Monte Carlo generator for Exclusive Diffraction
Version 2.0

Physics and Manual

R.A. Ryutin

Institute for High Energy Physics, NRC “Kurchatov Institute”,
142 281 Protvino, Russia

Abstract
ExDiff is a Monte Carlo event generator for simulation of Exclusive Diffractive processes in proton-proton collisions. The present version includes reactions: elastic scattering $pp \rightarrow pp$ at 7, 8, 13, 14 TeV; $pp \rightarrow p + R + p$, $R = f_0(1500)$, $f_0(1710)$, $f_2(1950)$ at 8 and 13 TeV, $f_2(1270)$ at 8 TeV, $f_2(2220)$ at 13 TeV. In the future versions many processes of Central Exclusive Diffractive Production will be added. This version is linked to PYTHIA 8 (to make resonance decays and hadronization) and also to ROOT and HEPMC output via PYTHIA interface. Also some test files of Born distributions for CEDP of two pions are added.

Keywords
Exclusive Diffraction – Elastic Scattering – Central Exclusive Diffractive Production – Pomeron-Pomeron – photon-Pomeron – proton-proton – cross sections – event generator – forward physics – Regge-Eikonal model
# Contents

1 Introduction  

2 Physics Overview
   2.1 Elastic scattering
   2.2 Central Exclusive Diffractive Production

3 Program Overview
   3.1 General information
   3.2 Basic classes and functions

4 Program Installation

5 Getting Started with the Simple Example
   5.1 For developers

Index of basic classes, methods and variables
1 Introduction

Substantial fraction (about 40% at LHC) of total cross section of pp interactions is due to diffractive processes. And about 60% of these events are exclusive. So, simulation of such events is one of the basic tasks at LHC.

At this moment there is no unique definition of diffraction:

- Interactions where the beam particles emerge intact or dissociated into low-mass states.
- Interactions mediated by t-channel exchange of object with the quantum numbers of the vacuum, color singlet exchange or Pomeron. Descriptions of the Pomeron are based on phenomenological approaches or on QCD.

In general such processes lead to final state particles separated by large rapidity gaps. However only a fraction of Large Rapidity Gap (LRG) events is due to diffractive processes, gaps can arise also from fluctuations in the hadronisation processes.

The key experimental trigger for diffraction is the angle distribution, which gives the typical diffractive pattern with zero-angle maximum and one or, sometimes, two dips. Here wave properties of hadrons play the main role. From this distribution we can make the conclusion about size and shape of the scatterer, or the "interaction region".

Elastic scattering is the basic exclusive process, i.e. so called "standard candle". Advantages of this process are:

- Clear signature: both initial particles remain intact and should be detected in the final state.
- Small number of variables in the differential cross-section.
- Large value of the cross-section.
- Huge number of experimental data at different energies.
- From the theoretical point of view: we have the possibility to extract size and shape of the “interaction region” from the slope and fine structure of the $t$-distribution.
- Also we can “calibrate” diffractive models for further calculations of absorptive corrections in other exclusive processes.

Experimental difficulty is mainly due to closeness of final protons to beams. That is why we need special runs of an accelerator to avoid different contaminations like pile-up events, for example.

Another process is the Central Exclusive Diffractive Production (CEDP). CEDP gives us unique experimental possibilities for particle searches and investigations of diffraction itself. This is due to several advantages of the process:

- clear signature of the process: both protons remain intact plus LRG;
- possibility to use ”missing mass method” that improve the mass resolution;
- usually background is strongly suppressed, especially for “hard” production, due to $J_z = 0$ selection rule;
- spin-parity analysis of the central system can be done by the use of azimuthal distributions;
- interesting measurements concerning the interplay between ”soft” and ”hard” scales are possible [1],[2].

All these properties are realized in common CMS/TOTEM detector measurements at LHC [3].

Theoretically calculations of CEDP and elastic scattering are closely related, especially when we try to take into account all the unitarity effects (absorption, “gap survival probability”).

In the paper we present the Monte-Carlo event generator ExDiff. The generator is devoted to the simulation of Exclusive Diffractive processes in proton-proton collisions.
(elastic process and CEDP of low mass resonances in this version). All these processes are depicted in the Fig. 1.

The present version includes reactions: elastic scattering $pp \rightarrow pp$ at 7, 8, 13, 14 TeV; CEDP of low mass resonances $pp \rightarrow p + R + p$, $R = f_0(1500), f_0(1710), f_2(1950)$ at 8 and 13 TeV, $f_2(1270)$ at 8 TeV, $f_2(2220)$ at 13 TeV. In the future versions many other processes of CEP will be added: $pp \rightarrow p + R + p$, where $R$ is a resonance like Higgs boson, for example; $pp \rightarrow p + A + B + p$, where $A, B$ are hadrons, jets or gauge bosons (see Figs. 1(c)-(j)).

![Diagram](image)

Figure 1: Amplitudes for exclusive diffractive processes: a) elastic scattering with $T_{el}$ presented in (5); b) general process of central exclusive diffractive production (CEDP) with central production born amplitude $C$ and absorptive corrections $V_{in} = V(s, b)$ and $V_{out} = V(s^t, b)$ presented in (24), (25). Versions of the CEDP Born amplitudes are: c) low mass resonance production; d) di-hadron production; e) high mass resonance production; f) high mass di-hadron production; g) di-jet production; h) nonperturbative photon-pomeron low mass resonance production; i) perturbative photon-pomeron high mass resonance production and j) high mass di-jet production. Possible additional corrections due to unitarization procedure are depicted as dotted ovals. In d) and f) it is possible to produce also di-bosons like $\gamma\gamma$, $ZZ$ or $WW$ instead of di-hadrons $hh$. 
2 Physics Overview

2.1 Elastic scattering

2.1.1 Kinematics

The diagram of the elastic scattering $p + p \rightarrow p + p$ is presented in Fig. 1. The momenta are $p_1, p_2, p_1', p_2'$ respectively. In the center-of-mass frame these can be represented as follows ($p \equiv (p_0, p_z, \vec{p}), \vec{p} \equiv (p_x, p_y)$):

$$p_1 = \left( \sqrt{\frac{s}{2}}, \frac{\sqrt{s}}{2} \beta, 0 \right), \quad p_2 = \left( \frac{-\sqrt{s}}{2}, \frac{\sqrt{s}}{2} \beta, 0 \right).$$

With this notation, the four momentum transfer is

$$\Delta = \left( 0, \frac{-t}{\sqrt{s}} \beta, \vec{\Delta} \right), \quad p_1' = p_1 - \Delta, \quad p_2' = p_2 + \Delta,$$

$$-t = \tau^2 = \frac{s\beta^2}{2} \left( 1 - \sqrt{1 - \frac{4\Delta^2}{s\beta^2}} \right) \approx \Delta^2, \quad \beta = \sqrt{1 - \frac{4m_p^2}{s}}.$$

Cross-section for elastic scattering can be calculated as

$$\frac{d\sigma_{el}}{d\tau^2} = \frac{|T_{el}(s, \tau)|^2}{16\pi s^2},$$

$$T_{el}(s, \tau) = 4\pi s \int_0^\infty db^2 J_0(b\tau) T_{el}(s, b).$$

2.1.2 Model for the elastic scattering

Here is the outlook of the model for elastic scattering used in this version of the generator.

In the Regge-eikonal approach the elastic nonflip scattering amplitude looks like

$$T_{el}(s, b) = \frac{e^{2i\delta_{el}(s, b)} - 1}{2i},$$

$$\delta_{el}(s, b) = \frac{1}{16\pi s} \int_0^\infty d\tau^2 J_0(b\tau) \delta_{el}(s, \tau),$$

where $s$ and $t = -\tau^2$ are the Mandelstam variables, $b = |\vec{b}|$ is the impact parameter, and eikonal $\delta(s, \tau)$ is parametrized as in the work [4]

$$\delta_{el}(s, \tau) = g_{ppP}^2(-\tau^2) \left( i + \frac{\pi(\alpha_P(-\tau^2) - 1)}{2} \right) \pi\alpha'_P(-\tau^2) \left( \frac{s}{2s_0} \right)^{\alpha_P(-\tau^2)},$$

where

$$\alpha_P(t) = 1 + \frac{\alpha_P(0) - 1}{1 - \frac{t}{\tau_a}}, \quad g_{ppP}(t) = \frac{g_{ppP}(0)}{(1 - a_g t)^2},$$
### 2.2 Central Exclusive Diffractive Production

#### 2.2.1 Kinematics of CEDP

Let us consider the kinematics of two CEDP processes

\[
\begin{align*}
  h_1(p_1) + h_2(p_2) &\rightarrow h_1(p'_1) + R(p_R) + h_2(p'_2), \\
  h_1(p_1) + h_2(p_2) &\rightarrow h_1(p'_1) + \{a(k_a) + b(k_b)\} + h_2(p'_2),
\end{align*}
\]

(10)

(11)

with four-momenta indicated in parentheses. Initial hadrons remain intact, \(\{a b\}\) can be di-boson or di-hadron system and \(R\) denotes a resonance, \(\text{"+"}\) signs denote LRG.

We use the following set of variables:

\[
\begin{align*}
  s &= (p_1 + p_2)^2, \\
  s' &= (p'_1 + p'_2)^2, \\
  t_{1,2} &= (p_{1,2} - p'_{1,2})^2, \\
  s_{1,2} &= (p'_{1,2} + p_R)^2 \text{ or } (p_{1,2} + k_a + k_b)^2,
\end{align*}
\]

(12)

\[
\begin{align*}
  s_{1(a,b)} &= (p'_1 + k_{a,b})^2, \\
  s_{2(a,b)} &= (p'_2 + k_{a,b})^2, \\
  t_{a,b} &= (p_1 - p'_1 - k_{a,b})^2 = (p_2 - p'_2 - k_{b,a})^2, \\
  \bar{s} &= \frac{s - 2m^2}{2} + \frac{s}{2} \sqrt{1 - \frac{4m^2}{s}} \simeq s.
\end{align*}
\]

(13)

In the light-cone representation \(p = \{p_+, p_-, \vec{p}_\perp\}\)

\[
\begin{align*}
  p_1 &= \left\{ \sqrt{\frac{s}{2}}, \frac{m^2}{\sqrt{2s}}; \vec{0} \right\}, \Delta_1 = \left\{ \xi_1 \sqrt{\frac{s}{2}}, \frac{-\bar{\Delta}_1^2 - \xi_1 m^2}{(1 - \xi_1)\sqrt{2s}}; \bar{\Delta}_1 \right\}, \\
  p_2 &= \left\{ \frac{m^2}{\sqrt{2s}}, \sqrt{\frac{s}{2}}; \vec{0} \right\}, \Delta_2 = \left\{ -\bar{\Delta}_2^2 - \xi_2 m^2 (1 - \xi_2)\sqrt{2s}; \xi_2 \sqrt{\frac{s}{2}}; \bar{\Delta}_2 \right\}, \\
  p'_{1,2} &= p_{1,2} - \Delta_{1,2}, \\
  p'_{1,2}^2 &= m^2,
\end{align*}
\]

(14)

and additional notations for the \(2 \rightarrow 4\) process are (approximately, for \(|\bar{\Delta}_{1,2} | \ll |\vec{k}|\))

\[
\begin{align*}
  k_{a,b} &\approx \left\{ \frac{m_+}{\sqrt{2}} e^{y_c \pm \Delta y}, \frac{m_-}{\sqrt{2}} e^{-y_c \pm \Delta y}; \pm \vec{k} \right\}, \bar{k}_{a,b} \approx \pm \vec{k}, \\
  k_{a,b}^2 &= m_0^2, m_1^2 = m_0^2 + \vec{k}^2.
\end{align*}
\]

(15)

Here \(\xi_{1,2}\) are fractions of hadrons' longitudinal momenta lost, \(y_c\) denotes the rapidity of the central system, \(\Delta y = (y_b - y_a)/2\), where \(y_{a,b}\) are rapidities of particles \(a, b\). From the

| Parameter | Value |
|-----------|-------|
| \(\alpha_P(0) - 1\) | 0.109 |
| \(\tau_a\) | 0.535 GeV² |
| \(g_{ppP}(0)\) | 13.8 GeV |
| \(a_g\) | 0.23 GeV⁻² |
above notations we can obtain the relations:

\[ t_{1,2} = \Delta_{1,2}^2 \simeq -\frac{\Delta_{1,2}^2 + \xi_{1,2}^2 m^2}{1 - \xi_{1,2}} \simeq -\Delta_{1,2}^2, \xi_{1,2} \to 0 \]

\[ s_{1,2} \simeq \xi_{2,1}s, \tau_{1,2} = \sqrt{-t_{1,2}}, \]

\[ M^2 = (\Delta_1 + \Delta_2)^2 \simeq \xi_1 \xi_2 s + t_1 + t_2 - 2\sqrt{t_1 t_2} \cos \phi_0 \]

\[ M^2_\perp = \xi_1 \xi_2 \simeq M^2 + |t_1| + |t_2| + 2\sqrt{t_1 t_2} \cos \phi_0 \]

\[ \cos \phi_0 = \frac{\Delta_1 \Delta_2}{|\Delta_1||\Delta_2|}, \quad 0 \leq \phi_0 \leq \pi \]  

(16)

and additional set for the $2 \to 4$ process

\[ s_{1(a,b)} \simeq m^2(1 - \xi_1)^2 + m_0^2 + \frac{M}{M_\perp} s_1 (1 \pm \tanh \Delta y)(1 - \xi_1), \]

\[ s_{2(a,b)} \simeq m^2(1 - \xi_2)^2 + m_0^2 + \frac{M}{M_\perp} s_2 (1 \mp \tanh \Delta y)(1 - \xi_2), \]

\[ \hat{t}_{a,b} \simeq m_0^2 - \frac{M M_\perp}{2} (1 \pm \tanh \Delta y), \]

\[ m_\perp \simeq \frac{M_\perp}{2 \cosh \Delta y}. \]  

(17)

We write approximate values here for simplicity, but we use exact ones in the generator.

Physical region of diffractive events with two large rapidity gaps is defined by the following kinematical cuts:

\[ 0.01 \text{ GeV}^2 \leq |t_{1,2}| \leq \sim 1 \text{ GeV}^2, \]  

(18)

\[ \xi_{\text{min}} \simeq \frac{M^2}{s \xi_{\text{max}}} \leq \xi_{1,2} \leq \xi_{\text{max}} \sim 0.1, \]  

(19)

\[ (\sqrt{-t_1} - \sqrt{-t_2})^2 \leq \kappa \leq (\sqrt{-t_1} + \sqrt{-t_2})^2 \]  

(20)

\[ \kappa = \xi_1 \xi_2 s - M^2 \ll M^2 \]

We can write the relations in terms of $y_{1,2}$ (rapidities of hadrons) and $y_c$. For instance:

\[ \xi_{1,2} \simeq \frac{M_\perp}{\sqrt{s}} e^{\pm y_c}, \quad |y_c| \leq y_0 = \ln \left( \frac{\sqrt{s} \xi_{\text{max}}}{M} \right), \]

\[ |y_{1,2}| = \frac{1}{2} \ln \left( \frac{1 - \xi_{1,2}}{m^2 - t_{1,2}} \right), \]

\[ |y_c| \leq 6.5, \quad |y_{1,2}| \geq 8.75 \text{ for } \sqrt{s} = 7 \text{ TeV}, \]

\[ |\tanh \Delta y| \leq \sqrt{1 - \frac{4m_0^2}{M^2}}. \]  

(21)
We use on-mass-shell conditions (where ∆ is the generator. Here are additional exact kinematical formulae for processes 2 to 3(4), which we use in 2.2.2 Exact kinematics for CEDP (additions)

\begin{equation}
\frac{d\sigma_{CEDP}^R}{d\Delta^2d\Delta_2^2d\phi_0d\phi_c} \sim \frac{|T_{CEDP}^R|^2}{2^8\pi^4ss'},
\end{equation}

\begin{equation}
\frac{d\sigma_{CEDP}^{ab}}{d\Delta^2d\Delta_2^2d\phi_0d\phi_c dM^2d\Phi_{ab}} \sim \frac{|T_{CEDP}^{ab}|^2}{2^9\pi^5ss'},
\end{equation}

\begin{equation}
d\Phi_{ab} = \frac{d^4k_a}{(2\pi)^2} \delta(k_a^2 - m_0^2) \delta(k_b^2 - m_0^2)
\end{equation}

\begin{equation}
= \frac{d\vec{k}^2}{8\pi M^2\sqrt{1 - \frac{4(k^2 + m_0^2)}{M^2}}} = \frac{d\Delta y}{16\pi \cosh^2 \Delta y}.
\end{equation}

\(T_{CEDP}\) is given by the following analytical expression:

\begin{equation}
T_{CEDP}(p_1, p_2, \Delta_1, \Delta_2) = \int \frac{d^2\vec{q}_T}{(2\pi)^2} \frac{d^2\vec{q}_T'}{(2\pi)^2} V(s, \vec{q}_T) \times C(p_1 - q_T, p_2 + q_T, \Delta_1, \Delta_2) V(s', \vec{q}'_T),
\end{equation}

\begin{equation}
V(s, \vec{q}_T) = \int d^2\vec{b} e^{i\vec{q}_T\cdot\vec{b}} V(s, b), \quad V(s, b) = \sqrt{1 + 2iT_{el}(s, b)}
\end{equation}

where \(\Delta_{1T} = \Delta_1 - q_T - q'_T, \Delta_{2T} = \Delta_2 + q_T + q'_T\), \(C\) is the “bare” amplitude of the process \(p + p \rightarrow p + X + p\).

2.2.2 Exact kinematics for CEDP (additions)

Here are additional exact kinematical formulae for processes 2 to 3(4), which we use in the generator.

Let us make a little bit new definitions for \(\xi_{1,2}\), which are approximately equal to the values of \([14]\): here for any momentum \(p\) we use the notation \(p = (p_0, p_z, \vec{p})\)

\begin{align*}
p_1 &= \left(\frac{\sqrt{s}}{2}, \beta \frac{\sqrt{s}}{2}, 0\right); \quad p_2 = \left(\frac{\sqrt{s}}{2}, -\beta \frac{\sqrt{s}}{2}, 0\right) \\
\Delta_1 &= \left(\xi_1 \frac{\sqrt{s}}{2} + A, \xi_1 \frac{\sqrt{s}}{2} + B, \Delta_1\right); \quad \Delta_2 = \left(\xi_2 \frac{\sqrt{s}}{2} - A, -\xi_2 \frac{\sqrt{s}}{2} - B, \Delta_2\right) \\
p_c = \Delta_1 + \Delta_2 &= \left(M_\perp \text{Cosh} \gamma_c, M_\perp \text{Sinh} \gamma_c, \Delta_1 + \Delta_2\right); \quad \xi_{1,2} = \frac{M_\perp}{\sqrt{s}} e^{\pm \gamma_c}.
\end{align*}

We use on-mass-shell conditions \((p_i - \Delta_i)^2 = m_p^2\) and have two equations for \(A\) and \(B\):

\begin{align*}
\xi_1 \sqrt{s} (A - B) + A^2 - B^2 - \Delta_1^2 - \sqrt{s} \left( A - \beta B + \xi_1 \frac{\sqrt{s}}{2} (1 - \beta) \right) &= 0, \quad (27) \\
-\xi_2 \sqrt{s} (A + B) + A^2 - B^2 - \Delta_2^2 + \sqrt{s} \left( A + \beta B - \xi_2 \frac{\sqrt{s}}{2} (1 - \beta) \right) &= 0. \quad (28)
\end{align*}
By adding and subtracting (27) and (28) we have:

\[ A = X B + Y, \quad X = \frac{\xi_2 - \xi_1}{2 - \xi_2 - \xi_1}, \quad Y = \frac{\bar{\Delta}_2^2 - \bar{\Delta}_1^2}{\sqrt{s}(2 - \xi_1 - \xi_2)} + X \frac{\sqrt{s}}{2}(1 - \beta), \]

\[ |X| \ll 1, \quad Y \sim \frac{m_p^2}{\sqrt{s}}, \]

\[ 2(1 - X^2)B^2 - U B + V = 0, \quad B = \frac{U \pm \sqrt{U^2 - 8(1 - X^2)V}}{4(1 - X^2)}, \quad (29) \]

\[ U = 4XY + \sqrt{s}(2\beta - \xi_1 - \xi_2 + X(\xi_1 - \xi_2)), \]

\[ V = \bar{\Delta}_1^2 + \bar{\Delta}_2^2 + \frac{(\xi_1 + \xi_2)\sqrt{s}}{2}(1 - \beta) + (\xi_2 - \xi_1)\sqrt{s}Y - 2Y^2 \quad (30) \]

We have to choose the root with minus sign in (29), since for the plus sign we obtain \( B > \sqrt{s} \). And for the minus sign we have \( A, B \sim m_p^2/\sqrt{s} \ll 1 \).

In the case of di-hadron CEDP it is convenient to use variables \( y_{a,b}, \bar{k}_m = \bar{k}_a - \bar{k}_b \).

Cross-section looks as

\[ d\sigma_{\pi p \to \pi h_a h_b} = \frac{|T|^2}{2\sqrt{s}(s - 4m_p^2)} \frac{1}{2^{12}\pi^8} \frac{|\bar{\Delta}_1|d|\bar{\Delta}_1||\bar{\Delta}_2|d|\bar{\Delta}_2|d\phi_1 d\phi_2}{|p_{1z}'E_{2z}' - p_{2z}'E_{2z}'|} dy_a dy_b \frac{1}{4} d^2\bar{k}_m, \quad (31) \]

where \( T \) is the amplitude of the di-hadron CEDP, and \( p_{1z}' \) is the appropriate root of the system of equations

\[ u = \sqrt{s} - m_{\perp a} \text{Cosh}(y_a) - m_{\perp b} \text{Cosh}(y_b) = \sqrt{m_{1a}^2 + p_{1z}'^2} + \sqrt{m_{1b}^2 + p_{2z}'^2}, \]

\[ v = -m_{\perp a} \text{Sinh}(y_a) - m_{\perp b} \text{Sinh}(y_b) = p_{1z}' + p_{2z}', \quad (32) \]

\[ |p_{1z}'E_{2z}' - p_{2z}'E_{2z}'| = \frac{1}{2} \sqrt{(u^2 - v^2 - (m_{\perp 1} - m_{\perp 2})^2)(u^2 - v^2 - (m_{\perp 1} + m_{\perp 2})^2)}, \]

\[ p_{1z}' = \frac{v(u^2 - v^2 + m_{\perp 1}^2 + m_{\perp 2}^2) \pm u\sqrt{(u^2 - v^2 - (m_{\perp 1} - m_{\perp 2})^2)(u^2 - v^2 - (m_{\perp 1} + m_{\perp 2})^2)}}{2(u^2 - v^2)}, \]

\[ m_{\perp 1}^2 = m_{\perp a}^2 + \bar{\Delta}_1^2, \quad m_{\perp a,b}^2 = m_0^2 + \bar{k}_{a,b}^2 \quad (33) \]

And we take the root \( p_{1z}' \) with plus sign, since another one gives backward scattering and its contribution to the amplitude is negligible.

### 2.2.3 Model for CEDP

In the case of the eikonal representation of the elastic amplitude \( T_{el} \) we have

\[ V(s, \bar{q}_T) = \int d^2\bar{\theta} e^{i\bar{q}_T \bar{b}} e^{i\delta_{el}(s,b)}, \quad (34) \]

In this case amplitude (24) can be rewritten as

\[ T^{\text{CEDP}}(\bar{\Delta}_1, \bar{\Delta}_2) = \int d^2\bar{\theta} e^{-i\bar{b}^{-\Omega}(s,b) - \Omega(s',b)} \times \int d^2\bar{k} e^{i\bar{b}} C(\bar{\Delta} - \bar{k}, \bar{\Delta} + \bar{k}), \]

\[ \Omega(s, b) = -i\delta_{el}(s,b), \]

\[ \bar{\Delta} = \bar{\Delta}_2 + \bar{\Delta}_1 = \frac{\bar{\Delta}_2 - \bar{\Delta}_1}{2}, \quad \bar{\delta} = \bar{\Delta}_2 - \bar{\Delta}_1, \quad \bar{k} = \bar{\delta} + \bar{q}_T + \bar{q}_T'. \quad (35) \]
Calculations for concrete expressions of $C$ in the general case were considered in \cite{2,5}. Here we present the model for resonances in CEDP based on the schemes for single diffractive dissociation \cite{6} and tensor resonance decays \cite{7}. Absorptive corrections are calculated according to \cite{6}.

\[
C(t_1, t_2, \xi_1, \xi_2) = \pi^2 g_{ppP}(0, 0) \left\{ \prod_{n=1}^{2} g_{ppP}(t_n) \alpha'_{P}(t_n) \left( i + \tan \frac{\pi(\alpha_{P}(t_n) - 1)}{2} \right) \left( \frac{1}{\xi_n} \right)^{\alpha_{P}(t_n)} \right\} \eta_j
\]

\[
\eta_0 = 1, \quad \eta_1 = \Delta_1^\mu e_\mu^{(\lambda)}(\Delta_1 + \Delta_2), \quad \eta_2 = \Delta_1^\mu \Delta_1^\nu e_{\mu\nu}^{(\lambda)}(\Delta_1 + \Delta_2)
\]

(36)

where $J$ is the spin of a central particle, and $e^{(\lambda)}(\Delta_1 + \Delta_2)$ are states with the helicity $\lambda$. 
3 Program Overview

3.1 General information

ExDiff is written in a modular way using C++. The general structure of the generator is shown on the Fig. 2. In this subsection there is an outlook. You can find more detailed description of all the elements in next subsections.

![Figure 2: General structure of the generator. Classes, subroutines and files.](image)

#### 3.1.1 Data card

The first element is the data card. Let us consider an example

```bash
../ExDiff2.0/config/card/*.card:
1 0 0 1 1000 0 0
```

```
IDauthors IDprocess IDenergy version Nevents IDinput_format IDoutput_format
```

Only the first line with int type of numbers is used. Sequence of parameters of the first string with possible values for ExDiff 2.0:

- **IDauthors** defines authors of the model used for a process:
  - **IDauthors=1**: A.A. Godizov
  - **IDauthors=2**: R.A. Ryutin (some test distributions of the di-pion CEDP)
  - **IDauthors=3**: V.A. Petrov and R.A. Ryutin (in the future versions).

- **IDprocess** defines the process:
  - **IDprocess=0**: elastic $p + p \rightarrow p + p$ scattering;
IDprocess=1: CEDP $p + p \rightarrow p + f_0(1710) + p$ at 8, 13 TeV;
IDprocess=2: CEDP $p + p \rightarrow p + f_2(1950)$ (sum of all helicities) + $p$ at 8, 13 TeV;
IDprocess=20: CEDP $p + p \rightarrow p + f_0(1710)$ (helicity 0) + $p$ at 8, 13 TeV;
IDprocess=21: CEDP $p + p \rightarrow p + f_2(1950)$ (helicity $\pm 1$) + $p$ at 8, 13 TeV;
IDprocess=22: CEDP $p + p \rightarrow p + f_2(1950)$ (helicity $\pm 2$) + $p$ at 8, 13 TeV;
IDprocess=3: CEDP $p + p \rightarrow p + \pi^+ \pi^- + p$ at 0.2, 1.96, 7, 8, 13 TeV (test Born distributions);
IDprocess=4: CEDP $p + p \rightarrow p + f_0(1500) + p$ at 8, 13 TeV;
IDprocess=5: CEDP $p + p \rightarrow p + f_2(1270)$ (sum of all helicities) + $p$ at 8 TeV;
IDprocess=50: CEDP $p + p \rightarrow p + f_2(1270)$ (helicity 0) + $p$ at 8 TeV;
IDprocess=51: CEDP $p + p \rightarrow p + f_2(1270)$ (helicity $\pm 1$) + $p$ at 8 TeV;
IDprocess=52: CEDP $p + p \rightarrow p + f_2(1270)$ (helicity $\pm 2$) + $p$ at 8 TeV;
IDprocess=6: CEDP $p + p \rightarrow p + f_2(2220)$ (sum of all helicities) + $p$ at 13 TeV;
IDprocess=60: CEDP $p + p \rightarrow p + f_2(2220)$ (helicity 0) + $p$ at 13 TeV;
IDprocess=61: CEDP $p + p \rightarrow p + f_2(2220)$ (helicity $\pm 1$) + $p$ at 13 TeV;
IDprocess=62: CEDP $p + p \rightarrow p + f_2(2220)$ (helicity $\pm 2$) + $p$ at 13 TeV;

- **IDenergy** defines the collision energy of the process:
  IDenergy=0: 13 TeV;
  IDenergy=1: 14 TeV (reserved for future versions);
  IDenergy=2: 7 TeV;
  IDenergy=3: 8 TeV;
  IDenergy=4: 14 TeV;
  IDenergy=5: 1.96 TeV; (only for process 3)
  IDenergy=6: 0.2 TeV; (only for process 3)

- **version** is auxiliary parameter that is used to define different versions. Default value is 1. For processes 2, 20, 21, 22, 5, 50, 51, 52 version should be set to 2.

- **Nevents** defines number of events to generate.

- **IDinput_format** defines a set of variables for kinematics. Default value is 0. It means that we use the following sets of variables: \{t, \phi\} (elastic scattering); \{\tau_1, \tau_2, \phi_0, \ln \xi, \phi_1\} (CEDP resonance production); \{\tau_1, \tau_2, \phi_0, M_e, y_c; \Delta y, \phi^*, \phi_1\} (CEDP di-jet, di-hadron, di-boson production), \tau_{1,2} = |\Delta_{1,2}|. It is possible to use any set of kinematical variables by adding new information to the ExDiff::KinCM class.

- **IDoutput_format** defines output format. Default value is 0. In ExDiff v1.0 the simple file ./ExDiff2.0/output/A1M1E0_1F0.exdiff generated, where the output looks like

```exdiff
============================== ExDiff Event ===============================
------------------------------------------------------------------------------
| ID  | E      | px      | py      | pz      |
|-----|--------|---------|---------|---------|
| 2212| 6.50000000e+03 | 0.00000000e+00 | 0.00000000e+00 | 6.49999993e+03 |
| 2212| 6.50000000e+03 | 0.00000000e+00 | 0.00000000e+00 | -6.49999993e+03 |
|-----|--------|---------|---------|---------|
| 2212| 6.49922495e+03 | 1.20107011e-01 | 1.07105789e-01 | 6.49922488e+03 |
| 10331| 1.72158400e+00 | -3.95672633e-02 | -9.27803020e-02 | -1.71445119e-01 |
| 2212| 6.49905351e+03 | -8.05397481e-02 | -1.43254592e-02 | -6.49905344e+03 |
```
| ID  | E         | px         | py         | pz         |
|-----|-----------|------------|------------|------------|
| 2212| 6.50000000e+03 | 0.00000000e+00 | 0.00000000e+00 | 6.49999993e+03 |
| 2212| 6.50000000e+03 | 0.00000000e+00 | 0.00000000e+00 | -6.49999993e+03 |
| 2212| 6.49922495e+03 | 1.20107011e-01 | 1.07105789e-01 | 6.49922488e+03 |
| 10331 | 1.72153048e+00 | -3.95672633e-02 | -9.27803302e-02 | -1.71445119e-01 |
| 2212| 6.49905351e+03 | -8.05397481e-02 | -1.43254592e-02 | -6.49905344e+03 |

where the first column contains particles IDs (standard PDG numbering scheme [8]), and other four columns contain their four-momentum. Output filename is constructed from parameters in the data card:

```
A<IDauthor>M<IDprocess>E<IDenergy>_<version>F<IDinput_format>.exdiff
```

In the present version you can set output format to 1 (PYTHIA output), 2 (PYTHIA + ROOT Tree output), 3 (PYTHIA + HEPMC2 output), 4 (PYTHIA + HEPMC3 output). You can see details in the section 4.

### 3.1.2 Block of models

This block contains data files (simple txt files) obtained somehow from external calculations. Naming of these files is the same as for *.exdiff.*

The file `../ExDiff2.0/modell data/*.grid` contains one column with values of kinematical variables that author use to parametrize differential cross section of the process under consideration. For each variable $v_i$ we have the interval divided by $N_v_i$ parts. The file looks like

1rst value of $v_1$
2nd value of $v_1$
...
$(N_v_1+1)$th value of $v_1$
1rst value of $v_2$
...
$(N_v_2+1)$th value of $v_2$
1rst value of $v_3$
...
$(N_v_3+1)$th value of $v_3$
...
$(N_v_n+1)$th value of $v_n$

Values are of the double type. $n$ is the number of variables in the set.

By default sets of variables for different kinematics are

- 2 → 2 process: \{t, \phi\};
- 2 → 3 process: \{\tau_1, \tau_2, \phi_0, \ln \xi_1, \phi_1\};
- 2 → 4 process: \{\tau_1, \tau_2, \phi_0, M_c, V_y, V_{\Delta y}, \phi^*, \phi_1\}, where $V_y = y / \ln(0.1 \sqrt{s} / M_\perp)$, $V_{\Delta y} = \Delta y / \text{ArcCosh}(M / (2m^*))$, where $m^*$ is the mass of the central hadron.

11
\(\phi, \phi_1\) variables are uniformly distributed in the interval \(0 \to 2\pi\), that is why their values are simply generated as \(2\pi \times \text{(random number in the } [0, 1] \text{ interval)}\). So, finally we have one significant variable for \(2 \to 2\) process, four variables for \(2 \to 3\) process and seven variables for \(2 \to 4\) process.

The file `../ExDiff2.0/modeldata/*.dat` contains one column with values of the cross section of the process. This file is loaded to the multidimensional array which corresponds to the class `ExDiff::NDarray` (see below).

### 3.2 Basic classes and functions

Here we describe only basic classes and some of their methods and variables which can be used by developers.

**ExDiff::NDarray**

**Purpose:** to initialize multidimensional dynamical array of any type or class.

**Initialization example:**

```cpp
... 
std::vector<std::size_t> dim = {129, 129, 9, 9}; 
ExDiff::NDarray<double> dat(dim); 
...
```

Here we have vector `dim` which defines number of node points for each variable. Indexes are \(0 \to 128\) for the first variable, the same for the second one and so on.

**Methods:**

- `T& get(const std::vector<std::size_t>& indexes)`
  gets the value of the cross section (\(T\) is `double` in examples) in the node point defined by the vector of indexes of variables. **Example:**

  ```cpp
  std::vector<std::size_t> indexes = {1, 2, 3, 4}; 
  double cs = dat.get(indexes); 
  ```

- `std::vector<std::size_t>& GetDimensions()`
  gets the value of the private `dimensions` vector, which is defined by the vector `dim` in the initialization example above. **Example:**

  ```cpp
  std::vector<std::size_t> gdim = dat.GetDimensions(); 
  ```

- `std::vector<T>& GetValues()`
  gets the value of the private `values` vector of any type or class \(T\), which is filled by numbers loaded from `*.dat` file or calculated in some other way. **Example:**

  ```cpp
  std::vector<double> gvalues = dat.GetValues(); 
  ```

- `std::size_t computeTotalSize(const std::vector<std::size_t>& dimensions)`
  calculates the total size of the array by multiplication of all dimensions.

- `std::size_t computeIndex(const std::vector<std::size_t>& indexes)`
calculates the global index of the array from the input node point \textit{indexes} to read the value of the cross section from the vector \textit{values}, i.e. converts vector of indexes to the single index. \textbf{Example:} (used by \textit{get})

\begin{verbatim}
return values[computeIndex(indexes)];
\end{verbatim}

\texttt{std::vector<std::size_t> computeIndexes(std::size_t index)}

converts back the single index to the vector of indexes of the node point.

\texttt{printStats()}

prints the general information about the class object.

\texttt{LoadFromFile(std::string filename)}

loads data from a *.dat file after the initialization.

\textbf{Variables:}

\begin{verbatim}
std::vector<std::size_t> dimensions (\textit{input}) : private vector, which contains dimensions of the array (numbers of node points for each variable).

std::vector<T> values (\textit{input}) : private vector, which contains values of any class or type \textit{T} loaded from a file or calculated in some other way.
\end{verbatim}

\textbf{Used by:} iFunc, Interface.

\textbf{Includes:} \texttt{<cmath> <complex> <cstdlib> <iostream> <fstream> <map> <string>}

\texttt{<utility> <vector> <cassert>}

\texttt{ExDiff::NDgrid}

\textbf{Purpose:} to initialize special array which contains node points for each variable in the set (from *.grid file or by the use of some calculations).

\textbf{Initialization example:}

\begin{verbatim}
...;
std::vector<std::size_t> dim = \{129,129,9,9\};
ExDiff::NDgrid<double> grid(dim);
...
\end{verbatim}

Here we have vector \texttt{dim} which defines numbers of node points for each variable. Indexes are 0 \rightarrow 128 for the first variable, the same for the second one and so on.

\textbf{Methods:}

\texttt{T& GetVar(const std::size_t& varindex, const std::size_t& index)}

gets the value of a variable (type \textit{T} is double as in examples above) with the number \textit{varindex} (in the set of variables) in the node point with the number \textit{index}. Use this method only to change node points of the grid inside an object of the class.

\begin{verbatim}
std::vector<size_t>& GetDimensions()
\end{verbatim}
gets the value of the private dimensions vector, which is defined by the vector dim in the initialization example above. **Example:**

```cpp
std::vector<std::size_t> gdim = grid.GetDimensions();
```

```cpp
std::vector<T>& GetValues()
```
gets the value of the private values vector of any type or class T, which is filled by numbers loaded from *.grid file or calculated in some other way. **Example:**

```cpp
std::vector<double> gvalues = grid.GetValues();
```

```cpp
std::vector<T>& get(const std::vector<std::size_t>& _indexes)
```
gets corresponding vector node point (values of each variable are of the type or class T).

```cpp
std::size_t computeIndex(const std::size_t& varindex, const std::size_t& index)
```
computes the global index of the concrete variable (with the number varindex in the set) node point (with the node number index for this variable) in the NDgrid vector.

```cpp
std::size_t computeTotalSize(const std::vector<std::size_t>& _dimensions)
```
calculates the total size of the array by summation of all dimensions.

```cpp
printC()
```
prints the general information about the class object.

```cpp
LoadFromFile(std::string filename)
```
loads data from a *.grid file after the initialization.

**Variables:**

```cpp
std::vector<std::size_t> dimensions (input) : private vector, which contains dimensions of the array (numbers of node points for each variable).
```

```cpp
std::vector<T> values (input) : private vector, which contains values of any class or type T loaded from a file or calculated in some other way.
```

**Used by:** iFunc, Interface.

**Includes:** `<cmath` `<complex` `<cstdlib` `<iostream` `<fstream` `<map` `<string` `<utility` `<vector` `<cassert`

**ExDiff::iFunc**

**Purpose:** to initialize special class which contains a multidimensional interpolation function. Node points and values of the function are taken from *.grid (or by the use of some calculations) and *.dat files correspondingly.

**Initialization example:**
```
...  
std::vector<std::size_t> dim;
```
dim = {129,129,9,9};
...
ExDiff::NDarray<double> dat_.(dim);
ExDiff::NDgrid<double> grid_.(dim);
...
// load *.dat and *.grid
ExDiff::iFunc fungen(dat_,grid_);
...

Methods:

double Calc(std::vector<double>)
calculates the value of the interpolation function in the point defined by vector
of variables. Example:

... std::vector<double> p = {1.2345,2.3456,3.4567,4.5678}; double value = fungen.Calc(p);
...

int CheckPoint(std::vector<double>& _point)
private method which returns 0 if vector _point lies inside the interpolation
range and 1 if it is outside the interpolation range.

NDgrid<double>& GetGrid()
returns an object of the private class iGrid, which contains data from *.grid
file.

NDarray<double>& GetTab()
returns an object of the private class iTab, which contains data from *.dat
file.

std::vector<std::size_t>& GetDimensions()
returns the vector of dimensions which is equal to dim in the initialization
example.

std::vector<double>& GetPoint()
returns the private vector point which contains the last point of calculations.

std::vector<double>& GetMinFunPoint()
returns private vector minfunpoint where the function is minimal.

std::vector<double>& GetMaxFunPoint()
returns private vector maxfunpoint where the function is maximal.

double& GetMinFun()
returns private variable minfunvalue which is equal to the minimal value of
the function.

double& GetMaxFun()
returns private variable maxfunvalue which is equal to the maximal value of
the function.
void PrintVector(std::vector<size_t>, std::size_t)
void PrintVector(std::vector<double>, std::size_t)
    prints size_t or double vector

void printC()
    prints the general information about the class object.

Variables:

std::vector<std::size_t> dimensions : private vector, which contains dimensions of the array (numbers of node points for each variable).

std::vector<double> minvars : private vector, which contains minimal values of variables in the interpolation range.

std::vector<double> maxvars : private vector, which contains maximal values of variables in the interpolation range.

std::vector<double> minfunpoint : private vector, which contains point where the function is minimal.

std::vector<double> maxfunpoint : private vector, which contains point where the function is maximal.

double minfunvalue : private variable, which contains the minimal value of the function.

double maxfunvalue : private variable, which contains the maximal value of the function.

Used by: Generator, Interface.
Includes: <cmath> <complex> <cstdlib> <iostream> <fstream> <map> <string> <utility> <vector> <cassert> "NDarray.h" "NDgrid.h" "I.h" "PI.h"

ExDiff::Generator

Purpose: to initialize special class which is applied to a multidimensional interpolation function, and generates a set of variables according to this function. iFunc class is used to create the input object.

Initialization example:

...  
ExDiff::Generator gen(fungen);
ExDiff::newrand();
...

fungen is taken from the initialization example of the class ExDiff::iFunc. ExDiff::newrand() sets the random initial condition for new generation by the use of a system clock.

Methods:
double RNUM()
    returns random number between 0 and 1.

void printC()
    prints the general information about the object of the class.

ExDiff::iFunc CalcMinus(const ExDiff::iFunc&)
    calculates new iFunc object of \( N - 1 \) variables by resummation in the last variable from the vector of variables, where \( N \) is the number of variables of the input object.

std::vector<ExDiff::iFunc> CalcFUNG(const ExDiff::iFunc&)
    calculates a full set of interpolation functions of lower dimensions, including also the total integral of the input interpolation function, and outputs results to private vector FUNG and the variable FUNGTOT.

double& GetFUNGTOT()
    gets the total cross-section from the private variable FUNGTOT.

double GenVar(const std::vector<double>& x, const std::vector<double>& f, const double& R)
    generates a variable from one-dimensional interpolation function, represented by the std::vector array f. x contains node points, f contains values of the function in these node points, R is the input random number.

double EGenVarFast1(const std::vector<double>& x, const std::vector<double>& f, const std::vector<double>& F, const double& R)
    generates a variable from one-dimensional interpolation function, represented by the std::vector array f. x contains node points, f contains values of the function in these node points, F contains integrals of the function, R is the input random number.

double IntVec(const std::vector<double>& x, const std::vector<double>& f)
    calculates the total integral of one-dimensional interpolation function. x contains node points, f contains values of the function in these node points.

std::vector<double> Generate()
    generates the output set (std::vector) of variables.

Variables:

ExDiff::iFunc FUN : private variable which contains an input object of class iFunc (initial multidimensional interpolation function).

std::vector<ExDiff::iFunc> FUNG : private vector which contains an input object of class iFunc (initial multidimensional interpolation function) and all the functions of lower dimensions.

double FUNGTOT : private variable which contains the total cross-section.
Used by: Interface.
Includes: `<cmath> <complex> <cstdlib> <iostream> <fstream> <map> <string>
<utility> <vector> <cassert> <time.h> <ctime>
"NDarray.h" "NDgrid.h" "iFunc.h" "newrand.h" "I.h" "PI.h"

ExDiff::Constants

Purpose: to initialize special class which contains fundamental constants: couplings, masses, spins and IDs of particles.

Initialization example:

```cpp
... 
ExDiff::Constants c; 
...
```

Methods:

void Print()

prints all the constants.

double _mass(int ID_)

returns the mass of a particle by the use of its ID number.

int _dspin(int ID_)

returns 2s + 1, where s is the particle spin.

Variables:

const double m_p, m_n, m_pi0, m_pi, m_H, m_Gra, m_R, m_Glu, m_Z, m_W : public variables which contain masses of proton, neutron, π₀, π⁺, Higgs boson, graviton, glueball, Z and W bosons.

const double alpha_EM, Lam_QCD : public variables which contain electromagnetic αₑ and QCD αₛ couplings.

const double VEV, M_Pl : public variables which contain vacuum expectation value 246 GeV and the Plank mass.

const int IDproton, ID1710, ID1950, ID1500, ID1270, ID2220, IDpi0, IDpiplus, IDpiminus, IDdef : public variables which contain PDG (or PYTHIA's) IDs of particles.

Used by: Interface, Event, KinCM.
Includes: `<cmath> <complex> <cstdlib> <iostream> <fstream> <map> <string>
<utility> <vector> <cassert>
"I.h" "PI.h"

ExDiff::Particle

Purpose: to initialize special class which contains four-momentum, mass, spin, color, ID of a relativistic particle.
Initialization example:

```cpp
ExDiff::Constants c;
std::vector<double> v = {c.m_p, 0.0, 0.0, 0.0};
ExDiff::Particle pa(v, c.m_p, 2212, 2, 0, 0);
```

Methods:

```cpp
void Print()
    prints all the information about the particle.

double Rapidity(std::vector<double>& p_)
    calculates particle rapidity. p_ is the input four-momentum.

double Pseudorapidity(std::vector<double>& p_)
    calculates particle pseudorapidity. p_ is the input four-momentum.

double Theta(std::vector<double>& p_)
    calculates particle polar angle. p_ is the input four-momentum.

double Phi(std::vector<double>& p_)
    calculates particle azimuthal angle. p_ is the input four-momentum.

double Lmult(std::vector<double>& p1_, std::vector<double>& p2_)
    returns scalar product of p1_ and p2_ four momenta.

std::vector<double> Lminus(std::vector<double>& p1_, std::vector<double>& p2_)
    returns four momenta which is equal to p1_ minus p2_.

std::vector<double> Lplus(std::vector<double>& p1_, std::vector<double>& p2_)
    returns four momenta which is equal to p1_ plus p2_.

void Lprint(std::vector<double>& vec_)
    prints the four-momentum vec_.

int OnShell()
    returns on-shell status of the particle:
    0 : particle is on-shell, \( p^2 = m^2 \)
    1 : \( 0 < p^2 < m^2 \)
    -1 : \( -m^2 < p^2 < 0 \)
    2 : \( p^2 > m^2 \)
    -2 : \( p^2 < -m^2 \)
    3 : \( m = 0 \)
    4 : \( m < 0 \) (error value)

void Reset(const std::vector<double>& p_, const double& m_,
    const int& ID_, const int& DSpin_,
    const int& Colour_, const int& Anticolour_)
    resets parameters of a particle (all the private variables).

void ResetP(const std::vector<double>& p_)
```
resets only four-momentum of the particle.

\[
\text{const std::vector<double>& } _p() \\
\text{returns private variable which contains four-momentum of the particle.}
\]

double& _px() \\
double& _py() \\
double& _pz() \\
double& _E() \\
double& _m() \\
double& _pp() \\
double& _pT() \\
double& _p3D() \\
int& _DSpin() \\
int& _ID() \\
int& _Colour() \\
int& _Anticolour() \\
\text{return private variables which contain } p_x, p_y, p_z, E, p^2, p_T = \sqrt{p_x^2 + p_y^2}, p_{3D} = \sqrt{p_x^2 + p_y^2 + p_z^2}, 2s + 1 (s \text{ is the spin}), \text{ID number, colour and anticolour of the particle.}

\text{Variables:}

\[
\text{std::vector<double> } p : \text{ private vector which contains four-momentum of the particle } E, p_x, p_y, p_z.
\]

\[
double E, \text{ px, py, pz, m, pp, pT, p3D} : \text{private variables which contain } E, p_x, p_y, p_z, m, p^2, p_T = \sqrt{p_x^2 + p_y^2}, p_{3D} = \sqrt{p_x^2 + p_y^2 + p_z^2}.
\]

\[
\text{int ID, DSpin, Colour, Anticolour} : \text{private variables which contain ID, 2s + 1, colour and anticolour of the particle.}
\]

\text{Used by:} \text{Interface, Event, KinCM.}

\text{Includes:} \text{<cmath> <complex> <cstdlib> <iostream> <fstream> <map> <string> <utility> <vector> <cassert>}

"I.h" "PI.h"

\begin{verbatim}
ExDiff::Event
\end{verbatim}

\text{Purpose:} \text{to initialize special class which contains all the particles of an event.}

\text{Initialization example:}

\begin{verbatim}
... \\
ExDiff::Event ev(particles); \\
...
\end{verbatim}

where \text{particles} is of the \text{std::vector<Particle>} type.

\text{Methods:}

\text{void Print(int format)}

\text{prints parameters of particles in different form.}

\text{format=0 : all the parameters of every particle.}
format=1: mass, on-shell status and four-momentum of every particle.
format=2: four-momenta of particles.
format=3: four-momenta of final particles.

```
std::vector<Particle>& _particles()
    returns private vector particles.

void AddToFile(int format, std::string filename)
    add an event to file *.exdiff in special format.
    format=0: intrinsic ExDiff format.

std::vector<int> ExDiff::Event::TakeFromFile(int number, int format, std::string filename)
    loads an event number number from the file filename in special format.
    format=0: intrinsic ExDiff format.
    returns int vector, which contains numbers of
    [0]: particles in the event,
    [1]: events in the file,
    [2]: strings in the file.

void ExDiff::Event::TakeEvent(std::vector<int> Nfile, int number, int format, std::string filename)
    loads an event number number from the file filename in special format.
    format=0: intrinsic ExDiff format.
    The file is the same as in the previous method. Vector Nfile is obtained
    by the use of TakeFromFile:
    ...
    std::vector<int> Nfile = ev.TakeFromFile(ev_number,format,filename);
    ev.TakeEvent(Nfile,ev_number,format,filename);

int CheckSum()
    returns 0, if momentum conservation law is carried out, 1 in other cases.
```

Variables:

```
std::vector<ExDiff::Particle> particles: private vector which contains all the particles.
```

Used by: Interface.

Includes:
```
<cmath> <complex> <cstdlib> <iostream> <fstream> <map> <string>
<utility> <vector> <cassert> <iomanip> <limits>
"I.h" "PI.h" "Particle.h" "Constants.h"
```

```
ExDiff::KinCM
```

Purpose: to initialize special class which contains all kinematical variables for different processes.

Initialization example:
```
...  
ExDiff::KinCM kinematics(sqs_,0,genvec,masses_,ids_);
...  
```
where `sqs_` is the central mass frame collision energy; 0 defines the set of variables, `genvec` contains generated variables, `masses_` and `ids_` contain all the final(!) particles masses and IDs. Other way to initialize an object of the class is

```cpp
ExDiff::KinCM kinematics(sqs_,particles_);
```

where `particles_` is the vector of type `ExDiff::Particle` which contains all the particles of a process.

**Methods:**

`void Print()`

prints all the information on kinematical variables of the process.

`std::vector<Particle> EvParticles()`

returns the vector of all the particles including initial protons (to use it in the `ExDiff::Event` class objects).

`void ResetVars(const std::vector<double>& variables_)`

sets new values only to kinematical variables of the process without changing masses, ids, spins and so on. Also it verifies conservation laws.

`int Phys()`

checks out physical region conditions for the process (conservation laws, positivity of particles energies, all particles are on-shell) and returns

- 0: everything is correct
- 1: some of energies are negative
- 2: some particles are off-shell
- >2: other errors

`std::vector<double> Transform_CMpp_to_CMdd(std::vector<double>& k_)`

makes transformation of the Lorentz vector `k_` calculated in the C.M. frame of two colliding protons to the frame where \( \Delta_1 + \Delta_2 = 0 \) and \( \Delta_{10} + \Delta_{20} = \sqrt{(\Delta_1^2 + \Delta_2^2)} \) (the rest frame of a central resonance or system of particles). All the vectors are defined in \([14]\).  

`std::vector<double> Transform_CMdd_to_CMpp(std::vector<double>& k_)`

makes transformation of the Lorentz vector `k_` calculated in the reference frame where \( \Delta_1 + \Delta_2 = 0 \) and \( \Delta_{10} + \Delta_{20} = \sqrt{(\Delta_1^2 + \Delta_2^2)} \) (the rest frame of a central resonance or system of particles) to the C.M. frame of two colliding protons. All the vectors are defined in \([14]\).  

`std::vector<double> Transform_CMpp_to_CMdd(std::vector<double>& k_, std::vector<double>& Delta_1_, std::vector<double>& Delta_2_)`

makes transformation of the Lorentz vector `k_` calculated in the C.M. frame of two colliding protons to the frame where \( \Delta_1 + \Delta_2 = 0 \) and \( \Delta_{10} + \Delta_{20} = \sqrt{(\Delta_1^2 + \Delta_2^2)} \) (the rest frame of a central resonance or system of particles). `Delta_1_` and `Delta_2_` contain vectors \( \Delta_{1,2} \) calculated in the C.M. frame of two colliding protons. All the vectors are defined in \([14]\).  

`std::vector<double> Transform_CMdd_to_CMpp(std::vector<double>& k_)`
std::vector<double>& Delta_1, std::vector<double>& Delta_2)
  makes transformation of the Lorentz vector \( \mathbf{k} \) calculated in the reference frame
where \( \Delta_1 + \Delta_2 = 0 \) and \( \Delta_{10} + \Delta_{20} = \sqrt{(\Delta_1 + \Delta_2)^2} \) (the rest frame of a central resonance or system of particles) to the C.M. frame of two colliding protons. Delta_1 and Delta_2 contain vectors \( \Delta_{1,2} \) calculated in the C.M. frame of two colliding protons. All the vectors are defined in [14].

int Cuts(int& icut_)
  checks out cuts on different variables (it is used in the data analysis of different collaborations like STAR, CDF, ALICE, CMS etc.) and returns
  0 : cuts are satisfied
  1 : cuts are not satisfied
  icut_ defines the specific cut (see file Kinematics.cpp).

void DeltasInit()
  sets exact \( \Delta_{1,2} \) vectors (auxiliary function).

void VtoP2(std::size_t& ID, std::vector<double>& _masses,
  std::vector<int>& _IDs)
  sets all the final particles momenta from the set of variables for the process
  \( 2 \rightarrow 2 \). _masses and _IDs are vectors of masses and ids of two final particles.
  _ID identifies the set of kinematical variables for the process:
  _ID=0 : \(|t|, \phi\);
  _ID=1 : \( \theta \) (polar scattering angle of the final hadron), \( \phi \) (azimuthal angle of the final hadron);
  _ID=2 : \( y_h \) (rapidity of the final hadron), \( \phi \);
  _ID=3 : \(|\vec{\Delta}|\) (transverse momentum of the final hadron), \( \phi \);

void VtoP3(std::size_t& ID, std::vector<double>& _masses,
  std::vector<int>& _IDs)
  similar to previous one, but it sets all the final particles momenta from the set of variables for the process 2 \( \rightarrow 3 \). _ID identifies the set of kinematical variables for the process:
  _ID=0 : \(|\vec{\Delta}_1 T|, |\vec{\Delta}_2 T|, \phi_0 \) (azimuthal angle between final protons), \( ln \xi_1, \phi_1 \)
  (azimuthal angle of the first final hadron);
  _ID=1 : \(|\vec{\Delta}_1 T|, |\vec{\Delta}_2 T|, \phi_0, y_c \) (rapidity of a central particle), \( \phi_1 \);

void VtoP4(std::size_t& ID, std::vector<double>& _masses,
  std::vector<int>& _IDs)
  similar to previous one, but it sets all the final particles momenta from the set of variables for the process 2 \( \rightarrow 4 \). _ID identifies the set of kinematical variables for the process:
  _ID=0 : \(|\vec{\Delta}_1 T|, |\vec{\Delta}_2 T|, \phi_0, M \) (mass of a central system), \( V_{\eta_1} = y_c/\ln(\sqrt{s}\xi_{max}/M_\perp) \)
  \( y_c \) is the rapidity of the central system), \( V_{\eta_j} = \eta_j/ArcCosh(M/(2m^2)) \)
  \( \eta_j \) is the rapidity of the first particle of the central system in its rest frame, or, in other words, in the C.M. frame of two central particles. For example, consider \( \pi^+\pi^- \) central production. In this case \( \eta_c \equiv \eta_{\pi^+} \)
  or \( \eta_j \equiv \eta_{\pi^-} \) in the \( \pi^+\pi^- \) C.M. frame. It is equal to \( \Delta y \) in [15], \( \phi_j \)
  (azimuthal angle of the particle, which we choose to fix \( \eta_j \), in the same reference frame), \( \phi_1 \) (azimuthal angle of the first final proton, usually has uniform distribution);
  _ID=1 : \(|\vec{\Delta}_1 T|, |\vec{\Delta}_2 T|, \phi_0, \phi_1 \) (azimuthal angle of the first final proton, usually has
uniform distribution), \( M \) (mass of a central system), \( y_c \) (rapidity of the central system), \( \eta_j \) (rapidity of the first particle in the central system in its rest frame, or, in other words, in the C.M. frame of two central particles. For example, consider \( \pi^+\pi^- \) central production. In this case \( \eta_j \equiv y_{\pi^+} \) or \( \eta_j \equiv y_{\pi^-} \) in the \( \pi^+\pi^- \) C.M. frame. It is equal to \( \Delta y \) in (15)), \( \phi_j \) (azimuthal angle of the particle, which we choose to fix \( \eta_j \), in the same reference frame);

\[ \text{ID}=2 : \quad |\vec{\Delta}_1|, |\vec{\Delta}_2|, \phi_0, \phi_1, \xi_1, \xi_2 \quad \text{(see (14))}, \eta_j, \phi_j; \]
\[ \text{ID}=3 : \quad D_\rho, \phi_0, \phi_1, \xi_1, \xi_2 \quad \text{(see (14))}, \eta_j, \phi_j; \text{ here } |\vec{\Delta}_1| = D_\rho \cos(\phi_0), \]
\[ |\vec{\Delta}_2| = D_\rho \sin(\phi_0) \text{ (symmetrycal case)}. \]
\[ \text{ID}=4 : \quad |\vec{\Delta}_1|, |\vec{\Delta}_2|, \phi_0, \phi_1. \]

\begin{verbatim}
void PtoV2()
    calculates all the variables for a 2 \( \rightarrow \) 2 process in the second initialization case (see above), when we use momenta as input.

void PtoV3()
    calculates all the variables for a 2 \( \rightarrow \) 3 process in the second initialization case (see above), when we use momenta as input.

void PtoV4()
    calculates all the variables for a 2 \( \rightarrow \) 4 process in the second initialization case (see above), when we use momenta as input.

X .Y()
    give the access to the private variable Y of the type (class) X (see below the list of private variables). For example std::vector<Particle>& .particles() returns private vector particles, double& .Eta_j() returns Eta_j variable.
\end{verbatim}

Variables:

ExDiff::Particle hadron1, hadron2 : private objects which contain information on initial hadrons.

std::vector<Particle> particles : private object which contains information on final particles.

std::vector<double> variables : private vector which contains values of variables used in the input set.

double sqs : private variable which contains \( \sqrt{s} \) (collision energy).

double t, phi, DeltaT, Delta3D, theta, y

std::vector<double> Delta : private variables which contain \(|t|, \phi, |\vec{\Delta}| = \sqrt{\Delta_x^2 + \Delta_y^2}, \sqrt{\Delta_x^2 + \Delta_y^2 + \Delta_z^2}, \theta, y, \Delta \) of a 2 \( \rightarrow \) 2 process (see (14)).

double t_1, t_2, phi_12, phi_1, xi_1, xi_2, y_c, M_T, M_c, DeltaT_1, DeltaT_2, Delta3D_1, Delta3D_2

std::vector<double> Delta_1, Delta_2 : private variables which contain \(|t_1|, |t_2|, \phi_0, \phi_1, \xi_1, \xi_2, y_c, M_T, M_c, |\vec{\Delta}_1|, |\vec{\Delta}_2|, \sqrt{\Delta_x^2 + \Delta_y^2 + \Delta_z^2}, \sqrt{\Delta_x^2 + \Delta_y^2 + \Delta_z^2}, \Delta_1, \Delta_2 \) (see (14),(16)).
double t, s, Eta, Theta, Phi: additional private variables which contain $\hat{t}$, $\hat{s} = M^2$, $\eta_j = \eta = -\ln \tan \frac{\theta_j}{2}$, $\theta_j$, $\phi_j$ (see (15),(17)).

double y_a, y_b, k_ab_x, k_ab_y, phi_ab: additional private variables which contain rapidities of two central particles a and b, $y_a$ and $y_b$, x and y components of difference between their transverse momenta $k_m = k_a - k_b$, azimuthal angle of the $k_m$ (see (15),(17)).

int flag: private variable which contains 0 if everything is correct in the kinematics and nonzero values in other cases.

std::size_t ID
std::vector<double> masses
std::vector<int> IDs private variables (vectors) which contain information on the set of input variables, masses and ids of final particles.

Used by: Interface.
Includes: <cmath> <complex> <cstdlib> <iostream> <fstream> <map> <string> <utility> <vector> <stdio.h> <cassert>
"I.h" "PI.h" "Particle.h" "Constants.h"

ExDiff::Interface

Purpose: to initialize special class which provides convenient input/output.

Initialization example:

... ExDiff::Interface iogen(datacard);
...

where datacard is the name of a configuration (*.card) file.

Methods:

void GeneratorInfo()
  prints the general information on the generator (authors, version and so on).

void PrintPars()
  prints the general information on the process (authors, process, C.M. energy, version, number of output events, input/output format).

vector<double> MassesInit()
  Initialization of masses. Returns vector of final masses.

vector<int> IDsInit()
  Initialization of masses. Returns vector of final IDs of particles.

ExDiff::Generator GeneratorInit()
  Initialization of the basic generator. Returns the ExDiff::Generator object. Auxiliary function.

ExDiff::KinCM KinematicsInit(ExDiff::Generator& generator)
  Initialization of the basic kinematics. Returns the ExDiff::KinCM object. Auxiliary function.
ExDiff::Event EventInit(ExDiff::KinCM& kin_)
Initialization of the basic event. Returns the ExDiff::Event object. Auxiliary function.

void GetCard(const char * filename)
reads parameters from the configuration file.

ExDiff::Event GenerateEvent(ExDiff::Generator& gen_, ExDiff::KinCM& kin_, ExDiff::Event& ev_)
creates single event according to input data.

void GenerateFile(ExDiff::Generator& gen_, ExDiff::KinCM& kin_, ExDiff::Event& ev_)
creates final output file according to input data.

void GenerateFileCuts(ExDiff::Generator& gen_, ExDiff::KinCM& kin_, ExDiff::Event& ev_)
creates final output file according to input data and special cuts during the generation process (analyzer).

std::string ResultFileName()
creates output filename which is used by default.

std::string DatFileName()
creates input *.dat filename.

std::string GridFileName()
creates input *.grid filename.

int& _X()
returns private variable X. For example, _Nevents returns the private variable Nevents (see below).

Variables:

int IDauthors, IDprocess, IDenergy, version, Nevents, kinformat, outformat:
private variables which define authors, process, C.M. energy, version of the data files, number of events, set of input variables and an output format.

double CSTOT :
public variables which defines the total cross-section

int SETVAR, SETCUT :
auxiliary public variables which define cuts and sets of variables for analyzers.

Used by: main.
Includes: <fstream> <cassert> <cmath> <complex> <cstdlib> <iostream> <map>
<string> <utility> <vector> <stdio.h> <time.h> <iomanip> <limits>
<string> <ctime> "newrand.h" "NDarray.h" "NDgrid.h" "iFunc.h"
"Generator.h" "Particle.h" "Constants.h" "Kinematics.h" "Event.h"
4 Program Installation

Some materials related to the ExDiff physics and generator can be found on the web page

https://exdiff.hepforge.org/

To get the code of the generator one should download the file

https://exdiff.hepforge.org/code/ExDiff2.0.zip

The program is written essentially entirely in standard c++ and compatible with c++11, and should run on any platform with such a compiler.

The following installation procedure is suggested for the Linux users, it was tested with CERN SLC7.

1. install external packages like Pythia, ROOT, HEPMC etc. if you need
2. unpack files to your folder, say, ExDiff2.0
   - use Makefile.inc (we take it from Pythia 8 examples folder) to set your pythia parameters: ROOT_USE, HEPMC2_USE, HEPMC3_USE and all the paths.
   - you can copy also Makefile from Pythia 8 examples folder and add the string
     "-include MakefileExDiff.inc" before "clean"
3. Change the card file in the folder config or use sample files.
4. Go to the main ExDiff2.0 folder with the Makefile
5. $ make ExDiff creates executable without external packages
6. $ make ExDiffPY creates executable linked to Pythia 8
7. $ make ExDiffROOT creates executable linked to Pythia 8 and ROOT. After that you obtain *.root file. To use it inside ROOT you have to include main92.so library.
8. $ make ExDiffHEPMC2 creates executable linked to Pythia 8 and HEPMC2
9. $ make ExDiffHEPMC3 creates executable linked to Pythia 8 and HEPMC3
10. $ ExDiff <cardfile name> [output file name>
    or
    $ ExDiffPY <cardfile name> [output file name>
    etc.
11. $ cd output
12. see output file in the folder output

In the main ExDiff2.0 folder you can see some files:
readme.txt contains brief description of the generator;
mainExDiff.cc is the main function for the generator without external packages;
mainExDiffPY.cc is the main function for the generator linked to Pythia 8;
mainExDiffROOT.cc linked to Pythia 8 and ROOT;
mainExDiffHEPMC2.cc linked to Pythia 8 and HEPMC2;
mainExDiffHEPMC3.cc linked to Pythia 8 and HEPMC3;
Makefile, Makefile.inc, MakefileExDiff.inc define compilation instructions;
Getting Started with the Simple Example

The Simple Example could look as following (file mainExDiff.cc):

```cpp
#include <fstream>
#include <cassert>
#include <cmath>
#include <complex>
#include <cstdlib>
#include <iostream>
#include <map>
#include <string>
#include <utility>
#include <vector>
#include <stdio.h>
#include <time.h>
#include <iomanip>
#include <limits>
#include <string>
#include <ctime>
#include "inc/I.h"
#include "inc/PI.h"
#include "inc/newrand.h"
#include "inc/NDarray.h"
#include "inc/NDgrid.h"
#include "inc/iFunc.h"
#include "inc/Generator.h"
#include "inc/Particle.h"
#include "inc/Constants.h"
#include "inc/Kinematics.h"
#include "inc/Event.h"
#include "inc/Interface.h"

using namespace ExDiff;

int main(int argc, const char** argv){

    // final construction
    // datacard is the argument of the main file (filename)
    const char* datacard = argv[1];

    // reading input data card
    // create interface
    Interface iogen(datacard);
    iogen.GetCard(datacard);
```
ugen().
ugen.PrintPars();

// generator initialization
std::cout << "Initialization of ExDiff table ................." << std::endl;

Generator gen(ugen().GetFUN(), ugen().GetFUNG(),
ugen().GetFUNGTOT());

KinCM kin(ugen().Get_sqs(), ugen().Get_particles());

Event ev(ugen().GetParticles());

UIL
// for output=1,2,3 (PYTHIA8, ROOT, HEPMC etc.)
if (ugen()._outformat() > 0 && ugen()._outformat() < 10){
    std::cout << "Using this card file, you have to link external packages," << std::endl;
    std::cout << "such as PYTHIA, ROOT, HEPMC etc. " << std::endl;
    std::cout << "or change the output format in the card file" << std::endl;
    std::cout << "Now generator is writing the usual file *.exdiff for the process" << std::endl;
    ugen().GenerateFile(gen,kin,ev);
}

if (ugen()._outformat() == 0) {
    // for intrinsic output format
    std::cout << "Writing to the *.exdiff file for the process ............" << std::endl;
    ugen().GenerateFile(gen,kin,ev);
}

// Analyzers can be added (see mainExDiff.cc)
if (ugen()._outformat() >= 10) {
    // for intrinsic output format
    std::cout << "CUTS TO CHECK ++++++++++++++++++++++++++++++++++++
    std::cout << "SETVAR = " << ugen().SETVAR << " " << ugen().SETCUT << std::endl;
    std::cout << "Writing to *.exdiff* files for the process with cuts ............" << std::endl;
    ugen().GenerateFileCuts(gen,kin,ev);
}

return 0;
First, we set the object of the `ExDiff::Interface` class, then print the general information, make initializations, and finally generate an output file. If you need another type of output, you can use other `.cc` files in the basic folder as examples.

5.1 For developers

In this subsection we show how you can make modifications to perform another type of output format. For this purpose we can use direct access to values of momenta and ids of all the particles. In the file `ExDiff2.0/src/Interface.cpp` you can find some comments:

```cpp
// FOR DEVELOPERS ================================
// You can use directly all the parameters of particles in the event
// to make your own output or connection to your software
// instead of the string

ev.AddToFile(outformat,outputfile);

// use your own method with the following parameters of all particles ...
// i: 0 => ev._particles().size()-1 [i changes 0=>3 (2 to 2), ...  
// ev._particles().size() = number of particles in the event
// ev._particles()[i]._ID() = ID of the particle
// ev._particles()[i]._m() = mass of the particle
// ev._particles()[i]._E() = energy of the particle
// ev._particles()[i]._px() = px of the particle
// ev._particles()[i]._py() = py of the particle
// ev._particles()[i]._pz() = pz of the particle
// FOR DEVELOPERS ================================
```

You could create your own class for output. In further versions of the generator we will optimize different output formats. Now we use PYTHIA 8 examples to make output into ROOT trees and HEPMC.

Aknowledgements

Authors of the generator and the IHEP diffractive group thank to Simone Giani, Robert Ciesielski, Jan Kaspar, Gustavo Gil Da Silveira, Michele Arneodo, Joao Varela and other members of the LHC diffractive community for useful discussions and help.
References

[1] V.A. Petrov and R.A. Ryutin, *Patterns of the exclusive double diffraction*, J. Phys. G 35, 065004 (2008).

[2] R.A. Ryutin, *Visualizations of exclusive central diffraction*, Eur. Phys. J. C 74, 3162 (2014).

[3] CT-PPS Technical Design Report, CERN-LHCC-2014-021, CMS-TDR-013, TOTEM-TDR-003 (8 September 2014).

[4] A. Godizov, *Effective transverse radius of nucleon in high-energy elastic diffractive scattering*, Eur. Phys. J. C 75, 224 (2015).

[5] R.A. Ryutin, *Exclusive Double Diffractive Events: general framework and prospects*, Eur. Phys. J. C 73, 2443 (2013).

[6] A. Godizov, *The ground state of the Pomeron and its decays to light mesons and photons*, Eur. Phys. J. C 76, 361 (2016).

[7] A. Godizov, *High-energy single diffractive dissociation of nucleons and the 3P-model applicability ranges*, Nucl. Phys. A 955, 228 (2016).

[8] PDG particle numbering scheme
http://pdg.lbl.gov/2016/reviews/rpp2016-rev-monte-carlo-numbering.pdf
# Index of basic Classes, Methods and Variables

| Class/Method               | Page |
|---------------------------|------|
| ExDiff::NDarray           | 12   |
| ExDiff::NDarray::printC   | 13   |
| ExDiff::NDarray::get      | 12   |
| ExDiff::NDarray::GetDimensions | 12 |
| ExDiff::NDarray::GetValues | 12   |
| ExDiff::NDarray::computeTotalSize | 12 |
| ExDiff::NDarray::computeIndex | 12 |
| ExDiff::NDarray::computeIndexes | 13 |
| ExDiff::NDarray::LoadFromFile | 13 |
| ExDiff::NDarray:: dimensions | 13 |
| ExDiff::NDarray:: values  | 13   |
| ExDiff::NDgrid            | 13   |
| ExDiff::NDgrid::GetVar    | 13   |
| ExDiff::NDgrid::GetDimensions | 13 |
| ExDiff::NDgrid::GetValues | 14   |
| ExDiff::NDgrid::get       | 14   |
| ExDiff::NDgrid::computeIndex | 14 |
| ExDiff::NDgrid::computeTotalSize | 14 |
| ExDiff::NDgrid::printC    | 14   |
| ExDiff::NDgrid::LoadFromFile | 14 |
| ExDiff::NDgrid:: dimensions | 14 |
| ExDiff::NDgrid:: values  | 14   |
| ExDiff::iFunc             | 14   |
| ExDiff::iFunc::Calc       | 15   |
| ExDiff::iFunc::CheckPoint | 15   |
| ExDiff::iFunc::GetGrid    | 15   |
| ExDiff::iFunc::GetTab     | 15   |
| ExDiff::iFunc::GetDimensions | 15 |
| ExDiff::iFunc::GetPoint   | 15   |
| ExDiff::iFunc::GetMinFunPoint | 15 |
| ExDiff::iFunc::GetMaxFunPoint | 15 |
| ExDiff::iFunc::GetMinFun  | 15   |
| ExDiff::iFunc::GetMaxFun  | 15   |
| ExDiff::iFunc::PrintVector | 16   |
| ExDiff::iFunc::printC     | 16   |
| ExDiff::iFunc:: dimensions | 16 |
| ExDiff::iFunc:: minvars   | 16   |
| ExDiff::iFunc:: maxvars   | 16   |
| ExDiff::iFunc:: minfunpoint | 16 |
| ExDiff::iFunc:: maxfunpoint | 16 |
| ExDiff::iFunc:: minfunvalue | 16 |
| ExDiff::iFunc:: maxfunvalue | 16 |
| ExDiff::Generator         | 16   |
| ExDiff::Generator::RNUM   | 17   |
| ExDiff::Generator::printC | 17   |
| ExDiff::Generator::CalcMinus | 17 |
| ExDiff::Generator::CalcFUNG | 17 |
| ExDiff::Generator::GetFUNGTOT | 17 |
ExDiff::KinCM::VtoP2
ExDiff::KinCM::VtoP3
ExDiff::KinCM::VtoP4
ExDiff::KinCM::PtoV2
ExDiff::KinCM::PtoV3
ExDiff::KinCM::PtoV4
ExDiff::KinCM::Y
ExDiff::KinCM:: hadron1, hadron2
ExDiff::KinCM:: particles
ExDiff::KinCM:: variables
ExDiff::KinCM:: sqs
ExDiff::KinCM:: t, phi, DeltaT,
ExDiff::KinCM:: Delta3D, theta, y, Delta
ExDiff::KinCM:: t_1, t_2, phi_12, phi_1,
ExDiff::KinCM:: xi_1, xi_2, y_c, M_T, M_c
ExDiff::KinCM:: DeltaT_1, DeltaT_2,
ExDiff::KinCM:: Delta3D_1, Delta3D_2
ExDiff::KinCM:: Delta_1, Delta_2
ExDiff::KinCM:: t_hat, s_hat, Eta_j, Theta_j, Phi_j
ExDiff::KinCM:: y_a, y_b, k_ab_x, k_ab_y, phi_ab
ExDiff::KinCM:: flag
ExDiff::KinCM:: ID, masses, IDs
ExDiff::Interface
ExDiff::Interface::GeneratorInfo
ExDiff::Interface::PrintPars
ExDiff::Interface::MassesInit
ExDiff::Interface::IDsInit
ExDiff::Interface::KinematicsInit
ExDiff::Interface::EventInit
ExDiff::Interface::GetCard
ExDiff::Interface::GenerateEvent
ExDiff::Interface::GenerateFile
ExDiff::Interface::GenerateFileCuts
ExDiff::Interface::ResultFileName
ExDiff::Interface::DatFileName
ExDiff::Interface::GridFileName
ExDiff::Interface::X
ExDiff::Interface:: IDauthors, IDprocess, IDenergy,
ExDiff::Interface:: version, Nevents,
ExDiff::Interface:: kinformat, outformat
ExDiff::Interface:: CSTOT
ExDiff::Interface:: SETVAR, SETCUT