Design of the pluto event generator

To cite this article: I Fröhlich *et al* 2010 *J. Phys.: Conf. Ser.* **219** 032039

View the article online for updates and enhancements.

Design of the Pluto Event Generator

I. Fröhlich[1], T. Galatyuk[1], R. Holzmann[2], J. Markert[1,2],
B. Ramstein[3], P. Salabura[2,4] and J. Stroth[1,2]

[1] Institut für Kernphysik, Goethe-Universität, 60438 Frankfurt, Germany

[2] GSI Helmholtzzentrum für Schwerionenforschung GmbH, 64291 Darmstadt, Germany

[3] Institut de Physique Nucléaire d'Orsay, CNRS/IN2P3, 91406 Orsay Cedex, France

[4] Smoluchowski Institute of Physics, Jagiellonian University of Cracow, 30-059 Kraków, Poland

E-mail: `Froehlich@physik.uni-frankfurt.de`

Abstract.
We present the design of the simulation package Pluto, aimed at the study of hadronic interactions at SIS and FAIR energies. Its main mission is to offer a modular framework with an object-oriented structure, thereby making additions such as new particles, decays of resonances, new models up to modules for entire changes easily applicable. Overall consistency is ensured by a plugin- and distribution manager. Particular features are the support of a modular structure for physics process descriptions, and the possibility to access the particle stream for on-line modifications. Additional configuration and self-made classes can be attached by the user without re-compiling the package, which makes Pluto extremely configurable.

1. Introduction

Due to the fact that experimental setups are usually not suited to cover the complete phase space, event generators are very important tools for experiments. Practically, the experimental physicist needs a tool in hand which allows them to control almost all kinematic variables with a manageable user interface and to exchange the physics models which have to be compared to measured data.

In our contribution, we present the software structure of the Pluto event generator [1] originally developed for the HADES experiment [2] but successfully used by other collaborations in the hadronic physics field as well. Its recent redesign partly discussed in this contribution enhanced its flexibility and provided new features which allow Pluto to meet new challenges coming with the detector studies for the new FAIR experiments PANDA [3] and CBM [4].

Pluto is a collection of C++ classes, adding up to the framework of a simulation package for hadronic physics interactions in the energy regime up to a few GeV. It is launched interactively within the ROOT [5] environment, and makes use of ROOT only, without requiring additional packages. The focus is on streamlining particle generators by providing the tools to set up and manipulate particles, reaction channels, and complex reactions, as well as applying experimental filters on the reaction products, such as geometrical acceptance and kinematical conditions. Typical simulations may be executed with a few lines of input, with no expertise required on the part of the user. It standardizes simulations by providing a common platform that is accessible via the analysis environment (ROOT), with a straightforward user interface that does not inhibit non-experts in simulations.

The output may be analyzed on line, or forwarded to a digitization package. The Pluto framework includes models for hadronic and electromagnetic decays, resonance spectral functions with mass-dependent widths, and anisotropic angular distributions for selected channels. A decay-manager interface enables "cocktail" calculations. An extensive particle data base is available, with capabilities to support user-defined ones. Various particle properties and decay modes are included in the data base. Thermal distributions are implemented, enabling multi-hadron decays of hot fireballs.

The general philosophy of Pluto is that on one hand the users should be able to drive the package in a very easy way, while on the other hand are allowed to customize it without much restrictions. This means that the steering application has the opportunity to add new decays/particles among with new models up to the incorporation of plugins[1].
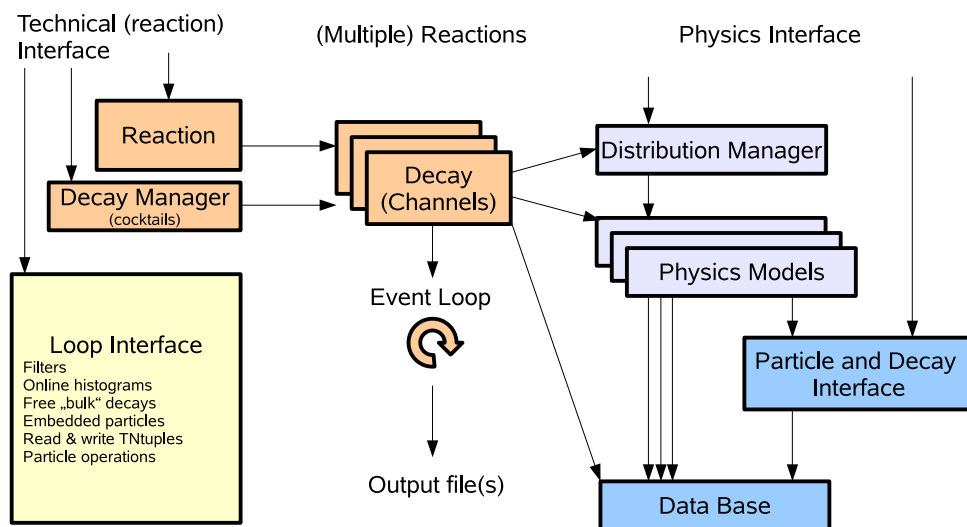

Figure 1. Structure and design of Pluto: the (orange) boxes in the upper left show the user classes to set up and handle reactions (and decay channels). The (yellow) box on the left is the interface for accessing the particle stream while the event loop is running. The (blue shaded) boxes in the lower right corner represent the data base for the particles and decays among with a user interface for customization. The (light blue) boxes above are the distribution interface and the included physics models (color online).

The structure and design of Pluto are sketched in Fig. 1. The framework can be roughly divided into 3 parts: a user interface for event production, a physics package, and an event loop interface. In this report we would like to concentrate on the latter two issues: the first one allows to add and/or select physics cases, where models can be either build-in models or runtime replacements, which will be discussed in Sec. 2. This area of Pluto works even outside the event loop for the calculation of parameters (mass distributions, decay widths,...). The other block allows for external particle decay and acceptance filter classes (see Sec. 3) and filling of on-line histograms (see Sec. 4).

As the interface for event production is described in detail elsewhere [1] in this context we will give only a short example. After all models have been configured accordingly, an elementary reaction chain is defined with a character string containing the reaction products. To perform,

---

[1] The PC gaming industry, making benefit from charge-free down-loadable packages, calls such extension features also "mod".

e.g., the Δ^+ production in the *pp* reaction with a consecutive Dalitz decay as discussed in the following sections, the commands as defined below are sufficient:

```
PReaction *r = new PReaction("1.25","p","p","p D+ [p dilepton [e+ e-] ]");
r->Print();
r->Loop(10000);
```

with the number 1.25 as the kinetic beam energy in GeV, the beam and target particle indicated by following two options and finally a string describing the decay. Particles are defined via their data base name (as "p" for a proton or "D+" for a Δ^+). The setup of the reaction channels and the attachment of all selected distributions and models is done by the framework, with the possibility to dump the complete chain and its models via the `Print()` method. Finally, the command `Loop(n)` produces *n* events.

Then, event-by-event, the individual steps in each reaction are executed via a common interface of all classes making distributions (see [1] for details) which do the calculation of final particle tracks (also the unstable ones) which are stored in an array.

2. The models
2.1. Architecture
In the Pluto framework, each physics process is described by a number of objects, instantiated from dedicated classes. Beside the objects which communicate with the event loop and set the 4-momenta of the resulting particle tracks (to form the above-mentioned "distributions"), Pluto offers methods to make calculations including various channels and their accompanied functions in advance to (or even without) the event-loop. For this purpose, a relative data base has been included, which connects particles and decays to "models". At least one (so-called "*primary*") model is needed per decay in order to describe the respective mass shape (usually calculated by a mass-dependent Breit-Wigner in a recursive way) and the partial/total decay width. Each selected model can be accessed from all locations of the software package as well as by the user.

The choice of which of the allover offered physics models are linked to the data base can be done inside the production ROOT-macro by calling a distribution manager (for details about the handling see [1]), which itself takes care that only one primary model per decay and particle is selected, respectively. Moreover, Pluto offers the user the freedom to attach user-defined models based on its own sources or even new classes (written in C++) at runtime, thereby replacing built-in models. These features make Pluto an extremely versatile and highly configurable tool for simulations, while guaranteeing overall consistency at all times.

One specific feature of Pluto is to use non-monolithic decay models which allows for the splitting of the underlying physics process into different models in a very granular way (e.g., to exchange form factors or total cross sections). This turned out to be a very important tool in order to check various scenarios along with measured data. Therefore Pluto allows for the attachment of secondary models for all kinds of purposes. Here, a secondary model is an object for a particle/decay returning a (complex) number as a function of a defined number of values. We do not attempt to have a base class for each kind of function and model (e.g., a total cross section base class, a base class for $d\Gamma/dm$, etc.) because this makes further extensions difficult. Our approach is as follows: for each class of secondary models a new entry (defined by a unique name) is added dynamically to the data base. Consequently, each model is bound to two data base keys: one to link it to a decay (or particle) and one more for the type of secondary model.

Anyhow, there the user does not need to take care about these issues, as the following template example shows.

2.2. Template example

In Pluto, each model is inherited from the base class `PChannelModel` with a lot of useful features. In order to offer the result of the calculation the `GetWeight()` method has to implemented. Let us create such an example class as a template for the following discussion and future developments:

```
class PHelperFunctionTest : public PChannelModel {
public:
  PHelperFunctionTest(const Char_t *id, const Char_t *de, Int_t key);
  Double_t GetWeight(Double_t *x, Int_t *opt=NULL);
}
```

with an realization of the (in this case 1-dimensional) function:

```
Double_t PHelperFunctionTest::GetWeight(Double_t *x, Int_t *opt) {
  Double_t xx = x[0];
  val = ...
  return val;
};
```

Objects can be instantiated to work as a model for specific physics processes by the usage of a parser residing inside the constructor of the base class:

```
PHelperFunctionTest *obj = new
  PHelperFunctionTest("unique_name@D+_to_p_dilepton/helper","Test",-1);
```

Here, the `unique_name` is used to identify the object inside the distribution manager. The string hereafter binds the new object to a decay which is in this case $\Delta^+ \to p\gamma^*$. The optional appendix `/helper` enables the new object to be a secondary model of this type.

Let us now assume the primary model has to access its helper function. In this case it is sufficient to execute the following command at a single place (an `Init()` function is available for that):

```
PChannelModel *sec_model=GetSecondaryModel("helper");
```

and execute the daughter method `sec_model->GetWeight(value)` whenever required. The access to alien secondary models is possible as well but a further discussion on this topic would exceed the scope of this report.

2.3. Example: form factors

The power of such a splitting can be demonstrated best with an real example. For this purpose let us use the study of the electromagnetic Δ Dalitz decay $\Delta \to N\gamma^* \to Ne^+e^-$. As explained in the introduction Pluto reads this as a multi-step process: in the first step, the Δ mass is sampled using a mass-dependent width. In the second step, the already sampled Δ mass m_Δ is used by the following Dalitz decay model to sample the di-electron (=virtual photon) mass m_{ee}.

By using a dedicated plugin:

```
makeDistributionManager()->Exec("dalitz_mod: krivoruchenko");
```

the following model for the di-electron mass spectrum is enabled, using the prescription of Ref. [6] with:

$$\frac{\mathrm{d}\Gamma_{m_\Delta}^{\Delta \to Ne^+e^-}(m_{ee})}{\mathrm{d}m_{ee}} = \frac{2\alpha}{3\pi m_{ee}}\Gamma_{m_\Delta}^{\Delta \to N\gamma^*}(m_{ee}) \tag{1}$$

and the decay rate given by Krivoruchenko *et al* [7]:

$$
\begin{aligned}
\Gamma_{m_\Delta}^{\Delta \to N\gamma^*}(m_{\gamma^*}) &= \left(G_{m_\Delta}^{\Delta \to N\gamma^*}(m_{\gamma^*})\right)^2 \\
&\quad \times \frac{\alpha}{16} \frac{(m_\Delta + m_N)^2}{m_\Delta^3 m_N^2} \sqrt{y_+ y_-^3}, \\
y_\pm &= (m_\Delta \pm m_N)^2 - m_{ee}^2
\end{aligned}
\tag{2}
$$

where the index N refers to the produced nucleon, e is the electron charge, and $G_{m_\Delta}^{\Delta \to N\gamma^*}(m_{\gamma^*})$ represents the electromagnetic transition amplitudes. Here, from the software point of view, it is important to know that this factor is a separate object in Pluto making the addition of new complementary descriptions very easy. This avoids the usage of nasty flags which is error-prone. E.g., following the QED approach (assuming that no hadronic transition form factor is involved) from Ref.[7] gives:

$$
\left(G_{m_\Delta}^{\Delta \to N\gamma^*}(m_{\gamma^*})\right)^2 = G_M^2 + 3G_E^2 + \frac{m_{\gamma^*}^2}{2m_\Delta^2} G_C^2
\tag{3}
$$

with G_M, G_E, and G_C as the magnetic, electric and Coulomb transition form factors.

Alternatively, one can switch to the VMD-like form factor of Iachello and Wan [8] which is already implemented and is discussed in more detail in a separate publication [9].

After activating this plugin, both above-mentioned versions (VMD and QED) are offered, appearing as objects with their unique names in the directory-like distribution manager structure.

```
makeDistributionManager()->Print("vmd");
--------------------
PDistributionManager
--------------------
  root                Root group : 494 objects (of 504), subgroups: 9
     particle_models  Mass sampling of particles : 27 objects (of 29)
     decay_models     Phase space mass sampling & decay partial widths:
                      219 objects (of 225)
     polar_angles     Polar angles in elementary particle production :
                      10 objects (of 10)
     genbod_models    Momentum sampling : 220 objects (of 220)
         ...
     vmd              VMD form factors : 0 objects (of 2)
( )  D+_vmd_ff        VMD ff for D+ -> p e+e-
 (X) D+_qed_ff        QED ff for D+ -> p e+e-
         ...
     qed              QED form factors : 2 objects (of 2)
```

Possible conflicts between the QED and VMD models are indicated. The user can now activate all models inside the directory via:

```
 makeDistributionManager()->Enable("vmd");
```

3. The event loop interface

The aim of the event loop interface is to allow access to the particle bulk before it is written to disk by providing an interface for bulk decays (also using 3rd party event generators such as Pythia [10]), file access, implantation of embedded particles for detector
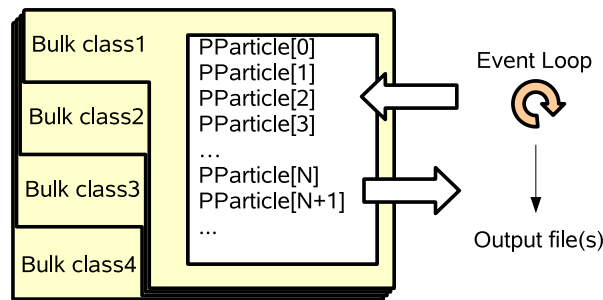
Figure 2. The bulk interface.

studies, and filter classes. It works - as indicated in Fig. 2 - by stacking individual bulk classes (by `PReaction::AddBulk(PBulkInterface * obj)`) and making use of the base-class `Modify()` method which handles the embedding and removing of particles or marks them as decayed/invalid.

Like in the previous section, user classes can be compiled - if needed - and added on-the-fly, which could be very helpful in the context of FAIR experiments studies, as the following applications show:

3.1. File access:
Particle tracks might be added from (or written to) user files in any self-defined format. This feature makes Pluto open for other collaborations and enables it to be used as an "afterburner" for externally created simulation data or output of transport models.

3.2. Embedded particles:
Test particles can be embedded into a background reaction which is an important tool for detectors studies, e.g. to test tracking in a high multiplicity environment. Moreover, along with the previous feature the mixing of embedded particles with either real events or 3rd party simulation is one of the applications. In the first case, the reconstructed vertex of the analyzed experimental data is used to simulate rare probes (as the di-electrons) at exactly the same vertex. In this way the efficiency of these probes can be calculated with the best background one can have, namely measured data.

3.3. Bulk decay:
In addition to a well defined reaction chain, Pluto offers free "bulk" decays, i.e. the decay of (embedded) particles following the calculated mass-dependent branching ratios. At this stage it is possible as well to employ 3rd party event generators. Only to give an example: Pluto can generate embedded particles like resonances with a selected distribution and the Pythia package (or Pluto itself) can be used for their realistic free decay.

3.4. Detector acceptance class:
The particles created in the previous steps can be pre-filtered using a "detector acceptance" user class following a class definition as outlined below:

```
class PMyDetector : public PBulkInterface {
private:
  Double_t *acc;
public:
```
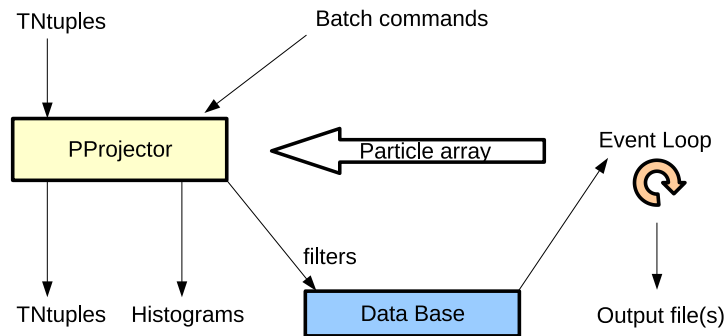
Figure 3. Overview of the `PProjector` design.

```
  Bool_t Modify(PParticle ** stack, int *decay_done, int * num, int stacksize);
};


PMyDetector::PMyDetector() {
   acc = makeStaticData()->GetBatchValue("acc_value");
};
```

with `acc_value` as a booked batch value (see below for more information) and, e.g.:

```
 Bool_t PMyDetector::Modify(PParticle ** stack, ... ) {
 *acc=1.;
 for (int i=0; i< *num; i++) {
   PParticle * cur = stack[i];
   if ("check for cur") *acc=0.;
   }
 }
```

4. Projecting events

4.1. General idea

The usual way to analyze generated events (beside a full Monte-Carlo production including digitization) is to open the simulation file with an appropriate analysis macro, loop over the number of events, read the particle objects and do the required operations inside the loop. E.g., in hadronic physics, in particular for exclusive studies, usually missing and invariant masses are reconstructed, and possibly after boost operations angular correlations are investigated. In addition, signal selection has to be done not only on a single track level but using particle correlations (such as a invariant/missing mass selection). This prevents to use the features provided by `TTree::Draw()` only, as these are fixed to single track properties.

The main aim of the "PProjector" (see Fig. 3) is to offer a simplified analysis method following a similar idea as `TTree::Draw()` in order to project particles (and their correlations) onto on-line histograms and/or `TNtuple`s by making use of the bulk interface described in the previous section. This means, no analysis macro is needed for a large number of typical observables.

There are two ways to include the projector into the event loop. Either one or more of its instances are added to the bulk interface (for an alternating mixing with other classes), or, which is sufficient for the most cases, the short version `PReaction::Do(char * command)` can be used, where `command` follows a C++-like (but more simple) syntax. Each of these commands - just forming steps in a batch - is executed inside the event loop.

4.2. Histograms

To fill histograms one can use the extended method `PReaction::Do(THx * histo, char * command)` where `histo` points to a 1/2/3-dimensional histogram.

In this case the command has to define the key variables `_x, _y` and `_z` as the respective axis values, depending on the chosen dimension of `histo`. The Pluto particle array objects can be accessed via its Pluto particle name inside brackets as `[name,num]` with the optional `num` as the consecutive number of equal particles inside each event. All objects can be combined with a number of build-in methods, moreover all browsable methods of the ROOT-class `TLorentzVector` can be employed, as the following example indicates:

```
r->Do(histo1,"_x = [D+]->M()");
```

which fills the 1-dimensional `histo1` with the Δ^+ mass. Invariant masses (composite particles) can be formed by adding the objects:

```
"_x = ([e+] + [e-])->M2()"
```

and stored for a consequent use in semi-colon separated sub-commands without requiring a "new" operator:

```
"dilepton = ([e+] + [e-]); _x = dilepton->M2()"
```

Let us now come to a more complex example, which is the calculation of the polar emission angle of one of the outgoing protons in a $p + p$ reaction (N.B. Pluto treats the c.m. system as a particle named `p + p`):

```
"p1 = [p,1]; p1->Boost([p + p]); _x = cos(p1->Theta())"
```

In this example, in the first sub-command the first appearing proton is copied to the object `p1` which is furthermore boosted into the c.m. energy frame and in the last step the cosine of its polar angle is written as a value to the x-axis.

4.3. TNtuple in- and output

`TNtuples` are very useful as they allow one to make additional selections (and definitions of the histogram axis) at a later stage. Here, the projection to `TNtuples` is a very simple task using the `PProjector` following the syntax as described above. Let us now assume, the momenta of simulated η's should be stored. Then the first step would be to define the `TNtuple` with branch names:

```
TNtuple *ntuple = new TNtuple("ntuple","...","eta_px:eta_py:eta_pz");
```

Hereafter, the `Do()`-method has to be called like in the above-indicated example but with a pointer to a `TNtuple`-object instead of a histogram and the variable names of the ntuple instead of the axis names:

```
"eta_px = [eta]->Px(); eta_py = [eta]->Py(); eta_pz = [eta]->Pz()"
```

The filling of the ntuple is done by Pluto; only (but on purpose) the opening and closing of the root-file has to be called externally.

In addition to the writing of `TNtuples` it is also possible to read (back) any ntuple, modify its values and write it back or make histograms. Let us use the ntuple of the previous example which uses the ntuple (usually from a file) and defines an *empty* reaction, but attaches the ntuple as a 3rd party input:

```
PReaction my_reaction;
my_reaction.Input(ntuple);
```

The next step is to use the branches of the ntuple in the batch command line to, e.g., reconstruct the 4-momentum:

```
"myeta = P3M(eta_px,eta_py,eta_pz,0.54745)"
```

with `P3M` as the short syntax for a vector constructor with 3 momenta and the particle mass as parameters. At this point any kind of mathematical operation available via `TFormula` can be applied as well. Finally using `Loop()` without a number reads all events from the ntuple.

4.4. Filter commands

The employment of batch commands allows the user to define on-line filters on single particle properties as well as kinematic variables which are accessible as indicated above. These filters can be used to select events for single histograms only or forwarded to the event loop to apply physical constraints, before a full event is written to disk, hence reducing the event file. Variables prepared in a detector class as outlined in Sec. 3.4 can be integrated very easily.

This is realized by using the `if`-command, followed by the variable or a combined condition based on particle observables. The remaining batch sub-commands are executed only under a positively evaluated condition. It goes without saying that the condition statement can combine all methods among with the established `TFormula` syntax:

```
"if(obj->P() > 0.3 && obj->P() < 0.6); dowhatever..."
```

In this case, histograms and ntuples in the `Do()` method are only filled under the defined condition, or, "dowhatever" can be used to re-define variables.

In this context, it should be noted, that variables, starting with "**#**" are interpreted by the Pluto event loop as a condition for the final event file. In our next case, an event is accepted if both the electron and the positron are produced with polar angles in the range of 18 to 85 degrees:

```
"theta_ep = ([e+]->Theta() * 180.)/TMath::Pi()"
"theta_em = ([e-]->Theta() * 180.)/TMath::Pi()"
"#filter = 1; if(theta_ep<18 || theta_ep>85 || theta_em<18
                theta_em>85); #filter = 0"
```

Consequently, by making use of such "filter" variables, the detector class `MyDetector` from Sec. 3.4 can be employed as follows:

```
"#acc_filter = acc_value"
```

Optionally, the prepared acceptance flag can be combined with kinematical conditions as described above. In such a case, pre-filtering on all physics observables can be done in order to avoid the compute-intensive digitization of the rejected events.

5. Summary

In summary, we presented new developments for the Pluto event generator which moves this package from being a pure event generator to a comprehensive framework. The particular features are the bookkeeping of various models and the user interface for their simple selection. The new event loop interface allows the user to mix several sources, to define and enable online selections, project to histograms and to make a lot of operations. Detector acceptance and resolution can be included without much overhead, which is of particular importance for detector studies in the context of the FAIR project.

In the near future, we will implement general purpose resolution scenarios using a common data format and file readers to transport models (like UrQMD [11] and HSD [12]). This will allow the user to test very rapidly the performance of different detector configuration options.

Acknowledgments

References

[1] Fröhlich I *et al.*, PoS **ACAT2007**, 076 (2007) *Preprint* arXiv:0708.2382 [nucl-ex].
[2] Agakishiev G *et al.* [The HADES Collaboration], *Preprint* arXiv:0902.3478 [nucl-ex].
[3] Messchendorp J. G. [PANDA Collaboration], *Preprint* arXiv:0711.1598 [nucl-ex]; Seitz B [PANDA Collaboration], *Prepared for 16th International Workshop on Deep Inelastic Scattering and Related Subjects (DIS 2008), London, England, 7-11 Apr 2008.*
[4] Friese V, Nucl. Phys. A **774**, 377 (2006); Senger P, Galatyuk T, Kresan D, Kiseleva A and Kryshen E, PoS **CPOD2006**, 018 (2006); Friese V, PoS **CPOD07**, 056 (2007).
[5] http://root.cern.ch
[6] Wolf G, Batko G, Cassing W, Mosel U, Niita K and Schaefer M, Nucl. Phys. A **517** (1990) 615.
[7] Krivoruchenko M I and Faessler A, Phys. Rev. D **65**, 017502 (2002) *Preprint* arXiv:nucl-th/0104045.
[8] Wan Q and Iachello F, Int. J. Mod. Phys. A **20** (2005) 1846; Wan Q, Ph.D. Thesis, Yale University, New Haven, Connecticut (2007); Iachello F, priv. comm. (2008).
[9] Fröhlich I *et al.*, *Preprint* arXiv:0909.5373 [nucl-ex].
[10] Sjostrand T, Mrenna A and Skands P, JHEP 0605 (2006) 026, *Preprint* hep-ph/0603175.
[11] Bleicher M *et al.*, J. Phys. G **25**, 1859 (1999) *Preprint* arXiv:hep-ph/9909407.
[12] Cassing W and Bratkovskaya E L, Phys. Rept. **308**, 65 (1999).