ASEE AMERICAN SOCIETY FOR ENGINEERING EDUCATION

Paper ID #40435

Integration by Gambling

Dr. Murat Tanyel, Geneva College

Murat Tanyel is a professor of engineering at Geneva College. He teaches upper level electrical and biomedical engineering courses. Prior to Geneva College, Dr. Tanyel taught at Dordt College (now Dordt University) in Sioux Center, IA.

Integration by Gambling

Murat Tanyel
Geneva College, Beaver Falls, PA 15010
Email: mtanyel@geneva.edu

Abstract
In modern science and engineering, numerical methods play an important role. Thus, our department requires our mechanical and biomedical engineering students to take CPE 111, Intro to Engineering Computation, in which they learn how to solve problems using MATLAB®. Our electrical and computer engineering students are introduced to the world of computing in CSC 101 using the C programming language with the added bonus of LabVIEW® in EGR 225, the Signals & Systems course. Overseeing both CPE 111 and EGR 225, I regularly look for entertaining examples in which students can learn programming concepts while trying to solve what they will perceive as real-world problems. Having taught MAT 350, Numerical Methods, in Fall '22 I came across Monte Carlo integration which is a good candidate for teaching about for loops and, because of the number of calculations involved, about timing issues. This paper will describe Monte-Carlo integration and provide examples that can be used in CPE 111 with MATLAB and in EGR 225 with LabVIEW. We will also observe the execution timing of the same problems on both platforms.

Introduction
Numerical methods are ubiquitous in engineering education. Almost every engineering textbook I have chosen utilizes a software package or programming environment to demonstrate concepts. In my area, the computer tools of choice are MATLAB and its derivative Simulink, LabVIEW, and Multisim. C and its derivatives are still the main toolkit of our computer engineering students. To prepare our students for proficiency in computer aided applications, the Engineering Department requires mechanical and biomedical engineering students to take CPE 111, an introduction to MATLAB. Electrical and computer engineering students take CSC 101, an introduction to programming with C^{++}. Furthermore, electrical and computer engineering students are required to take EGR 225, Signals & Systems. They and any other engineering student who takes EGR 225 as an elective are introduced to LabVIEW.

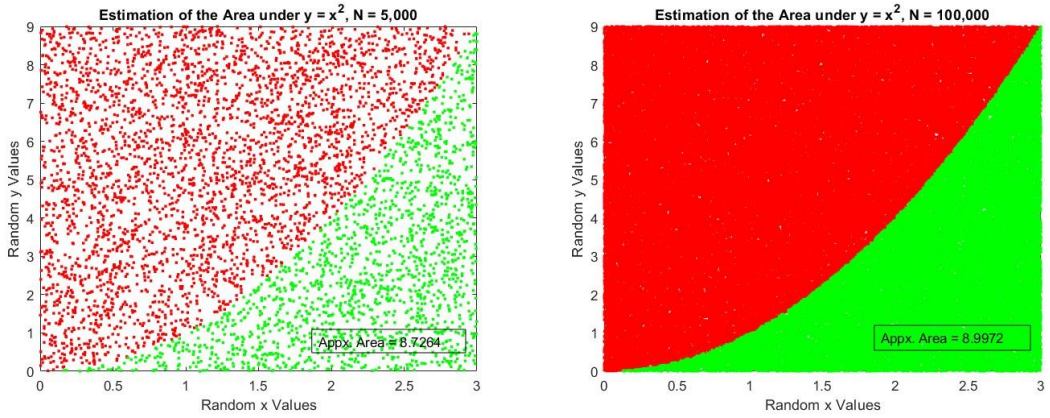
Teaching MAT 350 has reacquainted me with some routines that may be easily adopted to freshman – sophomore level engineering students' technical expertise. I have chosen to adopt Monte Carlo integration methods to generate exercises for CPE 111 students taking MATLAB and EGR 225 students learning to compute with LabVIEW. This paper will describe two approaches to Monte-Carlo integration, provide examples with code for their implementation, compare run times and offer some conclusions in the next sections.

Monte Carlo Integration
In general, Monte Carlo methods predict outcomes of various phenomena through games of chance. While these methods have been around for a long time, the work of John von Neumann and Stanislaw Ulam, along with the development and proliferation of the modern computer, has

advanced these techniques and promoted them to research tools[1]. Georges-Louis Leclerc, Comte de Buffon's 1773 experiment of dropping needles on a surface with parallel lines repeatedly to estimate the value of π is considered to be one of the earliest examples of the Monte Carlo method[1,2].

Monte Carlo integration is employed to estimate integrals of multivariable functions that are difficult, or impossible, to integrate analytically. However, in order to keep the discussion simple enough for a sophomore who may or may not have taken Calculus III, I am going to keep to positive functions of one variable. I will first describe the method adopted from the textbook I used for Numerical Analysis[1]: To calculate the integral of $f(x)$ in the range from $x = a$ to $x = b$, we can generate a random number x_i uniformly distributed between a and b. Then we generate a random number y_i uniformly distributed between 0 and maximum of $f(x)$ in the range. If y_i is $\leq f(x_i)$, we call it a hit and increment a counter. We repeat this process many times and, at the end, the approximate value of the integral will be the number of hits divided by the total number of trials multiplied by the area of the rectangle defined by our x and y ranges.

As our first example, we will integrate $f(x) = x^2$ from 0 to 3. The exact value of the integral is 9, as any beginner in calculus would easily calculate. Figure 1 demonstrates this technique.



(a) (b)

Figure 1: *Demonstration of using Monte Carlo method in the estimation of the area under $f(x) = x^2$ from 0 to 3,* ***(a)*** *with 5,000 Monte Carlo trials,* ***(b)*** *with 100,000 Monte Carlo trials. The green dots represent hits and red dots represent misses. The approximate area is the total area of the rectangle (3 x 9) multiplied by the ratio of the number of hits to the total number of dots.*

Figure 1 (a) is a plot of random x_i and y_i pairs generated in 5,000 Monte Carlo trials. The points are green if y_i is $\leq x_i^2$ (hits) and red if not (misses). The area under the curve $f(x) = x^2$ is the total area spanned by the graph (3 x 9) times the ratio of the number of green dots to the total number of dots. With 5,000 trials, the approximation to the integral comes out to be 8.7264. When the number of trials is increased to 100,000 the approximate gets closer to the actual value: 8.9972 (Figure 1-b).

A simpler application of the method, the crude Monte Carlo method, is described by Victor Cumer in his article in Towards Data Science[3]. In this method, we generate N random numbers, x_i, uniformly distributed between the lower and upper integration limits (a, b). The approximation to the integral, I, is given by

$$I = \frac{b-a}{N-1}\sum_{i=1}^{N} f(x_i) \quad (\mathbf{1}).$$

Panchal provides a statistical backing for this formula in Cantor's Paradise[4]. Both Cumer and Panchal provide visual images provided by Scratchapixel[5], duplicated in Figure 2. They also show how adding up the area of the rectangles in Figure 2 and averaging the sum gets closer to the integral of the function as the number of random points increases.

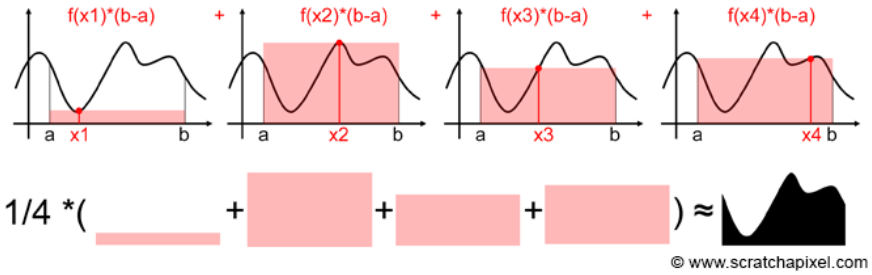
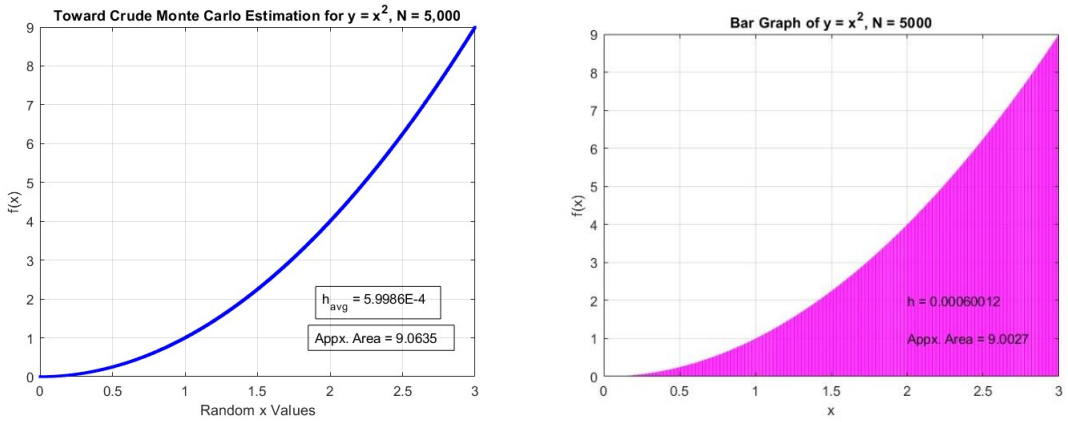


***Figure 2**: Demonstration of crude Monte Carlo method in the estimation of the area under a function as provided in [5]. Used under the terms of CC BY-NC-ND 4.0 License.*

The explanation in Figure 2 may not be that intuitive to some. However, to those who have performed rectangular integration, our second example might provide some insight (Figure 3).



(a) (b)

***Figure 3**: Demonstration of crude Monte Carlo method in the estimation of the area under $f(x) = x^2$ from 0 to 3, **(a)** plot of x_i vs $f(x_i)$ with 5,000 random numbers, **(b)** bar graph of $f(x)$ from $x = 0$ to 3 with 5,000 equally spaced data points*

Figure 3 (a) is a plot of the random x_i versus $f(x_i)$ with N = 5,000 points. Figure 3 (b) is a plot of the function $f(x) = x^2$ with 5,000 equally spaced x values. It employs a bar plot to show the area under the curve. The rectangular integration approximation, I, to the function would be

$$I = \sum_{i=1}^{N} h \times f(x_i) = h\sum_{i=1}^{N} f(x_i) \quad (\mathbf{2})$$

where h is the step size:

$$h = \frac{b-a}{N-1} \quad (\mathbf{3}).$$

With $b - a = 3$ and $N = 5{,}000$ h would be 6.0012×10^{-4}, as noted in Figure 3 (b). Substituting Eq. 3 into Eq. 2 results in a similar expression to Eq. 1. The only difference is that the x_i in Eq. 1 are randomly sampled and those in Eq. 2 are sampled with a step size of h. For a uniform distribution, we would expect the average distance between x_i to approach this h for large N. A calculation of this average distance resulted in $h_{avg} = 5.9986 \times 10^{-4}$, as noted in Figure 3 (a). With these similarities, Eq. 1 looks like a probabilistic version of Eq. 2, which may be a picture more in tune with the intuitions of one who has performed many a rectangular integration.

Implementation of Monte-Carlo Integration with MATLAB & LabVIEW

In this section, we will see the implementation of both Monte Carlo trials and crude Monte Carlo integration using MATLAB and LabVIEW. As our third example, we will use the same illustration that MAT 350 students worked on. We are going to work with the function

$$f(x) = \frac{1}{\log(2)} \log(x+1) \quad \mathbf{(4)}$$

where the log function is the natural log (i.e., ln), as in mathematics and MATLAB nomenclature. We are going to estimate its integral from $x = 0$ to 1. We are going to see how many Monte Carlo trials will bring us close to the analytical solution. The analytical evaluation of this integral is straightforward, making use of integration by parts. However, the evaluation of the result will still be approximate since it involves taking natural logarithms. WolframAlpha solution approximates this integral as

$$\int_0^1 \frac{\log(x+1)}{\log(2)} dx = \frac{\log(4) - 1}{\log(2)} \approx 0.55730$$

The MATLAB code in Figure 4 will ask for N, the number of Monte Carlo trials and report the outcome.

```
N = input('Enter number of Monte Carlo trials: ');
numberin = 0;   % Counter for number under the curve for the function.
for i=1:N  % Loop over points.
    x = rand;  % Random point in the x range.
    y = rand;  % Random point in the y range.
    if y <= (1/log(2))*log(x+1)   % See if point is under the curve.
        numberin = numberin + 1;    % If so, increment counter.
    end
end
intapp = numberin/N;  % Approximation to the integral.
disp(['The approximation to the integral with ',num2str(N), ' points is: '])
disp(intapp)
```

Figure 4: *MATLAB code to estimate* $\int_0^1 \frac{\log(x+1)}{\log(2)} dx$

The LabVIEW VI, whose front panel and block diagram are depicted in Figure 5 will achieve the same result. With the criterion that the estimate of the integral should match the value obtained through WolframAlpha in the first three significant digits, N = 1,000,000 yielded a value of 0.5572 with MATLAB and 0.557156 with LabVIEW.

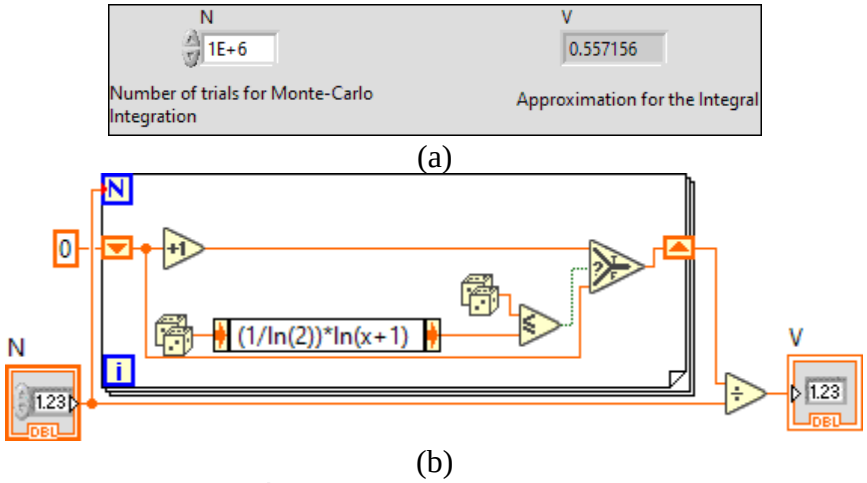


***Figure 5**: LabVIEW code to estimate $\int_0^1 \frac{\log(x+1)}{\log(2)} dx$, **(a)** front panel, **(b)** block diagram. It should be noted that LabVIEW uses engineering nomenclature for the natural log function.*

```
%This routine evaluates the integral of 1/log(2))*log(x+1) from 0 to 1
%with N Monte-Carlo trials.
N = input('Enter number of Monte-Carlo trials: ');
x=rand(1,N);
fx=(1/log(2)).*log(x+1);
intapp = sum(fx)/(N-1);
disp(['The approximation to the integral with ',num2str(N), ' points is: '])
disp(intapp)
```

***Figure 6**: MATLAB code to estimate $\int_0^1 \frac{\log(x+1)}{\log(2)} dx$ with the crude Monte Carlo method*

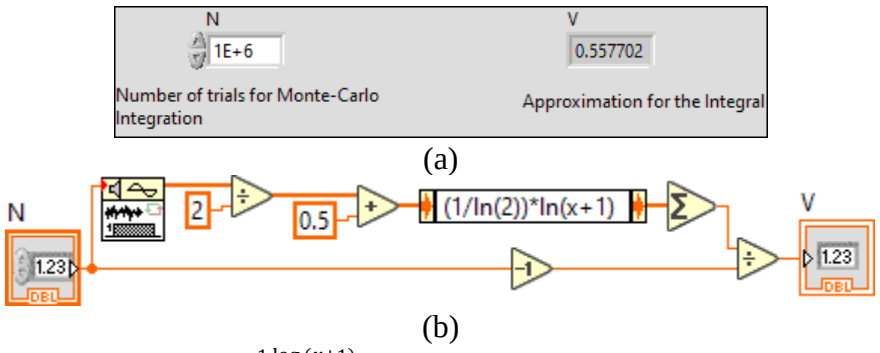


***Figure 7**: LabVIEW code to estimate $\int_0^1 \frac{\log(x+1)}{\log(2)} dx$ with the crude Monte Carlo method, **(a)** front panel, **(b)** block diagram.*

Our fourth example is shown in Figures 6 & 7. Figure 6 contains the MATLAB code to perform the crude Monte Carlo integration for the function in equation (4), along with Figure 7 which displays the LabVIEW implementation. With the same criterion as before, the MATLAB

calculation resulted in a value of 0.5576 and the LabVIEW VI returned a value of 0.557702 after 1,000,000 iterations.

Timing Trials

Obtaining timing information is straightforward in MATLAB. Two stopwatch functions, *tic* and *toc* are associated with timekeeping. *tic* records the current time and *toc* uses this recorded time to calculate the elapsed time. Since MATLAB is a textual language, one would place the *tic* and the *toc* at the beginning and end of the lines of code one is timing. In this most simple use of the stopwatch, MATLAB reports the elapsed time in seconds after the script is run. In the Monte Carlo trials, placing the *tic* before the for loop in Figure 7 and the *toc* after the calculation of the variable `intapp` will accomplish the task. LabVIEW, on the other hand, is a graphical language. Flow of data, rather than lines of code, determines how VIs run. For this reason, the blocks of functions need to be placed in sequence before one can use its timing functions to calculate elapsed times. Figure 8 depicts the block diagram of Figure 5 rearranged to measure the time elapsed.

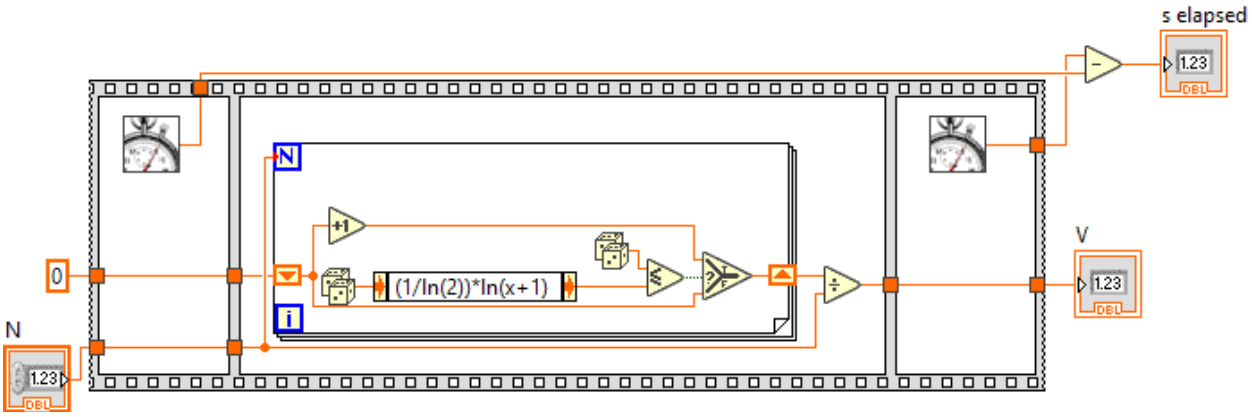


***Figure 8**: LabVIEW code using a sequence structure to calculate the elapsed time in the estimation of* $\int_0^1 \frac{\log(x+1)}{\log(2)} dx$.

After N = 100,000,000 trials, MATLAB reported 8.224780 seconds while LabVIEW was less than a second faster at 7.45476 seconds.

After similar changes were made for the crude Monte Carlo method, MATLAB improved its performance by dropping to 5.017071 seconds for 1E+8 data points, faster than LabVIEW's 5.94987 seconds. One reason in this reversal of order might be due to the fact that the Monte Carlo trials of Figures 4 and 5 involved generating single variables in a loop while the crude Monte Carlo integration of Figures 6 and 7 involved arrays. MATLAB was originally designed to perform matrix (i.e., array) calculations fast. Since the crude Monte Carlo integration makes use of arrays, MATLAB is at an advantage in this procedure. Another minor difference in the code of the two packages could be due to the array random number generators. In MATLAB, the array random number generator generates a uniformly distributed array between 0 and 1, just like the scalar application. In LabVIEW, on the other hand, scalar number generator will yield a uniform distribution between 0 and 1, while the array generator's values vary between -1 and 1. Therefore, the array had to be divided by 2 and shifted by 0.5 (Figure 7-b) to obtain the range of

random numbers. However, these extra operations did not seem to be that significant; removing the division by 2 and addition of 0.5 decreased the elapsed time to only 5.51638 seconds.

Conclusions

The Monte-Carlo trials based on Buffon's needle-dropping experiment of 1773 and the crude Monte Carlo integration can be used in an introductory MATLAB programming course, like CPE 111 or in introduction to LabVIEW. This paper has provided four examples which can be easily implemented at freshman and sophomore level.

The timing trials on these examples have revealed that while LabVIEW may be slightly faster in scalar calculations, MATLAB performs faster on array calculations.

It should be noted that there are much more efficient algorithms to estimate integrals of functions of one variable; these examples serve to demonstrate the principles behind the Monte Carlo method at freshman-sophomore level.

Bibliography

[1] A. Greenbaum and T. Chartier, *Numerical Methods: Design, Analysis, and Computer Implementation of Algorithms*, Princeton, NJ, USA: Princeton University Press, 2012

[2] "Monte Carlo method," *Encyclopaedia Britannica*, Encyclopaedia Britannica, Inc., https://www.britannica.com/science/Monte-Carlo-method, Accessed 18 Jan. 2023

[3] V. Cumer, "The Basics of Monte Carlo Integration," *Towards Data Science*, Oct. 26, 2020. [Online] Available: https://towardsdatascience.com/the-basics-of-monte-carlo-integration-5fe16b40482d

[4] S. Panchal, "Demystifying Monte Carlo Integration," *Cantor's Paradise*, Mar. 27, 2022, [Online] Available: https://medium.com/cantors-paradise/demystifying-monto-carlo-integration-7c9bd0e37689

[5] "Monte Carlo Integration," *Monte Carlo Methods in Practice*, Scratchapixel 3.0, https://www.scratchapixel.com/lessons/mathematics-physics-for-computer-graphics/monte-carlo-methods-in-practice/monte-carlo-integration.html, Accessed 16 Jan. 2023