Universidad Complutense de Madrid

IAAC - PRACTICA 0

Yaco Alejandro Santiago Pérez

Asignatura: INTELIGENCIA ARTIFICIAL APLICADA A INTERNET DE LAS COSAS

-

Master IOT

9 de febrero de 2020

Índice general

Capítulo 1

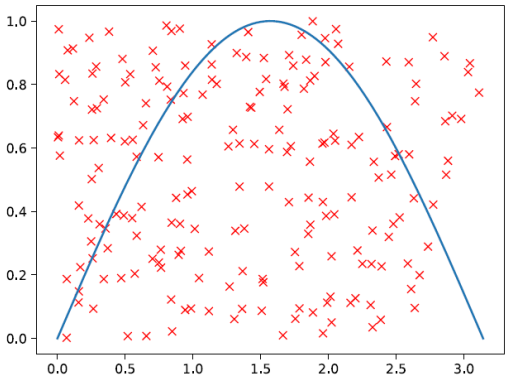
Introducción

Esta práctica consiste en la utilización del ***Método de Monte Carlo*** para la obtención del area bajo la curva de una función, con la ayuda de la biblioteca **numpy**[1]

Este método consiste en generar aleatoriamente puntos (en rojo en la figura) dentro de ese rectángulo y aproximar el valor de la integral por el porcentaje de puntos que caen por debajo de la función en cuestión.

Integración por el método de Monte Carlo

$$I = \int_a^b f(x)dx = F(b) - F(a) \qquad I \approx \frac{N_{debajo}}{N_{total}}(b-a)M$$



Ndebajo es el número de puntos *(x; y)* generados aleatoriamente cuya coordenada y es menor que el valor de la función *f(x)* para ese valor de x y *Ntotal* es el número total de puntos generados aleatoriamente dentro del rectángulo.

[1] https://numpy.org/

Capítulo 2

Objetivos

A lo largo de esta práctica, se pretende alcanzar los siguientes objetivos:

- Implementación de la función `integra_mc(fun, a, b, num_puntos=10000)`
- Calcular el área mediante el método de *Monte Carlo*.
- Realizarlo mediante operaciones. vectoriales, y mediante la utilización de bucles
- Realizar una comparación del tiempo que consumen cada uno de estos métodos.

Capítulo 3

Explicación del código

3.1. Funciones

Las funciones empleadas, en orden de llamada, son las siguientes:

- **funcion(x):** Es la función que recibe la coordenada X y devuelve la coordenada Y tras la transformación pertinente.
  En mi caso, en ella he definido una parábola.

- **main():** Se definen las variables necesarias para el calculo: *punto a*, *punto b*, y *número de puntos*, tras ello hace la llamada a la siguiente función.

- **compara_tiempos(num_puntos,a,b):** El objetivo de esta función es realizar **20** ejecuciones del método desarrollado para el calculo del área.
  Este método generará **20** números aleatorios entre 100 y el numero de puntos definido.
  A continuación se procederá a realizar las llamadas al método desarrollado **integra_mc()**.

  Con los datos obtenidos en las llamadas, generará una *gráfica* comparando el tiempo de ejecución en función de si se utilizan **operaciones vectoriales** o **bucles**. Esta gráfica se almacena en *times.png*.

- **integra_mc(fun, a, b, num_puntos=10000):** Este método únicamente se ocupará de realizar dos llamadas a integra_mc_mode() y devolver sus respuestas en un array de dos elementos.

- **integra_mc_mode(fun, a, b, mode, num_puntos=10000):** Este es el método más importante, donde se realizan la operaciones con el fin de calcular la integral.
  Su explicación paso a paso se detalla en la **sección 3.2**.

  En esta función se diferencia la manera de realizar operaciones sobre los *arrays* de coordenadas. En función del valor de la variable **mode** se va a realizar una ejecución con operaciones **vectoriales***(mode = 0)*, o mediante **bucles***(mode = 1)*.

  Finalmente, se llama al método **pintaFun()** que generará una gráfica.

- **pintaFun(puntosX, puntosY, x, y, encima, debajo, mode, num_puntos):** Este método plasmará con la ayuda de la biblioteca *matplotlib.pyplot* [1] los datos sobre una gráfica.
  Almacenará en un archivo *.png* dentro del directorio */result/* los 40 archivos generados.

3.2. Método principal: integra_mc_mode()

En este método se generan los puntos X de las coordenadas de la función (*puntosX)*, a continuación, se obtienen las coordenadas Y (*puntosY* de los puntos en base a la función (definida en *fun(x))*.
Con estos puntos ya tenemos la representación sobre el plano de la función definida.

Ahora, entre el 0 y el punto máximo de la función en el intervalo *[a;b]*, se van a generar coordenadas x e y aleatorias.

Por ultimo, se comprueba cuales son los puntos generados aleatoriamente que están por debajo de la función, es decir, qué puntos tienen sus coordenadas Y por debajo de *fun(X)*.

Contando el número de puntos que cumple la condición, y sabiendo el número total de puntos generados (por defecto 10.000) se calcula el *área* y el *% de área* que se encuentra por debajo de la función.

[1] https://matplotlib.org/api/pyplot_api.html

Capítulo 4

Ejecución

En este capitulo voy a presentar una ejecución del código.

Los **valores** definidos son los siguientes:

```
num_puntos=10000
a=100
b=1000
```

La función definida en **funcion(x)** es:

```
a = 1
b = 0
c = 0
return ((a * x) ** 2) + (b * x) + c
```

La cual tiene el siguiente aspecto:

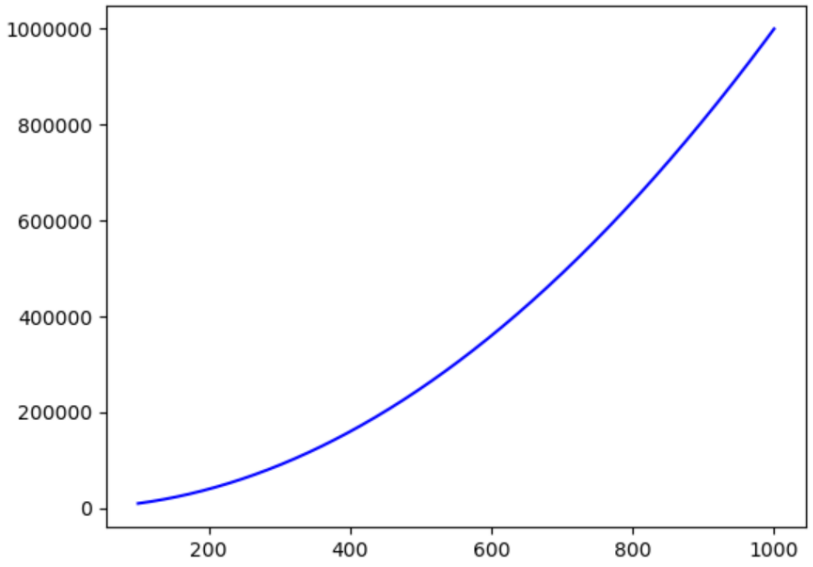


Figure 4.1: *Función del test*

4.1. Resultados

Tras realizar la ejecución, podemos observar los resultados tanto en los *logs* como en las gráficas generadas.

4.1.1. Logs

```
[MODE] -------------- VECTORIZADO [100] --------------
[INFO] Minimo: 0 | Maximo: 999980.0001009209
[INFO](Numero de puntos)  | Debajo: 36 | Encima: 64 | Total: 100 |
[SOLUCION] Area por debajo: 36.0%
[SOLUCION] Area por debajo: 323993520.0326984 unidades
[TIEMPO]: 0.0
[MODE] ---------------- BUCLES [100] ----------------
[INFO] Minimo: 0 | Maximo: 999980.0001009209
[INFO](Numero de puntos)  | Debajo: 30 | Encima: 70 | Total: 100 |
[SOLUCION] Area por debajo: 30.0%
[SOLUCION] Area por debajo: 269994600.0272486 unidades
[TIEMPO]: 140.625
```

Figure 4.2: *Primera ejecución de la batería de pruebas. Puntos generados: 100*

```
[MODE] -------------- VECTORIZADO [10000] --------------
[INFO] Minimo: 0 | Maximo: 999980.0001009209
[INFO](Numero de puntos)  | Debajo: 3696 | Encima: 6304 | Total: 10000 |
[SOLUCION] Area por debajo: 36.96%
[SOLUCION] Area por debajo: 332633347.23357034 unidades
[TIEMPO]: 0.0
[MODE] ---------------- BUCLES [10000] ----------------
[INFO] Minimo: 0 | Maximo: 999980.0001009209
[INFO](Numero de puntos)  | Debajo: 3657 | Encima: 6343 | Total: 10000 |
[SOLUCION] Area por debajo: 36.57%
[SOLUCION] Area por debajo: 329123417.4332161 unidades
[TIEMPO]: 171.875
```

Figure 4.3: *Última ejecución de la batería de pruebas. Puntos generados: 10000*

Esta ejecución generó **40 gráficas** de puntos, 20 hechas por *bucles* y 20 hechas por *operaciones vectoriales*.

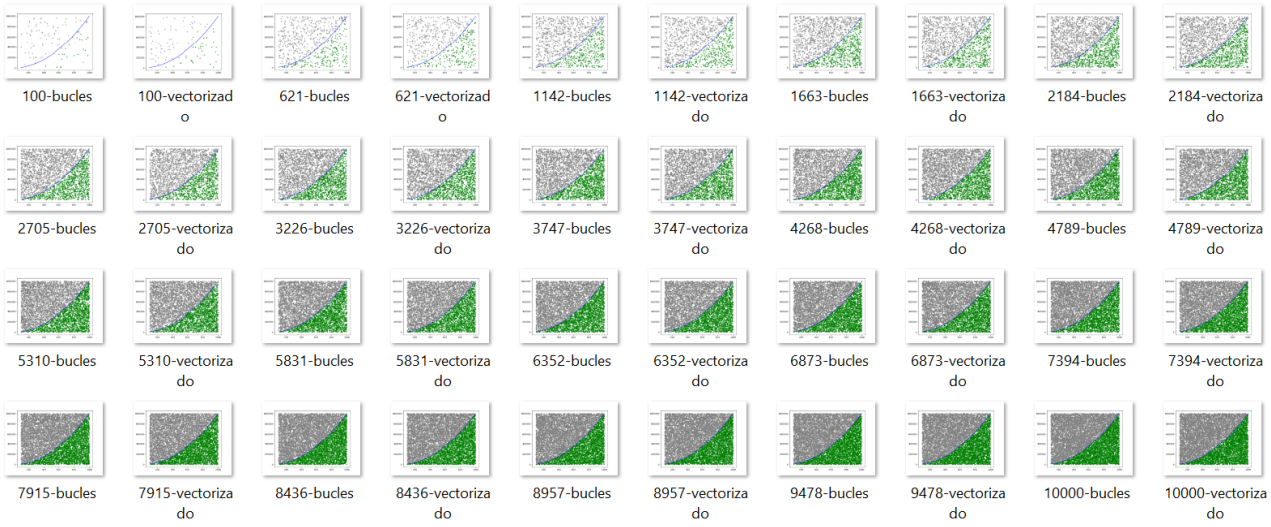


Figure 4.4: *Gráficas de las ejecuciones*

4.1.2. Gráficas

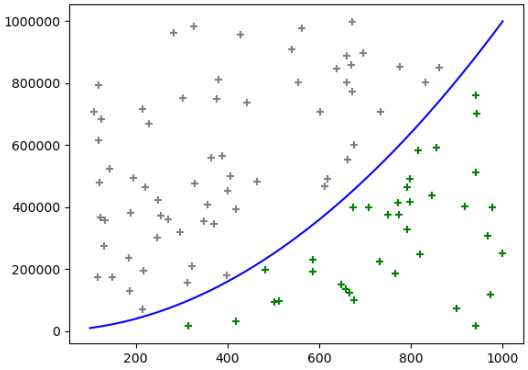


Figure 4.5: *Primera ejecución de la batería de pruebas. Puntos generados: 100*

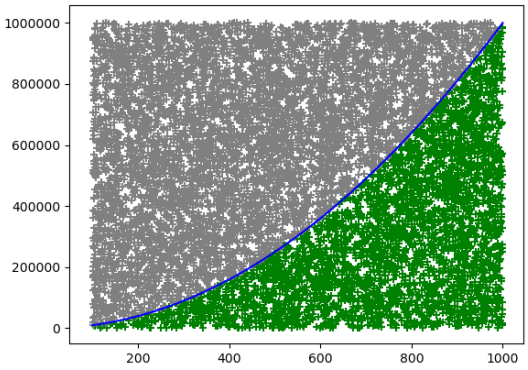


Figure 4.6: *Última ejecución de la batería de pruebas. Puntos generados: 10000*

Los **tiempos de ejecución** son los siguientes:

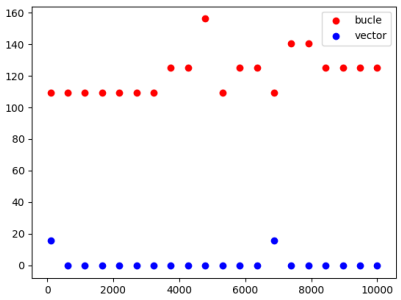


Figure 4.7: *Gráfica de tiempos de las ejecución*

Capítulo 5

Conclusiones

Como conclusiones de este desarrollo, puedo afirmar lo siguiente:

5.1. La eficiencia de las operaciones vectoriales

Aplicando operaciones directamente sobre *arrays* y obteniendo *arrays*.
Es mucho **más eficiente** que la iteración de dichos *arrays* y la operación elemento a elemento, como salta a la vista en la **figura 6.3**.

5.2. Más puntos equivale a mayor precisión

Mientras mayor sea el número de puntos generados para el calculo de la integral con el método de **Monte Carlo**, mayor será la precisión del resultado.

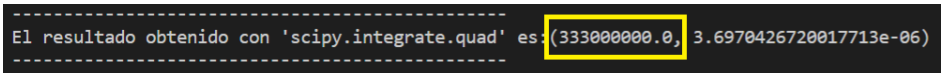


Figure 5.1: *Valor de la integral obtenido con scipy.integrate.quad*

Haciendo referencia a la primera y la última ejecución, podemos observar como es **más acertada** la última ejecución con 10000 puntos generados.

Figure 5.2: *Valor de la integral obtenido 100 puntos*

Figure 5.3: *Valor de la integral obtenido 10000 puntos*

Capítulo 6

Ejecución adicional

Con el fin de aportar más datos que confirmasen mis conclusiones, realicé otra ejecución, de la cual voy a presentar los resultados.
En este caso la ejecución se realizó con **100.000 puntos**.

6.1. Resultados

6.1.1. Logs

Como se puede observar en los ovalos verde, se realiza con **100.000** puntos.

En los rectangulos amarillos se puede observar que la solución en comparación a la de **scipy.integrate.quad** es mucho más acertada.

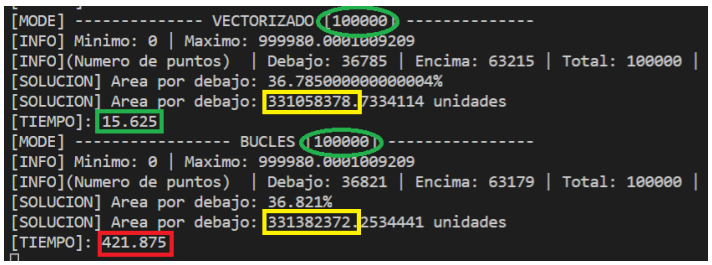


Figure 6.1: *Logs obtenidos con 100000 puntos*

6.1.2. Gráficas

Esa mayor aproximación se plasma en la siguiente gráfica donde prácticamente toda la superficie del rectángulo está poblada de puntos.

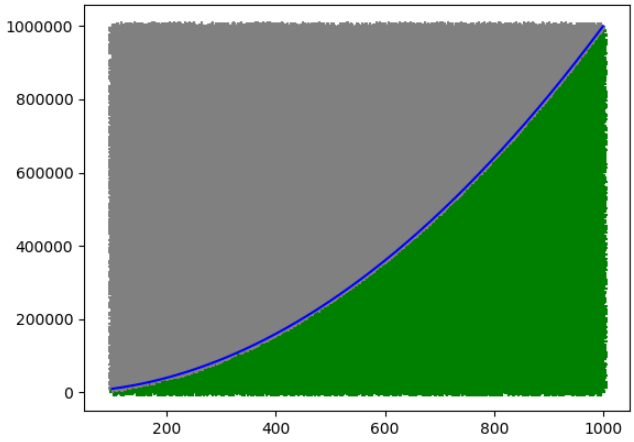


Figure 6.2: *Última ejecución de la batería de pruebas. Puntos generados: 100000*

Y por ultimo, respecto a la eficiencia, los **tiempos de ejecución** en el caso de las operaciones en bucle se disparan mientras que las vectoriales apenas han aumentado.

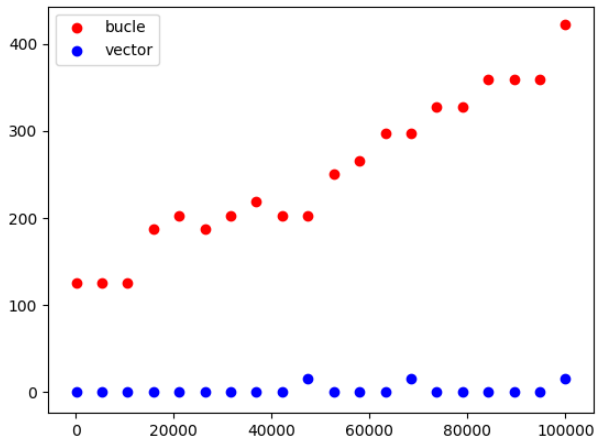


Figure 6.3: *Gráfica de tiempos de las ejecución*