Games of Chance: Exploring the Monte Carlo Technique for Numerical Integration and Computing Probabilities with Improper Integrals

Note: You may notice differences between this Maple worksheet and the equivalent Mathematica notebook. These differences were introduced to preserve the content of these modules and were necessary because of major functional differences between Maple and Mathematica.

Introduction

OBJECTIVE: Learn the Monte Carlo method for approximating an integral that cannot be integrated symbolically, and see an application of improper integrals.

How can you use a game of chance to evaluate an integral and what do games of chance have to do with improper integrals? This module will give you insight into both of these issues.

1. You will generate random points within a fixed region and then estimate the area of the desired portion by considering the percentage of random points that fall within the boundaries of the desired portion.

2. You will consider a probability density function defined over an infinite interval and explore the computation of means, moments, and probabilities associated with the density function.

Technology Guidelines

NOTE: If you have just finished a worksheet, **restart** *Maple* before executing a new worksheet.
TO OPEN SECTIONS,
Click on the **PLUS** sign at the left hand side of the screen *or* select **Expand All Sections** from the **View** drop down menu.

TO STOP AN EXECUTION
Click on **STOP** button from the toolbar.

ORDER OF EXECUTION
Execute commands in the order given. Do not skip any *Maple* Input lines within a given worksheet

Alternatively, you can execute the entire worksheet by selecting the **Execute Worksheet** command from the **Edit** drop down menu.

SAVING WORKSHEETS.

You can save anytime to any directory you choose, and it is wise to save often.
EXPERIENCING MAJOR PROBLEMS
Save if appropriate, and then shut down *Maple* and start it up again.

Part I: The Monte Carlo Method

Following is an example of a numerical integration technique called the Monte Carlo method. In this method, the area between a curve (with nonnegative *y*-values) and the x-axis is completely enclosed in a rectangle, and points are randomly selected within that rectangular region. Then the number of points that land under the curve is determined, and the area under the curve (down to the x-axis) is approximated as the percentage of generated points that lie under the curve times the area of the entire rectangle. The function we used here as an example cannot be integrated symbolically, so we are using *Maple's* numerical integrator to come up with an answer that we would expect to be very close to the actual one.

The following commands will make the dimensions of the box selected 4 by 1, giving an area of 4. First we will graph our function.

```
> restart;
```

```
> f := x->1/(sqrt(2*Pi))*exp(-0.5*x^2);
```

$$f := x \rightarrow \frac{e^{(-0.5\,x^2)}}{\sqrt{2\,\pi}}$$

```
> xmin := -2:
```

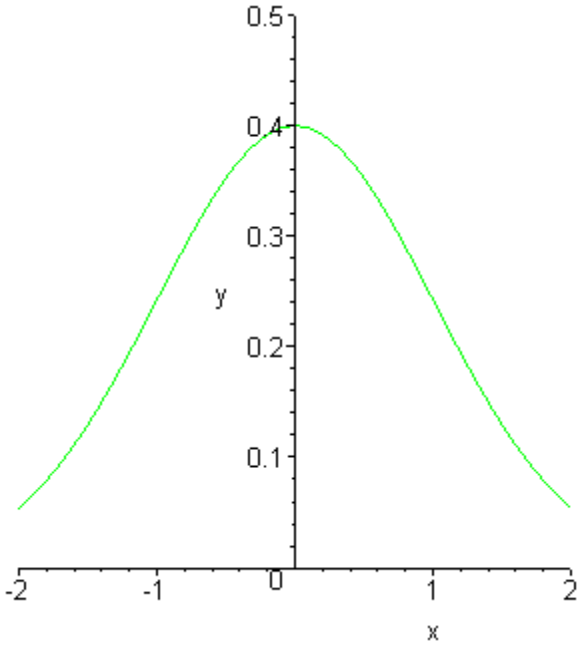
```
> xmax := 2:
```

```
> ymin := 0:
```

```
> ymax := .5:
```

```
> pf := plot(f(x), 'x'=xmin..xmax, y=ymin..ymax, color=GREEN):
```

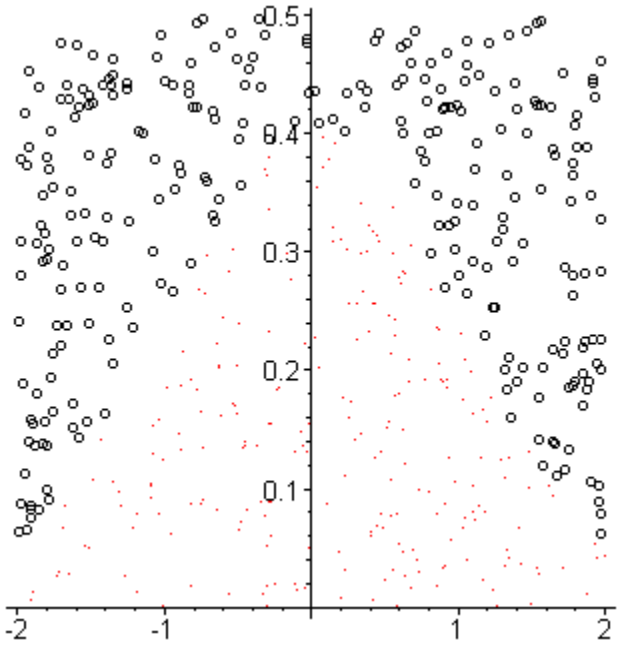
```
> pf;
```



In utilizing the **Random** command in *Maple*, we specify the type of numbers we want and the interval over which we want those numbers randomly selected. Next, we will generate 500 points and see how they fall.

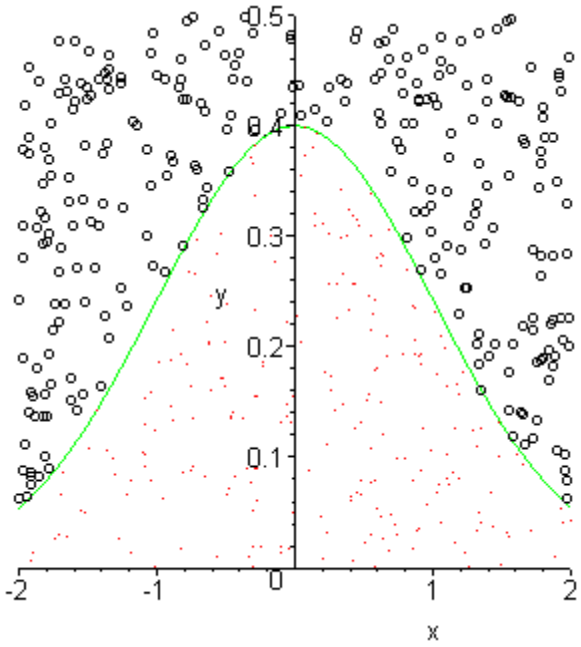
```
> numberofpoints := 500:

> areaofbox := (xmax-xmin)*(ymax-ymin):

> count := 0:

> listout := []:

> listin := []:

> rf := rand(0..10000):

> for i from 1 to numberofpoints do
    px := evalf(xmin + (xmax-xmin)*rf()/10000):
    py := evalf(ymin + (ymax-ymin)*rf()/10000):
    if py < evalf(f(px)) then
      count := count+1:
      listin := [op(listin), [px,py]]:
    else
      listout := [op(listout), [px,py]]:
    fi:
  od:
```

```
> pout := plots[listplot](listout,style = POINT, symbol=CIRCLE):

> pin := plots[listplot](listin, style=POINT, symbol=POINT, color=RED):

> plots[display]({pout,pin});
```



The red points are those that are under our curve. Do they look right?

```
> plots[display]({pout,pin,pf});
  actual := evalf(int(f(x), x=xmin..xmax));
  estimate := evalf(areaofbox*count/numberofpoints);
  err:= estimate-actual;
  percenterror := err/actual:
  print(`relative error = `,percenterror);
```



actual := 0.9544997361

estimate := 0.9200000000

err := -0.0344997361

relative error = , -0.03614431183

We have given the actual value as determined by the software and have computed the error. How does your error compare to those of your classmates? Each time you run this, you will get a different answer. It might be better or it might be worse. Does this method seem very accurate? Analyze the error involved in such an approximation. In a later module on using Monte Carlo techniques for approximating volumes, in the "You Try It" exercises below, see if your error gets smaller when you select more random points. You will also check out the technique for a function of your own choosing.

You Try It: Part I

Generating 1000 Random Points

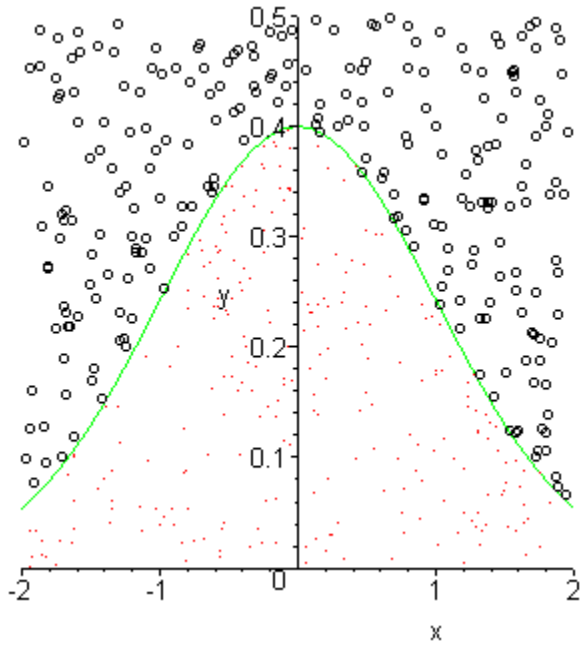
Redo the Monte Carlo method above, but generate twice as many random points. Compare your error to the results above.

> **numberofpoints := 500:**

> **areaofbox := (xmax-xmin)*(ymax-ymin):**

> **count := 0:**

```
> listout := []:

> listin := []:

> rf := rand(0..10000):

> for i from 1 to numberofpoints do
    px := evalf(xmin + (xmax-xmin)*rf()/10000):
    py := evalf(ymin + (ymax-ymin)*rf()/10000):
    if py < evalf(f(px)) then
      count := count+1:
      listin := [op(listin), [px,py]]:
    else
      listout := [op(listout), [px,py]]:
    fi:
  od:

> pout := plots[listplot](listout,style = POINT, symbol=CIRCLE):

> pin := plots[listplot](listin, style=POINT, symbol=POINT, color=RED):

> plots[display]({pout,pin,pf});
  actual := evalf(int(f(x), x=xmin..xmax));
  estimate := evalf(areaofbox*count/numberofpoints);
  err := estimate-actual;
  percenterror := err/actual:
  print(`relative error = `,percenterror);
```



$$actual := 0.9544997361$$

$$estimate := 1.032000000$$

$$err := 0.0775002639$$

$$relative\ error = , 0.08119464152$$

How does your percentage error compare to that when only 500 points were generated?

Selecting a Different Function

When applying a function of your own to this procedure, be certain to determine the range in which you enclose your desired area (values of **xmin**, **smax**, **ymin**, **ymax**). In choosing a function, choose one that is not negative in the specified domain, and note that you are finding the area between the surface and the *x*-*y* plane. The only commands that you need to enter in the following template are your function, minimum and maximum values for your variables, except for leaving your ymin at 0o. You should realize that the bigger your rectangular region, the more points you will need to generate to get an answer that is reasonably accurate. Why?

```
> restart;
  f := x->1-sin(x^2);
```

$$f := x \to 1 - \sin(x^2)$$

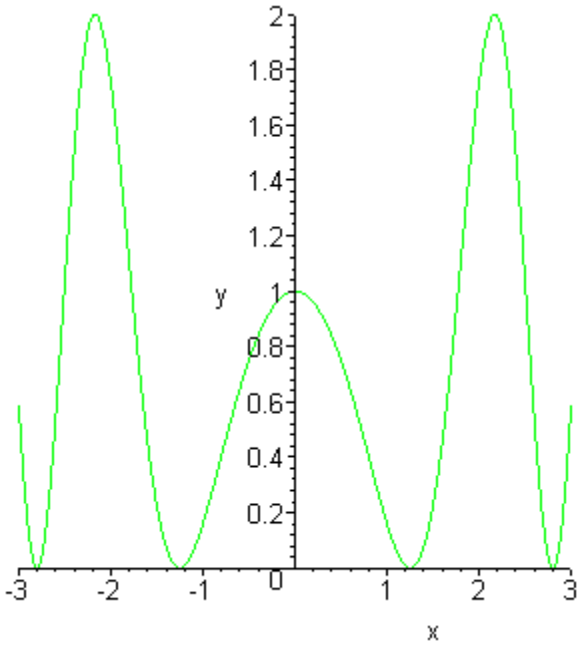
```
> xmin := -3:
```

```
> xmax := 3:
```

```
> ymin := 0:
```

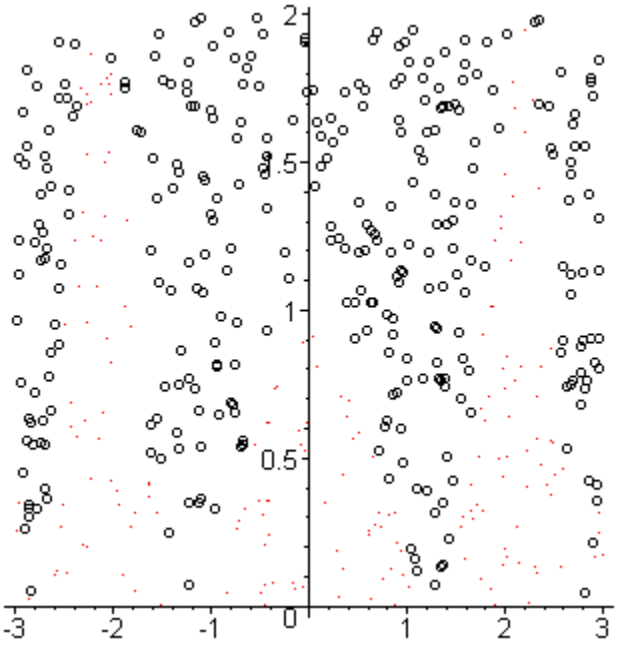
```
> ymax := 2:
```

```
> pf := plot(f(x), 'x'=xmin..xmax, y=ymin..ymax, color=GREEN):
  print(pf);
```

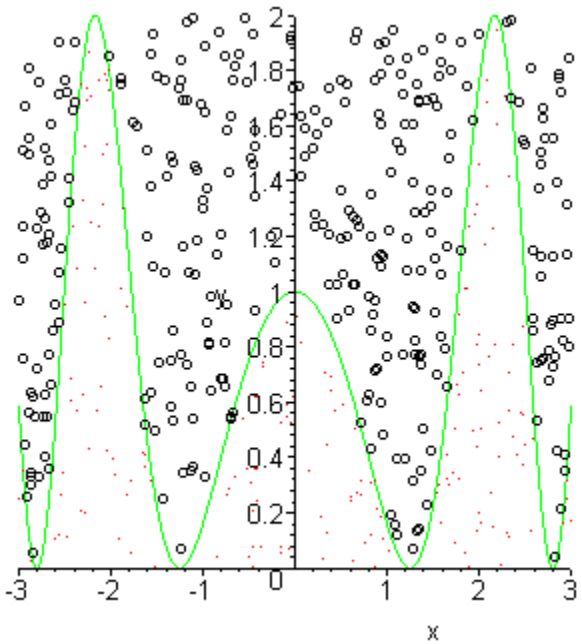



```
> numberofpoints := 500:

> areaofbox := (xmax-xmin)*(ymax-ymin):

> count := 0:

> listout := []:

> listin := []:

> rf := rand(0..10000):

> for i from 1 to numberofpoints do
    px := evalf(xmin + (xmax-xmin)*rf()/10000):
    py := evalf(ymin + (ymax-ymin)*rf()/10000):
    if py < evalf(f(px)) then
      count := count+1:
      listin := [op(listin), [px,py]]:
    else
      listout := [op(listout), [px,py]]:
    fi:
  od:

> pout := plots[listplot](listout,style = POINT, symbol=CIRCLE):

> pin := plots[listplot](listin, style=POINT, symbol=POINT, color=RED):
```

```
plots[display]({pout,pin});
```




```
> plots[display]({pout,pin,pf});
  actual := evalf(int(f(x), x=xmin..xmax));
  estimate := evalf(areaofbox*count/numberofpoints);
  err := estimate-actual;
  percenterror := err/actual:
  print(`relative error = `,percenterror);
```



$$actual := 4.452874947$$

$$estimate := 4.320000000$$

$$err := -0.132874947$$

$$relative\ error = , -0.02984026019$$

How does your percentage error compare with the others you've done? If it is larger, is your rectangle larger? If it is smaller, is your rectangle smaller?

Part II: Probability Density Functions and Improper Integrals

The Standard Normal Distribution

The function that we looked at above $f(x) = \frac{1\, e^{(-.5\, x^2)}}{\sqrt{2\,\pi}}$, happens to represent the

probability density function for the standard normal distribution, a distribution that is used extensively in experimentation and in probability and statistics. We will use our computational tool to find:

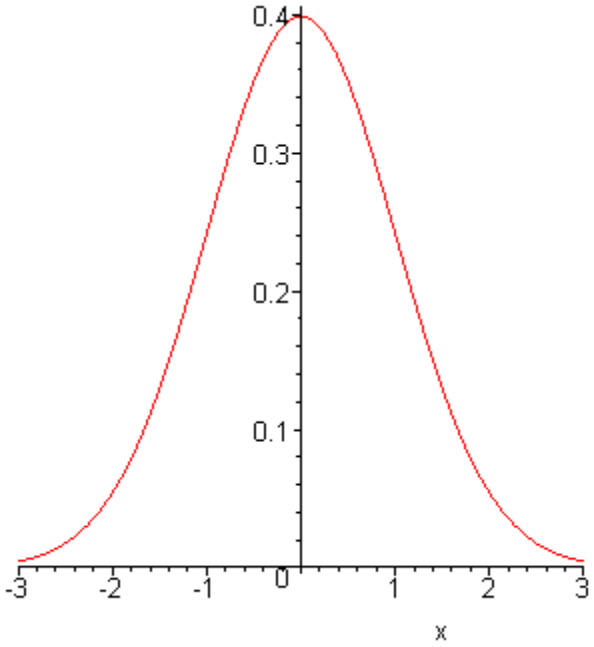
the total mass, which is the area under this curve from $-\infty$ to $+\infty$,

the mean value of *x*, which is the first moment about 0,
the variance of *x*, which is the second moment about the mean, and the standard deviation, which is the square root of the variance.

```
> restart;
  f := x->1/(sqrt(2*Pi))*exp(-0.5*x^2);
```

$$f := x \to \frac{e^{(-0.5\, x^2)}}{\sqrt{2\,\pi}}$$

```
> plot(f(x), x=-3..3);
  print(`total mass = `, int(f(x), x=-infinity..infinity));
  print(`mean = `, int(x*f(x), x=-infinity..infinity));
  var := int(x^2*f(x), x=-infinity..infinity):
  print(`variance = `, var);
  print(`standard deviation = `, sqrt(var));
```



total mass = , 1.

mean = , 0.

variance = , 1.

standard deviation = , 1.

Computing Probabilities for the Normal Distribution

The bell-shaped curve above is used extensively in applied problems, including error analysis. To compute the probability that *x* is greater than .5, we would find the area under the density function from .5 to ∞ .

> **print(`probability that x is greater than .5 is `, int(f(x), x=.5..infinity));**

probability that x is greater than .5 is , 0.3085375387

Similarly, to find the probability that *x* is less than -.67, we find the area under the density function from -∞ to -.67 .

> **print(`probability that x is less than -0.67 is `, int(f(x), x=-infinity..-0.67));**

probability that x is less than -0.67 is , 0.2514288951

Therefore the "-.67" value of *x* represents approximately the 25th percentile for this distribution. What does the term percentile mean to you? Does it correspond to this

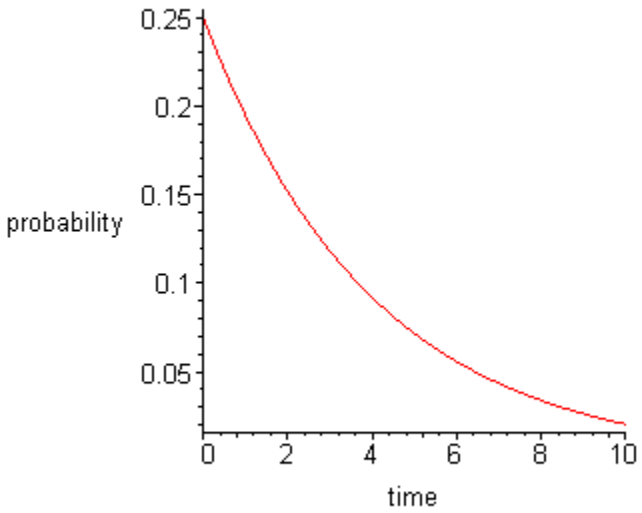
interpretation?

You Try It: Part II

What if you wished to examie the length of time that people spend waiting in line at a drive-up fastfood restaurant? Frequently, the waiting times follow what is called an exponential distribution, whose density function is written as $f(t) = \lambda e^{(-\lambda t)}$, for certain values of λ .

You should note that this function will be defined only for values of t greater than 0, since it would make no sense to talk about waiting a negative amount of time. Assume that time is measured in minutes.

```
> restart;
  lambda := .25:
  f := t->lambda*exp(-lambda*t):
  plot(f(x),x=0..10, labels=["time","probability"]);
  print(`total mass = `, int(f(x), x=0..infinity));
  mean := int(x*f(x), x=0..infinity):
  print(`expected wait = `, mean);
  var := int((x-mean)^2*f(x), x=0..infinity):
  print(`variance = `, var);
  print(`standard deviation of wait time = `, sqrt(var), ` minutes`);
```



total mass = , 1.

expected wait = , 4.

variance = , 16.

standard deviation of wait time = , 4.000000000, *minutes*

Consider the shape of the exponential distribution. Observe that the horizontal axis represents waiting times, while the vertical axis represents the relative frequency of those waiting times. This distribution applies to many waiting time scenarios for which most people wait a shorter period of time and only a few wait a longer period of time.

Compute the probability of waiting:
more than 6 minutes
less than 3 minutes
Change the numbers to the appropriate values for each of the cases above.

> **int(f(x), x=1..5);**

0.4922959862

> ?

>

Now go back to the set of commands and change the value of λ . See what happens to the mean and standard deviation. What does the value of lambda mean in the context of the problem?