Monte Carlo radiative transfer

Question 1a: *Random numbers from a non-flat distribution*

Rejection method

This exercise involved producing a non-flat distribution obeying

$$P(\mathrm{x})\mathrm{dx} = \exp(-\mathrm{x})\,\mathrm{dx} \tag{1}$$

using two methods: the rejection method and the cumulative function approach. The rejection method starts with a random distribution of points. The distribution obeying equation 1 is achieved by rejecting and redrawing all x values for which the corresponding y lies above the function $P(\mathrm{x})$.

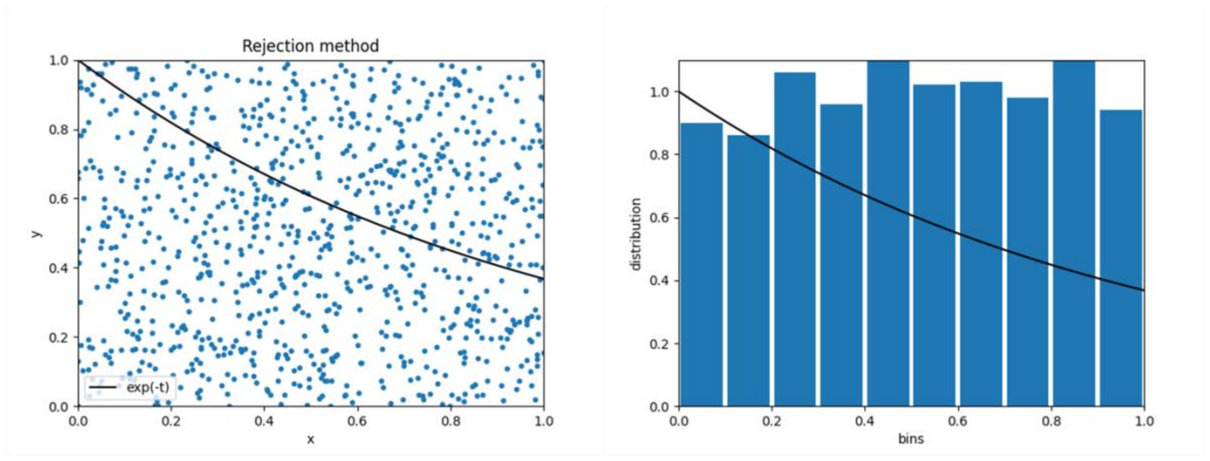

Figure 1: Left panel: Distribution of points from a random number generator. Right panel: Binned values of the random distribution before rejection method applied.

The rejection algorithm is applied, while y values lie above the function a new random x value is generated until this condition is no longer true. 3526 x values were rejected and replotted from an original sample size of 1000.

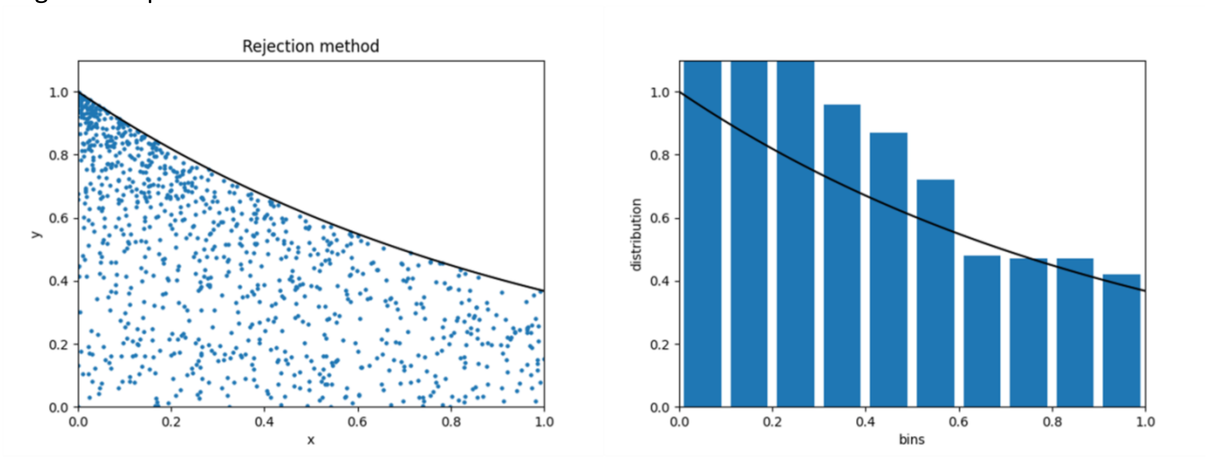

Figure 2: Left panel: Distribution of points from a random number generator after rejection method applied. Right panel: Binned values of the random distribution after rejection method applied. (10 bins).

Figure two demonstrates the random draws are approaching the distribution $P(\mathrm{x})$, although since the x values are resampled there is a spike in the distribution at low x values as expected. This rejection method is computationally wasteful as it blindly re-plots draws until a condition is met, in this case it took 3526 draws. A variation to this method is to replot x and y values, this will remove the slanted distribution in figure 2. Taking a larger number of random numbers and increasing the range of x values gives a cleaner, more accurate distribution.

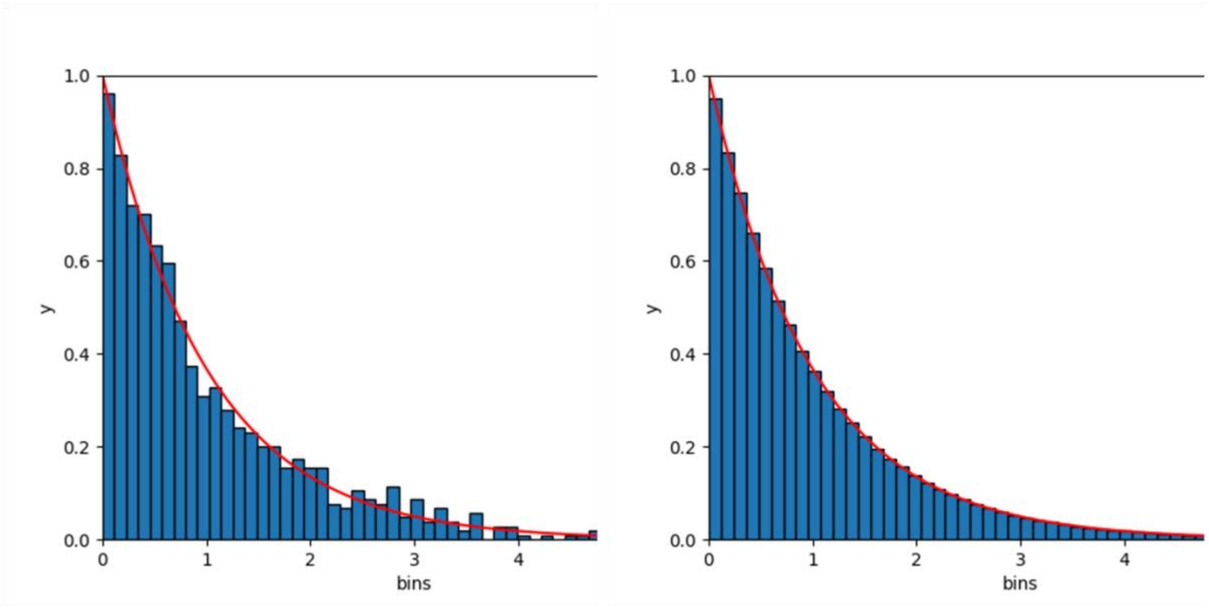

Figure 3: Left panel: Binned distribution of 1,000 points from a random number generator after rejection method applied. Right panel: Binned distribution of 1,000,000 points from a random number generator after rejection method applied. (50 bins).

Increasing the sampling size smoothens out the distribution. Although this method is computationally inefficient, it can be seen the rejection method is robust and straight forward.

Cumulative method

Alternatively, the cumulative function approach samples values using the cumulative distribution function of $P(\mathrm{x})$ denoted by $P_{cum}(\mathrm{x})$ and defined as

$$P_{cum}(\mathrm{x}) = \int_0^x P(x')dx'. \tag{2}$$

As given by the integral, the cumulative distribution increases monotonically with $P(\mathrm{x})$. By drawing y values from a flat distribution between 0 and 1 onto the cumulative distribution function. The corresponding x values follow the distribution of $P(\mathrm{x})$. When $P(\mathrm{x}) = \exp(-x)$, x values are sampled according to:

$$y = P_{cum}(\mathrm{x}) \Rightarrow \mathrm{x} = -\ln(1-\mathrm{y}). \tag{3}$$

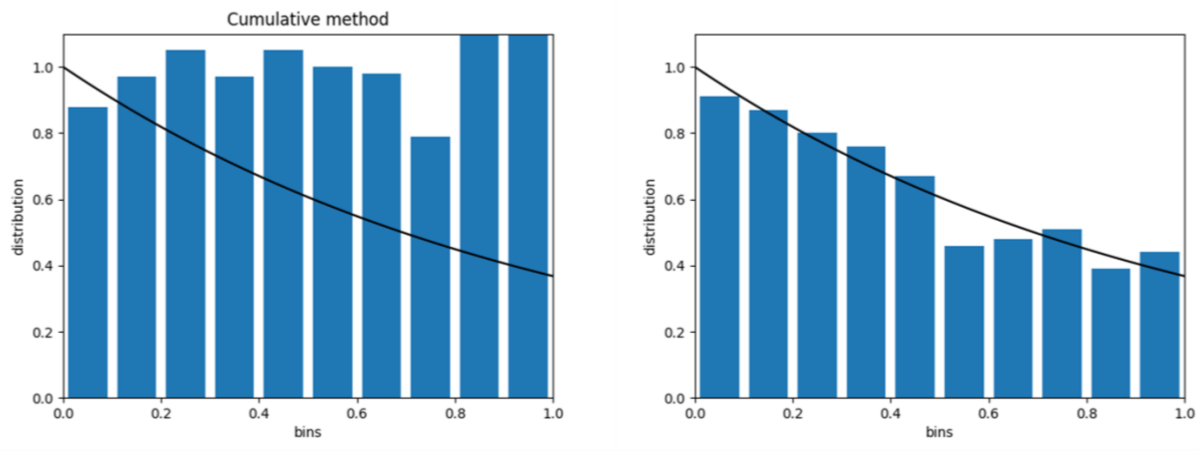

Figure 4: Left panel: Binned distribution of points from a random number generator. Right panel: Binned values of the random distribution after cumulative method applied. (10 bins).

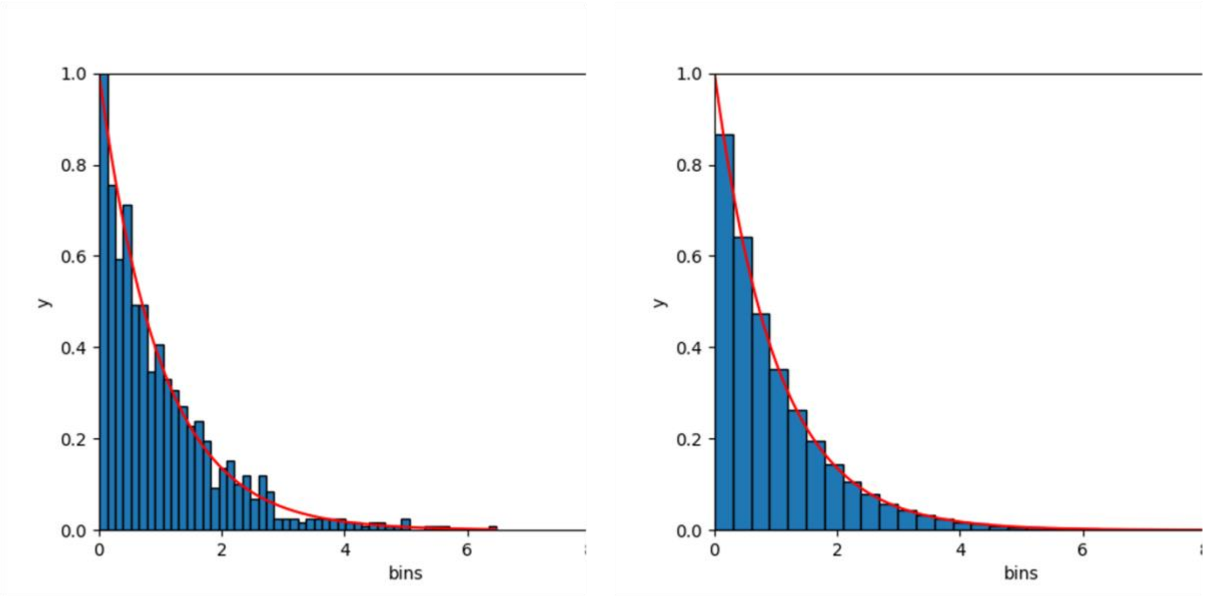

Figure 5: Left panel: Binned distribution of 1,000 points using the cumulative frequency method. Right panel: Binned distribution of 1,000,000 points using the cumulative frequency method. (50 bins).

With a large number of samples, the distribution approaches $P(\mathrm{x})$, this method does not reject values making it a faster method to achieve the same distribution as in figure 3. Using a larger number of samples the execution time of the rejection method increases at a faster rate than he cumulative method.

The cumulative method is more efficient since it avoids having to re-sample values by sampling $P(\mathrm{x})$ directly. At 1000 data points the cumulative method consistently has an execution time ~ 1.28 times faster than the rejection method. At 1,000,000 data points the cumulative execution time is time ~ 1.6 times faster. Furthermore, the rejection method has larger complex time since the algorithm must re-sample data until a condition is met, resulting in more iterations than the cumulative method.

Question 1b: *Monte Carlo scattering, isotropic*

This exercise involved modelling the isotropic scattering of 1,000,000 photons, following a flowchart for a numerical implementation of Monte Carlo radiative transfer.

In this code I have defined z values less than 0 ($z\ min$) as the wrong direction and z values greater than 1 ($z\ max$) as having escaped.

Initially I created structures and functions to use within the main body of code. I created a polar vector and cartesian vector struct, to store polar and cartesian co-ordinate variables respectively. I created a polar vector function to generate the polar co-ordinates r, θ and φ. Where (mean free path) r is calculated as the ratio of a random number obeying the distribution of $P(\mathrm{x})$ against optical depth. θ is calculated from a non-flat distribution obeying cos in the range π and 0. φ is calculated randomly in the range π to $-\pi$, z is also calculated in this function as $z = r\cos(\theta)$. I have a cartesian vector function which makes the standard transformations from polar co-ordinates to cartesian co-ordinates. (This cartesian function was not used in following the flowchart, although was used for plots and monitoring x, y and z. co-ordinates)

For the main body of the code, I ran a while loop until the condition of 1,000,000 photons being binned has been satisfied. At the start of the loop, I call the polar function and struct previously mentioned, to generate initial conditions for the photon. While this photon is in the range of $z\ max$ and $z\ min$ it undergoes a scatter/absorption check based on albedo. If the photon undergoes scattering additional values are generated using the polar vector function, the initially calculated z value is summed to the new additional values, all the other values are replaced. If the photon is absorbed, it is removed from this while loop and removed from the count. Photons stay in this while loop until it either goes in the wrong direction, in which case it is removed and the counter is set back, or the photon escapes (> $z\ max$) and this photon is then stored in bins of equal size angel. (The binning was completed in python)

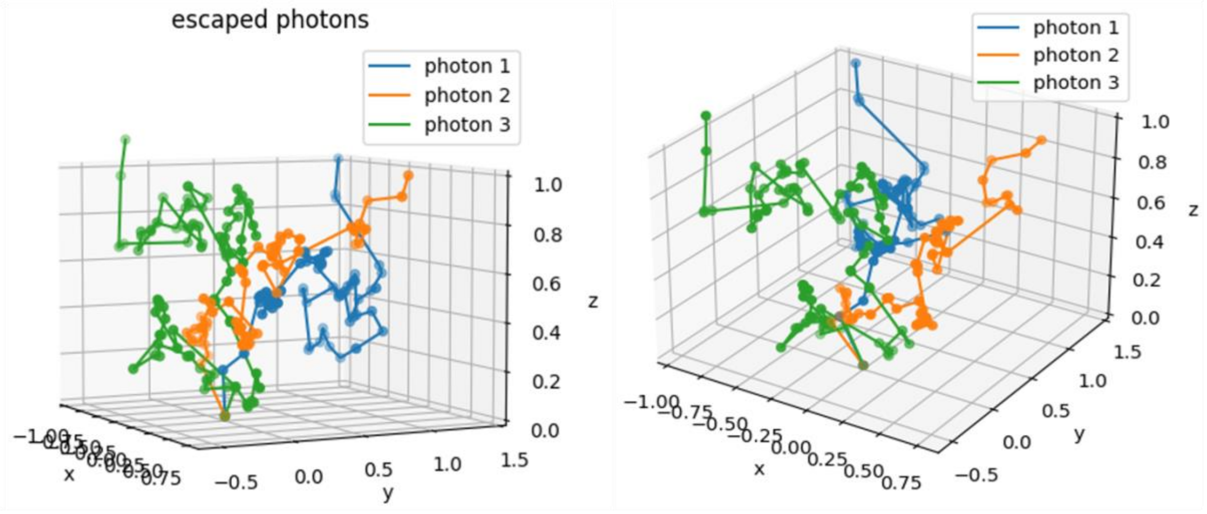

Figure 4: Trace of x, y and z co-ordinates of the first three photons to escape (> $z\ max$).

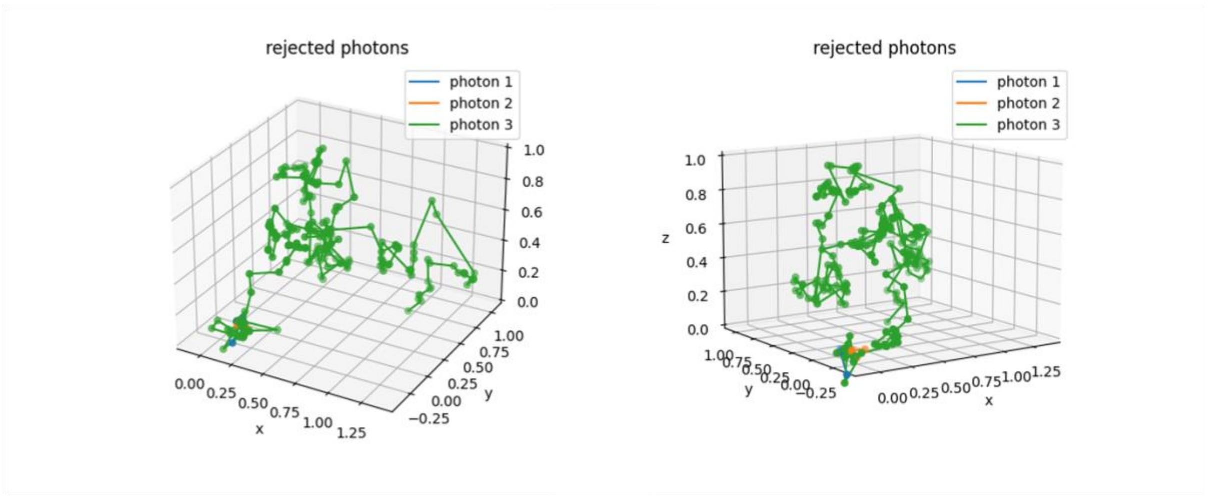

Figure 5: Trace of x, y and z co-ordinates of the first three photons to go in the wrong direction ($< z\ min$).

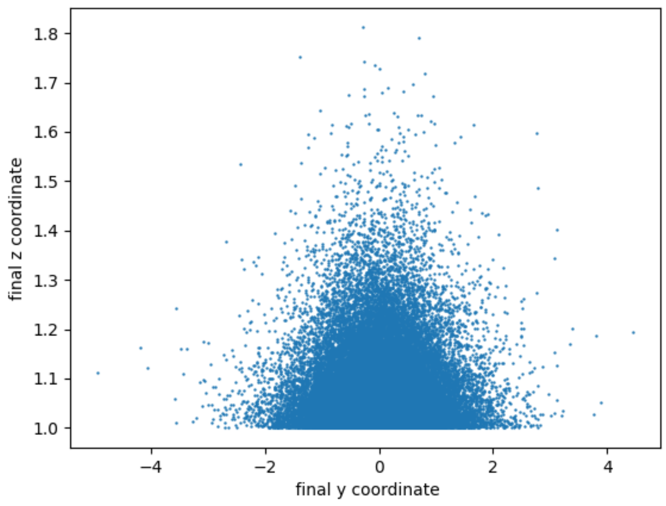

Figure 6: Trace of final y and z co-ordinates, displaying photon distribution.

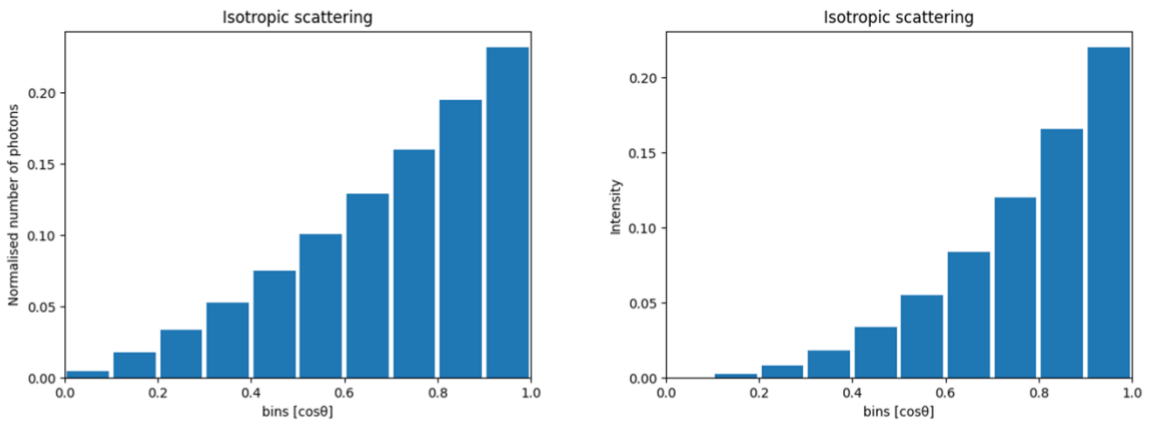

Figure 7: Left panel: Binned normalised values of photons (No in bin/No of photons). Right panel: Binned intensity of photons (Normalised value/midpoint value). The x axis is midpoints from 0 to 1, translated from $\cos(\pi/2)$ to $\cos(0)$.

Question 1c: *Rayleigh scattering*

This exercise involved modelling the Rayleigh scattering of blue photons and other colours, these photons have an alpha value of 10 and 0.1 respectively.

Please note with more time I would have calculated final positions at z = 1, rather than the end of the final path.

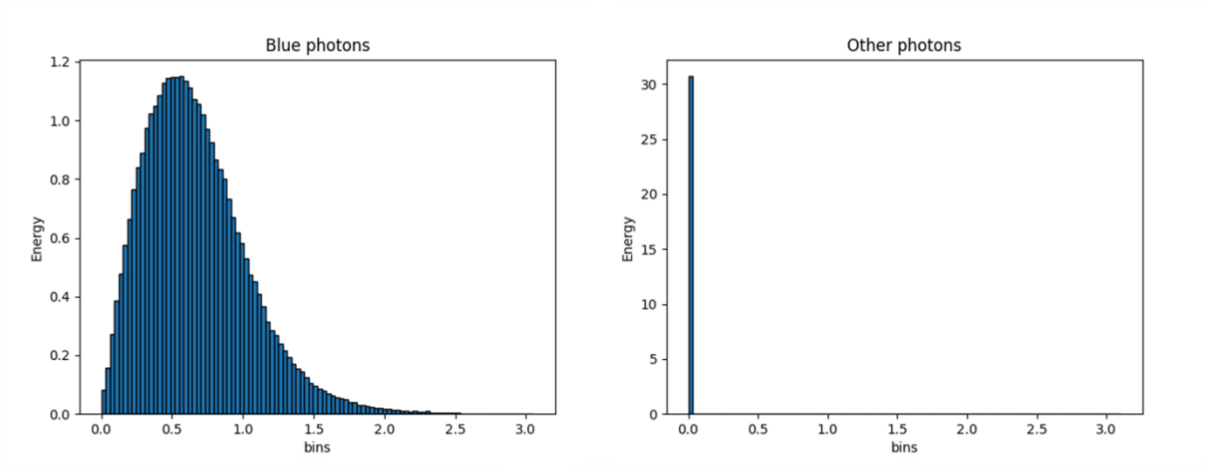

Figure 8: Left panel: Binned final theta values of blue photons. Right panel: Binned theta values of other photons.

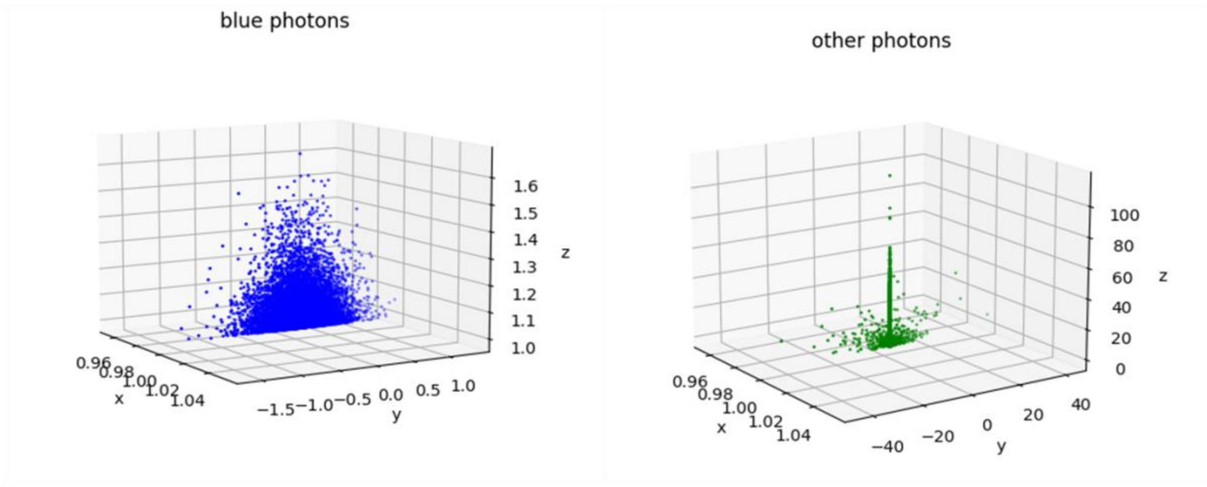

Figure 9: Left panel: Final position of 1,000 blue photons. Right panel: Final position of 1,000 other photons.

The colour of the sky is blue due to the scattering of light from the sun at large angles. Photons scatter off the atmosphere. This scattering is known as Rayleigh scattering. I modelled the Rayleigh scattering of blue light which has an alpha value of 10 and other light which has a value of 0.1. This means the mean free path of scattered blue photons is significantly smaller than that of other light, resulting in denser distributions of blue light than other colours. This is represented by figure 9 which depicts the final cartesian distribution of photons. Blue photons also have a more even distribution of energy as shown by figure 8, resulting in a larger distribution of light when undergoing Rayleigh scattering. Evidently Rayleigh scattering is more effective for blue photons than other photons light

Rayleigh's scattering of blue light suggests the sky is blue, further analysis could be modelling Mie scattering to analyse particles larger than a wavelength.

Source code

```
#include <stdio.h> // pre-compiler statement
#include <stdlib.h> // free
#include <math.h> // pre-compiler statement, remember to LINK
EXPLICITLY using "-lm"
#include <time.h> // clock



/**
 * Random number generator
 * @return
 * a random number in the range 0 to 1
 */
double random_number()
{
    static int64_t seed = 1;
    static int32_t m = 2147483647;
    static int16_t a = 16807;
    static int16_t c;

    seed = (a * seed + c) % m;
    return (double) seed / m;
}

/**
 * Exponential distribution function
 * @param x
 * @return
 * y values obeying this function
 */
double f(double x)
{
    return exp(-x);
}


/**\
 * Cumulative distribtuion function
 * @param y
 * @return
 * x values obeying this cumulaitve function
 */
double f_cum(double y)
{
    return -log(1-y);
}

/**
 * polar vector used to store polar coordinates
 */
typedef struct polar_vector
{
    double r;
    double theta;
    double phi;
    double z;
} polar_vector;

/**
 * data type polar vector
 */
struct polar_vector r1;

/**
 * initialised vector for polar coordinates
 */
double outputPolarVector[4] = {0, 0, 0, 0};

/**
 * function calculating polar coordinates
 */
void generatePolarVector()
{
    /// r is the ratio of cumulative distribution and optical depth
    outputPolarVector[0] = (double) f_cum(random_number())/10;
    /// theta calculated from a non-flat distribution
    outputPolarVector[1] = (double) acos(2*random_number()-1);
    /// phi calculated using a random number generator
    outputPolarVector[2] = (double) random_number()*2*M_PI - M_PI;
    /// z is r multiplied by cos theta
    outputPolarVector[3] = (double) ((outputPolarVector[0])  *
cos(outputPolarVector[1]));
}

/// un-normalised Rayleigh function
double rayleigh(double x)
{
    return (3/(16*M_PI))*(1+(cos(x)*cos(x)));
}

/**
 * / initialised vector for cartesian coordinates
 */
double outputCartesianVector[3] = {0, 0, 0};

/**
 * function calculating cartesian coordinates
 * @param r
 * @param theta
 * @param phi
 */
void generateCartesianVector(double r, double theta, double phi)
{
    outputCartesianVector[0] = (double) r*sin(theta)*cos(phi);
    outputCartesianVector[1] = (double) r*sin(theta)*sin(phi);
    outputCartesianVector[2] = (double) r*cos(theta);
}

/**
 * // xyz vector used to store cartesian coordinates
 */
typedef struct xyz_vector
{
    double x;
    double y;
    double z;
} xyz_vector;

/**
 * data type cartesian vector
 */
struct xyz_vector r2;

/**
 * initialised matrix for rotation
 */
double rotationMatrix[4][4];
double inputMatrix[4][1] = {0.0, 0.0, 0.0, 0.0};
double outputMatrix[4][1] = {0.0, 0.0, 0.0, 0.0};

/**
 * matrix multiplication function
 */
void multiplyMatrix()
{
    /// combine inverse and original rotation to give the relative 
output
    for(int i = 0; i < 4; i++){
        for(int j = 0; j < 4; j++){
            outputMatrix[i][j] = 0;
            for(int k = 0; k < 4; k++){
                outputMatrix[i][j] += rotationMatrix[i][k] * 
inputMatrix[k][j];
            }
        }
    }
}

/**
 * rotation matrix function
 * @param angle
 *
 * @param u
 * @param v
 * @param w
 */
void RotationMatrix(double angle, double u, double v, double w)
{
    double L = (u*u + v*v + w*w);
    double u2 = u*u;
    double v2 = v*v;
    double w2 = w*w;
    /// ref - https://stackoverflow.com/questions/45160580/rotation-
about-an-arbitrary-axis-in-3-dimensions-using-matrix
    /// rotation about arbitrary axis in 3 dimensions
    rotationMatrix[0][0] = (u2 + (v2 + w2) * cos(angle)) / L;
    rotationMatrix[0][1] = (u * v * (1 - cos(angle)) - w * sqrt(L) * 
sin(angle)) / L;
    rotationMatrix[0][2] = (u * w * (1 - cos(angle)) + v * sqrt(L) * 
sin(angle)) / L;
    rotationMatrix[0][3] = 0.0;
    rotationMatrix[1][0] = (u * v * (1 - cos(angle)) + w * sqrt(L) *
sin(angle)) / L;
    rotationMatrix[1][1] = (v2 + (u2 + w2) * cos(angle)) / L;
    rotationMatrix[1][2] = (v * w * (1 - cos(angle)) - u * sqrt(L) *
sin(angle)) / L;
    rotationMatrix[1][3] = 0.0;
    rotationMatrix[2][0] = (u * w * (1 - cos(angle)) - v * sqrt(L) *
sin(angle)) / L;
    rotationMatrix[2][1] = (v * w * (1 - cos(angle)) + u * sqrt(L) *
sin(angle)) / L;
    rotationMatrix[2][2] = (w2 + (u2 + v2) * cos(angle)) / L;
    rotationMatrix[2][3] = 0;
    rotationMatrix[3][0] = 0;
    rotationMatrix[3][1] = 0;
    rotationMatrix[3][2] = 0;
    rotationMatrix[3][3] = 1;
}

/**
 * initialised vector for polar coordinates
 */
double outputRayleighScattering[3] = {0, 0, 0};

/**
 * function generating Rayleigh scattering polar coordinates
 * @param tau
 */
void generateRayleighScattering(float tau)
{

    /// r is the ratio of cumulative distribution and optical depth
    outputRayleighScattering[0] = (double) f_cum(random_number())/tau;

    /// theta calculated from a non-flat distribution using the
rejection method
    /// obeying the Rayleigh phase function
    double theta_vals = (double) acos(2*random_number()-1);
    double y_vals = (double) (3/(8*M_PI))*random_number();
    while (y_vals > rayleigh(theta_vals))
    {
        theta_vals = (double) acos(2*random_number()-1);
        y_vals = (double) (3/(8*M_PI))*random_number();
    }
    outputRayleighScattering[1] = (double) theta_vals;

    /// phi calculated using a random number generator
    outputRayleighScattering[2] = (double) random_number()*2*M_PI -
M_PI;
}

/**
 * initialised vector for polar coordinates
 */
double Generate_rotation[3] = {0, 0, 0};

/**
 * function rotating Rayleigh coordinates and tracking values
 * @param r
 * @param theta
 * @param phi
 * @param x
 * @param y
 * @param z
 */
void generateRotation(double r, double theta, double phi, double x,
double y, double z) {
    /**
     * xyz vector used to store initial and final cartesian co-
ordinates
     */
    typedef struct xyz_vector {
        double x1;
        double y1;
        double z1;
        double x2;
        double y2;
        double z2;
    } xyz_vector;

    /**
     * data type cartesian vector
     */
    struct xyz_vector r3;

    /// old xyz points
    r3.x1 = x;
    r3.y1 = y;
    r3.z1 = z;

    /// generate new cartesian co-ordinates from polars
    generateCartesianVector(r, theta, phi);

    /// new xyz points
    r3.x2 = outputCartesianVector[0];
    r3.y2 = outputCartesianVector[1];
    r3.z2 = outputCartesianVector[2];

    /// initial points to transform
    inputMatrix[0][0] = r3.x2;
    inputMatrix[1][0] = r3.y2;
    inputMatrix[2][0] = r3.z2;
    inputMatrix[3][0] = 1;

    /// axis vector
    double u = r3.x2 - r3.x1;
    double v = r3.y2 - r3.y1;
    double w = r3.z2 - r3.z1;

    /// rotation to generate arbitrary axis and combine matrixs to give
output
    RotationMatrix(theta, u, v, w);
    multiplyMatrix();

    /// output co-ordiantes
    Generate_rotation[0] = outputMatrix[0][0];
    Generate_rotation[1] = outputMatrix[1][0];
    Generate_rotation[2] = outputMatrix[2][0];

}

int main()
{
    // 1 a

    /// initial variables
    int i, i_bin;
    int Naccept = 0;
    int N_samples = 1000000;
    int fmax = 1;

    /// call malloc to allocate heap space
    double *X = malloc(N_samples*sizeof(double));
    double *Y = malloc(N_samples*sizeof(double));

    // 1 b

    /// initial variables
    int N = 1000000;
    int N_remove = 0;
    int N_bins = 10;
    int binned[N_bins];
    double zmax = 1.0;
    double albedo = 1.0;
    double dx = 1./N_bins;

    // 1 c

    /// initial variables
    double x_init = 0;
    double y_init = 0;
    double z_init = 0;
    double blue_t = 10;
    double other_t = 0.1;
    double u, v, w;

    //
########################################################################
######
    //////////////////////////////////////  1 a
///////////////////////////////////
    printf("############## 1 a ##############\n");
    //
########################################################################
######


    /// handling errors
    if ( (X == NULL) || (Y == NULL) ){
        printf("ERROR: malloc failed! \n");
        exit(0);
    }

    /// Generate 1000 random numbers for X and Y in the range 0-1 and
0-y_max
```

```
    for (i=0; i<N_samples; i++)
    {
        X[i] = (double) random_number()*6;
        Y[i] = (double) random_number() * fmax;
    }

    /// clock begins to measure time for the rejection method
    clock_t rbegin = clock();

    /// store output as a txt file to plot in python
    FILE *fp=NULL;
    fp=fopen("rejection_method.text", "w");

    /// Rejection Method
    for (i = 0; i < N_samples; i++) {

        /// rejecting and reassigning x values
        while (Y[i] > f(X[i])){
            X[i] = (double) random_number()*6;
            Y[i] = (double) random_number() * fmax;
        }

        /// accepted x values
        fprintf(fp, "%lf\n", X[i]);

    }

    /// clock ends
    clock_t rend = clock();
    double rejection_time_spent = (double)(rend - rbegin) /
CLOCKS_PER_SEC;
    printf("rejection method: %1.8f us\n",
rejection_time_spent*1000000);


    /// re-initialise values
    for (i=0; i<N_samples; i++)
    {
        Y[i] = (double) random_number() * fmax;
    }

    /// clock begins to measure time for the cumulative method
    clock_t begin = clock();

    /// store output as a txt file, to then plot
    FILE *fp1=NULL;
    fp1=fopen("cumulative_method.text", "w");

    /// Cumulative method
    for (i = 0; i < N_samples; i++) {
        X[i] = f_cum(Y[i]);

        fprintf(fp1, "%lf\n", X[i]);
    }

    /// clock ends
    clock_t end = clock();
    double time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
    printf("cumulative method: %1.8f us\n", time_spent*1000000);


    /// deallocating memory
    free(X);
    X = NULL;
    free(Y);
    Y = NULL;

    //
########################################################################
######
    //////////////////////////////////////  1 b
///////////////////////////////////
    printf("############## 1 b ##############\n");
    //
########################################################################
######

    /// clear out the bins first and initialise values
    for (i = 0; i < N_bins; i++)
        binned[i] = 0;

    Naccept = 0;


    /// Generate photons
    while (Naccept < N) {
        /// While the total number of photons that have beena accepted
is less than N = 1,000,000
        /// photons are generated in polar and z co-ordinates
        generatePolarVector();
        r1.r = outputPolarVector[0];
        r1.theta = outputPolarVector[1];
        r1.phi = outputPolarVector[2];
        r1.z = outputPolarVector[3];
        /// while the photon is between z = z_min = 0 and z = z_max = 1
        while (r1.z <= zmax && r1.z >= 0) {
            /// check for scattering or absorption
            if (albedo >= random_number()) {
                /// if the albedo is larger than a random probability
                /// the photon undergoes scattering and new polar and z
values are generated
                generatePolarVector();
                r1.r = outputPolarVector[0];
                r1.theta = outputPolarVector[1];
                r1.phi = outputPolarVector[2];
                /// z value is updated
                r1.z = outputPolarVector[3] + r1.z;
            } else {
                /// if the albedo is less than the random probability
                /// the photon undergoes absorption and is removed
                r1.z = 0;
                break;
            }
        }
        /// photons have escaped the while loop condition
        if (r1.z >= zmax) {
            /// photons in the correct direction
            /// bin the photons respectively between 10 bins
            i_bin = (int) (cos(r1.theta) / dx);
            binned[i_bin]++;
            Naccept++;

        }
            /// photons have gone in the wrong direction or have been
absorbed
        else {
            /// counter of removed photons increases
            N_remove++;
        }
    }

    /// update these values for bins
    printf("midpoint, Number of photons binned\n");
    for (i = 0; i < N_bins; i++) {
        printf("%1.3f,  %d\n", ((i + 0.5) * dx), binned[i]);
    }



    //
########################################################################
######
    ////////////////////////////////////  1 c
///////////////////////////////////
    printf("############## 1 c ##############\n");
    //
########################################################################
######

    /// initialize values
    Naccept = 0;
    /// store output as a txt file, to then plot
    FILE *fp5=NULL;
    fp5=fopen("XYZ_other.text", "w");


    while (Naccept < N)
    {
        /// generate initial polar values for scattering
        generateRayleighScattering(other_t);
        r1.r = outputRayleighScattering[0];
        r1.theta = 0;
        r1.phi = outputRayleighScattering[2];
        /// generated rotation matrix following the initial polar
conditions
        generateRotation(r1.r, r1.theta, r1.phi, x_init, y_init,
z_init);
        /// initial injected xyz co-ordinates
        r2.x = Generate_rotation[0];
        r2.y = Generate_rotation[1];
        r2.z = Generate_rotation[2];

        while ( r2.z <= zmax && r2.z >= 0 ){
            /// check for scattering or absorption
            if (albedo >= random_number()) {
                /// if the albedo is larger than a random probability
                /// the photon undergoes Rayleigh scattering
                generateRayleighScattering(other_t);
                /// polar values re-generated
                r1.r = outputRayleighScattering[0];
                r1.theta = outputRayleighScattering[1];
                r1.phi = outputRayleighScattering[2];
                /// rotation matrix updated according to polar co-
ordinates
                generateRotation(r1.r, r1.theta, r1.phi, r2.x, r2.y,
r2.z);
                r2.x = Generate_rotation[0] + r2.x;
                r2.y = Generate_rotation[1] + r2.y;
                r2.z = Generate_rotation[2] + r2.z;


            } else {
                /// if the albedo is less than the random probability
                /// the photon undergoes absorption
                r2.z = 0;
                break;
            }
        }
        if (r2.z >= zmax){
            /// Tracking the number of accepted photons and storing
cartesian co-ordiantes
            Naccept++;
            fprintf(fp5, "%lf\t%lf\t%lf\n", r1.z, r2.y, r2.z);

        }
            /// photons have gone in the wrong direction or have been
absorbed
        else {
            /// counter of removed photons increases
            N_remove++;
        }

        }
    printf("The sky is blue");

    return 0;
}
```