

# Pluto<sup>++</sup>

## A Monte Carlo simulation tool for hadronic physics

**GSI Report 3, July 2000**

**Marios A. Kagarlis, GSI**

revised in March 2005

by R. Holzmann, GSI

A versatile package for Monte Carlo simulations of hadronic interactions in C++ is presented, designed for compatibility with the ROOT analysis environment. Realistic models of resonance production, hadronic, and electromagnetic decays are implemented, motivated by the physics program of HADES. Empirical angular-distribution parametrizations for selected processes are utilized as well, such as resonance excitation in hadronic interactions, and nucleon-nucleon elastic scattering. The code comprises a self-contained framework for stand-alone principle simulations, including an extensive data base of elementary particles and properties with support for additional user-input data, as well as utilities for the implementation of elementary detector setups and acceptance cuts. A standard interface for further on- and off-line processing of generated events with GEANT is also supplied. User-defined tasks via macros and derived classes are facilitated by the flexible design of the code.

## I. Introduction

Simulations are an integral part of experimental programs associated with scattering experiments and particle accelerators. Such studies are required both in order to understand the properties of experimental setups (e.g. reaction-dependent acceptances of various types of detectors), as well as to gain insight into the processes of interest, so that relevant experiments may be optimized and experimental spectra may be interpreted.

This package<sup>\*</sup> is predominantly geared towards elementary hadronic, and to a lesser extent heavy-ion induced reactions at intermediate to moderately high energies, motivated by the physics program of [HADES](#) [1]. As the latter has entered the commissioning phase, the need for realistic principle simulations is evident and growing, and a standardized and efficient tool to facilitate them is forthcoming.

Monte Carlo simulations are often carried out by dedicated and consequently inflexible codes, designed expressly for the task of generating a particular reaction under specific conditions. Although such an approach may be adequate for simple one-time tasks, on a larger scale it results in repetitive programming, and may render comparisons intractable and results difficult to verify. In the context of [HADES](#) a uniform, as well as flexible and efficient simulation package is desirable.

## II. Overview

The guiding principles in the development of **Pluto<sup>++</sup>** have been the following:

- a. The analysis environment of the [HADES](#) experiment is [ROOT](#) [2]. For on-line accessibility and compatibility with the data structures a [ROOT](#) based simulation code is preferable.
- b. The choice of [ROOT](#) dictates C++ as the programming language, which is also the natural choice for a modular and reusable code.
- c. Standard resources ought to be used, such as the [CLHEP](#) [3] library, which had been explicitly utilized in earlier releases until the required utilities were imported in [ROOT](#).
- d. The [HADES](#) Collaboration uses [HGeant](#) [4], a code based on [GEANT3.21](#) [5], for simulating charged-particle tracks through the spectrometer. A "fast" simulation package for principle simulations with

simplified detector geometries is required in addition, which can also serve as event generator to complement elaborate simulations with [HGeant](#). Further development is predominantly required in the area of particle generators and the implementation of physics models.

e. Required user input is minimized with built-in defaults automating unambiguous tasks. User-defined tasks are facilitated via macros and derived classes.

f. Compatibility with the [HADES](#) analysis package [HYDRA](#) is ensured by the common [ROOT](#) platform.

## II.a Structure

A set of five classes comprise the framework of an event-generator package ( Fig 1), and provide additional tools to facilitate principle simulations, such as the possibility to input simple detector geometries and impose geometrical and kinematical acceptance cuts.

A data base of particles commonly encountered at intermediate energies, including their quantum numbers, properties, decay modes, and static branching ratios is contained in the class [PData](#). Member functions for input/output management of additional user-defined particles are also provided, as well as physics models for the calculation of spectral functions, branching ratios of hadronic resonances, the random sampling of masses, total and partial widths, branching ratios, and lifetimes.

The [PParticle](#) class defines "particle" objects, the most elementary unit in the context of simulations with this package, and contains functions for handling particle observables. In the parlance of C++, this class "inherits" from native 3- and 4-vector [ROOT](#) classes. Particle-interaction models are implemented in the [PChannel](#) class, through which single-step reaction or decay channels are defined, These models include empirical angular-distribution parametrizations for selective processes. A channel "object" represents any single step in a reaction chain, comprising of a parent, a decay model, and the decay products. A succession of channels adds up to a full reaction chain, handled by the [PReaction](#) class, which also contains functions for the execution of simulated event loops. Complementary to this is the [PFilter](#) class, through which detector-specific acceptance filters may be imposed. The latter utilizes a native [ROOT](#) class that permits the setting up of formulas as string objects, translating to filter constraints. Last, multi-reaction "cocktail" calculations are facilitated via the [PDecayManager](#) class (contributed by [Volker Hejny](#)).

A number of basic utility functions are supplied by the [PUtils](#) class, for elementary operations such as angular-momentum algebra (i.e. Clebsch-Gordan and Racah coefficients), array sorting, and random-number generation from standard distributions. The latter are based on native [ROOT](#) random-number generators, to be discussed in more detail in a later Section.

A limited number of post-generation detector-specific analysis classes are also distributed, including [PTrackH](#) for the [HADES](#) spectrometer, and [PTrack](#) for the Crystal Barrel setup (undocumented, contributed by [James Ritman](#) and Marc-André Pleier). These may serve as examples of more elaborate user-implemented analysis routines with elaborate detector setups.

Macros for the compilation of the package (Linux platform), as well as example simulations are also contained in the distribution. A shared-object library module is produced from the compilation of the source code, for use with the [ROOT](#) analysis package via its Cint interface (C++ command lines).

**Figure 1** The class structure of **Pluto<sup>++</sup>**: Black arrows indicate inheritance, colored arrows output data flow, and dotted arrows optional operations.



a thermal source [18], and the implementation of a model for  $NN$  scattering on the deuteron with one spectator nucleon [19], take advantage of the elementary hadronic interaction models contained in the code, extending its capabilities towards addressing reactions with complex nuclei.

Detector-specific features are envisioned to be implemented predominantly via user-defined classes and macros, as e.g. with [PTrackH](#). Nonetheless some basic [HADES](#) features have been implemented, as e.g. the  $\pi^\pm$  beam profile from Ref. [20], as well as a skeleton class intended for the future implementation of additional spectrometer features ([PHADES](#)) that currently contains functions for the smearing of charged-particle 4-momenta for a specific [HADES](#) setup [21] (see also [AIII](#)).

## II.c Output

The output is controlled by a number of parameters, and consists of information on generated events such as particle id's, 4-vectors, production vertices, and success or failure with regard to acceptance filters. The [ROOT TTree](#) structure is utilized, "trees" whose "branches" or "leaves" are complex objects rather than simple integer- or real-valued variables. This permits the storage of particle objects directly on file, and offers versatility in keeping track of details such as vertex and filter information, as well as in subsequently retrieving, manipulating, and plotting particle-related observables. Output data may be further processed by [HGeant](#) either on-line event-by-event, or in batch from properly formatted ASCII output files that are also generated by the package on demand. The output structure will be later discussed in more detail (Fig 2).

## III. Description

This Section discusses the main features of the code. Documentation on the [ROOT](#) (<http://root.cern.ch/>) and [HGeant](#) resources is independently available, and may be useful as a cross reference for better understanding the philosophy and functionality of certain aspects of the package. ROOT-generated HTML [files](#) may be created from the compiled module, which may serve as a detailed user manual. Examples are demonstrated in Section [IV](#). Together with the application macros accompanying the distribution package, they are intended as a starting point for the user to gain some hands-on experience with executing simulations with this code. A selective summary of the theoretical models that have been implemented in the package is given in Appendix [I](#), and some numerical aspects are discussed in Appendix [II](#). The [HADES](#) detector-specific features that are built in are reviewed in Appendix [III](#).

### III.a PData Class

A data base currently comprising of hadronic resonances in the energy range of [HADES](#), leptons, the photon (real and virtual as dilepton), and some light ions (e.g. deuteron, triton, alpha) are currently included in the permanent data base which is coded in the class [PData](#). The particles are assigned names, pid numbers in the [GEANT3](#) convention, and quantum numbers that are required by the algorithms in the code: static mass and width (units of GeV), charge, parity, spin, isospin, baryon number and meson flag, or lepton number. Decay modes including a decay-product identification scheme, static branching ratios, and a text description are also included. The permanent data base contains all the particles typically encountered at intermediate energies, and is static (i.e. does not need to be instantiated by a constructor, and is available for use upon loading the shared-object simulation library module). Additional user-defined particle data may be loaded via I/O functions, up to a total of 999 particles. Upon utilization of this option dynamic arrays are set up, persisting during the current [ROOT](#) session, effectively assimilating declaration by a constructor. Non-permanent and permanent particles are treated on an equal footing with regard to the functionality of the member functions and the available physics models.

External data may be loaded either from an ASCII file prepared by the user (recommended mode), or interactively. In the latter case the user has the option of saving non-permanent data on file for future use, to guarantee consistency if generated events are to be analyzed during a separate [ROOT](#) session. A number of (static global friend) functions are available for i/o operations:

**listParticle( ), listParticle(pid), listParticle("name"):** Without an argument, a complete list of the available particles with partially-listed properties is displayed. With either the particle id

(integer), or name (character string) as arguments, all the available properties on a particle are displayed, including decay modes.

**listModes( ), listModes(pid), listModes("name"):** As above, for the decay modes.

**loadData( ), loadData("file name"):** Without an argument, the user enters a dialog sequence to input external data interactively; with a file name (character string) as input argument, data are loaded from file.

**storeData("file name"):** Write non-permanent data to file.

**clearData( ):** Dump non-permanent data and revert to the permanent data base; if external data have been loaded interactively and have not been previously saved, they are written on a file whose name is assigned automatically before clearing.

The calculation of dynamic (mass-dependent) observables is also induced by the [PData](#) class; in particular, decay widths are calculated for the following types of decays:

- a. Decay of any hadronic resonance into any two stable or unstable hadrons: this covers the majority of resonance decay modes.
- b. Decay of a baryon (resonance excitation) into a nucleon and two pions, the latter coupled to s-wave (angular momentum) and zero isospin. This is a commonly occurring baryonic-resonance decay mode. For the purpose of calculating the decay width, in this case the two pions are treated as a single dipion.
- c. Dalitz-decay modes, enumerated [earlier](#) in Section [II.b](#).
- d. Direct vector-meson dilepton decays.

For resonances that decay via any of the above-mentioned decay modes mass-dependent total widths and branching ratios are also calculated. Otherwise, static data from the data base are used. In calculating the total widths 1) unitarity is assumed (i.e. the sum of the branching ratios of all the decay modes that are kinematically accessible is unity), and 2) the mass distribution function over its valid range is normalized to unity.

The calculation of partial and total widths for resonances with multiply embedded decays (i.e. producing unstable particles which themselves proceed to decay) is performed recursively, based on formalism [\[6-14\]](#) generalized by the author to encompass heavy resonances in general, with arbitrary nested-decay "depth". This extension enables the rapid calculation of realistic spectral functions for real or fictitious heavy resonances whose properties can be tweaked by the user, and goes beyond the capabilities of well-established theoretical models and corresponding codes that are currently available in the literature (see Section [A1a](#)).

The following scheme is implemented in sampling widths or branching ratios: The first time a particle is referenced (i.e. by requesting its total or partial width or branching ratio for any mode as a function of mass), the function **getDepth(pid)** is called, which determines the number of nested decays of the referenced particle, identifies recognized nested decay modes (enumerated earlier), and sets up corresponding flags. In the process, a host of dynamical arrays are updated as needed, to accommodate new flags and data. Subsequently, for all of the nested particles and decay modes that had not been previously accessed (through, possibly, the decay chain of another heavy resonance referenced at an earlier time), the partial and total width arrays are dynamically updated and calculated from innermost (stable products) to outermost (parent particle) on a 100-point mass mesh from threshold up to a maximum energy. The first call requires typically from a few CPU seconds (for resonances with "depth" 1, such as e.g. the Delta resonance), up to a few CPU minutes for heavy  $N^*$  or  $\Delta^*$  resonances with a multitude of embedded decays. Subsequent calls are rapid, returning values by interpolation from the mesh calculated and stored on memory during the first call.

Mass sampling on the other hand takes place on line using the rejection method ([Appendix II](#)). Simultaneous two-product sampling is implemented (in cases of hadronic decays to two unstable particles), and whether one or two product masses are sampled the process is rapid, rendering it unnecessary to store memory-consuming arrays. This is also the case for dilepton-mass sampling (i.e. massive virtual photon) in Dalitz decays.

With mass-dependent total and partial widths available dynamic branching ratios may also be calculated, for the recognized decay modes as listed above (otherwise static B.R.'s are used), and decay modes may be sampled (see



[PData](#) function `pickChannel()`).

### III.b PParticle Class

A [particle](#) is a [Lorentz vector](#), together with a `particle_id` (`pid`) and a `weight`. The `pid` convention in the [PParticle](#) class is consistent with [GEANT3](#), except for the additional unstable particles of **Pluto<sup>++</sup>**. The `weight` is unity by default, unless explicitly set otherwise, and is updated self-consistently depending on the physics model of the interaction that produces a given particle. Composite particles made up of two (but no more) constituent particles may be defined, where the `pid` assignment follows the ansatz  $pid = pid1 * 1000 + pid2$  for the two constituent `pid`'s. The "addition" is used for this operation, intended for the creation of a quasi particle at the entrance channel from the interaction of a beam particle (1st constituent) with a target (2nd constituent). For composite particles the 4-vector is the sum of the constituent 4-vectors, and the weight is the product of the constituent weights (uncorrelated weights assumed).

Several constructors are available for instantiating particles, as well as functions to return physical observables such as the momentum, velocity- or Lorentz- vector, weight, rapidity, mass, angles, etc. Standard operations such as boosts, translations, rotations, etc. are inherited from the parent [TLorentzVector](#) class.

Particles participating in a simulation are instantiated in advance, and subsequently updated during the execution of an event loop. In this way unnecessary invocation of time-consuming constructors and destructors is avoided. This scheme is adhered to in general, for subsequent class objects as well, and it will become clearer in the examples of Section [IV](#). Particles are instantiated with the mass pole, but masses are reassigned automatically via sampling if appropriate, e.g. for unstable resonances, during the execution of an event loop.

### III.c PChannel Class

A channel is a single step in a reaction process, consisting of a parent, elementary or quasi-particle from a beam-target interaction, and its subsequent decay into a number of decay products via a specified decay mode. The [PChannel](#) default constructor requires as minimum input a pointer to an array of pointers to the parent and decay particles, and the number of decay particles (default two). Additional constructors are provided, adapted to facilitate multi-hadron thermal decay modes of quasi-particle fireballs (see [PFireball](#)). A channel instantiated with the minimum input arguments is treated by default as a N-body phase-space decay with isotropically distributed cm scattering angles, and masses either sampled from a Breit-Wigner distribution with fixed (static) width in the case of resonances, or fixed to the mass-pole value otherwise.

The calculation of dynamical (mass-dependent) observables is enabled, for channels matching those recognized by the code, via either additional input arguments, or member functions after instantiation by default. There are three relevant "physics-model" flags (suppressed by default): a) **BeamFlag**: beam-sampling option for [HADES](#) pion beams (i.e. momentum sampling from an empirical beam profile), b) **MassFlag**: mass sampling from dynamically calculated spectral functions (see Section [III.a](#)), and c) **AngleFlag**: cm scattering-angle sampling (if anisotropic). Activation (i.e. setting to 1) is ignored (i.e. flags are reset to 0) if the options are not applicable or a physics model is not available. The options may be changed collectively via the `setDefault(int i, int j, int k)`, or individually via the `setBeamFlag(int)`, `setMassFlag(int)`, and `setAngleFlag(int)` functions. The effect of activating the sampling options is the following:

- a. **BeamFlag**=1 has an effect only if the parent is a quasi particle (see discussion on [composite particles](#) in Section [III.b](#)), and an empirical beam profile is available for the beam. Currently, a beam profile is only available for  $\pi^\pm$  beams (Ref. [\[20\]](#)). Moreover, the parent particle must have been instantiated either via the kinetic energy constructor, or with three components of momentum of which only  $p_z$  may be non zero. In these cases the parent momentum is interpreted as the central beam momentum, around which sampling is performed. In all the other cases the beam flag is reset to zero, and the beam momentum is kept fixed (see also Section [AIII](#)).
- b. **MassFlag**=1 causes
  - i. resonance masses to be sampled from realistic spectral functions, as discussed in Section [III.a](#) in connection with the [PData](#) class. If the width cannot not be dynamically calculated (e.g. for hadronic decay modes with number of products>3, or other cases not covered in the discussion of Section [III.a](#)) and the static width from the data base is used instead, resonance

masses are sampled from a standard Breit-Wigner distribution with fixed (the static) width. Likewise, for stable particles, the mass is fixed (the mass pole from the data base). If a channel only consists of particles in either of the latter two categories, then the mass flag is reset to zero and mass-sampling options are subsequently ignored.

ii. dilepton masses for massive (virtual) photons produced in Dalitz decays, enumerated earlier in Section [II.b](#).

iii. the [default action](#) is invoked, and the flag to be reset to zero, in all the other cases.

c. **AngleFlag**=1 induces scattering-angle sampling for a [select number of channels](#) listed in Section [II.b](#), with anisotropic cm angular distributions for which empirical parametrizations are available to the code. Otherwise the flag is reset to zero and default option of isotropy is assumed.

Whereas the physics options are suppressed by the default channel constructor, they are activated by default if the channels are passed as arguments to a reaction, discussed in the following Section, which generally will be the case. Thus, in the majority of cases, the user has no need to explicitly set the physics options, provided the channels of interest participate as steps in a complete reaction chain, instantiated via the [PReaction](#) class. If, however, [channels](#) are employed as stand-alone objects outside of a [reaction](#), or other than the default physical models are being explored, then these options need to be explicitly set as needed. The various possibilities are further illustrated with examples in Section [IV](#).

Besides the few builtin angular distributions, external angular distributions, implemented as [TF1](#) and [TF2](#) objects, can be attached to a [PChannel](#) with function **setAngleFunction()**. This feature has been implemented by [Ingo Fröhlich](#) of Giessen University.

Following the assignment of product masses such that energy conservation is respected, the parent decay is induced. This is the principal function of the [PChannel](#) class, and it is followed by a boost of the product 4-vectors into the lab frame, and an update of their respective weights as required by the interaction model employed for the particular interaction that is being simulated.

### III.d PReaction Class

A [reaction](#) is a complete physical process, consisting of one or several steps ([PChannels](#)). The [PReaction](#) constructor requires as input a [PChannel](#)-type double pointer directed to an array of individual [PChannel](#)-type pointers, a character-string specifying a file name including directory but without a suffix, and the number of constituent channels (default is 2). Additional constructor arguments are flags of integer type specifying output, decay-mode, and vertex-calculation options. These are the following:

a. **f0**: Output options for the [ROOT](#) file (default 0):

A [ROOT](#) output file is always generated, named as specified by the character-string argument of the constructor followed by the suffix **".root"**. The resulting file contains information stored in a [TTree](#) named **"data"**.

**0**: Only the *tracked* (i.e. stable) particles are stored in the ROOT file.

**1**: All the particles are stored in the ROOT file. In particular, these are:

i. If the reaction begins with the decay of an elementary particle, that particle and all the subsequent products in the reaction chain, including unstable intermediate particles, are stored on file.

ii. If the entry channel involves a [composite](#) parent particle, as discussed in Section [III.b](#), but the beam and target are fixed (i.e. the [beam-sampling](#) option is unavailable), then that (fixed) composite particle and all the subsequent products, including unstable intermediate particles, are written on file.

iii. If, as in case (ii), the entry channel involves a composite parent particle but [beam-sampling](#) is also performed, then the beam, target, and all the subsequent products are written on file.

b. **f1:** Decay-mode options (default 0):

**0:** The channel physics defaults are enabled (if applicable), as [previously](#) discussed in Section [III.c](#).

**1:** The [PReaction](#) constructor does not reset the channel physics options, which are therefore frozen to what they had been set to when passed to the [PReaction](#) constructor. These options may still be changed manually by invoking corresponding [PChannel](#) member functions directly on the intended channels. Such changes have an effect in event loops subsequently executed via invocations of the [loop\(\)](#) function.

c. **f2:** Vertex-calculation options (default 0):

**0:** This option is off (no vertex calculation).

**1:** Production vertices are calculated for those particles that are written on file (depending on the [output option](#)). Depending on the [beam-sampling](#) option, the origin is considered to be the parent, or beam and target, vertex. This is also the case for the products of the [first](#) channel. For particles produced in subsequent channels the production vertex is calculated by adding straight-line segments successively, each of length obtained as the product of the parent vector velocity times a lifetime randomly sampled from an exponential decay-time distribution. An assumption of absence of any external magnetic fields is implicit.

d. **f3:** ASCII output options (default 0):

**0:** No ASCII output.

**1:** ASCII output files, formatted for input to [HGeant](#), are produced. The naming convention is: the character-string argument supplied in the constructor followed by the suffix "[<i>.evt](#)", where [<i>](#) is incremented with each invocation of the function [loop\(\)](#) as earlier for the trees. A new ASCII file per invocation of [loop\(\)](#) is created. Irrespective of the [output option](#), ASCII files contain **only tracked** particles, as defined in the related discussion. Due to the specific format requirement only a subset of the information contained in the [ROOT](#) output files is also included in the ASCII output (e.g. events that have failed a cut applied via a [filter](#) are excluded from the ASCII, but not the [ROOT](#) output files). The output also depends on the setting of the vertex-calculation flag f3. Invoked from the [PDdecayManager](#) class, a separate ASCII file is opened for each reaction channel processed.

**2:** Common ASCII output file for all reaction channels processed by a [PDdecayManager](#).

Once the reaction has been set up, geometrical and kinematical [filters](#) may be imposed. These are discussed in the [next section](#).

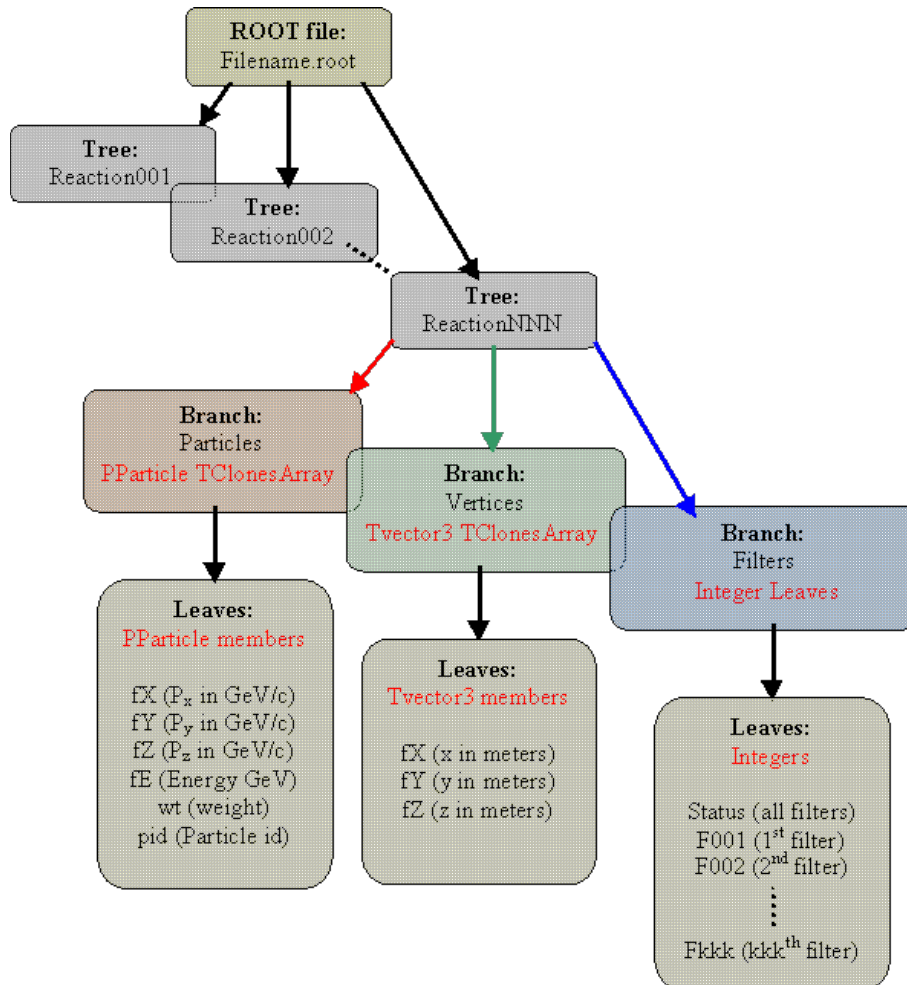
The [tree structures](#) in the standard [reaction](#) output contain [branches](#) that are set-up by the native [ROOT TClonesArray](#) class. This is the most efficient [ROOT](#) device both in terms of CPU time, as well as memory and disk-space requirements, of the various possibilities of setting up tree structures. This method is applicable only for the storage of identical fixed-size objects (clones) with elementary-type members (integer/real), such as [PReaction](#) "events". In particular, up to three [branches](#) may be generated per tree in the output file:

- a. A [branch](#) of [particles](#) is always present, at a minimum. This contains either *all* or the *tracked* particles, depending on the [output option](#).
- b. If the [vertex-calculation option](#) is enabled, then a [TVector3](#) (3-vector) [branch](#) is created, containing the production vertices.
- c. If filters apply, then a simple [branch](#) of integer entries is created, with as many leaves as the number of filters applied incremented by one. Success (0) or failure (1) is recorded per event per filter. The extra entry (named "**Status**") records the overall success (0) or failure (*i*) of an event, where *i* is the serial number of the filter that was failed *first*. Thus, the success or failure of an event can be determined just by checking the **Status** leaf of the filter [branch](#), whereas the individual filter entries contain detailed information.

The output-file structure is illustrated in Fig. [2](#). Output files can be browsed on-line with the [ROOT](#) browser ([TBrowser](#)).

**Figure 2** The ROOT output file tree structure, produced in executing an event loop.





Due to the flexibility of the [ROOT tree](#) structure [particles](#) can be retrieved directly from file, with the full functionality of the [PParticle](#)-class member functions instantly available. Moreover, the special features of the [TClonesArrays](#) method permit the retrieval of either entire objects, or selective members (e.g. integer or double type entries such as momenta, energies, and pid numbers in the case of particles), in the latter case saving both CPU time and memory space if the full-object information is unnecessary. The [PReaction](#) class effectively does the book keeping of a reaction process, whereas most of the physics content resides in the [PData](#) and [PChannel](#) classes. The [PReaction](#) member function [loop\(N\)](#) induces the execution of the simulation and generates **N** events. This function manages the sequence of decays induced by the [channels](#), updates the [particles](#) following each channel decay (or aborts the event loop if the [channel](#) function [decay](#) terminates with error status), calculates the production vertices if requested, and checks for success or failure if filters apply.

The [Print\(\)](#) member function displays information on the [particles](#), [channels](#), and [filters](#) comprising the reaction, together with all the selected options.

User selection functions and user analysis functions, implemented as Cint-compilable C++ functions, can be attached to

a reaction via the [setUserSelection\(int \(\\*\)\(PParticle\\*\)\)](#) and [setUserAnalysis\(int \(\\*\)\(PParticle\\*\\*\)\)](#) calls. These functions are called from within the event loop, and can be used to filter particles and/or whole events in the output.

These functions offer more

flexibility than the filters discussed in the next section.

### III.e PFilter Class

A [filter](#) object is instantiated by two arguments: a pointer to a [reaction](#), and a character string specifying explicitly the algebraic expression of the condition which is to be satisfied. The [PFilter](#) class implicitly invokes the [ROOT](#) class [TFormula](#) for the interpretation of the character string, and for transposing it to a mathematical formula. A variety of expressions are acceptable, inherited from the [TFormula](#) class, including trigonometric functions, exponentiation, and standard C++ logic (e.g. [&&](#) for [and](#) and [||](#) for [or](#)). Several additional [particle](#) observables are valid keywords in the

[PFilter](#) class. These are passed with the character expression and require an integer argument, namely the serial number of the particle which is to satisfy the specified constraint. The particle-counting convention must be consistent with that of the [PReaction](#) class, and can be displayed e.g. by utilizing the [PReaction](#) [Print](#) function prior to setting up the filters (see examples in Section [IV](#)).

In the following, which are valid [filter](#) keywords, the integer argument is the [particle](#) serial number in the [reaction](#):

1. **px(int)** : x-component of momentum (GeV/c)
2. **py(int)** : y-component of momentum (GeV/c)
3. **pz(int)** : z-component of momentum (GeV/c)
4. **ptot(int)** : norm of momentum vector (GeV/c)
5. **perp(int)** : normal component of momentum (in the xy plane, normal to the beam in the z-direction) (GeV/c)
6. **ener(int)** : energy (GeV)
7. **mass(int)** : mass (GeV/c<sup>2</sup>)
8. **thet(int)** : polar angle (rad)
9. **phi(int)** : azimuthal angle (rad)
10. **rapi(int)** : rapidity
11. **weig(int)** : weight
12. **pid(int)** : the particle id (in the [HGeant](#) convention, if applicable)

For additional keywords, the user is referred to the [TFormula](#) class. As an example of the syntax, the statement **"cos(thet(3))\*2>0.5 && (pz(5)<1. || pz(6)<pz(5))"** is a valid [filter](#) expression applying a polar-angle cut on particle number 3, and momentum z-component projection cuts on particles No. 5 and 6 (red denotes valid [formula](#) keywords, and green additional [filter](#) ones).

The [TFormula](#) class supports up to three variables and up to ten parameters. The [PFilter](#) class makes use only of the parameters, effectively setting up constant expressions with no variable dependence. To illustrate how [TFormula](#) interprets [PFilter](#) expressions, the string expression in the previous example would be transposed to the [formula](#) **"cos([0])\*2>0.5 && ([1]<1. || [2]<[1])"**, where [0-2] are [TFormula](#) parameters, namely *one parameter per combination of particle serial number and observable*. It follows that up to ten such combinations per [filter](#) are allowed. This does not pose a limitation since multiple [filters](#) may simultaneously be applied on any given [reaction](#), thus any number of constraints may be imposed.

Invocation of the [filter](#) constructor results in accessing the referenced [reaction](#), setting up or updating the dynamical array of [filter](#) pointers that is a member of the [reaction](#) class, and appending the current [filter](#) as the last entry in that array. The [filter](#) particles are subsequently matched with the [reaction](#) particles, and the filter is attached following that [channel](#) which produces the latest of the [filter particles](#). During an event loop the filter condition is checked for success or failure as soon as all the particles in the filter have been created.

Additional member functions of the [PFilter](#) class are **Suspend()** and **Resume()**, temporarily deactivating and reactivating a [filter](#), and **Print()**, displaying the condition imposed by, and the status of the filter (i.e. active or suspended) .

### III.f PDecayManager Class

The [Decay Manager](#) is a front end to [PReaction](#), for simulations that need to cover a whole set of possible reaction chains ("cocktails"), written and documented by Volker Hejny. It uses particle properties, including (static) branching ratios, from the [PData](#) class. Additional particles and decay branches may be included via the I/O member functions of the latter.

The standard way to set up a multi-step reaction is the following:

#### ● Declaration of PDecayManager

```
root> PDecayChannel *c;
root> PDecayManager *pdm = new PDecayManager;
```

#### ● Preparation of the decay modes

The user should assure that all the required channels are included in the simulation. This can be done by

switching on any of the available channels, either supplied by default, or previously loaded via [PData](#) I/O member functions by the user, e.g.:

```
root> pdm->SetDefault("w");           // include  $\omega$  decay modes
root> pdm->SetDefault("pi0");          // include  $\pi^0$  decay modes
```

## ● Preparation of the entrance channel

First the initial state is built (e.g. proton with 2.62 GeV kinetic energy on deuteron):

```
root> PParticle *p = new PParticle("p",2.62);           // proton beam
root> PParticle *d = new PParticle("d",0.,0.,0.,1.875613,45); // deuteron target
root> PParticle *s = new PParticle(*p + *d);           // composite
                                                         quasiparticle
```

Then the final state(s) are declared. Here only one channel is used, but more channels can be added. If 's' is a predefined particle and has a list of decay modes associated with it in [PData](#), 'c' can be omitted and the predefined list can be used. Otherwise the [PDecayChannel](#) constructor (supporting Class used by the [Decay Manager](#)) may be utilized, as illustrated in the first two lines of the sequence below. Once the final states have been specified, the reaction can be initialized:

```
root> c = new PDecayChannel;
root> c->AddChannel(1.0,"p","d","omega"); // include  $\omega$  decay modes
root> pdm->InitReaction(s,c);             // initialize the reaction
```

## ● Execution of the simulation

```
root> int n =
pdm->loop(10000,0,"pdomomega",0,0,0,0,0);
```

The arguments are:

### 1. Number of events: 10000

For reasons of normalization this number is the sum of event weights. The actual event number is returned by the **loop** function.

### 2. Weight flag: 0

If this is set, it acts as an additional normalization factor that adjusts the weight of the decay chain and all the product particles.

Otherwise, the number of events for one chain is calculated from the chain weight.

### 3. Reaction name: "pdomomega"

Used in setting up output file names.

### 4. Flags : f0, f1, f2, f3

These are the same as for [PReaction](#).

### 5. Random flag: if=0, process reactions sequentially, if=1, sample reactions in random order.

## ● Output

The [ROOT](#) output file consists of two [trees](#), one [info](#) and one [data tree](#). The former is filled for each decay chain and contains the following [branches](#):

1. *rStart*: event number in 'data', where the chain starts.
2. *rStop*: event number in 'data', where the chain stops.
3. *rEntries*: number of events in this chain.

4. *rWeight*: weight of this chain (only for information; for analysis it is included either in the number of events, or in the particle weight).

5. *rDescription*: a character array containing an ASCII description of the decay chain.

The [data tree](#) contains the [usual](#) particle and vertex entries (see [PReaction](#)). [Filters](#) are currently incompatible with the [Decay Manager](#) class, due to the fact that filter conditions use the specific particle serial number of [PChannel](#).

### III.g PFireball Class

The [PFireball](#) class is derived from [PParticle](#) and serves for the simulation of particle distributions typical for heavy-ion reactions, i.e. of thermal or quasi-thermal nature. Momentum distributions, angular distributions and multiplicity distributions (including impact-parameter sampling) are modelled. Several instances of [PFireball](#) can coexist to simulate multiple sources concurrently, namely all baryon and meson species produced in a heavy-ion collision, as well as sources of different temperature (e.g. spectators vs. participant). [PFireball](#) supports 2-component thermal distributions, longitudinal broadening, radial blast (Siemssen-Rasmussen model), directed and elliptic flow, and impact-parameter sampled multiplicities. These distributions are implemented in the helper class [PThermal](#). For details on how to use these classes, see the [sample macros](#).

## IV. Examples

The examples of this section together with the ROOT-generated HTML class-description [files](#) are intended as a user manual.

### IV.a Basics

Loading of the simulation module and some related simple [ROOT](#) operations are illustrated:

```
root [1] gSystem->Load("libPhysics");           // loads ROOT physics
                                                // module
root [2] gSystem->Load("~/user/pluto_directory/pluto.so"); // loads Pluto
root [3] double x1[]={1.,2.,3.}, x2[]={1.,2.,3.,4.}, e=4.0; // defines arrays of doubles
root [4] float y1[]={1.,2.,3.}, y2[]={1.,2.,3.,4.};       // defines arrays of floats
root [5] TVector3 v1(1.,2.,3.), v2(x1), v3(y1), v4(v1);     // identical 3-vectors
root [6] cout << " v11=" << v1.X() << " v22=" << v2.Y() << " v43=" << v4.Z()
<<endl;           // member function
v11=1 v22=2 v43=3 // invocation
root [7] TLorentzVector l1(1.,2.,3.,4.), l2(x2), l3(y2), l4(v1,e); // identical 4-vectors
l11=1 l22=2 l44=4 // member function
root [8] cout << " l11=" << l1.Px() << " l22=" << l2.Y() << " l44=" << l4.E() <<endl; // invocation
l11=1 l22=2 l44=4 // returned output
root [9] l1.Boost(v1); // boosts l1 along vector v1
root [10] l2.Boost( l3.BoostVector() ); // boosts l2 along
                                         // momentum of l3
root [11] double x=PUUtils::sampleFlat(), y=PUUtils::sampleBW(.77,.151); // random number generators
```

This sequence loads the [ROOT](#) physics module (including 3- and 4-vector algebra that are required) and the simulation package, sets up arrays of double and single precision reals, declares [3](#)- and [4](#)-vectors invoking various constructors, displays some methods for retrieving their coordinates, and implements boosts of a [4](#)-vector along a [3](#)-vector (velocity), where the function [BoostVector\(\)](#) returns the velocity component of a [4](#)-vector. Two cases of random number calls are also shown, from flat and Breit-Wigner distributions, where in the latter case the centroid and width are passed as arguments. Additional methods are available for operations such as retrieving the amplitude and transverse component of vectors, or the rapidity for Lorentz vectors, invoking translations and rotations etc, for which the user is referred to the native [ROOT TLorentzVector](#) class.

## IV.b Data Base

The following particles and decay modes are currently available in the permanent data base ([PData](#)):

```
root [1] listParticle()
```

pid	name	mass	width (GeV/c**2)
0	dummy	0.000000	stable
1	g	0.000000	stable
2	e+	0.000511	stable
3	e-	0.000511	stable
4	nu	0.000000	stable
5	mu+	0.105658	2.995918e-19
6	mu-	0.105658	2.995918e-19
7	pi0	0.134976	7.835860e-09
8	pi+	0.139570	2.528376e-17
9	pi-	0.139570	2.528376e-17
10	K0L	0.497672	1.273138e-17
11	K+	0.493677	5.314163e-17
12	K-	0.493677	5.314120e-17
13	n	0.939566	7.420656e-28
14	p	0.938272	stable
15	anti_p	0.938272	stable
16	K0S	0.497672	7.373274e-15
17	eta	0.547450	1.180000e-06
18	Lambda	1.115684	2.500806e-15
19	Sigma+	1.189370	8.237950e-15
20	Sigma0	1.192550	8.894759e-06
21	Sigma-	1.197436	4.450387e-15
22	Xi0	1.314900	2.269697e-15
23	Xi-	1.321320	4.015938e-15
24	Omega	1.672450	8.007448e-15
25	anti_n	0.939566	7.420656e-28
26	anti_Lambda	1.115684	2.500806e-15
27	anti_Sigma-	1.197436	8.237950e-15
28	anti_Sigma0	1.192550	8.894759e-06
29	anti_Sigma+	1.189370	4.450387e-15
30	anti_Xi0	1.314900	2.269697e-15
31	anti_Xi+	1.321320	4.015938e-15
32	anti_Omega+	1.672450	8.007448e-15



34	D0	1.232000	1.200000e-01
35	D++	1.232000	1.200000e-01
36	D+	1.232000	1.200000e-01
37	D-	1.232000	1.200000e-01
38	NP11+	1.440000	3.500000e-01
39	ND13+	1.520000	1.200000e-01
40	NS11+	1.535000	1.500000e-01
41	rho0	0.769900	1.507000e-01
42	rho+	0.769900	1.507000e-01
43	rho-	0.769900	1.507000e-01
45	d	1.875613	stable
46	t	2.809250	stable
47	alpha	3.727417	stable
49	He3	2.809230	stable
50	dimuon	0.211320	0.000000e+00
51	dilepton	0.001022	0.000000e+00
52	w	0.781940	8.430000e-03
53	eta'	0.957700	2.010000e-04
54	sigma	0.500000	6.000000e-01
55	phi	1.019413	4.430000e-03
56	DP330	1.600000	3.500000e-01
57	DP33++	1.600000	3.500000e-01
58	DP33+	1.600000	3.500000e-01
59	DP33-	1.600000	3.500000e-01
60	DS310	1.620000	1.500000e-01
61	DS31++	1.620000	1.500000e-01
62	DS31+	1.620000	1.500000e-01
63	DS31-	1.620000	1.500000e-01
64	NP110	1.440000	3.500000e-01
65	ND130	1.520000	1.200000e-01
66	NS110	1.535000	1.500000e-01
67	Jpsi	3.096880	8.700000e-05

```
root [2] listModes()
```

No.	decay mode	BR	code
0)	mu+ --> e+ + neutrino + neutrino	1.000000	4004002
1)	mu- --> e- + neutrino + neutrino	1.000000	4004003
2)	pi0 --> photon + photon	0.988000	1001

3)	$\pi^0 \rightarrow \text{dilepton} + \text{photon (Dalitz)}$	0.012000	1051
4)	$\pi^+ \rightarrow \mu^+ + \text{neutrino}$	1.000000	4005
5)	$\pi^- \rightarrow \mu^- + \text{neutrino}$	1.000000	4006
6)	$K^0 \text{ long} \rightarrow \pi^0 + \pi^0 + \pi^0$	0.211000	7007007
7)	$K^0 \text{ long} \rightarrow \pi^+ + \pi^- + \pi^0$	0.126000	7009008
8)	$K^0 \text{ long} \rightarrow \pi^+ + \mu^- + \text{neutrino}$	0.136000	4006008
9)	$K^0 \text{ long} \rightarrow \pi^- + \mu^+ + \text{neutrino}$	0.136000	4005009
10)	$K^0 \text{ long} \rightarrow \pi^+ + e^- + \text{neutrino}$	0.194000	4003008
11)	$K^0 \text{ long} \rightarrow \pi^- + e^+ + \text{neutrino}$	0.194000	4002009
12)	$K^+ \rightarrow \mu^+ + \text{neutrino}$	0.635000	4005
13)	$K^+ \rightarrow \pi^+ + \pi^0$	0.212000	9008
14)	$K^+ \rightarrow \pi^+ + \pi^+ + \pi^-$	0.056000	9008008
15)	$K^+ \rightarrow \pi^+ + \pi^0 + \pi^0$	0.017000	7007008
16)	$K^+ \rightarrow \pi^0 + \mu^+ + \text{neutrino}$	0.032000	4005007
17)	$K^+ \rightarrow \pi^0 + e^+ + \text{neutrino}$	0.048000	4002007
18)	$K^- \rightarrow \mu^- + \text{neutrino}$	0.635000	4006
19)	$K^- \rightarrow \pi^- + \pi^0$	0.212000	7009
20)	$K^- \rightarrow \pi^- + \pi^- + \pi^+$	0.056000	8009009
21)	$K^- \rightarrow \pi^- + \pi^0 + \pi^0$	0.017000	7007009
22)	$K^- \rightarrow \pi^0 + \mu^- + \text{neutrino}$	0.032000	4006007
23)	$K^- \rightarrow \pi^0 + e^- + \text{neutrino}$	0.048000	4003007
24)	$n \rightarrow p + e^- + \text{neutrino}$	1.000000	4003014
25)	$K^0 \text{ short} \rightarrow \pi^+ + \pi^-$	0.686000	9008
26)	$K^0 \text{ short} \rightarrow \pi^0 + \pi^0$	0.314000	7007
27)	$K^0 \text{ short} \rightarrow \pi^+ + \pi^- + \text{photon}$	0.002000	1009008
28)	$\eta \rightarrow \text{photon} + \text{photon}$	0.392000	1001
29)	$\eta \rightarrow \pi^0 + \pi^0 + \pi^0$	0.322000	7007007
30)	$\eta \rightarrow \pi^+ + \pi^- + \pi^0$	0.231000	8009007
31)	$\eta \rightarrow \pi^+ + \pi^- + \text{photon}$	0.048000	8009001
32)	$\eta \rightarrow \text{dilepton} + \text{photon (Dalitz)}$	0.004900	1051
33)	$\Lambda \rightarrow p + \pi^-$	0.639000	9014
34)	$\Lambda \rightarrow n + \pi^0$	0.358000	7013
35)	$\Lambda \rightarrow n + \text{photon}$	0.002000	1013
36)	$\text{anti-}n \rightarrow \text{anti-}p + e^+ + \text{neutrino}$	1.000000	4002015
37)	$\Delta^0 \rightarrow p + \pi^-$	0.330000	9014
38)	$\Delta^0 \rightarrow n + \pi^0$	0.660000	7013
39)	$\Delta^0 \rightarrow n + \text{photon}$	0.005500	1013

40)	Delta0 --> dilepton + n (Dalitz)	0.004500	13051
41)	Delta++ --> p + pi+	1.000000	8014
42)	Delta+ --> p + pi0	0.660000	7014
43)	Delta+ --> n + pi+	0.330000	8013
44)	Delta+ --> p + photon	0.005500	1014
45)	Delta+ --> dilepton + p (Dalitz)	0.004500	14051
46)	Delta- --> n + pi-	1.000000	9013
47)	N*(1440)+ --> p + pi0	0.216667	7014
48)	N*(1440)+ --> n + pi+	0.433333	8013
49)	N*(1440)+ --> Delta++ + pi-	0.125000	9035
50)	N*(1440)+ --> Delta+ + pi0	0.083333	7036
51)	N*(1440)+ --> Delta0 + pi+	0.041667	8034
52)	N*(1440)+ --> p + rho0	0.008195	41014
53)	N*(1440)+ --> n + rho+	0.016390	42013
54)	N*(1440)+ --> p + pi0 + pi0	0.025000	7007014
55)	N*(1440)+ --> p + pi+ + pi-	0.050000	9008014
56)	N*(1440)+ --> p + photon	0.000415	1014
57)	N*(1520)+ --> p + pi0	0.183333	7014
58)	N*(1520)+ --> n + pi+	0.366667	8013
59)	N*(1520)+ --> Delta++ + pi-	0.100000	9035
60)	N*(1520)+ --> Delta+ + pi0	0.066667	7036
61)	N*(1520)+ --> Delta0 + pi+	0.033333	8034
62)	N*(1520)+ --> p + rho0	0.066667	41014
63)	N*(1520)+ --> n + rho+	0.133333	42013
64)	N*(1520)+ --> p + pi0 + pi0	0.014967	7007014
65)	N*(1520)+ --> p + pi+ + pi-	0.029933	9008014
66)	N*(1520)+ --> p + photon	0.005100	1014
69)	N*(1535)+ --> p + eta	0.387500	17014
70)	N*(1535)+ --> Delta++ + pi-	0.005000	9035
71)	N*(1535)+ --> Delta+ + pi0	0.003333	7036
72)	N*(1535)+ --> Delta0 + pi+	0.001667	8034
73)	N*(1535)+ --> p + rho0	0.013333	41014
74)	N*(1535)+ --> n + rho+	0.026667	42013
75)	N*(1535)+ --> p + pi0 + pi0	0.010000	7007014
76)	N*(1535)+ --> p + pi+ + pi-	0.020000	9008014
77)	N*(1535)+ --> N*(1440)+ + pi0	0.023333	7038
78)	N*(1535)+ --> N*(1440)0 + pi+	0.046667	8064

79)	$N^*(1535)^+ \rightarrow p + \text{photon}$	0.002500	1014
80)	$\rho^0 \rightarrow \pi^+ + \pi^-$	0.999955	9008
81)	$\rho^0 \rightarrow e^+ + e^-$	0.000045	3002
82)	$\rho^+ \rightarrow \pi^+ + \pi^0$	1.000000	7008
83)	$\rho^- \rightarrow \pi^- + \pi^0$	1.000000	7009
84)	$\text{dilepton} \rightarrow e^+ + e^-$	1.000000	3002
85)	$\omega \rightarrow \pi^+ + \pi^- + \pi^0$	0.888000	7009008
86)	$\omega \rightarrow \pi^0 + \text{photon}$	0.085000	1007
87)	$\omega \rightarrow \pi^+ + \pi^-$	0.026339	9008
88)	$\omega \rightarrow \text{dilepton} + \pi^0 \text{ (Dalitz)}$	0.000590	7051
89)	$\omega \rightarrow e^+ + e^-$	0.000071	3002
90)	$\eta' \rightarrow \eta + \pi^- + \pi^+$	0.438000	9008017
91)	$\eta' \rightarrow \rho^0 + \text{photon}$	0.302000	1041
92)	$\eta' \rightarrow \eta + \pi^0 + \pi^0$	0.207000	7007017
93)	$\eta' \rightarrow \omega + \text{photon}$	0.030100	1052
94)	$\eta' \rightarrow \text{photon} + \text{photon}$	0.021100	1001
95)	$\eta' \rightarrow \pi^0 + \pi^0 + \pi^0$	0.001540	7007007
96)	$\eta' \rightarrow \text{dilepton} + \text{photon (Dalitz)}$	0.000260	1051
97)	$\phi \rightarrow K^+ + K^-$	0.491000	12011
98)	$\phi \rightarrow K_{0L} + K_{0S}$	0.341000	16010
99)	$\phi \rightarrow \pi^+ + \pi^- + \pi^0$	0.153790	7009008
100)	$\phi \rightarrow \eta + \text{photon}$	0.012600	1017
101)	$\phi \rightarrow \pi^0 + \text{photon}$	0.001310	1007
102)	$\phi \rightarrow e^+ + e^-$	0.000300	3002
103)	$\Delta(1600)^0 \rightarrow p + \pi^-$	0.058333	9014
104)	$\Delta(1600)^0 \rightarrow n + \pi^0$	0.116667	7013
105)	$\Delta(1600)^0 \rightarrow \Delta^+ + \pi^-$	0.293333	9036
106)	$\Delta(1600)^0 \rightarrow \Delta^0 + \pi^0$	0.036667	7034
107)	$\Delta(1600)^0 \rightarrow \Delta^- + \pi^+$	0.220000	8037
108)	$\Delta(1600)^0 \rightarrow p + \rho^-$	0.075000	43014
109)	$\Delta(1600)^0 \rightarrow n + \rho^0$	0.150000	41013
110)	$\Delta(1600)^0 \rightarrow N(1440)^+ + \rho^-$	0.016633	43038
111)	$\Delta(1600)^0 \rightarrow N(1440)^0 + \rho^0$	0.033267	41064
112)	$\Delta(1600)^0 \rightarrow n + \text{photon}$	0.000100	1013
113)	$\Delta(1600)^{++} \rightarrow p + \pi^+$	0.175000	8014
114)	$\Delta(1600)^{++} \rightarrow \Delta^{++} + \pi^0$	0.330000	7035
115)	$\Delta(1600)^{++} \rightarrow \Delta^+ + \pi^+$	0.220000	8036

116)	Delta(1600)++ --> p + rho+	0.225000	42014
117)	Delta(1600)++ --> N(1440)+ + rho+	0.050000	42038
118)	Delta(1600)+ --> p + pi0	0.116667	7014
119)	Delta(1600)+ --> n + pi+	0.058333	8013
120)	Delta(1600)+ --> Delta++ + pi-	0.220000	9035
121)	Delta(1600)+ --> Delta+ + pi0	0.036667	7036
122)	Delta(1600)+ --> Delta0 + pi+	0.293333	8034
123)	Delta(1600)+ --> p + rho0	0.150000	41014
124)	Delta(1600)+ --> n + rho+	0.075000	42013
125)	Delta(1600)+ --> N(1440)+ + rho0	0.033267	41038
126)	Delta(1600)+ --> N(1440)0 + rho+	0.016633	42064
127)	Delta(1600)+ --> p + photon	0.000100	1014
128)	Delta(1600)- --> n + pi-	0.175000	9013
129)	Delta(1600)- --> Delta0 + pi-	0.220000	9034
130)	Delta(1600)- --> Delta- + pi0	0.330000	7037
131)	Delta(1600)- --> n + rho-	0.225000	43013
132)	Delta(1600)- --> N(1440)0 + rho-	0.050000	43064
133)	Delta(1620)0 --> p + pi-	0.083333	9014
134)	Delta(1620)0 --> n + pi0	0.166667	7013
135)	Delta(1620)0 --> Delta+ + pi-	0.314507	9036
136)	Delta(1620)0 --> Delta0 + pi0	0.039313	7034
137)	Delta(1620)0 --> Delta- + pi+	0.235880	8037
138)	Delta(1620)0 --> p + rho-	0.053333	43014
139)	Delta(1620)0 --> n + rho0	0.106667	41013
140)	Delta(1620)0 --> n + photon	0.000300	1013
141)	Delta(1620)++ --> p + pi+	0.250000	8014
142)	Delta(1620)++ --> Delta++ + pi0	0.353820	7035
143)	Delta(1620)++ --> Delta+ + pi+	0.235880	8036
144)	Delta(1620)++ --> p + rho+	0.160000	42014
145)	Delta(1620)+ --> p + pi0	0.166667	7014
146)	Delta(1620)+ --> n + pi+	0.083333	8013
147)	Delta(1620)+ --> Delta++ + pi-	0.235880	9035
148)	Delta(1620)+ --> Delta+ + pi0	0.039313	7036
149)	Delta(1620)+ --> Delta0 + pi+	0.314507	8034
150)	Delta(1620)+ --> p + rho0	0.106667	41014
151)	Delta(1620)+ --> n + rho+	0.053333	42013
152)	Delta(1620)+ --> p + photon	0.000300	1014



153)	$\Delta(1620)^- \rightarrow n + \pi^-$	0.250000	9013
154)	$\Delta(1620)^- \rightarrow \Delta^0 + \pi^-$	0.235880	9034
155)	$\Delta(1620)^- \rightarrow \Delta^- + \pi^0$	0.353820	7037
156)	$\Delta(1620)^- \rightarrow n + \rho^-$	0.160000	43013
157)	$N^*(1440)^0 \rightarrow p + \pi^-$	0.216667	9014
158)	$N^*(1440)^0 \rightarrow n + \pi^0$	0.433333	7013
159)	$N^*(1440)^0 \rightarrow \Delta^+ + \pi^-$	0.125000	9036
160)	$N^*(1440)^0 \rightarrow \Delta^0 + \pi^0$	0.083333	7034
161)	$N^*(1440)^0 \rightarrow \Delta^- + \pi^+$	0.041667	8037
162)	$N^*(1440)^0 \rightarrow p + \rho^-$	0.008195	43014
163)	$N^*(1440)^0 \rightarrow n + \rho^0$	0.016390	41013
164)	$N^*(1440)^0 \rightarrow n + \pi^+ + \pi^-$	0.050000	9008013
165)	$N^*(1440)^0 \rightarrow n + \pi^0 + \pi^0$	0.025000	7007013
166)	$N^*(1440)^0 \rightarrow n + \text{photon}$	0.000415	1013
167)	$N^*(1520)^0 \rightarrow p + \pi^-$	0.366667	9014
168)	$N^*(1520)^0 \rightarrow n + \pi^0$	0.183333	7013
169)	$N^*(1520)^0 \rightarrow \Delta^+ + \pi^-$	0.033333	9036
170)	$N^*(1520)^0 \rightarrow \Delta^0 + \pi^0$	0.066667	7034
171)	$N^*(1520)^0 \rightarrow \Delta^- + \pi^+$	0.100000	8037
172)	$N^*(1520)^0 \rightarrow p + \rho^-$	0.133333	43014
173)	$N^*(1520)^0 \rightarrow n + \rho^0$	0.066667	41013
174)	$N^*(1520)^0 \rightarrow n + \pi^+ + \pi^-$	0.029933	9008013
175)	$N^*(1520)^0 \rightarrow n + \pi^0 + \pi^0$	0.014967	7007013
176)	$N^*(1520)^0 \rightarrow n + \text{photon}$	0.005100	1013
177)	$N^*(1535)^0 \rightarrow p + \pi^-$	0.153333	9014
178)	$N^*(1535)^0 \rightarrow n + \pi^0$	0.306667	7013
179)	$N^*(1535)^0 \rightarrow n + \eta$	0.387500	17013
180)	$N^*(1535)^0 \rightarrow \Delta^+ + \pi^-$	0.005000	9036
181)	$N^*(1535)^0 \rightarrow \Delta^0 + \pi^0$	0.003333	7034
182)	$N^*(1535)^0 \rightarrow \Delta^- + \pi^+$	0.001667	8037
183)	$N^*(1535)^0 \rightarrow p + \rho^-$	0.013333	43014
184)	$N^*(1535)^0 \rightarrow n + \rho^0$	0.026667	41013
185)	$N^*(1535)^0 \rightarrow n + \pi^+ + \pi^-$	0.020000	9008013
186)	$N^*(1535)^0 \rightarrow n + \pi^0 + \pi^0$	0.010000	7007013
187)	$N^*(1535)^0 \rightarrow N^*(1440)^+ + \pi^-$	0.023333	9038
188)	$N^*(1535)^0 \rightarrow N^*(1440)^0 + \pi^0$	0.046667	7064
189)	$N^*(1535)^0 \rightarrow n + \text{photon}$	0.002500	1013

The properties of any one of the available particles may be viewed in more detail, by passing the pid or code name of that particle as argument to **listParticle**:

```
root [3] listParticle(38)                                     // listParticle("NP11+") is
                                                             equivalent

Code name:      NP11+

pid:            38

Mass pole:      1.440000 (GeV/c**2)

Vacuum width:  0.350000 (GeV/c**2)

Charge:         1

Baryon No:      1

(Parity)Spin:  +1/2

Isospin:        1/2

This particle decays via the following 10 decay mode(s):

No.              decay mode                                BR           code
47)  N*(1440)+ --> p + pi0                                0.216667      7014
48)  N*(1440)+ --> n + pi+                                0.433333      8013
49)  N*(1440)+ --> Delta++ + pi-                          0.125000      9035
50)  N*(1440)+ --> Delta+ + pi0                           0.083333      7036
51)  N*(1440)+ --> Delta0 + pi+                           0.041667      8034
52)  N*(1440)+ --> p + rho0                               0.008195      41014
53)  N*(1440)+ --> n + rho+                               0.016390      42013
54)  N*(1440)+ --> p + pi0 + pi0                         0.025000      7007014
55)  N*(1440)+ --> p + pi+ + pi-                         0.050000      9008014
56)  N*(1440)+ --> p + photon                            0.000415      1014
```

The data have been taken from Ref. [22], observing the following additional rules:

1. Only the dominant modes, which are relevant for HADES physics, have been included.
2. If a range of branching ratios is cited, then a rough average is assigned to the relevant mode.
3. Unitarity is assumed, i.e. the decay strength is exhausted via the included decay modes; alternatively phrased, the sum of branching ratios of the available decay modes for any unstable particle in the data base is unity.
4. Each possible combination of product charges (i.e. isospin projections) is included as a separate decay mode, with branching ratio scaled by the proper isospin weight factor. Tools are provided for the calculation of isospin factors (Clebsch-Gordan and Racah coefficients for the addition of two and three angular momenta respectively; see functions **cgc()** and **racah()** in **PUtils**, and **getIW()** in **PParticle**).

Additional particles, up to a total of 999 (permanent and user-defined combined) are supported and may be loaded externally, either interactively, or from a properly formatted user ASCII file (recommended), provided the above rules are adhered to. The following example file (included in the distribution as **missing\_Delta.dat**) illustrates the format. The data represent a "missing"  $\Delta$  excitation, with properties as predicted in Ref. [23], and decay modes assumed identical to other known  $\Delta$  excitations with identical spin and isospin:

The first line of input is the number of external particles to be loaded, followed by data pertaining to each input particle. Note that each charge (isospin) state is treated as a distinct particle. For each particle, one line of input is required containing the particle id, name (continuous string of up to thirty characters without space), mass pole and vacuum width (GeV), charge, 2 x isospin, parity (0 if irrelevant), 2 x spin, number of decay modes, baryon number, meson flag (0 or 1), and lepton number (integers). Allowed pid assignments are any non-occupied integers up to 999. Permanent data-base particles cannot be overridden, neither can user-defined data without first clearing them (using the **clearData** command). The default pid assignment is zero (recommended), as used in the example file above. It induces the code to automatically assign as pid number that of the last particle already loaded in the data base, incremented by unity. Note that the spin and isospin quantum numbers are doubled, in order that integers be used.

Following the required line of input per particle, as many lines as the "number of decay modes" specified there must follow. Each of these must contain the static branching ratio for that mode (observing the rules laid out earlier with respect to isospin-weight factors), product code (see below), and description (string up to 100 characters long enclosed in quotes, with all characters including spaces allowed).

The product code nomenclature, for  $N$  decay products, is  $\text{pid}_N \cdot 10^{3(N-1)} + \text{pid}_{N-1} \cdot 10^{3(N-2)} + \dots + \text{pid}_1$ , which is stored in the data base as a character string (not integer) in order to avoid the implicit length limitation of integers (translating into number of products). This convention sets the limit to 999 supported particles.

A second decay-product convention is also supported, adopted to facilitate multi-hadron thermal decays of quasi-particle fireballs (see [PFireball](#) class). This is intended for use with corresponding alternative [PChannel](#) constructors. A file (**fireball\_example.dat**) is included in the distribution to illustrate a user-prepared data file utilizing the alternative decay-product nomenclature. Multi-hadron thermal decays typically involve decay modes with large numbers of identical hadrons as products, for which the previous ansatz is cumbersome (e.g. for seventy zero-charge pions **70\*pi0** is the alternative format, which is preferable to the earlier ansatz).

Returning to the example of the "missing"  $\Delta$  resonance, the file is loaded as follows:

```
root [4] loadData("missing_Delta.dat")
root [5] listParticle()

pid    name          mass          width (GeV/c**2)
...
67     Jpsi           3.096880     8.700000e-05
68     DP334++        1.985000     3.000000e-01
69     DP334+         1.985000     3.000000e-01
70     DP3340         1.985000     3.000000e-01
71     DP334-         1.985000     3.000000e-01

root [6] listParticle("DP334+")

Code name:    DP334+
pid:          69
Mass pole:    1.985000 (GeV/c**2)
Vacuum width: 0.300000 (GeV/c**2)
Charge:       1
Baryon No:    1
(Parity)Spin: +3/2
Isospin:      3/2
This particle decays via the following 10 decay mode(s):

No.          decay mode          BR          code
```

195)	Delta(1985)+ --> p + pi0	0.116671	7014
196)	Delta(1985)+ --> n + pi+	0.058331	8013
197)	Delta(1985)+ --> Delta++ + pi-	0.220002	9035
198)	Delta(1985)+ --> Delta+ + pi0	0.036670	7036
199)	Delta(1985)+ --> Delta0 + pi+	0.293333	8034
200)	Delta(1985)+ --> p + rho0	0.150002	41014
201)	Delta(1985)+ --> n + rho+	0.075001	42013
202)	Delta(1985)+ --> N(1440)+ + rho0	0.033260	41038
203)	Delta(1985)+ --> N(1440)0 + rho+	0.016630	42064
204)	Delta(1985)+ --> p + photon	0.000100	1014

Note that due to the implicit rule of unitarity, only the relative ratios of branching ratios for a given particle are relevant and not their absolute value, since upon loading, user-defined particle branching ratios are renormalized to unity sum.

While loading external data, a number of arrays are reset dynamically in a self-consistent way. Though internal indices and ordering of arrays may change, these changes do not affect the execution of the simulation so long as member functions provided for the task of retrieving indices are utilized. The most important indices are the particle id, and decay mode serial number. It is recommended that the particle name be used as argument, since it is fixed, whereas the pid number may change depending on the order in which external data were loaded. Likewise, though the decay-mode index may change depending on the order of loading user-defined particles and pid assignments, the functions **getPosition("particle\_name")** and **NChannels("particle\_name")** will return the serial number of the first occurring decay mode of the referenced particle (-1 if none), and the number of decay modes, respectively. Once loaded, non-permanent data are treated identically with permanent data, except that the former may be cleared.

The use of some [PData](#) functions is demonstrated below:

```

root [6] PData::getPosition("DP334+")           // get 1st decay-mode index
(int)195
root [7] PData::NChannels("DP334+")             // get number of decay modes
(int)10
root [8] PData::LMass("DP334+")                 // particle mass threshold (GeV/c^2)
(double)1.385000000000000023e+00
root [9] PData::Mode(200)                      // retrieve info for mode No 200 (see previous
                                                // table)
root [10] int *i=PData::intCache()              // address of integer cache where info was
                                                // stored
root [11] i[0]                                 // number of decay products for mode No 200
(int)2
root [12] i[1]                                 // 1st product pid for mode No 200
(int)14
root [13] i[2]                                 // 2nd product pid for mode No 200
(int)41
root [14] PData::Modes("DP334+")               // get full decay-mode info for particle
                                                // number of decay modes: subsequently for
                                                // i[1]...number of products and pids as with
                                                // Mode(), for all modes
root [15] i[0]
(int)10
root [16] PData::getDepth("DP334+")            // number of nested (known) decays (discussed
(int)4                                           below)
root [17] PData::getDepth("DP334+",1)          // number of nested (known) hadronic decays
(int)3                                           (see below)

```

```

root [18] PData::getEmin(202)           // kinematical threshold for decay-mode No 202
                                         (GeV)
(double)2.209900000000000020e+00
root [19] PData::Width("DP334+")       // static width (GeV)
(double)2.99999999999999989e-01
root [20] PData::Mass("DP334+")        // mass pole (GeV/c2)
(double)1.985000000000000010e+00
root [21] PData::Width("DP334+",2.1)   // mass-dependent width for m=2.1 GeV/c2
(double)3.17560246555717463e-01
root [22] PData::sampleM("DP334+")    // random sampling from a relativistic Breit-
                                         Wigner distribution with mass-dependent width
(double)1.84764157850924438e+00
root [23] PData::sampleM("DP334+")
(double)2.08315448311922990e+00
                                         // mass-dependent branching ratio for mode No
root [24] PData::getBR(202,2.1)        202; zero returned since below threshold (see
                                         line 18)
(double)0.000000000000000000e+00
root [25] PData::getBR(202,2.3)        // repeat, with mass above kinematical threshold;
                                         unitarity respected for all masses
(double)8.30789109195612628e-03
                                         // pick a decay mode, sampling from the mass-
root [26] PData::pickChannel("DP334+",2.1) dependent branching ratios of kinematically open
                                         modes
                                         // decay-mode index returned, as in previous
(int)196                                table
                                         // as above, but return info at integer array *i, in a
root [27] PData::pickChannel("DP334+",2.1,i) format identical to that of the function Mode()
                                         shown earlier

```

The function **getDepth()** examines whether the decay products are unstable, and if they are, counts the number of [known](#) nested decay modes as discussed in Section [III.a](#). By default the total number of known embedded decays are return, whereas with flag=1 the known nested hadronic modes only are counted. The purpose of the latter is the following: Mass-dependent total widths are calculated only for those particles that have at least one known hadronic mode available in the data base. For particles with only semi-leptonic or leptonic modes in the data base, although the known decay widths and related branching ratios are calculated, mass dependent total widths are not calculated but rather the static total widths are used. This is so since in the latter case the mass dependence of the total width is negligible. Thus, for example, a mass-dependent total width is calculated for the  $D_s$  meson (with hadronic decay modes), and subsequently used for sampling its mass from a proper Breit-Wigner distribution with mass-dependence in the width (see Fig [5b](#)), but not so e.g. for the  $\pi_1$ , whose mass is not sampled, although the mass-dependent decay width and branching ratio for its Dalitz-decay mode is calculated.

The function **Mode(index)**, with a decay-mode index as argument, returns information on the referenced mode. The information is stored either in a local integer-cache array, whose address may be retrieved as shown with the function **intCache()**, or in an array (sufficiently long) whose address is passed to **Mode()** as a second argument. The first entry of the integer-cache array contains the number of decay products for the referenced mode, followed by the pid numbers of these products. Likewise, the function **Modes(pid)**, where the argument in this case is the particle id, or its name as a character string, returns the full information of all the particle's decay modes. For the latter, the first entry of the integer cache contains the number of decay modes, followed by the number of decay products for the first mode and product id's, and so on until the last product.

The function **getDepth()** initializes the process of resetting dynamical arrays, and is implicitly called upon loading user-defined data on the first call of any function calculating dynamic (mass-dependent) observables. It compares the referenced particle's decay modes with known channels, keeps counters, sets up flags for the calculation of partial and total widths, and recursively calls itself for each decay product of the referenced particle until stable products are reached, in order to count the nested decays. This is done only once, i.e. nested particles for which widths have been previously calculated are skipped. On subsequent calls the "depth" is retrieved from an internal dynamical array, where it is stored on the first call.



Likewise, the first time a dynamical function is called (e.g. `Width()` with the pid and a mass as arguments), the total and partial (i.e. for each known decay mode) mass-dependent widths are calculated on a 100-point mesh from the mass threshold up to twelve times the full (static) width above the mass pole. This is done observing unitarity, i.e. normalizing the sum of all the mass-dependent branching ratios for those modes that are accessible kinematically to unity, as well as normalizing distribution-function integrals to unity. For instance, in a decay mode such as that with index 202 above, where a Roper resonance and a  $\rho$  meson are produced, the corresponding integral that is normalized to unity is the convolution of the respective relativistic Breit-Wigner distributions with mass-dependent width times the center-of-mass momentum of the resonances, which is the effective cutoff factor (see Section [A1.a](#)). Unitarity is demonstrated below, in summing the mass-dependent branching ratios for all the decay modes of the example "missing"  $\Delta$  resonance, for a fixed mass ( $2.3 \text{ GeV}/c^2$ ):

```
root [39] double sum=0.
root [42] int j, j1=PData::getPosition("DP334+");
root [42] int j2=j1+PData::NChannels("DP334+");
root [45] for (j=j1;j<j2;++j)
sum+=PData::getBR(j,2.3);
root [46] sum
(double)1.0000000000000000e+00
```

The first call in the loop effectively takes up most of the CPU. This typically ranges from a few seconds for calculating the width of relatively simple resonances with no embedded decays (e.g. the  $\Delta(1232)$  resonance) to a few minutes for complex heavy resonances, as in the present example of the "missing"  $\Delta$  resonance. On return from the first call total and partial widths are calculated on a mesh as discussed earlier and stored in dynamical matrices. Subsequent calls are rapid, returning values by interpolation from the stored matrices. The methods of calculating widths in `PData` were introduced in Section [III.a](#), and are further elaborated in Section [A1.a](#).

The functions `getBR()` and `pickChannel()` are complementary. For those modes that the code calculates mass-dependent partial and total widths, the former returns either the mass-dependent branching ratio of the referenced decay mode if a mass argument is included, or the static branching ratio from the data base if no mass argument is given. If the referenced decay mode is not among those for which the code calculates widths, then the static branching ratio is returned. In both cases, if a mass argument is given which falls below the energy threshold for that decay mode, zero is returned. The function `pickChannel()` with a pid, or particle name as first argument, and a mass as the second argument, first calculates the branching ratios for all the referenced particle's decay modes and the given mass, and subsequently picks randomly, weighed properly, a decay channel. It returns this information either as decay-channel index, in the nomenclature adopted in the data base of `PData`, or in format identical to that of the function `Mode()`, namely the number of products and their pid's in an integer-cache array.

Mass sampling is rapid, even for complicated cases such as the present example. The rejection method is used, found to be far more efficient than native ROOT-supplied methods. In the case of masses, sampling is achieved on the fly, without resorting to storing arrays on memory as with total and partial widths. The simultaneous sampling of two unstable-hadron masses is demonstrated, for Roper-resonance and  $\rho$ -meson production in the decay of the "missing"  $\Delta$  resonance considered here, via the decay mode with index No. 202. The simulation is contained in the macro shown below (included in the distribution as macro `mass_sampling.C`), the execution of which results in the output also shown in the following table, and the histogram of Fig [3](#).

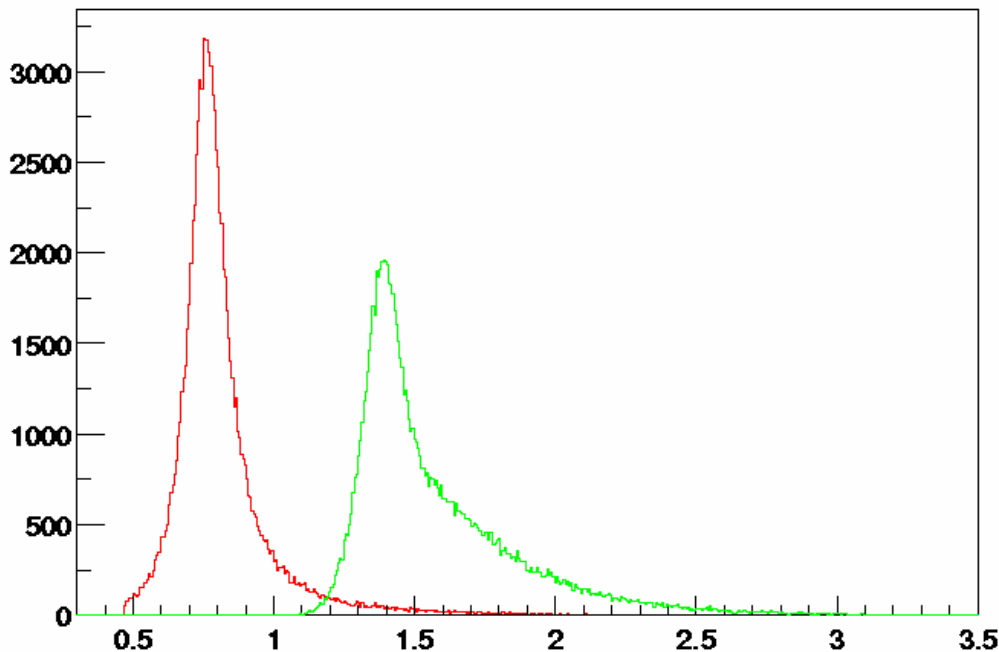
```
{
TH1F h1("h1","NP11+",500,0.3,3.5); // macro mass_sampling.C
TH1F h2("h2","rho0",500,0.3,3.5); // set up histograms
h1->SetLineColor(3); // Roper in green
h2->SetLineColor(2); // rho in red
double m[2]; // product-mass array
double ecm, emin=PData::getEmin(222); // cm energy variable, threshold for mode No 202
TStopwatch timer; // set up a timer
timer.Start(); // start timing
for (int j=0;j<100000;++j) { // 100k event loop
ecm=emin+2.*PUtls::sampleFlat(); // pick a random valid cm parent energy
PData::sampleM(ecm,222,m); // sample product masses for mode No 202
```

```

h1->Fill(m[0]);           // fill Roper-mass histogram
h2->Fill(m[1]);           // fill rho-mass histogram
}                          // loop completed
timer.Stop();             // stop timing
printf("CPU time=%f\n",timer.CpuTime()); // display CPU time
h2->Draw();                // draw histograms
h1->Draw("same");
}
root [47] .x test.C       // execute the macro
CPU time=251.240000      // on-screen output

```

**Figure 3** The simultaneous sampling of the Roper resonance (in green) and rho meson (in red) is illustrated, for the decay mode with index No 202 of a "missing"  $\Delta$  resonance, as discussed in the text, and 100k events. The x-axis is the mass variable ( $\text{GeV}/c^2$ ), and the histogram is produced by executing the macro above. The execution time (CPU) on a 200 MHz Linux PC was 251.24 sec.



#### IV.c Particles

An example of declaring a proton is shown, passing the kinetic energy (GeV) as argument (with the momentum assumed along the z axis). The only argument that is required in the particle constructor is either the pid or name, in the conventions of the resident data base of the [PData](#) class. If no other arguments are supplied, a "default" particle is created with zero momentum and mass equal to its energy. The energy assigned by the constructor is the mass pole. Several additional constructors are available.

Some [PParticle](#) member functions are demonstrated. More are inherited from the native [ROOT TLorentzVector](#) parent class. The weight is unity by default, unless another value is specified in the constructor. Weights are updated in the course of executing the simulation, e.g. accounting for branching ratios, kinematical factors, and interaction models.

```

root [1] PParticle p("p",2.); // proton with kinetic energy (GeV)
root [2] p.Print();           // print member function
p (0.000000,0.000000,2.784437;2.938272) wt = 1.000000, pid =
14
root [3] double m=p.M();      // the mass
root [4] double r=p.Rapidity(); // the rapidity

```

```

root [5] double th=p.Theta(); // the polar angle (rad)
root [6] double ph=p.Phi(); // the azimuthal angle (rad)
root [7] TVector3 v=p.Vect(); // the 3-vector (momentum)
                                component
root [8] double pt=p.Perp() // normal momentum component
root [9] int id=p.ID(); // the particle id
char *name=p.Name(); // the particle name

```

The concept of "composite" quasi-particles was first [presented](#) in Section [III.c](#), to be used as the "parent" particle of the entry channel. An example of a such a particle is illustrated next:

```

root [1] PParticle pi("pi-",0,0,1); // pi- with 1 GeV/c momentum along the z-axis
root [2] PParticle p("p"); // proton at rest (default constructor)
root [3] PParticle q=pi+p; // composite particle: beam + target (in this order)
root [4] q.Print(); // displays info
quasi-particle (0.000000,0.000000,1.000000;1.947965) wt = 1.000000, pid1 = 9,
pid2 = 14
root [5] q.ID() // composite particle pid convention: pid2*1000 +
(int)14009 pid1
// quasi-particle pid

```

#### IV.d Channels

In the framework of **Pluto++**, a [channel](#) is a step in a reaction process, consisting of a parent, its decay, and the decay products. The use of the [PChannel](#) class is illustrated in the decay of the composite particle **q** of the previous example into a new proton and a negative pion:

```

root [6] PParticle pi2("pi-"); // new pi-
root [7] PParticle p2("p"); // new proton
root [8] *s[]={&q,&p2,&pi2}; // double-pointer to array of parent and
                                products
root [9] PChannel c(s); // constructor equivalent to c(s,2,0,0,0)
root [10] c.Print(); // print function available for all the Pluto
Classes
pi- + p --> p + pi-
Interaction model: fixed-momentum beam // reflects the options set by the reaction
isotropic angular distribution flags
fixed-width Breit-Wigner
resonance

```

The [PChannel](#) constructor requires as minimum argument a double pointer to an array of the parent and product particles. In the example above the defaults were invoked for the remaining arguments, namely, 2 for the number of product particles, and [MassFlag](#)=0, [AngleFlag](#)=0, [BeamFlag](#)=0 (Section [III.c](#)) suppressing the physics of the interaction. This reflects in the output of the [Print\(\)](#) function. The physics defaults may be enabled by resetting the flag values. In this example the only relevant modification concerns the beam-sampling option that, if set to 1, will result in the [sampling of the beam momentum](#) from a [known profile](#) in the course of executing a simulation (discussed later). The other two flags, even if activated (i.e. set to 1), will be reset to zero, since no mass-dependent total widths are calculated (see the relevant [discussion](#) in Section [IV.b](#)), and no angular-distribution model is available (therefore isotropy is assumed):

```

root [11] c.setBeamFlag(1); // turn on beam sampling
root [12] c.Print();
pi- + p --> p + pi-
Interaction model: HADES pion beam: Dp/p=+/-4%,
resolution=0.003
isotropic angular distribution
fixed-width Breit-Wigner resonance

```

the effect would have been the same if the constructor had been invoked as `c(s,2,0,0,1)` in line [9](#). The core function of the [PChannel](#) class is to induce [particle decay](#). A failure may occur due to several reasons, as e.g. if conservation of

energy is violated. On failure an error code is returned.

```

root [13] c.decay()           // induce a decay: the beam-sampling option is
                             // on
(int)0                       // decay was successful
root [14] p2.Print();        // examine product proton after decay
p (0.217936,-0.447494,0.941629;1.419425) wt = 1.000000, pid = 14
root [15] pi2.Print();      // examine product pion
pi- (-0.217936,0.447494,0.048996;0.519257) wt = 1.000000, pid = 9
root [16] PParticle q2=p2+pi2; // check kinematics: beam momentum is reset
                             // per event
root [17] q2.Print();        // compare with parent (line 4)
quasi-particle (0.000000,0.000000,0.990626;1.938682) wt = 1.000000, pid1 = 14,
pid2 = 9
root [18] c.setBeamFlag(0);  // turn off beam sampling
                             // induce decay again with beam fixed to initial
                             // momentum
root [19] c.decay();
root [20] PParticle q2=p2+pi2; // recombine products
root [21] q2.Print();        // compare again with parent
quasi-particle (0.000000,0.000000,1.000000;1.947965) wt = 1.000000, pid1 = 14,
pid2 = 9

```

The differences in energy and momentum before and after the first decay are due to the fact that beam sampling takes place as specified in line 11. The result is that the beam 4-vector is reset event by event. Turning this option off, the beam momentum remains fixed as input in the constructor, and energy and momentum conservation after **decay()** is verified. Note that decay product 4-vectors are automatically boosted to the lab frame.

An example of simple-minded  $\omega$ -Dalitz decay concludes this section. It demonstrates how to run a simple simulation with the **PParticle** and **PChannel** classes. The arguments of the **PChannel** constructor are: A double pointer of **PParticle** type referencing an array of pointers to the parent and product particles, the number of product particles (2), and the mass-sampling option (1). The remaining arguments are skipped, meaning the default values are used. These are the beam sampling option (default 0), and the anisotropic angular distribution sampling option, that in this particular example are irrelevant. Enabling the mass sampling option means that the channel constructor is instructed to compare this particular decay to the physics scenarios that are hard-wired in the code. This check results in this channel being identified as a Dalitz decay (see the [relevant discussion](#) in Section [III.c](#)).

```

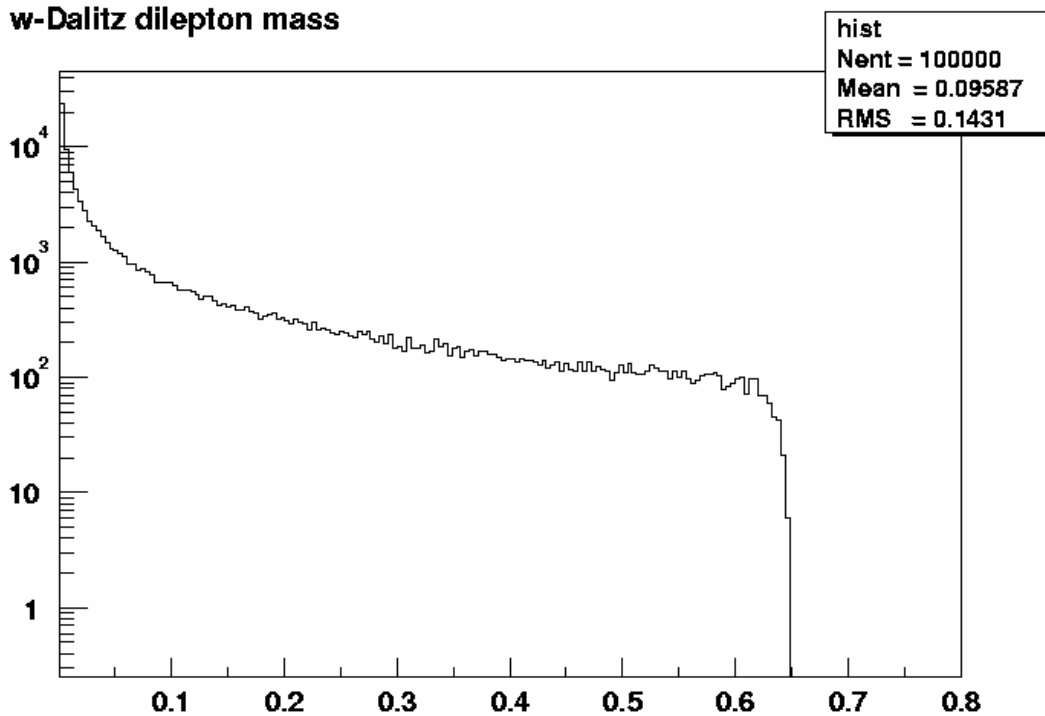
root [1] PParticle w("w",0,0,1); // create an w-meson with Pz=1 GeV/c
root [2] PParticle g("dilepton"); // create a dilepton (virtual photon)
root [3] PParticle pi("pi0");      // create a pi0
root [4] PParticle *p[]={&w,&pi,&g}; // pointer to w, pi0, dilepton array
root [5] PChannel c(p,2,1);        // channel: 2 products, dilepton mass-
                             // sampling
root [6] c.Print();               // display channel info
omega --> pi0 + photon
Interaction model: isotropic angular distribution // reflects options set up by the flags
VMD Dalitz decay
root [7] c1=new TCanvas("c1","Dalitz ",200,10,700,500); // ROOT graphics
root [8] hf=new TH1F("hist","w-Dalitz dilepton // set up histogram
mass",200,0.001022,0.8);
root [10] for (int i=0;i<100000;++) {if (!c.decay()) hf.Fill(g.M() );} // 100k event loop
root [11] hf.Draw();              // plot w-mass histogram

```

Note the *if* statement in line 10: As commented earlier, **decay** is an integer-type function returning 0 on success, or a non-zero integer error code on failure. Although in this particular example there is no possibility of failure (i.e. the kinematics is always possible), a check is advisable in order to prevent filling histograms with meaningless events.

The result of this simulation is shown in Fig. 4. The  $\omega$ -mass is distributed according to Eq. (13). The numerical implementation of this distribution function is discussed in Section [All.c](#).

**Figure 4** Virtual-photon mass spectrum produced in the Dalitz decay of an  $\omega$  meson.



## IV.e Reactions

A full reaction is now considered:  $q \rightarrow n + \omega \rightarrow n + \pi^0 + \gamma^* \rightarrow n + \pi^0 + e^+ + e^-$  with  $q = \pi^- + p$  a composite particle, the "sum" of a pion beam of kinetic energy  $T=1.1$  GeV, and a proton target at rest. This reaction consists of three [channels](#): 1)  $q \rightarrow n + \omega$  followed by 2) the Dalitz decay of the  $\omega$ -meson  $\omega \rightarrow \pi^0 + \gamma^*$ , and finally 3) the dilepton decay of the virtual photon  $\gamma^* \rightarrow e^- + e^+$ . The [PReaction](#) class constructor declares the reaction, after the participating particles and channels have been defined by the constructors of the [PParticle](#) and [PChannel](#) classes:

```

root [1] PParticle pim("pi-",1.1);           // pi- beam of T=1.1 GeV (kinetic
root [2] PParticle p("p");                  // proton target at rest
root [3] PParticle n("n");                  // neutron at rest
root [4] PParticle g("dilepton");           // virtual photon at rest
root [5] PParticle w("w");                  // omega at rest
root [6] PParticle pi0("pi0");              // pi0 at rest
root [7] PParticle em("e-");               // electron at rest
root [8] PParticle ep("e+");               // positron at rest
root [9] PParticle q=pim+p;                // quasi-particle: beam + target
root [10] PParticle *s1[]={&q,&n,&w}, *s2[]={&w,&pi0,&g}, // double pointers of PParticle type
*s3[]={&g,&em,&ep};
root [11] PChannel c1(s1), c2(s2), c3(s3); // three channels declared
root [12] c1.Print();                      // entry channel
pi- + p --> n + w
Interaction model: fixed-momentum beam
isotropic angular distribution           // physics options are suppressed
fixed-width Breit-Wigner

resonances
root [13] c2.Print();                      // note the physics model:
omega --> pi0 + dilepton
Interaction model: isotropic angular distribution
fixed-width Breit-Wigner                // not yet recognized as Dalitz

```



```

resonances
root [14] c3.Print(); // the dilepton decay channel
dilepton --> e- + e+
    Interaction model: isotropic angular distribution // isotropic phase-space decay
                    fixed product masses

```

Note that the three channels were instantiated with only the [PChannel](#)-type double-pointer as argument, so that the remaining [PChannel](#) constructor arguments were set to the default values suppressing the "physics" of the channels. This is why the "interaction model" for the three channels interpreted to be a simple 2-body phase-space decay, the default trivial interaction model, implemented in the [PChannel](#) member function [genbod\(\)](#) [26] (an adaptation from the homonymous CERNLIB function). The matching of a channel with a physical decay mode, to the extent that it is known to the code, may be enabled by passing a number of flags of value 1 to the [PChannel](#) constructor (see the [relevant discussion](#) in Section [III.c](#)). Alternatively, after declaration of the channels, the flags may be [reset](#) by [PChannel](#) member functions. In this particular example, the relevant physics that has been suppressed is the possibility to [sample pion-beam momenta](#), and the [mass](#) of the [dilepton](#) for the  $\omega$ -Dalitz decay (see Section [A1.b.2](#)). The physics of the channels, however, are also restored by default upon declaring a reaction to which they participate (see [discussion](#) in Section [III.d](#)):

```

root [15] PChannel *c[]={&c1,&c2,&c3}; // double pointer of PChannel
                                         type
root [16] PReaction r(c,"w_dalitz",3,1,0,0,1); // instantiation of the PReaction
root [17] r.Print(); // display reaction info

Reaction of 8 Particles interacting via 3 Channels
Reaction Particles:
  0. pi- (beam)
  1. p (target)
  2. n (tracked particle 0)
  3. w
  4. pi0 (tracked particle 1)
  5. dilepton
  6. e- (tracked particle 2)
  7. e+ (tracked particle 3)
Reaction Channels:
  1. pi- + p --> n + w
      Interaction model: HADES pion beam: Dp/p=+/-4%, resolution=0.003 // physics restored
                      w-production angular distribution
                      mass-dependent-width Breit-Wigner resonances
  2. w --> pi0 + dilepton
      Interaction model: isotropic angular distribution
                      VMD Dalitz decay
  3. dilepton --> e- + e+
      Interaction model: isotropic angular distribution
                      fixed product masses

Output Files:
Root : w_dalitz.root, all particles on file.
Ascii: w_dalitz.evt, tracked particles on file.

```

- The [reaction](#) is instantiated in line [16](#). The arguments passed to the constructor are a) a pointer to an array of pointers to [channels](#) b) a character string for the output file names, c) the number of [channels](#) (default is 2, here 3 channels), d) the [output option](#) is 1: [ROOT](#) output containing all the particles on file (serial numbers 0-7) e) the [decay-mode](#) option (default is 0, resetting the [channel](#) flags to allow matching with known decay modes and restoring the physics- 1 would have left the [channels](#) unchanged) f) the [vertex-calculation](#) option is zero (no production vertices calculated) and g) [ASCII output](#) for the *tracked particles* only (tracked-particle serial numbers 0-3) will be produced, in the [HGeant](#) input format.
- For additional output options see the discussion in Section [III.c](#).
- The scheme of passing the channels to the reaction is identical to the way particles were passed to channels. This is the most economical scheme, namely passing a pointer to an array of pointers, along with the number of entries in that array. If the channels were to be

passed directly to the reaction a different constructor for every possible number of participating channels would have been required.

- So far the [reaction](#), participating [channels](#), and [particles](#) in this example have only been instantiated. No output has been produced yet, neither has the simulation been executed. All the objects are instantiated once, and subsequently their entries are accessed and updated during the execution of an event loop (discussed next).

To execute the simulation, the [reaction](#) function `loop(N)` is invoked for  $N$  events:

```
root [18] r.loop(100000);
PReaction: calculating widths in
PData...
...widths calculated in 6.440000
sec
New loop: 100000 events
20% done in 30.640000 sec
40% done in 60.410000 sec
60% done in 89.330000 sec
80% done in 121.610000 sec
100% done in 167.190000 sec
CPU time 141.440000 sec
```

At the end of the event loop events have been [written on file](#). This example illustrates a complete, unconstrained simulation. Next, geometrical acceptance filters are applied.

#### IV.f Filters

The [HADES](#) geometrical acceptance during the September 1999 run is considered as an example here, namely polar angles in the range of 18-85 degrees, azimuthal angles from 2 to 58 degrees (Sector VI of the spectrometer), or from -122 to -178 degrees (Sector III). The trigonometric functions have a range of [0-180] degrees for polar, and [-180,180] (i.e. not [0,360] degrees) for azimuthal angles. Angle arguments are entered in radians in the [filter](#) constructor.

The filter as described is applied on the reaction of the previous example:

```
root [19] PFilter f1(&r,"(thet(6)>0.31415926536 && thet(6)
<1.4835299) && (thet(7)>0.31415926536 && thet(7)<1.4835299)");
root [20] PFilter f2(&r,"(0.03490658504<phi(6) && phi(6)
<1.0122909662) || (-3.1066860685<phi(6) && phi(6)
<-2.1293016874)");
root [21] PFilter f3(&r,"(0.03490658504<phi(7) && phi(7)
<1.0122909662) || (-3.1066860685<phi(7) && phi(7)
<-2.1293016874)");
root [22] r.Print();
...
```

Reaction Filters:

1. Active at Channel 3:  
(thet(6)>0.31415926536 && thet(6)<1.4835299) &&  
(thet(7)>0.31415926536 && thet(7)<1.4835299)
2. Active at Channel 3:  
(0.03490658504<phi(6) && phi(6)<1.0122909662) ||  
(-3.1066860685<phi(6) && phi(6)<-2.1293016874)
3. Active at Channel 3:  
(0.03490658504<phi(7) && phi(7)<1.0122909662) ||  
(-3.1066860685<phi(7) && phi(7)<-2.1293016874)

Output Files:

Root : w\_dalitz.root, all particles on file.

Ascii: w\_dalitz.evt, tracked particles on file.

```
root [23] r.loop(100000);
```

```

PReaction: calculating widths in PData...
...widths calculated in 0.020000 sec
New loop: 100000 events
20% done in 38.980000 sec
40% done in 85.820000 sec
60% done in 128.130000 sec
80% done in 173.650000 sec
100% done in 213.360000 sec
CPU time 152.640000 sec
42864 events failed filter 1
68956 events failed filter 2
68982 events failed filter 3

```

The polar-angle filter is first in line [19](#), requiring both the electron (particle No. 6) and the positron (particle No. 7) to be accepted. The counting scheme for the particles is indicated in the output of `Print()`. The azimuthal-angle filters for the electron and the positron individually are set up in the subsequent two lines.

All the particles are valid as filter arguments, whether surviving or decaying. The left-hand side of the first channel is special, and whether the beam and target are counted individually (as in this example), or their composite quasi-particle sum is counted collectively as particle No. 0, depends on whether or not the beam is sampled. Here, since beam-sampling is activated, the beam 4-vector is sampled event by event, therefore both the beam and target particles are updated per event, and it is convenient to record them individually. Should the beam-sampling option be off, or a beam profile unavailable, the quasi-particle sum of beam and target would be fixed, and particle No. 0 would have been that quasi particle.

A filter is automatically attached at that channel by which all the particles participating in that filter have been produced - in this case channel 3. Therefore a check on whether the filter is passed or failed by an event is performed following the decay induced at that channel. `Suspend()` and `Resume()` are `PFilter` member functions that can temporarily de-activate and reactivate a filter. The status of the filter (i.e. active or not) is also indicated in the output of `Print()`. Counters of the number of failed events per filter are displayed on exit from the event loop. Each filter performs a check against all the events, irrespective of whether a previous filter has been failed or not. Thus, there may be an overlap of multiply failed events counted by each of the filters (see the related [discussion](#) of Section [III.d](#)).

- The arguments required by the `filter` constructor are a) a pointer to a [reaction](#), and b) a [valid](#) character expression specifying the condition to be applied by the filter.
- In this example `loop()` has been invoked the first time without, and the second time with filters. In each case a [tree](#) has been written on file, of which the first has one [branch](#) (Particles) and the second has two [branches](#) (Particles and Filters). Since the [output-option](#) had been set to 1, an ASCII file was also created in the two runs, named in this particular example `w_dalitz.evt`. Of these files, the first contains the tracked particles (indicated in the output of `Print()`) for all the events (since no filters applied), whereas the second file contains the tracked particles of successful events only. The ROOT output file, however, contains **all** particles, intermediate and tracked.

## IV.g Output

ASCII output files have the following format (shown for two events) suitable for input to [HGeant](#):

1	4	1.088528e+00	0.	2			
1.319154e+00	8.570251e-03	-1.361242e-01	9.158500e-01	13	14009	14009	1.000000e+00
4.119939e-01	2.246808e-01	3.175009e-01	-1.523261e-02	7	14009	52	1.000000e+00
4.231599e-01	-2.271476e-01	-1.764984e-01	3.103488e-01	3	14009	51	1.000000e+00
1.206196e-02	-6.103448e-03	-4.878375e-03	9.174923e-03	2	14009	51	1.000000e+00
2	4	1.100051e+00	0.	2			
1.042649e+00	7.221160e-03	1.678006e-01	4.196721e-01	13	14009	14009	1.000000e+00
1.953898e-01	9.067385e-02	6.620157e-02	-8.575631e-02	7	14009	52	1.000000e+00
5.175738e-01	-6.038183e-02	-1.831765e-01	4.802945e-01	3	14009	51	1.000000e+00

The event-header consists of the event serial number, the number of particles, the beam kinetic energy (GeV) for that event, the impact parameter (relevant only for heavy ions), and a flag specifying the amount of information written out. The tracked (surviving) particles follow, in this example the neutron, neutral pion, electron and positron. From left to right, the entries are the Energy (GeV), Px, Py, and Pz (GeV/c) momentum components, the particle id, source id, parent id and the weight. Modifications have been implemented in [HGeant](#) to allow [processing **Pluto**<sup>++</sup>-generated events on the fly from within a [ROOT](#) session. If this option is used, an ASCII output file is not necessary, the event is copied directly into a FORTRAN common block accessed from GEANT.

The macro shown below retrieves data from the [ROOT](#) file [w\\_dalitz.root](#) produced in the example of the previous section,

which can be viewed by opening a [ROOT](#) browser via the [TBrowser](#) constructor.

#### Analysis macro [analyze\\_w\\_dalitz.C](#)

```
{
TFile *f = (TFile*)gROOT->GetListOfFiles()->FindObject("w_dalitz.root");
if (!f) f = new TFile("w_dalitz.root");
TTree *Reaction = (TTree*)gDirectory->Get("data");           // get tree from file
                                                             // must create
TClonesArray *evt=new TClonesArray("PParticle",8);          TClonesArray
TArrayI *Status = new TArrayI(3);                           // overall filter status
Reaction->SetBranchAddress("Particles",&evt);                // PParticle-array address
Reaction->SetBranchAddress("Filters",Status->GetArray());      // filter address
const long double r2d=180./3.14159265358979323846;           // converter rad to deg
TVector3 v1,v2;                                              // 3-vector variables
                                                             // pointer to array of
PParticle *par[8];                                           PParticles

TH1F *hf1=new TH1F("hf1","a. pi- beam momentum profile",200,1.18,1.28);
TH1F *hf2=new TH1F("hf2","b. w-meson mass",200,0.72,0.84);
TH1F *hf3=new TH1F("hf3","c. dilepton mass",200,0.,0.8);
TH1F *hf4=new TH1F("hf4","d. polar angle all e+ and e- (deg)",200,0,180);
TH1F *hf5=new TH1F("hf5","e. polar angle accepted e+ and e- (deg)",200,0,180);
TH1F *hf6=new TH1F("hf6","f. accepted e+e- opening angle (deg)",200,0,180);
c1 = new TCanvas("c1","pi- + p-> n + w -> n + pi0 + gamma* -> n + pi0 + e- +
e+",200,10,900,900);
c1->Divide(3,2);
c1_3->SetLogy();
c1_6->SetLogy();
Int_t nentries = Reaction->GetEntries();                      // get number of entries
for (Int_t i=0; i<nentries;i++) {                             // enter event loop
    Reaction->GetEntry(i);                                     // current event
    par[0]=(PParticle*)evt.At(0);                             // retrieve the pi- beam
                                                             particle
    par[3]=(PParticle*)evt.At(3);                             // retrieve the w-meson
    par[5]=(PParticle*)evt.At(5);                             // retrieve the dilepton
    par[6]=(PParticle*)evt.At(6);                             // retrieve the electron
    par[7]=(PParticle*)evt.At(7);                             // retrieve the positron
    hf1->Fill(par[0]->Pz());                                   // beam Pz momentum
                                                             profile
    hf2->Fill(par[3]->M());                                    // w-meson mass
    hf3->Fill(par[5]->M());                                    // dilepton mass
    hf4->Fill(r2d*(par[6]->Theta()));                          // electron polar angle
                                                             (deg)
    hf4->Fill(r2d*(par[7]->Theta()));                          // positron polar angle
                                                             (deg)
}
```

```

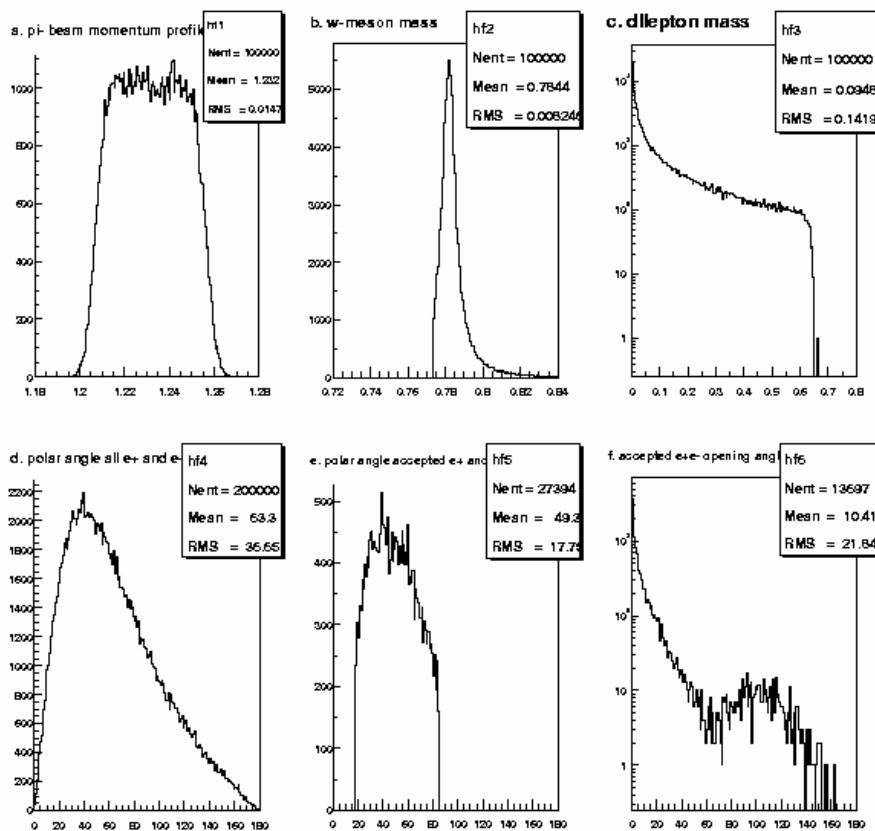
if ( (*Status)[0] == 0 ) {
    hf5->Fill(r2d*(par[6]->Theta()));
    hf5->Fill(r2d*(par[7]->Theta()));
    v1=par[6]->Vect();
    v2=par[7]->Vect();
    hf6->Fill(r2d*
(acos(v1.Dot(v2)/(v1.Mag()*v2.Mag()))));
}
}
c1_1->cd();
hf1->Draw();
c1_2->cd();
hf2->Draw();
c1_3->cd();
hf3->Draw();
c1_4->cd();
hf4->Draw();
c1_5->cd();
hf5->Draw();
c1_6->cd();
hf6->Draw(); }

```

// accepted dileptons only  
// electron polar angle  
(deg)  
// positron polar angle  
(deg)  
// e- momentum vector  
// e+ momentum vector  
// e+e- opening angle (deg)

Executing this macro (e.g. by typing `.x analyze_w_dalitz.C`) produces the histograms of Fig. 5:

**Figure 5** Spectra produced in executing the event-generation and analysis macros discussed in the text.



Panel a) illustrates the charged-pion [beam profile](#), as [discussed earlier](#). Panel b) displays the typical skewed mass-dependent-width Breit-Wigner shape of a hadronic resonance, in this case the  $\omega$  meson (calculated in [PData](#)). Panel c) displays the shape which is typical of virtual photon mass spectra, produced in Dalitz decays, in this case for  $\omega$

-Dalitz decay as described by Eq. (13) and the techniques of Section [All.c](#). Panels d) and e) show the polar-angle distributions of the electrons and positrons, the former for all and the latter for the ones that cleared all three filters (i.e. both electron and positron accepted). Panel f) shows the  $e^+e^-$  opening angle for the accepted dileptons. Opening-angle spectra may be useful in identifying cuts to suppress the combinatorial background in realistic situations.

The macro structure shown above is most likely to be useful in simulations, reconstructing entire particle objects from file, so that [PParticle](#) member functions may subsequently be invoked. It is also possible to only retrieve a limited number of integer or double-type members, in this case the momenta, energies, pid's, or weights of particles (see the related [discussion](#) of Section [III.d](#)), in which case CPU time and memory can be saved (although typical macros will take only a few CPU seconds to execute either way) by retrieving selectively. This is possible due to the [TClonesArray](#) features, used in creating [branches](#) in the output [tree](#). The skeleton macro below illustrates how to selectively enable individual branches, and can be obtained by executing the native [ROOT TTree](#) member function **MakeCode()**:


```
root [24] TFile output("w_dalitz.root");
root [25] t=(TTree*)output.Get("data");
root [26] t->MakeCode("test_macro.C");
```

Effectively, [the lines in blue](#) in the earlier analysis macro need to be replaced by corresponding statements that can be copied from the output of **MakeCode()**, pointing to entries selectively instead of entire objects.

## IV.h Macros

A number of macros are distributed with the code, some of which are presented in this section. All macros can be downloaded from the [Pluto macro page](#).

### IV.h.1 pp-induced Dalitz decay

Here  Dalitz decay is discussed, from  $pp$  scattering at 1.5 GeV, with the HADES geometrical acceptance of September 1999.

The simulation and analysis macros follow:

#### Simulation macro [pp\\_delta\\_dalitz.C](#)

```
// p + p -> p + Delta+ -> p + p + gamma* -> p + p + e- + e+
// This macro tests:
// 1. The mass-dependent width Breit-Wigner distribution of the Delta
// 2. The anisotropic (s+p wave) production angle for the pp->pDelta channel
// 3. The dilepton mass for Delta Dalitz decay
{
  gSystem->Load("libPhysics");
  gSystem->Load("~/user/pluto_directory/pluto.so");
  PParticle *p1=new PParticle("p", 0,0,1.5);
  PParticle *p2=new PParticle("p"), *p3=new PParticle("p"), *p4=new PParticle("p");
  PParticle *delta=new PParticle("D+");
  PParticle *g=new PParticle("dilepton");
  PParticle *em=new PParticle("e-");
  PParticle *ep=new PParticle("e+");
  PParticle *q=new PParticle(*p1+*p2);
  PParticle *s1[]={q,p3,delta}, *s2[]={delta,p4,g}, *s3[]={g,em,ep};
  PChannel *c1=new PChannel(s1), *c2=new PChannel(s2), *c3=new PChannel(s3), *cc[]={c1,c2,c3};
  PReaction *r=new PReaction(cc,"pp_delta_dalitz",3,1,0,0,0);
```

```

PFilter *f1=new PFilter(r,"(thet(5)>0.31415926536 && thet(5)<1.4835299) &&
(thet(6)>0.31415926536 && thet(6)<1.4835299)");
PFilter *f2=new PFilter(r,"(0.03490658504<phi(5) && phi(5)<1.0122909662) ||
(-3.1066860685<phi(5) && phi(5)<-2.1293016874)");
PFilter *f3=new PFilter(r,"(0.03490658504<phi(6) && phi(6)<1.0122909662) ||
(-3.1066860685<phi(6) && phi(6)<-2.1293016874)");
r->Print();
r->loop(100000);
}

```

The execution of this macro produces a [ROOT](#) output file named **pp\_delta\_dalitz.root**, and the following output on screen:

```

root [2] .x pp_delta_dalitz.C
Reaction of 7 Particles interacting via 3 Channels
Reaction Particles:
0. quasi-particle (fixed p beam and p target)
1. p
2. D+
3. p
4. dilepton
5. e-
6. e+
Reaction Channels:
1. p + p -> p + D+
    Interaction model: fixed-momentum beam
                        N+N->N+Delta angular distribution
                        mass-dependent-width Breit-Wigner resonances
2. D+ -> p + dilepton
    Interaction model: isotropic angular distribution
                        VMD Dalitz decay
3. dilepton -> e- + e+
    Interaction model: isotropic angular distribution
                        fixed product masses
Reaction Filters:
1. Active at Channel 3:
    (thet(5)>0.31415926536 && thet(5)<1.4835299) &&
(thet(6)>0.31415926536 && thet(6)<1.4835299)
2. Active at Channel 3:
    (0.03490658504<phi(5) && phi(5)<1.0122909662) ||
(-3.1066860685<phi(5) && phi(5)<-2.1293016874)
3. Active at Channel 3:
    (0.03490658504<phi(6) && phi(6)<1.0122909662) ||
(-3.1066860685<phi(6) && phi(6)<-2.1293016874)
Output Files:
Root : pp_delta_dalitz.root, all particles on file.

PReaction: calculating widths in PData...
...widths calculated in 4.520000 sec
New loop: 100000 events
20% done in 33.740000 sec
40% done in 67.920000 sec
60% done in 101.760000 sec
80% done in 137.080000 sec
100% done in 175.670000 sec
CPU time 169.820000 sec
41290 events failed filter 1
68910 events failed filter 2
69072 events failed filter 3

```



The analysis macro follows:

**Analysis macro `analyze_pp_delta_dalitz.C`**

```
{
gROOT->Reset();
TFile *f =
(TFile*)gROOT->GetListOfFiles()->FindObject("pp_delta_dalitz.root");
if (!f) f = new TFile("pp_delta_dalitz.root");
TTree *Reaction = (TTree*)gDirectory->Get("data");
TClonesArray *evt=new TClonesArray("PParticle",7);
Reaction->SetBranchAddress("Particles",&evt);
PParticle *par[7];
TArrayI *Status = new TArrayI(3);
Reaction->SetBranchAddress("Filters",Status->GetArray());
const long double r2d=180./3.14159265358979323846;
TVector3 v1,v2;
TH1F *hf1=new TH1F("hf1","Delta mass",200,1.1,1.4);
TH1F *hf2=new TH1F("hf2","Delta lab. production angle (deg)",200,0.,45.);
TH1F *hf3=new TH1F("hf3","Dilepton mass",100,0.,0.37);
TH1F *hf4=new TH1F("hf4","accepted e+ and e- scattering angle
(deg)",200,10,90);
TH1F *hf5=new TH1F("hf5","accepted dilepton rapidity",200,0,2);
TH1F *hf6=new TH1F("hf6","accepted e+e- opening angle",200,0.,160.);
c1 = new TCanvas("c1","p + p -> p + Delta -> p + p + gamma* -> p + p + e- +
e+",200,10,900,900);
c1->Divide(3,2);
c1_3->SetLogy();
c1_6->SetLogy();
Int_t nentries = Reaction->GetEntries();
for (Int_t i=0; i<nentries;i++) {
    Reaction->GetEntry(i)
    par[2]=(PParticle*)evt.At(2);
    par[4]=(PParticle*)evt.At(4);
    par[5]=(PParticle*)evt.At(5);
    par[6]=(PParticle*)evt.At(6);
    hf1->Fill(par[2]->mass());
    hf2->Fill(r2d*(par[2]->Theta()));
    hf3->Fill(par[4]->M());
    if ( (*Status)[0] == 0) {
        hf4->Fill(r2d*(par[5]->Theta()));
        hf4->Fill(r2d*(par[6]->Theta()));
        hf5->Fill(par[4]->Rapidity());
        v1=par[5]->Vect();
        v2=par[6]->Vect();
        hf6->Fill(r2d*(acos(v1.Dot(v2)/(v1.Mag()*v2.Mag()))));
    }
}
c1_1->cd();
hf1->Draw();
c1_2->cd();
hf2->Draw();
c1_3->cd();
hf3->Draw();
c1_4->cd();
hf4->Draw();
c1_5->cd();
```

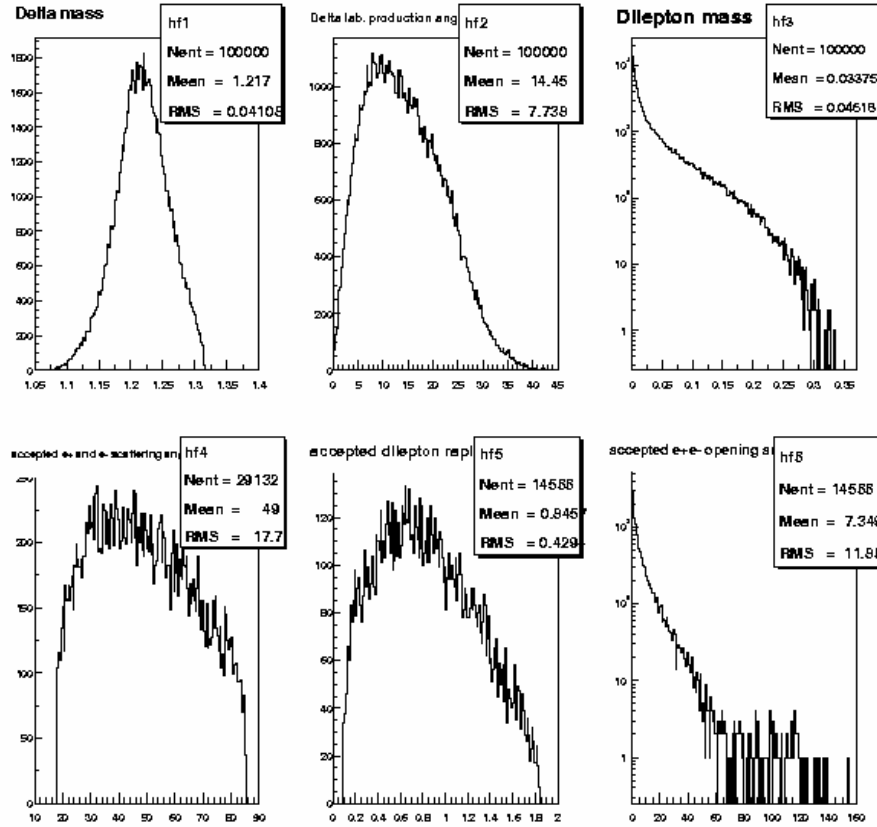
```

hf5->Draw();
c1_6->cd();
hf6->Draw();
}

```

This produces the histograms of Fig. 6:

**Figure 6** Spectra produced in executing the  $\Delta$ -Dalitz decay simulation discussed in the text.



The  $\Delta$  resonance mass is sampled from the distribution function of Eq. (6), and the dilepton mass in  $\Delta \rightarrow \gamma^* N$  from that of Eq. (14). The top three panels include all the events, whereas the three bottom ones are for the accepted dileptons only.

#### IV.h.2 pp elastic scattering

Elastic  $pp$  scattering is useful for detector and spectrometer calibration studies. It is therefore useful to have in hand a convenient parametrization for sampling realistic scattering angles. A toy-model parametrization had been developed at GSI [10] (see related discussion at [web-docs.gsi.de/~webhades/computing/pluto/NN/pp\\_elastic.html](http://web-docs.gsi.de/~webhades/computing/pluto/NN/pp_elastic.html)). This has been replaced by a parametrization based on a phase-shift analysis encompassing the world data, from an algorithm (SAID) supplied by R. Arndt [16] (class PSaid). This yields elastic  $pp$  scattering distributions accurate to within a fraction of 1% for proton beam energies expected for HADES experiments. An example of a  $pp$  elastic-scattering simulation for 1.687 GeV/c incident protons is shown here as an illustration. It should be noted that the sampled range of angles in the center of mass is [1,179] degrees, in order to avoid the singularities at forward and backward angles due to the Coulomb potential. This is not a limitation since extreme angles practically coincide with the beam path, where no detection is possible.

##### Simulation macro pp\_elastic.C

```

{
gSystem->Load("libPhysics");
gSystem->Load("~user/pluto_directory/pluto.so");

```

```

PParticle *p1=new PParticle("p",0,0,1.687);
PParticle *p2=new PParticle("p");
PParticle *p3=new PParticle("p");
PParticle *p4=new PParticle("p");
PParticle *q=new PParticle(*p1+*p2);
PParticle *s[]={q,p3,p4};
PChannel *c1=new PChannel(s);
PChannel *c[]={c1};
PReaction *r=new PReaction(c,"pp_elastic",1,0,0,0,0);
r->Print();
r->loop(100000);
}

```

#### On-screen output

```

root [1] .x pp_elastic.C
Reaction of 3 Particles interacting via 1 Channels
Reaction Particles:
  0. quasi-particle (fixed p beam and p target)
  1. p (tracked particle 0)
  2. p (tracked particle 1)
Reaction Channels:
  1. p + p --> p + p
      Interaction model: fixed-momentum beam
                        pp elastic angular distribution
                        fixed product masses

Output Files:
  Root : pp_elastic.root, tracked particles on file.

```

```

New loop: 100000 events
20% done in 6.880000 sec
40% done in 14.070000 sec
60% done in 20.580000 sec
80% done in 27.160000 sec
100% done in 33.480000 sec
CPU time 28.660000 sec

```

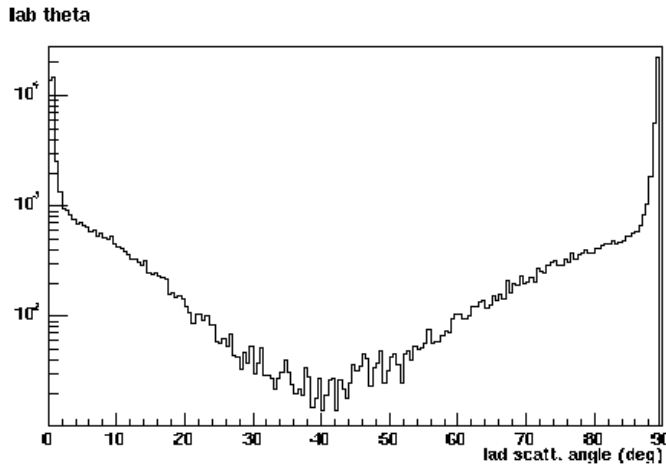
#### Analysis macro analyze\_pp\_elastic.C

```

{
gROOT->Reset();
TFile *f =
(TFile*)gROOT->GetListOfFiles()->FindObject("pp_elastic.root");
if (!f) { f= new TFile("pp_elastic.root"); }
TTree *Reaction = (TTree*)gDirectory->Get("data");
TClonesArray *evt=new TClonesArray("PParticle",2);
Reaction->SetBranchAddresses("Particles",&evt);
const long double r2d=180./3.14159265358979323846;
PParticle *par[2];
TH1F *hf1=new TH1F("hf1","lab theta",180,0,90);
c1 = new TCanvas("c1","pp elastic",200,10,600,800);
TPad pad1("pad1","",0.05,0.01,0.95,0.99);
pad1.SetFillColor(11);
pad1.Draw();
Int_t nentries = Reaction->GetEntries();
for (Int_t i=0; i<nentries;i++) {
  Reaction->GetEntry(i);
  par[0]=(PParticle*)evt.At(0);
  hf1->Fill(r2d*par[0]->Theta()); }
hf1->Draw(); }

```

**Figure 7** Elastic *pp* scattering angular distribution with proton beam momentum 1.687 GeV/c and 100k events.



### IV.h.3 $\eta'$ production in the $p+d$ cocktail

The following macro illustrates the use of the [Decay Manager](#), for  $\eta'$  production in  $p + d$ . In total, eighty five reaction chains contribute to this "cocktail", each weighed by the proper branching ratio retrieved from [PData](#). The output file contains the combined spectra.

#### Simulation macro [etap.C](#)

```
{
gROOT->Reset();
PDecayChannel *c;
PDecayManager *pdm = new PDecayManager;
pdm->SetVerbose(1); // comment out to mute details
pdm->SetDefault("eta"); // enable desired decay channels
pdm->SetDefault("w");
pdm->SetDefault("rho0");
pdm->SetDefault("eta");
pdm->SetDefault("pi0");
pdm->SetDefault("dilepton");
c = new PDecayChannel;
c->AddChannel(1.0, "p", "d", "eta"); // define entry channel: p+d-->p+d+eta'
PParticle *p = new PParticle("p",2.62); // instantiate the proton (beam)
PParticle *d = new PParticle("d"); // instantiate the deuteron (target)
PParticle *s = new PParticle(*p + *d); // composite quasiparticle p+d
pdm->InitReaction(s,c); // instantiate the "cocktail" reaction
Int_t n = pdm->loop(10000,0,"etap",0,0,0,2,0); // execute simulation for a total nb. of evts =10000
cout << "Events processed: " << n << endl; // display total number of events on exit
}
```

The corresponding analysis macro ([analyze\\_etap.C](#)) is distributed with the simulation package and demonstrates the use of the info tree.

## V. Summary

A simulation framework in C++ has been presented, inspired from the physics of HADES, with the potential to grow into a comprehensive ROOT-based simulation package for hadronic physics. The built-in functionality of ROOT as an analysis environment, including the versatile tree structure, provides powerful tools that are fully exploited, enabling complex operations such as the calculation of spectral functions from first principles for hadronic resonances with multiple decay modes. This capability, together with a number of theoretical and empirical hadronic-interaction

models implemented in the code, provide tools for realistic simulations of elementary hadronic interactions, such as resonance excitation and decay, elastic proton-proton scattering, Dalitz decays, and direct dilepton decays of vector mesons. Moreover, the addition of the decay manager interface that trivializes the setting up and execution of multi-channel ("cocktail") simulations, together with the implementation of the thermal model, enable the handling of multi-hadron decays of hot fireballs, and provide basic tools for studies of thermally produced hadrons and the distributions of their observables, and comparison studies with theoretical models such as the UrQMD and BUU. The versatility and reusability of the code, as demonstrated by a number of detector-specific analysis classes and simulation macros distributed with the package, allow for rapid principle simulation studies adapted for particular experimental conditions, detector setups and geometries. It is hoped that this package will become a useful tool as HADES data begin to flow in the coming years, and continues to grow and be further refined with user contributions.

## VI. Acknowledgements

Many thanks are due to Christoph Ernst, for his input regarding theoretical models and their implementation, and to Rene Brun for his frequent assistance with [ROOT](#)-related questions, at the early stages of this project. Two (undocumented) classes ([PTrackH](#) and [PTrack](#)) and macros for detector-specific analysis have been contributed by James Ritman and Marc-André Pleier. These include simplified detector geometries (e.g. simple plane equations for MDC planes, transverse momentum "kick" to approximate the bending of charged-particle tracks by the magnetic field etc), and can serve as examples for additional user-implemented features. The decay-manager interface and supporting template classes, documentation, and macros, have been written by Volker Hejny. The decay manager greatly enhances the utility of the package, by enabling otherwise intractable multi-channel "cocktail" calculations. Johan Messchendorp has parametrized the energy dependence of  $\omega$  production in  $\pi N \rightarrow N \omega$ , and has been an early user, demonstrating the speed and flexibility of the code. Romain Holzmann has implemented the thermal model and the fireball ([PFireball](#)) class, as well as a model for proton-deuteron scattering off a nucleon with the second nucleon as spectator, and the GEANT interface for on-line processing of simulated events. These additions illustrate the direction towards which the package could be expanded to incorporate more collective and bulk nuclear-matter features, an enhancement which is required for realistic heavy-ion simulations. Wolfgang Kuehn and Volker Metag were early believers in the utility of this package, and have provided support and insightful input, which reflects in many of the changes and enhancements of this last release. The author is grateful to Richard Arndt, whose seminal work on phase-shift analyses of  $NN$  and  $\pi N$  elastic scattering data over the years is the standard in the literature, who has kindly supplied an algorithm from his code ([SAID](#)) that is implemented for  $pp$  elastic scattering angular distributions. This replaces a toy-model parametrization formerly implemented in the code, still based on [SAID](#), developed at GSI by Lorenzo Cazon as a project sponsored by the International Student Summer School (1999). Richard Arndt has also provided  $\pi N$  parametrizations that have yet to be implemented in the code. The author finally wishes to express his gratitude to Bengt Friman, whose input has been invaluable in developing the model that has been implemented in the data-base class for the calculation of hadronic-resonance widths and spectral functions.

## VII. Bibliography

- [1] P. Salabura for the [HADES](#) collaboration, *HADES: A High Acceptance Dielectron Spectrometer*, *Acta Phys. Polon.* **B27** (1996) 421.
- [2] Rene Brun and Fons Rademakers, [ROOT](#) - An Object Oriented Data Analysis Framework, *Proceedings AIHENP'96 Workshop, Lausanne, Sep. 1996*, *Nucl. Inst. & Meth. in Phys. Res. A* **389** (1997) 81.
- [3] Leif Lönblad, [CLHEP](#) - a Project for Designing a C++ Class Library for High Energy Physics, *Comp. Phys. Comm.* **84** (1994) 307.
- [4] Heike Schön, [HADES](#) - Ein Dielektronenspektrometer hoher Akzeptanz für relativistische Schwerionenkollisionen, Ph.D. Dissertation, Johann Wolfgang Goethe University, Frankfurt am Main (1995) unpublished.
- [5] [GEANT](#): [CERN](#) Program Library Long Writups Q123.
- [6] [C. Ernst](#), *Dileptonen als Signal für in-Medium-Effekte in relativistischen Schwerionenkollisionen*, Diplomarbeit, Johann Wolfgang Goethe University, Frankfurt am Main (1998) unpublished.
- [7] [Gy. Wolf](#), G. Batko, W. Cassing, U. Mosel, K. Niita, M. Schäfer, *Dilepton Production in Heavy-Ion Collisions*, *Nuc. Phys.* **A517** (1990) 615.
- [8] J.H. Koch, N. Ohtsuka, and .J. Moniz, *Nuclear Photoabsorption and Compton Scattering at Intermediate Energy*, *Ann. Phys.* **154** (1984) 99.
- [9] S. Teis, W. Cassing, M. Effenberger, A. Hombach, U. Mosel, and Gy. Wolf, *Pion-Production in Heavy-Ion Collisions at*

- SIS Energies, *Z. Phys.* **A 356** (1997) 421.
- [10] B. Friman, private communication.
- [11] Gy. Wolf, private communication.
- [12] L.G. Landsberg, *Phys. Rep.* **128** (1985) 301.
- [13] V. Dmitriev, O. Sushkov, and C. Gaarde, *Delta-Formation in the  $1H(3He,3H)++\Delta$  Reaction at Intermediate Energies*, *Nucl. Phys.* **A459** (1986) 503.
- [14] J.S. Danburg, M.A. Abolins, O.I. Dahl, D.W. Davies, P.L. Hoch, J. Kirz, D.H. Miller, and R.K. Rader, *Phys. Rev. D* **2** (1970) 2564.
- [15] R. Arndt and R.L. Workman, *Few Body Syst. Suppl.* **7** (1994) 64; Program SAID [http://said.phys.vt.edu/said\\_branch.html](http://said.phys.vt.edu/said_branch.html).
- [16] R. Arndt, private communication.
- [17] L. Cazon Boado, and M.A. Kagarlis "Monte Carlo Simulations of Elastic Proton-Proton Scattering", project sponsored by the GSI International Student Summer School, 1999.
- [18] Siemens and Rasmussen, *PRL* **42** (1979) 880.
- [19] Lacombe et al., *PL B* **101** (1981) 139.
- [20] R.S. Simon for the PION Collaboration, *Secondary Pion Beams at GSI, Progress in Particle and Nuclear Physics* **42** (1999) 247.
- [21] R. Schicker et. al., *Acceptance and Resolution Simulation Studies for the Dielectron Spectrometer HADES at GSI*, *Nucl. Instrum. Meth.* **A380** (1996) 586.
- [22] [Particle Data Group](#), *Review of Particle Properties*, *Phys. Rev.* **D54** (1996) 1 (and earlier editions).
- [23] S. Capstick and W. Roberts, *nucl-th/9708848*.
- [24] W. Cassing and E.L. Bratkovskaya, *Hadronic and Electromagnetic Probes in Hot and Dense Nuclear Matter*, *Phys. Rep.* **308** (1999) 65.
- [25] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery, [Numerical Recipes](#) in C : The Art of Scientific Computing, Cambridge University Press (2nd edition), Cambridge, January 1993.
- [26] F. James, *Monte Carlo Phase Space*, CERN 68-15 (1968)
- [27] M. Matsumoto and T. Nishimura, *Mersenne Twistor: A 623-dimensionally equidistributed uniform pseudorandom number generator*, *ACM Transactions on Modeling and Computer Simulation*, Vol. 8, No. 1, January 1998, pp 3–30; The Mersenne Twistor homepage is located at <http://www.math.keio.ac.jp/~matumoto/emt.html>

## Appendix I: Hadronic Physics

The objective of **Pluto<sup>++</sup>** is to provide a hadronic-physics simulation tool for the [HADES](#) experimentalist, by which to simulate reactions of interest, study properties of the spectrometer, and eventually analyze data. Nonetheless, realistic simulations require well-founded, reliable theoretical models, such as e.g. the BUU [[7,9,24](#)] and [UrQMD](#) (Ref. [[6](#)] and references therein), as well as empirical parametrizations of data when available [[13-15](#)].

This package incorporates models that address 1) resonance widths and mass distributions in the nuclear medium [[6-12](#)], 2) Dalitz-decay and direct dilepton decay channels [[6,12](#)], 3) anisotropic scattering angles for  $NN \rightarrow N\Delta$  [[13](#)] and  $\pi N \rightarrow N\omega$  [[14](#)] using available parametrizations from the literature, and 4)  $NN$  elastic scattering in the energy range of interest for [HADES](#) [[15-17](#)]. The thermal model has also been implemented [[18](#)], for thermal multi-hadron decays of hot quasi-particle fireballs, as well as proton-deuteron scattering [[19](#)].

An important effect that must be taken into account for realistic simulations is the deviation of resonance shapes from fixed-width Breit-Wigner distributions, which is typically modeled as a mass-dependence in the resonance width. This is particularly important for resonances with large widths (i.e. predominantly strongly decaying), such as the  $\rho$ ,  $\Delta$ ,  $N^*$ , and  $\Delta^*$ -resonance excitations for which the effect is largest. The next few sections discuss the formalism behind the calculation of partial and total widths in the code.

### AI.a Two-body hadronic decays

The majority of well-established resonance hadronic decays involve channels with two decay products, whereas multi-product decay modes are generally considered to be the outcome of a series of successive two-body decays through intermediate resonances [[7,9,11,22](#)]. This is also the approach taken in **Pluto<sup>++</sup>**, where two-body hadronic decay widths are calculated explicitly, whereas decay modes involving more than two decay products are avoided, or treated as simple phase-space. In the latter case the static, rather than mass-dependent, widths and branching ratios

from the data base are used. This is not in reality a limitation, since a multi-product decay is generally, and more correctly, treated as a sequence of two-body decays, for which the code calculates mass-dependent observables explicitly. An exception to the rule of avoiding  $N>2$  product decays is the decay mode  $N(\pi\pi)_{L=0}$ , frequently encountered in heavy nucleon and  $\Delta$  resonances, where the two final pions are not the decay products of a mesonic decay, but rather couple to a s-wave and zero isospin. This particular mode, for the purposes of calculating the parent width only, is also treated as a nucleon + dipion two-body decay.

The resonance two-body hadronic-decay width is derived from a well-known ansatz [22], which, e.g. for the  $\Delta$ -resonance typically reads as [7,9]

$$\Gamma(m) = \left( \frac{q}{q_r} \right)^3 \frac{m_r}{m} \left( \frac{v(q)}{v(q_r)} \right)^2 \Gamma_r \quad (1)$$

where the subscript  $r$  in general refers to resonance observables corresponding to the mass pole for the decay mode in hand, which for the  $\Delta$  resonance are for the mass  $m_r = 1.232 \text{ GeV}/c^2$  and for the width  $\Gamma_r = 120 \text{ MeV}$ , whereas unsubscripted variables refer to the corresponding actual-mass observables. The dependence on the two decay products enters via the terms  $q$  and  $q_r$ , namely the (equal in absolute value) center-of-mass momenta of the two decay products  $q = \langle p_{ij}(m) \rangle$ , where

$$\langle p_{ij}(m) \rangle = p_{cm}(m) = \frac{\lambda^{1/2}(m^2, m_i^2, m_j^2)}{2m} \quad (2)$$

and

$$\lambda(m^2, m_i^2, m_j^2) = \left( m^2 - (m_i + m_j)^2 \right) \left( m^2 - (m_i - m_j)^2 \right) \quad (3)$$

with  $m_{ij}$  the masses of the two decay products (for  $q_r$  replace  $m$  with  $m_r$ ). For the  $\Delta$  resonance, calculating the decay width is a simple matter since its decay strength is essentially exhausted in the  $N\pi$  channel (branching ratio >99%). This means that  $\Gamma_r$ , the  $N\pi$ -decay width, coincides with the total width. The product masses  $m_{ij}$  are simply the pion and nucleon masses, which for all purposes may be considered fixed, since the pion has no strong decay modes and its width has a negligible effect in Eq. (1). An additional dependence on the decay mode enters via the angular momentum transfer, implicit in the cubic exponent of the ratio  $q/q_r$  in Eq. (1), which originates from  $2L_{tr}+1$ , with  $L_{tr}=1$  the transferred orbital angular momentum for the  $\Delta$  resonance decay (almost entirely p wave).

Variants of Eq. (1) are also encountered in the literature, particularly with regard to the expression in the right-most bracket that represents the effective cutoff. We follow Ref [9] which uses for the  $\Delta$  resonance the cutoff parametrization of Ref. [8] as shown in Eq. (1), with

$$v(q) = \frac{\beta^2}{\beta^2 + q^2} \quad (4)$$

and the parameter  $\beta = 300 \text{ MeV}$ .

In general, for arbitrary resonances other than the  $\Delta$ ,  $\Gamma_r$  refers to the decay width of the parent resonance with mass equal to the mass pole via the decay mode specified by the identity of the two decay products and the transition angular momentum. In **Pluto**<sup>++</sup> the total (static) width scaled by the (static) branching ratio of the considered decay mode is taken as  $\Gamma_r$ , and the cutoff is parametrized following the ansatz of Ref [9] both for baryons and mesons. The transfer angular-momentum dependence is in general a non-trivial matter, since several multipoles may be interfering. Nonetheless, data are sparse to non-existent for high multipole transitions, and it is generally a reasonable approximation to treat the decay width as entirely due to the lowest-allowed multipole, which is what is done in **Pluto**<sup>++</sup>, on the basis of angular-momentum coupling and parity considerations [10].

The greatest complication, however, which arises in the general case of a resonance decay with either or both of the decay products unstable, is due to the fact that the product masses are in general not fixed as in the case of the  $\Delta$  resonance, but can take on values from a distribution function. The distribution function typically used for hadronic resonances is some variant of a Breit-Wigner distribution with mass dependence in the width, where width here is the



total width

$$\Gamma_{\text{tot}}(m) = \sum_{i=1}^N \Gamma_i(m) \quad (5)$$

$N$  is the total number of decay modes, and the sum is over partial decay widths, such as that of Eq. (1) for  $\Delta$  decays. Following Ref [9] we use the relativistic form of the Breit Wigner distribution

$$g(m) = \frac{1}{\pi} \frac{m^2 \Gamma_{\text{tot}}(m)}{(m_r^2 - m^2)^2 + m^2 \Gamma_{\text{tot}}^2(m)} \quad (6)$$

In Ref [9] and elsewhere few specific cases, with one of the decay products unstable but decaying to stable products and the other with fixed mass (we will refer to this as "depth" 2 decay), are treated explicitly. In general, calculating the decay width when both the decay products are unstable hadrons, or with "depth">2, i.e. with arbitrarily many embedded decays, is a rather complicated matter. In **Pluto++** these cases are treated explicitly, making it possible to calculate realistic spectral functions for heavy  $N^*$ , and  $\Delta^*$ -resonances with multiple decay modes and a large "depth", i.e. nested unstable hadron decays (see Fig 3). This is done as follows: Let us first consider the case where one of the decay products (say the first) is an unstable hadron and the other has a fixed mass ( $m_2$ ). The decay width becomes

$$\Gamma(m) = \int_{m_{\text{min}}^{m-m_2}} dm_1 g_1(m_1) p_{\text{cms}}(m, m_1, m_2) \Gamma(m, m_1, m_2) \quad (7)$$

where we have introduced the dependence on the product masses explicitly in  $\Gamma(m, m_1, m_2)$ . The convolution of the Breit-Wigner shape  $g_1(m_1)$ , as in Eq. (6), with the center-of-mass momentum of the two products in the rest frame of the parent particle  $p_{\text{cms}}(m, m_1, m_2)$ , as in Eq. (2) with the product-mass dependence explicitly introduced, is the effective distribution function for the mass variable  $m_1$ , and weighs properly the contribution of each combination of product masses in the decay width. The center-of-mass momentum term is effectively a cutoff, required to guarantee a smooth falloff at the kinematical limit [10,11].

This expression is generalized in the case of two unstable hadrons

$$\Gamma(m) = \int_{m_{\text{min}}^{m-m_1}} dm_1 g_1(m_1) \int_{m_{\text{min}}^{m-m_1}} dm_2 g_2(m_2) p_{\text{cms}}(m, m_1, m_2) \Gamma(m, m_1, m_2) \theta(m \geq m_1 + m_2) \quad (8)$$

where the Breit-Wigner shape of the second unstable hadron is accounted for as well. For Eqs. (7-8) to hold the weighing must be consistent, namely the probability distribution must be integrated over the full range of validity, and the integral (i.e. the total probability) be normalized to unity. The normalization factor is obtained by removing  $\Gamma(m, m_1, m_2)$  from Eqs. (7-8) and evaluating the remaining integral. Subsequently, corresponding normalization factors are divided out from Eqs. (7-8). As mass threshold for an unstable particle is taken its mass pole minus two times its static width (ansatz by trial-and-error), and for a decay mode the sum of the mass poles of the decay products. As upper limit for resonance masses is taken the mass pole plus twelve times its static width. The step function in Eq. (8) formally expresses conservation of energy, and forces the integral to vanish if it is violated.

In case of nested widths, namely of decay products that they themselves decay, it is clear that the ansatz of Eqs. (7-8) must be repeated for each decay product. This is so since the product total width, entering via Eq. (8), is the sum of terms (7-8) in case of hadronic decays. In fact, product-particle total widths must be known *in advance* of calculating a parent hadronic-decay width of the form (7-8). In practice this is achieved via recursive calls: During the calculation of a resonance total width, as in Eq. (5), individual decay widths are first calculated for each decay mode. At that point product-particle "depths" are examined for each hadronic mode: If both the products are stable, then an ansatz analogous to Eq. (1) yields the decay width for that mode immediately. Otherwise, if either or both of the products have "depth", then the function calculating the width calls itself recursively for each product until stable particles are reached, at which point the innermost width is calculated. Subsequently, as the function exits successively each recursive call from innermost to outermost, the nested widths are calculated one layer at a time until the outermost (parent) particle is reached.

## AI.b Dalitz decays

The total width, Eq. (5), involves the sum of partial decay widths, only some of which are of the form (7-8) for decay modes involving two hadrons as products. Another process of interest for the [HADES](#) physics program is the Dalitz decay of pseudoscalar, vector mesons, and the  $\Delta$  resonance. In these processes one of the two decay products is a virtual (massive) photon, which subsequently decays to a dilepton  $e^+e^-$  pair. These are sources of background dileptons, if direct vector-meson dilepton decays are the focus as in the [HADES](#) program (Section [AI.c](#)), but are of interest on their own accord as probes of electromagnetic form factors.

The virtual-photon mass distribution functions for Dalitz decays are discussed below [\[6.12\]](#).

### AI.b.1 Pseudoscalar mesons

The processes of interest are  $\pi^0, \eta, \eta' \rightarrow \gamma^* \gamma \rightarrow e^+e^- \gamma$ . The first step involves two photons, a real (massless) and a virtual (massive). In the second step the virtual photon decays into a dilepton  $e^+e^-$  pair. For our purposes, the issues are the determination of a mass distribution from which to sample virtual-photon  $\gamma^*$  masses, on one hand, and an expression for the decay width, which we will use with Eq. (5) to calculate the total resonance width, and subsequently the mass-dependent branching ratio.

For pseudoscalar-meson Dalitz decays the mass dependence of the Dalitz-decay width is given by [\[6.12\]](#)

$$\frac{d\Gamma}{dm\Gamma_{A \rightarrow 2\gamma}} = \frac{4\alpha}{3\pi m} \sqrt{\frac{1-4m_e^2}{m^2}} \left(1 + \frac{2m_e^2}{m^2}\right) \left(1 - \frac{m^2}{m_A^2}\right)^3 |F_A(m^2)|^2 \quad (9)$$

where the index A refers to the (parent) pseudoscalar meson, and  $m$ ,  $m_e$ , and  $m_A$  are the dilepton, electron, and pseudoscalar masses, and  $F_A(m^2)$  is the parent form factor

$$F_{AB}(m^2) \approx 1 + m^2 \frac{dF_{AB}}{dm^2} \Big|_{m^2 \rightarrow 0} = 1 + m^2 b_{AB} \quad b_{AB} = \quad \text{for the } \pi^0 \quad (10)$$

$$F(m^2) = \left(1 - \frac{m^2}{\Lambda_i^2}\right)^{-1} \quad \Lambda_\eta = 0.72 \pm 0.09 \text{ GeV} \quad \text{for the } \eta \quad (11)$$

$$|F(m^2)|^2 = \frac{\Lambda^2(\Lambda^2 + \gamma^2)}{(\Lambda^2 - m^2) + \Lambda^2\gamma^2} \quad \Lambda_{\eta'} = 0.76 \text{ GeV} \quad \text{for the } \eta' \quad (12)$$

$$\gamma_{\eta'} = 0.10 \text{ GeV}$$

Eq. (9) is used as the effective distribution function from which virtual-photon masses are sampled. Its integral also yields the partial decay width for pseudoscalar Dalitz-decay modes. It is pointed out that Eq. (9) depends explicitly on the two-photon (real) decay width of the parent pseudoscalar meson (denominator of the left hand side). In practice, in the code, the static branching ratio for the two-photon decay mode from the data base is used to scale pseudoscalar Dalitz-decay widths derived from Eq. (9).

### AI.b.2 Vector mesons

Currently only  $\omega \rightarrow \gamma^* \pi^0 \rightarrow e^+e^- \pi^0$  vector-meson Dalitz decay is implemented in the code. The mass dependence of the decay width is

$$\frac{d\Gamma}{dm\Gamma_{A \rightarrow B\gamma}} = \frac{2\alpha}{3\pi m} \sqrt{\frac{1-4m_e^2}{m^2}} \left(1 + \frac{2m_e^2}{m^2}\right) \left[ \left(1 + \frac{m^2}{m_A^2 - m_B^2}\right)^2 - \left(\frac{2m_A m}{m_A^2 - m_B^2}\right)^2 \right]^{\frac{3}{2}} |F_{AB}(m^2)|^2 \quad (13)$$

where the notation is as in Eq. (9), and the new index B refers to the  $\pi^0$ . The form factor is as in Eq. (13), with  $\Lambda_\omega = 0.65 \text{ GeV}$ , and  $\gamma_\omega = 0.04 \text{ GeV}$ . The vector-meson Dalitz-decay width is scaled by the pion-photon (real) width (denominator on the left-hand side of Eq. (13)). In practice, in the code, the static branching ratio for the pion-photon decay mode of the  $\omega$  meson from the data base is used to scale the width calculated from Eq. (13).

### AI.b.3 $\Delta$ resonance

For  $\Delta \rightarrow \gamma^* N \rightarrow e^+ e^- N$ , the mass-dependence of the width is calculated directly from the matrix element, without scaling factors as in the previous two cases [6,12]:

$$\frac{d\Gamma_{A \rightarrow B e^+ e^-}}{dm} = \frac{2\alpha}{3\pi m} \sqrt{\frac{1-4m_e^2}{m^2}} \left(1 + \frac{2m_e^2}{m^2}\right) \Gamma_{A \rightarrow B \gamma^*} \quad (14)$$

where

$$\Gamma_{A \rightarrow B \gamma^*} = \frac{|\vec{p}_{cm}|}{8\pi m_A^2} |M_{A \rightarrow B \gamma^*}|^2 \quad (15)$$

is the full width, with the indices  $A$  and  $B$  referring to the parent ( $\Delta$  resonance) and product nucleon respectively, and  $p_{cm}(m_A, m_B, m)$  the (common) product center-of-mass momentum in the parent rest frame, as in Eq. (2). The matrix element of Eq. (15) is [6]

$$|M|^2 = e^2 G_M^2 \frac{(m_A + m_N)^2 ((m_A - m_N)^2 - m^2)}{4m_N^2 ((m_A + m_N)^2 - m^2)^2} \quad (16)$$

$$(7m_A^4 + 14m_A^2 m^2 + 3m^4 + 8m_A^3 m_N + 2m_A^2 m_N^2 + 6m^2 m_N^2 + 3m_N^4)$$

where the index  $N$  refers to the produced nucleon,  $e$  is the electron charge, and  $G_M = 2.7$  is the coupling constant.

### AI.c Vector-meson dilepton decays

Vector mesons couple to photons, as is well known from the Vector Meson Dominance (VMD) model, and have a direct dilepton decay mode  $\rho^0, \omega, \phi \rightarrow e^+ e^-$ . The decay products (electron-positron) have obviously fixed masses, therefore mass sampling is not an issue here, but since this is perhaps the most important process from the point of view of [HADES](#), the decay width (and branching ratio) are explicitly calculated in the code. The mass dependence of the direct vector-meson dilepton decay width for is given by [6]

$$\Gamma_{V \rightarrow e^+ e^-}(m) = \frac{c_V}{m^3} \sqrt{\frac{1-4m_e^2}{m^2}} \left(1 + \frac{2m_e^2}{m^2}\right) \quad (17)$$

where the index  $V$  refers to one of  $\rho^0$ ,  $\omega$ , and  $\phi$ , and  $c_V$  is  $3.079 \cdot 10^{-6}$ ,  $0.287 \cdot 10^{-6}$ , and  $1.450 \cdot 10^{-6} \text{ GeV}^4$  respectively [18].

### AI.d Unitarity condition

In Sections [AI.a-AI.c](#) we discussed the types of decay modes, as enumerated in Section [II.b](#), for which mass-dependent decay widths are calculated in the code, and presented the formalism behind these calculations. The unitarity condition observed by the code in calculating the total resonance width of Eq. (5), and branching ratios, conclude this discussion.

For those decay modes that are identified by the code as any of the above, the decay width is calculated explicitly, as discussed earlier, as a function of mass. The "known" decay modes have an implicit energy threshold, which can be deduced by observation from the formulas of the previous sections, below which their respective decay widths vanish. This implicit threshold is therefore a self-checking condition, and ensures e.g. that calculated mass-dependent partial widths are zero for unrealistically low invariant masses.

Such an implicit condition is lacking for those modes that are not "known" to the code, for which the static branching ratios are used from the data base. In this case a mass-independent branching ratio is used, which could in principle be applied for unrealistically low invariant masses. Therefore an explicit condition, or energy threshold, must be established.

The following ansatz is used throughout the code, regarding energy thresholds: The range for total resonance widths is taken to be  $[m_\gamma - 2\Gamma_\gamma, m_\gamma + 12\Gamma_\gamma]$ , where the mass and width refer to the mass pole and static width from the data base. Total widths are taken to be zero outside of this range. The total width, however, is the sum of partial decay widths as in Eq. (5). Each of the decay modes have individual energy thresholds, as some may be accessible while others inaccessible for a given invariant mass. As threshold for individual decay modes is taken the sum  $E_{thr} = \sum_{i=1}^n m_i$ ,

where  $m_i$  are the mass poles of the  $n$  decay products of the individual mode. The latter condition is imposed not only on the "known" decay modes, but on those whose static branching ratios are used as well. Thus, a decay mode is considered kinematically inaccessible if the available invariant mass falls below the latter threshold.

This discussion can be cast more compactly into the following condition for the (mass-dependent) branching ratio of a decay mode with index  $i$ , as applied in the code:

$$BR_i(m) = \begin{cases} \frac{\Gamma_i(m)}{\Gamma_{tot}(m)}, & m \geq \max(m_\gamma - 2\Gamma_\gamma, E_{thr}^i) \\ 0, & m < \max(m_\gamma - 2\Gamma_\gamma, E_{thr}^i) \end{cases} \quad (18)$$

Note that Eq. (18) is valid for all the decay modes, including those for which the static branching ratios are used. In the latter case the mass dependence enters only through the explicit conditions on the right hand side.

With Eq. (18), the total width of Eq. (5) can be recast into the unitarity condition

$$\sum_{i=1}^n BR_i(m) = 1 \quad (19)$$

observed in the code, where the sum is over the decay modes of any given resonance, and the condition holds in the range  $[m_\gamma - 2\Gamma_\gamma, m_\gamma + 12\Gamma_\gamma]$ .

## AI.e Angular distributions

By default, the code samples scattering angles in the center-of-mass frame of the parent particle isotropically. For a few select channels, however, empirical parametrizations of angular distributions are implemented. These include  $\Delta$  production in  $NN \rightarrow N\Delta$  [13],  $\omega$  production in  $\pi N \rightarrow N\omega$  [14], and  $pp$  elastic scattering [15-16]. The former two are briefly reviewed below. The latter is implemented in the code as an algorithm based on the program SAID, with extensive documentation available at <http://said.phys.vt.edu/>, offered by Richard Arndt.

### AI.e.1 Angular distribution in $NN \rightarrow N\Delta$

This follows Ref. [13], which is in excellent agreement with the data. The direct and exchange matrix elements, averaged over all the spin states, are given by

$$\frac{1}{4} \sum_{\lambda_1 \lambda_2 \lambda_3 \lambda_4} |M(direct)|^2 = \left( \frac{g_\pi f_\pi^*}{m_\pi} \right)^2 \frac{F^4(t, m)}{(t - m_\pi^2)^2} t [t - (m - m_N)^2] \frac{[t - (m + m_N)^2]^2}{3m^2} \quad (20)$$

$$\begin{aligned} \frac{1}{4} \sum_{\lambda_1 \lambda_2 \lambda_3 \lambda_4} (M_a^+ M_\delta + M_\delta^+ M_a) &= \left( \frac{g_\pi f_\pi^*}{m_\pi} \right)^2 \frac{F^2(t, m) F^2(u, m)}{(t - m_\pi^2)(u - m_\pi^2)} \frac{1}{6m^2} [tu + (m^2 - m_N^2)(t + u) - m^4 + m_N^4] \\ &\times [tu + m_N(m + m_N)(m^2 - m_N^2)] [tu - (m^2 + m_N^2)(t + u) + (m + m_N)^4] [tu - m_N(m - m_N)(m^2 - m_N^2)] \end{aligned} \quad (21)$$

where  $f_\pi = 1.008$  and  $f_\pi^* = 2.202$  are the  $\pi N$  and  $\pi \Delta$  coupling constants with all the particles assumed on-shell,  $u$  and  $t$  are the standard Mandelstam variables,  $m$  is the  $\Delta$  resonance mass as sampled from Eq. (6), and  $F(t, m)$  is the mass-dependent form factor

$$F(t, m) = F(t) \sqrt{\frac{m_r}{m} \frac{\beta^2 + q_r^2}{\beta^2 + q^2}} \quad (22)$$

where  $\beta = 300$  MeV,  $q$  and  $q_r$  are as in Eq. (6), and

$$F(t) = \frac{\Lambda^2 - m^2}{\Lambda^2 - t} \quad (23)$$

is the unmodified form-factor, for  $\Lambda = 0.63$  GeV from fits to data [13]. The argument  $t$  in the form factor is one of the Mandelstam variables. Likewise, exchanging  $t$  with  $u$  one gets  $F(u, m)$  in the matrix elements above. The mass dependence in the form factor is slightly modified from the corresponding expression of Ref. [13], the difference coming from the mass-dependent width of Eq. (1) from Refs. [7,8]. These matrix elements result in the following expression for the differential cross section

$$\frac{d\sigma}{dt} = \frac{1}{64\pi} |M|^2 \frac{1}{4I^2} \quad (24)$$

where

$$I = \sqrt{(p_1 p_2)^2 - M_N^4} \quad (25)$$

is a kinematical factor with  $p_{1,2}$  the beam- and target-proton 4-vectors, and  $M_N$  the nucleon mass. In switching from  $d\sigma/dt$  of Eq. (24) to  $d\sigma/d\Omega$ , actually sampled by the code when scattering angles are picked, the extra cm-momentum factor  $p_{cms}$  introduced in Eqs. (7-8) arises naturally. Thus, sampling the  $\Delta$  mass  $m$  from Eq. (7) and subsequently the center-of-mass scattering angle from Eq. (24) yields spectra consistent with differential cross sections, in good agreement with experimental data. The scattering-angle dependence enters through the Mandelstam variables in the matrix elements.

### Al.e.2 Angular distribution in $\pi N \rightarrow N \omega$

This follows Ref. [14], where the angular distribution for  $\omega$  production in  $\pi N \rightarrow N \omega$  in the center-of-mass frame was found to be sharply (exponentially) peaked at forward angles, consistent with the parametrization

$$f(\cos\theta) = 1 + \alpha e^{-4(1-\cos\theta)} \quad (26)$$

where  $\alpha$  depends on the invariant mass, and has been parametrized by Johan Messchendorp as

$$\alpha(\sqrt{s}) = 6.9 \times 10^{-3} e^{2.873\sqrt{s}} \quad (27)$$

by fitting the data of Ref. [14].

## Appendix II: Random sampling

A recurring numerical problem in simulations is random sampling from generally complicated distribution functions either non-integrable analytically, or with a non-invertible integral. The minimum requirement in addressing this problem is access to a reliable random-number generator passing well-established tests of "randomness" (see e.g. Ref. [25]). In **Pluto**<sup>++</sup> the Mersenne Twistor [27] is used, adapted in the native **ROOT** class **TRandom3** from the **CLHEP** library (<http://wwwinfo.cern.ch/asd/lhc++/clhep/>). To streamline the initialization of random-generator modules for standard distributions such as uniform, Gaussian, Breit-Wigner, and Poisson in **Pluto**<sup>++</sup>, in-line functions are provided in the utilities class **PUtills**.

### All.a Distribution functions

For sampling more complex distribution functions, such as those encountered in Appendix I, additional numerical

techniques must be employed. A straightforward approach that is extensively used in [ROOT](#) is sampling from histograms, or memory-resident grids, where the data or analytical distribution functions have previously been stored on a mesh. For one-, two-, and three-variable distribution functions (classes [TF1](#), [TF2](#), and [TF3](#) and the function [GetRandom](#)), on the first call memory-resident integrals are evaluated on a mesh. Sampling is subsequently rapid, form interpolations of the stored data. Nevertheless, these methods only allow the sampling of all the variables simultaneously. For example, for a [TF3](#) function all three variables may only be sampled simultaneously, without the possibility of say fixing one variable and sampling the others. As situations of the latter variety often occur in **Pluto<sup>++</sup>** simulations, the native [ROOT](#) methods are insufficient and the Rejection method (see e.g. Ref. [\[25\]](#)) has been extensively utilized in developing additional algorithms that are employed in the code.

This Section reviews the Rejection method and its implementation in connection with the distribution functions of Appendix [I](#).

### All.a.1 Well-behaved functions

For a physical observable of known physical range  $x \in [x_i, x_k]$  that is described by a distribution function  $p(x)$ , the probability of finding  $x < x_0$  in the range  $x_i < x_0 < x_k$  is

$$P_{<x_0} = \int_{x_i}^{x_0} p(x) dx \quad (28)$$

If it so happens that  $p(x)$  is analytically integrable over the range of  $x$ , then the integral of Eq. [\(28\)](#) yields a function

$$F(x_0) = y \quad (29)$$

If, moreover, Eq. [\(29\)](#) is invertible, namely its inverse  $F^{-1}$  can be obtained, then

$$x_0 = F^{-1}(y) \quad (30)$$

observes the distribution  $p(x)$ . Thus, for a distribution function  $p(x)$  that satisfies Eqs. (28-30), sampling is straightforward provided one has access to a flat random-number generator, using the following recipe:

First, the normalization of the distribution function is determined:

$$N = \int_{x_i}^{x_k} p(x) dx \quad (31)$$

Since the probability of finding  $x$  somewhere within its known physical range is unity,  $y$  of Eq. [\(29\)](#) is between 0 (for  $x_0 = x_i$ ) and  $N$  (for  $x_0 = x_k$ ). A random number  $r$  picked from a flat distribution  $0 < r < 1$  can be assigned the meaning of the probability that  $x$  be below some yet unknown  $x_0$ , namely  $y$  of Eq. [\(29\)](#) can be assigned the value

$$y = Nr \quad (32)$$

Finally  $x_0$  is determined from Eq. [\(30\)](#), and obeys the probability distribution  $p(x)$ .

### All.a.2 Resonance lifetimes

An example of a well-behaved distribution function, such as discussed above, is that of the resonance lifetimes:

$$p(t) = \frac{1}{\tau_0} e^{-t/\tau_0} \quad (33)$$

This is already normalized to unity, i.e.  $\int_0^\infty p(t) dt = 1$ . Following the recipe of the previous section, let  $y \in [0,1]$  be the probability to find  $t$  below a certain value,

$$\int_0^t p(t) dt = y = 1 - e^{-t/\tau_0} \quad (34)$$

and obtain that value by inverting Eq. (34):

$$r = -r_0 \ln(y') \quad (35)$$

where  $y' = 1 - y \in [0,1]$  is a random number sampled from a flat distribution.

This method is used in the [PReaction](#) member function [updateVertex](#), which calculates production vertices for the reaction particles during execution of the event loop.

### All.a.3 Arbitrary distributions and the Rejection Method

Arbitrary distribution functions  $p(x)$  in general cannot be analytically integrated as in Eq. (28), or inverted as in Eq. (30). A function  $f(x) > p(x)$  can always be found, nevertheless, over the range  $x \in [x_i, x_h]$  of  $p(x)$ , that does satisfy Eqs. (28-30). To visualize that this is so one can think of drawing a constant line over  $p(x)$ , which satisfies the required condition. This can be generalized for higher dimensionalities, i.e. multi-variable distribution functions. Having guessed at such a function  $f(x)$ , bounding the distribution function of interest from above over its entire range and having the properties (28-30), it can be shown [25] that those values  $x_0$  of  $f(x)$  obtained as in Eq. (30) are distributed according to  $p(x)$  over its range  $x \in [x_i, x_h]$ , provided

$$r' < \frac{p(x)}{f(x)} \quad (36)$$

is satisfied, where  $r'$  is a random number from a uniform distribution. Thus, so long as a test function  $f(x)$  with the desired properties can be guessed at, Eq. (36) is the effective criterion for picking values of  $x$  that satisfy the distribution function  $p(x)$ . This is the Rejection method, and its implementation for sampling from the distribution functions of Appendix I is discussed next.

### All.b Resonance masses

In two-body hadronic decays the products are assigned masses as discussed in Section [Al.a](#), and in particular Eqs. (7.8). In the most general case where two unstable hadrons are produced, the mass assignments are simultaneous from the distribution function

$$p(m = \sqrt{s}, m_1, m_2) = \begin{cases} \frac{1}{N} g_1(m_1) g_2(m_2) p_{cms}(m = \sqrt{s}, m_1, m_2), & m_1 + m_2 \leq \sqrt{s} \\ 0, & m_1 + m_2 > \sqrt{s} \end{cases} \quad (37)$$

where  $N$  is the normalization factor obtained from

$$N = \int_{m_{min}^1}^{m_{max}^1} dm_1 g_1(m_1) \int_{m_{min}^2}^{m_{max}^2} dm_2 g_2(m_2) p_{cms}(m, m_1, m_2) \mathcal{A}(m \geq m_1 + m_2) \quad (38)$$

following the discussion next to Eqs. (7.8). If one product is stable, say the second, then only the mass of one particle is sampled, and a simplification occurs by dropping  $g_2(m_2)$  and the integral over  $m_2$ . In either case, it should be noted that one of the function variables is the system invariant mass which is fixed by the channel kinematics. Therefore, in practice, an algorithm is sought that permits fixing one variable in a three-, or two-variable function, while simultaneously sampling values for the remaining variables.

This task is performed by the function [sampleM\(\)](#) of the class [PData](#), invoking [sampleM1\(\)](#) and [sampleM2\(\)](#) for the respective cases of one or two unstable particles. This is accomplished by using a "guess" function, and the Rejection-method criterion of Eq. (36), in the case of two unstable projects generalized for two dimensions.

The case with one unstable product is considered first, with  $m_2$  fixed and  $m_1$  to be sampled. We use the notation  $m_l$  for the mass threshold of the unstable hadron, determined from the [ansatz](#) of Section [Al.d](#),  $m_h = \sqrt{s} - m_2$  for the maximum available mass fixed by the kinematics of the decay process, and  $m_r$  for the mass pole from the data base.



The "guess" function  $f(m)$  depends on the range of  $m_1$ :

1. If  $m_h > m_r$ , let  $m_a = (m_h + m_r)/2$  be some intermediate mass between the mass pole and the maximum available mass. We choose as guess function a constant line from threshold to  $m_a$ , and from there to  $m_h$  a mass-dependent line eventually dropping to zero. This is formulated into the relations

$$\begin{aligned} m_i < m \leq m_a: \quad f(m) &= p(\text{scale} \times \sqrt{s}, m_r, m_2) = c_1; \quad a_1 = c_1(m_a - m_i), \\ m_a < m \leq m_h: \quad f(m) &= c_1 - \frac{c_1}{m_h - m_a}(m_h - m); \quad a_2 = \frac{1}{2}c_1(m_h - m_a); \end{aligned} \quad (39)$$

where the mass range, the corresponding guess function, and integral of the latter over its range are shown (units of GeV).

2. If  $m_h > m_r$ , that is the system invariant mass does not quite suffice for the unstable particle mass to reach its mass pole, then only the left (low-mass) tail of the distribution function is accessible, and a constant line over the distribution function is taken as the guess function over the entire range:

$$m_i < m \leq m_h \quad f(m) = \text{scale} \times p(\sqrt{s}, m_r, m_2) = c_2 \quad \text{area} = c_2(m_h - m_i). \quad (40)$$

In both cases  $\text{scale}$  is a free parameter initially set to 1.1, found by trial-and-error to be a good starting value, whose purpose is to lift the guess function sufficiently above the actual distribution function, a condition that is required by the Rejection method. Occasionally a higher value is required for this parameter, whose update is automatically induced by the code upon failure of the algorithm without warning to the user.

In practice the mass is the required outcome of the algorithm, whose range is not known *a priori*, and consequently cannot be used for choosing a suitable guess function from Eqs. (39-40). Instead, a value is first assigned to the integral of the guess function, or area under the curve  $\alpha = \text{area} \cdot r$ , where  $\text{area} = a_1 + a_2$  or  $a_1$  for the two cases (39-40) respectively, and  $0 < r < 1$  is a uniform random number (see introductory comments to Section [AII](#)). A comparison of  $\alpha$  with the total  $\text{area}$  settles the mass range. Finally the integral is inverted retracing the steps of Eqs. (30-32), to yield the mass corresponding to the area assignment  $\alpha$ . A final comparison of the guess with the distribution function evaluated at that value, according to the criterion of Eq. (36), determines whether the mass is valid or rejected, in the latter case leading to the repetition of this ansatz for a new selection of  $\alpha$  until convergence is achieved.

This algorithm, as used in the code, is formulated by restating Eqs. (39-40) as follows:

1. If  $m_h > m_r$  for  $m_a = (m_h + m_r)/2$

$$\begin{aligned} m &= \frac{\alpha}{c_1} + m_i; \quad 0 < \alpha \leq a_1 \\ m &= m_h - \sqrt{\frac{2(\text{area} - \alpha)(m_h - m)}{c_1}}; \quad a_1 < \alpha \leq \text{area} \end{aligned} \quad (41)$$

2. otherwise

$$m = \frac{\alpha}{c_2} + m_i; \quad 0 < \alpha \leq \text{area} \quad (42)$$

In the above case of one-dimensional sampling the guess function was chosen to optimize the efficiency of the sampling algorithm. Whereas any function that caps the distribution function and satisfies Eqs. (30-32) can be used as a guess function for the Rejection method, it is evident from the acceptance criterion of Eq. (36) that the instances of failure, i.e. invalid values of the variable which must therefore be rejected, increases the further the test function is "above" the distribution function. The algorithm is most efficient if a test function can be found such that it satisfies all the required properties, bounds the distribution function from above, but is as close to the latter as possible so as not to fail Eq. (36) often. In the case of one-dimensional sampling a constant value for the test function leading up to the mass pole is not a bad choice, since the rise of the distribution function (left tail) for hadronic resonances is typically very sharp. Past the mass pole the falloff is generally much more gradual involving long tails, and a constant guess function would have been rather inefficient. Thus a mass-dependent line equation dropping to zero at some safely distant limit is a better choice.

Such optimization is difficult to achieve in two-dimensional sampling with two unstable hadrons, since a "guess plane" must be found as test function whose shape is generally complicated. Thus, in this case, a constant plane is used. Using the same notation as before (see the comments preceding Eq. (39)), for  $[m_e^i, m_h^i]$  the valid mass range and  $m_\gamma^i$  the mass poles of the particles with indices  $i=1,2$ , the sampling algorithm is summarized in the relations:

$$\begin{aligned}
f(m_1, m_2) &= p(\text{scale} \times \sqrt{s, m_x^1, m_x^2}) = c; \quad m_x^i = \min(m_x^i, m_k^i), \quad i \in [1, 2] \\
\text{area} &= c \times (m_k^1 - m_\ell^1), \quad a = r_1 \times \text{area}, \quad 0 \leq r_1 \leq 1 \\
\text{volume} &= \text{area} \times (m_k^2 - m_\ell^2), \quad v = r_2 \times \text{volume}, \quad 0 \leq r_2 \leq 1 \\
m_1 &= m_\ell^1 + \frac{a}{c} \\
m_2 &= m_\ell^2 + \frac{v}{a}
\end{aligned} \tag{43}$$

Despite the lack of optimization this is still a rapidly converging algorithm, as illustrated in the example of Fig 3 for two-dimensional sampling in what is likely to be the most complicated and time-consuming type of mass sampling to be encountered. It should be noted that the CPU time quoted there includes the calculation of total and partial widths in [PData](#), which themselves represent the most complicated cases likely. In one dimension the required CPU is a small fraction of that of two-dimensional sampling.

### All.c Dalitz-dilepton mass

Pseudoscalar-meson  $\pi^0, \eta, \eta' \rightarrow e^+ e^- \gamma$ , vector-meson  $\omega \rightarrow e^+ e^- \pi^0$ , and  $\Delta$  resonance  $\Delta \rightarrow e^+ e^- N$  Dalitz decays were discussed in Sections [Al.b.1](#), [Al.b.2](#), and [Al.b.3](#), where the distribution functions of the massive virtual photons produced in these channels were also presented. The physical range of dilepton masses is  $m \in [2m_e, m_{res} - m_X]$ , where  $m_X$  is the mass of the dilepton's partner (zero in the case of pseudoscalar Dalitz decay). The notation  $\Delta m = m_{res} - m_X - 2m_e$  is used, with  $m_{res}$  the parent mass pole.

Repeating the steps of the previous Section, a uniform random number  $r$  helps assign a value  $\alpha = \text{area} \cdot r$  to the integral of a guess function  $f(m)$ , associated with the probability of finding a mass up to  $m_0$

$$\alpha = \int_{2m_e}^{m_0} f(m) dm \tag{44}$$

then  $m_0$  is obtained by inverting the equation, and the criterion of Eq. (36) determines whether it is accepted or not.

The test function

$$f(m) = \frac{\Delta m}{m} \tag{45}$$

is applied in the cases of  $\pi^0$ ,  $\eta$ , and  $\Delta$ , as well as  $\omega$  Dalitz decays, for the latter case only if  $m_\omega \leq m_{res}$ . The resulting test mass is

$$m_0 = \frac{m_{res}^r}{(2m_e)^{r-1}} \tag{46}$$

where  $r$  is the uniform random number of the area assignment, and  $m_{res}$  the mass pole of the parent.

For  $\eta'$  Dalitz decay the test function is

$$f(m) = \frac{\Delta m}{m} + \frac{\Delta m}{m_{res} - m + 2m_e} \tag{47}$$

yielding the mass

$$m_0 = \frac{m_{\eta'} + 2m_e}{1 + (2m_e/m_{\eta'})^{2r-1}} \tag{48}$$

Last, for  $\omega$ -Dalitz decay and  $m_{res} < m_\omega$ , two test-function profiles are used:

$$f_1(m) = \frac{\Delta m}{m} + \frac{\beta}{m_\omega - m + 2m_e - m_X} \quad \beta = (m_\omega - \Delta m) \cdot p(m_{res} - m_X) \tag{49}$$

if  $2m_e \leq m \leq m_{res} - m_X$ , or

$$f_2(m) = f_1(m_{res} - m_X) \cdot \frac{m_\omega - m_X - m}{m_\omega - m_{res}} \quad (50)$$

if  $m_{res} - m_X < m \leq m_\omega - m_X$ .

The index  $res$  in Eqs (49-50) denotes the  $\omega$ -meson mass pole, whereas the index  $\omega$  refers to the actual sampled mass.

These two respective regions yield areas:

$$area_{m \leq m_{res} - m_X} = (\Delta m + \beta) \cdot \ln \left( \frac{m_{res} - m_X}{2m_e} \right) \quad 2m_e \leq m \leq m_{res} - m_X \quad (51)$$

and

$$area_{m > m_{res} - m_X} = f_1(m_{res} - m_X) \cdot \frac{m_\omega - m_{res}}{2} \quad m_{res} - m_X < m \leq m_\omega - m_X \quad (52)$$

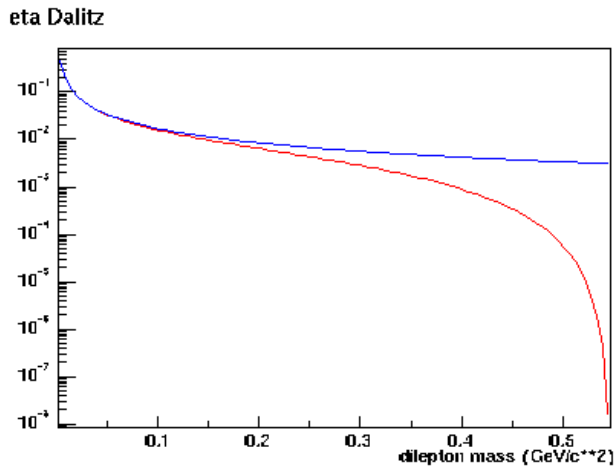
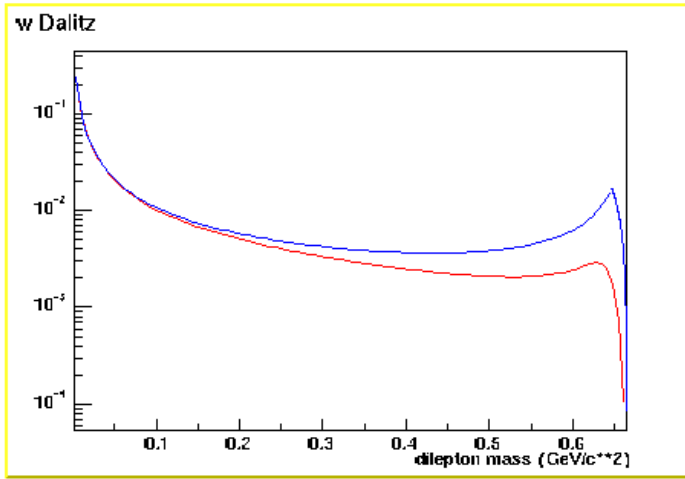
For a random number  $r$ ,  $\alpha = area \cdot r$ , and total  $area = area_{m \leq m_{res} - m_X} + area_{m > m_{res} - m_X}$ , the two regions yield

$$m_0 = 2m_e e^{\frac{\alpha}{\Delta m + \beta}} \quad (53)$$

$$m_0 = m_\omega - m_X - \sqrt{\frac{2 \cdot area \cdot (m_\omega - m_{res}) \cdot (1-r)}{\beta}} \quad (54)$$

The distribution and test functions are illustrated as red and blue curves respectively in Fig. 8, for  $\eta'$  (bottom panel) and  $\omega$  Dalitz decay (top panel), the latter for  $m_{res} < m_\omega$ , i.e. Eqs. (49-50).

**Figure 8** Virtual-photon mass sampling in Dalitz decays. The actual VMD distribution functions are shown (red) for  $\omega$  (top) and  $\eta'$  (bottom) Dalitz decay, together with the corresponding guess functions (blue). The efficiency of the sampling algorithm improves for a choice of test function that bounds the distribution function, as required, but does not overshoot it by far.



### All.d Scattering angles

The  $NN \rightarrow N\Delta$  channel was discussed in Section A1.d. The angular distribution of Eq. (24) is approximated by the test function

$$f(x) = b_1 + 3b_3|x| \quad (55)$$

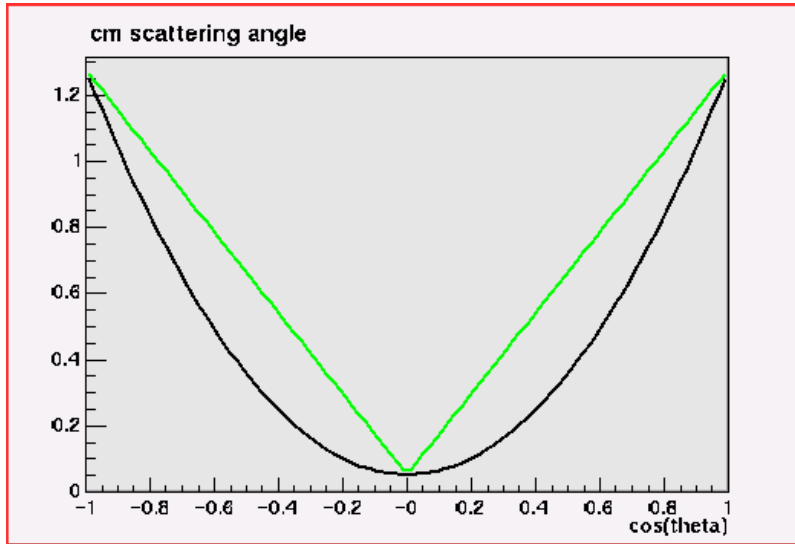
where  $x = \cos \theta$  over the range  $x \in [-1, 1]$  for  $\theta \in [0, 180](\text{deg})$ ,  $b_1 = d\sigma/dt(x=0)$ , and  $b_2 = d\sigma/dt(x=1) - b_1$ , and  $d\sigma/dt$  from Eq. (24).

This yields an integral  $area = 2b_1 + 3b_3$  over the entire range. If  $r$  is a uniform random number between 0 and 1, and  $\alpha = area \cdot r$ , then

$$x_0 = \pm \frac{b_1}{3b_3} \left( 1 - \sqrt{1 \pm \frac{3b_3}{b_1^2} \cdot area \cdot (1 - 2r)} \right) \begin{cases} + : \alpha \leq area/2 \\ - : \alpha > area/2 \end{cases} \quad (56)$$

is distributed according to Eq. (24), as long as the criterion of Eq. (36) is satisfied. The distribution and test functions are illustrated in Fig. 10 as black and green curves respectively, for the invariant mass fixed to  $2.4 \text{ GeV}/c^2$ .

**Figure 9** Scattering-angle sampling is illustrated for the production angle of the  $\Delta$  resonance in  $NN \rightarrow N\Delta$ , in the center-of-mass rest frame with invariant mass set to  $2.4 \text{ GeV}/c^2$ . The black curve represents the actual distribution function, and the green curve the test function. The proximity of the two results in an efficient sampling algorithm.



Similar techniques are applied for  $\pi\pi$  production, Eqs. (26-27), and the  $pp$ -elastic scattering, the latter implemented in the class [PSaid](#).

## Appendix III: Detector-dependent aspects

The package is predominantly designed for stand-alone principle simulations, although further processing of generated events with [HGeant](#) is possible and a standard interface is supported. Detailed detector geometries are intended for the most part as user-defined macros and derived classes. Nevertheless, some basic tools are supplied, as for example the class [PFilter](#) for geometrical acceptance cuts, and the possibility to attach user **selection** and/or **analysis** functions in [PReaction](#).

The [HADES](#) pion-beam profile is also available, for realistic pion-induced reactions, taking into account the dispersion and resolution of the beam. Due to the structure of the code the physics of an interaction, including the beam profile, are contained in the [PChannel](#) class. The beam-sampling option was introduced in Section III.c. The momentum profile for [HADES](#) charged-pion beams has been studied recently [20]. The dispersion and resolution were found to be

$$\frac{\Delta p}{p} = \pm 4\% \quad (57)$$

$$\sigma = 0.003 \cdot p_0 \quad (58)$$

where the central momentum is in the z-direction. Beam sampling may be enabled for the entry channel if the beam is a charged pion, provided the beam particle is declared with only a z-component of momentum (see the related [discussion](#) in Section III.c).

A skeleton class is also included ([PHADES](#)), which can accommodate additional user-defined [HADES](#)-specific features, containing a "smearing" function that introduces an uncertainty to charged-particle 4-momenta due to the spectrometer resolution for the setup of Ref. [21].

Two additional classes, written by James Ritman and Marc-André Pleier, [PTrackH](#) for [HADES](#) and [PTrack](#) for [Crystal Barrel](#), are included in the distribution. The former, for example, defines [HADES](#) components such as the MDC planes in a straightforward simple manner as plane equations, and checks whether charged-particle trajectories of simulated events intersect the planes to determine acceptance. Likewise, the bending of charged-particle trajectories due to the [HADES](#) field is approximated by a transverse momentum "kick" between the second and third planes. Such simplified models are often realistic enough, if the full detail of an [HGeant](#) simulation is not necessary, and have the advantage that setting up, executing, and analyzing a high-statistics simulation is accomplished with **Pluto**<sup>++</sup> on-line, in a matter of minutes.

---

\*The current release is **4.06** of 20.01.2005

Document last updated on 18.3.2005 [R.Holzmann](#)