Monte Carlo Integration

Created by Haroon Sakhi

Introduction

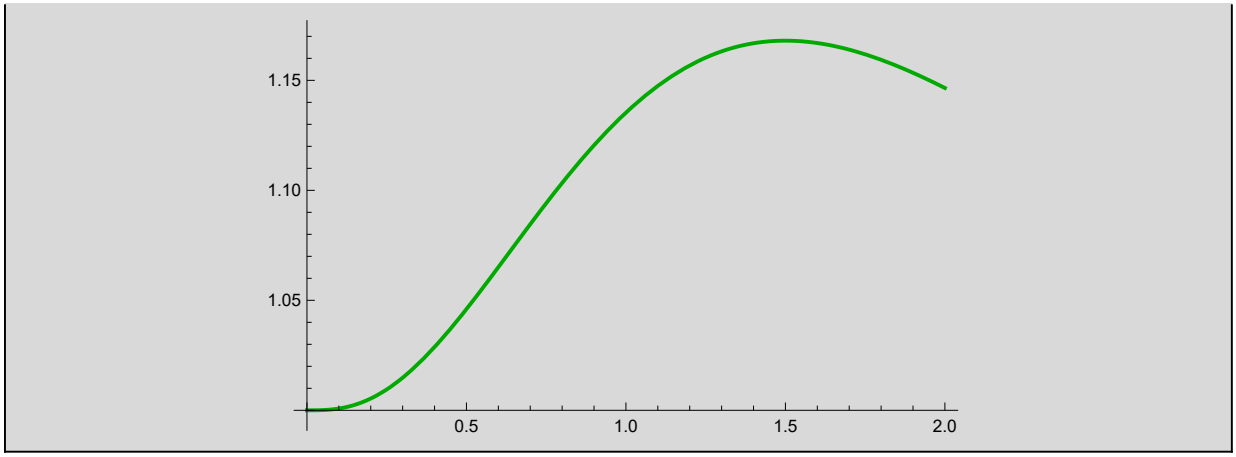
Monte Carlo integration is a numerical technique that estimates the area under a curve using random sampling. It is part of the broader Monte Carlo methods, which rely on probability and large sample sizes to solve complex numerical problems.

Problem Setup

Suppose we have a function f(x) representing a curve, and we want to find the area under this curve

```
Plot[1 + x^3 Exp[-2 x], {x, 0, 2}, PlotStyle → {Thick, Darker[Green]}]
```

Out[•]=



In many cases, direct analytical integration is either impossible or impractical. For such situations, numerical approaches like Monte Carlo integration provide an alternative solution:

$$I = \int_a^b f(x)\, d\, x$$

Monte Carlo Integration

The method begins by enclosing the curve within a rectangular boundary. The choice of the rectangle's

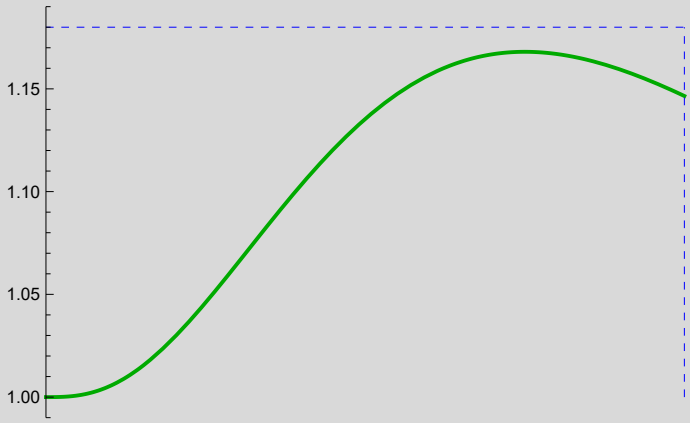
dimensions is crucial, as it should be as tight as possible to ensure efficiency

In[217]:=

```
f[x_] := 1 + x^3 Exp[-2 x]

xMin = 0;
xMax = 2;
yMin = 1;
yMax = 1.18;

Show[Plot[f[x], {x, xMin, xMax}, PlotStyle → {Thick, Darker[Green]}],
Graphics[{Dashed, Blue, Line[{{xMin, yMax}, {xMax, yMax}}], (*Top horizontal line*)
   Line[{{xMax, yMin}, {xMax, yMax}}] (*Right vertical line*)}],
Axes → True, AxesOrigin → {0, 0},
 PlotRange → {{xMin, xMax}, {yMin, yMax}}, ImageSize → Medium]
```



The ratio between the area of the rectangle A and the area under the curve I remains constant, if we do not change the domain of the curve and the rectangle's dimensions :

$$\frac{A}{I} = C$$

Similarly, if we randomly generate N points within the rectangle, and n of those fall under the curve, then:

$$\frac{N}{n} = C$$

Equating these expressions, we derive:

$$I = \frac{A\,n}{N}$$

Since the area of a rectangle is given by $A = l \times h$, the formula simplifies to:

$$I = \frac{(l \times h)\,n}{N}$$

Algorithm

To implement Monte Carlo integration, we follow these steps:

- Define the function $f(x)$.
- Determine the dimensions of the enclosing rectangle.
- Generate N random points within the rectangle.
- Count how many points n fall below $f(x)$.
- Compute the area using $I = \frac{(l \times h)\, n}{N}$.

Example Problem

Defining the Function $f(x)$.

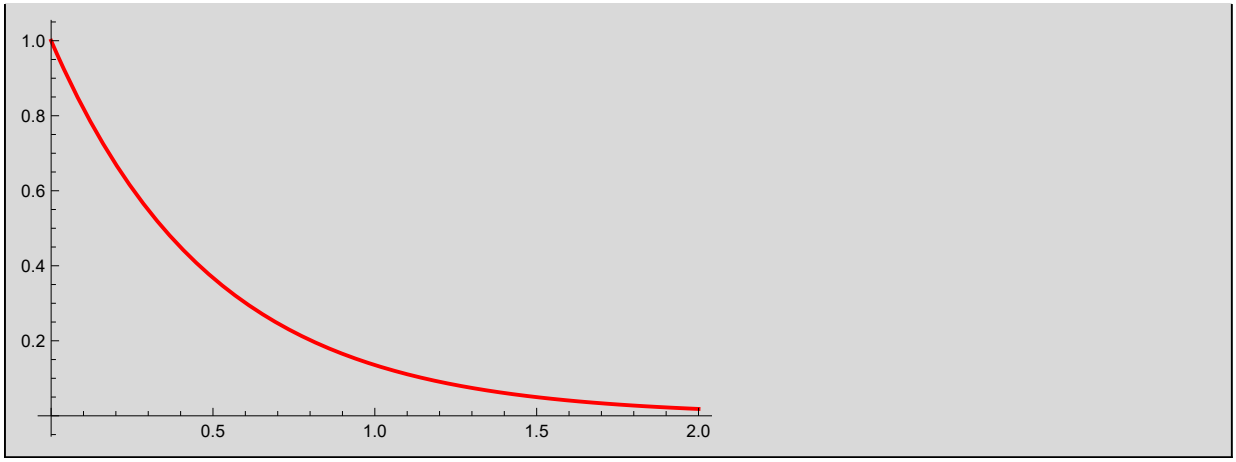
Consider the function:

In[2]:=
```
f[x_] := Exp[-2 x]
```

Visualization

```
Plot[Exp[-2 x], {x, 0, 2}, PlotStyle → {Thick, Red}]
```

Out[•]=



Determining the rectangle dimensions

Let us find the area under this curve for domain [0, 2]. The maximum value of the curve in this domain

is easy to figure out and is $y = 1$. Hence we will construct a rectangle of dimensions (2×1),

In[3]:=
```
length = 2;
height = 1;
totalPoints = 100;
pointsUnderCurve = 0;
```

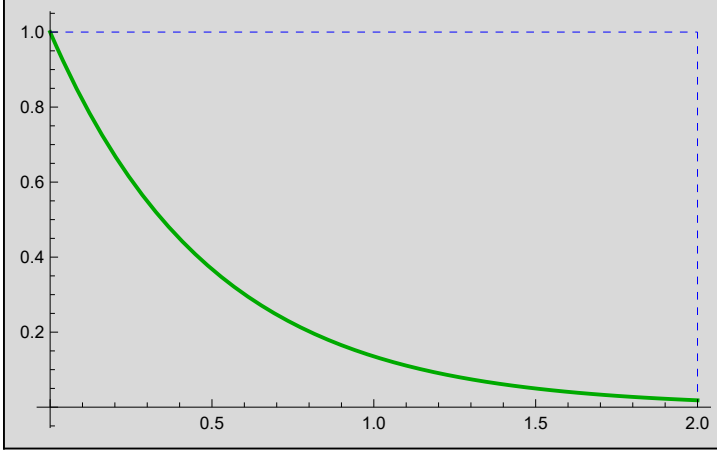
Visualization

Which would look something like this,

In[20]:=
```
f[x_] := Exp[-2 x]

xMin = 0;
xMax = 2;
yMin = 0;
yMax = 1;

Show[Plot[f[x], {x, xMin, xMax}, PlotStyle → {Thick, Darker[Green]}],
Graphics[{Dashed, Blue, Line[{{xMin, yMax}, {xMax, yMax}}], (*Top horizontal line*)
   Line[{{xMax, yMin}, {xMax, yMax}}] (*Right vertical line*)}],
Axes → True, AxesOrigin → {0, 0},
 PlotRange → {{xMin, xMax}, {yMin, yMax}}, ImageSize → Medium]
```



Random Points

The area of the rectangle therefore would be 2. For simplicity let us generate a mere 100 points and see what ratio $\left(\frac{n}{N}\right)$ do we get.

In[13]:=
```
Do[x = RandomReal[{0, 2}];
   y = RandomReal[{0, 1}];
   If[y < f[x], pointsUnderCurve++], {totalPoints}];
```

Results

After generation the answer we get for the area under the curve is,

In[17]:=
```
area = length * height;
areaUnderCurve = area * (pointsUnderCurve / totalPoints);

Print["Estimated Area under the curve: ", areaUnderCurve];
```

Estimated Area under the curve: $\frac{23}{50}$

$$I = 0.46$$

Computational Efficiency in Mathematica

Mathematica provides several built-in features that optimize Monte Carlo simulations without the need for low-level programming, such as efficient random sampling, where its RandomReal[] function is optimized for high-performance random number generation—unlike Python, which struggles with large-scale Monte Carlo simulations due to its interpreted nature—while methods like BlockRandom ensure reproducibility in simulations.