Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана (национальный исследовательский университет)» (МГТУ им. Н.Э. Баумана)

Лабораторная работа

По дисциплине: «Вычислительная физика»

Тема: «Локальная оценка потока»

Выполнила: В.К. Сафонова

Студентка группы: ФН4-81

Проверил: Р.Х. Хасаншин

г. Москва 2021 г.

Оглавление

Теоретическая часть

Решение уравнения Фредгольма 2-го рода посредством метода Монте-Карло.

$$\Psi(\vec{r},\vec{E})=q(\vec{r},\vec{E})+\iint K(\vec{r'},\vec{E'},\vec{r},\vec{E})\,d\vec{r'}\,d\vec{E'}\,\Psi(\vec{r'},\vec{E'}) \qquad (1)$$

Это – уравнение Фредгольма 2-го рода. Здесь: – решение уравнения (1)

Исследования и вероятностная интерпретация многомерных интегральных уравнений Фредгольма 2-го рода имеет существенное значение для метода Монте-Карло. С другой стороны, использование алгоритмов случайного блуждания частиц позволяет построить эффективный модифицированный метод Монте-Карло.

Обозначим точку в 6-мерном фазовом пространстве через $x=(\vec{r},\vec{E})\in\Gamma$. Случайное блуждание частиц в некоторой среде представляет собой конечное число состояний $(x_1,x_2,\ldots,x_k)$.

Для описания траектории частиц достаточно ввести следующий набор параметров:

1. плотность вероятности первого столкновения $P_1(x_1)$;
2. плотность вероятности перехода из точки $x'\in\Gamma$ в точку $x\in\Gamma\,P(x',x)$;
3. плотность вероятности поглощения $P(x)$.

На предложенные функции накладывают следующие условия:

$$\int_\Gamma P_1(x)\,dx=1;\,P_1(x)>0$$

$$\int_\Gamma P(x',x)\,dx'=1=P(X);\,P(x)\geq 0$$

Таким образом, столкновения в окрестности точки x может быть первым или последующим после столкновения в точке x'.

$$x:\Psi(x)=P_1(x)+\int_\Gamma P(x',x)\,\Psi(x')\,dx' \qquad (2)$$

Уравнение (2) является вероятностным аналогом уравнения (1).

Моделирование называется аналоговым, если при расчёте методом Монте-Карло используются реальные вероятностные законы. При построении схемы использовали аналоговый метод.

Рассмотрим функционал:

$$I_g = \iint \Psi(\vec{r},\vec{E}) g(\vec{r},\vec{E}) d\vec{r}\, d\vec{E} \qquad (3)$$

Наша задача – построить случайную величину (оценку), математическое ожидание которой будет совпадать со значением формулы (3).

Для решения в методе Монте-Карло используют разностную оценку. Рассмотрим оценку по столкновению:

$$\eta(\alpha) = \sum_{m=1}^{k} W_m(\alpha) g(x_m)$$

Эта оценка рассчитывается по всем точкам траектории, где W_m – вес частицы, испытавшей m столкновений.

$$W_m = \frac{q_1(x_1)}{p_1(x_1)} W(x_1,x_2) W(x_2,x_3) \ldots W(x_{m-1},x_m),$$

где $q_1(x_1)$ – плотность первых столкновений;

$$W(x',x) = \begin{cases} \dfrac{K(x',x)}{p(x',x)}, p(x',x) \neq 0 \\ 0, p(x',x) = 0 \end{cases}$$

Локальная оценка потока

В методе Монте-Карло доказывается, что оценка:

$$\eta(x) = \sum_{m=1}^{k} W_m(\alpha) g(x_m)$$

является несмещенной оценкой I_g для плотности столкновений, так как вклад рассеянного излучения $q_1(\vec{r},\vec{E})$ всегда можно оценить аналитически. При дальнейшем рассмотрении это слагаемое опустим.

$$q_1(\vec{r},\vec{E}) = \int d\vec{r'}\, q(\vec{r},\vec{E}) T(\vec{r'},\vec{r} \vee \vec{E})$$

Функционалы типа

$$I_g = \int_{\Gamma} \Psi(x) g(x) dx$$

позволяют вычислить среднее значение плотности столкновения и плотности потоков по всей области задачи интересующей функции и если градиент поля велик, то оценки дадут очень грубый результат.

Для повышения точности разбивают пространство на подпространства, которые позволяют стянуть оценку детектора в точку, при этом в каждой точке взаимодействия с детектором учитываются виртуальные вклады из каждой точки взаимодействия.

Построим локальную оценку потока. Нас интересует плотность потока Ψ. Поставим точку в уравнение плотности столкновений. Так как будут интересовать рассеянные фотоны, то поделим обе части (1) на Σ_S.

$$\varphi(x^*) = \int \frac{K(x', x^*)}{\Sigma_s(x^*)} \Psi(x') dx'$$

$$\eta(\alpha) = \sum_{m=1}^{k} W_m(\alpha) \frac{K(x', x^*)}{\Sigma_s(x^*)}$$

Ядро интегрального уравнения имеет δ-функцию, от которой можно избавиться путем интегрирования по некоторой области $\Delta\vec{\Omega}^*$. Для оценки потока получим величину:

$$\eta_1(\alpha) = \sum_{m=1}^{k} W_m(\alpha) \frac{\exp[-\tau(\vec{r}_m; \vec{r}^*; E_m)]}{\left|\vec{r}_m - \vec{r}^{*2}\right|} \frac{\Sigma_d(\vec{r}_m; E_m \to E_m^*)}{\Sigma(\vec{r}_m^*; E_m^*)}$$
$$\Delta(E_m; \Delta E_m) \Delta(\vec{\Omega}_m; \Delta\vec{\Omega}_m),$$

$$\Delta(x; \Delta x) = \begin{cases} 1, x \in (x'; x' + \Delta x) \\ 0, x \notin (x'; x' + \Delta x) \end{cases}$$

Определение параметров частицы после столкновения

Параметры после столкновения включают энергию и направление движения рассеянной первичной частицы. Они определяются ядром столкновений и поперечными сечениями (дифференциальными и интегральными).

Если произошёл фотоэффект и нас не интересует история фотоэлектрона, то мы переходим к следующему фотону из заданной статистики. Если же произошло комптоновское рассеяние, то мы вынуждены рассматривать историю рассеянного фотона и направление его движения. При комптоновском рассеянии фотон с первоначальной энергией E' в результате взаимодействия с электроном передаёт ему часть энергии и изменяет направление своего движения. Поскольку скорость атомных электронов очень мала по сравнению со скоростью света, то можно считать электрон свободным и покоящимся.

Энергия рассеянного фотона и угол рассеяния связаны формулой:

$$E=\frac{E'}{1+\frac{E'}{mc^2}(1-cos\theta)}.$$

Введя обозначение $\alpha=\frac{E}{m_0c^2}$ и $\alpha'=\frac{E'}{m_0c^2}$ получим выражение энергии рассеянного фотона в единицах энергии массы покоя электрона

$$\alpha=\frac{\alpha'}{1+\alpha'(1-cos\theta)}.$$

Алгоритм программы

1. W = 1; N = 0;
2. N = N+1;
3. Розыгрыш точки x_0, y_0, z_0 рождения, энергии и направления вылета фотона;
4. Розыгрыш длины свободного пробега;
5. Расчёт координат точки взаимодействия x_1, y_1, z_1;
6. Проверка нахождения частицы в области задачи, если в области, то выполняется следующий шаг, иначе переходим в пункт 3;
7. Розыгрыш типа взаимодействия. Если взаимодействие – комптоновское, то продолжаем, иначе переходим в пункт 3;
8. Расчёт параметров рассеянного фотона. Определение энергетической группы k; проверка разыгранной удельной энергии, если она не входит в заданные пределы, то переходим в пункт 3;
9. Переход к старой системе координат;
10. Расчёт веса частицы $W(k) = W\frac{(k)*\Sigma_d}{\Sigma}$. Если $W(k) > 10^{-11}$, то переходим на следующую ступень, иначе – в пункт 3;
11. Вычисление отношения дифференциальной и полной сечений комптоновского рассеяния, а также расстояние до детектора
12. Суммирование виртуальных вкладов в показание детектора
13. Розыгрыш длины свободного пробега L_2
14. $x_0=x_1$; $y_0=y_1$; $z_0=z_1$; $L_1=L_2$;
15. Если $N < N_{MAX}$, то переходим в пункт 5, иначе рассчитываем и выводим оценку плотности потока

Условие задачи

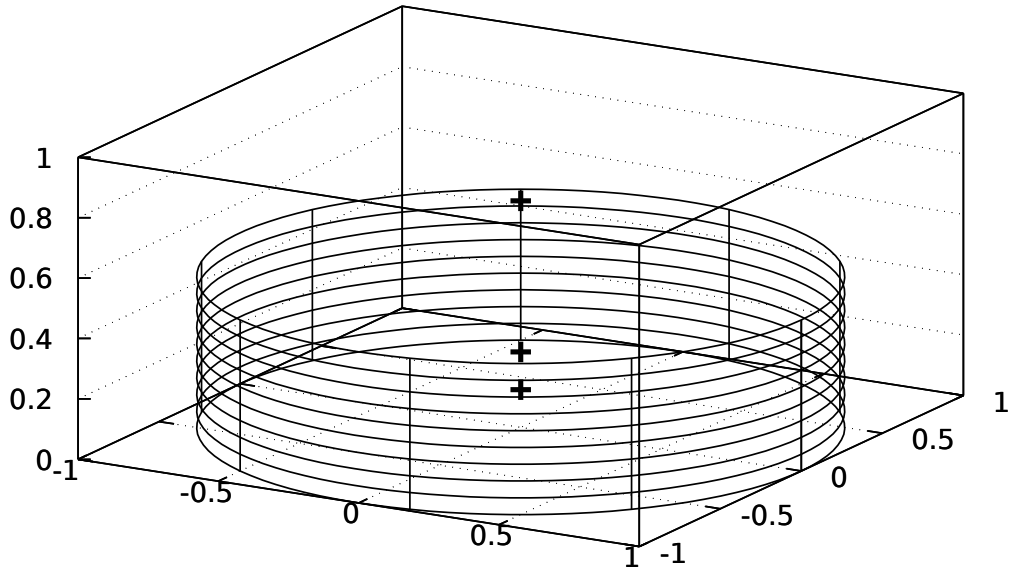

Рис. 1: Границы «саркофага» и положение детекторов.

Источник (см. рис. 2) представляет из себя эллипс, вписанный в нижнее основание цилиндра.

Результаты

Для наглядности далее результаты будут предложены для разного числа частиц.

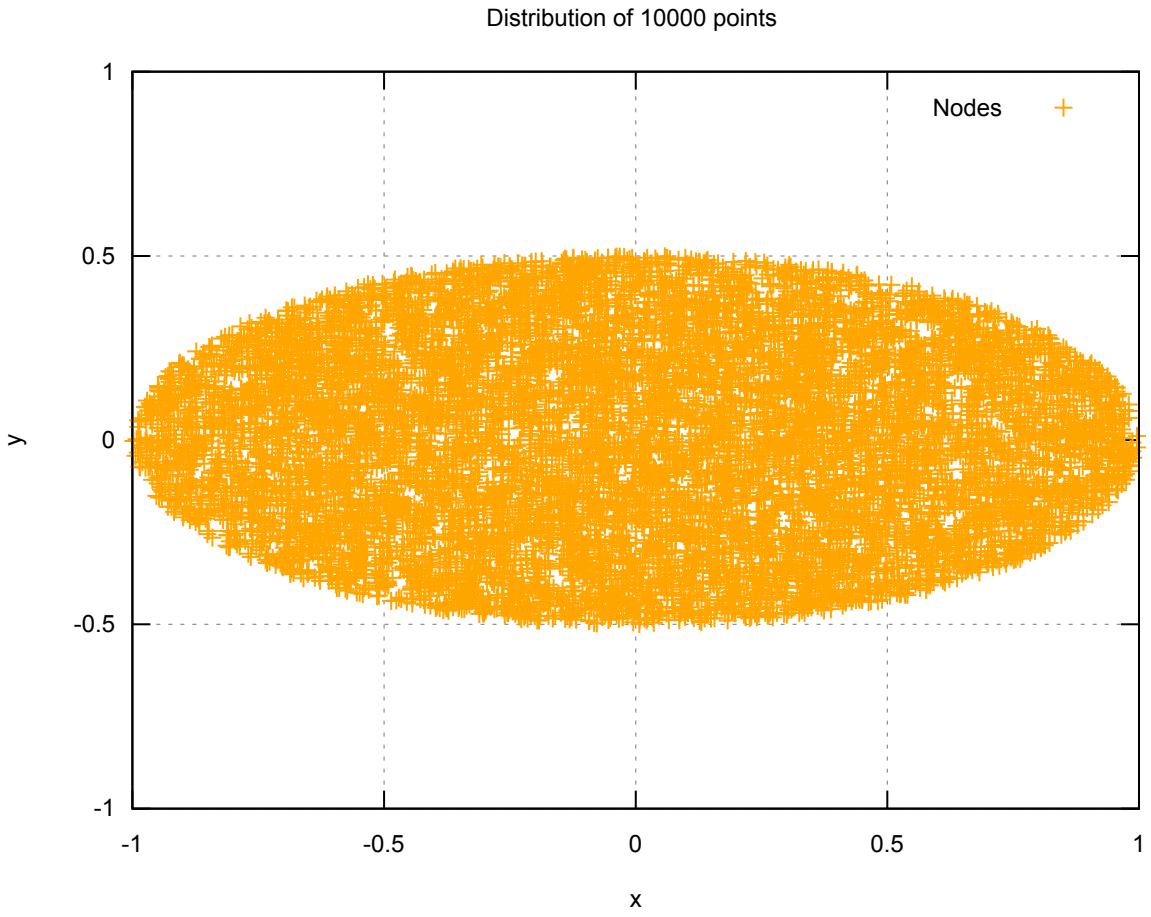

Рис. 2: Равномерное распределение точек рождения.

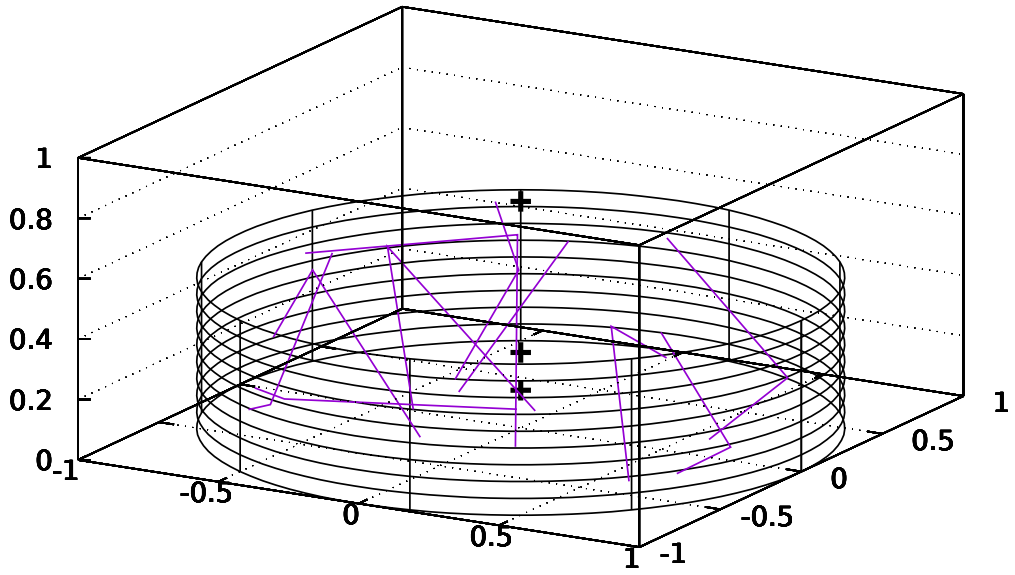

Рис. 3: Траектории движения частиц для небольшого числа точек.

Наибольший практический интерес представляют данные с детектора, находящегося внутри материала «саркофага». В условиях данной задачи материалом является алюминий (Al).

Для удобства представления результатов частицы были разделены на энергетические группы «шириной» по 0.1 МэВ каждая. Первой группе соответствует значение 0.1 МэВ, последней – 2 МэВ.

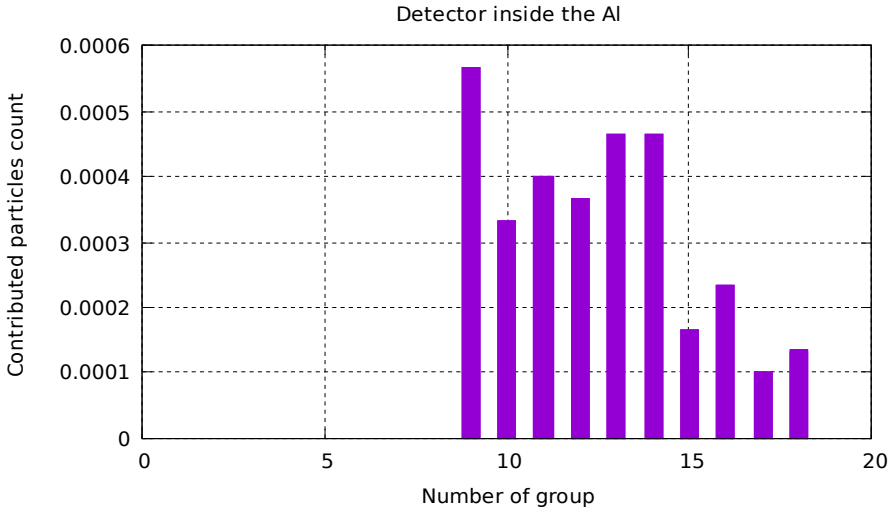

Рис. 4: Гистограмма количества частиц, распределённых по энергетическим группам , вклад которых детектор зафиксировал относительно начального количества частиц.

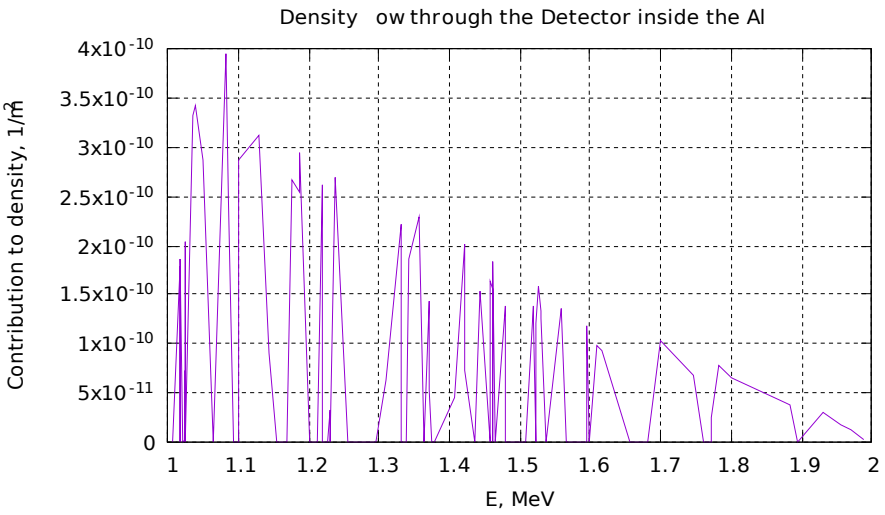

Рис. 5: Вклад в детектор, находящийся в алюминии частиц с определённой энергией.

Для сравнения имеет смысл рассмотреть вклад частиц в детектор, находящийся на границе раздела «воздух-алюмий». Соответствующие графики представлены на рис. 6 и 7.

Очевидно графики, демонстрирующий то, какие группы внести свой вклад в детектор, будут совпадать для детекторов, находящихся в воздухе.

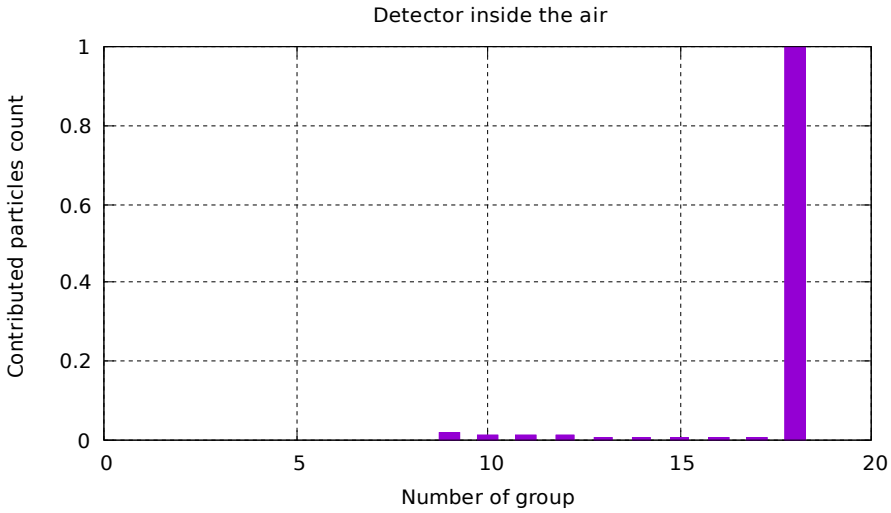

Рис. 6: Гистограмма количества частиц, распределённых по энергетическим группам, вклад которых детектор зафиксировал, относительно начального количества частиц.

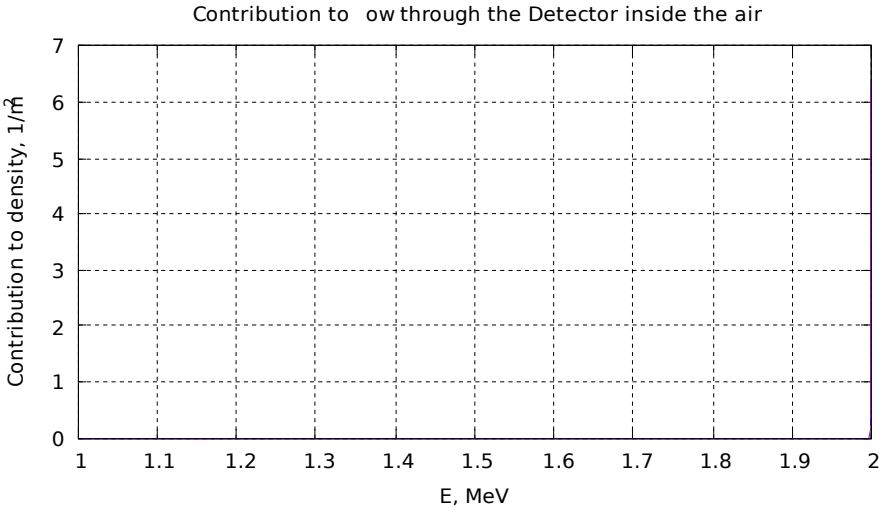

Рис. 7: Вклад в детектор, находящийся в алюминии частиц с определённой энергией.

К сожалению представленные на рис. 7 данные не дают качественных заключений о вкладе частиц, находящихся вне последней энергетической группы в детектор, в следствие много большего вклада от частиц ещё не взаимодействующих.

Поскольку практический смысл имеет только оценка по всей траектории, представление графика без последней энергетической группы так же не даёт значимых данных.

Также ниже представлена таблица средних значений потока сквозь детекторы, указанные в условии задачи.

№ детектора	Среда, в которой находится детектор	Координаты детектора (x, y, z)	Среднее значение потока
1.	Воздух	(0, 0, 0.125)	2.37724
2.	Воздух	(0, 0, 0.25)	1.30123
3.	Алюминий	(0, 0, 0.75)	8.29957e-11

Из таблицы легко видеть, что поток быстро убывает с увеличением расстояния от источника. Средний поток, проходящий через детектор, находящийся в алюминии много меньше потока, проходящего сквозь детекторы в воздухе.

Выводы

Посредством метода «Монте-Карло» было смоделировано явления прохождения частиц (γ-квантов) сквозь комплексную среду, представленную воздухом и алюминием.

Была проведена локальная оценка потока в указанных точках – на границах раздела и в рассматриваемых средах.

Приложения

Программа для решения задачи на языке C++

```
/* Note that you need the GNUPlot.
 *
 * macOS
 *
 * The easiest way to install it thought the Homebrew.
 * If you are not familiar with homebrew, read more about it here:
https://brew.sh/
 * To install GNUPlot:
 * brew install gnuplot
 *
 * Linux
 *
 * You know how this works, don't you?
 */

#include <iostream>
#include <cmath>
#include <random>
#include <vector>
#include <utility>
#include <fstream>
#include <string>
#include <tuple>
#include <array>
#include <algorithm>
#include <iterator>
#include <memory>
#include <stdexcept>

const int N = 3.0e4; //Number of points. //Do not use more than 0.5e4 on old
computers!

const double R = 1;
const double pi = 3.14159265359;

double E_min = 1.0e5;
const double E_0 = 2.0e6;
const double E_e = 0.51099895e6;

const double group_range = 1.0e5;

const int number_of_energy_groups = (E_0 - E_min) / group_range + 1; //Well,
it's not safe I know.

std::vector<double> borders_of_groups (number_of_energy_groups);

std::array<std::vector<double>, N> Energies;
```

```
typedef std::tuple<double, double, double> coord;

typedef std::tuple<double, double, double, double> longDoubleTuple;

//There's a plane equation presented like Ax+By+Cz+D=0.
const std::vector<longDoubleTuple> planes = {std::make_tuple(0, 0, 1, -0.5)};

void data_file_creation (std::string DataType, std::vector<coord>& xx);

void data_file_creation (std::string DataType, std::vector<double>& xx,
std::vector<coord>& yy);

void data_file_creation (std::string DataType, std::vector<std::pair<int,
int>>& xx);

void data_file_creation (std::string DataType, std::vector<std::pair<int,
double>>& xx);

void data_file_creation (std::string DataType, std::vector<std::pair<double,
double>>& xx);

void default_distribution_plot (std::string& name, std::string& data,
std::string xlabel,
                                std::string ylabel, std::string title);

longDoubleTuple beam_direction (double sigma);

coord coordinates_of_the_interaction (longDoubleTuple& beams);

int energy_group(double& E);

std::array<std::vector<coord>, N> interactions (std::vector<coord>& points);

double cos_t (coord& A, coord& B, coord& C);

void plot(std::array<std::vector<coord>, N>& points);

std::vector<longDoubleTuple> database_read (std::string name);

double linear_interpolation (double& x_0, double& y_0, double& x_1, double&
y_1, double& x);

std::vector<std::tuple<double, double, double>> interpolated_database
(std::vector<longDoubleTuple>& default_database);

std::string exec(std::string str);

std::string PATH = exec("echo $PWD");

std::vector<std::tuple<double, double, double>> sigmas_air;
std::vector<std::tuple<double, double, double>> sigmas_Al;
```

```
std::tuple<double, double, double> interpolation_for_single_particle (int&
group, double& E,

std::vector<std::tuple<double, double, double>>& sigmas);
const double h = 0.5;

std::vector<coord> detectors = {std::make_tuple(0, 0, 0.25*h),
                                std::make_tuple(0, 0, 0.5*h),
                                std::make_tuple(0, 0, 1.5*h)};

const int detectors_number = detectors.size();

std::vector<std::vector<std::vector<double>>> eta (detectors_number);

std::vector<std::vector<std::pair<double, double>>> particle_eta
(detectors_number);

std::vector<std::vector<int>> groups_in_detector (detectors_number);

std::vector<std::vector<std::pair<int, int>>> detector_readings ();

void interpolation_plot (std::string matter, std::vector<double>& E,
                         std::vector<std::tuple<double, double, double>>&
sigmas);

void path_def (std::string& path);

void detector_statistics_plot (std::string data, std::string& title);

coord vector_creation (coord& A, coord& B);

void coordinates_from_tuple (double& x, double& y, double& z, coord& point);

std::vector<std::vector<std::pair<int, double>>> flow_through_detector
(std::array<std::vector<coord>, N>& interactions);

void detector_plot (std::string data, std::string title);

std::vector<coord> birth_of_photons ();

std::random_device rd;  //Will be used to obtain a seed for the random number
engine
std::mt19937 gen(rd()); //Standard mersenne_twister_engine seeded with rd()

int main() {

    std::cout << "Databases collecting...\t";

    path_def(PATH);
    borders_of_groups.at(0) = (E_min);
    double E = E_min;
```

```
    std::generate(borders_of_groups.begin()+1, borders_of_groups.end(), [&] {
return E += group_range; });
    std::vector<longDoubleTuple> air_data = std::move(database_read(PATH +
"air_sigmas_database"));
    sigmas_air = std::move(interpolated_database(air_data));
    data_file_creation("air", borders_of_groups, sigmas_air);
    std::vector<longDoubleTuple> Al_data = std::move(database_read(PATH +
"Al_sigmas_database"));
    sigmas_Al = std::move(interpolated_database(Al_data));
    data_file_creation("Al", borders_of_groups, sigmas_Al);

    std::cout << "Done!" << std::endl;

    std::cout << "Computing..." << std::endl;

    std::vector<coord> born_points = std::move(birth_of_photons());
    std::string name = "Distribution of " + std::to_string(N) + " points";
    data_file_creation(name, born_points);
    for (int i = 0; i < detectors_number; i++) //Did not allocate memory -
died.
        eta[i].resize(number_of_energy_groups);
    std::array<std::vector<coord>, N> interaction_points =
std::move(interactions(born_points));
    std::vector<std::vector<std::pair<int, int>>> detectors_data =
std::move(detector_readings());
    std::vector<std::string> names;
    std::vector<std::vector<std::pair<int, double>>> density =
flow_through_detector (interaction_points);
    for (int i = 0; i < detectors_number; i++) {
        names.emplace_back("detector_" + std::to_string(i));
        data_file_creation(names[i], detectors_data[i]);
        std::sort(particle_eta[i].begin(), particle_eta[i].end(), [] (auto&
left, auto& right)
        {return left.first < right.first;});
        //data_file_creation(names[i]+"_density", density[i]);
        data_file_creation(names[i] + "_density", particle_eta[i]);
    }

    std::cout << "Done!" << std::endl;

    std::cout << "Plotting...\t";

    default_distribution_plot(name, name, "x", "y", name);
    interpolation_plot("air", borders_of_groups, sigmas_air);
    interpolation_plot("Al", borders_of_groups, sigmas_air);
    data_file_creation("detectors", detectors);
    for (int i = 0; i < detectors_number; i++) {
        double x, y, z;
        coordinates_from_tuple(x,y, z, detectors[i]);
        std::string title = "Detector inside the ";
        title += (z <= h) ? "air" : "Al";
        detector_statistics_plot(names[i], title);
```

```
        detector_plot(names[i] + "_density", "Density flow through the " +
title);
    }
    plot(interaction_points);

    std::cout << "Done!" << std::endl;

    return 0;
}

std::vector<coord> birth_of_photons () {
    std::vector <coord> coords;
    double x, y, a, b, z = 0;
    std::uniform_real_distribution<> dis(0.0, 1.0);
    for (int i = 0; i < N; i++) {
        do {
            a = 1;
            b = a / 2.0;
            x = 2*dis(gen) - 1;
            y = 2*dis(gen) - 1;
        } while (std::pow(x, 2) / std::pow(a, 2) + std::pow(y, 2) /
std::pow(b, 2) > 1);
        coords.emplace_back(std::make_tuple(x, y, z));
    }
    return coords;
}

//Function creates a data-file with coordinates. It's useful for plotting.
void data_file_creation (std::string DataType, std::vector<coord>& xx) {
    //For reading created files via Matlab use command: M =
dlmread('/PATH/file'); xi = M(:,i);
    std::ofstream fout;
    fout.open(PATH + DataType);
    for (int i = 0; i < xx.size(); i++)
        fout << std::get<0>(xx[i]) << '\t' << std::get<1>(xx[i]) << '\t'
             << std::get<2>(xx[i]) << '\t' << std::endl;
    fout.close();
}

int norm (std::vector<std::pair<int, int>>& data) {
    int max_count = 0;
    for (int i = 0; i < data.size(); i++)
        if (data[i].second > max_count)
            max_count = data[i].second;
    return max_count;
}

void data_file_creation (std::string DataType, std::vector<std::pair<int,
int>>& data) {
    std::ofstream fout;
    DataType = PATH + DataType;
    fout.open(DataType);
    double n = (DataType.find("_2") != std::string::npos) ? N : norm(data);
```

```
    for (int i = 0; i < data.size(); i++)
        fout << std::get<0>(data[i]) << '\t' << std::get<1>(data[i]) / n <<
std::endl;
    fout.close();
}

void data_file_creation (std::string DataType, std::vector<std::pair<int,
double>>& data) {
    std::ofstream fout;
    DataType = PATH + DataType;
    fout.open(DataType);
    for (int i = 0; i < data.size(); i++)
        fout << (std::get<0>(data[i]) * group_range + borders_of_groups[0])
<< '\t' << std::get<1>(data[i]) << std::endl;
    fout.close();
}

void data_file_creation (std::string DataType, std::vector<std::pair<double,
double>>& data) {
    std::ofstream fout;
    DataType = PATH + DataType;
    fout.open(DataType);
    for (int i = 0; i < data.size(); i++)
        fout << std::get<0>(data[i]) / 1.0e6 << '\t' << std::get<1>(data[i])
<< std::endl;
    fout.close();
}

//Function plots points of borning. It shows the initial distribution.
void default_distribution_plot (std::string& name, std::string& data,
std::string xlabel, std::string ylabel, std::string title) {
    //if you have problems with ".svg", you have to change ".svg" to ".pdf"
in strings bellow.
    FILE *gp = popen("gnuplot  -persist", "w");
    if (!gp)
        throw std::runtime_error ("Error opening pipe to GNUplot.");
    std::vector<std::string> stuff = {"set term svg",
                                      "set out \'" + PATH + name + ".svg\'",
                                      "set xrange [-1:1]",
                                      "set yrange [-1:1]",
                                      "set xlabel \'" + xlabel + "\'",
                                      "set ylabel \'" + ylabel + "\'",
                                      "set grid xtics ytics",
                                      "set title \'" + title + "\'",
                                      "plot \'" + PATH + data + "\' using 1:2
lw 1 lt rgb 'orange' ti \'Nodes\'",
                                      "set key box top right",
                                      "set terminal pop",
                                      "set output",
                                      "replot", "q"};
    for (const auto& it : stuff)
        fprintf(gp, "%s\n", it.c_str());
    pclose(gp);
```

```
}

template<typename T, size_t... Is>
auto abs_components_impl(T const& t, std::index_sequence<Is...>) {
    return std::sqrt((std::pow(std::get<Is>(t), 2) + ...));
}

template <class Tuple>
double abs_components(const Tuple& t) {
    constexpr auto size = std::tuple_size<Tuple>{};
    return abs_components_impl(t, std::make_index_sequence<size>{});
}

//Function returns the beam direction and free run length for particle.
/* Note: always cos_gamma > 0, but it's n*/
longDoubleTuple beam_direction (double SIGMA) {
    std::uniform_real_distribution<> dis(0.0, 1.0);
    double L, mu, a, b, cos_psi, cos_gamma, d = 10;
    do {
        mu = 2 * dis(gen) - 1;
        do {
            a = 2 * dis(gen) - 1;
            b = 2 * dis(gen) - 1;
            d = std::pow(a, 2) + std::pow(b, 2);
            L = -log(dis(gen)) / SIGMA;
        } while (d > 1 || !std::isfinite(L));
        cos_psi = a / std::sqrt(d);
        cos_gamma = std::sqrt(1.0 - (std::pow(mu, 2) + std::pow(cos_psi,
2)));
    } while (std::pow(mu, 2) + std::pow(cos_psi, 2) > 1);
    return std::make_tuple(cos_gamma, mu, cos_psi, L);
}

//Function returns vector of beam direction.
coord coordinates_of_the_interaction (longDoubleTuple& beam) {
    double x = std::get<2>(beam) * std::get<3>(beam);
    double y = std::get<1>(beam) * std::get<3>(beam);
    double z = std::get<0>(beam) * std::get<3>(beam);
    return std::make_tuple(x, y, z);
}

void coordinates_from_tuple (double& x, double& y, double& z, coord& point) {
    x = std::get<0>(point);
    y = std::get<1>(point);
    z = std::get<2>(point);
}

//The functions bellow using for solving systems of linear equations via LU-
decomposition.
void LU_decomposition (std::vector<std::array<double, 3>>& default_matrix,
                       std::vector<std::array<double, 3>>& L,
                       std::vector<std::array<double, 3>>& U) {
    U = default_matrix;
```

```
    for (int i = 0; i < default_matrix.size(); i++) {
        for (int j = 0; j < default_matrix[i].size(); j++) {
            U.at(i).at(j) = 0;
            L.at(i).at(j) = 0;
        }
        L.at(i).at(i) = 1;
    }
    for (int i = 0; i < default_matrix.size(); i++) {
        for (int j = 0; j < default_matrix[i].size(); j++) {
            double sum = 0;
            for (int k = 0; k < i; k++)
                sum += L[i][k]*U[k][j];
            if (i <= j)
                U.at(i).at(j) = default_matrix[i][j] - sum;
            else
                L.at(i).at(j) = (default_matrix[i][j] - sum) / U[j][j];

        }
    }
}

std::vector<double> direct (std::vector<std::array<double, 3>>& L,
std::vector<double>& b) {
    std::vector<double> y;
    for (int i = 0; i < L.size(); i++) {
        double sum = 0;
        for (int j = 0; j < i; j++)
            sum += L[i][j] * y[j];
        y.emplace_back(b[i] - sum);
    }
    return y;
}

std::vector<double> reverse (std::vector<std::array<double, 3>>& U,
std::vector<double>& y) {
    std::vector<double> x = y;
    for (int i = y.size()-1; i >= 0; i--) {
        for (int j = i + 1; j < y.size(); j++)
            x.at(i) -= U[i][j] * x[j];
        x.at(i) /= U[i][i];
    }
    return x;
}

coord solve (std::vector<std::array<double, 3>> matrix, std::vector<double>&
free_numbers_column) {
    std::vector<std::array<double, 3>> L(3), U(3);
    LU_decomposition(matrix, L, U);
    std::vector<double> straight_run_results = std::move(direct(L,
free_numbers_column));
    std::vector<double> x = std::move(reverse(U, straight_run_results));
    return std::make_tuple(x[0], x[1], x[2]);
}
```

```
//Function returns roots of quadratic equation (a*t^2 + b^2*t + c == 0)
taking coefficients.
std::vector<double> quadratic_equation_solve (double& a, double& b, double&
c) {
    double D = std::pow(b, 2) - 4*a*c;
    double t_1, t_2;
    if (D >= 0 && !std::isnan(D)) {
        t_1 = (-b + std::sqrt(sqrt(D))) / 2.0 / a;
        t_2 = (-b - std::sqrt(sqrt(D))) / 2.0 / a;
    } else
        t_1 = t_2 = 0;
    return {t_1, t_2};
}

coord definition_of_intersection_points_cylinder (coord& initial_point,
longDoubleTuple& beam) {
    double x, x_init, y, y_init, z, z_init, cos_alpha, cos_beta, cos_gamma,
A, B, C, D;
    cos_alpha = std::get<2>(beam);
    cos_beta = std::get<1>(beam);
    cos_gamma = std::get<0>(beam);
    coordinates_from_tuple(x_init, y_init, z_init, initial_point);
    //a*t^2 + b*t + c == 0
    double a = std::pow(cos_alpha, 2) + std::pow(cos_beta, 2);
    double b = 2*(cos_alpha*x_init + cos_beta*y_init);
    double c = std::pow(x_init, 2) + std::pow(y_init, 2) - 1;

    std::vector<double> roots = std::move(quadratic_equation_solve(a, b, c));

    for (int i = 0; i < roots.size(); i++) {
        x = x_init + roots[i]*cos_alpha;
        y = y_init + roots[i]*cos_beta;
        z = z_init + roots[i]*cos_gamma;
        if (z > 0 && z <= 0.5) break;
    }
    return (z > 0 && z <= 0.5) ? std::make_tuple(x, y, z) : initial_point;
}

//First of all you need to analise is cos's and axes...
coord definition_of_intersection_points_cap (coord& initial_point,
longDoubleTuple& beam) {
    double x, x_init, y, y_init, z, z_init, cos_alpha, cos_beta, cos_gamma,
A, B, C, D;
    cos_alpha = std::get<2>(beam);
    cos_beta = std::get<1>(beam);
    cos_gamma = std::get<0>(beam);
    coordinates_from_tuple(x_init, y_init, z_init, initial_point);
    coord intersection_coordinate;
    int i = 0;
    do {
        A = std::get<0>(planes[i]);
        B = std::get<1>(planes[i]);
        C = std::get<2>(planes[i]);
```

```
        D = std::get<3>(planes[i]);
        std::vector<std::array<double, 3>> matrix = {{cos_beta, -cos_alpha,
0},
                                                     {0, cos_gamma,  -
cos_beta},
                                                     {A,        B,
C}};
        std::vector<double> right_part = {x_init*cos_beta - y_init*cos_alpha,
                                          y_init*cos_gamma - z_init*cos_beta,
                                          -D};
        intersection_coordinate = std::move(solve(matrix, right_part));
        coordinates_from_tuple(x, y, z, intersection_coordinate);
        i++;
    } while (i < planes.size() && !(std::pow(x, 2) + std::pow(y, 2) <= 1 && z
> 0 && z <= 0.5)
             || intersection_coordinate == initial_point);
    if (intersection_coordinate == initial_point || !(std::pow(x, 2) +
std::pow(y, 2) <= 1 && z > 0 && z <= 0.5))
        return initial_point;
    else
        return intersection_coordinate;
}

//Just a function for creating vector with two points.
coord vector_creation (coord& A, coord& B) {
    return std::make_tuple(std::get<0>(B) - std::get<0>(A),
                           std::get<1>(B) - std::get<1>(A),
                           std::get<2>(B) - std::get<2>(A));
}

template<typename T, size_t... Is>
auto scalar_prod_components_impl(T const& t, T const& t1,
std::index_sequence<Is...>, std::index_sequence<Is...>) {
    return ((std::get<Is>(t) * std::get<Is>(t1)) + ...);
}

template <class Tuple>
double scalar_prod_components(const Tuple& t, const Tuple& t1) {
    constexpr auto size = std::tuple_size<Tuple>{};
    return scalar_prod_components_impl(t, t1,
std::make_index_sequence<size>{}, std::make_index_sequence<size>{});
}

template<typename T, size_t... Is>
auto sum_components_impl(T const& t, std::index_sequence<Is...>) {
    return (std::get<Is>(t) + ...);
}

template <class Tuple>
double sum_components(const Tuple& t) {
    constexpr auto size = std::tuple_size<Tuple>{};
    return sum_components_impl(t, std::make_index_sequence<size>{});
}
```

```
coord vector_offset (coord& frame_of_reference, coord& vector) {
    return std::make_tuple(std::get<0>(frame_of_reference) +
std::get<0>(vector),
                           std::get<1>(frame_of_reference) +
std::get<1>(vector),
                           std::get<2>(frame_of_reference) +
std::get<2>(vector));
}

coord definition_of_intersection_points (coord& initial_point,
longDoubleTuple& beam) {
    coord intersection_coordinate;
    intersection_coordinate =
std::move(definition_of_intersection_points_cylinder(initial_point, beam));
    double error = 1.0e-15;
    if (std::abs(abs_components(intersection_coordinate) -
abs_components(initial_point)) < error)
        intersection_coordinate =
std::move(definition_of_intersection_points_cap(initial_point, beam));
    return intersection_coordinate;
}

/* Function returns coordinate of interaction for particles.
 * If it interaction outside the sarcophagus functions returns the point of
intersection with one of the planes,
 * otherwise it returns the interaction point in air. */
coord definition_of_interaction_points (coord& initial_point,
longDoubleTuple& direction) {
    double x_init, y_init, z_init;
    coordinates_from_tuple(x_init, y_init, z_init, initial_point);
    coord intersection_point =
definition_of_intersection_points(initial_point, direction);
    coord intersection_vector = vector_creation(initial_point,
intersection_point);
    coord free_run_direction = coordinates_of_the_interaction(direction);
    coord free_run = vector_offset(initial_point, free_run_direction);
    // Well, we have an error when try to compare two doubles.
    // So we have to enter the amount of acceptable error.
    double error = 1.0e-15;
    if (std::pow(x_init, 2) + std::pow(y_init, 2) <= 1 && z_init >= 0 &&
z_init < 1)
        if (abs_components(intersection_vector) < abs_components(free_run))
            return intersection_point;
        else
            return free_run;
    else
        if (std::abs(abs_components(intersection_vector)) < error)
            return free_run;
        else
            return intersection_point;
}
```

```
//Function returns the probability for types of interaction for environment
(which defines with argument).
std::vector<std::pair<double, std::string>> statistical_weight
(std::tuple<double, double, double>& sigma,
                                                        double& sum)
{
    std::vector<std::pair<double, std::string>> ans;
    double p_Compton = std::get<0>(sigma) / sum;
    double p_ph = std::get<1>(sigma) / sum;
    double p_pp = std::get<2>(sigma) / sum;
    ans = {std::make_pair(p_Compton, "Compton"), std::make_pair(p_ph, "ph"),
std::make_pair(p_pp, "pp")};
    return ans;
}

//Function return random interaction type.
std::string interaction_type (std::vector<std::pair<double, std::string>>& p)
{
    std::sort(p.begin(), p.end());
    std::uniform_real_distribution<> dis(0.0, 1.0);
    double gamma = dis(gen);
    return (gamma <= p[0].first) ? p[0].second : (gamma <= p[1].first) ?
p[1].second : p[2].second;
}

//Function returns cos(a, b), where a, b -- vectors;
double cos_t(coord& A, coord& B, coord& C) {
    coord a = std::move(vector_creation(A, B));
    coord b = std::move(vector_creation(B, C));
    return scalar_prod_components(a, b) / (abs_components(a) *
abs_components(b));
}

//Function returns linear interpolation of interaction cross section for
particles among neighboring borders.
double linear_interpolation (double& x_0, double& y_0, double& x_1, double&
y_1, double& x) {
    return y_0 + (y_1 - y_0)/(x_1 - x_0) * (x - x_0);
}

double density_estimation (double& W, double sigma_0, double sigma_1, int
group, int& detector_number,
                           double& distance, std::tuple<double, double,
double>& particle_sigma) {
    double tmp = (W * std::exp(-distance) / std::pow(distance, 2) *
                  std::get<0>(particle_sigma) / ((sigma_0 + sigma_1) / 2.0));
    eta[detector_number][group].emplace_back((distance != 0) ? tmp : 0);
    return tmp;
}

void flow_detection (double& sigma_sum, std::vector<std::pair<double,
std::string>>& p, std::string& type,
```

```
                    double& E, std::string environment, coord&
particle_coordinate, double& W) {
    int group = energy_group(E);
    std::vector<std::tuple<double, double, double>> sigma;
    if (environment == "air")
        sigma = sigmas_air;
    else
        sigma = sigmas_Al;
    std::tuple<double, double, double> particle_sigma =
(interpolation_for_single_particle(group, E, sigma));
    sigma_sum = sum_components(particle_sigma);
    p = statistical_weight(particle_sigma, sigma_sum);
    type = interaction_type(p);
    W *= p[0].first;
    if (type == "Compton" && W > 1.0e-11) {
        double x_det, y_det, z_det;
        for (int i = 0; i < detectors.size(); i++) {
            coordinates_from_tuple(x_det, y_det, z_det, detectors[i]);
            coord tau = vector_creation(particle_coordinate, detectors[i]);
            double distance = abs_components(tau);
            if (z_det <= h) { // We have to include particles only inside the
box.
                if (environment == "air") {
                    groups_in_detector.at(i).emplace_back(group);
                    double contribution = (density_estimation(W,
std::get<0>(sigmas_air[group]), std::get<0>(sigmas_air[group+1]),
                                                              group, i,
distance, particle_sigma));
                    particle_eta[i].emplace_back(std::make_pair(E,
contribution));
                }
            } else { // Only outside particles.
                if (environment == "Al") {
                    if (E == E_0) continue;
                    groups_in_detector.at(i).emplace_back(group);
                    double contribution = density_estimation(W,
std::get<0>(sigmas_Al[group]), std::get<0>(sigmas_Al[group+1]),
                                                             group, i,
distance, particle_sigma);
                    particle_eta[i].emplace_back(std::make_pair(E,
contribution));
                }
            }
        }
    } else E = 0;
}

//The function returns energy steps for every particle.
std::array<std::vector<coord>, N> interactions (std::vector<coord>& points) {
    std::vector<std::pair<double, std::string>> p_air, p_Al;
    std::vector<std::tuple<double, double, double>> sigmas;
    std::vector<double> Energy;
    std::array<std::vector<coord>, N> interaction_points;
```

```
    double sigma_2_air_sum = sum_components(sigmas_air[sigmas_air.size()-1]);
    double x, y, z, alpha, E, sigma_sum, cos_ab;
    for (int i = 0; i < points.size(); i++) {
        interaction_points.at(i).emplace_back(points[i]);
        alpha = E_0 / E_e;
        std::string type;
        double W = 1.0;
        bool flag = false;
        coord A, C, B;
        longDoubleTuple direction;
        do {
            if (flag == 1) {
                E = alpha * E_e;
                Energy.emplace_back(E);
                interaction_points.at(i).emplace_back(B);
                if (std::abs(x) <= 1 && std::abs(y) <= 1 && z <= 1)
                    flow_detection(sigma_sum, p_air, type, E, "air", B, W);
                else
                    flow_detection(sigma_sum, p_Al, type, E, "Al", B, W);
                direction = std::move(beam_direction(sigma_sum));
                C = definition_of_interaction_points(B, direction);
                coordinates_from_tuple(x, y, z, C);
                cos_ab = cos_t(A, B, C);
                A = B;
                B = C;
                alpha /= 1 + (1 - cos_ab)*alpha;
            } else {
                A = points[i];
                direction = std::move(beam_direction(sigma_2_air_sum));
                B = definition_of_interaction_points(A, direction);
                for (int j = 0; j < detectors_number; j++) {
                    double x_det, y_det, z_det;
                    coordinates_from_tuple(x_det, y_det, z_det,
detectors[j]);
                    coord tau = vector_creation(A, detectors[j]);
                    double distance = abs_components(tau);
                    if (z_det < 0.5) { // Just for born including.
                        double contribution = (density_estimation(W,
std::get<0>(sigmas_air[sigmas_air.size() - 2]),

std::get<0>(sigmas_air[sigmas_air.size() - 1]),

number_of_energy_groups - 1, j, distance,

sigmas_air[sigmas_air.size() - 1]));
                        particle_eta[j].emplace_back(std::make_pair(E_0,
contribution));
                    }
                }
                flag = true;
            }
            if (std::isnan(alpha)) break;
            coordinates_from_tuple(x, y, z, B);
```

```
        } while (type.empty() == 1 || type == "Compton" && E > E_min);
        Energies.at(i) = Energy;
    }
    return interaction_points;
}

//The function plots the trajectories of particles. So it not fast, so you
can comment it in main().
void plot(std::array<std::vector<coord>, N>& points) {
    FILE *gp = popen("gnuplot  -persist", "w");
    if (!gp)
        throw std::runtime_error("Error opening pipe to GNUplot.");
    std::vector<std::string> stuff = {"set term pdf",
                                      "set output \'" + PATH +
"Hedgehog.pdf\'",
                                      //"set term pop",
                                      "set multiplot",
                                      "set grid xtics ytics ztics",
                                      "set xrange [-1:1]",
                                      "set yrange [-1:1]",
                                      "set zrange [0:1]",
                                      "set key off",
                                      "set ticslevel 0",
                                      "set border 4095",
                                      "splot \'" + PATH + "detectors\' u
1:2:3 lw 3 lt rgb 'black'",
                                      "set parametric",
                                      "set urange [0:2*pi]",
                                      "set vrange [0:0.5]",
                                      "splot cos(u),sin(u),v lw 1 lt rgb
\'black\' title \'cylinder\'",
                                      "unset parametric",
                                      "splot '-' u 1:2:3 w lines"};
    for (const auto& it : stuff)
        fprintf(gp, "%s\n", it.c_str());
    double x, y, z;
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < points[i].size(); j++) {
            coordinates_from_tuple(x, y, z, points[i][j]);
            fprintf(gp, "%f\t%f\t%f\n", x, y, z);
        }
        fprintf(gp, "%c\n%s\n", 'e', "splot '-' u 1:2:3 w lines");
    }
    fprintf(gp, "%c\n", 'q');
    pclose(gp);
}

//Function return the number of energy group for particle.
int energy_group(double& E) {
    for (int i = borders_of_groups.size() - 1; i > 0; i--)
        if (E >= borders_of_groups[i - 1] && E <= borders_of_groups[i])
            return i - 1;
}
```

```
namespace std {
    istream& operator >> (istream& in, longDoubleTuple& data) {
        double first, second, third, fourth;
        in >> first >> second >> third >> fourth;
        data = {first, second, third, fourth};
        return in;
    }

    ostream& operator << (ostream& out, const longDoubleTuple& data) {
        auto [first, second, third, fourth] = data;
        out << first << ' ' << second << ' ' << third << ' ' << fourth << '
';
        return out;
    }
}

/* Function returns data in std::vector of std::tuple from text file include
the interaction cross section for particle
 * with determined energy in environment. File looks like matrix 3xN. */
std::vector<longDoubleTuple> database_read (std::string name) {
    std::ifstream inFile(name);
    std::vector<longDoubleTuple> tuples_vector;
    copy(std::istream_iterator<longDoubleTuple> {inFile},
         std::istream_iterator<longDoubleTuple> {},
         back_inserter(tuples_vector));
    //copy(tuples_vector.begin(), tuples_vector.end(),
std::ostream_iterator<longDoubleTuple>(std::cout, "\n"));
    return tuples_vector;
}

//Function returns database received via linear interpolation. It will be
useful for presentation of results.
std::vector<std::tuple<double, double, double>> interpolated_database
(std::vector<longDoubleTuple>& default_database) {
    std::vector<std::tuple<double, double, double>> new_data;
    int i, j, k;
    for (j = 0; j < default_database.size(); j++)
        if (std::get<0>(default_database[j]) == borders_of_groups[0])
            k = j;
    i = 0;
    j = k;
    double lb_E, lb_Compton, lb_ph, lb_pp, rb_E, rb_Compton, rb_ph, rb_pp,
Compton, ph, pp;
    while (j < default_database.size()) {
        if (std::get<0>(default_database[j + 1]) -
std::get<0>(default_database[j]) == group_range) {

new_data.emplace_back(std::make_tuple(std::get<1>(default_database[j]),

std::get<2>(default_database[j]),

std::get<3>(default_database[j])));
            j++;
```

```
            i = (k == 0) ? j : j - k;
            continue;
        }
        lb_E = std::get<0>(default_database[j]);
        lb_Compton = std::get<1>(default_database[j]);
        lb_ph = std::get<2>(default_database[j]);
        lb_pp = std::get<3>(default_database[j]);
        rb_E = std::get<0>(default_database[j+1]);
        rb_Compton = std::get<1>(default_database[j+1]);
        rb_ph = std::get<2>(default_database[j+1]);
        rb_pp = std::get<3>(default_database[j+1]);
        do {
            Compton = linear_interpolation(lb_E, lb_Compton, rb_E,
rb_Compton, borders_of_groups[i]);
            ph = linear_interpolation(lb_E, lb_ph, rb_E, rb_ph,
borders_of_groups[i]);
            pp = linear_interpolation(lb_E, lb_pp, rb_E, rb_pp,
borders_of_groups[i]);
            new_data.emplace_back(std::make_tuple(Compton, ph, pp));
            i++;
        } while (borders_of_groups[i] < rb_E && i <
borders_of_groups.size());
        j++;
    }
    return new_data;
}

//The group determines by left border. The function returns interpolated
interaction cross sections for single particle.
std::tuple<double, double, double> interpolation_for_single_particle (int&
group, double& E,

std::vector<std::tuple<double, double, double>>& sigmas) {
    double lb_E = borders_of_groups[group];
    double lb_Compton = std::get<0>(sigmas[group]);
    double lb_ph = std::get<1>(sigmas[group]);
    double lb_pp = std::get<2>(sigmas[group]);
    double rb_E = borders_of_groups[group+1];
    double rb_Compton = std::get<0>(sigmas[group+1]);
    double rb_ph = std::get<1>(sigmas[group+1]);
    double rb_pp = std::get<2>(sigmas[group+1]);
    double Compton = Compton = linear_interpolation(lb_E, lb_Compton, rb_E,
rb_Compton, E);
    double ph = linear_interpolation(lb_E, lb_ph, rb_E, rb_ph, E);
    double pp = linear_interpolation(lb_E, lb_pp, rb_E, rb_pp, E);
    return std::make_tuple(Compton, ph, pp);
}

void interpolation_plot(std::string matter, std::vector<double>& E,
                        std::vector<std::tuple<double, double, double>>&
sigmas) {
    std::string data_location = PATH + matter;
    FILE *gp = popen("gnuplot  -persist", "w");
```

```
    if (!gp)
        throw std::runtime_error("Error opening pipe to GNUplot.");
    std::vector<std::string> stuff = {"set term svg",
                                      "set out \'" + PATH + "Photon Cross
Sections for " + matter + ".svg\'",
                                      "set grid xtics ytics",
                                      "set title \'Photon Cross Sections for
" + matter + "\'",
                                      "plot \'" + data_location + "\' u 1:2 w
lines ti \'Compton Scattering\',\'"
                                      + data_location + "\' u 1:2 lw 1 lt rgb
'black' ti \'Compton Scattering nodes\',\'"
                                      + data_location + "\' u 1:3 w lines ti
\'Photoelectric\',\'"
                                      + data_location + "\' u 1:3 lw 1 lt rgb
'black' ti \'Photoelectric nodes\',\'"
                                      + data_location + "\' u 1:4 w lines ti
\'Pair Production\',\'"
                                      + data_location + "\' u 1:4 lw 1 lt rgb
'black' ti \'Pair Production nodes\'",
                                      "set xlabel \'Energy, eV\'",
                                      "set ylabel \'Cross sections,
cm^2/g\'",
                                      "set terminal wxt",
                                      "set output",
                                      "replot", "q"};
    for (const auto& it : stuff)
        fprintf(gp, "%s\n", it.c_str());
    pclose(gp);
}

void data_file_creation (std::string DataType, std::vector<double>& xx,
std::vector<coord>& yy) {
    //For reading created files via Matlab use command: M =
dlmread('/PATH/file'); xi = M(:,i);
    std::ofstream fout;
    DataType = PATH + DataType;
    fout.open(DataType);
    for(int i = 0; i < xx.size(); i++)
        fout << xx[i] << '\t' << std::get<0>(yy[i]) << '\t' <<
std::get<1>(yy[i])
             << '\t' << std::get<2>(yy[i]) << '\t' << std::endl;
    fout.close();
}

//Just a function for returning the terminal output.
std::string exec (std::string str) {
    const char* cmd = str.c_str();
    std::array<char, 128> buffer;
    std::string result;
    std::unique_ptr<FILE, decltype(&pclose)> pipe(popen(cmd, "r"), pclose);
    if (!pipe) throw std::runtime_error("popen() failed!");
    while (fgets(buffer.data(), buffer.size(), pipe.get()) != nullptr)
```

```
        result += buffer.data();
    result = result.substr(0, result.length()-1);
    return result;
}

/* When you use cmake, files located in "cmake-build-debug",
 * when you use smth like "g++ -o %project_name main.cpp",
 * files located in root-directory of project.
 * So this function allows store files in root-directory of project
 * in both cases described. */
void path_def (std::string& path) {
    if (path.find("cmake-build-debug") != std::string::npos)
        path += "/../";
    else
        path += '/';
}

//Function returns number of particles of every group for every detector.
std::vector<std::vector<std::pair<int, int>>> detector_readings () {
    std::vector<std::vector<std::pair<int, int>>> ans (detectors_number);
    for (int i = 0; i < detectors_number; i++) {
        std::vector<std::pair<int, int>> data (number_of_energy_groups);
        for (int j = 0; j < number_of_energy_groups; j++) {
            data.at(j).first = j;
            data.at(j).second = 0;
            for (int k = 0; k < groups_in_detector[i].size(); k++)
                if (groups_in_detector[i][k] == j)
                    data.at(j).second++;
        }
        ans.at(i) = data;
    }
    return ans;
}

void detector_statistics_plot (std::string data, std::string& title) {
    data = PATH + data;
    FILE *gp = popen("gnuplot  -persist", "w");
    if (!gp)
        throw std::runtime_error("Error opening pipe to GNUplot.");
    std::vector<std::string> stuff = {"set term eps",
                                      "set output \'" + data + ".eps\'",
                                      "set key off",
                                      "set grid xtics ytics",
                                      "set xlabel \'Number of group\'",
                                      "set ylabel \'Contributed particles
count\'",
                                      "set title \'" + title + "\'",
            //"set yrange [0:" + std::to_string(N) + "]",
                                      "set xrange [0:" +
std::to_string(number_of_energy_groups) + "]",
                                      "set boxwidth 0.5",
                                      "set style fill solid",
```

```
                                        "plot \'" + data + "\' using 1:2 with
boxes",
                                        "set terminal pop",
                                        "set output",
                                        "replot", "q"};
    for (const auto& it : stuff)
        fprintf(gp, "%s\n", it.c_str());
    pclose(gp);
}

//We have Energies for every particle and coordinates of intersection there.
std::vector<std::vector<std::pair<int, double>>> flow_through_detector
(std::array<std::vector<coord>, N>& interactions) {
    std::vector<std::vector<std::pair<int, double>>> ans (detectors_number);
    std::vector<std::vector<double>> eta_sum (detectors_number); //Consists
sum for every group;
    std::vector<int> number_of_contributed_particles;
    for (int i = 0; i < eta.size(); i++) {
        int contributed_particles = 0;
        for (int j = 0; j < eta[i].size(); j++) {
            contributed_particles += eta[i][j].size();
            double sum_of_elems = 0;
            std::for_each(eta.at(i).at(j).begin(), eta.at(i).at(j).end(), [&]
(double n) { sum_of_elems += (std::isnan(n)) ? 0 : n; });
            eta_sum.at(i).emplace_back(sum_of_elems);
        }
        number_of_contributed_particles.emplace_back(contributed_particles);
    }
    for (int i = 0; i < detectors_number; i ++) {
        double sum = 0;
        for (int j = 0; j < eta_sum[i].size(); j++) {
            sum += (std::isnan(eta_sum[i][j])) ? 0 : eta_sum[i][j];
            ans.at(i).emplace_back(std::make_pair(j, eta_sum[i][j]));
        }
        double x, y, z;
        coordinates_from_tuple(x, y, z, detectors[i]);
        std::cout << "Average flow trough (" << x << ", " << y << ", " << z
<< "):\t"
                  << sum / number_of_contributed_particles[i] << std::endl;
    }
    return ans;
}

void detector_plot (std::string data, std::string title) {
    data = PATH + data;
    FILE *gp = popen("gnuplot  -persist", "w");
    if (!gp)
        throw std::runtime_error("Error opening pipe to GNUplot.");
    std::vector<std::string> stuff = {"set term eps",
                                        "set output \'" + data + ".eps\'",
                                        "set key off",
                                        "set grid xtics ytics",
                                        "set xlabel \'E, MeV\'",
```

```
                                        "set ylabel \'Contribution to density,
1/m^2\'",
                                        "set title \'" + title + "\'",
                                        "plot \'" + data + "\' using 1:2 with
lines",
                                        "set terminal pop",
                                        "set output",
                                        "replot", "q"};
    for (const auto& it : stuff)
        fprintf(gp, "%s\n", it.c_str());
    pclose(gp);
}
```