# Evaluation of given integral using Monte Carlo Sample Mean method

For the given integral——> $\int_0^1 4\sqrt{(1-x^2)}dx$

## What does the code do and what is the Sample Mean algorithm?

The code has been written keeping in mind that the no. of integral evaluations are 1000( m ) and the no. of points used to determine each integral value is also 1000 ( n )

The code uses Monte Carlo Sample Mean method to evaluate the integral mentioned above, then calculates the **standard deviation** in the value of **m integrals calculated** and the **standard deviation** in the value of **m*n function evaluations**. It then calculates the actual error by taking the difference between the evaluated value of the integral and the analytical value of the integral. The evaluated value of the integral is calculated by taking average of the m integral values evaluated. The analytical value was calculated using standard integration techniques and it turned out to be $\pi$.

The algorithm of Sample Mean method basically treats $4\sqrt{(1-x^2)}$ as a function and evaluates it's value at random point between the integral bounds, i.e., 0 to 1 in this case. To get the integral value it sums all such function evaluation values, multiplies it by the difference of the integral bounds (in this case, 1 - 0) and divides by the total number of points on which the function was evaluated.

The algorithm is of the form: $F_m = (b-a) < f > = (b-a)\frac{1}{n}\sum_{i=1}^{i=n} f(x_i)$ where $x_i$ is a random point b/w 0 and 1 for the ith summation. $F_m$ is the mth integral evaluation evaluated using n points and hence n function evaluations.

To calculate the standard deviation of the m integrals we use the formula:

$$\sigma_{integral} = \sqrt{\frac{1}{m}\sum_{\alpha=1}^{\alpha=m}(M_\alpha - M_{avg})^2}$$ where $M_a$ is the ath integral evaluation.

To calculate the standard deviation of the m*n function evaluations we use the formula:

$$\sigma_{functional} = \sqrt{\frac{1}{mn}\sum_{\alpha=1}^{\alpha=m}\sum_{i=1}^{i=n}(f_{\alpha,i} - M_{avg})^2}$$ where $f_{a,i}$ is the a*ith function evaluation, each evaluation done at a random point between 0 and 1 in our case.

Here $M_{avg} = \frac{1}{m}\sum_{\alpha=1}^{\alpha=n} M_\alpha$

**$M_{avg}$ is our integral evaluation too.**

Actual error is $|M_{avg} - \pi|$

The result of running the code once are pasted below:

```
Standard deviation for the Integral's value is  0.02797782950808661


Standard deviation for the Function's value is  0.8931737679426521


The actual error in numerical value when compared to analytical value is  0.0009084042555245198
```

We can see that $\sigma_{integral}$ is indeed almost equal to $\frac{\sigma_{functional}}{\sqrt{n}}$ which is the relation we derived theoretically during the course.

## Explanation of the code cell-wise:

### Cell #01:

The first cell just imports certain libraries that we've used in our code from time to time. The libraries imported are **numpy**, **random** and **math.** We created aliases for numpy and random namely **np** and **rd** just for the sake of a cleaner looking code.

### Cell #02:

The second cell is a function named **func( x )** that I defined. The function returns the value of the arithmetic expression $4\sqrt{1-x^2}$ for a given value of x.

## Cell #03:

This cell is where we implement our Monte Carlo algorithm as python3 code.
We start by declaring some basic parameters  as variables and assigning them values which are already known to us ( values of **m**, **n**) and initialising some other variables which we'll use in our logic(**f**, **ans[ ], integral_arr[ ]**, **f_arr[ ]**).

The **nested for loop** work such that the inner for loop iterates for 1000 times for each iteration of the outer for loop which itself iterates for 1000 times. So the **total number of iterations here are $10^6$.**

In each iteration of the inner for loop, it basically evaluates the value of the function at a random point, the random number generator used is **rd.random( )** which generates a random value between 0 and 1.

Value of the function at such a value is evaluated (by calling the function func( x )) and then that value is stored in the array ans[ ] and the array f_arr[ ].
After the 1000 iterations of the inner loop is complete continue to store the value of our integral evaluation by taking an *average* in accordance to our sample mean algorithm and storing it in the array integral_arr[ ]. The array ans[ ] is initialised to be an empty array again after this so that it can be used again to store the new values of our function evaluations.

In such a way, each of the 1000 inner loop iterations, which correspond to one outer loop iteration, give us a single value of an integral. All of these integral values are stored in the array integral_arr[ ] to be used later. Also, all of the $10^6$ function evaluations are also stored in the array f_arr[ ] to be used later.

**M** stores the average value of the 1000 integral evaluations.

In the later part of my code I basically calculated the standard deviation in the values of the 1000 integral values evaluated and the $10^6$ function values evaluated. To do so each of their respective arrays were first converted to numpy arrays so as to facilitate ease of operations on them without having to use a lot of iterative structures. The standard deviation is calculated using the formulae mentioned above and then stored in **std_dev_in** and **std_dev_fn.**

To calculate the actual error, we take the difference between the value stored in M and $\pi$.

The standard deviation in integral value, functional evaluations; and the actual error are then printed.

The program then ends.

## What are the corner cases or the edge cases for our problem statement?

My job in this assignment was to evaluate an integral over a priorly specified number of points and iterations so there are no edge cases as such that rise during the coding part.

## What is the Space and Time Complexity of the code?

## Time Complexity:

### Total Number of additions:

$m * (n - 1)$ ——> During the nested for loop
$(m - 1)$ ——> summing and storing in **M**
$(m - 1)$ ——>During calculation of **std_dev_in**
$(m * n - 1)$ ——>During calculation of **std_dev_fn**

= $2mn + m - 3$

### Total number of subtractions:

$m * n$ ——> During the nested for loop - function call
$m$ ——> During the nested for loop - storing in **integral_arr** call
$m$ ——> Outside the nested for loop - subtracting from each element of **nump_integral_arr, M**
$m * n$ ——> Outside the nested for loop - subtracting from each element of **nump_f_arr, M**
1 ——>Subtracting from **M**, $\pi$

= $2(mn + m) + 1$

### Total number of multiplications:

$2 * m * n$ ——> During the nested for loop - function call
$m$ ——> During the nested for loop - storing in **integral_arr** call
$m$ ——> Outside the nested for loop - squaring each element of **nump_integral_arr**
$m * n$ ——> Outside the nested for loop - squaring each element of **nump_f_arr**

= $3mn + 2m$

### Total number of division operations:

$m$ ——> During the nested for loop - function call
3 ——> During the rest of the code

= $m + 3$

### Total number of times an array was converted to a numpy array:

2

### Total number of calls to rd.random( ):

$m * n$ ——> During the nested for loop - function call

### Total number of array assignment operations:

$(2 * m * n) + (m)$——> During the nested for loop

### Total number of times an array was converted to a numpy array:

2

## Space Complexity:

### Total space occupied by the array variables:

**ans ——>** No. Of elements is : n
**integral_arr——>** No. Of elements is : m
**f_arr——>** No. Of elements is : m*n
**nump_integral_arr——>** No. Of elements is : m
**nump_f_arr——>** No. Of elements is : m*n

**Space occupied:** $64 bits * (n + 2m + 2mn)$

### Total space occupied by the non-array variables:

**m——>** Space occupied - 32 bits
**n——>** Space occupied - 32 bits
**f——>** Space occupied - 32 bits
**M——>** Space occupied - 32 bits

**Space occupied:** $4 * 32 = 128 bits$

**Total space occupied =** $(128 + 64 * (n + 2m + 2mn)) bits$

## What is the present Status-Quo of Monte Carlo methods?

Monte Carlo methods, in their various forms, find uses in many places in the industry. Basically any industry which requires the measure of risks and uncertainty can find uses of Monte Carlo methods.

To mention a few, Quantum Monte Carlo algorithms are used to solve the many body problem for quantum systems. In experimental particle physics, Monte Carlo methods are used for designing detectors, understanding their behaviour and comparing experimental data to theory. Modern weather forecasting is heavily dependent on ensemble models and these ensemble models utilise Monte Carlo methods. Monte Carlo methods also have far reaching uses in financial markets, mostly in risk analysis and options trading.

As far as the efficiency of the Monte Carlo methods go, they vary vastly from one method to the other largely based on the number of random points generated and also on the way in which they are generated (the distribution using which we generated them).
In our case, $\sigma_m^2 = \dfrac{\sigma^2}{n}$ where $\sigma_m$ is the standard deviation in the value of the Integral evaluation and $\sigma$ is the standard deviation in the value of the functional evaluations. To reduce $\sigma_m$ we can either increase n or reduce $\sigma$.

As a rule of thumb, more the number of random points or evaluations that we generate, more is the accuracy of our result and lesser is the standard deviation of the result values. There is, however, a trade off between increasing number of points and the run-time of the code. We generally try to make our choices of the number of points in such a way so as to make the program accurate enough but also not too large.
Generating the random points along a certain distribution also helps us reduce the variance in the functional evaluations itself which thereby decrease the variance in the value of the result evaluations, which brings us to better algorithms like Importance Sampling Method.