Práctica 0: Python

Hemos implementado un algoritmo en Python que calcula el valor de una integral definida por el método de Monte Carlo. Dada una función *f*, los límites de integración *a* y *b* y un número de puntos N, podemos calcular el valor de la integral del siguiente modo:

In [1]:
```python
def integrate_mc(f, a, b, N=10000):

    # Calculamos el punto máximo de la función
    x = np.linspace(a, b, N)
    y = f(x)
    M = np.amax(y)

    # Contamos el número de puntos debajo de la curva
    x_sample = np.random.uniform(a, b, N)
    y_sample = np.random.uniform(0, M, N)
    below_index = np.where(f(x_sample) >= y_sample)

    # Calculamos la integral
    rect_area = (b - a) * M
    I = (len(below_index[0])/N) * rect_area
```

Para comprobar que el valor de la integral obtenida es correcto, añadimos una instrucción para calcular el valor exacto de la integral, gracias a una función de SciPy. Luego, comparamos ambos resultados para obtener una medida del error cometido en el cálculo aproximado mediante el método de Monte Carlo:

In [2]:
```python
def integrate_mc(f, a, b, N=10000):

    # Calculamos el punto máximo de la función
    x = np.linspace(a, b, N)
    y = f(x)
    M = np.amax(y)

    # Contamos el número de puntos debajo de la curva
    x_sample = np.random.uniform(a, b, N)
    y_sample = np.random.uniform(0, M, N)
    below_index = np.where(f(x_sample) >= y_sample)

    # Calculamos la integral
    rect_area = (b - a) * M
    I = (len(below_index[0])/N) * rect_area
    i = scipy.integrate.quad(f, a, b)

    # Presentamos los resultados
    print("La integral aproximada es: {}".format(I))
    print("La integral exacta es: {}".format(i[0]))
    print("El error es: {}".format(abs(i[0] - I)))
```

Nuestra primera prueba será con la función seno. Utilizaremos la versión de NumPy, ya que la de la librería estándar Math no funciona como se espera con los arrays de NumPy; sin embargo, sí utilizaremos la librería Math para definir la constante Pi. Así, primero definiremos nuestra función:

```
In [4]: def fun(x):
            return np.sin(x)
```

Y luego la utilizaremos como parámetro en la función *integrate_mc*. Para este test utilizaremos un valor de 0 para *a* y un valor de π para *b*; para N dejaremos el valor por defecto de 10000 puntos. Los resultados son los siguientes:

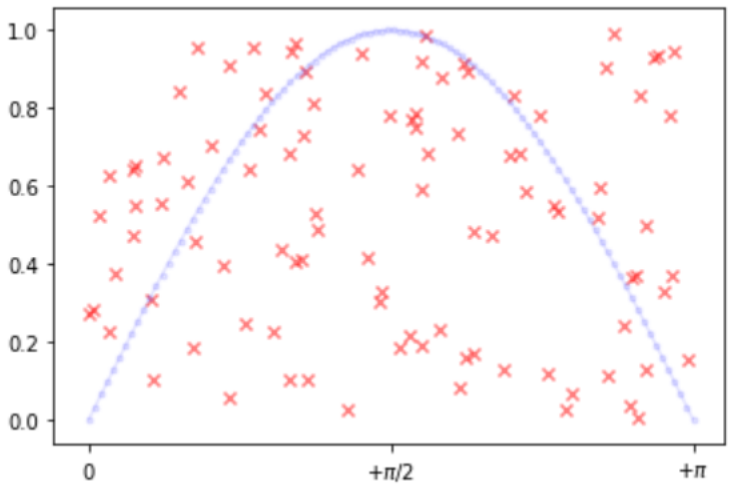
```
In [9]: integrate_mc(fun, 0, math.pi)

La integral aproximada es: 1.9883139659222329
La integral exacta es: 2.0
El error es: 0.011686034077767138
```

El cálculo es bastante exacto. Para obtener una representación visual del proceso, vamos a utilizar Matplotlib, concretamente el submódulo Pyplot:

```
plt.figure()
plt.xticks([0, np.pi/2, np.pi], [r'$0$', r'$+\pi/2$', r'$+\pi$'])
plt.plot(x, y, c='b', marker='.', alpha=0.1)
plt.scatter(x_sample, y_sample, c='r', marker='x', alpha=0.5)
plt.show()
```

En esta ocasión utilizaremos tan solo 100 puntos, para que la figura resultante sea más vistosa; el resultado no variará demasiado:

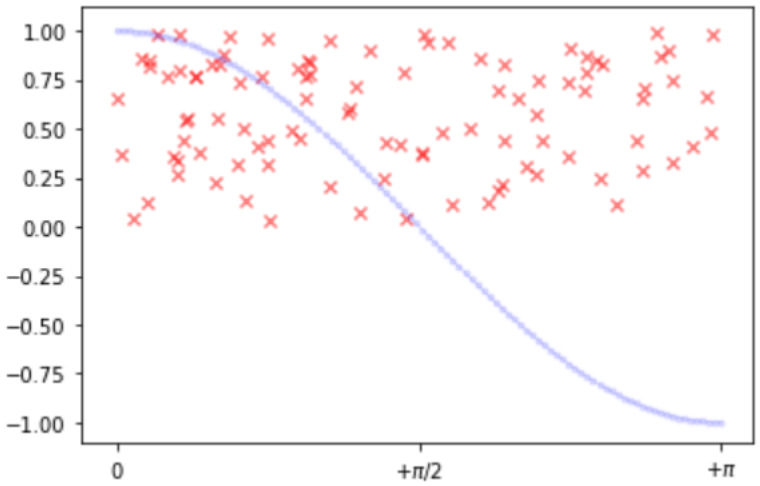
```
In [19]: integrate_mc(fun, 0, math.pi, 100)

La integral aproximada es: 1.9161303005489967
La integral exacta es: 2.0
El error es: 0.08386969945100331
```



No obstante, nuestro cálculo tiene ciertas carencias. Si en vez de la función seno utilizamos la función coseno, estos son los resultados que obtenemos:

```
In [21]: integrate_mc(fun, 0, math.pi, 100)
```

```
La integral aproximada es: 1.0995574287564276
La integral exacta es: 3.6775933888827275e-17
El error es: 1.0995574287564276
```



Hemos pasado por alto las curvas con valores negativos de y. En la figura, podemos comprobar que no se han lanzado puntos en toda el área que recorre la curva; esto se debe a que hemos asumido que el valor mínimo de la función siempre será 0. Modificando el código original de *integrate_mc* podemos corregir este defecto:

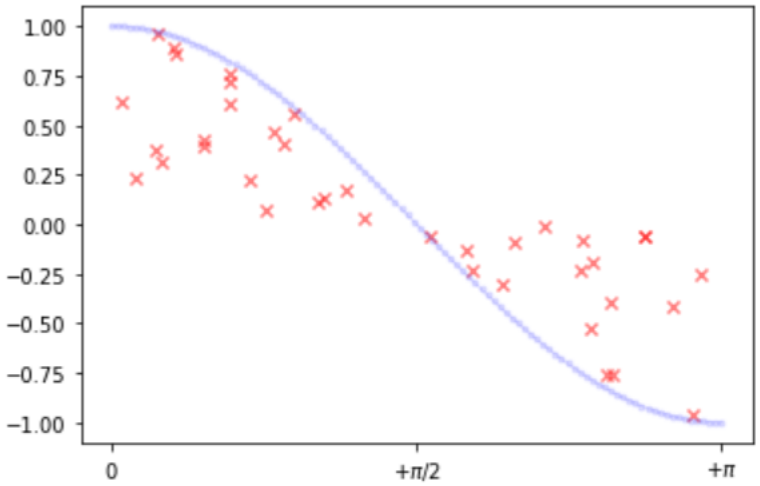
```
In [18]: def integrate_mc(f, a, b, N=10000):

             # Calculamos el punto máximo de la función
             x = np.linspace(a, b, N)
             y = f(x)
             M = np.amax(y)
             m = np.amin(y)

             # Contamos el número de puntos debajo de la curva
             x_sample = np.random.uniform(a, b, N)
             y_sample = np.random.uniform(m, M, N)
             below_index = np.where(f(x_sample) >= y_sample)

             # Calculamos la integral
             rect_area = (b - a) * (M - m)
             I = (len(below_index[0])/N) * rect_area
```

El problema ahora es que las condiciones para considerar que un punto está “bajo la curva” se han modificado. Si el punto es positivo en y, debe permanecer bajo la curva para ser contabilizado como parte del área de la integral definida; en cambio, si el punto es negativo en y, debe permanecer por encima de la curva. Volvemos a modificar nuestro código para reflejar esta nueva condición; además, en esta ocasión pintaremos sólo los puntos que forman parte del área, para comprobar que se cumplen las condiciones que exigimos:

```
below_index = np.where(
    ((f(x_sample) >= 0) & (y_sample >= 0) & (f(x_sample) >= y_sample)) |
    ((f(x_sample) < 0) & (y_sample < 0) & (f(x_sample) < y_sample)))
```

```
In [25]: integrate_mc(fun, 0, math.pi, 100)
```

```
La integral aproximada es: 2.4504422698000385
La integral exacta es: 3.6775933888827275e-17
El error es: 2.4504422698000385
```



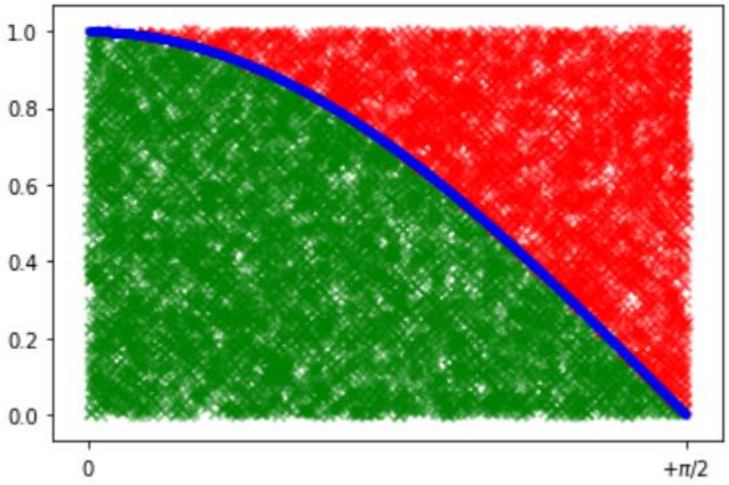
No obstante, la integral del coseno entre 0 y π es 0, puesto que la curva dibuja dos áreas, una positiva y otra negativa, pero de igual valor. Nuestra función *integrate_mc* no puede determinar si una curva va a cortar el eje de abscisas, o si aparecen áreas negativas pero, al menos, hemos podido corregir el cálculo básico de áreas:

In [24]:
```
def integrate_mc(f, a, b, N=10000):

    # Calculamos el punto máximo de la función
    x = np.linspace(a, b, N)
    y = f(x)
    M = np.amax(y)
    m = np.amin(y)

    # Contamos el número de puntos debajo de la curva
    x_sample = np.random.uniform(a, b, N)
    y_sample = np.random.uniform(m, M, N)

    below_index = np.where(
        ((f(x_sample) >= 0) & (y_sample >= 0) & (f(x_sample) >= y_sample)) |
        ((f(x_sample) < 0) & (y_sample < 0) & (f(x_sample) < y_sample)))

    above_index = np.where(
        ((f(x_sample) >= 0) & (y_sample >= 0) & (f(x_sample) < y_sample)) |
        ((f(x_sample) < 0) & (y_sample < 0) & (f(x_sample) >= y_sample)) |
        ((f(x_sample) >= 0) & (y_sample < 0)) |
        ((f(x_sample) < 0) & (y_sample >= 0)))

    # Calculamos la integral
    rect_area = (b - a) * (M - m)
    I = (len(below_index[0])/N) * rect_area
    i = scipy.integrate.quad(f, a, b)

    # Presentamos los resultados
    print("La integral aproximada es: {}".format(I))
    print("La integral exacta es: {}".format(abs(i[0])))
    print("El error es: {}".format(abs(abs(i[0]) - abs(I))))

    plt.figure()
    plt.xticks([0, np.pi/2, np.pi], [r'$0$', r'$+\pi/2$', r'$+\pi$'])
    plt.plot(x, y, c='b', marker='.', alpha=0.1)
    plt.scatter(x_sample[below_index], y_sample[below_index], c='r', marker='x', alpha=0.5)
    plt.scatter(x_sample[above_index], y_sample[above_index], c='r', marker='x', alpha=0.5)
    plt.show()
```

Por tanto, si la curva que proporcionamos a la función *integrate_mc* tiene varias partes el resultado será incorrecto; asimismo, si el área bajo la curva entre los límites de integración proporcionados es negativa, la función calculará un valor absoluto, por lo que trampearemos el resultado exacto de la integral para que de siempre un valor absoluto. Por último, para facilitar la lectura a los resultados, coloreamos los puntos del área de integración en verde, y los demás en rojo:

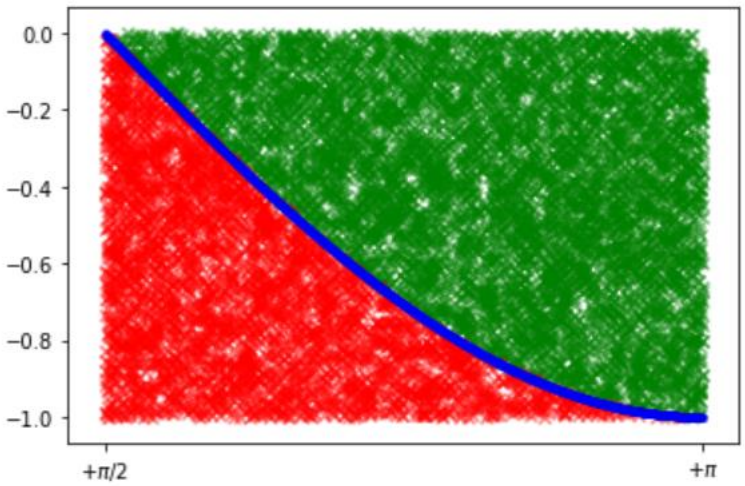
```
In [31]: integrate_mc(fun, 0, math.pi/2, 10000)
```

```
La integral aproximada es: 1.001225578699067
La integral exacta es: 0.9999999999999999
El error es: 0.0012255786990670314
```




```
In [32]: integrate_mc(fun, math.pi/2, math.pi, 10000)
```

```
La integral aproximada es: 1.004053012087298
La integral exacta es: 0.9999999999999999
El error es: 0.004053012087298025
```



A continuación se presenta una tabla con los resultados obtenidos para diferentes funciones, con tests para 100, 1000 y 10000 puntos. Añadimos además una medida de tiempo orientativa.

f	*a*	*b*	*N*	*I (MC)*	*I*	*Error*	*T (ms)*
np.sin(x)	0	π	10000	2.0008	2.0	0.0008	948.24
np.sin(x)	0	π	1000	1.9729	2.0	0.0270	345.23
np.sin(x)	0	π	100	2.0417	2.0	0.0417	250.09
np.cos(x)	0	π/2	10000	0.9983	1.0	0.0016	585.84
np.cos(x)	0	π/2	1000	0.9786	1.0	0.0213	281.11
np.cos(x)	0	π/2	100	0.9738	1.0	0.0261	232.03
$x^2 + x$	-20	20	10000	5478.37	5333.33	145.04	485.82
$x^2 + x$	-20	20	1000	5093.42	5333.33	239.90	335.41
$x^2 + x$	-20	20	100	5715.24	5333.33	381.91	231.80
np.sqrt(1 - x^2)	0	1	10000	0.7796	0.7853	0.0057	483.21
np.sqrt(1 - x^2)	0	1	1000	0.766	0.7853	0.0193	259.66
np.sqrt(1 - x^2)	0	1	100	0.84	0.7853	0.0546	223.81
np.log(x)	1	15	10000	26.834	26.62	0.2138	492.56
np.log(x)	1	15	1000	26.69	26.62	0.0697	281.88
np.log(x)	1	15	100	27.297	26.62	0.6763	231.41

Podemos observar que los tiempos de ejecución disminuyen con el número de puntos empleado para determinar la integral, como se esperaba (obviamos la anomalía de la primera ejecución). La función parábola es la que menos precisión parece ofrecer, aunque el área que calcula es bastante superior al del resto de curvas.

No obstante, el objetivo de la práctica es comparar los tiempos de ejecución de un algoritmo que utilice las funciones vectoriales de NumPy y otro que utilice un procedimiento iterativo. Para realizar esta comparativa vamos a reformular el código final de la función *integrate_mc*, devolviéndolo a su estado básico, ya que ahora sólo utilizaremos la función seno (entre 0 y π) como ejemplo:

In [3]:
```python
def integrate_mc(f, a, b, N=10000):

    tic = time.time()

    # Calculamos el punto máximo de la función
    x = np.linspace(a, b, N)
    y = f(x)
    M = np.amax(y)

    # Contamos el número de puntos debajo de la curva
    x_sample = np.random.uniform(a, b, N)
    y_sample = np.random.uniform(0, M, N)
    below_index = np.where(f(x_sample) >= y_sample)

    # Calculamos la integral
    rect_area = (b - a) * M
    I = (len(below_index[0])/N) * rect_area
    i = scipy.integrate.quad(f, a, b)

    # Presentamos los resultados
    print("La integral aproximada es: {}".format(I))
    print("La integral exacta es: {}".format(i[0]))
    print("El error es: {}".format(abs(i[0] - I)))

    toc = time.time()
    return 1000 * (toc - tic)
```

Y la versión iterativa:

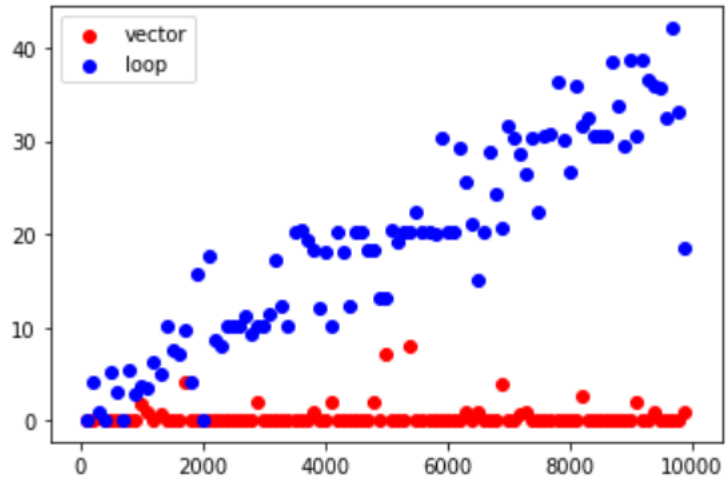
In [5]:
```python
def integrate_mc_slow(f, a, b, N=10000):

    tic = time.time()

    # Calculamos el punto máximo de la función
    x = np.linspace(a, b, N)
    y = f(x)
    M = np.amax(y)

    # Contamos el número de puntos debajo de la curva
    x_sample = np.random.uniform(a, b, N)
    y_sample = np.random.uniform(0, M, N)

    num_points_below = 0
    for i in range(N):
        if f(x_sample[i] >= y_sample[i]):
            num_points_below += 1

    # Calculamos la integral
    rect_area = (b - a) * M
    I = (num_points_below / N) * rect_area
    i = scipy.integrate.quad(f, a, b)

    # Presentamos los resultados
    print("La integral aproximada es: {}".format(I))
    print("La integral exacta es: {}".format(i[0]))
    print("El error es: {}".format(abs(i[0] - I)))

    toc = time.time()
    return 1000 * (toc - tic)
```

Además, definiremos una función para comparar los tiempos de ejecución de ambos procedimientos. Plantearemos un elevado número de casos de prueba, en los que iremos aumentando progresivamente el número de puntos N que utilizamos en la integración:

```
In [6]: def compara_tiempos():

            num_points = np.arange(100, 10000, 100)
            times_mc = []
            times_mc_slow = []

            for points in num_points:
                times_mc += [integrate_mc(f, 0, math.pi, points)]
                times_mc_slow += [integrate_mc_slow(f, 0, math.pi, points)]

            plt.figure()
            plt.scatter(num_points, times_mc, c='red', label='vector')
            plt.scatter(num_points, times_mc_slow, c='blue', label='loop')
            plt.legend()
            plt.show()
```

El resultado que obtenemos es aplastantemente obvio: la versión vectorizada se mantiene muy constante a lo largo de todos los tests, mientras que la versión iterativa crece de manera lineal.

```
In [16]: compara_tiempos()
```



**Los resultados obtenidos en las gráficas son producto de utilizar la función time.time() en vez de time.process_time().*

***Las gráficas se han obtenido desde Jupyter Notebook, dado que los resultados que ofrecían el prompt de Anaconda3 o IPython no eran satisfactorios y presentaban muchas anomalías.*