 3.5) y se dispara exponencialmente, llegando a rozar los 500 segundos la función iterativa en el caso de los 10^8 puntos, en cambio la función vectorizada apenas llega a los 300 segundos.

A partir de esta gráfica podemos dar por hecho que ambas funciones son aptas si vamos a trabajar con un número de puntos pequeño, pero si queremos hacerlo con números muy grandes la función que utiliza vectorización es claramente superior si queremos ahorrar una gran cantidad de tiempo.