Monte Carlo Lectures by Prof Moskal

Satyam Tiwari
satyam.tiwari@doctoral.uj.edu.pl

Contents

Module 1: Mathematical Foundations

This module explores fundamental Monte Carlo techniques for generating random variables according to specific probability distributions.

A) Sampling (Accept-Rejection Sampling)

1. Concept: Standard vs. Custom Distributions

Monte Carlo simulations rely on generating random numbers that follow specific patterns.

- **Standard Distributions:** Common patterns like the **Normal (Gaussian)** distribution (bell curve) and **Poisson** distribution (counting discrete events) are often built into libraries.
- **Accept-Rejection Method:** This is a powerful algorithm used when we have a custom Probability Density Function (PDF), f(x), that is difficult to invert directly. We generate a random point in a bounding box; if the point falls below the curve f(x), we "accept" it as a valid sample.

2. Implementation Details The code first benchmarks standard generators using TRandom3 to create a Normal distribution (mu=0, sigma=1) and a Poisson distribution (mu=5).

For the Custom Distribution, the code defines a linear PDF:

$$f(E) = 1.0 - \frac{E}{340.0}$$

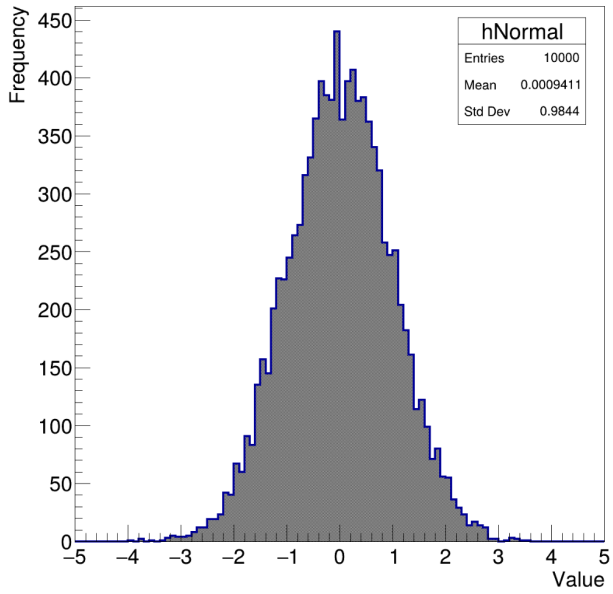
The accept-reject loop generates a candidate energy E and a test value y. The sample is accepted only if y < f(E).

- Sampling from Normal & Poisson Distribution

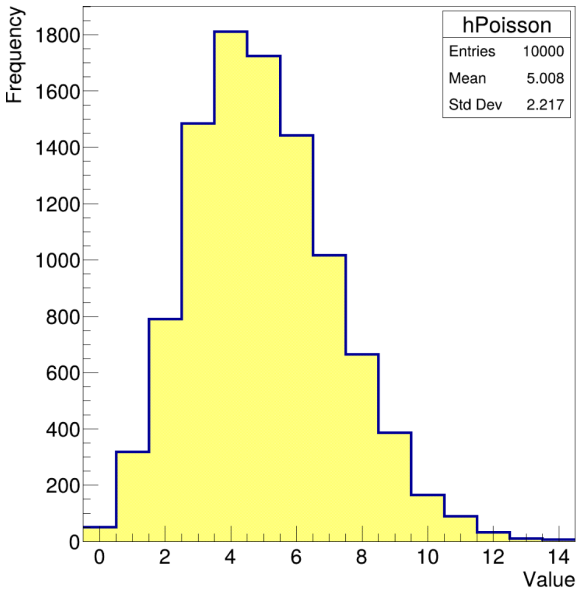
```
void Part1A_Sampling_from_Normal_and_Poisson_Distribution() {

    TRandom3 *randGen = new TRandom3(0);

    const int nSamples = 10000;

    double normalMean = 0.0;
    double normalStdDev = 1.0;
    double poissonMean = 5.0;

    TH1F *hNormal = new TH1F("hNormal", "Normal
```

```
Distribution;Value;Frequency", 100, -5, 5);
    TH1F *hPoisson = new TH1F("hPoisson", "Poisson
Distribution;Value;Frequency", 15, -0.5, 14.5);

    for(int i = 0; i < nSamples; ++i) {
        hNormal->Fill(randGen->Gaus(normalMean, normalStdDev));
        hPoisson->Fill(randGen->Poisson(poissonMean));
    }

    TCanvas *c1 = new TCanvas("c1", "Sampling from Distributions", 1200,
600);
    c1->Divide(2,1);

    c1->cd(1);
    hNormal->Draw();
    hNormal->Fit("gaus");

    c1->cd(2);
    hPoisson->Draw();

    hNormal->SetStats(1);
    hPoisson->SetStats(1);
}
```

Normal Distribution

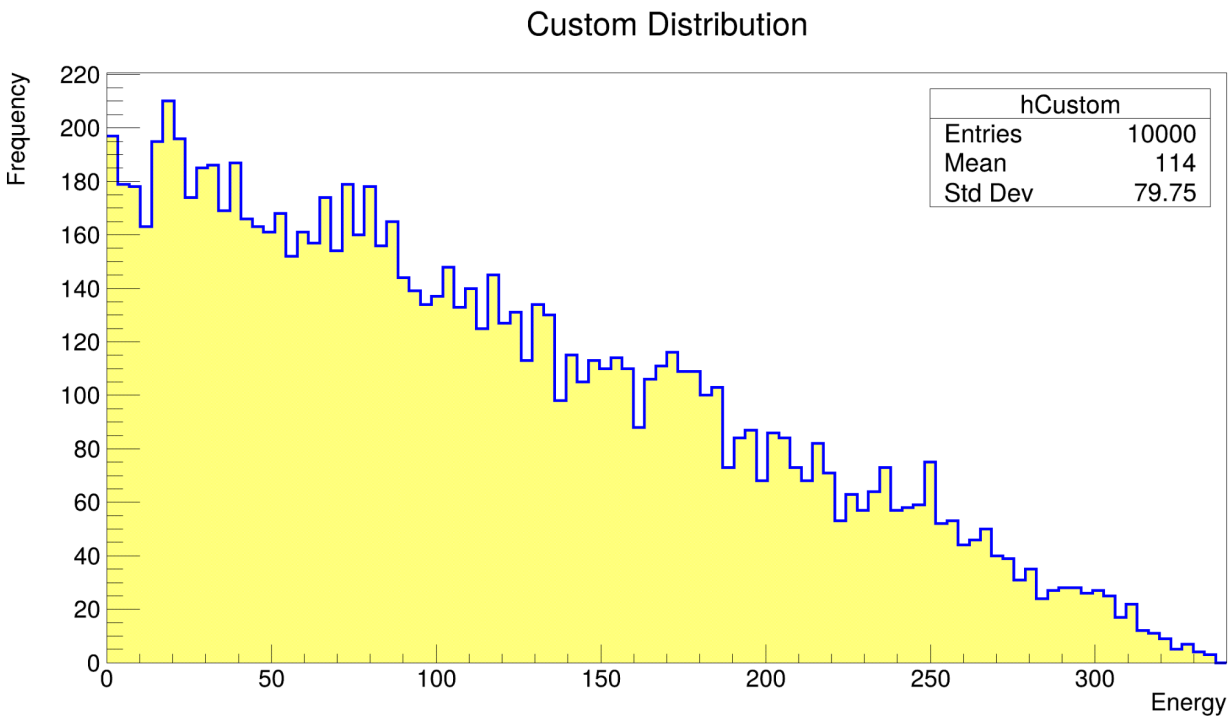


Poisson Distribution



- Sampling from Custom Distribution

```
float f(float E){
    return 1.0 - E/340.0;
}

float accept_reject(TRandom3* randGen, float Emax, float fmax){
    while(true){
        float E = randGen->Uniform(0, Emax);
        float y = randGen->Uniform(0, fmax);
        if(y < f(E)){
            return E;
        }
    }
}

void Part1B_Sampling_from_Custom_Distribution() {

    TRandom3 *randGen = new TRandom3(0);

    const int nSamples = 10000;
    const float Emax = 340.0;
    const float fmax = 1.0;

    TH1F *hCustom = new TH1F("hCustom", "Custom
Distribution;Energy;Frequency", 100, 0, Emax);

    for(int i = 0; i < nSamples; ++i) {
        float sample = accept_reject(randGen, Emax, fmax);
        hCustom->Fill(sample);
    }

    // hCustom->Scale(1.0 / (hCustom->Integral("width"))); // Normalize to
PDF

    TCanvas* c1 = new TCanvas("c1", "Sampling from Custom Distribution",
800, 600);
    hCustom->SetLineColor(kBlue);
    hCustom->SetLineWidth(2);
    hCustom->Draw("HIST");

}
```



3. Analysis of Results

- **Normal/Poisson:** The histograms validate the generator. The Normal plot shows a perfect bell curve centered at 0, and the Poisson plot shows the characteristic discrete steps centered near 5.
- **Custom:** The histogram hCustom clearly displays the triangular shape defined by the function, starting high at E=0 and linearly decreasing to 0 near E=340, confirming the algorithm works.

B) Exponential Distribution (Inverse Transform Sampling)

1. Concept: Inverse Transform Sampling

When the Cumulative Distribution Function (CDF) of a distribution can be mathematically inverted, we can generate samples directly from a uniform distribution.

For an exponential distribution (common in radioactive decay), the inversion formula used is:

$$x = -\frac{\ln(1-u)}{\lambda}$$

where u is a uniform random number between 0 and 1.

2. Implementation Details

The code implements this formula with a decay constant lambda = 0.5.

3. Memoryless Property

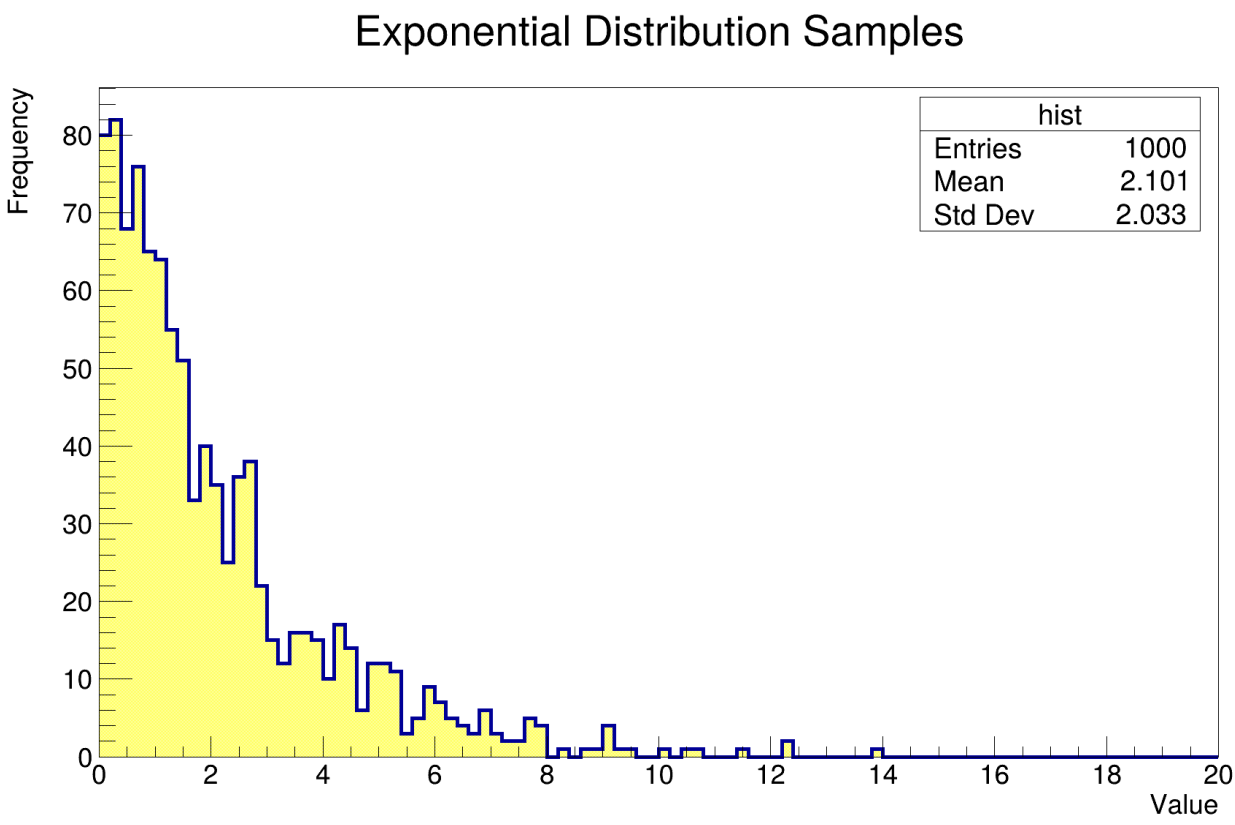
The exponential distribution is unique because it is "memoryless." This means the probability of an event occurring in the next instant is independent of how much time has already passed.

- **The Experiment:** The code simulates two scenarios:
    - **Memoryless:** A particle has already waited time t=2.0. We calculate total time t + additional wait.
    - **Standard:** A fresh particle with no waiting time.

4. Analysis of Results The histograms show that the shape of the decay is identical in both cases. The "Memoryless" plot is simply the standard plot shifted by t=2, proving that the *remaining* lifetime distribution does not change regardless of how long the particle has existed.

- Sampling from Exponential Distribution

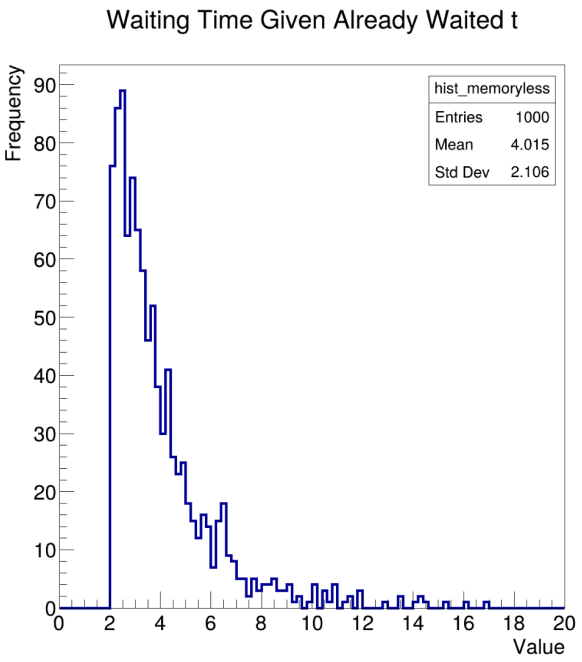
```
void Part2A_Sampling_from_Exponential_Distribution() {

    const int num_samples = 1000;
    const double lambda = 0.5;

    TRandom3 *randGen = new TRandom3(0);

    TH1F *hist = new TH1F("hist", "Exponential Distribution
Samples;Value;Frequency", 100, 0, 20);

    for (int i = 0; i < num_samples; ++i) {
        double u = randGen->Uniform(0.0, 1.0);
        double x = -log(1 - u) / lambda; // Inverse transform sampling

        hist->Fill(x);
    }

    TCanvas *canvas = new TCanvas("canvas", "Exponential Distribution
Sampling", 800, 600);
    hist->Draw();

}
```

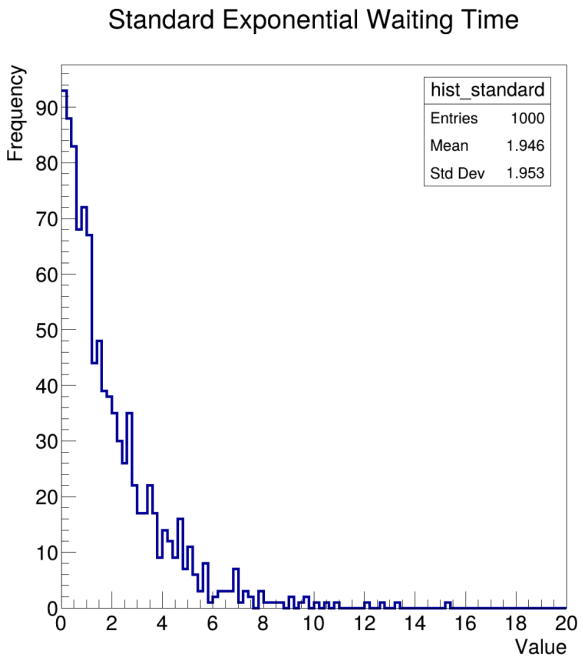


- Memoryless Property of Exponential Distribution

```
void Part2B_Memoryless_Property_of_Exponential_Distribution() {

    const int num_trials = 1000;
    const double lambda = 0.5;
    const double t = 2.0; // Time already waited

    TRandom3 *randGen = new TRandom3(0);

    TH1F *hist_memoryless = new TH1F("hist_memoryless", "Memoryless
Property;Value;Frequency", 100, 0, 20);
    TH1F *hist_standard = new TH1F("hist_standard", "Standard
Exponential;Value;Frequency", 100, 0, 20);

    for (int i = 0; i < num_trials; ++i) {
        double u1 = randGen->Uniform(0.0, 1.0);
        double x_memoryless = t + (-log(1 - u1) / lambda); // Memoryless
property

        hist_memoryless->Fill(x_memoryless);
```

```
        double u2 = randGen->Uniform(0.0, 1.0);
        double x_standard = -log(1 - u2) / lambda;

        hist_standard->Fill(x_standard);
    }

    TCanvas *canvas = new TCanvas("canvas_memoryless", "Memoryless Property
of Exponential Distribution", 1200, 600);
    canvas->Divide(2,1);

    canvas->cd(1);
    hist_memoryless->SetTitle("Waiting Time Given Already Waited t");
    hist_memoryless->Draw();

    canvas->cd(2);
    hist_standard->SetTitle("Standard Exponential Waiting Time");
    hist_standard->Draw();
}
```





C) Generating Uniformly Distributed Points in a Circle & a Sphere

1. Concept: The Jacobian Problem

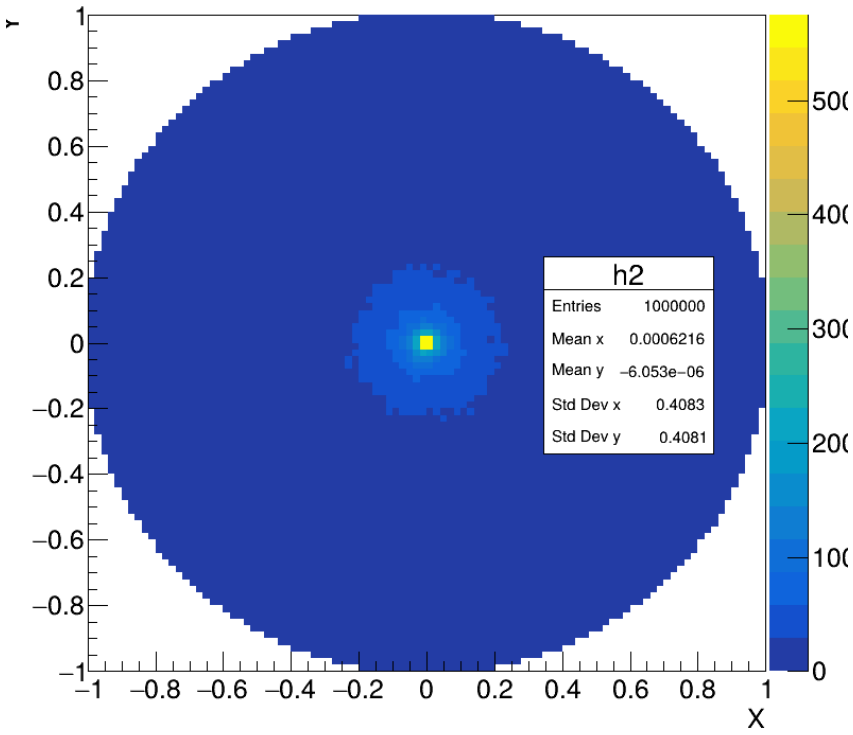
Generating points uniformly inside a circle or sphere is non-trivial. Simply choosing a random radius r and angle θ uniformly concentrates points in the center (because there is less area/volume at the center than at the edge).

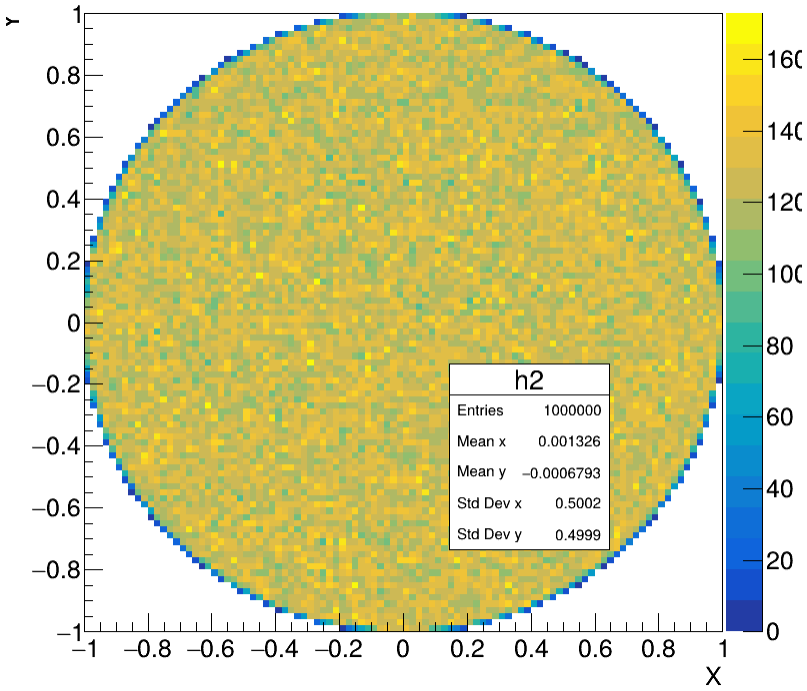
- **Correct Approach:** We must account for the increasing area/volume as r increases.
  - **Circle:** Sample r proportional to $u^{1/2}$.
  - **Sphere:** Sample r proportional to $u^{1/3}$.

2. Implementation Details

- **Circle:** The code correctly uses `sqrt(rand->Uniform(0, 1))` to adjust the radius.
- **Sphere:** The code uses `cbrt` (cube root) for the radius and generates the polar angle θ using `acos` to ensure uniform coverage over the surface area.

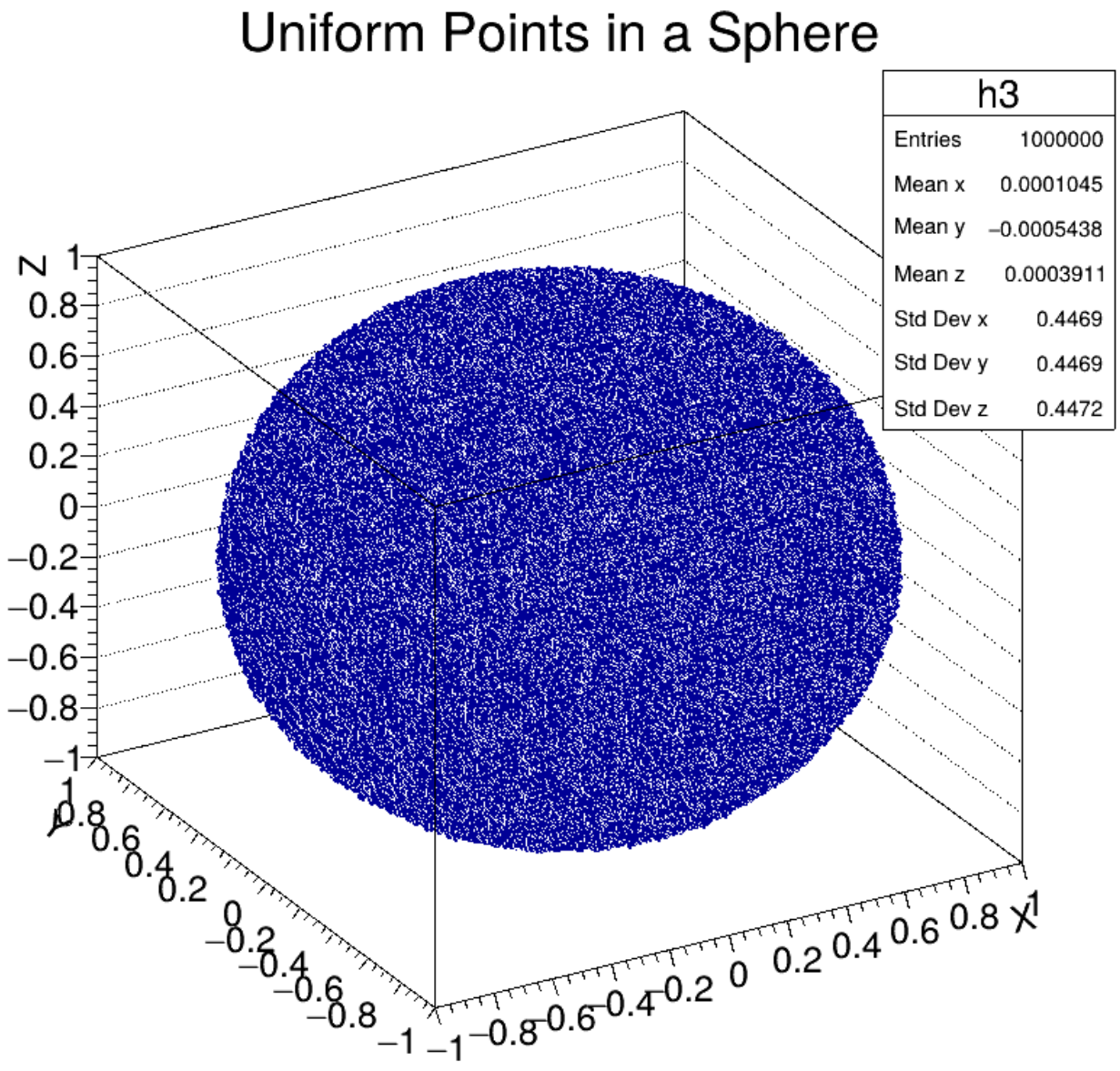
```
void Part1C_Generating_Uniform_Points_in_a_Circle() {

    TRandom3 *rand = new TRandom3(0);

    const int nPoints = 1000000;
    double r, theta;

    TCanvas *c1 = new TCanvas("c1", "Uniform Points in a Circle", 800,
800);
    TH2D *h2 = new TH2D("h2", "Uniform Points in a Circle;X;Y", 100, -1, 1,
100, -1, 1);

    for (int i = 0; i < nPoints; i++) {
        r = sqrt(rand->Uniform(0, 1)); // Corrected for uniform
distribution in circle
        //r = rand->Uniform(0, 1); // Original incorrect method
        theta = rand->Uniform(0, 2 * M_PI);

        double x = r * cos(theta);
        double y = r * sin(theta);

        h2->Fill(x, y);
    }

    h2->Draw("COLZ");
    c1->Update();

}
```






```
void Part1C_Generating_Uniform_Points_in_a_Sphere() {
```

```
    TRandom3 *rand = new TRandom3(0);

    const int nPoints = 1000000;
    double r, theta, phi;

    TCanvas *c1 = new TCanvas("c1", "Uniform Points in a Sphere", 800,
800);
    TH3D *h3 = new TH3D("h3", "Uniform Points in a Sphere;X;Y;Z", 100, -1,
1, 100, -1, 1, 100, -1, 1);

    for (int i = 0; i < nPoints; i++) {
        r = cbrt(rand->Uniform(0, 1));
        theta = acos(1 - 2 * rand->Uniform(0, 1));
        phi = rand->Uniform(0, 2 * M_PI);

        double x = r * sin(theta) * cos(phi);
        double y = r * sin(theta) * sin(phi);
        double z = r * cos(theta);

        h3->Fill(x, y, z);
    }

    h3->Draw("BOX");
    c1->Update();
}
```



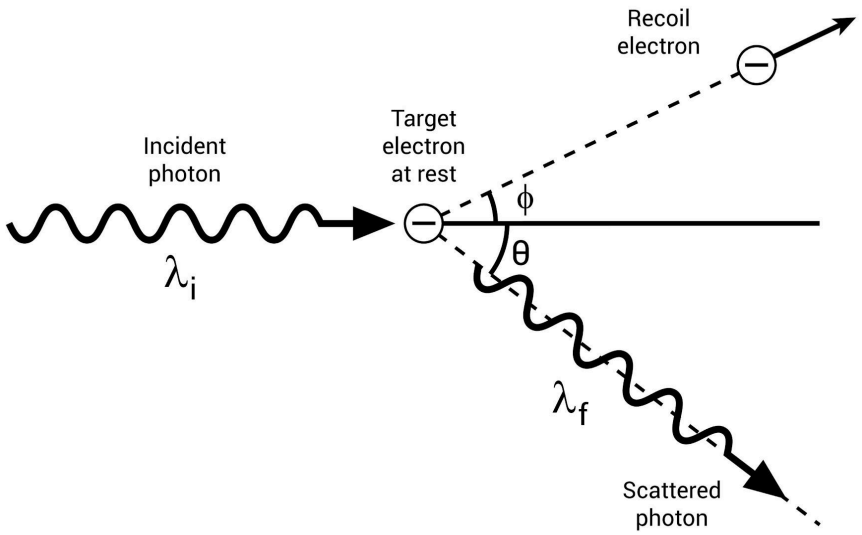
3. Analysis of Results

- **2D Plot:** The plot h2 shows a uniform disk of color. If the square root had not been used, the center would be bright yellow (high density) fading to blue at the edges.
- **3D Plot:** The sphere plot h3 shows a perfectly solid ball of points with no clustering at the origin (0,0,0), confirming the geometric transformation was correct.

Module 2: Physics of Interactions & Decay

A) Klein Nishina - Detailed Analysis

1. Concept: Compton Scattering



This simulation models **Compton Scattering**, where a photon interacts with a free electron. The physics is governed by the **Klein-Nishina formula**, which predicts:

1. **Angular Distribution:** Photons are more likely to scatter forward than backward at high energies.
2. **Energy Shift:** The photon loses energy depending on the scattering angle θ.

2. Mathematical Model

The code implements the scattered energy equation:

$$E_{\gamma'} = \frac{E_\gamma}{1 + (E_\gamma / m_e c^2)(1 - \cos\theta)}.$$

It also calculates the Differential Cross Section (DCS), which acts as the probability density function for the scattering angle.

3. Implementation Details

- **Total Cross Section:** The code numerically integrates the DCS to find the total interaction probability (2.89 * 10^{-29} m^2 or similar) .
- **Sampling:** It uses the Accept-Rejection method (from Module 1) to select scattering angles based on the physical probability curve.

```
double scattered_energy(double initial_energy, double scatter_angle) {
    const double electron_rest_mass = 511.0; // keV
    double energy_ratio = 1 + (initial_energy / electron_rest_mass) * (1 -
cos(scatter_angle));
    return initial_energy / energy_ratio;
}

double differential_cross_section(double initial_energy, double
scatter_angle) {
    const double r0 = 2.818e-15; // m
    double E_scat = scattered_energy(initial_energy, scatter_angle);
    double ratio = E_scat / initial_energy;
    double term1 = ratio * ratio;
    double term2 = ratio + (1.0 / ratio) - sin(scatter_angle) *
sin(scatter_angle);
    return 0.5 * r0 * r0 * term1 * term2; // m^2 / sr
}

double total_cross_section(double initial_energy) {
    const int num_steps = 1000;
    double total_cs = 0.0;
    double dtheta = M_PI / num_steps;

    for (int i = 0; i < num_steps; ++i) {
        double theta = i * dtheta + dtheta / 2;
        double dcs = differential_cross_section(initial_energy, theta);
        total_cs += dcs * sin(theta) * dtheta;
    }
    return 2.0 * M_PI * total_cs;
}

double accept_reject_sampling(double initial_energy, TRandom3* rand) {
    const double max_dcs = differential_cross_section(initial_energy, 0.0);
    while (true) {
        double theta = rand->Uniform(0, M_PI);
        double f = rand->Uniform(0, max_dcs);
        double f_theta = differential_cross_section(initial_energy, theta);
        if (f < f_theta) return theta;
```

```
    }
}

void Klein_Nishina_Detailed_Analysis() {
    const double initial_energy = 511.0; // keV
    TRandom3 *rand = new TRandom3(0);
    const int num_samples = 10000;

    double total_cs = total_cross_section(initial_energy);
    std::cout << "Total Cross Section: " << total_cs << " m^2" <<
std::endl;

    TH1D *h_angle = new TH1D("h_angle", "Scattering Angle
Distribution;Angle (rad);Counts", 100, 0, M_PI);
    TH1D *h_energy = new TH1D("h_energy", "Scattered Photon Energy;Energy
(keV);Counts", 100, 0, initial_energy);
    TH2D *h_angle_energy = new TH2D("h_angle_energy", "Energy vs
Angle;Angle (rad);Energy (keV)", 100, 0, M_PI, 100, 0, initial_energy);

    for (int i = 0; i < num_samples; ++i) {
        double theta = accept_reject_sampling(initial_energy, rand);
        double E_scat = scattered_energy(initial_energy, theta);
        h_angle->Fill(theta);
        h_energy->Fill(E_scat);
        h_angle_energy->Fill(theta, E_scat);
    }

    // Analytical differential cross section vs angle
    TGraph *g_dcs = new TGraph();
    for (int i = 0; i <= 100; ++i) {
        double theta = i * M_PI / 100.0;
        double dcs = differential_cross_section(initial_energy, theta);
        g_dcs->SetPoint(i, theta, dcs);
    }

    TCanvas *c1 = new TCanvas("c1", "Klein-Nishina Analysis", 1200, 800);
    c1->Divide(2,2);

    c1->cd(1);
    h_angle->Draw();

    c1->cd(2);
    h_energy->Draw();
```

```
    c1->cd(3);
    g_dcs->SetTitle("Differential Cross Section;Angle (rad);dσ/dΩ
(m^{2}/sr)");
    g_dcs->Draw("AL");

    c1->cd(4);
    h_angle_energy->Draw("COLZ");

    c1->Update();
}
```

4. Analysis of Results

The 4-panel canvas provides a complete physical picture:

- **Top Left (h_{angle}):** The distribution of scattering angles. It is not uniform; it peaks near 0 (forward scattering).
- **Top Right (h_{energy}):** The spectrum of scattered photon energies.
- **Bottom Left (g_{dcs}):** The theoretical Klein-Nishina curve. The histogram in the top left matches this theoretical curve.
- **dcsBottom Right (h_{angle_energy}):** A 2D correlation plot. It visualizes the strict kinematic relationship: small angles correspond to high energy (little energy loss), while large angles (backscattering) result in lower energy.