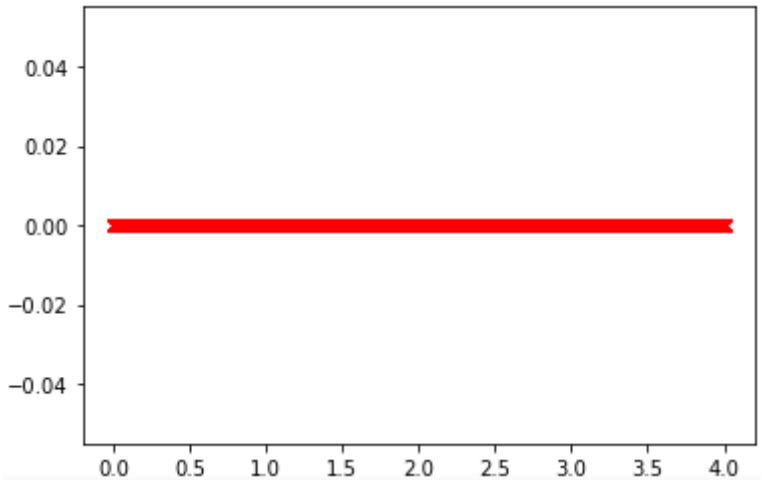
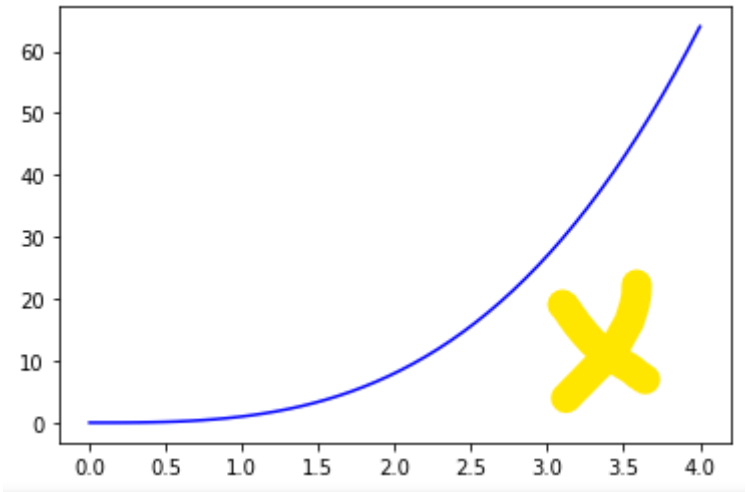
Práctica 0 (Vectorización)

¿Cómo he planteado la práctica?

Primero he creado un vector de puntos aleatorios en el eje X, estos puntos van desde A hasta B



Queremos calcular la integral con los números que tenemos ***debajo*** de nuestra ***función***

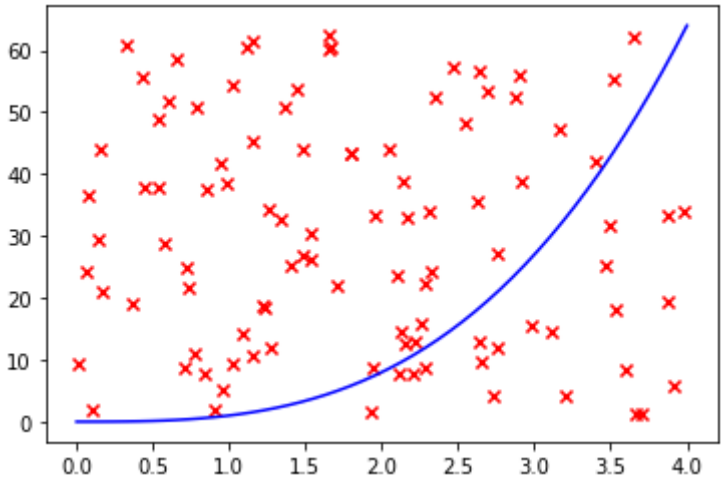


Así que la ecuación dada para aproximarnos a la integral es:

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

Ahora obtenemos M, que es el máximo valor de la función en cualquier valor de X que hemos generado aleatoriamente.

Posteriormente, añadimos a nuestro vector el eje Y de la misma forma, aunque ahora los límites son de 0 a M.



A partir de aquí vamos a hacerlo de dos formas:

- **Bucles**: Recorremos todos los elementos del eje X y les aplicamos la función, este resultado lo comparamos con el punto aleatorio generado correspondiente en el eje Y, si es menor, está dentro del área y por lo tanto sumamos 1, si es mayor, no hacemos nada

- **Vectorización**: Comparamos directamente los resultados, de forma que nos devuelva un vector de booleanos, a este vector le aplicamos un sumatorio.

Tras ello, en ambas formas, aplicamos la fórmula que nos permite aproximar la integral.

NOTA: Todo ello se podría usar en una única función para evitar repetir código, pero he preferido hacerlo en dos funciones distintas en pro de un aprendizaje académico.

Código comentado y gráficas

1. Importar las librerías

```
import time
import numpy as np
import matplotlib.pyplot as plt
from scipy import integrate
```

2. Crear funciones (Podría implementarse con funciones lambda, he preferido implementarlo en otras funciones con fines didácticos)

```
def lineal(x):
    return x

def cuadrado(x):
    return x ** 2

def cubo(x):
    return x ** 3
```

3. Función de bucles y función vectorizada

```
def integra_mc_bucles(fun, a, b, num_puntos = 10000):

    # COMIENZA EL TIEMPO
    tic = time.process_time()
    n_debajo = 0

    # GENERAMOS EL EJE X DE LOS PUNTOS
    puntos = np.random.uniform(a, b, num_puntos)

    # BUSCAMOS EL MAXIMO M DEL EJE X
    M = np.max(fun(puntos))

    #linea = np.linspace(a, b, 50)
    #plt.plot(linea, fun(linea), c = 'blue')

    # ANIADIMOS EL EJE 'Y' AL ARRAY
    puntos = np.array((puntos, np.random.uniform(0, M, num_puntos)))

    #plt.scatter(puntos[0], puntos[1], c = 'red', marker='x')


    # RECORREMOS CON UN BUCLE EL EJE 'Y' DE LOS PUNTOS
    #  Y COMPROBAMOS SI ESTAN POR DEBAJO DE LA FUNCION
    for i in range(num_puntos):
        if puntos[1, i] < fun(puntos[0, i]):
            n_debajo += 1

    # APLICAMOS LA FORMULA
    I = (n_debajo / num_puntos) * (b - a) * M

    #TERMINA EL TIEMPO
    toc = time.process_time()

    print("El método propio de bucles ha calculado:", I)

    return((toc - tic) * 1000)
```

```python
def integra_mc_vectorizado(fun, a, b, num_puntos=10000):

    # COMIENZA EL TIEMPO
    tic = time.process_time()
    n_debajo = 0

    # GENERAMOS EL EJE X DE LOS PUNTOS
    puntos = np.random.uniform(a, b, num_puntos)

    # BUSCAMOS EL MAXIMO M DEL EJE X
    M = np.max(fun(puntos))

    # ANIADIMOS EL EJE 'Y' AL ARRAY
    puntos = np.array((puntos, np.random.uniform(0, M, num_puntos)))

    # COMPARAMOS SI VECTOR 'Y' ESTA POR DEBAJO DE LA FUNCION,
    # DEVOLVEMOS LOS BOOLEANOS Y LOS CONTAMOS, TODO ELLO VECTORIZADO
    n_debajo = np.sum(puntos[1, :] < fun(puntos[0, :]))

    # APLICAMOS LA FORMULA
    I = (n_debajo / num_puntos) * (b - a) * M

    #TERMINA EL TIEMPO
    toc = time.process_time()

    print("El método propio de la vectorización ha calculado:", I)

    return((toc - tic) * 1000)
```

4. Ahora usamos las dos funciones anteriores y comparamos el resultado con la librería scipy. Como es lógico el tiempo de bucles es superior al tiempo de vectorización.

```python
print("El tiempo de los bucles es: {} milisegundos\n".format(integra_mc_bucles(cubo, 0, 4, 100000)))
print("El tiempo de la vectorización es: {} milisegundos\n".format(integra_mc_vectorizado(cubo, 0, 4, 100000)))
print("El método de scipy ha calculado: {}".format(integrate.quad(cubo, 0, 4)[0]))
```

```
El método propio de bucles ha calculado: 63.950704125921426
El tiempo de los bucles es: 125.0 milisegundos

El método propio de la vectorización ha calculado: 63.34637155339504
El tiempo de la vectorización es: 15.625 milisegundos

El método de scipy ha calculado: 64.0
```

NOTA: Los resultados de ambos métodos propios difieren, esto es lógico porque en ambas funciones se crean puntos aleatorios, por lo que son distintos, aunque la diferencia es relativamente despreciable.

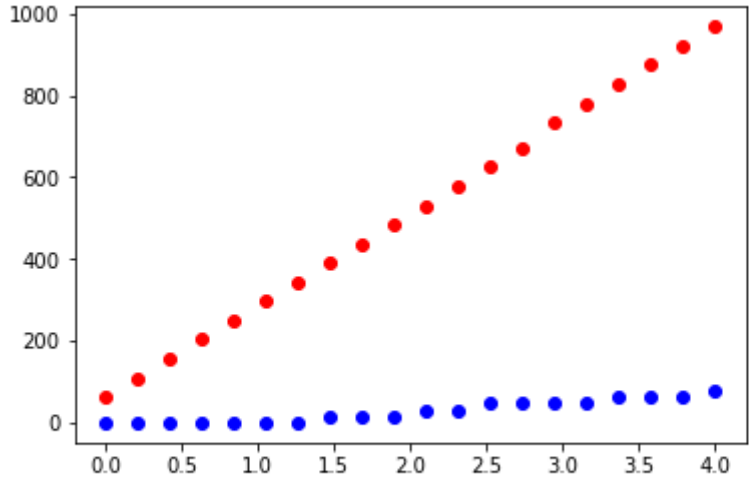
5. Ahora solo queda comparar los tiempos, en este caso, a través de una gráfica y numéricamente, ¿cuántas veces más es eficiente?

```
def compara_tiempos(fun, a, b, num_puntos=10000):

    # Tiempos acumulados
    acumulado_bucles = 0
    acumulado_vect = 0

    # Veces a recorrer el bucle
    sizes = np.linspace(a, b, 20)

    plt.figure()

    # Sumamos los tiempos (No los metemos en un array)
    # Dibujamos el punto en la grafica
    for i in sizes:
        acumulado_bucles += integra_mc_bucles(fun, a, b, num_puntos)
        plt.plot(i, acumulado_bucles, c = 'red', marker='o')

        acumulado_vect += integra_mc_vectorizado(fun, a, b, num_puntos)
        plt.scatter(i, acumulado_vect, c = 'blue', marker='o')


    # Mostramos la grafica
    plt.show()

    return (acumulado_bucles / acumulado_vect)
```

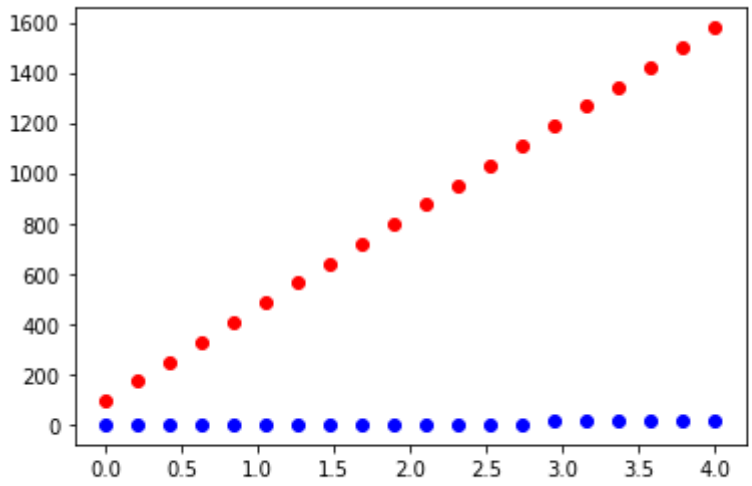
6. Y llamar a la función anterior, en este caso, como habíamos creado 3 funciones, vamos a probarlas.

```
print("La vectorización en la función lineal es: x{:.2f} veces más eficiente\n\n".format(compara_tiempos(lineal, 0, 4, 100000)))
print("La vectorización en la función cuadrado es: x{:.2f} veces más eficiente\n\n".format(compara_tiempos(cuadrado, 0, 4, 100000
print("La vectorización en la función cubo es: x{:.2f} veces más eficiente\n\n".format(compara_tiempos(cubo, 0, 4, 100000)))
```

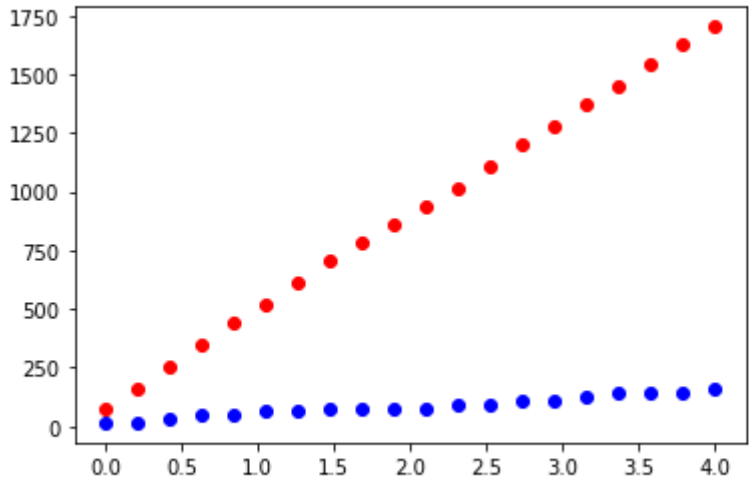
7. Resultados




```
La vectorización en la función lineal es: x12.40 veces más eficiente
```




```
La vectorización en la función cuadrado es: x101.00 veces más eficiente
```




```
La vectorización en la función cubo es: x10.90 veces más eficiente
```