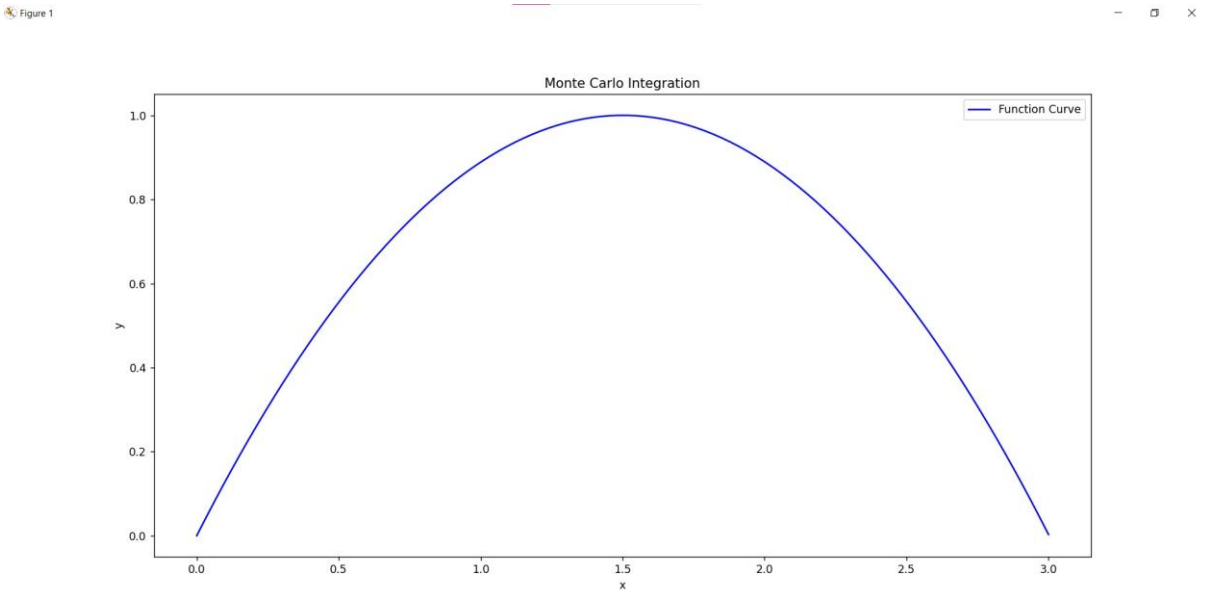
Machine Learning and Big Data – Assignment 1

Arda Harman

I will briefly explain about all the parts at my assignment and sections of the codes at my assignment

1st Part: How am I solving the problem “finding integral” with Monte Carlo



With Monte Carlo, we take random samples from the integral by randomly generating points

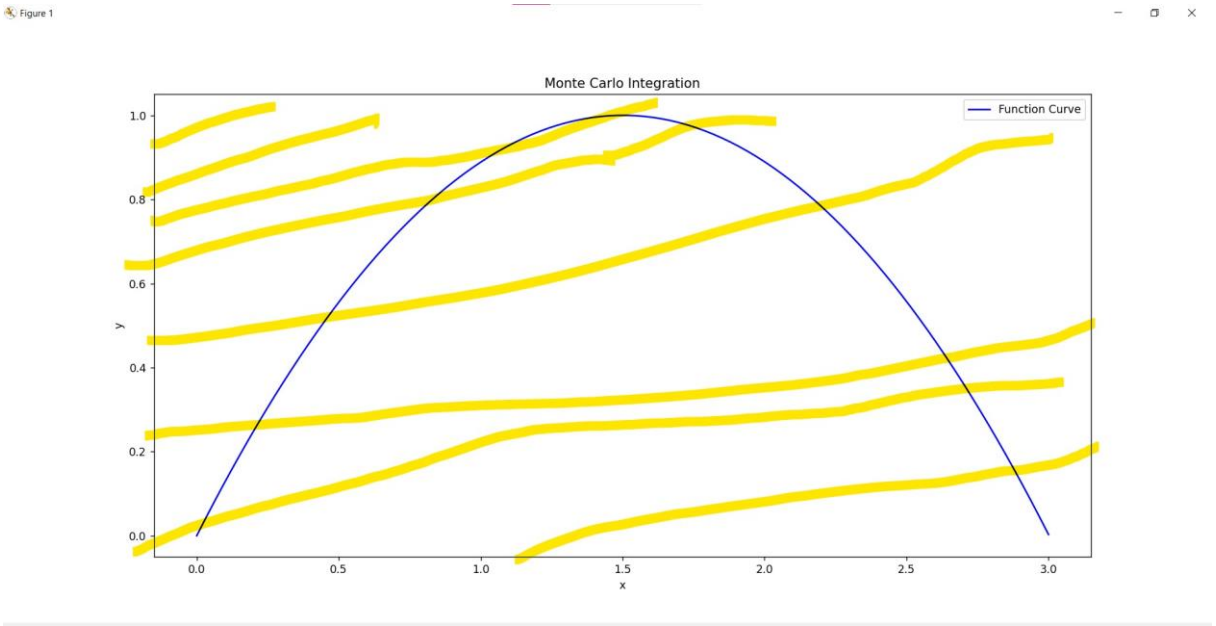
If a point remains inside the integral, we increase the amount of points which are under the area

By measuring how many samples are inside the integral, we can find the integral

We need to find all the necessary data at this formula to be able to calculate “integral”

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

I first calculated the area of the whole shape

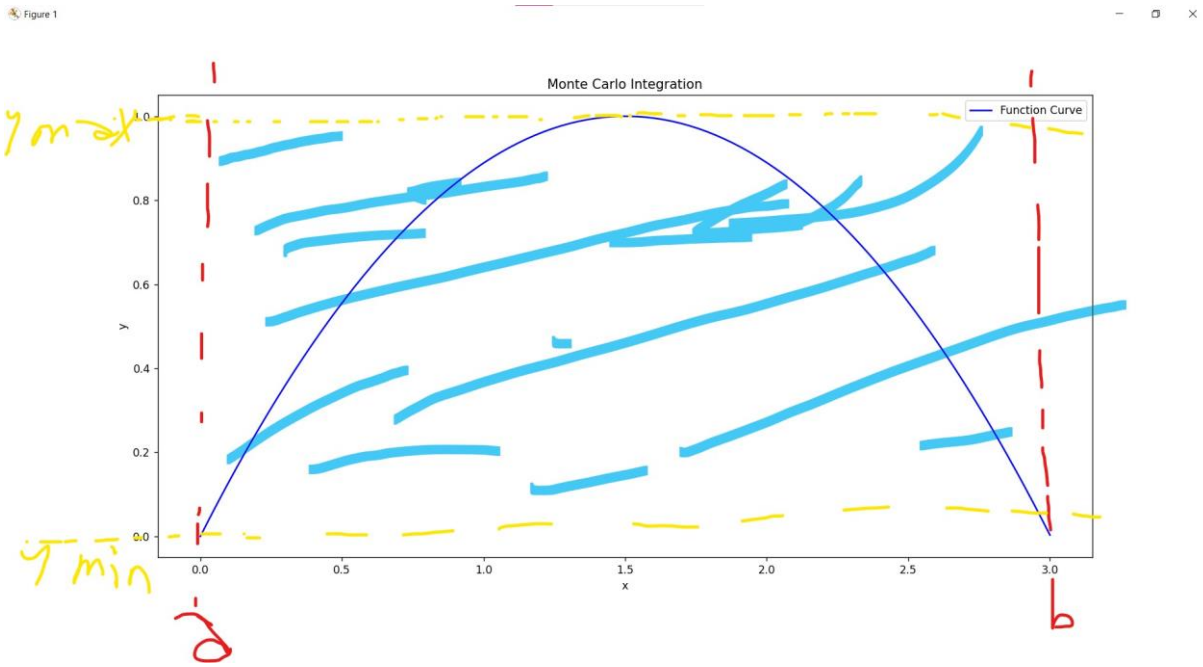


To do that, I need to find the values of the sides of the shape

a and b is already given… so that side's length is just (b-a)

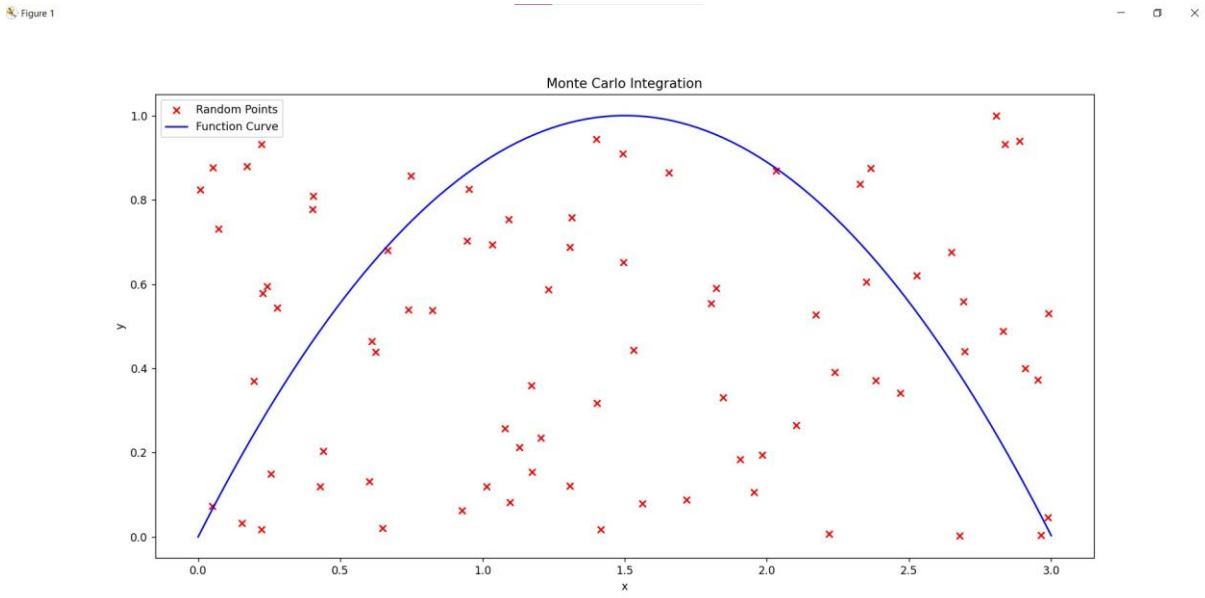
y_max and y_min should be found to calculate that side

I used every value between a and b to find the y_max and y_min



After that, I did random sampling and find the ratio of

"amount of samples inside of integral / amount of all samples"



2nd part: How am I using “iterative” and “vectorized” approaches

Through my assignment, I solved the same problem with 2 different approaches:

“Iterative” and “Vectorized”

- Iterative approach
  - Every step is done with a for loop
    - Operations are done element by element
- Vectorized approach
  - Every step is done with vectorization
    - Operations have used the list itself

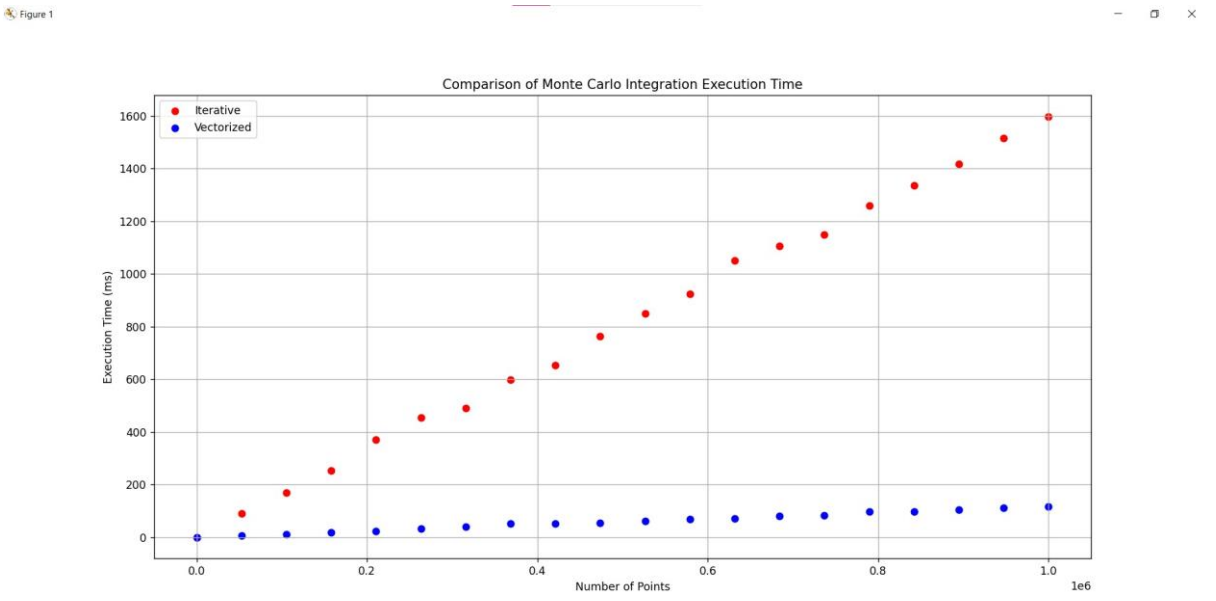
ITERATIVE

```python
def integral_iterative(fun, a, b, num_puntos=10000):
    # determine area
    x_values = np.zeros(num_puntos)
    space_amount = float(b - a) / num_puntos

    for i in range(0, num_puntos):
        x_values[i] = a + i*space_amount

    max_y = f(x_values[0])

    for x in x_values:
        if f(x) > max_y:
            max_y = f(x)

    min_y = f(x_values[0])

    for x in x_values:
        if f(x) < min_y:
            min_y = f(x)

    # determine distribution
    sample_x_values = np.zeros(num_puntos)

    for i in range(num_puntos):
        number = random.uniform(a, b)
        sample_x_values[i] = number


    sample_y_values = np.zeros(num_puntos)

    for i in range(num_puntos):
        number = random.uniform(min_y, max_y)
        sample_y_values[i] = number

    # creating distribution
    no_of_under_integral = 0
    for i in range(num_puntos):
        if sample_y_values[i] < f(sample_x_values[i]) :
            no_of_under_integral = no_of_under_integral + 1

    integral = float((no_of_under_integral / num_puntos)) * (b - a) * (max_y - min_y)

    return integral
```

VECTORIZED

```python
def integral_vectorized(fun, a, b, num_puntos=10000):
    # determine area
    x_values = np.linspace(a, b, num_puntos)

    max_y = max(f(x_values))
    min_y = min(f(x_values))

    # determine distribution
    sample_x_values = np.random.uniform(a, b, num_puntos)
    sample_y_values = np.random.uniform(min_y, max_y, num_puntos)

    # creating distribution
    no_of_under_integral = np.sum(sample_y_values < f(sample_x_values))

    integral = float((no_of_under_integral / num_puntos)) * (b - a) * (max_y - min_y)

    return integral
```

RESULTS THAT I GOT




```
PS C:\Users\ardah\UCM> python -u "c:\Users\ardah\UCM\Mach_Learn&Big_Data\assignment1_reel.py
Integral (Iterative): 1.9946479473802918, Time: 197.68762588500977 ms
Integral (Vectorized): 2.0065639092556498, Time: 10.462045669555664 ms
Integral (Quad): (2.0024999999999995, 2.2232216068118753e-14)
PS C:\Users\ardah\UCM>
```

My whole code

```python
import time
import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import quad
import random
# Define the function
def f(x):
    return -0.444*x*x + 1.333*x

def integral_iterative(fun, a, b, num_puntos=10000):
    # determine area
    x_values = np.zeros(num_puntos)
    space_amount = float(b - a) / num_puntos

    for i in range(0, num_puntos):
        x_values[i] = a + i*space_amount

    max_y = f(x_values[0])

    for x in x_values:
        if f(x) > max_y:
            max_y = f(x)

    min_y = f(x_values[0])

    for x in x_values:
        if f(x) < min_y:
            min_y = f(x)

    # determine distribution
    sample_x_values = np.zeros(num_puntos)

    for i in range(num_puntos):
        number = random.uniform(a, b)
        sample_x_values[i] = number


    sample_y_values = np.zeros(num_puntos)

    for i in range(num_puntos):
        number = random.uniform(min_y, max_y)
        sample_y_values[i] = number

    # creating distribution
    no_of_under_integral = 0
    for i in range(num_puntos):
        if sample_y_values[i] < f(sample_x_values[i]) :
            no_of_under_integral = no_of_under_integral + 1

    integral = float((no_of_under_integral / num_puntos)) * (b - a) * (max_y - min_y)

    return integral



def integral_vectorized(fun, a, b, num_puntos=10000):
    # determine area
    x_values = np.linspace(a, b, num_puntos)

    max_y = max(f(x_values))
    min_y = min(f(x_values))

    # determine distribution
    sample_x_values = np.random.uniform(a, b, num_puntos)
    sample_y_values = np.random.uniform(min_y, max_y, num_puntos)

    # creating distribution
    no_of_under_integral = 0
    for i in range(num_puntos):
        if sample_y_values[i] < f(sample_x_values[i]) :
            no_of_under_integral = no_of_under_integral + 1

    integral = float((no_of_under_integral / num_puntos)) * (b - a) * (max_y - min_y)

    return integral


def integral_vectorized(fun, a, b, num_puntos=10000):
    # determine area
    x_values = np.linspace(a, b, num_puntos)

    max_y = max(f(x_values))
    min_y = min(f(x_values))

    # determine distribution
    sample_x_values = np.random.uniform(a, b, num_puntos)
    sample_y_values = np.random.uniform(min_y, max_y, num_puntos)

    # creating distribution
    no_of_under_integral = np.sum(sample_y_values < f(sample_x_values))

    integral = float((no_of_under_integral / num_puntos)) * (b - a) * (max_y - min_y)

    return integral


def compara_tiempos():
    sizes = np.linspace(100, 100000, 20)
    times_iterative = []
    times_vectorized = []

    for size in sizes:
        start_time_iterative = time.time()
        integral_iterative(f, 0, 3, int(size))
        end_time_iterative = time.time()
        time_iterative = (end_time_iterative - start_time_iterative) * 1000  # Convert to milliseconds
        times_iterative.append(time_iterative)

        start_time_vectorized = time.time()
        integral_vectorized(f, 0, 3, int(size))
        end_time_vectorized = time.time()
        time_vectorized = (end_time_vectorized - start_time_vectorized) * 1000  # Convert to milliseconds
        times_vectorized.append(time_vectorized)

    plt.figure()
    plt.scatter(sizes, times_iterative, c='red', label='Iterative')
    plt.scatter(sizes, times_vectorized, c='blue', label='Vectorized')
    plt.xlabel('Number of Points')
    plt.ylabel('Execution Time (ms)')
    plt.title('Comparison of Monte Carlo Integration Execution Time')
    plt.legend()
    plt.grid(True)
    plt.show()

    print(f"Integral (Iterative): {integral_iterative(f, 0, 3, int(size))}, Time: {time_iterative} ms")
    print(f"Integral (Vectorized): {integral_vectorized(f, 0, 3, int(size))}, Time: {time_vectorized} ms")
    print(f"Integral (Quad): {quad(f, 0, 3)}")

# Call the function to compare execution times
compara_tiempos()
```