```
1: // Eikonal density and screening class
 2: //
 3: //
 4: // (c) 2017-2019 Mikael Mieskolainen
 5: // Licensed under the MIT License <a href="http://opensource.org/licenses/MIT">http://opensource.org/licenses/MIT</a>.
7: // C++
 8: #include <fstream>
9: #include <future>
10: #include <iomanip>
11: #include <iostream>
12: #include <vector>
13: #include <cmath>
14:
15: // Own
16: #include "Graniitti/MAux.h"
17: #include "Graniitti/MEikonal.h"
18: #include "Graniitti/MForm.h"
19: #include "Graniitti/MMath.h"
20: #include "Graniitti/MTimer.h"
21: #include "Graniitti/MPDG.h"
22:
23: // Libraries
24: #include "json.hpp"
25: #include "rang.hpp"
26:
27:
28: using gra::aux::indices;
29: using gra::math::msqrt;
30: using gra::math::pow2;
31: using gra::math::zi;
32:
33:
34: namespace gra {
35:
36:
            // Eikonal screening numerical integration parameters
37:
            // N.B. Compile will make significant optimizations if boundaries
38:
            // are const variables.
39:
            namespace MEikonalNumerics {
40:
                    constexpr double MinKT2 = 1E-6;
41:
42:
                    constexpr double MaxKT2 = 25.0;
43:
                    unsigned int NumberKT2 = 0;
44:
                           logKT2 = false;
                    bool
45:
                    constexpr double MinBT = 1E-6;
46:
47:
                    constexpr double MaxBT = 10.0 / PDG::GeV2fm;
48:
                    unsigned int NumberBT = 0;
49:
                           logBT = false;
                    bool
50:
                                   double FBIntegralMinKT = 1E-9;
51:
                    constexpr
                                 double FBIntegralMaxKT = 30.0;
52:
                    constexpr
53:
                    constexpr unsigned int FBIntegralN = 10000;
54:
55:
                    constexpr double MinLoopKT = 1E-4;
56:
                    double
                                  MaxLoopKT = 1.75;
57:
58:
                    std::string GetHashString() {
                            std::string str = std::to_string(MEikonalNumerics::MinKT2) +
59:
                                               std::to_string(MEikonalNumerics::MaxKT2) +
60:
61:
                                                std::to_string(MEikonalNumerics::NumberKT2) +
                                                std::to_string(MEikonalNumerics::logKT2) +
62:
63:
                                                std::to_string(MEikonalNumerics::MinBT) +
                                                std::to_string(MEikonalNumerics::MaxBT) +
64:
                                                std::to_string(MEikonalNumerics::NumberBT) +
65:
66:
                                                std::to_string(MEikonalNumerics::logBT) +
67:
                                                std::to_string(MEikonalNumerics::FBIntegralMinKT) +
68:
                                                std::to_string(MEikonalNumerics::FBIntegralMaxKT) +
69:
                                                std::to_string(MEikonalNumerics::FBIntegralN)
70:
71:
                             return str;
72:
73:
74:
            void ReadParameters() {
75:
76:
                     // Read and parse
77:
                    using json = nlohmann::json;
78:
                    const std::string inputfile = gra::aux::GetBasePath(2) + "/modeldata/" + "NUMERICS.json";
79:
                                              = gra::aux::GetInputData(inputfile);
80:
                    const std::string data
81:
                    json j;
82:
83:
                    try {
```

```
./MEikonal.cc
                          Thu Mar 28 15:24:44 2019
   84:
                                j = json::parse(data);
   85:
   86:
                                // JSON block identifier
   87:
                                const std::string XID = "NUMERICS_EIKONAL";
   88:
   89:
                                //MaxKT2 = j[XID]["MaxKT2"];
                                NumberKT2 = j[XID]["NumberKT2"];
   90:
   91:
                                logKT2 = j[XID]["logKT2"];
   92:
   93:
                                //MaxBT = j[XID]["MaxBT"]; MaxBT /= gra::math::GeV2fm; // Input as fermi, program
 uses GeV^{-1}
   94:
                                NumberBT = j[XID]["NumberBT"];
   95:
                                logBT
                                          = j[XID]["logBT"];
   96:
   97:
                                //FBIntegralMaxKT = j[XID]["FBIntegralMaxKT"];
                                                = j[XID]["FBIntegralN"];
   98:
                                //FBIntegralN
   99:
  100:
                                //MaxLoopKT = j[XID]["MaxLoopKT"];
  101:
  102:
                       } catch (...) {
                                std::string str =
  103:
  104:
                                    "MEikonalNumerics::ReadParameters: Error parsing " + inputfile + " (Check for e
xtra/missing commas)";
  105:
                               throw std::invalid_argument(str);
  106:
                      }
  107:
               }
  108:
  109:
  110:
               // Screening loop (minimum values)
  111:
               unsigned int NumberLoopKT = 15; // Number of kt steps unsigned int NumberLoopPHI = 12; // Number of phi steps
  112:
  113:
  114:
  115:
               // User setup (ND can be negative, to get below the default)
              void SetLoopDiscretization(int ND) {
  116:
                       NumberLoopKT = std::max(3, 3 * ND + (int)NumberLoopKT);
  117:
                       NumberLoopPHI = std::max(3, 3 * ND + (int)NumberLoopPHI);
  118:
  119:
               }
  120:
  121: } // Namespace MEikonal ends
  122:
  123:
  124: MEikonal::MEikonal() {
  125: }
  126:
  127:
  128: MEikonal::~MEikonal() {
  129: }
  130:
  131:
  132: // Return total, elastic, inelastic cross sections
  133: void MEikonal::GetTotXS(double& tot, double& el, double& in) const {
  134:
              tot = sigma_tot;
               el = sigma_el;
  135:
               in = sigma_inel;
  136:
  137: }
  138:
  139:
  140: // Construct density and amplitude
  141: void MEikonal::S3Constructor(double s_in, const std::vector<gra::MParticle>& initialstate_in, bool onlyeiko
nal) {
  142:
  143:
               // This first
  144:
               MEikonalNumerics::ReadParameters();
  145:
  146:
               // Mandelstam s and initial state
  147:
               s = s in;
  148:
               INITIALSTATE = initialstate_in;
  149:
  150:
               // Calculate hash based on all free variables -> if something changed,
  151:
               // calculate new densities
  152:
  153:
               // Proton density
  154:
  155:
               {
  156:
                        std::cout << "Initializing <eikonal density> array:" << std::endl;</pre>
                       MBT.sqrts = msqrt(s); // FIRST THIS
  157:
  158:
                       MBT.Set("bt", MEikonalNumerics::MinBT, MEikonalNumerics::MaxBT, MEikonalNumerics::NumberBT,
MEikonalNumerics::logBT);
                       MBT.InitArray(); // Initialize (call last!)
 159:
  1.60:
                        const std::string hstr = std::to_string(s) + std::to_string(INITIALSTATE[0].pdg) + "_" + st
d::to_string(INITIALSTATE[1].pdg) +
```

```
./MEikonal.cc
                          Thu Mar 28 15:24:44 2019
  162:
                                                  PARAM_SOFT::GetHashString() + MEikonalNumerics::GetHashString();
                        const unsigned long hash = gra::aux::djb2hash(hstr);
  163:
  164:
                        const std::string filename = gra::aux::GetBasePath(2) +
                                                                                    "/eikonal/" + "MBT_" + std::to_stri
  165:
ng(INITIALSTATE[0].pdg) +
                                                                                     "_" + std::to_string(INITIALSTATE[
 166:
1].pdg) +
  167:
                                                       "_" + gra::aux::ToString(msqrt(s),0) + "_" + std::to_string(ha
sh);
  168:
  169:
                        // Try to read pre-calculated
  170:
                        bool ok = MBT.ReadArray(filename);
  171:
  172:
                        while (!ok) { // Problem, re-calculate
  173:
                                 // Pointer to member function: ReturnType (ClassType::*) (ParameterTypes...)
  174:
                                 std::complex<double> (MEikonal::*f) (double) const = &MEikonal::S3Density;
  175:
                                 S3CalculateArray(MBT,f);
  176:
                                 MBT.WriteArray(filename, true);
  177:
                                ok = MBT.ReadArray(filename);
  178:
                        }
  179:
               }
  180:
  181:
                // Calculate cross sections (is fast)
  182:
               S3CalcXS();
  183:
                // Init cut Pomerons (is fast)
  184:
  185:
               S3InitCutPomerons():
  186:
  187:
                // Amplitude
  188:
               if (onlyeikonal == false) {
                        std::cout << "Initializing <eikonal amplitude> array:" << std::endl;</pre>
  189:
  190:
  191:
                        MSA.sqrts = msqrt(s); // FIRST THIS
  192:
                        MSA.Set("kt2", MEikonalNumerics::MinKT2, MEikonalNumerics::MaxKT2, MEikonalNumerics::Number
KT2. MEikonalNumerics::logKT2);
                        MSA.InitArray(); // Initialize (call last!)
  193:
  194:
  195:
                        const std::string hstr = std::to_string(s)
  196:
                                                                            + std::to_string(INITIALSTATE[0].pdg)
  197:
                                                                            + std::to_string(INITIALSTATE[1].pdg)
  198:
                                                                            + PARAM_SOFT::GetHashString()
  199:
                                                                            + MEikonalNumerics::GetHashString();
  200:
  201:
                        const unsigned long hash = gra::aux::djb2hash(hstr);
                        const std::string filename = gra::aux::GetBasePath(2) + "/eikonal/" + "MSA_" +
  202:
  203:
                                                                                     std::to_string(INITIALSTATE[0].pdg
  + "_" + std::to_string(INITIALSTATE[1].pdg) +
  204:
                                                       "_" + gra::aux::ToString(msqrt(s),0) + "_" + std::to_string(ha
sh);
  205:
  206:
                        // Try to read pre-calculated
  207:
                        bool ok = MSA.ReadArray(filename);
  208:
  209:
                        while (!ok) { // Problem, re-calculate
                                 // Pointer to member function: ReturnType (ClassType::*) (ParameterTypes...)
  210:
  211:
                                 std::complex<double> (MEikonal::*f) (double) const = &MEikonal::S3Screening;
  212:
                                 S3CalculateArray(MSA, f);
  213:
                                MSA.WriteArray(filename, true);
  214:
                                ok = MSA.ReadArray(filename);
  215:
                        }
  216:
                // Tag it done
  217:
  218:
               S3INIT = true;
  219: }
  220:
  221:
  222: // Proton bt-density by Fourier-Bessel transform of the t-density
  223: // see <a href="http://mathworld.wolfram.com/HankelTransform.html">http://mathworld.wolfram.com/HankelTransform.html</a>
  224: //
  225: //
  226: // For eikonalization see:
  227: // [REFERENCE: Desgrolard, Giffon, Martynov, Predazzi, https://arxiv.org/abs/hep-ph/9907451v2]
  228: // [REFERENCE: Desgrolard, Giffon, Martynov, Predazzi, https://arxiv.org/abs/hep-ph/0001149]
  229: //
  230: //
  231: // For some discussion about Odderon versus Pomeron, see: 232: // [REFERENCE: Ewerz, Maniatis, Nachtmann, https://arxiv.org/abs/1309.3478]
  233: std::complex<double> MEikonal::S3Density(double bt) const{
  234:
  235:
                // Discretization of kt
  236:
                const double kt_STEP =
  237:
                    (MEikonalNumerics::FBIntegralMaxKT - MEikonalNumerics::FBIntegralMinKT) /
```

238:

MEikonalNumerics::FBIntegralN;

```
./MEikonal.cc
                                                           Thu Mar 28 15:24:44 2019
    239:
                                   //N + 1!
    240:
    241:
                                   std::vector<std::complex<double>> f (MEikonalNumerics::FBIntegralN + 1, 0.0);
    242:
    243:
                                   // Initial state configuration [NOT IMPLEMENTED = SAME FOR pp and ppbar]
    244:
                                   if (INITIALSTATE[0].pdg == PDG::PDG_p && INITIALSTATE[1].pdg == PDG::PDG_p) {
    245:
    246:
                                   // ppbar
    247:
                                   } else if ((INITIALSTATE[0].pdg == PDG::PDG_p && INITIALSTATE[1].pdg == -PDG::PDG_p) | |
    248:
    249:
                                                                                 (INITIALSTATE[0].pdg == -PDG::PDG_p && INITIALSTATE[1].pdg == PDG::PDG_p)) {
                                                     // TBD
    250:
    251:
                                  }
    252:
    253:
                                   // Loop over
    254:
                                  for (const auto& i : indices(f)) {
    255:
    256:
                                                      const double kt = MEikonalNumerics::FBIntegralMinKT + i * kt_STEP;
    257:
    258:
                                                      // negative, with Mandelstam t ~= -kt^2
                                                      const double t = -gra::math::pow2(kt);
    259:
    260:
    261:
                                                      // Proton form factors (could be here extended to multichannel)
    262:
                                                     const double F_i = gra::form::S3F(t);
    263:
                                                     const double F_k = gra::form::S3F(t);
    264:
    265:
                                                      // Pomeron trajectory alpha(t)
    266:
                                                      const double alpha_P = gra::form::S3PomAlpha(t);
    267:
    268:
                                                      // Pomeron exchange amplitude:
                                                     // [Regge signature x Proton Form Factor x coupling x Proton \left( \frac{1}{2} \right) = \frac{1}{2} \left( \frac{1}{2} \right) \left( \frac{1}{2} \right)
    269:
                                                      // Form Factor x coupling x Propagator ]
    270:
    271:
                                                      const double s0 = 1.0; // Typical (normalization) scale GeV^{-2}
    272:
    273:
                                                      // const std::complex<double> eta_0 =
                                                      // std::exp(gra::math::zi*PARAM_SOFT::PHASE_O);
    274:
    275:
                                                     std::complex<double> A =
    276:
    277:
                                                             gra::math::pow2(PARAM_SOFT::qN_P) * F_i * F_k *
    278:
                                                                gra::form::ReggeEta(alpha_P, 1) *
                                                               std::pow(s / s0, alpha_P - 1.0);// Pomeron (C-even)
    279:
    280:
    281:
                                                        // Value
                                                      f[i] = A * gra::math::BESSJ0(bt * kt) * kt;
    282:
                                                      //f[i] = A * std::cyl_bessel_j(0, bt * kt) * kt; // c++17
    283:
    284:
    285:
                                   // Fourier-Bessel transformation denominator
                                  const double TD = 2.0*gra::math::PI;
    286:
    287:
    288:
                                  return gra::math::CSIntegral(f, kt STEP) / TD;
    289: }
    290:
    291:
    292: // Calculate elastic screening amplitude by "Eikonalization",
    293: // in kt^2 obtained via bt-space Fourier-Bessel integral
    294: //
    295: //
    296: // Amplitude in bt-space: A_el(b_t) = i(1 - exp(i*XI(b_t)/2))
    297: //
    298: std::complex<double> MEikonal::S3Screening(double kt2) const {
    299:
    300:
                                   // Local discretization
    301:
                                  const double STEP = (MEikonalNumerics::MaxBT - MEikonalNumerics::MinBT) /
    302:
                                                                                                                                   MEikonalNumerics::FBIntegralN;
    303:
    304:
                                   const double kt = gra::math::msqrt(kt2);
    305:
                                   std::vector<std::complex<double>> f (MEikonalNumerics::FBIntegralN + 1, 0.0);
    306:
    307:
                                   // Numerical integral loop over impact parameter (b_t) space
    308:
                                  for (const auto& i : indices(f)) {
    309:
                                                      const double bt = MEikonalNumerics::MinBT + i * STEP;
    310:
    311:
                                                      const std::complex<double> XI = MBT.Interpolate1D(bt);
    312:
    313:
                                                      // I. STANDARD EIKONAL APPROXIMATION
    314:
                                                      const std::complex<double> A =
                                                             gra::math::zi * (1.0 - std::exp(gra::math::zi * XI / 2.0));
    315:
    316:
                                                     f[i] = A * gra::math::BESSJ0(bt * kt) * bt;
    317:
```

 $//f[i] = A * std::cyl_bessel_j(0, bt * kt) * bt; // c++17$

const double C = 2.0 * gra::math::PI; // phi-integral

318: 319: 320:

321:

```
322:
             // Numerical integration
323:
             return (2.0 * s) * C * gra::math::CSIntegral(f, STEP);
324: }
325:
326:
327: // Calculate screened total, elastic and inelastic cross sections
328: // in the eikonal model
329: //
330: //
331: //  \int d^2b f(b) = \int \int_0^{\infty} \int \int_0^{\infty} dx d\theta d\theta d\theta
                       332: //
333: //
334: void MEikonal::S3CalcXS() {
335:
            std::cout << "MEikonal::S3CalcXS:" << std::endl << std::endl;</pre>
336:
337:
             // Local discretization
             const unsigned int N = 2 * 3000; // even number
338:
339:
             const double STEP = (MEikonalNumerics::MaxBT - MEikonalNumerics::MinBT) / N;
340:
341:
             // Two channel eikonal eigenvalue solutions obtained via symbolic
             // calculation see e.g.
342:
             //
343:
             // [REFERENCE: Khoze, Martin, Ryskin, https://arxiv.org/abs/hep-ph/0007359v2]
344:
345:
             // [REFERENCE: Roehr, http://inspirehep.net/record/1351489/files/Thesis-2014-Roehr.pdf]
346:
             // Unitary transformation matrix
347:
             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*
348:
349:
350:
351:
             // Eigenvalues
352:
353:
           const std::vector<double> lambda = {
               gra::math::pow2(1.0 - PARAM_SOFT::gamma),
354:
                 gra::math::pow2(1.0 + PARAM_SOFT::gamma),
355:
356:
                 1.0 - gra::math::pow2(PARAM_SOFT::gamma),
                 1.0 - gra::math::pow2(PARAM_SOFT::gamma));
357:
             // -----
358:
359:
360:
             std::vector<std::complex<double>> f_tot(N + 1, 0.0);
361:
             std::vector<std::complex<double>> f_el(N + 1, 0.0);
362:
             std::vector<std::complex<double>> f_in(N + 1, 0.0);
363:
364:
365:
             // Two-channel eikonal, N+1!
             std::vector<std::vector<std::complex<double>>> f 2(
366:
367:
                 4, std::vector<std::complex<double>> (N + 1, 0.0));
368:
369:
             // Numerical integral loop over impact parameter (b_t) space
370:
             for (const auto& i : indices(f_tot)) {
371:
372:
                     const double bt = MEikonalNumerics::MinBT + i * STEP;
373:
374:
                      // Calculate density
                     const std::complex<double> XI = MBT.Interpolate1D(bt);
375:
376:
377:
378:
                     // Single-Channel eikonal
379:
380:
                     // 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)
381:
382:
383:
                      // TOTAL: Im A_el(s,b)
                     f_tot[i] = std::imag(A_el);
384:
385:
                      // ELASTIC: A_{el}(s,b)^2
386:
387:
                     f_el[i] = gra::math::abs2(A_el);
388:
                      // INELASTIC: 2 \text{Im A\_el}(s,b) - |A\_el(s,b)|^2
389:
                      f_in[i] = 2.0 * std::imag(A_el) - gra::math::abs2(A_el);
390:
391:
392:
393:
                      // Two-channel Eikonal solutions for pp->p(*)p(*)
394:
                      const std::vector<std::complex<double>> sol = {
395:
                          1.0 - std::exp(gra::math::zi * lambda[0] * XI / 2.0),
396:
                          1.0 - std::exp(gra::math::zi * lambda[1] * XI / 2.0),
397:
398:
                          1.0 - std::exp(gra::math::zi * lambda[2] * XI / 2.0),
                          1.0 - std::exp(gra::math::zi * lambda[3] * XI / 2.0)};
399:
400:
401:
                     for (std::size_t k = 0; k < 4; ++k) {</pre>
402:
```

// Amplitude squared

Thu Mar 28 15:24:44 2019

./MEikonal.cc

403:

```
./MEikonal.cc
                            Thu Mar 28 15:24:44 2019
  404:
                                   f_2[k][i] = gra::math::abs2(
                                        (sol[0] * cc[k][0] + sol[1] * cc[k][1] +
  405:
                                        sol[2] * cc[k][2] + sol[3] * cc[k][3]) / 4.0);
  406:
  407:
                                   f_2[k][i] *= bt; // Jacobian
  408:
                         }
  409:
                         // Jacobian
  410:
  411:
                         f_tot[i] *= bt;
                         f_el[i] *= bt;
  412:
                         f_in[i] *= bt;
  413:
  414:
                }
  415:
  416:
                // 2D-Integral factor
  417:
                const double IC = 2.0 * gra::math::PI;
  418:
  419:
                // Composite Simpson's rule, real is taken for C++ reasons in order to
                // be able to substitute into double
  420:
                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;
  421:
  422:
  423:
                // sigma_inel = sigma_tot - sigma_el; // cross check
  424:
  425:
  426:
                // Comments for the "multichannel" formalism [next version]
  427:
                // Total cross section:
                                              2*\int d^2b sum_ik |a_i|^2|a_k|^2 (1 -
  428:
                // \exp^{-Omega(s,b)/2)}
  429:
  430:
                std::cout << "Single Channel Eikonal:" << std::endl;</pre>
  431:
                printf(" Total xs:
                                           %0.3f mb \n", sigma_tot*1E3);
  432:
                // Elastic cross section: \int d^2b (sum_ik |a_i|^2|a_k|^2 (1 - // exp^ (-Omega(s,b)/2))^2
  433:
  434:
  435:
                printf(" Elastic xs:
                                           %0.3f mb \n", sigma_el*1E3);
  436:
  437:
                // Inelastic cross section: \int d^2b \ sum_ik \ |a_i|^2 |a_k|^2 (1 - a_k)^2
                // exp^{(-Omega(s,b)/2)}
  438:
                printf(" Inelastic xs: %0.3f mb \n\n", sigma_inel*1E3);
  439:
  440:
  441:
                const double sigma el 2 =
  442:
                   IC * std::real(gra::math::CSIntegral(f_2[0], STEP)) * PDG::GeV2barn;
  443:
                const double sigma sd a =
                   IC * std::real(gra::math::CSIntegral(f_2[1], STEP)) * PDG::GeV2barn;
  444:
  445:
                const double sigma_sd_b =
  446:
                    IC * std::real(gra::math::CSIntegral(f_2[2], STEP)) * PDG::GeV2barn;
  447:
                const double sigma_dd =
  448:
                   IC * std::real(gra::math::CSIntegral(f_2[3], STEP)) * PDG::GeV2barn;
  449:
  450:
                // Scale
  451:
                const double kappa = sigma_el / sigma_el_2;
  452:
                std::cout << "Two Channel normalized to Single Channel [PROTOTEST]" << std::endl;</pre>
  453:
                printf("Calculated k = \langle el1 \rangle / \langle el2 \rangle = %0.3f \n", kappa);
  454:
  455:
  456:
                sigma_diff[0] = sigma_el_2 * kappa * 1E3;
                sigma_diff[1] = sigma_sd_a * kappa * 1E3;
  457:
                sigma_diff[2] = sigma_sd_b * kappa * 1E3;
  458:
  459:
                sigma_diff[3] = sigma_dd * kappa * 1E3;
  460:
  461:
                printf(" pp
                               xs: %0.3f mb \n",
                                                       sigma_diff[0]);
                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]);
  462:
  463:
  464:
  465:
  466: }
  467:
  468: // Calculate interpolation arrays
  469: void MEikonal::S3CalculateArray(IArray1D& arr, std::complex<double> (MEikonal::*f)(double) const) {
  470:
                std::cout << "MEikonal::S3CalculateArray:" << std::endl;</pre>
  471:
                std::vector<std::future<std::complex<double>>> futures; // std::async return values
  472:
                MTimer timer(true);
  473:
  474:
                // Loop over discretized variable
  475:
                for (std::size_t i = 0; i < arr.F.size_row(); ++i) {</pre>
  476:
  477:
                         const double a = arr.MIN + i * arr.STEP;
  478:
                         arr.F[i][X] = a;
  479:
                          // Transform input to linear if log stepping, for the function
  480:
  481:
                         const double var = (arr.islog) ? std::exp(a) : a;
  482:
                gra::aux::PrintProgress(i / static_cast<double>(arr.N + 1));
  483:
  484:
                         futures.push_back(std::async(std::launch::async, f, this, var));
  485:
  486:
                gra::aux::ClearProgress();
```

```
std::cout << std::endl;</pre>
            printf("- Time elapsed: %0.1f sec \n\n", timer.ElapsedSec());
488:
489:
490:
             // Retrieve std::async values
             for (const auto& i : indices(futures)) {
491:
492:
                     arr.F[i][Y] = futures[i].get();
             }
493:
494: }
495:
496: // Write the array to a file
497: bool IArray1D::WriteArray(const std::string& filename, bool overwrite) const {
498:
499:
             // Do not write if file exists already
500:
             if (gra::aux::FileExist(filename) && !overwrite) {
                     // std::cout << "- Found pre-calculated" << std::endl;</pre>
501:
502:
                     return true:
503:
             }
504:
             std::ofstream file;
505:
             file.open(filename);
506:
             if (!file.is_open()) {
                     std::string str = "IArray1D::WriteArray: Fatal IO-error with: " + filename;
507:
508:
                     throw std::invalid_argument(str);
509:
             }
510:
511:
             MTimer timer(true):
             std::cout << "IArray1D::WriteArray: ";</pre>
512:
513:
            for (std::size_t i = 0; i < F.size_row(); ++i) {</pre>
514:
515:
                      // Write to file
516:
                     file << std::setprecision(15)</pre>
                               << std::real(F[i][X]) << ","
517:
                               << std::real(F[i][Y]) << ","
518:
519:
                               << std::imag(F[i][Y]) << std::endl;
520:
            printf("Time elapsed %0.1f sec \n", timer.ElapsedSec());
521:
522:
             file.close();
523:
            return true;
524: }
525:
526: // Read the array from a file
527: bool IArray1D::ReadArray(const std::string& filename) {
528:
529:
             std::ifstream file;
530:
             file.open(filename);
531:
             if (!file.is_open()) {
                     std::string str = "IArray1D::ReadArray: Fatal IO-error with: " + filename;
532:
533:
                     return false;
534:
535:
             std::string line;
             unsigned int fills = 0;
536:
             std::cout << "IArray1D::ReadArray: ";</pre>
537:
538:
539:
             for (std::size_t i = 0; i < F.size_row(); ++i) {</pre>
540:
                     \ensuremath{//} Read every line from the stream
541:
542:
                     getline(file, line);
543:
544:
                     std::istringstream stream(line);
545:
                     std::vector<double> columns(3, 0.0);
                     std::string element;
546:
547:
548:
                     // Get every line element (3 of them) separated by separator
549:
                     int k = 0;
550:
                     while (getline(stream, element, ',')) {
                              columns[k] = std::stod(element); // string to double
551:
552:
                              ++k;
553:
                              ++fills;
554:
                     F[i][X] = columns[0];
555:
556:
                     F[i][Y] = std::complex<double>(columns[1],columns[2]);
557:
558:
             file.close();
559:
             if (fills != 3 * (N + 1)) {
560:
                     std::string str = "Corrupted file: " + filename;
561:
                     std::cout << str << std::endl;</pre>
562:
563:
                     return false;
564:
             std::cout << "[DONE]" << std::endl;</pre>
565:
566:
             return true;
567: }
568:
```

Thu Mar 28 15:24:44 2019

./MEikonal.cc

569:

```
570: // Standard 1D-linear interpolation
 571: //
 572: std::complex<double> IArray1D::Interpolate1D(double a) const {
 573:
               const double EPS = 1e-5;
 574:
 575:
              if (a < MIN) { a = MIN; } // Truncate before (possible) logarithm</pre>
 576:
 577:
               // Logarithmic stepping or not
              if (islog) { a = std::log(a); }
 578:
  579:
  580:
               if (a > MAX*(1+EPS) ) {
                       printf("IArray1D::Interpolate1D(%s) Input out of grid domain: "
 581:
 582:
                               "%s = %0.3f [%0.3f, %0.3f] \n", name.c_str(), name.c_str(), a, MIN, MAX);
  583:
  584:
                       throw std::invalid_argument("Interpolate1D:: Out of grid domain");
  585:
               int i = std::floor((a - MIN) / STEP);
 586:
 587:
  588:
               // Boundary protection
               if (i < 0) { i = 0; } // Int needed for this, instead of unsigned int
 589:
               if (i >= (int)N) { i = N-1; } // We got N+1 elements in F
 590:
 591:
 592:
               // y = y0 + (x - x0) *[(y1 - y0)/(x1 - x0)]
 593:
              return F[i][Y] + (a - F[i][X]) * ((F[i + 1][Y] - F[i][Y]) / (F[i + 1][X] - F[i][X]));
 594: }
 595:
 596:
 597: // Calculate the number of cut soft Pomerons for the inelastic
 598: //
 599: void MEikonal::S3InitCutPomerons() {
 600:
 601:
               std::cout << "MEikonal::S3InitCutPomerons: [PROTOTEST]" << std::endl;</pre>
  602:
 603:
              // Numerical integral loop over impact parameter (b_t) space
              const double STEP = (MEikonalNumerics::MaxBT - MEikonalNumerics::MinBT) / MEikonalNumerics::NumberB
 604:
 605:
              P_array = std::vector<std::vector<double>> (MCUT, std::vector<double> (MEikonalNumerics::NumberBT+1,
0.0));
 606:
  607:
              for (std::size_t j = 0; j < MEikonalNumerics::NumberBT+1; ++j) {</pre>
 608:
 609:
                       const double bt = MEikonalNumerics::MinBT + j*STEP;
                       const double XI = std::imag(MBT.Interpolate1D(bt));
  610:
 611:
 612:
                       // Poisson probabilities P_m(bt)
 613:
                       for (std::size_t m = 1; m < MCUT; ++m) {</pre>
  614:
  615:
                               // Poisson ansatz
 616:
                               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 ...
 617:
 618:
                       }
  619:
              }
  620:
 621:
              // Impact parameter <bt> average probabilities
              P_cut.resize(MCUT, 0.0);
 622:
 623:
               for (std::size_t m = 0; m < MCUT; ++m) {</pre>
 624:
                      P_cut[m] = gra::math::CSIntegral(P_array[m], STEP) / (MEikonalNumerics::MaxBT - MEikonalNum
erics::MinBT);
                      printf("P_cut[m=%21u] = %0.5f \n", m, P_cut[m]);
 625:
  626:
               }
               std::cout << "-----
                                             ----" << std::endl;
 627:
  628:
              printf("P_cut[SUM] = %0.5f \n", std::accumulate(P_cut.begin(),P_cut.end(), 0.0));
  629:
 630:
              // Calculate zero-truncated average
 631:
              double avg = 0;
 632:
               for (std::size_t m = 1; m < P_cut.size(); ++m) {</pre>
  633:
                      avg += m * P_cut[m];
  634:
 635:
               printf("<P_cut[m>0]> = %0.2f \n\n", avg);
 636: }
 637:
  638: } // gra namespace ends
  639:
```