Monte Carlo Techniques

Exercises

Abhishek Subramanian

May 12, 2015

Contents

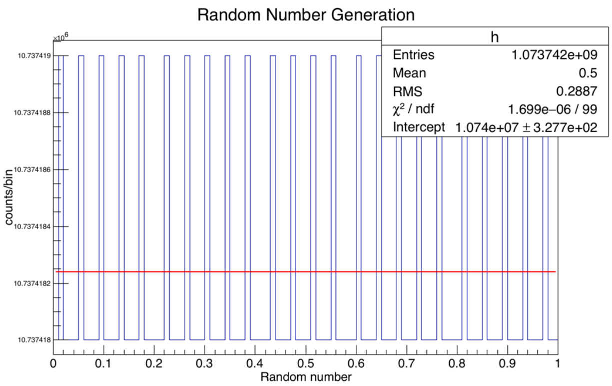
List of Figures

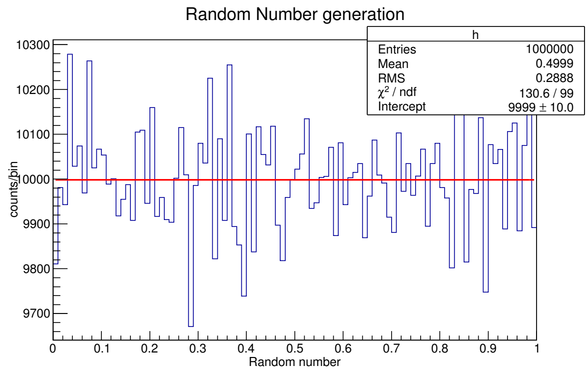
1 Uniform Random Sampling

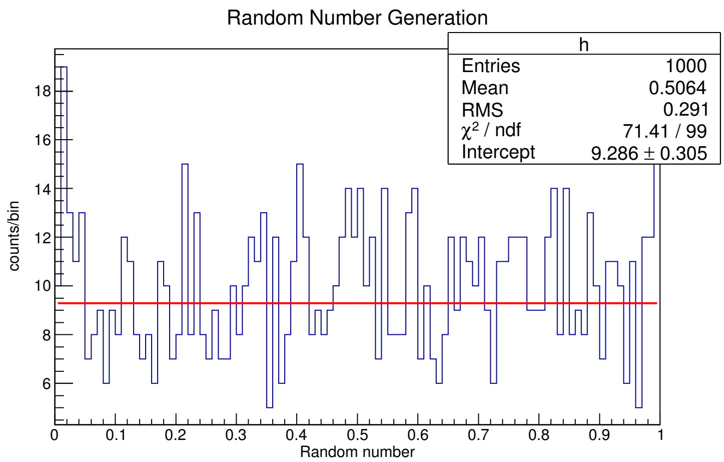
For this excercise pseudo random numbers were generated where the computer does the following

Initialization : i= 987654321
Iteration : i* 663608941

By performing the initialization and iteration it was verified that the length of the sequence is $2^{30} = 1073741824$.
χ^2 test was done to check the flatness of the distribution. The results of which are presented below.







From the plots above it is seen that the reduced $\chi^2 = \frac{\chi^2}{NDF}$ is less than one for the entire distribution and is of the order of 1 for histories of 1000 an 10^6 from the entire sequence. Thus the distribution is flat and the random number generator is uniform. The code[1] used for the iteration follows.

```
#include "TStopwatch.h"
#include "TMath.h"
#include "TCanvas.h"
#include "TF1.h"
#include "TH1.h"
#include "TStyle.h"

using namespace std;
void lcg()
{
  TStopwatch w; //Invoking the stopwatch
```

[1]The codes are run inside the root terminal

```
gStyle->SetOptFit(111);
/*
========================================================================
In the next three lines canvas is defined for the three histograms

========================================================================

*/
TCanvas *c1 =new TCanvas("c1","canvas",0,0,1080,864);
TCanvas *c2 = new TCanvas("c2","canvas",0,0,1080,864);
TCanvas *c3 = new TCanvas("c3","Bazinga",0,0,1080,864);
UInt_t i= 987654321; //initialization
Double_t norm=TMath::Power(2,32)-1;
Double_t j;
Int_t count=0;//setting counter for sequence length
TH1D *h=new TH1D("h", "Random Number Generation",100,0,1);//defining three
  histograms with bin width 100
TH1D *h1=new TH1D("h", "Random Number Generation",100,0,1);
TH1D *h2=new TH1D("h", "Random Number generation",100,0,1);
TF1 *f =new TF1("f","[0]*x+[1]"); //defining function to fit
f->SetParNames("slope","intercept"); //setting the name of parameters

w.Start();
/*
Iterations
*/
do
{
     i=i*663608941;
     j= i/norm;
  //cout<<y<<endl;
  //cout<<i<<endl;
  count++;
    h->Fill(j);
  if(count <=1000)
  {
    h1->Fill(j);
  }
  if(count <=1000000)
  {
    h2->Fill(j);
  }


  }while(i!=987654321);
w.Stop();
w.Print();

c1->cd();
h->Draw();
h->Fit("f","EM");
h->GetXaxis()->SetTitle("Random number");
h->GetXaxis()->CenterTitle();
h->GetYaxis()->SetTitle("counts/bin");
h->GetYaxis()->SetLabelFont(5);
h->GetYaxis()->CenterTitle();
c2->cd();
```

```
  h1->Draw();
  h1->GetXaxis()->SetTitle("Random number");
  h1->GetXaxis()->CenterTitle();
  h1->GetYaxis()->SetTitle("counts/bin");
  h1->GetYaxis()->CenterTitle();
  h1->Fit("f","EM");
  c3->cd();
  h2->Draw();
  h2->GetXaxis()->SetTitle("Random number");
  h2->GetXaxis()->CenterTitle();
  h2->GetYaxis()->SetTitle("counts/bin");
  h2->Getyaxis()->CenterTitle();
  h2->Fit("f","EM");

  cout<<"sequence length "<<count<<endl; //getting the length of the sequence
}
```

2 Random Sampling from Other Distributions

2.1 Inversion Method

Given is the following pdf

$$p(x) = \frac{2x}{(1+x^2)^2} \qquad 0 \le x < \infty \tag{1}$$

For the cdf to be in the range of 0 to 1, We require a normalised pdf. So we check if the pdf is normalised. To do so we integrate (1) within the given limits to check normalisation.

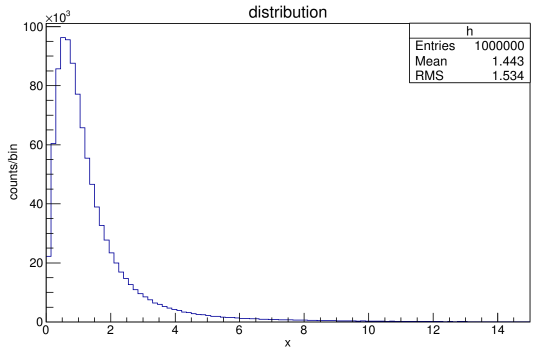
$$c(x) = \int_0^\infty \frac{2x}{(1+x^2)^2} dx = \left[-\frac{1}{t}\right]_1^\infty = 1 \tag{2}$$

$$\begin{aligned} \xi &= c(x) \\ x &= c^{-1}(\xi) \end{aligned} \tag{3}$$

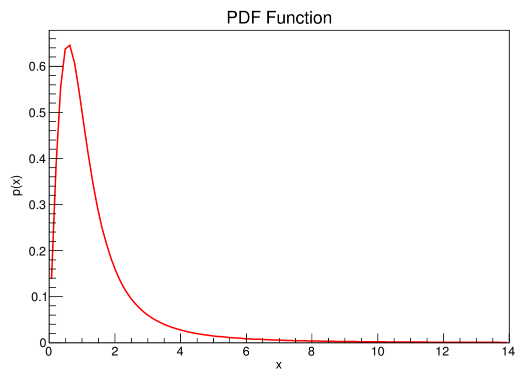
Integrating (1) and inverting to get x we have.

$$\sqrt{-1 - \frac{1}{\xi}} = x \tag{4}$$

Thus one can generate a random number ξ between -1 and 0 to see that x is distributed according to pdf. Building the function (4) by generating random numbers produces the following output.



(a) Figure showing the distribution of x according to the PDF function



(b) The pdf function

Figure 1: Plot showing distribution of x and the pdf function p(x)

2.2 Inversion and Rejection

A computer code which makes the random sampling (x,y) from a unit circle using inversion and rejection method was implemented. For inversion method the x and y values are sampled from the following equations.

$$
\begin{aligned}
x &= \sqrt{\xi_1}\cos(2\pi\xi_2) \\
y &= \sqrt{\xi_1}\sin(2\pi\xi_2)
\end{aligned} \tag{5}
$$

In the code for rejection method, the consideration of $2\pi\xi = \theta$ is made and θ is randomly generated in the range $[0,2\pi]$ so as to keep $\cos(\theta)$ in the range[-1,1]. Below is the code implemented for the rejection method.

```
#include "TStyle.h"
#include "TH1.h"
#include "TH2.h"
#include "TStopwatch.h"
#include "TF1.h"
#include "TCanvas.h"
#include "TMath.h"
#include "TRandom.h"
using namespace std;

void inv()
{
  TCanvas *c =new TCanvas("c","canvas",0,0,1080,860); //defining canvas to plot
    the outcome of the sampling.
  TStopwatch w; //invoking stopwatch

  TRandom3 generate; //invoking the random number generator
  Int_t i; // setting a counter for the for loop
  Double_t r1,r2; //declaring variables for the random numbers.
  TH2F *h =new TH2F("h","Inversion method",100,-1,1,100,-1,1); //defining a 2D
    histogram where

  w.Start(); //starting the stop watch
  for(i=0;i<1000000;i++)
    {
      r1=generate.Uniform(0,1.); //generating random number for r
      r2=generate.Uniform(0,TMath::TwoPi()); //generating random number for 2pi
      Double_t  x=TMath::Sqrt(r1)*TMath::Cos(r2); //using them in the formula
      Double_t y=TMath::Sqrt(r1)*TMath::Sin(r2);
      h->Fill(x,y); //filling the 2D histograms with the generated x and y

    }
  w.Stop(); //stopping the stopwatch
  w.Print(); //printing the output
  h->Draw(); //drawing the histogram
  /*
  =======================================================
  The next few lines are for naming the axis of the plot
  and changing the color of the marker
  =======================================================
  */
  h->GetXaxis()->SetTitle("x");
  h->GetYaxis()->SetTitle("y");
  h->GetXaxis()->CenterTitle();
```

```
  h->GetYaxis()->CenterTitle();
  h->SetMarkerColor(kBlue);

}
```

For the rejection method two random numbers are generated for x and y and the the condition to accept or reject is decided by the following equations and the condition given below.

$$
\begin{aligned}
x &= -1+2\xi_1 \\
y &= -1+2\xi_2 \\
(x^2 &+ y^2 < 1)
\end{aligned}
\tag{6}
$$

The code for the above was impleted and is given below

```
#include "TStyle.h"
#include "TH1.h"
#include "TH2.h"
#include "TStopwatch.h"
#include "TF1.h"
#include "TCanvas.h"
#include "TMath.h"
#include "TRandom.h"

void rej()
{
  TStopwatch t;//invoking stopwatch

  TRandom3 gen;//invoking the random number generator
  Int_t j=0,k=0,iN;


    TH2D *h = new TH2D("h","Rejection Method",100,-1,1,100,-1,1);
  TH1D *h2 = new TH1D("h2","Histogram",100,-1,1);
  t.Start();// starting the timer
  for(i=0;i<1000000;i++)// samplig a million times
    {
      Double_t r1=gen.Uniform(0,1);//generating random number for x
      Double_t r2=gen.Uniform(0,1);//generating random number for y
      Double_t x= -1+ 2*r1;//defining x
      Double_t y=-1+2*r2;//defining y
      if(x*x + y*y <=1)//checking the condition
        {
          h->Fill(x,y);//incrementing the counter and filling the histogram
          j++;
        }else
        {k++;
         }
      }

  t.Stop();//stopping the timer
  t.Print();//getting the output
  h->Draw();//drawing the histogram
  h->SetMarkerColor(kBlue);
}
```

For this particular case of sampling a unit circle using both the inversion and rejection methods, rejection method is much faster. The CPU time taken for rejectin method is 0.040 sec where as the inversion method takes 0.090 sec. In general it is better to have the fastest method to sample from a distribution. Hence even if a pdf is invertible it but the rejection method is much faster, rejection is preferred.

3 Numerical Estimationof π

To estimate the value of π by generating random numbers in a square, we can generate random numbers in the range [-1,1] along X and Y axis and then impose the condition that $x^2 + y^2 < 1$ for accepting or rejecting the point that was randomly shot into the square. We can then set two counters which count the accepted or rejected points. The value of π is then given by

$$\pi = 4 \frac{Number\ of\ accepted\ points}{Total\ number\ of\ points\ shot\ in\ the\ square} \tag{7}$$

By writing the computer code to implement that we see the fluctuation in the value of π and convergence towards actual value as the number of points shot in the square increases. It is observed that if 10000 points are shot in the square the number of accepted points is 7855.

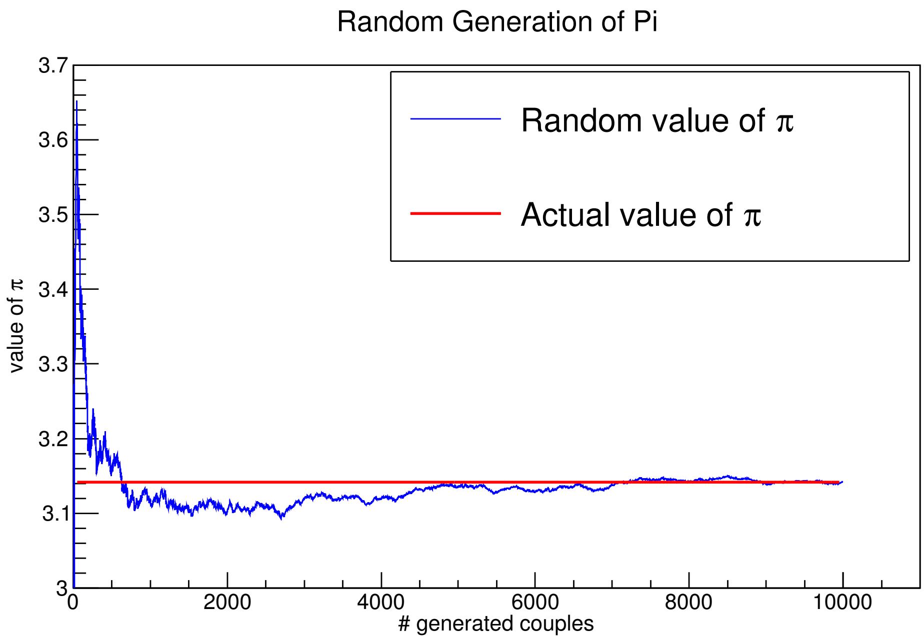


Figure 2: Graph showing the fluctuations in the value of π using the rejection method

from figure 2 it can be seen that the value of π fluctuates when the number of shot points is very low and starts converging when the generated couple crosses 6000. The code to used to study the fluctuations in the value of π is as follows.

```
#include "TGraph.h"
#include "TRandom.h"
#include "TStopwatch.h"
#include "TGraph.h"
#include "TAxis.h"
using namespace std;

void ex3()
{
```

```
  TCanvas *c =new TCanvas("c","Pi",0,0,1080,860);//creating a canvas to plot the
     fluctuations in pi

  TRandom3 gen;// invoking random number generator
  Double_t j=0,k=0; //setting counter j(counting points inside the circle) and k
    (counting points outside)
  Double_t pi;

  Int_t cnt,i,n;
  ofstream out; //pointing to ofstream
  out.open("file.txt");//opening a file named "file.txt"
  TF1 *f=new TF1("f","3.1415926535",0,10000); //using the actual value of Pi


    for(i=0;i<10000;i++)
      {
        Double_t r1=gen.Uniform(0,1);//generating random number for x
            Double_t r2=gen.Uniform(0,1);//generating random number for y
            Double_t x= -1+ 2*r1;//defining x
             Double_t y=-1+2*r2;//defining y


        if(x*x + y*y <1) //imposing condition and incrementing counter
          {
             n=j++;

            }
        else
          {
            k++;
            }
        pi= (j/(j+k))*4;
            out<<i<<"\t"<<pi<<std::endl;//writing the values of pi and number of
    couples to a file

        }




cout<<"number of points inside the circle is  "<<n<<endl; //counting number of
   points in the circle
out.close();
cout<<pi<<endl;// value of pi

c->cd(1);
TGraph *g= new TGraph("file.txt");//plotting the fluctuations in pi
g->Draw("AL");
g->SetTitle("Random Generation of Pi");

f->Draw("same");//drawing the function defined above in the same canvas to see
   the fluctuations

/*
The set of lines below is to change the color of line, set label to axis and to
   create legends.
```

```
*/
f->SetLineColor(kRed);
f->SetLineWidth(2);

g->GetYaxis()->SetRangeUser(3,3.7);
g->SetLineColor(kBlue);
g->GetXaxis()->SetTitle("# generated couples");
g->GetXaxis()->CenterTitle();
g->GetYaxis()->SetTitle("value of #pi");
g->GetYaxis()->CenterTitle();

leg = new TLegend(0.4,0.6,0.89,0.89);
leg->AddEntry(g,"Random value of #pi","l");
leg->AddEntry(f,"Actual value of #pi","l");
leg->Draw();

}
```

The value of π obtained using the rejection method is 3.1424.

3.1 Uncertainty Evaluation

To calculate the uncertainity in π the value of π is generated randomly many times using the rejectin method. As the error in this value scales of as $\frac{k}{\sqrt{N}}$ the value is calculated by fitting the generated points with a gausian. This process is repeated again several times and the value of k is obtained by plotting σ vs $\sqrt{N}$ and fitting with the standard deviation. The plot obtained is shown below.

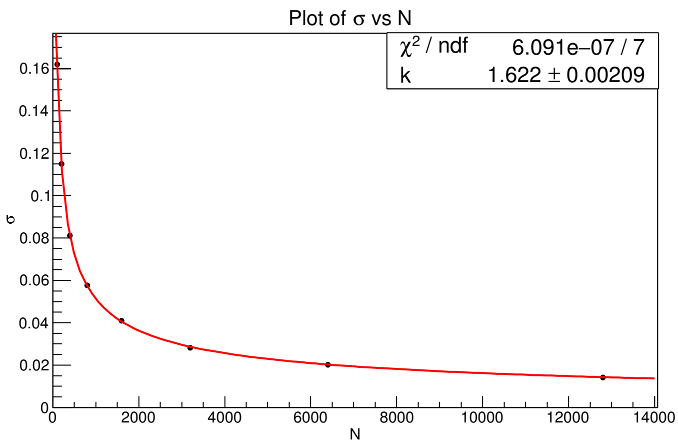


Figure 3: Graph of σ $vs\sqrt{N}$ showing that the points with N=1000 and 5000 and other points lie on the curve

From 3 it is observed that even as N increases the the σ values scale as $\frac{k}{\sqrt{N}}$. The code used for this purpose is presented in the page that follows.

```
#include "TGraph.h"
#include "TRandom3.h"
#include "TStopwatch.h"
#include "TGraph.h"
#include "TAxis.h"
#include "TF1.h"

void uncert()
{
  TCanvas *c =new TCanvas("c","Bazinga",0,0,1080,800);
  Double_t pi, k=0,m=0;
  TRandom3 gen;
  gStyle->SetOptFit(011);
  Int_t i, j, N=100;
  Double_t  sndv=100;
  TH1F *h = new TH1F("h","My histogram", 60000,0,6);
  TF1 *f=new TF1("f","gaus");
  TF1 *f1=new TF1("f1","[0]/TMath::Sqrt(x)");
  Double_t u,a,b;
  ofstream out;

  out.open("un.txt");
  for(int q=0;q<8;q++)
    {h->Reset();//resetting the histogram
      for(j=0;j<10000;j++)
  {k=0;//initializing k=0 in every run
    for(i=0;i<sndv;i++)
      {
        Double_t r1=gen.Uniform(0,1);//generating random number for x
        Double_t r2=gen.Uniform(0,1);//generating random number for y
        Double_t x= -1+ 2*r1;//defining x
        Double_t y=-1+2*r2;//defining y

        if(x*x+y*y<1)//rejection method to calculate pi
    {
      k++;
    }

        }
    pi=(k/sndv)*4;
    h->Fill(pi);
  }



      h->Fit("f");
      b=f->GetParameter(2);
      out<< sndv<< "\t"<< b<<endl;
      sndv*=2;
    }

out.close();
f1->SetParNames("k");
TGraph *g=new TGraph("un.txt");
g->SetMarkerStyle(20);
g->SetTitle("Plot of #sigma vs N");
g->Draw("AP");
g->Fit("f1");
```

```
g->GetXaxis()->SetTitle("N");
g->GetXaxis()->CenterTitle();
g->GetYaxis()->SetTitle("#sigma");
g->GetYaxis()->CenterTitle();
}
```

The estimated value of k is 1.622 thus for the estimate of pi to be better than 10^{-4} the number of couples should be atleast 10^8. Thus with $N = 10^8$ we have

$$\sigma = \frac{1.622}{\sqrt{10^8}} = 1.6448\ X\ 10^{-4} \tag{8}$$

4 Monte Carlo Integration

4.1 Unidimensional Integration

The motive of this question is to solve the following one dimensional integration by monte carlo method.

$$I_n = \int_0^1 x^n dx \tag{9}$$

In general to solution to a integral by monte carlo integral is obtained by sampling points randomly in the range of the integration and the value of the integration is given by.

$$I = \frac{V_\Omega}{N} \sum f(x) = V_\Omega \langle f(x) \rangle \tag{10}$$

In our case $V_\Omega = b - 1 = 1 - 0 = 1$ Thus the integration is just the average of the function f(x). To implement this integration the following code is used.

```
#include "TGraph.h"
#include "TRandom3.h"
#include "TStopwatch.h"
#include "TGraph.h"
#include "TAxis.h"
#include "TF1.h"

void ex41()
{

TRandom3 gen;//invoking the random number generator
Double_t sum=0;//declariation of sum and integration

Int_t i,num,N=1000000;//declaring variables for loop and dimension and number of
    sampling points
cout<<"enter a number from 1 to 5"<<endl;

cin>>num;
  for(i=0;i<N;i++)
  {
      Double_t rando=gen.Uniform(0,1);//generating random number in the range of
    integral


                      Double_t x=rando; //declaring x as raom
        Double_t y=TMath::Power(x,num);//declaring y as integral value of x with
    user entered power num
        sum+= y;
  }

cout<<"Value of the integration is   " << sum/N <<endl;
}
```

The result of the integration by monte carlo is compared with the actual vaue of the integral in the table that follows. Also $\frac{\Delta I}{I}$ is calculated for each D by using the following loop.

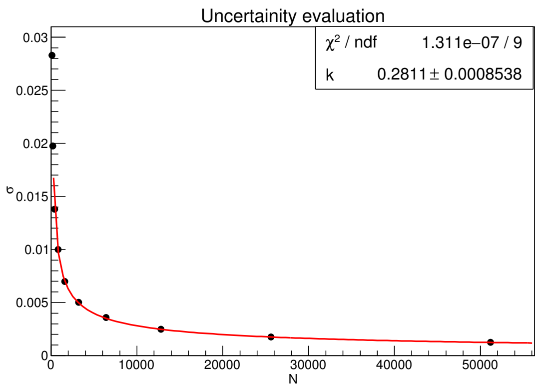
```
for(q=0;q<10;q++)
  {//for the q

    h->Reset();

     for(int j=0;j<10000;j++)
  {  //for the j
    sum=0;
    for(i=0;i<N;i++)
    {//for the i
        Double_t rando=gen.Uniform(0,1);//generating random number in the range
   of integral


                          Double_t x=rando;  //declaring x as raom
          Double_t y=TMath::Power(x,num);//declaring y as integral value of x
   with user entered power num
          sum+= y;
    }//for the i
    mean=sum/N;
    h->Fill(mean);
    }//for the j
h->Fit("f");
b=f->GetParameter(2);
out<<N<<"\t"<<b<<endl;
N*=2;
}//for q
```

As the $\frac{\Delta I}{I} \propto \frac{1}{\sqrt{N}}$ for monte carlo the value of proportionality constant is obtained by by plotting the $\sigma\ vsN$ and fitting it with $\frac{k}{\sqrt{N}}$. The two plots below show the same for D=1 and 2.



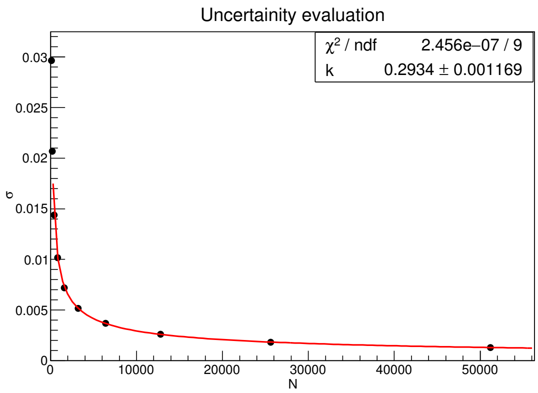


Figure 4: $\sigma\ vs\sqrt{N}$ for the integrals with D=1 and D=2

The table below gives the values of k the relative error along with the values of monte carlo integrals

D	I_{MC}	$I_{analytic}$	k	$\frac{\Delta I}{I}$
1	0.499867	0.5	0.2811	0.0002811
2	0.333152	0.333333	0.2934	0.0002934
3	0.249821	0.25	0.2785	0.0002785
4	0.199839	0.2	0.2595	0.0002595
5	0.166526	0.166666	0.2411	0.0002411

Table 1: Comparing the monte carlo integral to the true value $I_{analytic} = \frac{1}{n+1}$. Also the relative error of monte carlo is presented

The convergence of monte carlo integral goes with central limit theorem as $\frac{1}{\sqrt{N}}$. Thus to have a precision of at least 0.001 the number of sampling points should be 10^6. The results mentioned above have a precision of 0.001 as the number of sampling points used is 10^6.

4.2 Integration in N dimensions

This integration is performed in monte carlo and mid point integral. The idea of the montecarlo for one dimension can be extended to D dimensions by sampling within the range of these D dimensions and finding the average of these functions. The function to be integrated in D dimensions is given by.

$$I = \int_0^1 \int_0^1 \dots \int_0^1 \left(x_1^2 + x_2^2 + x_3^2 + \cdots + x_D^2\right) dx^D \\ D = 1,2\dots 8 \tag{11}$$

To perform the integral of the 8 dimensional function using the monte carlo method we sample 8 random numbers in the range [0,1] an million times and find the average of the function. The code illustrated below gives the value of the integral in 8 dimensions usin monte carlo method for the function in (11).

```
#include "TGraph.h"
#include "TRandom.h"
#include "TStopwatch.h"
#include "TGraph.h"
#include "TAxis.h"
#include "TMath.h"
using namespace std;
/*
=============================================================================

In the First part of the code the various functions are defined
in the second part the process of integration using monte carlo method is
   followed.

=============================================================================

The function to be integrated is a D dimensional function where D=1,...,8, Using
    the monte carlo method
*/


double func1(double x1)
{
Double_t y = TMath::Power(x1,2);
```

```
return y;
}


double func2(double x2,double x1)
{
Double_t y= TMath::Power(x2,2)+ func1(x1);
return y;
}
double func3(double x2,double x1,double x3)
{
Double_t y= func2(x1,x2)+TMath::Power(x3,2);
return y;
}


double func4(double x2,double x1,double x3,double x4)
{
Double_t y= func3(x1,x2,x3)+TMath::Power(x4,2);
return y;
}


double func5(double x2,double x1,double x3,double x4,double x5)
{
Double_t y= func4(x1,x2,x3,x4)+TMath::Power(x5,2);
return y;
}

double func6(double x2,double x1,double x3,double x4,double x5,double x6)
{
Double_t y= func5(x1,x2,x3,x4,x5)+TMath::Power(x6,2);
return y;
}

double func7(double x2,double x1,double x3,double x4,double x5,double x6,double
    x7)
{
Double_t y= func6(x1,x2,x3,x4,x5,x6) +TMath::Power(x7,2);
return y;
}

double func8(double x2,double x1,double x3,double x4,double x5,double x6,double
    x7,double x8)
{
Double_t y= func7(x1,x2,x3,x4,x5,x6,x7)+TMath::Power(x8,2);
return y;
}


/*
============================================================
code for integration starts below.Here numbers are generated randomly from  0 to
     1
=============================================================
*/
void ex4()
{
```

```
TRandom3 gen;
Double_t sum=0,mean,n=1000000;
for(Int_t i=0;i<n;i++)
  {
       Double_t x=gen.Uniform(0,1); // random number for x1
       Double_t t= gen.Uniform(0,1);// random number for x2
       Double_t a= gen.Uniform(0,1);//random number for x3
       Double_t b=gen.Uniform(0,1);//random number of x4
       Double_t c=gen.Uniform(0,1);//random number for x5
       Double_t d=gen.Uniform(0,1);//random number for x6
       Double_t e=gen.Uniform(0,1);//random number for x7
       Double_t f=gen.Uniform(0,1);//random number for x8
       Double_t z= func8(x,t,a,b,c,d,e,f);
       sum+=z;


  }

cout<<"Value of Integration is   "<<(sum/n)<<endl;
}
```

The error as mentioned in previous section scales of as $\frac{1}{\sqrt{N}}$ where N is the number of sampled points. For the above mentioned code the number of times the random numbers were generated in the given region is 10^6 thus the integral is precise upto 0.001. The value of the 8 dimensional integration is 2.66701

Mid point integral

To evaluate the integral

$$I = \int_a^b f(x)dx \tag{12}$$

The function is divided into a large number of rectangles of width h=$\frac{b-1}{N}$ where N is the number of rectangles. the value of the integration can then be approximated as

$$I \approx h \sum_{n=0}^{N-1} f(x_n) \tag{13}$$

Here $x_n = a + nh$. For N dimensions the space is divided into mesh of $\sqrt{N}\ X\ \sqrt{N}\ X \ldots\ X\ \sqrt{N}$. The value of the function to be integrated is found at the midpoints of these meshes and sumed. The code used to calculate the value of integration in dimensions D=1,..., 8 is shown below. The function part of the code is same as the one for monte carlo integration. Only the loops used to calculate the value of the integration is shown here.

```
void mid()
{

Int_t D;

  for(D=1; D<9; D++)
  {

     if(D==1)

{
```

```
        Double_t N=65536, n=65536;
        Double_t dx=1/n;
        Double_t sum=0,vmid;
         for(int i=0; i<n;i++)




       {

             Double_t x= dx/2+ (i/n);


            vmid= func1(x);
           sum+=vmid/N;
       }

cout<<"Value of Integral in 1 dimensions is  "<<sum<<endl;}
     if(D==2)
     {

        Double_t N=65536, n=256;
        Double_t dx=1/n;
        Double_t sum=0,vmid;
         for(int i=0; i<n;i++)
       for (int j=0;j<n;j++)



       {

             Double_t x= dx/2+ (i/n);
          Double_t two= dx/2+(j/n);

            vmid= func2(x,two);
           sum+=vmid/N;
       }

cout<<"Value of Integral in 2 dimensions is  "<<sum<<endl;}
     if(D==3)
     {

        Double_t N=64000, n=40;
        Double_t dx=1/n;
        Double_t sum=0,vmid;
         for(int i=0; i<n;i++)
       for (int j=0;j<n;j++)
       for(int k=0;k<n;k++)


       {

             Double_t x= dx/2+ (i/n);
          Double_t two= dx/2+(j/n);
          Double_t three=dx/2+(k/n);

            vmid= func3(x,two,three);
```

```
        sum+=vmid/N;
     }

cout<<"Value of Integral in 3 dimensions is  "<<sum<<endl;}
    if(D==4)
    {

       Double_t N=65536, n=16;
       Double_t dx=1/n;
       Double_t sum=0,vmid;
       for(int i=0; i<n;i++)
    for (int j=0;j<n;j++)
    for(int k=0;k<n;k++)
    for(int l=0;l<n;l++)

      {

            Double_t x= dx/2+ (i/n);
          Double_t two= dx/2+(j/n);
          Double_t three=dx/2+(k/n);
          Double_t four=dx/2+(l/n);
          //Double_t five=dx/2+(m/16);
         // Double_t six=dx/2+(m/n);
           vmid= func4(x,two,three,four);
          sum+=vmid/N;
      }

cout<<"Value of Integral in 4 dimensions is  "<<sum<<endl;}

    if(D==5)
{
     Double_t N=59049,n=9;
    Double_t sum=0,vmid,dx=1/n;

    for(int i=0; i<n;i++)
    for (int j=0;j<n;j++)
    for(int k=0;k<n;k++)
    for(int l=0;l<n;l++)
    for(int m=0;m<n;m++)

      {

            Double_t x= dx/2+ (i/n);
          Double_t two= dx/2+(j/n);
          Double_t three=dx/2+(k/n);
          Double_t four=dx/2+(l/n);
          Double_t five=dx/2+(m/n);
          Double_t six=dx/2+(m/n);
           vmid= func5(x,two,three,four,five);
          sum+=vmid/N;
      }

cout<<"Value of Integral in 5 dimensions is   "<<sum<<endl;}


if(D==6)
{
     Double_t N=46656,n=6;
```

```
    Double_t sum=0,vmid,dx=1/n;

    for(int i=0; i<n;i++)
    for (int j=0;j<n;j++)
    for(int k=0;k<n;k++)
    for(int l=0;l<n;l++)
    for(int m=0;m<n;m++)
    for(int p=0;p<n;p++)
      {

           Double_t x= dx/2+ (i/n);
         Double_t two= dx/2+(j/n);
         Double_t three=dx/2+(k/n);
         Double_t four=dx/2+(l/n);
         Double_t five=dx/2+(m/n);
         Double_t six=dx/2+(m/n);
          vmid= func6(x,two,three,four,five,six);
         sum+=vmid/N;
      }

cout<<"Value of Integral in 6 dimensions is   "<< sum<<endl;}
if(D==7)
{
     Double_t N=78125,n=5;
    Double_t sum=0,vmid,dx=1/n;

    for(int i=0; i<n;i++)
    for (int j=0;j<n;j++)
    for(int k=0;k<n;k++)
    for(int l=0;l<n;l++)
    for(int m=0;m<n;m++)
    for(int p=0;p<n;p++)
    for(int q=0;q<n;q++)
    {

           Double_t x= dx/2+ (i/n);
         Double_t two= dx/2+(j/n);
         Double_t three=dx/2+(k/n);
         Double_t four=dx/2+(l/n);
         Double_t five=dx/2+(m/n);
         Double_t six=dx/2+(p/n);
         Double_t seven=dx/2+(q/n);
          vmid= func7(x,two,three,four,five,six,seven);
         sum+=vmid/N;
      }

cout<<"Value of Integral in 7 dimensions is   "<< sum<<endl;}

if(D==8)
{
     Double_t N=65516,n=4;
    Double_t sum=0,vmid,dx=1/n;

    for(int i=0; i<n;i++)
    for (int j=0;j<n;j++)
    for(int k=0;k<n;k++)
    for(int l=0;l<n;l++)
    for(int m=0;m<n;m++)
```

```
    for(int p=0;p<n;p++)
    for(int q=0;q<n;q++)
    for(int r=0; r<n;r++)
    {

          Double_t x= dx/2+ (i/n);
        Double_t two= dx/2+(j/n);
        Double_t three=dx/2+(k/n);
        Double_t four=dx/2+(l/n);
        Double_t five=dx/2+(m/n);
        Double_t six=dx/2+(p/n);
        Double_t seven=dx/2+(q/n);
        Double_t eight =dx/2+(r/n);
         vmid= func8(x,two,three ,four ,five ,six ,seven ,eight);
        sum+=vmid/N;
      }

cout<<"Value of Integral in 8 dimensions is   "<< sum<<endl;}
}
}
```

The results obtained from both the methods are compared in the table below.

D	I=$\frac{D}{3}$	I_{mp}	I_{MC}
1	0.33333	0.33333	0.33271
2	0.66666	0.66666	0.6660
3	1	0.999844	1.00016
4	1.33333	1.3320	1.33313
5	1.66	1.6615	1.66677
6	2	1.98611	2.000
7	2.3333	2.31	2.33331
8	2.6666	2.6258	2.6670

Table 2: Comparing the Analalytical value of Integral with monte carlo and midpoint summation method

By comparing the monte carlo and midpoint summation method to calculate the integral in the table 2 it is clearly seen that monte carlo is more precise as one increases the dimension of the integral. The convergence of integral in the midpoint integral is.

$$\frac{\Delta I}{I} \propto \frac{1}{N^{\frac{2}{D}}} \tag{14}$$

So for lower dimensions the convergence is faster but as the dimension increases the convergence is slower than the monte carlo for which the convergence is $\frac{1}{\sqrt{N}}$ where N is the number of sampling points for monte carlo and the number of cells in the midpoint integral method. In general montecarlo is more precise than midpoint integral because it can achieve a better precision running for an equal time as midpoint summation method.

4.3 Extra Exercise

For the purpose of integrating the following function using monte carlo and midpoint integral the code described in the previous section is used the only difference being change of function in various dimensions. The function to be integrated is

$$I = \prod_{i=1}^{D} \int_0^1 \exp(-x_i) dx_i \tag{15}$$

Where D=1...8. The function is integrated using both monte carlo and midpoint methods and the comparison with the analytical value gives us the following.

D	I=$\left(1-\frac{1}{e}\right)^D$	I_{mp}	I_{MC}
1	0.632120	0.632121	0.632463
2	0.399576	0.399576	0.399724
3	0.252580	0.252561	0.252548
4	0.159661	0.159557	0.159718
5	0.1009251	0.100666	0.10093
6	0.0637968	0.068390	0.0637868
7	0.04032732	0.039859	0.0403427
8	0.02549173	0.024974	0.0254906

Table 3: Comparing the result of monte carlo with midpoint integral for the function (15)

Here also it is seen that the monte carlo integral becomes more efficient as the dimensions increase.

5 Tracking algorithms

In Monte Carlo the track of the particle is seen as random sequence of free flights which ends with an interaction. In this exercise we have to verify that the path lenght employed by geant4 and penelope are equivalent. To do so, individual step lengths s_i are sampled from $\mu_i\, e^{-\mu_i s_i}$. Using the code below this is done in brute force method. Also the the sampling fraction is done in using monte carlo.

```
#include "TGraph.h"
#include "TRandom.h"
#include "TStopwatch.h"
#include "TGraph.h"
#include "TAxis.h"
#include "TMath.h"
using namespace std;
/*
=========================
Defining various functions

=========================

*/
double func1(double x1)
{
Double_t y= -TMath::Log(x1);

return y;
}

double func2(double x2)
{
Double_t y=-0.5*(TMath::Log(x2));
return y;
}
double func3(double x3)
{
double y= -(1/3)*TMath::Log(x3);
return y;
}
//Main code begins
void tracking()
{
gStyle->SetOptFit(111);
TRandom3 gen;
Double_t s1, s2;
Double_t k=0,j=0,s; //defining counters to compute samping fraction of process
TF1 *f =new TF1("f","[0]*TMath::Exp(-[1]*x)"); //defining functin for fit
TH1F *h=new TH1F("h","My histogram",100,0,3);// defining a histogram

//sampling a million times
  for(int i=0;i<1000000;i++)
  {
    Double_t y1=gen.Uniform(0,1);

    Double_t y2=gen.Uniform(0,1);

    s1=func1(y1);
    s2=func2(y2);
```

```
            s=TMath::Min(s1,s2);
     if(s==s1)
     {
        k++;
       }
     else if(s==s2)
       {
             j++;
          }
    h->Fill(s);
  }
h->Draw();
h->Fit("f","EM");
cout<<"Probability of interaction for #mu = 2   "<< j/(j+k) <<"\t" << "value of
    interaction for #mu=1  "<<k/(j+k)<<endl;
h->GetXaxis()->SetTitle("s");
h->GetXaxis()->CenterTitle();
h->GetYaxis()->SetTitle("counts/bin");
h->GetYaxis()->CenterTitle();
h->GetYaxis()->SetTitleOffset(1.4);
}
```

When the distribution of randomly generated paths is fit with $\mu e^{-\mu s}$ it is observed that $\mu = \sum_i \mu_i = \mu_1 + \mu_2 = 3$. The distribution which is built is shown below.

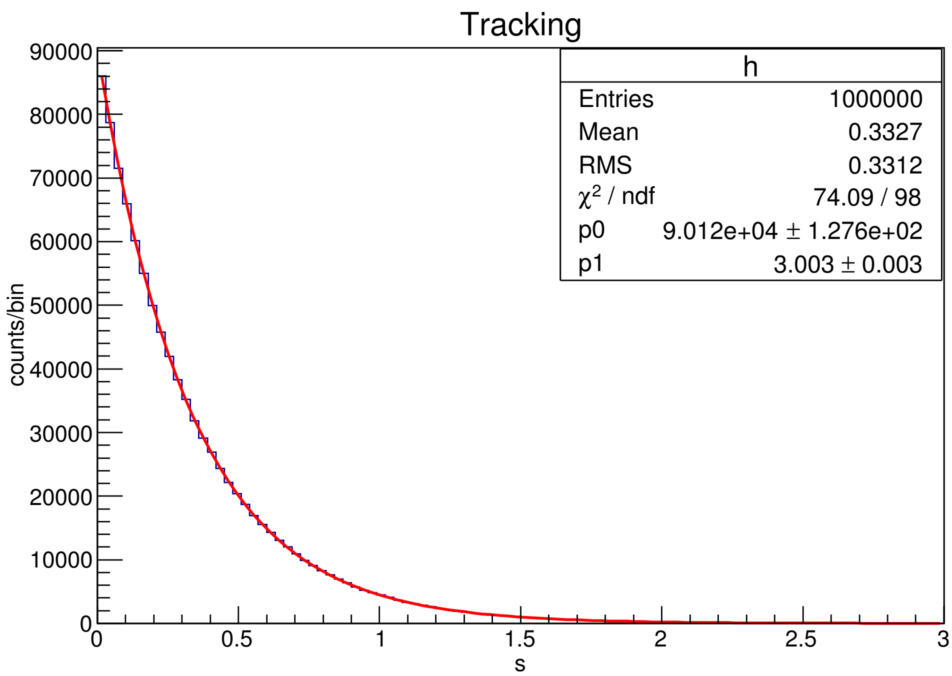


Figure 5: Histogram showing the sampling length

2

Consider a infinite homogeneous medium where the total attenuation coefficient is $\mu_1 + \mu_2 = \mu$ .Here μ is the interaction channel of two competing processes. To build the probability distribution p(s) starting from $s_1 \& s_2$ we know that

$$p(s) = p(s_1)P(s < s_2) + p(s_2)P(s < s_1) \quad (16)$$

Where we can write the

$$\begin{aligned} P(s < s_2) &= \mu_2 \int_s^\infty \exp(-\mu_2 s_2) ds_2 \\ P(s < s1) &= \mu_1 \int_s^\infty \exp(-\mu_1 s_1) ds_1 \end{aligned} \tag{17}$$

Thus (1) is writte as.

$$\begin{aligned} p(s) &= \mu_1 \exp(-\mu_1 s)\mu_2 \int_s^\infty \exp(-\mu_2 s_2) ds_2 + \mu_2 \exp(-\mu_2 s)\mu_1 \int_s^\infty \exp(-\mu_1 s_1) ds_1 \\ &= (\mu_1 + \mu_2) \exp(-\mu_1 - \mu_2)s = \mu \exp(-\mu s) \\ \mu_i &= \mu_1 + \mu_2 \end{aligned} \tag{18}$$

The above can be generalised to N dimensions as follows. where the probabilities are defined similar to (17)

$$\begin{aligned} p(s) &= p(s_1)P(s < s_2)\dots P(s < s_N) + p(s_2)P(s < s_1)\cdots + p(s_N)\prod_{i=1}^{N-1} P(s < s_i) \\ &= \mu_1 e^{-\mu_1 s}\, e^{-\mu_2 s}\, e^{-\mu_3 s}\dots e^{-\mu_N s} + \mu_2 e^{-\mu_1 s}\, e^{-\mu_3 s} e^{-\mu_4 s}\dots e^{-\mu_N s} + \cdots + \mu_N \prod_{i=1}^{N-1} e^{-\mu_i s} \\ &= (\mu_1 + \mu_2 + \cdots + \mu_N)\, e^{-(\mu_1+\mu_2+\mu_3+\cdots+\mu_N)s} \\ p(s) &= \sum_i \mu_i \exp(-\mu_i) s \end{aligned} \tag{19}$$

Thus p(s) can be written

$$p(s) = \mu\ e^{-\mu s} \tag{20}$$

Where,
$\mu \equiv \mu_1 + \mu_2 + \mu_3 + \dots + \mu_N$