# P0: Python - Memoria

En esta práctica se ha implementado un algoritmo de obtención de integrales entre dos puntos **(a, b)** de una función **f(x)**. Para ello, se ha aplicado el método de Montecarlo, consistiendo este en el posicionamiento aleatorio de puntos en la región entre *(a, b)* y *(0, máx f(x))*, y obteniendo cuántos de estos están presente en el área inferior delimitada por la función (aplicando después la fórmula proporcionada en el enunciado con este dato).

Así, nuestra función **integrate_mc** recibe los siguientes argumentos:

- **funct:** función a integrar.
- **a:** punto (en 'x') de comienzo de la región de la función a integrar.
- **b:** punto (en 'x') de final de la región de la función a integrar.
- **num_points:** número de puntos creados para aplicar la aproximación por Montecarlo.
- **vec_mode:** selector de implementación para aplicar los puntos (iterativa o vectorial).

## Cálculo Máximo Función

Con carácter general, independientemente de la implementación seleccionada, se generan 1000 puntos de **funct** para calcular por fuerza bruta el máximo valor en el rango (a, b). Tras seleccionar el máximo de entre estos puntos, se ejecuta la función seleccionada.

```
def integrate_mc(funct, a, b, num_points=10000, vec_mode=False):
    # empty function array
    function_dots = np.empty(shape=1000)

    # function max value brute force
    brute_force_values = 1000
    step = (b - a) / brute_force_values
    for i in range(0, brute_force_values):
        function_dots[i] = funct(a + step * i)

    # obtain max value
    max_value = np.max(function_dots)

    # montecarlo integrate result
    if vec_mode == False:
        return iterative_mode(funct, a, b, num_points, max_value)
    return vector_mode(funct, a, b, num_points, max_value)
```

## Implementación Iterativa

En la versión iterativa, se crea un array de dos filas, representando la primera la ‘x’ y la segunda la ‘y’ del punto aleatorio ‘i’.

Tras ello, se recorre en un bucle **for** todos los puntos a crear, manteniendo los márgenes establecidos (‘x’ entre a y b; ‘y’ entre 0 y max_value); y, en cada vuelta, si la imagen de la función en ‘x’ es mayor o igual a la ‘y’ aleatoria, se aumenta el contador.

Por último, con el número de datos obtenidos, se aplica la fórmula proporcionada y se obtiene la integral aproximada.

```
# montecarlo integration using iteration
def iterative_mode(funct, a, b, num_points, max_value):
    # first row the 'x' axis, second row the 'y' axis
    montecarlo_dots = np.empty(shape=(num_points, 2))
    count = 0

    for i in range(0, num_points):
        # we obtain the random point in both axis
        montecarlo_dots[i, 0] = np.random.uniform(a, b)
        montecarlo_dots[i, 1] = np.random.uniform(0, max_value)

        # if lower than the function 'y' axis value, add to the counter
        if(funct(montecarlo_dots[i, 0]) >= montecarlo_dots[i, 1]):
            count += 1

    # return the integrate value
    return (count / num_points) * ((b - a) * max_value)
```

## Implementación Vectorial

En la versión vectorial, en primer lugar se crean los puntos aleatorios, con dos arrays representado la ‘x’ y la ‘y’ del punto ‘i’. Para ello, se aprovecha el método proporcionado por la librería *numpy*: **numpy.random.uniform(min, max, num_points)**.

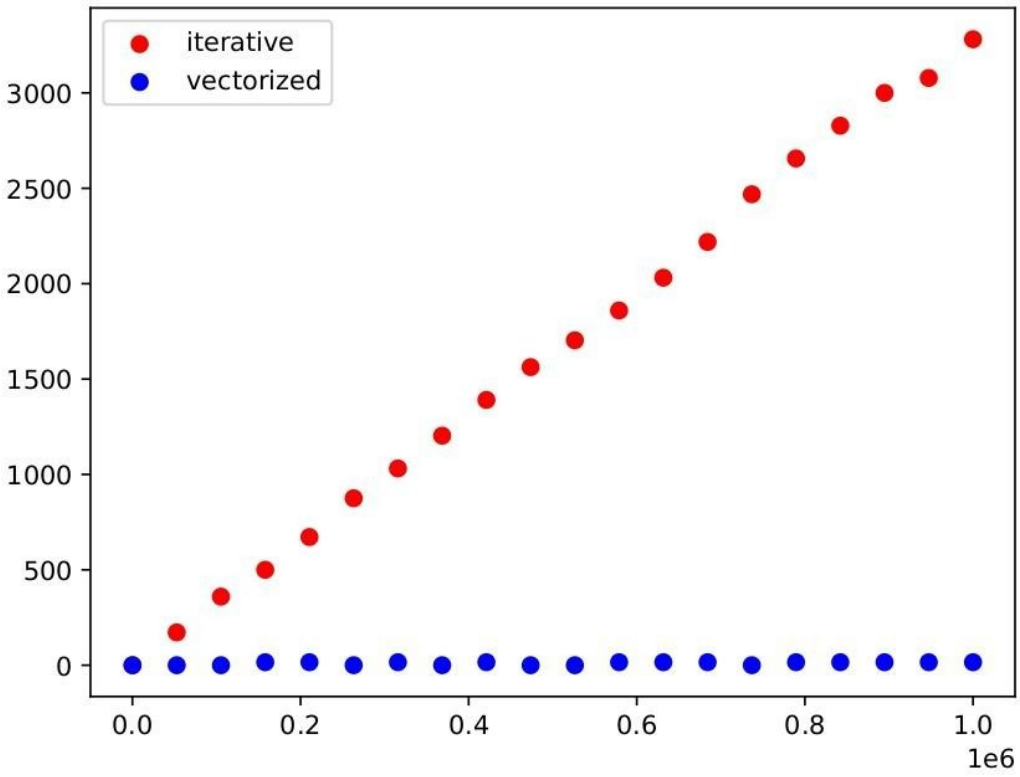
Tras ello, se obtiene la imagen de la función de todos los puntos pasando el array de ‘x’ aleatorias; y, tras ello, con la función **numpy.sum()**, obtenemos el número de puntos que están por debajo de la imagen.

```python
# montecarlo integration using vectorization
def vector_mode(funct, a, b, num_points, max_value):
    # result values from the given function
    function_dots_y = funct(np.random.uniform(a, b, num_points))
    # random values with function max cap
    montecarlo_dots_y = np.random.uniform(0, max_value, num_points)
    # number of dots below function
    count = np.sum(montecarlo_dots_y < function_dots_y)

    # return the integrate value
    return (count / num_points) * ((b - a) * max_value)
```

## Diferencia Tiempos Implementaciones

Como se solicita en el enunciado, como punto final se comparan los tiempos de cómputo de ambas representaciones. Los resultados se ven reflejados en esta gráfica, que muestra de manera clara cómo la implementación vectorial es claramente **más eficiente** que la iterativa.



```python
# auxiliary function to obtain execution time
def obtain_time(funct, a, b, num_points=10000, vec_mode=False):
    tic = time.process_time()
    integrate_mc(funct, a, b, num_points, vec_mode)
    toc = time.process_time()
    return 1000 * (toc - tic)

# function to test speed difference between both types of integration
def time_test(funct, a, b):
    # different sizes to test speed
    test_sizes = np.linspace(100, 1000000, 20)
    it_times = []
    vec_times = []

    # obtain speed for each size used
    for size in test_sizes:
        it_times += [obtain_time(funct, a, b, int(size), False)]
        vec_times += [obtain_time(funct, a, b, int(size), True)]

    # print results
    plt.figure()
    plt.scatter(test_sizes, it_times, c='red', label='iterative')
    plt.scatter(test_sizes, vec_times, c='blue', label='vectorized')
    plt.legend()
    plt.savefig('graph.pdf')
```