Take Your Chances: Try the Monte Carlo Technique for Numerical Integration in Three Dimensions

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 a multiple integral that cannot be integrated symbolically.

How can you use a game of chance to evaluate a multiple integral? That is just what you will do with this project. You will generate random points within a fixed region and then estimate the volume of the desired portion by considering the percentage of random points that fall within the boundaries of the desired portion. Since this will only estimate the exact volume, you will also explore the accuracy of this method.

In this module, you will notice the following warning "**Warning, unknown plot device**". You can expect this because we disable some of the plotting features in order to speed up some of the numerically intense procedures.

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

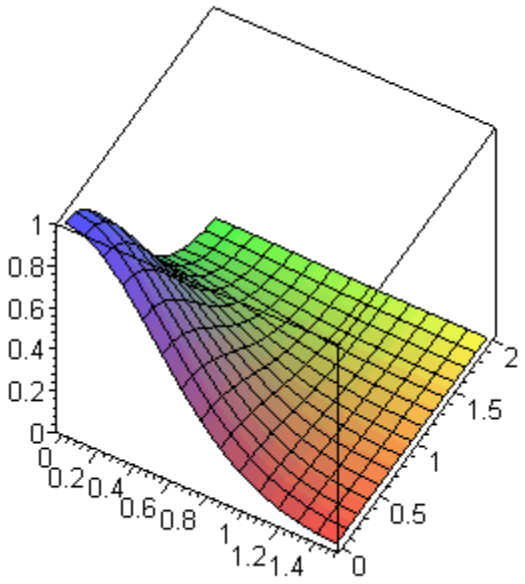
The Monte Carlo method is an example of a numerical integration technique. In this method, the volume under a surface is completely enclosed in a rectangular box and points within that box are randomly selected . The number of points that land under the surface is determined and the volume under the surface is approximated as the percentage of generated points that lying under the surface times the volume of the entire box. In the example that follows, 1000 random points are generated to estimate the volume. Note that the sample function used here cannot be integrated symbolically, so we are using *Maple's* numerical integrator, a different approximation technique, to come up with an answer we expect to be close to the actual one.

A special function **Random(a, b)** was created for this worksheet. The first argument **a** and **b** specifies the range of the random number to be generated. The following commands in this section will make the dimensions of the box selected 2 by 3 by 1, giving a volume of 6. When applying a function of your own to this procedure, be certain to construct carefully the box in which you enclose your desired volume. In choosing a function, be careful to choose one that is not negative in the specified domain and note that you are finding the volume between the surface and the *x*-*y* plane. We define the function and specify the region over which we will generate points and also look at the graph of the function and the region.

Once the graph is plotted, click and drag the plot to view it from different angles. You can do this for all 3D plots.

```
> plotsetup(default);
```

```
> restart:
  f:=(x,y)->exp(-2*x^2-y^2):
  xmin:=0:
  xmax:=1.5:
  ymin:=0:
  ymax:=2:
  zmin:=0:
  zmax:=1:
  pf:=plot3d(f,xmin..xmax,ymin..ymax,view=zmin..zmax,axes=boxed,
  orientation=[-61,40],grid=[15,15]):plots[display](pf);
  actualf:=evalf(Int(Int(f(x,y),x=xmin..xmax),y=ymin..ymax)):
  print(` actual volume = `,actualf);
```

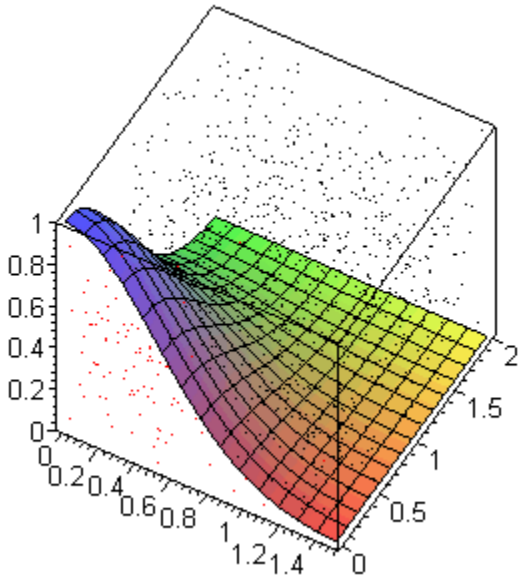


actual volume = , 0.5512701925

Now we will generate random points in our specified region and check to see whether or not they are under our surface. The points in **listin** are under the surface in the volume we are computing, and they will be plotted in red. The points in **listout** are not under the surface, and they will be plotted in black.

```
> numberofpoints:=1000:
  volumeofbox:=(xmax-xmin)*(ymax-ymin)*(zmax-zmin):
  count:=0:
  listin:=[]:
  listout:=[]:
  Random:=proc(a, b)
  evalf(rand(ceil(a*10^Digits)..floor(b*10^Digits))()/10^Digits);
  end:
  for i from 1 to numberofpoints do
   x:=Random(xmin,xmax):
   y:=Random(ymin,ymax):
   z:=Random(zmin,zmax):
    if z < evalf(f(x,y)) then
      listin:=[op(listin),[x,y,z]]:
      count:= count + 1:
    else listout:=[op(listout),[x,y,z]]
    fi:
  od:
  pout:=plots[pointplot3d](listout, symbol=POINT, color=black):
  pin:=plots[pointplot3d](listin,symbol=POINT,color=red):
  plots[display](pin, pout, pf);
  print(` actual = `, actualf);
  estimate:= evalf(volumeofbox*count/numberofpoints):
```

```
print(` estimate = `, estimate);
Error:= estimate - actualf:
print(` error = `, Error);
percenterror:= Error / actualf:
print(` relative error = `, percenterror);
```



actual = , 0.5512701925

estimate = , 0.6090000000

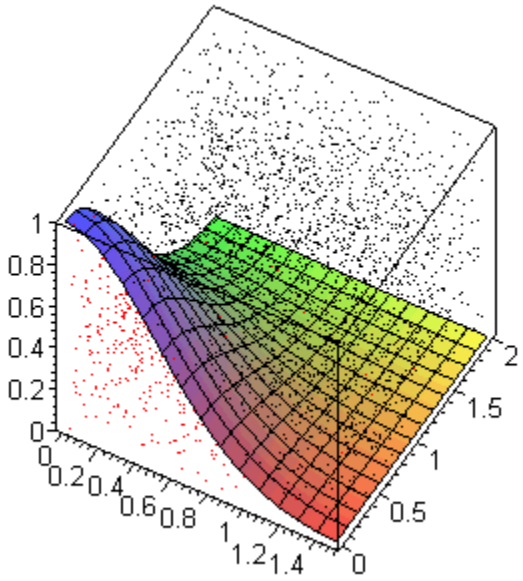
error = , 0.0577298075

relative error = , 0.1047214384

How does your error compare to that of one of your classmates or to a second calculation you produce? Note what happens when you increase the number of points you use for the calculation.

```
> numberofpoints:=3000:
  volumeofbox:=(xmax-xmin)*(ymax-ymin)*(zmax-zmin):
  count:=0:
  listin:=[]:
  listout:=[]:
  Random:=proc(a, b)
  evalf(rand(ceil(a*10^Digits)..floor(b*10^Digits))()/10^Digits);
  end:
  for i from 1 to numberofpoints do
   x:=Random(xmin,xmax):
   y:=Random(ymin,ymax):
   z:=Random(zmin,zmax):
    if z < f(x,y) then
      listin:=[op(listin),[x,y,z]]:
```

```
    count:= count + 1:
   else listout:=[op(listout),[x,y,z]]
  fi:
od:
pout:=plots[pointplot3d](listout,symbol=POINT,color=black):
pin:=plots[pointplot3d](listin,symbol=POINT,color=red):
plots[display](pin, pout, pf);
print(` actual = `, actualf);
estimate:= evalf(volumeofbox*count/numberofpoints):
print(` estimate = `, estimate);
Error:= estimate - actualf:
print(` error = `, Error);
relativwerror:= Error / actualf:
print(` relative error = `, relativeerror);
```



actual = , 0.5512701925

estimate = , 0.5950000000

error = , 0.0437298075

relative error = , *relativeerror*

Does this method seem very accurate? We will analyze the error involved in such an approximation.

You Try It: Part I

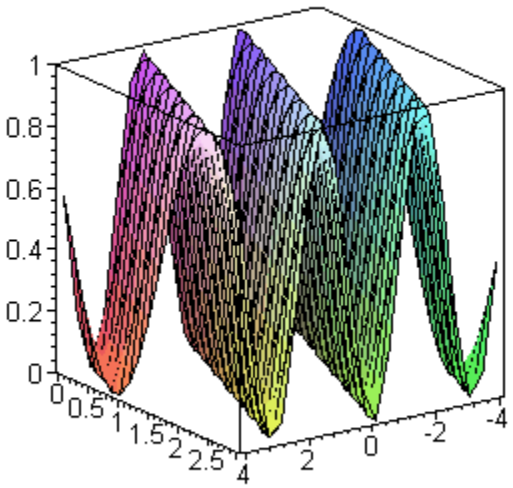
Choose your own function; just remember to make it nonnegative over the region of integration.

The only commands you need to enter in the following template are your function and minimum and maximum values for each of your variables (leave your **zmin** at 0). Then simply execute the rest of the worksheet, excluding the first cell that defined the function used above.

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?

We will begin by plotting the function and letting *Maple* estimate the volume between the surface and the *xy*-plane over the domain specified. Be sure to stay above the *xy*-plane.

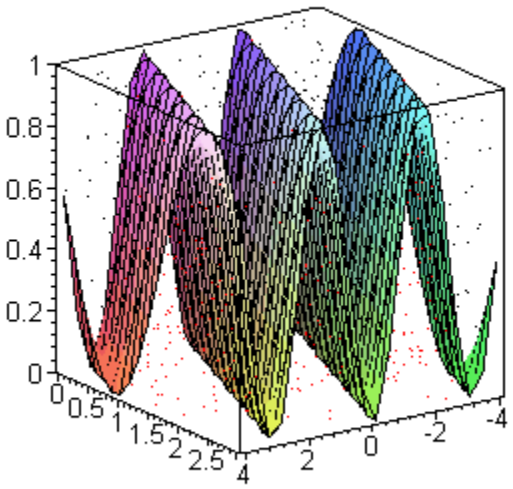
```
> unassign('x,y'):
  g:=(x,y)->sin(x-y)^2:
  xmin:=-4:
  xmax:=4:
  ymin:=0:
  ymax:=3:
  zmin:=0:
  zmax:=1:
  pf:=plot3d(g,xmin..xmax,ymin..ymax,view=zmin..zmax,axes=boxed,
  orientation=[-61,40],grid=[20,20],orientation=[55,72]):
  plots[display](pf);
  actualg:=evalf(Int(Int(g(x,y),x=xmin..xmax),y=ymin..ymax)):
  actual_volume := actualg;
```



$$actual\_volume := 12.06911051$$

```
> numberofpoints:=1000:
  volumeofbox:=(xmax-xmin)*(ymax-ymin)*(zmax-zmin):
  count:=0:
  listin:=[]:
  listout:=[]:
```

```
Random:=proc(a, b)
evalf(rand(ceil(a*10^Digits)..floor(b*10^Digits))()/10^Digits);
end:
for i from 1 to numberofpoints do
 x:=Random(xmin,xmax):
 y:=Random(ymin,ymax):
 z:=Random(zmin,zmax):
  if z < evalf(g(x,y)) then
    listin:=[op(listin),[x,y,z]]:
    count:= count + 1:
   else listout:=[op(listout),[x,y,z]]
  fi:
od:
pout:=plots[pointplot3d](listout,symbol=POINT,color=black):
pin:=plots[pointplot3d](listin,symbol=POINT,color=red):
plots[display](pin, pout, pf);
print(` actual = `, actualg);
estimate:= evalf(volumeofbox*count/numberofpoints):
print(` estimate = `, estimate);
Error:= estimate - actualg:
print(` error = `, Error);
relativeerror:= Error / actualg:
print(` relative error = `, relativeerror);
```



actual = , 12.06911051

estimate = , 12.

error = , -0.06911051

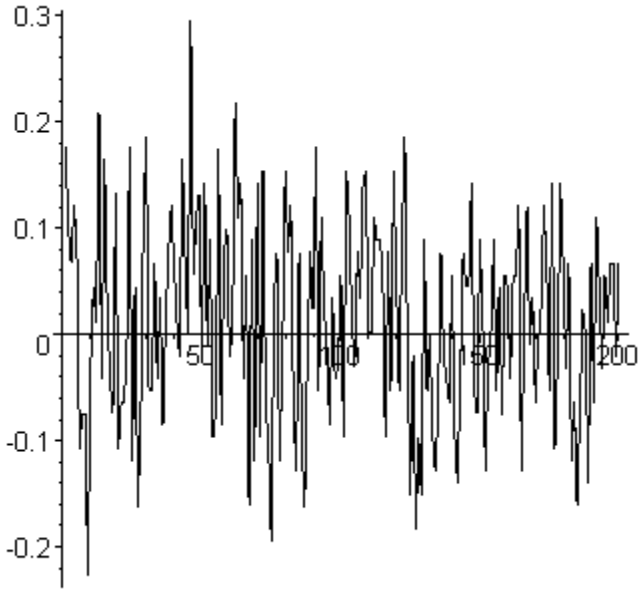
relative error = , -0.005726230607

Part II: Error Analysis

In this part, we follow the procedure above with 500 points; then, we repeat that procedure 200 times and look at the pattern of the errors. Because of the many steps involved, this computation may take several minutes.

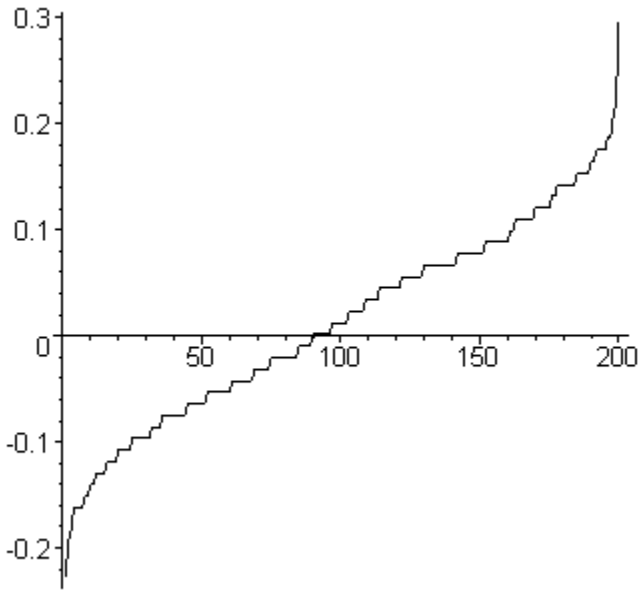
```
> plotsetup(gdi):
  symbol=POINT,
  unassign('x,y,z'):
  numberofpoints:=500:
  repeat:=200:
  xmin:=0:
  xmax:=1.5:
  ymin:=0:
  ymax:=2:
  zmin:=0:
  zmax:=1:
  count:=0:
  volumeofbox:=(xmax-xmin)*(ymax-ymin)*(zmax-zmin):
  errorlist:=[]:
  #actual:=evalf(Int(Int(f(x,y),x=xmin..xmax),y=ymin..ymax)):
  print(` actual = `, actualf);
  Random:=proc(a, b)
  evalf(rand(ceil(a*10^Digits)..floor(b*10^Digits))()/10^Digits);
  end:
  for i from 1 to repeat do
   count:=0;
   for j from 1 to numberofpoints do
    x:=Random(xmin,xmax):
    y:=Random(ymin,ymax):
    z:=Random(zmin,zmax):
    if z < evalf(f(x,y)) then count:= count + 1: fi:
   od:
    estimate:= evalf(volumeofbox*count/numberofpoints):
    Error:= estimate - actualf:
    errorlist:=[op(errorlist),Error]:
  od:
  percenterrorlist:=errorlist/actualf:
  plotsetup(default):
  plots[pointplot]([seq([i,percenterrorlist[i]],i=1..repeat)],connect=true);
```

$$actual = , 0.5512701925$$



Look at the pattern of your errors. These are referred to as random errors, and they should usually be centered around 0 and should show no particular pattern. We can order them and look at that picture as well.

```
> sortlist:=sort(percenterrorlist):
  plots[pointplot]([seq([i,sortlist[i]],i=1..repeat)],connect=true);
```



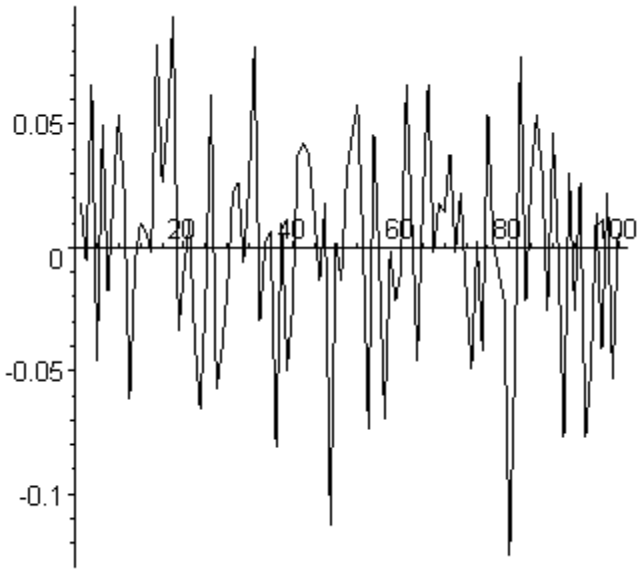
You should run this again, increasing the number of repeats and/or the number of points to see what happens to the errors. Which will decrease your error size?

You Try It: Part II

You can extend the problem you began in the previous **You Try It** section by analyzing your **errorlist** when you repeat the Monte Carlo method with 500 points being generated each of 100 times. Be patient.

```
> numberofpoints:=500:
  repeat:=100:
  g:=(x,y)->sin(x-y)^2:
  xmin:=-4:
  xmax:=4:
  ymin:=0:
  ymax:=3:
  zmin:=0:
  zmax:=1:
  volumeofbox:=(xmax-xmin)*(ymax-ymin)*(zmax-zmin):
  errorlist:=[]:
  print(` actual = `, actualg);
  count:=0:
  Random:=proc(a, b)
  evalf(rand(ceil(a*10^Digits)..floor(b*10^Digits))()/10^Digits);
  end:
  for i from 1 to repeat do
   count:=0:
   for j from 1 to numberofpoints do
    x:=Random(xmin,xmax):
    y:=Random(ymin,ymax):
    z:=Random(zmin,zmax):
    if z < evalf(g(x,y)) then count:= count + 1: fi:
   od:
    estimate:= evalf(volumeofbox*count/numberofpoints):
    Error:= estimate - actualg:
    errorlist:=[op(errorlist),Error]:
  od:
  percenterrorlist:= errorlist/actualg:
  plots[pointplot]([seq([i,percenterrorlist[i]],i=1..repeat)],
  connect=true);
  plotsetup(default):
```

$$actual = , 12.06911051$$



> ?

>

Are your errors centerd around 0? If not, there may be a mistake here.

See what happens to the range of your errors when you increase the **numberofpoints** command above.