```cpp
// Eikonal density and screening class
//
//
// (c) 2017-2019 Mikael Mieskolainen
// Licensed under the MIT License <http://opensource.org/licenses/MIT>.

// C++
#include <fstream>
#include <future>
#include <iomanip>
#include <iostream>
#include <vector>
#include <cmath>

// Own
#include "Graniitti/MAux.h"
#include "Graniitti/MEikonal.h"
#include "Graniitti/MForm.h"
#include "Graniitti/MMath.h"
#include "Graniitti/MTimer.h"
#include "Graniitti/MPDG.h"

// Libraries
#include "json.hpp"
#include "rang.hpp"


using gra::aux::indices;
using gra::math::msqrt;
using gra::math::pow2;
using gra::math::zi;


namespace gra {

        // Eikonal screening numerical integration parameters
        // N.B. Compile will make significant optimizations if boundaries
        // are const variables.
        namespace MEikonalNumerics {

                constexpr double MinKT2 = 1E-6;
                constexpr double MaxKT2 = 25.0;
                unsigned int NumberKT2 = 0;
                bool    logKT2 = false;

                constexpr double MinBT = 1E-6;
                constexpr double MaxBT = 10.0 / PDG::GeV2fm;
                unsigned int NumberBT = 0;
                bool    logBT = false;

                constexpr     double FBIntegralMinKT = 1E-9;
                constexpr     double FBIntegralMaxKT = 30.0;
                constexpr unsigned int   FBIntegralN = 10000;

                constexpr double MinLoopKT = 1E-4;
                double       MaxLoopKT = 1.75;

                std::string GetHashString() {
                        std::string str = std::to_string(MEikonalNumerics::MinKT2) +
                                          std::to_string(MEikonalNumerics::MaxKT2) +
                                          std::to_string(MEikonalNumerics::NumberKT2) +
                                          std::to_string(MEikonalNumerics::logKT2) +
                                          std::to_string(MEikonalNumerics::MinBT) +
                                          std::to_string(MEikonalNumerics::MaxBT) +
                                          std::to_string(MEikonalNumerics::NumberBT) +
                                          std::to_string(MEikonalNumerics::logBT) +
                                          std::to_string(MEikonalNumerics::FBIntegralMinKT) +
                                          std::to_string(MEikonalNumerics::FBIntegralMaxKT) +
                                          std::to_string(MEikonalNumerics::FBIntegralN)
                                          ;
                        return str;
                }

        void ReadParameters() {

                // Read and parse
                using json = nlohmann::json;

                const std::string inputfile = gra::aux::GetBasePath(2) + "/modeldata/" + "NUMERICS.json";
                const std::string data      = gra::aux::GetInputData(inputfile);
                json j;

                try {
                        j = json::parse(data);

                        // JSON block identifier
                        const std::string XID = "NUMERICS_EIKONAL";

                        //MaxKT2  = j[XID]["MaxKT2"];
                        NumberKT2 = j[XID]["NumberKT2"];
                        logKT2    = j[XID]["logKT2"];

                        //MaxBT   = j[XID]["MaxBT"]; MaxBT /= gra::math::GeV2fm; // Input as fermi, program uses GeV^{-1}
                        NumberBT  = j[XID]["NumberBT"];
                        logBT     = j[XID]["logBT"];

                        //FBIntegralMaxKT = j[XID]["FBIntegralMaxKT"];
                        //FBIntegralN     = j[XID]["FBIntegralN"];

                        //MaxLoopKT = j[XID]["MaxLoopKT"];

                } catch (...) {
                        std::string str =
                            "MEikonalNumerics::ReadParameters: Error parsing " + inputfile + " (Check for extra/missing commas)";
                        throw std::invalid_argument(str);
                }
        }

        // -----------------------------------------

        // Screening loop (minimum values)
        unsigned int NumberLoopKT  = 15;  // Number of kt steps
        unsigned int NumberLoopPHI = 12;  // Number of phi steps

        // User setup (ND can be negative, to get below the default)
        void SetLoopDiscretization(int ND) {
                NumberLoopKT  = std::max(3, 3 * ND + (int)NumberLoopKT);
                NumberLoopPHI = std::max(3, 3 * ND + (int)NumberLoopPHI);
        }

} // Namespace MEikonal ends


MEikonal::MEikonal() {
}


MEikonal::~MEikonal() {
}


// Return total, elastic, inelastic cross sections
void MEikonal::GetTotXS(double& tot, double& el, double& in) const {
        tot = sigma_tot;
        el  = sigma_el;
        in  = sigma_inel;
}


// Construct density and amplitude
void MEikonal::S3Constructor(double s_in, const std::vector<gra::MParticle>& initialstate_in, bool onlyeikonal) {

        // This first
        MEikonalNumerics::ReadParameters();

        // Mandelstam s and initial state
        s = s_in;
        INITIALSTATE = initialstate_in;

        // ---------------------------------------------------------------------------
        // Calculate hash based on all free variables -> if something changed,
        // calculate new densities

        // Proton density
        {
                std::cout << "Initializing <eikonal density> array:" << std::endl;
                MBT.sqrts = msqrt(s); // FIRST THIS
                MBT.Set("bt", MEikonalNumerics::MinBT, MEikonalNumerics::MaxBT, MEikonalNumerics::NumberBT, MEikonalNumerics::logBT);
                MBT.InitArray(); // Initialize (call last!)

                const std::string hstr = std::to_string(s) + std::to_string(INITIALSTATE[0].pdg) + "_" + std::to_string(INITIALSTATE[1].pdg) +
                                         PARAM_SOFT::GetHashString() + MEikonalNumerics::GetHashString();
                const unsigned long hash = gra::aux::djb2hash(hstr);
                const std::string filename = gra::aux::GetBasePath(2) +
                                                                        "/eikonal/" + "MBT_" + std::to_string(INITIALSTATE[0].pdg) +
                                                                         "_" + std::to_string(INITIALSTATE[1].pdg) +
                                             "_" + gra::aux::ToString(msqrt(s),0) + "_" + std::to_string(hash);

                // Try to read pre-calculated
                bool ok = MBT.ReadArray(filename);

                while (!ok) { // Problem, re-calculate
                        // Pointer to member function: ReturnType (ClassType::*)(ParameterTypes...)
                        std::complex<double> (MEikonal::*f)(double) const = &MEikonal::S3Density;
                        S3CalculateArray(MBT,f);
                        MBT.WriteArray(filename, true);
                        ok = MBT.ReadArray(filename);
                }
        }

        // Calculate cross sections (is fast)
        S3CalcXS();

        // Init cut Pomerons (is fast)
        S3InitCutPomerons();

        // Amplitude
        if (onlyeikonal == false) {
                std::cout << "Initializing <eikonal amplitude> array:" << std::endl;

                MSA.sqrts = msqrt(s); // FIRST THIS
                MSA.Set("kt2", MEikonalNumerics::MinKT2, MEikonalNumerics::MaxKT2, MEikonalNumerics::NumberKT2, MEikonalNumerics::logKT2);
                MSA.InitArray(); // Initialize (call last!)

                const std::string hstr = std::to_string(s)
                                                                 + std::to_string(INITIALSTATE[0].pdg)
                                                                 + std::to_string(INITIALSTATE[1].pdg)
                                                                 + PARAM_SOFT::GetHashString()
                                                                 + MEikonalNumerics::GetHashString();

                const unsigned long hash   = gra::aux::djb2hash(hstr);
                const std::string filename = gra::aux::GetBasePath(2) + "/eikonal/" + "MSA_" +
                                                                         std::to_string(INITIALSTATE[0].pdg)  + "_" + std::to_string(INITIALSTATE[1].pdg) +
                                             "_" + gra::aux::ToString(msqrt(s),0) + "_" + std::to_string(hash);

                // Try to read pre-calculated
                bool ok = MSA.ReadArray(filename);

                while (!ok) { // Problem, re-calculate
                        // Pointer to member function: ReturnType (ClassType::*)(ParameterTypes...)
                        std::complex<double> (MEikonal::*f)(double) const = &MEikonal::S3Screening;
                        S3CalculateArray(MSA, f);
                        MSA.WriteArray(filename, true);
                        ok = MSA.ReadArray(filename);
                }
        }
        // Tag it done
        S3INIT = true;
}


// Proton bt-density by Fourier-Bessel transform of the t-density
// see <http://mathworld.wolfram.com/HankelTransform.html>
//
//
// For eikonalization see:
// [REFERENCE: Desgrolard, Giffon, Martynov, Predazzi, https://arxiv.org/abs/hep-ph/9907451v2]
// [REFERENCE: Desgrolard, Giffon, Martynov, Predazzi, https://arxiv.org/abs/hep-ph/0001149]
//
//
// For some discussion about Odderon versus Pomeron, see:
// [REFERENCE: Ewerz, Maniatis, Nachtmann, https://arxiv.org/abs/1309.3478]
std::complex<double> MEikonal::S3Density(double bt) const{

        // Discretization of kt
        const double kt_STEP =
            (MEikonalNumerics::FBIntegralMaxKT - MEikonalNumerics::FBIntegralMinKT) /
            MEikonalNumerics::FBIntegralN;

        // N + 1!
        std::vector<std::complex<double>> f(MEikonalNumerics::FBIntegralN + 1, 0.0);

        // Initial state configuration [NOT IMPLEMENTED = SAME FOR pp and ppbar]
        // pp
        if (INITIALSTATE[0].pdg == PDG::PDG_p && INITIALSTATE[1].pdg == PDG::PDG_p) {
                // TBD
        // ppbar
        } else if ((INITIALSTATE[0].pdg ==  PDG::PDG_p && INITIALSTATE[1].pdg == -PDG::PDG_p) ||
                           (INITIALSTATE[0].pdg == -PDG::PDG_p && INITIALSTATE[1].pdg ==  PDG::PDG_p)) {
                // TBD
        }

        // Loop over
        for (const auto& i : indices(f)) {

                const double kt = MEikonalNumerics::FBIntegralMinKT + i * kt_STEP;

                // negative, with Mandelstam t ~= -kt^2
                const double t = -gra::math::pow2(kt);

                // Proton form factors (could be here extended to multichannel)
                const double F_i = gra::form::S3F(t);
                const double F_k = gra::form::S3F(t);

                // Pomeron trajectory alpha(t)
                const double alpha_P = gra::form::S3PomAlpha(t);

                // Pomeron exchange amplitude:
                // [Regge signature x Proton Form Factor x coupling x Proton
                //  Form Factor x coupling x Propagator ]
                const double s0 = 1.0; // Typical (normalization) scale GeV^{-2}

                // const std::complex<double> eta_O =
                // std::exp(gra::math::zi*PARAM_SOFT::PHASE_O);

                std::complex<double> A =
                    gra::math::pow2(PARAM_SOFT::gN_P) * F_i * F_k *
                    gra::form::ReggeEta(alpha_P, 1) *
                    std::pow(s / s0, alpha_P - 1.0);// Pomeron (C-even)

                // Value
                f[i] = A * gra::math::BESSJ0(bt * kt) * kt;
                //f[i] = A * std::cyl_bessel_j(0, bt * kt) * kt; // c++17
        }
        // Fourier-Bessel transformation denominator
        const double TD = 2.0*gra::math::PI;

        return gra::math::CSIntegral(f, kt_STEP) / TD;
}


// Calculate elastic screening amplitude by "Eikonalization",
// in kt^2 obtained via bt-space Fourier-Bessel integral
//
//
// Amplitude in bt-space:  A_el(b_t) = i(1 - exp(i*XI(b_t)/2))
//
std::complex<double> MEikonal::S3Screening(double kt2) const {

        // Local discretization
        const double STEP = (MEikonalNumerics::MaxBT - MEikonalNumerics::MinBT) /
                                                 MEikonalNumerics::FBIntegralN;

        const double kt = gra::math::msqrt(kt2);
        std::vector<std::complex<double>> f(MEikonalNumerics::FBIntegralN + 1, 0.0);

        // Numerical integral loop over impact parameter (b_t) space
        for (const auto& i : indices(f)) {

                const double bt = MEikonalNumerics::MinBT + i * STEP;
                const std::complex<double> XI = MBT.Interpolate1D(bt);

                // I. STANDARD EIKONAL APPROXIMATION
                const std::complex<double> A =
                    gra::math::zi * (1.0 - std::exp(gra::math::zi * XI / 2.0));

                f[i] = A * gra::math::BESSJ0(bt * kt) * bt;
                //f[i] = A * std::cyl_bessel_j(0, bt * kt) * bt; // c++17
        }
        const double C = 2.0 * gra::math::PI; // phi-integral

        // Numerical integration
        return (2.0 * s) * C * gra::math::CSIntegral(f, STEP);
}


// Calculate screened total, elastic and inelastic cross sections
// in the eikonal model
//
//
// \int d^2b f(b) = \int_0^\inf \int_0^{2\pi} r dr d\theta f(r,\theta)
//                = 2\pi \int_0^\inf f(r) r dr
//
void MEikonal::S3CalcXS() {
        std::cout << "MEikonal::S3CalcXS:" << std::endl << std::endl;

        // Local discretization
        const unsigned int N = 2 * 3000; // even number
        const double STEP = (MEikonalNumerics::MaxBT - MEikonalNumerics::MinBT) / N;

        // Two channel eikonal eigenvalue solutions obtained via symbolic
        // calculation see e.g.
        //
        // [REFERENCE: Khoze, Martin, Ryskin, https://arxiv.org/abs/hep-ph/0007359v2]
        // [REFERENCE: Roehr, http://inspirehep.net/record/1351489/files/Thesis-2014-Roehr.pdf]
        //
        // Unitary transformation matrix
        const MMatrix<double> cc = {{ 1, 1,  1,  1},  // pp
                                    {-1, 1,  1, -1},  // pN*
                                    {-1, 1, -1,  1},  // N*p
                                    { 1, 1, -1, -1}}; // N*N*
        // Eigenvalues
        const std::vector<double> lambda = {
            gra::math::pow2(1.0 - PARAM_SOFT::gamma),
            gra::math::pow2(1.0 + PARAM_SOFT::gamma),
            1.0 - gra::math::pow2(PARAM_SOFT::gamma),
            1.0 - gra::math::pow2(PARAM_SOFT::gamma)};
        // ----------------------------------------------------------------------------

        // N+1
        std::vector<std::complex<double>> f_tot(N + 1, 0.0);
        std::vector<std::complex<double>>  f_el(N + 1, 0.0);
        std::vector<std::complex<double>>  f_in(N + 1, 0.0);

        // Two-channel eikonal, N+1!
        std::vector<std::vector<std::complex<double>>> f_2(
            4, std::vector<std::complex<double>>(N + 1, 0.0));

        // Numerical integral loop over impact parameter (b_t) space
        for (const auto& i : indices(f_tot)) {

                const double bt = MEikonalNumerics::MinBT + i * STEP;

                // Calculate density
                const std::complex<double> XI = MBT.Interpolate1D(bt);

                // ----------------------------------------------------------------------
                // Single-Channel eikonal

                // Elastic amplitude A_el(s,b)
                const std::complex<double> A_el = gra::math::zi * (1.0 - std::exp(gra::math::zi * XI / 2.0)
);

                // TOTAL: Im A_el(s,b)
                f_tot[i] = std::imag(A_el);

                // ELASTIC: |A_el(s,b)|^2
                f_el[i] = gra::math::abs2(A_el);

                // INELASTIC: 2Im A_el(s,b) - |A_el(s,b)|^2
                f_in[i] = 2.0 * std::imag(A_el) - gra::math::abs2(A_el);
                // ----------------------------------------------------------------------

                // ----------------------------------------------------------------------
                // Two-channel Eikonal solutions for pp->p(*)p(*)
                const std::vector<std::complex<double>> sol = {
                    1.0 - std::exp(gra::math::zi * lambda[0] * XI / 2.0),
                    1.0 - std::exp(gra::math::zi * lambda[1] * XI / 2.0),
                    1.0 - std::exp(gra::math::zi * lambda[2] * XI / 2.0),
                    1.0 - std::exp(gra::math::zi * lambda[3] * XI / 2.0)};

                for (std::size_t k = 0; k < 4; ++k) {

                        // Amplitude squared
                        f_2[k][i] = gra::math::abs2(
                            (sol[0] * cc[k][0] + sol[1] * cc[k][1] +
                             sol[2] * cc[k][2] + sol[3] * cc[k][3]) / 4.0);
                        f_2[k][i] *= bt; // Jacobian
                }

                // Jacobian
                f_tot[i] *= bt;
                f_el[i] *= bt;
                f_in[i] *= bt;
        }

        // 2D-Integral factor
        const double IC = 2.0 * gra::math::PI;

        // Composite Simpson's rule, real is taken for C++ reasons in order to
        // be able to substitute into double
        sigma_tot  = 2.0 * IC * std::real(gra::math::CSIntegral(f_tot, STEP)) * PDG::GeV2barn;
        sigma_el   = IC  * std::real(gra::math::CSIntegral(f_el, STEP)) * PDG::GeV2barn;
        sigma_inel = IC  * std::real(gra::math::CSIntegral(f_in, STEP)) * PDG::GeV2barn;
        // sigma_inel = sigma_tot - sigma_el; // cross check

        // Comments for the "multichannel" formalism [next version]

        // Total cross section:    2*\int d^2b sum_ik |a_i|^2|a_k|^2 (1 -
        // exp^(-Omega(s,b)/2))
        std::cout << "Single Channel Eikonal:" << std::endl;
        printf(" Total xs:      %0.3f mb \n", sigma_tot*1E3);

        // Elastic cross section:  \int d^2b (sum_ik |a_i|^2|a_k|^2 (1 -
        // exp^(-Omega(s,b)/2))^2
        printf(" Elastic xs:    %0.3f mb \n",   sigma_el*1E3);

        // Inelastic cross section:  \int d^2b sum_ik |a_i|^2|a_k|^2 (1 -
        // exp^(-Omega(s,b)/2))
        printf(" Inelastic xs:  %0.3f mb \n\n", sigma_inel*1E3);

        const double sigma_el_2 =
            IC * std::real(gra::math::CSIntegral(f_2[0], STEP)) * PDG::GeV2barn;
        const double sigma_sd_a =
            IC * std::real(gra::math::CSIntegral(f_2[1], STEP)) * PDG::GeV2barn;
        const double sigma_sd_b =
            IC * std::real(gra::math::CSIntegral(f_2[2], STEP)) * PDG::GeV2barn;
        const double sigma_dd =
            IC * std::real(gra::math::CSIntegral(f_2[3], STEP)) * PDG::GeV2barn;

        // Scale
        const double kappa = sigma_el / sigma_el_2;

        std::cout << "Two Channel normalized to Single Channel [PROTOTEST]" << std::endl;
        printf("Calculated k = <el1>/<el2> = %0.3f \n", kappa);

        sigma_diff[0] = sigma_el_2 * kappa * 1E3;
        sigma_diff[1] = sigma_sd_a * kappa * 1E3;
        sigma_diff[2] = sigma_sd_b * kappa * 1E3;
        sigma_diff[3] = sigma_dd   * kappa * 1E3;

        printf(" pp   xs:  %0.3f mb \n",   sigma_diff[0]);
        printf(" pN*  xs:  %0.3f mb \n",   sigma_diff[1]);
        printf(" N*p  xs:  %0.3f mb \n",   sigma_diff[2]);
        printf(" N*N* xs:  %0.3f mb \n\n", sigma_diff[3]);

}

// Calculate interpolation arrays
void MEikonal::S3CalculateArray(IArray1D& arr, std::complex<double> (MEikonal::*f)(double) const) {
        std::cout << "MEikonal::S3CalculateArray:" << std::endl;
        std::vector<std::future<std::complex<double>>> futures; // std::async return values
        MTimer timer(true);

        // Loop over discretized variable
        for (std::size_t i = 0; i < arr.F.size_row(); ++i) {

                const double a = arr.MIN + i * arr.STEP;
                arr.F[i][X] = a;

                // Transform input to linear if log stepping, for the function
                const double var = (arr.islog) ? std::exp(a) : a;

        gra::aux::PrintProgress(i / static_cast<double>(arr.N + 1));
                futures.push_back(std::async(std::launch::async, f, this, var));
        }
        gra::aux::ClearProgress();
        std::cout << std::endl;
        printf("- Time elapsed: %0.1f sec \n\n", timer.ElapsedSec());

        // Retrieve std::async values
        for (const auto& i : indices(futures)) {
                arr.F[i][Y] = futures[i].get();
        }
}

// Write the array to a file
bool IArray1D::WriteArray(const std::string& filename, bool overwrite) const {

        // Do not write if file exists already
        if (gra::aux::FileExist(filename) && !overwrite) {
                // std::cout << "- Found pre-calculated" << std::endl;
                return true;
        }
        std::ofstream file;
        file.open(filename);
        if (!file.is_open()) {
                std::string str = "IArray1D::WriteArray: Fatal IO-error with: " + filename;
                throw std::invalid_argument(str);
        }

        MTimer timer(true);
        std::cout << "IArray1D::WriteArray: ";
        for (std::size_t i = 0; i < F.size_row(); ++i) {

                // Write to file
                file << std::setprecision(15)
                         << std::real(F[i][X]) << ","
                         << std::real(F[i][Y]) << ","
                         << std::imag(F[i][Y]) << std::endl;
        }
        printf("Time elapsed %0.1f sec \n", timer.ElapsedSec());
        file.close();
        return true;
}

// Read the array from a file
bool IArray1D::ReadArray(const std::string& filename) {

        std::ifstream file;
        file.open(filename);
        if (!file.is_open()) {
                std::string str = "IArray1D::ReadArray: Fatal IO-error with: " + filename;
                return false;
        }
        std::string line;
        unsigned int fills = 0;
        std::cout << "IArray1D::ReadArray: ";

        for (std::size_t i = 0; i < F.size_row(); ++i) {

                // Read every line from the stream
                getline(file, line);

                std::istringstream stream(line);
                std::vector<double> columns(3, 0.0);
                std::string element;

                // Get every line element (3 of them) separated by separator
                int k = 0;
                while (getline(stream, element, ',')) {
                        columns[k] = std::stod(element); // string to double
                        ++k;
                        ++fills;
                }
                F[i][X] = columns[0];
                F[i][Y] = std::complex<double>(columns[1],columns[2]);
        }
        file.close();

        if (fills != 3 * (N + 1)) {
                std::string str = "Corrupted file: " + filename;
                std::cout << str << std::endl;
                return false;
        }
        std::cout << "[DONE]" << std::endl;
        return true;
}


```

```
// Standard 1D-linear interpolation
//
std::complex<double> IArray1D::Interpolate1D(double a) const {

        const double EPS = 1e-5;
        if (a < MIN) { a = MIN; } // Truncate before (possible) logarithm

        // Logarithmic stepping or not
        if (islog) { a = std::log(a); }

        if (a > MAX*(1+EPS) ) {
                printf("IArray1D::Interpolate1D(%s) Input out of grid domain: "
                        "%s = %0.3f [%0.3f, %0.3f] \n", name.c_str(), name.c_str(), a, MIN, MAX);

                throw std::invalid_argument("Interpolate1D:: Out of grid domain");
        }
        int i = std::floor((a - MIN) / STEP);

        // Boundary protection
        if (i < 0)  { i = 0; } // Int needed for this, instead of unsigned int
        if (i >= (int)N) { i = N-1; } // We got N+1 elements in F

        // y = y0 + (x - x0)*[(y1 - y0)/(x1 - x0)]
        return F[i][Y] + (a - F[i][X]) * ((F[i + 1][Y] - F[i][Y]) / (F[i + 1][X] - F[i][X]));
}


// Calculate the number of cut soft Pomerons for the inelastic
//
void MEikonal::S3InitCutPomerons() {

        std::cout << "MEikonal::S3InitCutPomerons: [PROTOTEST]" << std::endl;

        // Numerical integral loop over impact parameter (b_t) space
        const double STEP = (MEikonalNumerics::MaxBT - MEikonalNumerics::MinBT) / MEikonalNumerics::NumberBT;
        P_array = std::vector<std::vector<double>>(MCUT, std::vector<double>(MEikonalNumerics::NumberBT+1, 0.0));

        for (std::size_t j = 0; j < MEikonalNumerics::NumberBT+1; ++j) {

                const double bt = MEikonalNumerics::MinBT + j*STEP;
                const double XI = std::imag(MBT.Interpolate1D(bt));

                // Poisson probabilities P_m(bt)
                for (std::size_t m = 1; m < MCUT; ++m) {

                        // Poisson ansatz
                        double P_m = std::pow(2*XI, m) / gra::math::factorial(m) * std::exp(-2*XI);
                        P_array[m][j] = P_m * bt; // *bt from jacobian \int d^2b ...
                }
        }

        // Impact parameter <bt> average probabilities
        P_cut.resize(MCUT, 0.0);
        for (std::size_t m = 0; m < MCUT; ++m) {
                P_cut[m] = gra::math::CSIntegral(P_array[m], STEP) / (MEikonalNumerics::MaxBT - MEikonalNumerics::MinBT);
                printf("P_cut[m=%2lu] = %0.5f \n", m, P_cut[m]);
        }
        std::cout << "----------------------------" << std::endl;
        printf("P_cut[SUM ] = %0.5f \n", std::accumulate(P_cut.begin(),P_cut.end(), 0.0));

        // Calculate zero-truncated average
        double avg = 0;
        for (std::size_t m = 1; m < P_cut.size(); ++m) {
                avg += m * P_cut[m];
        }
        printf("<P_cut[m>0]> = %0.2f \n\n", avg);
}

} // gra namespace ends

```