PH30110 Computational Astrophysics

Hand-in exercise 2024-2025 Semester 2 C Coursework 1

Computational Radiative Transfer

University of Bath

Candidate Number: 24367

March 26, 2025

PH30110: Computational Astrophysics

Candidate Number: 24367

Department of Physics,
University of Bath, Bath
BA2 7AY, UK

Date Submitted: March 26, 2025

Contents

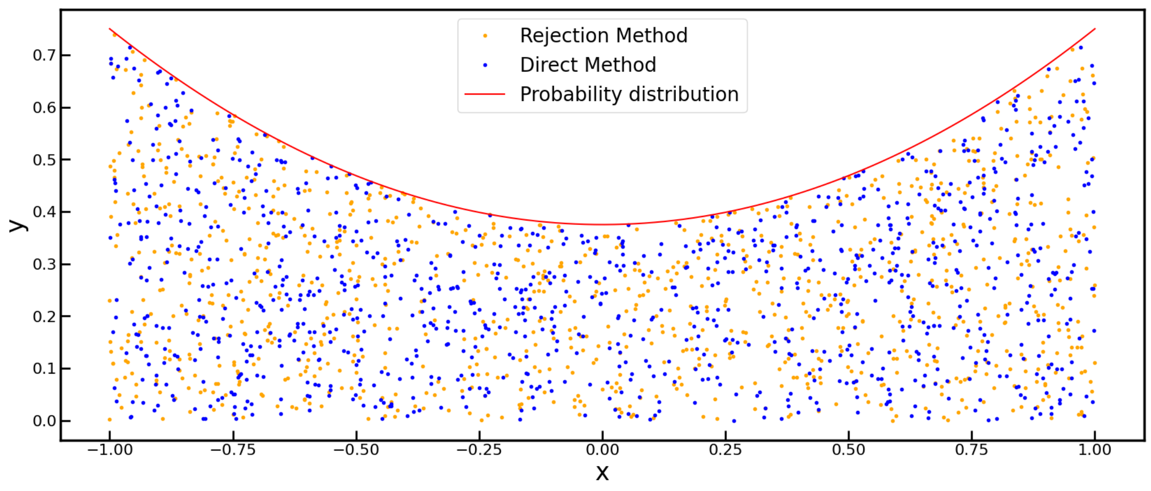

Figure 1: Overlay of $1,000$ samples from the rejection method (orange) and $1,000$ samples from the direct method (blue) drawn from the target distribution $f(x) = \frac{3}{8}(1 + x^2)$, shown by the red curve.

Question 1: Random Numbers from a Non-Flat Distribution

This question focuses on the aspect of random numbers from a non-flat distribution, namely those obeying the commonly used non-isotropic scattering function,

$$f(x) = P(\mu)d\mu = \frac{3}{8}(1 + \mu^2)d\mu \tag{1}$$

where $\mu = \cos\theta$.

A linear congruential generator (LCG), based on sequences of the following form was implemented as a starting point to generate pseudo-random numbers,

$$X_n = (aX_{n-1} + c) \bmod m$$

where a, c and m are integer numbers.

The rejection method generates candidate x values uniformly scaled to the interval $[-1, 1]$ and candidate y values scaled to the range $[0, 0.75]$ using the LCG for each iteration. For each candidate x, the value of the target function, $f(x)$, is computed using Eq.1, and the candidate is accepted if the corresponding y value is less than or equal to $f(x)$

In contrast, the direct method employs the inverse transform technique. For a random variable with cumulative distribution function (CDF) $F(x)$, if U is uniformly distributed in $[0, 1]$, then $x = F^{-1}(U)$ is distributed according to the PDF. The CDF is computed by integrating Eq.1, from -1 to x. Although the integral leads to a cubic equation in x, it can be inverted in closed form using methods from algebra. A uniformly distributed random number U is used to directly compute a sample x by inverting the cumulative distribution function, which involves solving a cubic equation through the use of cube-root and square-root functions. This approach guarantees that each iteration produces a valid sample, eliminating the need for any rejection. To further enhance computational efficiency, a lookup table is precomputed that maps a finely spaced grid of U values to their corresponding x values. At runtime, instead of repeatedly solving the cubic equation, the algorithm uses fast linear interpolation on this lookup table to approximate x for any given U. This pre-computation significantly reduces the computational overhead while maintaining accuracy, thereby speeding up the direct method considerably.

Both methods are timed using the system clock, and their outputs are saved to separate files, providing a practical demonstration of both the theoretical and computational aspects of non-isotropic random number generation in Monte Carlo simulations. The use of separate seed variables ensures the sequences of random numbers used in each part are independent. This avoids accidental correlations between x and y coordinates and between the rejection and direct methods.

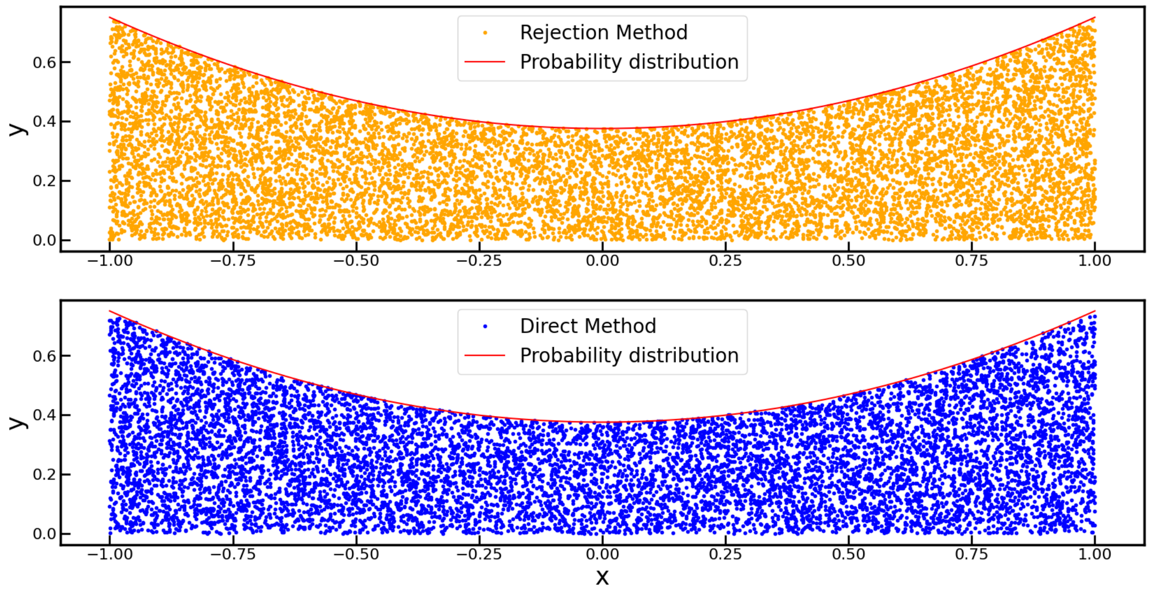

Figure 2: Independent scatter plots showing $10,000$ samples each for the rejection method (top, orange points) and the direct method (bottom, blue points), drawn from the distribution. The red curve depicts the probability density function as previously shown.

Fig.1 confirms how each sampling method populates the region under the curve, ultimately yielding the desired outcome and approaching the distribution. The figure was produced with a target of $1,000$ samples. A simulation producing $10,000$ samples was generated and is shown in Fig.2, showing that even with a higher density of points, both methods produce results below the distribution function. Overall, as the sample size increases from $1,000$ to $10,000$, the shape of $f(x)$ becomes more clearly outlined by the density of the points, demonstrating that both methods converge to the correct distribution.

To supplement the comparison between the rejection and direct sampling methods, Fig.3 was produced to show the computation times per N value of samples averaged over 10 runs. In addition, the ratio in computation time denoted as *speed-up* is shown, demonstrating the direct method's superior advantages when it comes to processing time, with an average increase in the processing time of 2.49 times for the samples plotted.

Question 2: Monte Carlo Scattering, Isotropic

In this section, the random walk approach is used to simulate the Monte Carlo radiative transfer of photons through an atmosphere. The atmospheric region is defined by the vertical boundaries

$$z_{\text{min}} = 0 \quad \text{and} \quad z_{\text{max}} = 200,$$

with an overall optical depth, τ, of

$$\tau_{\text{total}} = 10.$$

The conversion from optical depth to physical distance is handled via the parameter alpha,

$$\alpha = \frac{\tau_{\text{total}}}{z_{\text{max}} - z_{\text{min}}} = \frac{10}{200}.$$

This parameter is used to scale the random free path, which is sampled from an exponential distribution. The simulation proceeds as follows:

1. Each photon is launched from the bottom of the atmosphere (at $z = 0$) with an initial position of $(x, y, z) = (0, 0, 0)$. Its initial direction is chosen isotropically. Two random numbers are used: one to sample the azimuthal angle ϕ (via $\phi = 2\pi r_1$) and another to determine the cosine of the polar angle ($\mu = 2r_2 - 1$). The polar angle θ is then derived from μ, with the corresponding sine calculated as $\sin\theta = \sqrt{1 - \mu^2}$. The initial direction vector is given by

$$\text{dir}_x = \sin\theta\cos\phi, \quad \text{dir}_y = \sin\theta\sin\phi, \quad \text{dir}_z = \mu.$$

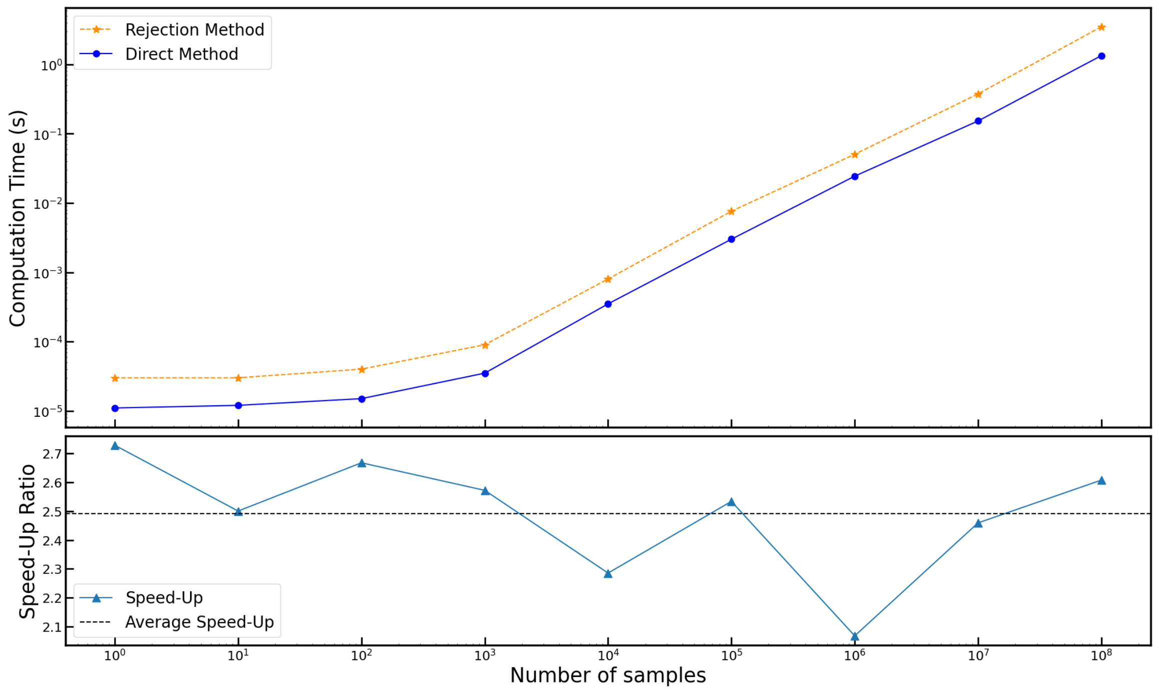

Figure 3: Comparison of the computation time (top panel) for the Rejection Method (orange) and the Direct Method (blue) over a range of sample sizes, plotted on a log scale. The lower panel shows the speed-up ratio (Rejection time / Direct time). Although the Rejection Method initially competes well, the Direct Method's performance becomes more advantageous at larger sample counts, as reflected by the rising speed-up.

2. Within the atmospheric region, the photon propagates by taking steps of random length. The step length is determined by sampling an optical depth τ_{step} from the exponential distribution $P(\tau) = e^{-\tau}$ using the transformation

$$\tau_{\text{step}} = -\ln(U),$$

where U is a uniformly distributed random number. The physical distance s travelled in one step is then computed as:

$$s = \frac{\tau_{\text{step}}}{\alpha}.$$

The photon's position is updated using

$$x \mathrel{+}= s\,\text{dir}_x, \quad y \mathrel{+}= s\,\text{dir}_y, \quad z \mathrel{+}= s\,\text{dir}_z.$$

3. After each step, the code checks whether the photon has exited the atmospheric region:

   - If $z > z_{\text{max}}$, the photon has escaped from the top. The exiting angle is determined from the current direction (specifically, the z-component, which is $\mu_{\text{exit}} = \text{dir}_z$), and this value is binned into one of 10 intervals between 0 and 1.
   - If $z < z_{\text{min}}$, the photon has escaped from the bottom, and it is discarded.
   - If the photon is still within the atmospheric region, a new random number is generated to decide whether it scatters or is absorbed. With an albedo of 1 (i.e. no absorption), scattering always occurs. In this case, a new direction is sampled in the same manner as at launch, giving a new set of angles (ϕ and θ) that reset the photon's propagation direction. This is depicted in Fig.4

4. The simulation iterates over incoming photons until one million photons have been observed to escape from the top of the atmospheric region. During this process, the code tracks the total number of photons launched, the number escaping from the top, and those escaping from the bottom.

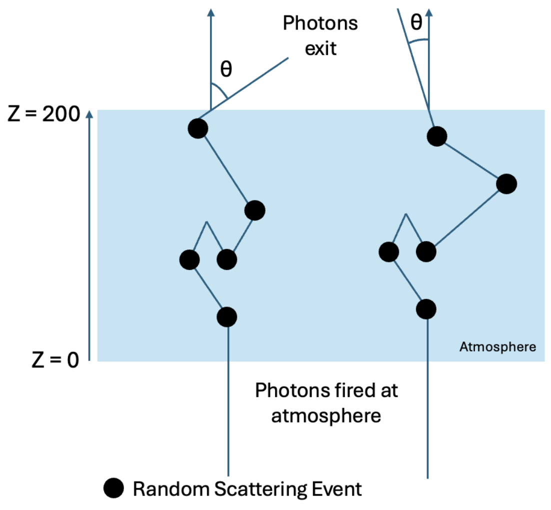

Figure 4: A schematic of the Monte Carlo random-walk approach used to model photon transport through an atmospheric region of height $z = 200$. Photons are launched at $z = 0$, scatter randomly (black dots), and may exit the top at various angles θ. Each dot represents a scattering event where the photon's direction is re-randomised.

5. Finally, the fraction of photons escaping in each of the 10 angular bins (based on μ) is calculated. The binning process involves determining the midpoint for each bin, and the resulting data is written to an output file. The final output includes the bins and the normalised intensity for each.

In addition, OpenMP is used to run many photon trajectories in parallel, thereby speeding up the Monte Carlo simulation. Each thread processes photons in batches using constant `CHUNK_SIZE`, rather than updating global counters on every photon. Locally, each thread accumulates values in counters such as `local_top`, `local_bottom`, and `local_launched`, which track how many photons escaped the top, escaped the bottom, or were launched by that thread. Periodically, these local totals are merged into the shared counters `top_escaped_global`, `bottom_escaped_global`, and `photons_launched_global` through an atomic capture block. Each thread also maintains its own random-number seeds (`local_seedA`, `local_seedB`) so random walks run independently in parallel. The code checks if `top_escaped_global` has reached one million photons, and if so, the threads stop. By structuring the code this way, using thread-local counters plus periodic atomic updates, the simulation efficiently distributes the workload across CPU cores and avoids excessive synchronisation, greatly reducing the runtime needed to achieve one million top-escaped photons.

Fig.5 shows how the normalised intensity of photons escaping through the top boundary depends on $\cos\theta$, where θ is the polar angle measured from the atmospheric normal, depicted in Fig.4. Due to the atmosphere being optically thick, $\tau_{\text{total}} = 10$, photons directed more vertically (higher $\cos\theta$) have a shorter path to exit and thus a greater likelihood of escaping. As a result, the intensity is higher at large $\cos\theta$ values. In contrast, photons travelling at steeper angles (smaller $\cos\theta$) must traverse a longer path within the atmospheric region and are more likely to scatter multiple times or exit through the bottom, leading to lower escape fractions at smaller $\cos\theta$. This trend is clearly reflected in the increasing curve from left to right in the plot.

Question 3: Rayleigh scattering

The final section is concerned with scattering in the non-isotropic case, Rayleigh scattering. Physically, Rayleigh scattering describes the probability that a photon scatters through an angle $\hat{\theta}$ between its *incoming* and *outgoing* directions. The corresponding differential cross-section can be written as

$$P(\hat{\theta})\, d\Omega \;=\; \frac{3}{4}\left(1 + \cos^2\hat{\theta}\right)\frac{d\Omega}{4\pi},$$

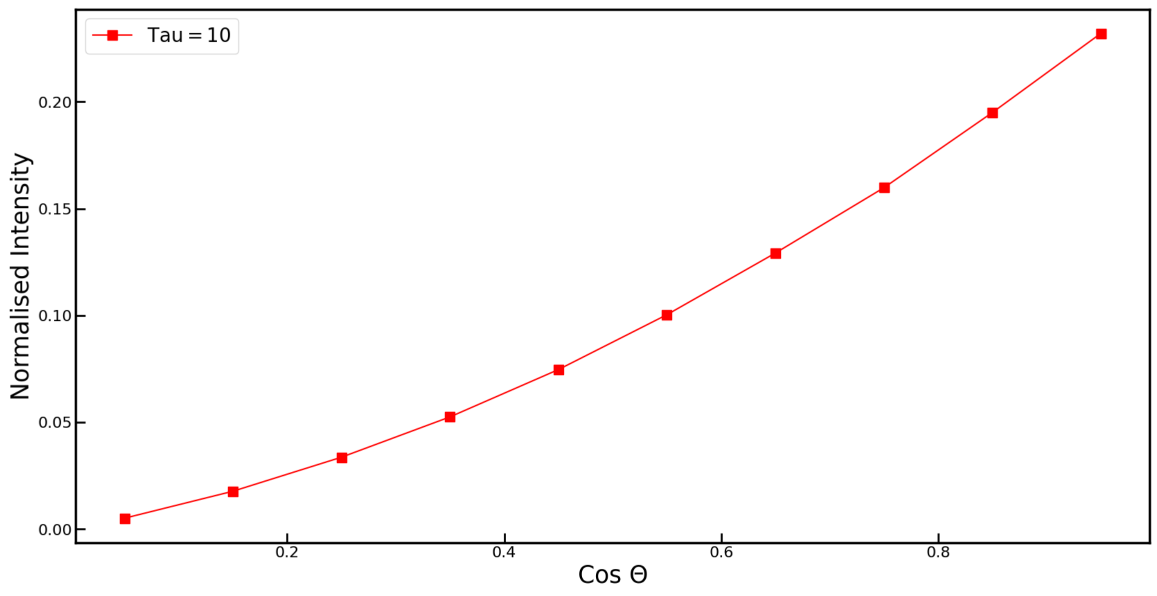

Figure 5: The fraction of photons escaping through the top of the atmospheric region, binned by $\cos\theta$. Higher values of $\cos\theta$ (more vertical angles) yield greater escape fractions, reflecting the shorter path through the atmospheric region.

where $\hat{\theta}$ is the angle between the incident and scattered rays, and $d\Omega$ is an element of solid angle.

This question builds on the code in question 2, utilising the `simulate_photon_transport()` function which models the random walk of photons through the atmosphere. Unlike the isotropic case, proper Rayleigh scattering requires that the scattering angle be measured relative to the photon's incoming direction. To achieve this, the scattering angle is sampled, and then the old direction is rotated into a new direction using a local orthonormal basis. To incorporate this, several adaptations are implemented. Firstly, the absorption check remains the same as before. The code below explains the changes implemented.

1. Sampling Scattering Angle
   - The cosine of the scattering angle (μ_s) is sampled using a uniform random number passed to the `direct_sample()` function.
   - This function inverts the CDF for the Rayleigh-like phase function $f(\mu) = \frac{3}{8}(1 + \mu^2)$ to obtain μ_s.
   - Additionally, an independent random number is used to sample the azimuthal angle (ϕ_s) uniformly over $[0, 2\pi]$.
2. Local Rotation
   - A local rotation is performed of the photon's direction relative to its incoming direction.
   - The current photon direction, $\mathrm{d_{in}} = (\mathrm{dir_x}, \mathrm{dir_y}, \mathrm{dir_z})$, is normalised to create a unit vector.
   - An orthonormal basis is constructed. A vector $\mathbf{h}$ is computed by taking the cross product of $\mathbf{n}$ (a unit vector of the photons incoming direction) with a reference vector (usually the z-axis unless $\mathbf{n}$ is nearly parallel to it, in which case the x-axis is used). A second perpendicular vector $\mathbf{u}$ is then obtained by computing the cross product of $\mathbf{n}$ and $\mathbf{h}$.
   - New photon direction is computed using $\mathrm{d_{out}} = \mu_s\mathbf{n} + \sqrt{1-\mu_s^2}\cos\phi_s\mathbf{h} + \sqrt{1-\mu_s^2}\sin\phi_s\mathbf{u}$. This is now the direction of the photon relative to the incoming direction.
3. Update photon's direction
   - The new direction vector from the local rotation replaces the old direction and is used for subsequent steps in the photon's random walk.

Fig.6 demonstrates the case where Rayleigh scattering has been implemented for $\tau = 10$ and $\tau = 0.1$. When photons of shorter wavelengths (blue light) propagate through the atmosphere, they experience a higher optical depth and thus scatter more frequently than longer-wavelength (redder)

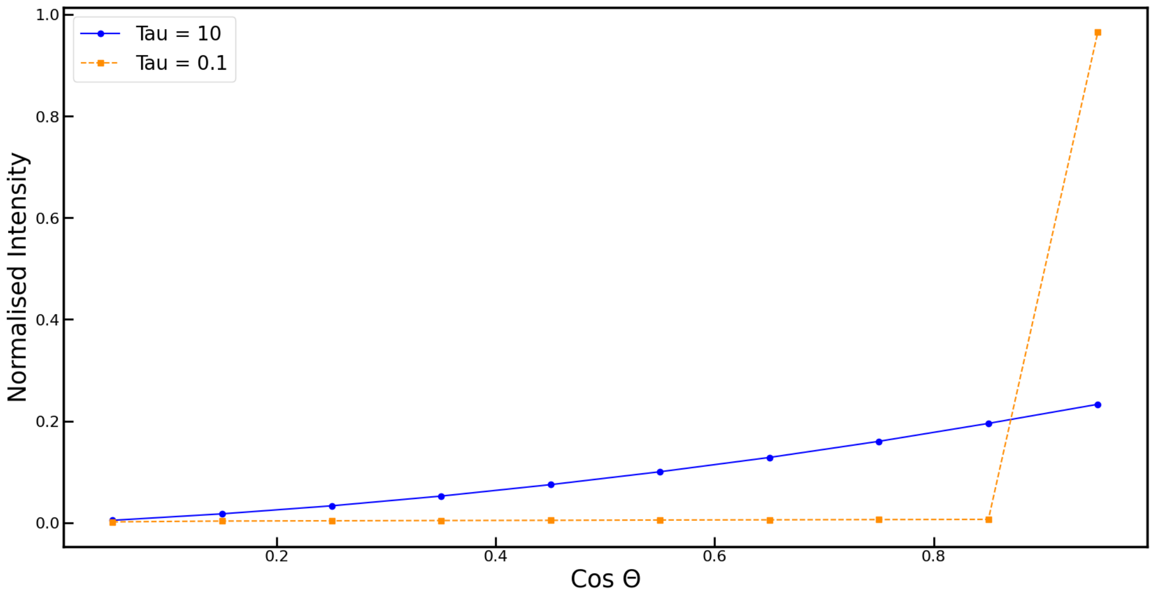

Figure 6: Comparison of the normalised intensity of photons escaping from the top of the atmosphere, binned by $\cos\theta$, for two different optical depths of $\tau = 10$ and $\tau = 0.1$. Higher τ causes more scattering events, concentrating escaping photons near $\cos\theta \approx 1$, while lower τ yields a flatter distribution over angle.

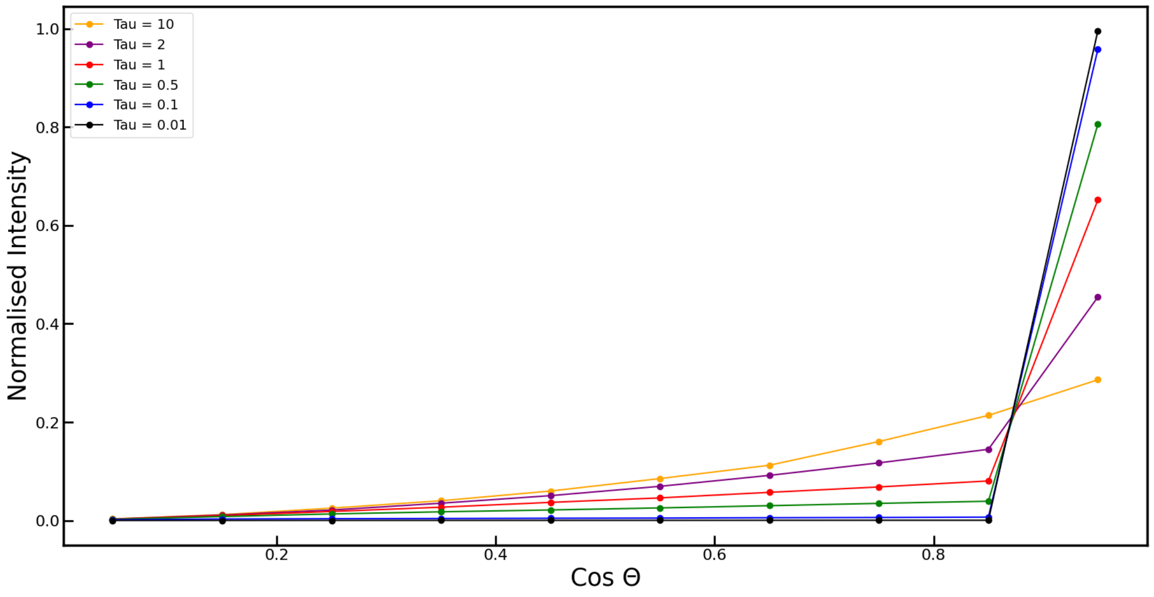

Figure 7: Comparison of the normalised intensity of photons escaping from the top of the atmosphere, binned by $\cos\theta$, for multiple optical depths ($\tau = 10, 2, 1, 0.5, 0.1, 0.01$). Higher τ causes more scattering events, concentrating escaping photons near $\cos\theta = 1$, while lower τ yields a flatter distribution.

photons. This is modelled by assigning a larger $\tau = 10$ for blue photons and a smaller $\tau = 0.1$ for other colours. The higher τ (blue) photons are scattered strongly and emerge at a wide range of angles away from the direct beam, leading to a pronounced intensity of blue light when looking overhead. Meanwhile, the direct sunlight beam loses much of its short-wavelength component, making the transmitted Sun itself appear redder, especially at lower angles. Consequently, this difference in scattering strength quantitatively explains why the sky appears blue (dominated by scattered short-wavelength photons) when one is not staring directly at the Sun.

Around $\cos(\theta) \approx 0.9$, the photons are travelling fairly close to vertically but not perfectly so, and in that regime, the fraction of photons that escape happens to converge for a range of optical depths. In an optically thick medium, angles near the vertical are most favoured for escape, whereas in an optically thin medium, nearly all angles have a decent escape probability. There can be a specific "pivot" angle (around $\cos(\theta) \approx 0.9$ in Fig.7) where these effects balance out, causing curves for different τ values to intersect. Numerics and finite sampling can further accentuate the intersection, but physically, it reflects a crossover point where neither extremely high nor extremely low optical depth dominates the escape probability.

Appendix

4.1 Full Code

```c
// ------------------------------------------------------------------------------------
// To compile: (Capital letter O not zero)
// gcc -fopenmp -O3 -ffast-math -march=native 24367.c -o 24367 -lm
// To run:
// ./24367
// ------------------------------------------------------------------------------------
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <omp.h>

// Global constants
const long long MODULUS = 2147483647LL; // 2^31 - 1
long long seed_scatter = 99991;          // For Q2 & Q3 scattering
long long seed1 = 1, seed2 = 107, seed3 = 123;

// Constants for Q1
const int NUM_SAMPLES = 10000;
int num_accepted = 0, count = 0;

// Constants for Q2 and Q3
const double Z_MIN = 0.0, Z_MAX = 200.0, ALBEDO = 1.0;
const long long N_PHOTONS_TARGET = 1000000LL; // Exactly 1,000,000 out top is needed
const int NUM_BINS = 10;
double tau_total = 10.0; // Change if needed to 0.1 for 'other colours' case

// Constants for lookup table for direct method
#define LUT_SIZE 10000    // Number of table entries
static double lookup_table[LUT_SIZE];
static int lookup_init = 0;

// Linear Congruential Generator (LCG) for [0,1) random
long long LCG(long long current, long long a, long long c, long long m) {
    return (a * current + c) % m;
}

// This inline function advances the LCG-based seed and returns a random number in [0, 1).
static inline double LCG_rand01(long long *seedptr, long long a) {
    *seedptr = LCG(*seedptr, a, 0, MODULUS);
    return (double)(*seedptr) / (double)MODULUS;
}

// The PDF for Q1 3/8*(1+x^2) for x in [-1,1]
double distribution_func(double x) {
    return (3.0 / 8.0) * (1.0 + x*x);
}

// Solve cubic for x in [-1,1] that satisfies F(x) = U. Inverts the CDF
double direct_sample(double U) {
    double half_q = 2.0 - 4.0 * U;
    double term   = sqrt(half_q*half_q + 1.0);
    double c1     = cbrt(-half_q + term);
    double c2     = cbrt(-half_q - term);
    return (c1 + c2);
}

 // Precompute the lookup table.
void init_lookup_table(void) {
        if (lookup_init)
        return;
    for (int i = 0; i < LUT_SIZE; i++) {
        double U = (double)i / (LUT_SIZE - 1);  // U spans 0 to 1
        lookup_table[i] = direct_sample(U);
    }
    lookup_init = 1;
}

 // Use linear interpolation on the lookup table to approximate direct_sample(U).
```

```
double direct_sample_lookup(double U) {
    if (!lookup_init)
        init_lookup_table();
    double pos = U * (LUT_SIZE - 1);
    int index = (int) pos;
    double frac = pos - index;
    if (index >= LUT_SIZE - 1)
         return lookup_table[LUT_SIZE - 1];
    return lookup_table[index]*(1 - frac) + lookup_table[index + 1]*frac;
}

// Q1: Rejection & Direct Methods (left single-threaded)
int question_1(void) {
   // Allocate arrays for storing sampled points (x,y) in rejection method
    double *rej_x = (double *)malloc(NUM_SAMPLES * sizeof(double));
    double *rej_y = (double *)malloc(NUM_SAMPLES * sizeof(double));
    if (!rej_x || !rej_y) {
        perror("Error: Memory allocation failed");
        return 1;
    }
   // --- Rejection Method ---
   clock_t start_rejection = clock();
   while (num_accepted < NUM_SAMPLES) {
       count++; // Total itertations accepted
       // Sample x in [-1,1], y in [0,0.75]
       double x_reject = 2.0 * LCG_rand01(&seed1, 16807) - 1.0;
       double y_reject = 0.75 * LCG_rand01(&seed2, 48271);
       // Evaluate PDF at x
       double fx = distribution_func(x_reject);
       // If (x_reject, y_reject) lies under curve then accept
       if (y_reject <= fx) {
           rej_x[num_accepted] = x_reject;
           rej_y[num_accepted] = y_reject;
           num_accepted++;
       }
   }
    clock_t end_rejection = clock();
    double time_rejection = (double)(end_rejection - start_rejection) /
   CLOCKS_PER_SEC;
    // Write rejection samples
    FILE *fptr_rejection = fopen("rejection_method.txt", "w");
    if (fptr_rejection) {
        for (int i = 0; i < NUM_SAMPLES; i++) {
            fprintf(fptr_rejection, "%.3f, %.3f\n", rej_x[i], rej_y[i]);
        }
        fclose(fptr_rejection);
    }
    // --- Direct Method ---
    // Allocate arrays for direct method samples
    double *dir_x = (double *)malloc(NUM_SAMPLES * sizeof(double));
    double *dir_y = (double *)malloc(NUM_SAMPLES * sizeof(double));
    if (!dir_x || !dir_y) {
        perror("Error: Memory allocation failed");
        free(rej_x);
        free(rej_y);
        return 1;
    }
    // Measure the lookup table build time
    clock_t start_lut = clock();
    init_lookup_table();
    clock_t end_lut = clock();
    double time_lut = (double)(end_lut - start_lut) / CLOCKS_PER_SEC;
    printf("Lookup table build time: %.8f s\n", time_lut);

    clock_t start_direct = clock();
    for (int i = 0; i < NUM_SAMPLES; i++) {
        // Generate U in [0,1], invert CDF to get x
        double U = LCG_rand01(&seed3, 16807);
        // Use lookup table with linear interpolation to compute x
        double x_val = direct_sample_lookup(U);
        // For plotting only: pick y as a random fraction up to f(x_val)
       double y_val = LCG_rand01(&seed3, 48271) * distribution_func(x_val);
        dir_x[i] = x_val;
        dir_y[i] = y_val;
    }
    clock_t end_direct = clock();
```

```c
    double time_direct = (double)(end_direct - start_direct) / CLOCKS_PER_SEC;
    // Write direct method output to file
    FILE *fptr_direct = fopen("direct_method.txt", "w");
    if (fptr_direct) {
        for (int i = 0; i < NUM_SAMPLES; i++) {
            fprintf(fptr_direct, "%.3f, %.3f\n", dir_x[i], dir_y[i]);
        }
        fclose(fptr_direct);
    }
    // Print summary of results
    printf("\nPart 1: Rejection Method:\n");
    printf("    Accepted samples  = %d\n", NUM_SAMPLES);
    printf("    Total iterations  = %d\n", count);
    printf("    Acceptance ratio  = %.3f\n", (double)NUM_SAMPLES / (double)count);
    printf("    Total CPU time (s)= %.8f\n", time_rejection);
    printf("    Time per sample   = %.8e s\n\n", time_rejection / NUM_SAMPLES);

    printf("Part 1: Direct Method (Lookup Table):\n");
    printf("    Generated samples = %d\n", NUM_SAMPLES);
    printf("    Total CPU time (s)= %.8f\n", time_direct);
    printf("    Time per sample   = %.8e s\n\n", time_direct / NUM_SAMPLES);
    // Avoid dividing by zero if times are extremely small
    if (time_rejection <= 0.0) time_rejection = 1e-6;
    if (time_direct <= 0.0) time_direct = 1e-6;
    double speedup = time_rejection / time_direct;
    printf("Speedup (Rejection / Direct) = %.4f\n\n", speedup);
    // Free all allocated memory
    free(rej_x); free(rej_y);
    free(dir_x); free(dir_y);
    return 0;
 }

// --- Isotropic Scattering for Q2---
double scatter_isotropic(double U) {
    // Map uniform random U in [0,1) to mu in [-1,1]
    // for an isotropic phase function: mu = 2U - 1
    return 2.0*U - 1.0;
}

// --- Rayleigh scattering for Q3 ---
double scatter_rayleigh(double U) {
    // Rayleigh scattering ~ (1 + cos^2 theta),
    // reuse direct_sample(U) which inverts the CDF.
    return direct_sample(U);
}

// write_bins: outputs bin index, fraction, mu_mid to 'fout'
static void write_bins(int *bin_counts, int num_bins, double dmu, long long top_escaped, FILE *fout) {
    for (int j = 0; j < num_bins; j++) {
        // mu_mid is the midpoint for the j-th bin in [0,1]
        double mu_mid = (j + 0.5) * dmu;
        // fraction = fraction of total top-escaped photons in this bin
        double fraction = 0.0;
        if (top_escaped > 0) {
            fraction = (double)bin_counts[j] / (double)top_escaped;
        }
        fprintf(fout, "%d, %e, %f\n", j, fraction, mu_mid);
    }
}

// Orthonormal rotation for Rayleigh scattering in local frame.
// rotate the direction vector 'd_in' into 'd_out' with a scattering angle mu and azimuth phi.
static void rotate_rayleigh_local(
    const double d_in[3], // Old direction
    double mu,            // cos (scattering angle)
    double phi,           // random azimuth in [0,2   )
    double d_out[3])      // New direction
    {
    // Normalise the old direction
    double len = sqrt(d_in[0]*d_in[0] + d_in[1]*d_in[1] + d_in[2]*d_in[2]);
    double nx = d_in[0]/len, ny = d_in[1]/len, nz = d_in[2]/len;
    double sin_theta = sqrt(1.0 - mu*mu);

    // Build local orthonormal basis
    // Find a vector h orthonormal to n then cross to get the 3rd vector u
    double hx, hy, hz;
    if (fabs(nz) < 0.9999) {
        // cross n with z-hat -> h = (ny, -nx, 0)
        hx = ny;
        hy = -nx;
        hz = 0.0;
    } else {
        // cross n with x-hat -> h = (0, nz, -ny)
        hx = 0.0;
        hy = nz;
        hz = -ny;
    }
    double h_len = sqrt(hx*hx + hy*hy + hz*hz);
    hx /= h_len;
    hy /= h_len;
    hz /= h_len;

    // Now u = n x h
    double ux = ny*hz - nz*hy;
    double uy = nz*hx - nx*hz;
    double uz = nx*hy - ny*hx;

    // The new direction in local coordinates
    d_out[0] = mu*nx + sin_theta*cos(phi)*hx + sin_theta*sin(phi)*ux;
    d_out[1] = mu*ny + sin_theta*cos(phi)*hy + sin_theta*sin(phi)*uy;
    d_out[2] = mu*nz + sin_theta*cos(phi)*hz + sin_theta*sin(phi)*uz;
}

 // Q2 and Q3 driver function to determine scattering type
typedef enum {
    INIT_ISOTROPIC = 0,
    INIT_VERTICAL  = 1
} InitDirectionType;

// Function pointer for scattering routines
typedef double (*ScatteringFunc)(double);

// Launch photons until n_photons_required escape the top boundary.
// Each photon is launched (either isotropic or vertical initial dir),
// and scatters (Rayleigh or Isotropic). Final directions are binned by mu in 10 bins
    .
// If top_escaped >= n_photons_required, the simulation stops.
void simulate_photon_transport(
    double tau_total, double zmin, double zmax,
    double albedo,long long n_photons_required,
    InitDirectionType init_type, ScatteringFunc scatter_func,
    const char *outfile_label) {
    // dmu = bin width in mu from 0..1 for 10 bins
    double dmu = 1.0 / (double)NUM_BINS;
    int *bin_counts_global = (int *)calloc(NUM_BINS, sizeof(int));
    if (!bin_counts_global) {
        fprintf(stderr, "Error: Could not allocate bin_counts.\n");
        return;
    }
    FILE *fout = fopen(outfile_label, "w");
    if (!fout) {
        perror("Error opening output file");
        free(bin_counts_global);
        return;
    }
    // alpha = tau_total / (zmax - zmin)
    double alpha = tau_total / (zmax - zmin);

    static long long top_escaped_global = 0;
    static long long bottom_escaped_global = 0;
    static long long photons_launched_global = 0;
    // Make sure each run starts from zero
    top_escaped_global = 0;
    bottom_escaped_global = 0;
    photons_launched_global = 0;

    // Process in chunks
    const int CHUNK_SIZE = 1000;
    // Start timing the simulation
    clock_t start_time = clock();
```

```c

    // Start parallel region
    #pragma omp parallel
    {
        // Each thread has local seeds, counters
        int tid = omp_get_thread_num();
        long long local_seedA = seed_scatter + 10000LL * tid;
        long long local_seedB = (seed_scatter + 12345LL) + 10000LL * tid;

        long long local_top = 0;    // how many escaped top from this thread
        long long local_bottom = 0; // how many escaped bottom
        long long local_launched = 0;

        int bin_counts_local[NUM_BINS];
        for (int i = 0; i < NUM_BINS; i++) {
            bin_counts_local[i] = 0;
        }

        // Keep going until top_escaped_global >= n_photons_required
        while (1) {
            long long curr_top;
            #pragma omp atomic read
            curr_top = top_escaped_global;
            if (curr_top >= n_photons_required) {
                break;
            }

            long long needed = n_photons_required - curr_top;
            long long chunk = (needed < CHUNK_SIZE) ? needed : CHUNK_SIZE;

            // Launch chunk photons
            for (int c = 0; c < chunk; c++) {
                local_launched++;

                // Initialise photon position and direction
                double x=0.0, y=0.0, z=0.0;
                double phi, mu;
                // If init_type == INIT_ISOTROPIC -> random direction
                if (init_type == INIT_ISOTROPIC) {
                    double r1 = LCG_rand01(&local_seedA, 16807);
                    double r2 = LCG_rand01(&local_seedB, 48271);
                    phi = 2.0 * M_PI * r1;
                    mu  = 2.0 * r2 - 1.0;
                // else (INIT_VERTICAL) -> mu=1, phi=0 -> straight up
                } else {
                    mu  = 1.0;
                    phi = 0.0;
                }
                double sin_theta = sqrt(1.0 - mu*mu);
                double dir_x = sin_theta*cos(phi);
                double dir_y = sin_theta*sin(phi);
                double dir_z = mu;

                // Random walk
                while (1) {
                    double tau_step = -log(LCG_rand01(&local_seedA, 16807));
                    double s = tau_step / alpha;
                    x += s*dir_x;
                    y += s*dir_y;
                    z += s*dir_z;
                    // If photon escapes top, bin by mu
                    if (z > zmax) {
                        double mu_exit = dir_z;
                        if (mu_exit < 0.0) mu_exit = 0.0;
                        int index = (int)(mu_exit / dmu);
                        if (index >= NUM_BINS) index = NUM_BINS - 1;
                        bin_counts_local[index]++;
                        local_top++;
                        break;
                    } else if (z < zmin) {
                        // If it escapes bottom
                        local_bottom++;
                        break;
                    } else {
                        // Possibly scatter or absorb
                        double rscat = LCG_rand01(&local_seedB, 48271);
                    if (rscat < albedo) {
                        // Decide new direction
                        if (scatter_func == scatter_rayleigh) {
                            double rU    = LCG_rand01(&local_seedA, 16807);
                            double mu_s = direct_sample(rU);
                            double r_phi = LCG_rand01(&local_seedB, 48271);
                            double phi_s = 2.0 * M_PI * r_phi;

                            double d_in[3] = {dir_x, dir_y, dir_z};
                            double d_out[3];
                            // rotate around old direction using mu_s, phi_s
                            rotate_rayleigh_local(d_in, mu_s, phi_s, d_out);
                            dir_x = d_out[0];
                            dir_y = d_out[1];
                            dir_z = d_out[2];
                        } else {
                            // Isotropic
                            double r_phi = LCG_rand01(&local_seedA, 16807);
                            double r_mu  = LCG_rand01(&local_seedB, 48271);
                            double phi_s = 2.0*M_PI * r_phi;
                            double mu_s  = scatter_func(r_mu);

                            double stheta = sqrt(1.0 - mu_s*mu_s);
                            dir_x = stheta * cos(phi_s);
                            dir_y = stheta * sin(phi_s);
                            dir_z = mu_s;
                        }
                    } else {
                        // absorbed -> done
                        break;
                    }
                }
            } // end random walk
        } // end chunk loop from parralel region

        // Merge local counters into global with atomic capture
        long long old_top, new_top;
        #pragma omp atomic capture
        {
            old_top = top_escaped_global;
            top_escaped_global = top_escaped_global + local_top;
        }
        new_top = old_top + local_top;
        if (new_top > n_photons_required) {
            #pragma omp critical
            {
                if (top_escaped_global > n_photons_required) {
                    top_escaped_global = n_photons_required;
                }
            }
        }
        // reset local_top so it is not added it again
        local_top = 0;
        // update bottom
        #pragma omp atomic
        bottom_escaped_global += local_bottom;
        local_bottom = 0;
        // update launched
        #pragma omp atomic
        photons_launched_global += local_launched;
        local_launched = 0;
        // merge bin arrays
        #pragma omp critical
        {
            for (int i = 0; i < NUM_BINS; i++) {
                bin_counts_global[i] += bin_counts_local[i];
                bin_counts_local[i] = 0;
            }
        }
        // If at or above the target, break
        long long after_top;
        #pragma omp atomic read
        after_top = top_escaped_global;
        if (after_top >= n_photons_required) {
            break;
        }
```

```c
        } // end while loop
    } // end parallel region

    clock_t end_time = clock();
    double elapsed = (double)(end_time - start_time) / CLOCKS_PER_SEC;

    printf("Simulation: %s\n", outfile_label);
    printf("    Photons launched : %lld\n", photons_launched_global);
    printf("    Escaped top      : %lld\n", top_escaped_global);
    printf("    Escaped bottom   : %lld\n", bottom_escaped_global);
    printf("    CPU time (s)     : %.8f\n \n", elapsed);

    // Write final bin data to file
    write_bins(bin_counts_global, NUM_BINS, dmu, top_escaped_global, fout);
    fclose(fout);
    free(bin_counts_global);
}

// Q2: Isotropic scattering
int question_2(void) {
    simulate_photon_transport(
        tau_total, Z_MIN, Z_MAX, ALBEDO,
        N_PHOTONS_TARGET, INIT_ISOTROPIC, scatter_isotropic,
        "Question_2_Isotropic.txt");
    return 0;
}

// Q3: Rayleigh scattering
int question_3(void) {
    simulate_photon_transport(
        tau_total, Z_MIN, Z_MAX, ALBEDO,
        N_PHOTONS_TARGET, INIT_VERTICAL, scatter_rayleigh,
        "Question_3_Rayleigh.txt");
    return 0;
}

int main(void) {
    question_1();
    question_2();
    question_3();
    return 0;
}
```

End of Report