Eur. Phys. J. C manuscript No.
(will be inserted by the editor)

CERN-PH-TH/2014-NNN, LPSC-NNN, LAPTH-NNN,
MCNET-14-NN
May 6, 2014

Designing and recasting LHC analyses with MadAnalysis 5

Eric Conte[1], Béranger Dumont[2], Benjamin Fuks[3,4], Chris Wymant[5]

[1] Groupe de Recherche de Physique des Hautes Énergies (GRPHE), Université de Haute-Alsace, IUT Colmar, 34 rue du Grillenbreit BP 50568, 68008 Colmar Cedex, France
[2] Laboratoire de Physique Subatomique et de Cosmologie, UJF Grenoble 1, CNRS/IN2P3, INPG, 53 Avenue des Martyrs, F-38026 Grenoble, France
[3] Theory Division, Physics Department, CERN, CH-1211 Geneva 23, Switzerland
[4] Institut Pluridisciplinaire Hubert Curien/Département Recherches Subatomiques, Université de Strasbourg/CNRS-IN2P3, 23 Rue du Loess, F-67037 Strasbourg, France
[5] Laboratoire d'Annecy-le-Vieux de Physique Théorique, 9 Chemin de Bellevue, F-74941 Annecy-le-Vieux, France

Received: date / Accepted: date

Abstract. We present an extension of the expert mode of the MadAnalysis 5 program dedicated to both the design and the reimplementation of physics analyses such as those performed in the context of high-energy physics collider experiments. In this document, we extensively detail the large set of predefined classes, functions and methods available to the user and emphasize the main novelties that include, among others, the possible definition of multiple selection regions and a user-friendly treatment for the implementation of event selection criteria. We illustrate the strengths of this approach with the examples of a reinterpretation of one CMS supersymmetry search and the design of one hadronically decaying monotop analysis.

1 Introduction

Within any given experimental analysis at the Large Hadron Collider (LHC) at CERN, reduction of the data-recording rate to a technically feasible level and the discarding of events irrelevant to the specific physics question being investigated both necessitate the implementation of selection criteria, widely referred to as *cuts*. At the experimental level, it is important to distinguish between two classes of cuts, those imposed at the trigger level and those imposed offline: events failing the former are not recorded at all (the information is lost), whereas events failing the latter are merely not considered for the final analysis. This distinction is nevertheless less important for the purposes of the reinterpretation of an analysis in the context of any sample of events other than real observed data, notably events generated by Monte Carlo simulations of collisions to be observed in the framework of a given physics model. In this case, both types of cuts amount to conditions on whether a given generated event is considered in the analysis or not. However, the reinterpretation of an analysis in general requires *de novo* implementation of the full set of cuts. Several frameworks have recently been released with this aim [1–3]. An alternative approach, which does not rely on event simulation, is also available and uses results published by the experimental collaborations in the context of so-called Simplified Models Spectra, as in the works of Refs. [4, 5]. While much faster, it is, however, less general.

In this work, we focus on the expert mode of running of the MadAnalysis 5 program [2,6] allowing one to implement any analysis based on a cut-and-count flow (in contrast to analyses relying on multivariate techniques) and investigate the associated effects on any Monte Carlo event sample. In this way, the program is suitable for both the reinterpretation of any existing analysis, and for the design of prospective analyses aiming to study specific physics effects.

While the implementation of an analysis has been greatly facilitated by the series of predefined functions and methods included in the SampleAnalyzer library shipped with the package, this task is often complicated by the presence of sub-analyses which we refer to as *regions* (with a nod to the terms *signal* and *control regions* commonly used in searches for physics beyond the Standard Model). The complication finds arises from the internal format handled by SampleAnalyzer, which assumes the existence of a single region. While this condition is convenient for prospective works yielding the design of new analyses, it is rarely fulfilled by existing analyses that one may want to recast. Therefore in order to allow the user to both design and recast analyses, we have extended the SampleAnalyzer internal format to support analyses with multiple regions defined by different sets of cuts. We have also expanded the code with extra methods and routines to facilitate the implementation of more complex analyses by the user.

In the context of analyses which effectively contain sub-analyses, a further useful classification of cuts can be made: namely into those which are common/shared by different regions, and those which are not, the latter serving to define the different sub-analyses themselves. Figure 1 schematically illustrates an analysis containing four regions, which are defined by two region-specific cuts imposed after two common cuts. Some thought is required concerning the best way to capture in an algorithm the set of selection requirements shown in Figure 1.

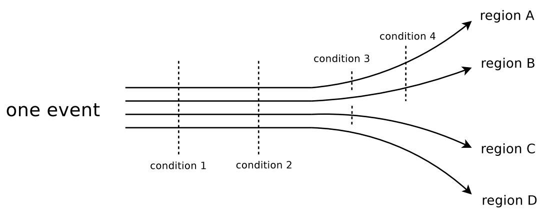

Fig. 1. Schematic illustration of the definition of different regions in which a given event can be counted (or not), based on different combinations of selection cuts.

For the common cuts (cuts 1 and 2 on the figure) this is clear: if the selection condition is failed, the event is vetoed (*i.e.*, we ignore it and move on to analyzing the next event). Thereafter we have two conditions to check (cuts 3 and 4), but they have different applicabilities to different regions. In terms of pseudo-code the most obvious, although not the most efficient, method of implementing this (ignoring the common cuts) is

```
count the event in region D
if (condition 3)
{
  count the event in region C
  if (condition 4)
  {
    count the event in region A
  }
}
if (condition 4)
{
  count the event in region B
}
```

One important drawback of this naive approach is the duplication of the check of the fourth condition. In the simple style of implementation of the cuts above, this is unavoidable: condition 4 must be checked both inside and outside the scope of condition 3. With the two region-specific cuts that we have here, there is only one such clumsy duplication present in the code. However as the number of such cuts grows, the situation rapidly gets worse. For instance, considering growing the decision tree shown in Figure 1 to include N region-specific cuts, combined in all possible permutations to define 2^N regions would deepen the nesting of the above pseudo-code and lead to $2^N - (N+1)$ unnecessary duplications of checks. Moreover, each of those needs to be carefully implemented by the user in the correct scope, a task becoming less and less straightforward for large values of N.

Ideally the algorithm should be structured so that there is *no* unnecessary duplication. Such a framework for easily and efficiently dealing with multiple regions is one of the new features of the latest version of SampleAnalyzer, the C++ core of the MadAnalysis 5 program. Both can be obtained from the MadAnalysis 5 website,

`https://launchpad.net/madanalysis5`

and all the features described in this paper are available from version 1.1.10 of the code onwards. This document supersedes the previous version of the manual for the expert mode of the program. Several examples of analysis, including those presented in this work, can be found within our Public Analysis Database [7].

The remainder of this paper is organized as follows. In Section 2, we recall the basic functionalities of MadAnalysis 5 allowing implementation of physics analyses in the expert mode of the program, which has been extended according to the needs of our users; we also introduce the new features of the SampleAnalyzer kernel. Two examples are provided in Section 3: the first is related to the reimplementation of a CMS analysis dedicated to a search for a supersymmetric partner of the top quark in monoleptonic events [8], and the second to the design of a monotop analysis where the monotop system decays in the hadronic mode [9]. Our conclusions are presented in Section 4.

2 The expert mode of MadAnalysis 5

In this section we present the manner in which physics analyses are implemented in the expert mode of MadAnalysis 5. We begin with a discussion of the creation of an analysis template (Section 2.1) and the addition of several analyses to the same template (Section 2.2). We then discuss the methods and classes allowing effective implementation of an analysis (Section 2.3), its compilation and execution (Section 2.4) and finally the structure of the output files (Section 2.5).

2.1 Creation of an analysis template

In the *expert mode* of the program, the user is asked to write his/her analysis in C++, using all the classes and methods of the SampleAnalyzer library. To begin implementing a new analysis, the user is recommended to use the Python interpreter of MadAnalysis 5 to create a working directory. This is achieved by starting MadAnalysis 5 as

```
./bin/ma5 -E
```

and by then following the instructions displayed on the screen – the user is asked for the names of the working directory and of his/her analysis, which we denote by `name` in the rest of Section 2. The directory that has been created contains three subdirectories: the `Input`, `Output` and `Build` directories.

The use of the `Input` directory is optional. It has been included in the analysis template in the aim of having a unique structure for both the normal and expert modes of MadAnalysis 5. In the normal mode, its purpose is to collect text files with the lists of paths to the event samples to analyze. The `Output` directory has been conceived to store the results of each execution of the analysis. The `Build` directory includes a series of analysis-independent files organized into several subdirectories, together with files to be modified by the user.

At the root of the `Build` directory, one can find one bash script together with its tcsh counterpart. Each of these scripts allows one to correctly set up the environment variables necessary to compile and execute an analysis within the MadAnalysis 5 framework. They can be initiated by typing in a shell the respective commands

```
source setup.sh      source setup.csh
```

A `Makefile` has also been created so that the standard commands

```
make      make clean      make mrproper
```

can be used to (re)compile the analysis. The final executable is obtained from two pieces – a library and the main program. The library originates from the merging of the SampleAnalyzer library and the analysis of the user, and is stored in the subdirectory `Build/Lib`. The main program is located in the `Build/Main` subdirectory and has been conceived following a simple structure. It first initializes the analysis, then runs the analysis over all events (possibly collected into several files) and eventually finalizes the results in the `Output` directory abovementioned.

The `Build` directory also contains the `SampleAnalyzer` subdirectory that stores the source and header files associated with the analysis under implementation (`Analyzer/name.cpp` and `Analyzer/name.h`), together with a Python script, `newAnalyzer.py`, dedicated to the implementation of several analyses into a single working directory. The `Analyzer` subdirectory additionally includes a list with all analyses implemented in the current working directory (`analysisList.cpp`). More information about those files is provided in the next subsections.

2.2 Merging several analyses in a single working directory

In Section 2.1, we have explained how to create a working directory containing a single analysis that is called, in our example, `name`. The analysis itself is implemented by the user in a pair of files `name.cpp` and `name.h`, which should be consistently referred to in the file `analysisList.cpp`. In addition, the main program (the file `Build/Main/main.cpp`) takes care of initializing and executing the analysis. This controls the implementation of a further new analysis – `newname` for the sake of the example – in the same working directory. This new analysis has to be written in C++ in the two files `newname.cpp` and `newname.h` (stored in the `Build/SampleAnalyzer/Analyzer` directory) and referred to in the `analysisList.cpp` file. Furthermore, the main program has to be modified so that it could initialize and execute the new analysis, simultaneously with the primary implemented analysis `name`.

All these tasks have been automated (with the exception of the implementation of the analysis itself) so that the user is only required to run the python script `newAnalyzer.py` by typing in a shell the command

```
./newAnalysis.py newname
```

from the `Build/SampleAnalyzer` directory.

2.3 Implementing an analysis in the MadAnalysis 5 framework

2.3.1 General features

As briefly sketched in the previous subsections, the implementation of a specific analysis within the MadAnalysis 5 framework consists of providing the analysis C++ source and header files `name.h` and `name.cpp`.

The header file contains the declaration of a class dedicated to the analysis under consideration. This class is defined as a daughter class inheriting from the base class `AnalysisBase`, and includes, in addition to constructor and destructor methods, three functions to be implemented by the user (in the source file `name.cpp`) that define the analysis itself. The first of these, dubbed `Initialize`, is executed just once prior to the reading of the user's set of events. In particular, it allows one both to declare selection regions and to associate them with a series of cuts and histograms. The second method, named `Execute`, is the core of the analysis and is applied to each simulated event provided by the user. It takes care of, among others, object definitions, the application of the selection cuts and the filling of the various histograms. Finally, the last function, `Finalize`, is called once the reading and analysis of all the events have been achieved. Apart from this, the user is obviously allowed to define his/her own set of functions and variables according to his/her purposes.

The splitting of the analysis into regions, the application of the selection criteria, and the filling of histograms are all controlled through the automatically initialized object `Manager()` – a pointer to an instance of the class `RegionSelectionManager`. The member methods of this class are listed in Table 1 and will be detailed in the next subsections in which we provide guidelines for the implementation of the functions `Initialize`, `Execute` and `Finalize` in the C++ source file `name.cpp`.

2.3.2 Initialization of an analysis

When the analysis is executed from a shell, the program first calls the `Initialize` method before starting to analyze one or several event samples.

Before detailing how to declare regions, histograms and cuts, we first encourage the user to include an electronic signature to the analysis that he/she is implementing and to ask the program to display it to the screen. Although this is neither mandatory nor standardized, this improves the traceability of a given analysis and provides information to the community about who has implemented the analysis and which reference works have been used. We strongly recommend including at least the names and e-mail addresses of the authors, a succinct description of the analysis and related experimental notes or publications. Taking the example of the CMS stop search in monoleptonic events [8] presented in Section 3, an electronic signature could possibly be

```
INFO << "Analysis: CMS-SUS-13-011, arXiv:1308.1586"
     << " (stop search, single lepton)\n";
INFO << "Authors: Conte, Dumont, Fuks, Wymant\n";
INFO << "E-mails: " << "conte@iphc.cnrs.fr, "
     <<             "dumont@lpsc.in2p3.fr, "
     <<             "fuks@cern.ch, "
     <<             "wymant@lapth.cnrs.fr\n";
INFO << "DOI: xx.yyyy/zzz\n";
INFO << "Please cite arXiv:YYMM.NNNN [hep-ph]\n";
```

where the last two lines refer to the Digital Object Identifier [10] of the analysis code (if available) and the physics publication for which this analysis reimplementation has been developed. The sample of code above also introduces the `INFO` message service of the SampleAnalyzer framework, which is detailed in Section 2.3.7.

As already mentioned, each analysis region must be properly declared within the `Initialize` function. This is achieved by making use of the `AddRegionSelection` method of the `RegionSelectionManager` class. This declaration requires provision of a name (as a string) which serves as a unique identifier for this region both within the code itself (to link the

`AddCut(...)`	Declares a cut and links it to a set of regions. A single string may be passed as an argument, corresponding to the user-defined name of one of the selection cuts of the analysis. If no other argument is provided, the cut is associated with all declared signal regions. Otherwise, a single string or an array of strings (the names of the relevant regions) can be specified.
`AddHisto(...)`	Declares an histogram. The first argument consists of the name of the histogram, the second one is an integer number determining the numbers of bins, the third and fourth arguments are the lower and upper bounds of the x-axis (given as floating-point numbers) and the optional last argument links all or some of the declared regions to the histogram (see the `AddCut` method for more information on this feature).
`AddRegionSelection(...)`	Declares a new region. This method takes a string, corresponding to a user-defined name for a region, as its argument.
`ApplyCut(...)`	Applies a given cut when called. This method takes two mandatory arguments. The first is a boolean variable whose value has already been set to `true` or `false`. This indicates whether the selection requirement associated with a given cut is satisfied. The second argument is the name of the considered cut, provided as a string. The method returns a boolean quantity specifying whether at least one region defined anywhere in the analysis is still passing all cuts so far (`true`) or not (`false`).
`FillHisto(...)`	Fills an histogram. The first argument is a string specifying the name of the considered histogram, and the second is a floating-point number providing the value of the observable being histogrammed.
`InitializeForNewEvent(...)`	To be called prior to the analysis of each event. This methods tags all regions as surviving the cuts, and initializes the weight associated with the current event to a user-defined value passed as an argument (given as a floating point precision number).
`SetCurrentEventWeight(...)`	Modifies the weight of the current event to a user-defined value passed as an argument (given as a floating-point number).

Table 1. Methods of the `RegionSelectionManager` class.

region to cuts and histograms) and in the output files that will be generated by the program. For instance, the declaration of two regions, dedicated to the analysis of events with a missing transverse energy $\not{E}_T > 200$ GeV and 300 GeV could be implemented as

```
Manager()->AddRegionSelection("MET>200");
Manager()->AddRegionSelection("MET>300");
```

As shown in these lines of code, the declaration of the two regions is handled by the `Manager()` object, an instance of the `RegionSelectionManager` class that is automatically included with any given analysis. As a result, two new regions are created and the program internally assigns the intuitive identifiers `"MET>200"` and `"MET>300"` to the respective regions.

Once all regions have been declared, the user can continue to the declaration of cuts and histograms. As for regions, each declaration requires a string name which acts as an identifier in the code and the output. Histogram declaration also asks for the number of bins (an integer number) and the lower and upper bounds of the range of the x-axis to be drawn (two floating-point numbers) to be specified. Both types of object must also be associated with one or more regions. In the case of cuts, this finds its source at the conceptual level: each individual region as distinct from the others is *defined* by its unique set of cuts. In the case of histograms, this allows one to establish the distribution of a particular observable *after* some region-specific cuts have been applied. The association of both types of object to their regions follows a similar syntax, using an optional argument in their declaration. This argument is either a string or an array of strings, each being the name of one of the previously declared regions. If this argument is absent, the cut/histogram is automatically associated with all regions. This feature can be used, for example, for preselection cuts that are requirements for considering an event in the analysis common to all regions.

As an illustrative example, the code

```
Manager()->AddCut("1lepton");
std::string SRlist[] = {"MET>200", "MET>300"};
Manager()->AddCut("MET>200 GeV",SRlist);
```

would create two preselection cuts, `"1lepton"` and `"MET>200 GeV"`, and assign them to the two previously declared regions `"MET>200"` and `"MET>300"`. Although both cuts are associated with both regions, for illustratory purposes we have shown two methods of doing this – using the syntax for automatically linking to all regions (here there are two) and explicitly stating both regions. As a second example, we focus on the declaration of an histogram of 20 bins representing the transverse momentum distribution of the leading lepton $p_T(\ell_1)$ in the range $[50, 500]$ GeV. In the case where the user chooses to associate it with the second region only, the line

```
Manager()->AddHisto("ptl1",20,50,500,"MET>300"};
```

must be added to the analysis code.

Finally, the `Initialize` function can also be used for the initialization of one or several user-defined variables that have been previously declared in the header file `name.h`.

`mc()->beamE().first`	Returns, as a floating-point number, the energy of the first of the colliding beams.
`mc()->beamE().second`	Same as `mc()->beamE().first` but for the second of the colliding beams.
`mc()->beamPDFauthor().first`	Returns, as an integer number, the identifier of the group of parton densities that have been used for the first of the colliding beams. The numbering scheme is based on the PdfLib [11] and LhaPdf [12] packages.
`mc()->beamPDFauthor().second`	Same as `mc()->beamPDFauthor().first` but for the second of the colliding beams.
`mc()->beamPDFID().first`	Returns, as an integer number, the code associated with the parton density set (within a specific group of parton densities) that has been used for the first of the colliding beams. The numbering scheme is based on the PdfLib [11] and LhaPdf [12] packages.
`mc()->beamPDFID().second`	Same as `mc()->beamPDFID().first` but for the second of the colliding beams.
`mc()->beamPDGID().first`	Returns, as an integer number, the Particle Data Group identifier defining the nature of the first of the colliding beams. The numbering scheme is based on the Particle Data Group review [13].
`mc()->beamPDGID().second`	Same as `mc()->beamPDGID().first` but for the second of the colliding beams.
`mc()->weightingmode()`	Returns, as an integer number, information on the weighting strategy used by the event generator [14, 15].
`mc()->processes()`	Returns a vector of instances of the `ProcessFormat` class associated with the set of subprocesses described by the sample. A `ProcessFormat` object contains information about the process identifier fixed by the generator (an integer number accessed via the `processId()` method), the associated cross section in pb (a floating-point number accessed via the `xsection()` method) and the related uncertainty (a floating-point number accessed via the `xsection_method()`), and the maximum weight carried by any event of the sample (a floating-point number accessed via the `maxweight()` method).
`mc()->xsection()`	Returns, as a floating-point number, the cross section in pb associated with the event sample.
`mc()->xsection_error()`	Returns, as a floating-point number, the (numerical) uncertainty on the cross section associated with the event sample.

Table 2. Methods of the `SampleFormat` class.

2.3.3 Using general information on Monte Carlo samples

Simulated events can be classified into two classes: Monte Carlo events either at the parton or at the hadron level, and reconstructed events after object reconstruction[1]. Monte Carlo (and not reconstructed) event samples in general contain global information on the sample, such as cross section, the nature of the parton density set that has been used, *etc.*. In the MadAnalysis 5 framework, these pieces of information are collected under the form of instances of the `SampleFormat` class and can be retrieved by means of the methods provided in Table 2.

The function `Execute` that must be implemented by the user takes, as a first argument, a `SampleFormat` object associated with the current analyzed sample. In this way, in the case where the sample is encoded within the Lhe [14, 15] or StdHep [16] format, the user is allowed to access most of the available information passed by the event generator. In contrast, the other event formats supported by MadAnalysis 5, namely the HepMc [17], Lhco [18] and (Root-based [19]) Delphes 3 [20] format, do not include any information of this kind so that the first argument of the `Execute` function is a null pointer. In the case where the user may need such information, it will have to be included by hand.

For instance, if we assume that we analyze an event sample containing $N = 10000$ events (N being stored as a double-precision number in the `nev` variable), the weight of each event could be calculated (and stored in the `wgt` variable for further use within the analysis) by means of the code sample

```
double lumi  = 20000.;
double nev   = 10000.;
double wgt = MySample.mc()->xsection()*lumi/nev;
```

The `MySample` object is an instance of the `SampleFormat` class associated with the sample being analyzed and we impose the results to be normalized to 20 fb^{-1} of simulated collisions (stored in pb^{-1} in the `lumi` variable).

2.3.4 Internal data format for event handling

In the SampleAnalyzer framework, both Monte Carlo and reconstructed events are internally handled as instances of a class named `EventFormat`. At the time of the execution of the

[1] Strictly speaking, there exists a third class of events once detector simulation has been included. In this case, the event final state consists of tracks and calorimeter deposits. MadAnalysis 5 has not been designed to analyze those events and physics objects such as (candidate) jets and electrons must be reconstructed prior to be able to use the program.

`ctau()`	Returns, as a floating-point number, the lifetime of the particle in millimeters.
`daughters()`	Returns, as a vector of pointers to `MCParticleFormat` objects, a list with the daughter particles that are either produced from the decay of the considered particle or from its scattering with another particle.
`momentum()`	Returns, as a `TLorentzVector` object [19], the four-momentum of the particle. All the properties of the four-momentum can be accessed either from the methods associated with the `TLorentzVector` class, or as direct methods of the `MCParticleFormat` class, after removing the capital letters from the method name. For instance, `pt()` is equivalent to `momentum().Pt()`. In addition, the methods `dphi_0_2pi(...)` and `dphi_0_pi(...)` return the difference in azimuthal angle normalized in the $[0, 2\pi]$ and $[0, \pi]$ ranges, respectively, between the particle and any other particle passed as an argument, whereas `dr(...)` returns their angular distance, the second particle being provided as an argument as well.
`mother1()`	Returns, as a pointer to an `MCParticleFormat` object, the mother particle of the considered particle.
`mother2()`	Returns, as a pointer to an `MCParticleFormat` object, the second mother particle of the considered particle in the case where it is produced from the scattering of two particles.
`pdgid()`	Returns, as an integer number, the Particle Data Group identifier defining the nature of the particle. The numbering scheme is based on the Particle Data Group review [13].
`spin()`	Returns, as a floating-point number, the cosine of the angle between the three-momentum of the particle and its spin vector. This quantity is computed in the laboratory reference frame.
`statuscode()`	Returns, as an integer number, an identifier fixing the initial-, intermediate- or final-state nature of the particle. The numbering scheme is based on Ref. [14].
`toRestFrame(...)`	Boosts the four-momentum of the particle to the rest frame of the one of a second particle (an `MCParticleFormat` object given as argument). The method modifies the momentum of the particle.

Table 3. Methods of the `MCParticleFormat` class.

analysis on a specific event, the `Execute` function receives such an `EventFormat` object as its second argument. The properties of this object reflect those of the current event and can be retrieved via the two methods,

```
event.mc()        event.rec()
```

which return a pointer to an `MCEventFormat` object encompassing information at the Monte Carlo event level, and a pointer to a `RecEventFormat` object specific for managing information at the reconstructed event level, respectively.

Focusing first on Monte Carlo events, the `MCEventFormat` class allows one to retrieve properties of all initial-state, intermediate-state and final-state particles of a given event. Particles are encoded as instances of the `MCParticleFormat` class whose associated methods are shown in Table 3. Additionally, general event information, such as the values for the gauge couplings or the factorization scale used, is also available if properly stored in the event file. Finally, the `MCEventFormat` class also contains specific methods for the computation of four global event observables: the amount of (missing) transverse energy E_T ($\not\!\!E_T$) and of (missing) transverse hadronic energy H_T ($\not\!\!H_T$). These quantities are calculated according to

$$E_T = \sum_{\text{visible particles}} |\boldsymbol{p}_T| , \qquad H_T = \sum_{\text{hadronic particles}} |\boldsymbol{p}_T| ,$$
$$\not\!\!E_T = |\not\!\boldsymbol{p}_T| = \Big| - \sum_{\text{visible particles}} \boldsymbol{p}_T \Big| , \tag{1}$$
$$\not\!\!H_T = |\not\!\!\boldsymbol{H}_T| = \Big| - \sum_{\text{hadronic particles}} \boldsymbol{p}_T \Big| ,$$

once the user has defined, in the initialization part of the analysis, which particles are invisible and which ones are hadronizing (by means of the configuration functions described in Section 2.3.8). The entire set of properties that can be employed to analyze a Monte Carlo event is shown in Table 4.

For example, a function dedicated to the selection of all the final-state electrons and positrons that are present in an event and whose transverse momentum is larger than 50 GeV could be implemented as

```
std::vector<const MCParticleFormat*> electrons;

for(unsigned int i=0;
    i<event.mc()->particles().size(); i++)
{
  const MCParticleFormat* prt =
    &event.mc()->particles()[i];

  if(prt->statuscode() != 1) continue;

  if(abs(prt->pdgid()) == 11)
  {
    if(prt->momentum().Pt()>50)
      electrons.push_back(prt);
  }
}
```

The first of the lines above indicates the declaration of a vector, dubbed `electrons`, of pointers to (constant) `MCParticleFormat` objects that will contain the selected electrons. With the next block of C++ commands, we loop over all the event particles (the `for` loop) and store the current particle into a temporary variable `prt`. We then discard non-final-state particles, which have a status code different from one (the first `if` statement). Finally, we fill the `electrons` vector with all electrons and positrons (with a Particle Data Group code equal to

`alphaQCD()`	Returns, as a floating-point number, the employed value for the strong coupling constant.
`alphaQED()`	Returns, as a floating-point number, the employed value for the electromagnetic coupling constant.
`particles()`	Returns, as a vector of pointers to `MCParticleFormat` objects the list of all final-, intermediate- and initial-state particles of the event.
`processId()`	Returns, as an integer number, the identifier of the physical process related to the considered event.
`scale()`	Returns, as a floating-point number, the employed value for the factorization scale.
`weight()`	Returns, as a floating-point number, the weight of the event.
`MET()`	Returns, as a pointer to a `MCParticleFormat` object, the missing transverse momentum $\not{p}_T$ of the event. The particles relevant for the calculation must be properly tagged as invisible (see Section 2.3.8).
`MHT()`	Returns, as a pointer to a `MCParticleFormat` object, the missing transverse hadronic momentum $\not{H}_T$ of the event. The particles relevant for the calculation must be properly tagged as invisible and hadronic (see Section 2.3.8).
`TET()`	Returns, as a floating-point number, the total visible transverse energy of the event E_T. The particles relevant for the calculation must not be tagged as invisible (see Section 2.3.8).
`THT()`	Returns, as a floating-point number, the total visible transverse hadronic energy of the event H_T. The particles relevant for the calculation must be properly tagged as hadronic, and not tagged as invisible (see Section 2.3.8).

Table 4. Methods of the `MCEventFormat` class.

`electrons()`	Returns, as a vector of pointers to `RecLeptonFormat` objects, all the reconstructed electrons of the event.
`jets()`	Returns, as a vector of pointers to `RecJetFormat` objects, all the reconstructed jets of the event.
`muons()`	Returns, as a vector of pointers to `RecLeptonFormat` objects, all the reconstructed muons of the event.
`photons()`	Returns, as a vector of pointers to `RecPhotonFormat` objects, all the reconstructed photons of the event.
`taus()`	Returns, as a vector of pointers to `RecTauFormat` objects, all the reconstructed hadronic taus of the event.
`tracks()`	Returns, as a vector of pointers to `RecTrackFormat` objects, all the reconstructed tracks of the event.
`genjets()`	Returns, as a vector of pointers to `RecJetFormat` objects, all the parton-level jets of the event.
`MCBquarks()`	Returns, as a vector of pointers to `MCParticleFormat` objects, all the parton-level b-quarks of the event.
`MCCquarks()`	Returns, as a vector of pointers to `MCParticleFormat` objects, all the parton-level c-quarks of the event.
`MCElectronicTaus()`	Returns, as a vector of pointers to `MCParticleFormat` objects, all the parton-level tau leptons that have decayed into an electron and a pair of neutrinos.
`MCHadronicTaus()`	Returns, as a vector of pointers to `MCParticleFormat` objects, all the parton-level tau leptons that have decayed hadronically.
`MCMuonicTaus()`	Returns, as a vector of pointers to `MCParticleFormat` objects, all the parton-level tau leptons that have decayed into a muon and a pair of neutrinos.
`MET()`	Returns, as a pointer to a `RecParticleFormat` object, the missing transverse momentum of the event as stored in the event file.
`MHT()`	Returns, as a pointer to a `RecParticleFormat` object, the missing transverse hadronic momentum $\not{H}_T$ of the event.
`TET()`	Returns, as a floating-point number, the total visible transverse energy of the event E_T.
`THT()`	Returns, as a floating-point number, the total visible transverse hadronic energy of the event H_T.

Table 5. Methods of the `RecEventFormat` class.

±11, as shown in the second `if` statement) whose transverse momentum is greater than 50 GeV (the third `if` statement).

We next present the methods that have been designed for the analysis of reconstructed events and which are part of the `RecEventFormat` class. This class contains functions (see Table 5) allowing access to two series of containers, the first ones gathering final state objects of a given nature and the second ones collecting specific generator-level (or equivalently parton-level) objects. All these containers can be further employed within an analysis so that the properties of the different objects can be retrieved and subsequently used, *e.g.*, for cuts and histograms. All the available methods associated with reconstructed objects have been collected in Table 6, while we recall that the `MCParticleFormat` class has been described in Table 3 (necessary for the handling of generator-level objects). In the case where some pieces of information (either specific properties of a given particle species or a given container itself) are absent from the event file, the related methods return null results.

Finally, as for the `MCEventFormat` class, specific functions (see Table 5) have been implemented to access the (missing) transverse energy and (missing) hadronic transverse energy of the event. While the $\not\!\!E_T$ variable is taken as stored in the event file and not calculated on the fly, the other variables are computed from the information on the reconstructed objects,

$$\begin{aligned}
E_T &= \sum_{\text{jets, charged leptons, photons}} \big|\boldsymbol{p}_T\big| \ , \\
H_T &= \sum_{\text{jets}} \big|\boldsymbol{p}_T\big| \ , \\
\not\!\!H_T &= \big|\not\!\!\boldsymbol{H}_T\big| = \Big| - \sum_{\text{jets}} \boldsymbol{p}_T \Big| \ ,
\end{aligned} \tag{2}$$

As an example, we show how an isolation requirement on final-state muons can be implemented. To do this we calculate an isolation variable $I_{\rm rel}$ defined as the amount of transverse energy, relative to the transverse momentum of the muon, present in a cone of radius $R = 0.4$ centered on the muon. We constrain this quantity to satisfy $I_{\rm rel} < 20\%$. A possible corresponding sample of C++ code would be

```
std::vector<const RecLeptonFormat *> MyMuons;
for(unsigned int i=0;
    i<event.rec()->muons().size(); i++)
{
  const RecLeptonFormat *Muon =
    &event.rec()->muons()[i];

  for(unsigned int j=0;
      j<Muon->isolCones().size(); j++)
  {
    const IsolationConeType *cone =
      &Muon->isolCones()[j];

    if(fabs(cone->deltaR()-0.4)<1e-3)
    {
      if(cone->sumPT()/Muon->momentum().Pt()<.20)
        MyMuons.push_back(Muon);
    }
  }
}
```

With those lines of code, we start by declaring a vector of pointers to `RecLeptonFormat` objects, named `MyMuons`, that will refer to the reconstructed muons that will be tagged as isolated. Then, we proceed with a `for`-loop dedicated to the computation of the $I_{\rm rel}$ variable for each of the final state muons. In the case where $I_{\rm rel}$ is smaller than 20%, the muon is added to the `MyMuons` container. In more details, this `for`-loop works as follows. The current muon is stored in a temporary variable called `Muon`. The calculation of $I_{\rm rel}$ relies, first, on the amount of calorimetric energy in a cone of radius $R = 0.4$ centered on the muon and second, on the transverse momentum of the current muon. The first of these two quantities is evaluated via the `isolCones()` method of the `RecLeptonFormat` class whereas the second one is derived from the muon four-momentum (obtained from the `momentum()` method of the `RecLeptonFormat` class). In the example above, we assume that muon isolation information associated with different cone sizes is available, including the choice $R = 0.4$. The second `for`-loop that has been implemented allows one to select desired value of R. The subsequent computation of the $I_{\rm rel}$ quantity is immediate. We refer to Ref. [21] for more detailed examples on this topic, in cases where event simulation is based on a modified version of Delphes 3 properly handling such a structure for the isolation information.

2.3.5 Applying cuts and filling histograms

The cuts that have been declared in the `Initialize` function of the analysis (see Section 2.3.2) are further employed within the `Execute` function by means of the `RegionSelectionManager` method `ApplyCut`. Its two arguments consist of a boolean quantity governing the cut condition (*i.e.*, it indicates whether the current event satisfies this cut) and a string which should be the name of one of the declared cuts.

This method starts by cycling through all regions that have been associated with this cut. For each region, it checks whether the region is still surviving all cuts applied so far by evaluating an internal boolean variable. In order to initialize this variable to `true` for all regions when starting to analyze a given event, the user must add, at the beginning of the `Execute` function,

```
Manager()->InitializeForNewEvent(myWeight);
```

where `MyWeight` is a floating-point number representing the weight of each event. The weight is used when histograms are filled and cut-flow charts calculated, and can be modified within the analysis by making use of the `SetCurrentEventWeight` method of the `RegionSelectionManager` class. If a given region is found to be already failing one of the preceding cuts (the internal surviving variable is set to the `false` value), the `ApplyCut` method continues to the next region associated with the considered cut. In contrast, if the region is surviving, the cut-flow for this region is updated according the cut condition (the boolean argument of the `ApplyCut` method) and the internal surviving variable will be kept as `true` or changed to `false` as appropriate.

Additionally, the analysis manager internally stores the total number of surviving regions, which is updated when a specific region fails a cut. This allows the `ApplyCut` method to determine and return, after cycling through the associated `RegionSelection` instances, a boolean quantity which is set

`btag()`	This method is specific to `RecJetFormat` objects and returns a boolean quantity describing whether the jet has been tagged as a *b*-jet.
`ctag()`	This method is specific to `RecJetFormat` objects and returns a boolean quantity describing whether the jet has been tagged as a *c*-jet.
`charge()`	Returns, as an integer number, the electric charge of the object (relative to the fundamental unit of electric charge e). This method is available for the `RecLeptonFormat`, `RecTauFormat` and `RecTrackFormat` classes.
`etaCalo()`	This method is specific to the `RecTrackFormat` class and returns, as a floating-point number, the pseudo-rapidity corresponding to the entry point of the track in the calorimeter.
`isolCones()`	Returns a vector of pointers to instances of the `IsolationConeType` class. This class allows one to retrieve information about the isolation of the object after defining a cone of a given size (a floating-point number accessed via the `deltaR()` method of the class) centered on it. The (integer) number of tracks in the cone is obtained by means of the `ntracks()` method, the sum of the transverse momentum of these tracks by means of the `sumPT()` method and the amount of calorimetric (transverse) energy in the cone by means of the `sumET()` method. The `isolCones()` method has only been implemented for the `RecTrackFormat`, `RecLeptonFormat`, `RecPhotonFormat` and `RecJetFormat` classes. A modified version of Delphes 3 that supports this structure has been introduced in Ref. [21].
`momentum()`	Returns, as a `TLorentzVector` object [19], the four-momentum of the particle. This method is available for all types of reconstructed objects. All the properties of the four-momentum can be accessed either from the methods associated with the `TLorentzVector` class, or as direct methods of the different classes of objects, after removing the capital letters from the method name. For instance, the method `pt()` is equivalent to `momentum().Pt()`. In addition, the methods `dphi_0_2pi(...)` and `dphi_0_pi(...)` return the difference in azimuthal angle normalized in the $[0, 2\pi]$ and $[0, \pi]$ ranges, respectively, between the object and any other object passed as an argument, whereas `dr(...)` returns their angular distance, the second object being provided as an argument as well.
`ntracks()`	Returns, as an integer number, the number of charged tracks associated with the reconstructed object. This method has been implemented for the `RecTauFormat` and `RecJetFormat` classes.
`pdgid()`	This method is specific to the `RecTrackFormat` class and returns, as an integer number, the Particle Data Group identifier defining the nature of the particle giving rise to the track. The numbering scheme is based on the Particle Data Group review [13].
`phiCalo()`	This method is specific to the `RecTrackFormat` class and returns, as a floating-point number, the azimuthal angle with respect to the beam direction corresponding to the entry point of the track in the calorimeter.
`sumET_isol()`	Returns, as a floating-point number, the amount of calorimetric (transverse) energy lying in a specific cone centered on the object. The cone size is fixed at the level of the detector simulation and this method is available for the `RecLeptonFormat` class (this information is available in the Lhco format).
`sumPT_isol()`	Returns, as a floating-point number, the sum of the transverse momentum of all tracks lying in a given cone centered on the object. The cone size is fixed at the level of the detector simulation and this method is available for the `RecLeptonFormat` class (this information is available in the Lhco format).
`EEoverHE()`	Returns, as a floating-point number, the ratio of the electromagnetic and hadronic calorimetric energy associated with the object. This method is available for the `RecLeptonFormat`, `RecTauFormat` and `RecJetFormat` classes.
`ET_PT_isol()`	Returns, as a floating-point number, the amount of calorimetric (transverse) energy lying in a given cone centered on the object calculated relatively to the sum of the transverse momentum of all tracks in this cone. The cone size is fixed at the level of the detector simulation and this method is available for the `RecLeptonFormat` class (this information is available in the Lhco format).
`HEoverEE()`	Returns, as a floating-point number, the ratio of the hadronic and electromagnetic calorimetric energy associated with the object. This method is available for the `RecLeptonFormat`, `RecTauFormat` and `RecJetFormat` classes.

Table 6. Methods allowing one to access the properties of the reconstructed objects.

`DisableColor()`	Switches off the display of messages in color. Colors are switched on by default, and the color scheme is hard-coded.
`EnableColor()`	Switches on the display of messages in color. Colors are switched on by default, and the color scheme is hard-coded.
`Mute()`	Switches off a specific message service. Services are switched on by default.
`UnMute()`	Switches on a specific message service. Services are switched on by default.

Table 7. Methods associated with a given message service. The available services are `INFO`, `WARNING`, `ERROR` and `DEBUG`.

to `false` in the case where not a single surviving region remains. The output of the `ApplyCut` method is thus equal to the boolean value of the statement *there is at least one region in the analysis, not necessarily one of those associated with this specific cut, which is still passing all cuts so far.* The moment at which it switches from `true` to `false` is the moment at which the present event should no longer be analyzed, and one should move on with the next event. It is therefore recommended, for efficiency purposes, to always call the `ApplyCut` method in the following schematic manner,

```
if ( !ApplyCut(...) )
  return;
```

with the `return` command allowing one to abort the analysis of the current event if all regions are failing the cuts applied so far.

Since, trivially, cuts keep some events and reject others, the distribution of an observable is affected by the placement of its histogram-filling command within the full sequence of cuts. Then since each region has its own unique set of cuts (by definition), the distribution of any observable is in general different for any two regions. However, it is meaningful to consider a single histogram as associated with multiple regions, *if* it is filled before any cuts are made that distinguish the regions. As an example, a typical format for processing an event would be a set of common pre-selection cuts, then the filling of various histograms (which are thus associated with all regions), then the application of the region-specific cuts (possibly followed by some further histogramming).

In MadAnalysis 5, we deal with this within the histogram-filling method of the `RegionSelectionManager` class, `FillHisto`, which takes as arguments a string and a floating-point number. The string should be the name of one of the declared histograms, and the floating-point number represents the value of the histogrammed observable for the event under consideration. This method can be called as in

```
Manager()->FillHisto("ptl1", val);
```

where `"ptl1"` is the name of the considered histogram (which corresponds to the example of Section 2.3.2) and `val` is the value of the observable of interest, namely the transverse momentum of the leading lepton in our case. The `FillHisto` method begins by verifying whether each of the regions associated with this histogram is surviving all cuts applied so far (via the internal surviving variable above-mentioned). In the case where all the associated regions are found surviving (failing) the cuts, the histogram is (not) filled. If a mixture of surviving and non-surviving regions is found, the program stops and displays an error message to the screen, as this situation implies that the histogram filling command has been called *after* at least one cut yields a distinction among the associated regions. This indicates an error in the design of the analysis.

2.3.6 Finalizing an analysis

Once all the events have been processed, the program calls the function `Finalize` that has to be provided with the analysis code. While the user is allowed to make use of this function for drawing histograms or deriving cut-flow charts as indicated in the manual for older versions of the program [2], the version of MadAnalysis 5 that has been introduced in this paper does not require the `Finalize` function to be implemented. Output files written according to the Saf format (see Section 2.5) are automatically generated.

2.3.7 Message services

The C++ core of MadAnalysis 5 includes a class of functions dedicated to the display of text to the screen at the time of the execution of the analysis. Whereas the standard C++ streamers allow two distinct levels of message – `std::cout` and `std:cerr` for normal and error messages – the SampleAnalyzer library allows the user to print messages that can be classified into four categories. In this way, information (the `INFO` function), warning (the `WARNING` function), error (the `ERROR` function) and debugging (the `DEBUG` function) messages can be displayed as in the following sample of code,

```
INFO    << "..." << endmsg;
WARNING << "..." << endmsg;
ERROR   << "..." << endmsg;
DEBUG   << "..." << endmsg;
```

Additionally, warning and error messages provide information on the line number of the analysis code that is at the source of the message. The effect of a given message service can finally be modified by means of the method presented in Table 7.

2.3.8 Physics services

The SampleAnalyzer core includes a series of built-in functions aiming to facilitate the writing of an analysis from the user viewpoint. More precisely, these functions are specific for particle identification or observable calculation and have been grouped into several subcategories of the C++ pointer `PHYSICS`. All the available methods are listed in Table 8, and we provide, in the rest of this section, a few more details, together with some illustrative examples.

`mcConfig().AddHadronicId(...)`	Adds a particle species, identified via its Particle Data Group code (an integer number given as argument), to the list of hadronizing particles. Mandatory for the computation of H_T and $\not{H}_T$ in the case of Monte Carlo events (see Section 2.3.4).
`mcConfig().AddInvisibleId(...)`	Adds a particle species, identified via its Particle Data Group code (an integer number given as argument), to the list of invisible particles. Mandatory for the computation of E_T and $\not{E}_T$ in the case of Monte Carlo events (see Section 2.3.4).
`mcConfig().Reset()`	Reinitializes the lists of invisible and hadronizing particles to empty lists.
`recConfig().Reset()`	Defines (reconstructed) leptons as isolated when no jet is present in a cone of radius $R = 0.5$ centered on the lepton.
`recConfig().UseDeltaRIsolation(...)`	Defines (reconstructed) leptons as isolated when no jet is present in a cone, with a radius given as a floating-point number in argument, centered on the lepton.
`recConfig().UseSumPTIsolation(...)`	Defines (reconstructed) leptons as isolated when both the sum Σ_1 of the transverse momentum of all tracks in a cone (of radius fixed at the level of the detector simulation) centered on the lepton is smaller than a specific threshold (the first argument) and the amount of calorimetric energy in this cone, relative to Σ_1, is smaller than another threshold (the second argument). This uses the information provided by the `sumPT_isol()` and `ET_PT_isol()` methods of the `RecLeptonFormat` class (see Table 6).
`Id->IsBHadron(...)`	Returns a boolean quantity indicating whether an `MCParticleFormat` object passed as argument is a hadron originating from the fragmentation of a b-quark.
`Id->IsCHadron(...)`	Returns a boolean quantity indicating whether an `MCParticleFormat` object passed as argument is a hadron originating from the fragmentation of a c-quark.
`Id->IsFinalState(...)`	Returns a boolean quantity indicating whether an `MCParticleFormat` object passed as argument is one of the final-state particles of the considered event.
`Id->IsHadronic(...)`	Returns a boolean quantity indicating whether an `MCParticleFormat` or a reconstructed object passed as argument yields any hadronic activity in the event.
`Id->IsInitialState(...)`	Returns a boolean quantity indicating whether an `MCParticleFormat` object passed as argument is one of the initial-state particles of the considered event.
`Id->IsInterState(...)`	Returns a boolean quantity indicating whether an `MCParticleFormat` object passed as argument is one of the intermediate-state particles of the considered event.
`Id->IsInvisible(...)`	Returns a boolean quantity indicating whether an `MCParticleFormat` or a reconstructed object passed as argument gives rise to missing energy.
`Id->IsIsolatedMuon(...)`	Returns a boolean quantity indicating whether a `RecLeptonFormat` object passed as a first argument is isolated within a given reconstructed event, passed as a second argument (under the format of a `RecEventFormat` object).
`Id->SetFinalState(...)`	Takes an `MCEventFormat` object as argument and defines the status code number associated with final-state particles.
`Id->SetInitialState(...)`	Takes an `MCEventFormat` object as argument and defines the status code number associated with initial-state particles.
`Transverse->AlphaT(...)`	Returns the value of the α_T variable [22], as a floating-point number, for a given (Monte Carlo or reconstructed) event passed as argument.
`Transverse->MT2(...)`	Returns, as a floating-point number, the value of the m_{T2} variable [23,24] computed from a system of two visible objects (the first two arguments, any particle class being accepted), the missing momentum (the third argument) and a test mass (a floating-point number given as the last argument).
`Transverse->MT2W(...)`	Returns, as a floating-point number, the value of the m_{T2}^W variable [25] computed from a system of jets (a vector of `RecJetFormat` objects in the first argument), a visible particle (given as the second argument, any particle class being accepted) and the missing momentum (the third argument). Only available for reconstructed events.

Table 8. Physics service methods.

As mentioned in Section 2.3.4, MadAnalysis 5 can compute the (missing) transverse energy and (missing) hadronic transverse energy associated with a given Monte Carlo event. This calculation however relies on a correct identification of the invisible and hadronizing particles. This information must be provided by means of the `mcConfig()` category of physics services, as for instance, in

```
PHYSICS->mcConfig().AddInvisibleId(1000039);
PHYSICS->mcConfig().AddHadronicId(5);
```

These intuitive lines of code indicate to the program that the gravitino (Particle Data Group identifier 1000039) yields missing energy and that the bottom quark (Particle Data Group identifier 5) will eventually hadronize.

An important class of methods shipped with the physics services consists of functions dedicated to the identification of particles and to the probing of their nature (invisible, hadronizing, *etc.*). They are collected within the `Id` structure attached to the `PHYSICS` object. For instance (see Table 8 for the other methods),

```
PHYSICS->Id->IsInvisible(prt)
```

allows one to test the (in)visible nature of the particle referred to by the pointer `prt`. Also, basic isolation tests on `RecLeptonFormat` objects can be performed when analyzing reconstructed events. Including in the analysis

```
PHYSICS->Id->IsIsolatedMuon(muon, event)
```

yields a boolean value related to the (non-)isolated nature of the reconstructed lepton `muon`, `event` being here a `RecEventFormat` object. Two isolation algorithms can be employed. By default, the program verifies that no reconstructed jet lies in a cone of radius $R = 0.5$ centered on the lepton. The value of R can be modified via the `recConfig()` category of physics services,

```
PHYSICS->recConfig().UseDeltaRIsolation(dR);
```

where `dR` is a floating-point variable with the chosen cone size. The user can instead require the program to tag leptons as isolated when both the sum of the transverse momentum of all tracks in a cone (of radius fixed at the level of the detector simulation) centered on the lepton is smaller than a specific threshold and when the amount of calorimetric energy in this cone, calculated relative to the sum of the transverse momentum of all tracks in the cone, is smaller than another threshold. This uses the information provided by the `sumPT_isol()` and `ET_ET_isol()` methods of the `RecLeptonFormat` class (see Table 6) and can be activated by implementing

```
PHYSICS->recConfig().UseSumPTIsolation(sumpt,et_pt);
```

where `sumpt` and `et_pt` are the two mentioned thresholds. For more sophisticated isolation tests, such as those based on the information encompassed in `IsolationConeType` objects possibly provided for reconstructed jets, leptons and photons (see Section 2.3.4), it is left to the user to manually implement the corresponding routines in his/her analysis.

In addition to identification routines, physics services include built-in functions allowing one to compute global event observables, such as several transverse variables that are accessible through the `Transverse` structure attached to the `PHYSICS` object. More information on the usage of these methods are provided in Table 8.

2.3.9 Sorting particles and objects

In most analyses, particles of a given species are identified according to an ordering in their transverse momentum or energy. In contrast, vector of particles as returned after the reading of an event are in general unordered and therefore need to be sorted. This can be achieved by means of sorting routines that can be called following the schematic form:

```
SORTER->(parts, crit)
```

In this line of code, `parts` is a vector of (Monte Carlo or reconstructed) objects and `crit` consists of the ordering criterion. The allowed choices for the latter are `ETAordering` (ordering in pseudorapidity), `ETordering` (ordering in transverse energy), `Eordering` (ordering in energy), `Pordering` (ordering in the norm of the three-momentum), `PTordering` (ordering in the transverse momentum), `PXordering` (ordering in the x-component of the three-momentum), `PYordering` (ordering in the y-component of the three-momentum) and `PZordering` (ordering in the z-component of the three-momentum).

2.4 Compiling and executing the analysis

In Section 2.1, we have pointed out that the `Build` subdirectory of the analysis template contains a `Makefile` script readily to be used. In this way, the only task left to the user after having implemented his/her analysis is to launch this script in a shell, directly from the `Build` directory. This leads first to the creation of a library that is stored in the `Build/Lib` subdirectory, which includes all the analyses implemented by the user and the set of classes and methods of the SampleAnalyzer kernel. Next, this library is linked to the main program and an executable named `MadAnalysis5Job` is generated (and stored in the `Build` directory).

The program can be run by issuing in a shell the command

```
./MadAnalysis5Job <inputfile>
```

where `<inputfile>` is a text file with a list of paths to all event files to analyze. All implemented analyses are sequentially executed and the results, generated according to the Saf format (see Section 2.5), are stored in the `Output` directory.

2.5 The structure of the output of an analysis

As indicated in the previous section, the program stores, after its execution, the results of the analysis or analyses that have been implemented by the user in the `Output` subdirectory of the working directory. First, a subdirectory with the same name as the input file (`<inputfile>` in the schematic example of Section 2.4) is created. If a directory thus named exists already, the code uses it without deleting its content. It contains a Saf file (updated if already existing) with global information on the analyzed event samples organized following an Xml-lie syntax:

```
<SampleGlobalInfo>
 # xsection  xsec_error  nevents  sum_wgt+  sum_wgt-
 0.00e+00    0.00e+00    0        0.00e+00  0.00e+00
</SampleGlobalInfo>
```

where the numbers have been taken equal to zero for the sake of the illustration. Those pieces of information are extracted from the read event files and kept equal to zero if not available. In addition, the format allows for header and footer tags (`SAFheader` and `SAFfooter`) omitted for brevity.

Secondly, a subdirectory specific to each of the executed analyses is created within the `<inputfile>` directory. The name of the subdirectory is the name of the associated analysis followed by an integer number chosen in such a way that the directory name is unique. This directory contains a Saf file with general information on the analysis (`name.saf`, `name` denoting a generic analysis name), a directory with histograms (`Histograms`) and a directory with cut-flow charts (`Cutflows`).

In addition to a header and a footer, the `name.saf` file, still encoded according to an Xml-like structure, contains a list with the names of the regions that have been declared in the analysis implementation. They are embedded in a `RegionSelection` Xml structure, as in

```
<RegionSelection>
 "MET>200"
 "MET>300"
</RegionSelection>
```

when taking the example of Section 2.3.2.

The `Histograms` subdirectory contains a unique Saf file with, again in addition to a possible header and footer, all the histograms implemented by the user. The single histogram declared in Section 2.3.2 would be encoded in the Saf format as in the following self-explanatory lines of code:

```
 <Histo>
  <Description>
   "ptl1"
   # nbins          xmin            xmax
   20               50              500
   # associated RegionSelections
   MET>300    # Region nr. 1
  </Description>
  <Statistics>
   0 0 # nevents
   0 0 # sum of event-weights over events
   0 0 # nentries
   0 0 # sum of event-weights over entries
   0 0 # sum weights^2
   0 0 # sum value*weight
   0 0 # sum value^2*weight
 </Statistics>
 <Data>
  0 0 # underflow
  0 0 # bin 1 / 20
  ...
  0 0 # bin 20 / 20
  0 0 # overflow
  </Data>
</Histo>
```

where the dots stand for the other bins that we have omitted for brevity. For the sake of the example, all values have been set to zero.

Finally, the `Cutflows` directory contains one Saf file for each of the declared regions, the filename being the name of the region followed by the `saf` extension. Each of these files contains the cut-flow chart associated with the considered region encoded by means of two types of Xml tags. The first one is only used for the initial number of events (`InitialCounter`) whereas the second one is dedicated to each of the applied cuts. Taking the example of the first of the two cuts declared in Section 2.3.2, the `MET_gr_200.saf` file (the `>` symbol in the region name has been replaced by `_gr_`) would read

```
<InitialCounter>
 "Initial number of events"    #
 0           0                 # nentries
 0.00e+00    0.00e+00          # sum of weights
 0.00e+00    0.00e+00          # sum of weights^2
</InitialCounter>
<Counter>
 "1lepton"                     # 1st cut
 0           0                 # nentries
 0.00e+00    0.00e+00          # sum of weights
 0.00e+00    0.00e+00          # sum of weights^2
</Counter>
```

which is again self-explanatory.

3 Illustrative examples

In this section, we show two examples of analyses making use of the new features of MadAnalysis 5 introduced in the previous section. First, we focus in Section 3.1 on the reinterpretation of a CMS search for stops in 8 TeV events with one single lepton, jets and missing energy [8]. Second, we investigate in Section 3.2 the implementation of a recent phenomenological analysis dedicated to the probing of monotop systems decaying in the hadronic mode [9].

3.1 Recasting a CMS search for supersymmetric partners of the top quark

We have performed an implementation of the CMS cut-based strategy for probing stops in the single lepton and missing energy channel as presented in Ref. [8]. The analysis contains 16 overlapping signal regions that share a set of common preselection cuts. In addition to simple requirements on the p_T and η of candidate objects, it includes, in particular, the following preselection criteria.

– Signal leptons are required to be isolated. This is defined as $\sum p_T^\ell < \min(5\text{ GeV}, 0.15\, p_T^\ell)$, where $\sum p_T^\ell$ is the sum of the transverse momentum of all particle flow (PF) particles in a cone of radius $R = 0.3$ centered on the lepton (but excluding the lepton itself whose transverse momentum is p_T^ℓ).
– The overlap between jets and leptons is resolved by discarding jets within $R = 0.4$ of a lepton. Four jets with moderate η ("central") are subsequently required, with at least one of these being tagged as originating from the fragmentation of a b-quark (commonly referred to as a b-tagged jet).
– One requires at least 100 GeV of missing transverse energy.
– Only one isolated lepton should be present. This is achieved by rejecting a second isolated lepton with $p_T > 5$ GeV, with an isolation requirement much looser than the previous one ($\sum p_T^\ell < 0.20 p_T^\ell$, using charged PF particles in a cone of radius $R = 0.3$).

– There should not be any isolated track of transverse momentum $p_T^{\text{track}} > 10$ GeV and whose electric charge is opposite to the one of the primary lepton. Isolation is enforced by constraining the sum of the transverse momentum of all charged PF particles in a cone of radius $R = 0.3$ centered on the track to be smaller than $0.10\,p_T^{\text{track}}$.
– Events with reconstructed hadronic taus are vetoed.
– In addition, just after preselection, the transverse mass obtained from the lepton and missing transverse energy is required to be larger than 120 GeV in order to reduce the SM background from $t\bar{t}$ and $W+$ jets events.

Next, the sixteen signal regions are distinguished by *i)* the separation between the missing transverse momentum and the two hardest jets ($\Delta\phi(\not{E}_T, j_1 \text{ or } j_2) > 0.8$), *ii)* the compatibility of the final state with the presence of a hadronically decaying top quark (by means of a χ^2 fit on the reconstructed top quark), *iii)* more stringent selections on the amount of missing transverse energy of the event, and finally *iv)* a requirement on M_{T2}^W. We refer to Refs. [7,8] for more details on these cuts and regions and recall that their implementation strictly obeys the syntax introduced in Section 2. For the sake of the example, we include below a snippet of code describing the implementation of the veto on the presence of isolated tracks. Since the reconstruction of events using the particle flow algorithm [?] is too complex to be reproduced, we only consider tracks in the inner detector. We start by creating a variable (`Tracks`) containing all the tracks whose transverse momentum is larger than 10 GeV and pseudorapidity satisfies $|\eta^{\text{track}}| < 2.1$,

```
for(unsigned int i=0;
    i<event.rec()->tracks().size(); i++)
{
  const RecTrackFormat *myTrack =
    &(event.rec()->tracks()[i]);
  double pt = myTrack->momentum().Pt();
  double abseta = fabs(myTrack->momentum().Eta());
  if(pt>10. && abseta<2.1) Tracks.push_back(myTrack);
}
```

Next, we iterate over this container and update a boolean variable `noIsolatedTrack` to `false` in the case where an isolated track with a charge opposite to the one of the primary lepton (contained in the `LeptonCharge` variable) is found,

```
noIsolatedTrack = true;
for(unsigned int i=0; i<Tracks.size(); i++)
{
  if(Tracks[i]->charge()!=LeptonCharge)
  {
    for(unsigned int j=0;
        j<Tracks[i]->isolCones().size(); j++)
    {
     const IsolationConeType *cone =
        &Tracks[i]->isolCones()[j];
     double pt = Tracks[i]->momentum().Pt();
     if(fabs(cone->deltaR()-0.3)<0.001)
       if(cone->sumPT()<.1*pt)
         { noIsolatedTrack = false; break; }
    }
  }
}
```

It is then sufficient to implement the verification of the cut condition as explained in Section 2.3.5,

```
if(!Manager()->ApplyCut(Check,"veto isol track"))
  return;
```

where we are assuming that a cut "`veto isol track`" has been declared in the method `Initialize`.

The validation process was based on (partonic) event samples provided by the CMS collaboration in LHE format, for the "T2tt" and "T2bW" simplified models (in which the stop always decays as $\tilde{t}_1 \to t\tilde{\chi}_1^0$ and $\tilde{t}_1 \to b\tilde{\chi}_1^+$, respectively). The LHE files were passed through Pythia 6 [26] for parton showering and hadronization, then processed by Delphes 3 for the simulation of the detector effects. The number of events after cuts and histograms produced by MadAnalysis 5 were then normalized to the correct luminosity after including cross sections at the next-to-leading order and next-to-leading logarithmic (NLO+NLL) accuracy, as tabulated by the LHC SUSY Cross Section Working Group [?]. Finally, the histograms and cut-flow charts we derive are compared to the publicly available ones, as is explained in more detail in Ref. [7] (see also [21]).

As an example, we present in Table 9 the cut-flow chart for the benchmark point $\tilde{t} \to t\tilde{\chi}_1^0$ (650/50) in one of the signal signal regions, denoted as $\tilde{t} \to t\tilde{\chi}_1^0$, high ΔM, $\text{E}_\text{T}^\text{miss} > 300$ GeV. The benchmark point corresponds to stop pair-production which subsequently decay with 100% branching fraction into $t\tilde{\chi}_1^0$, with $(m_{\tilde{t}}, m_{\tilde{\chi}_1^0}) = (650, 50)$ GeV, while the signal region is targeting $\tilde{t} \to t\tilde{\chi}_1^0$, when the mass difference $\Delta M = m_{\tilde{t}} - m_{\tilde{\chi}_1^0}$ and the missing transverse energy are large, as it can be the case for this benchmark point. The number of events in 9 has been obtained from the output Saf file and the CMS numbers from [?] (the small associated statistical uncertainties are not shown). As can be seen, the agreement is good since our results do not differ from the CMS ones by more than 20%, which is expected from fast simulation. More details can be found in Ref. [7].

$\tilde{t} \to t\tilde{\chi}_1^0$ (650/50) cut-flow chart for SR $\tilde{t} \to t\tilde{\chi}_1^0$, high ΔM, $\text{E}_\text{T}^\text{miss} > 300$ GeV		
cut	number of events	
	MadAnalysis 5	CMS
≥ 1 candidate lepton	62.1	
≥ 4 central jets	33.9	
$E_T^{\text{miss}} > 100$ GeV	31.4	29.7
≥ 1 b-tagged jet	27.1	25.2
veto isol lepton and track	22.5	21.0
veto hadronic tau	22.0	20.6
$\Delta\phi(E_T^{\text{miss}}, j_1 \text{ or } \text{j}_2) > 0.8$	18.9	17.8
hadronic $\chi^2 < 5$	12.7	11.9
$M_T > 120$ GeV	10.4	9.6
$E_T^{\text{miss}} > 300$ GeV	6.8	
$M_{T2}^W > 200$ GeV	5.1	4.2

Table 9. Cut-flow chart for the benchmark point $\tilde{t} \to t\tilde{\chi}_1^0$ (650/50) in the signal region $\tilde{t} \to t\tilde{\chi}_1^0$, high ΔM, $\text{E}_\text{T}^\text{miss} > 300$ GeV.

The CMS analysis of Ref. [8] contains, in addition to a cut-based analysis strategy, a second strategy relying on advanced multivariate techniques. One of the key variable of this anal-

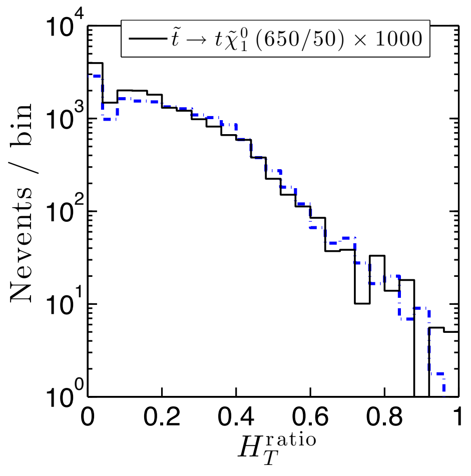

Fig. 2. Distribution of the kinematic variable $H_T^{\rm ratio}$ after the preselection for the benchmark point $\tilde{t} \to t\tilde{\chi}_1^0$ (650/50). The solid black line has been obtained with MadAnalysis 5, while the dashed blue line corresponds to the CMS results.

ysis consists of a quantity denoted by $H_T^{\rm ratio}$ and defined as the fraction of the total scalar sum of the jet transverse energies, making use of jets with a transverse momentum larger than 30 GeV and a pseudorapidity $|\eta| < 2.4$, that lies in the same hemisphere as the missing momentum. For illustrative purposes, we present below a way to fill an histogram representing this variable. The definition of H_T being different from Eq. (2), our sample of code includes its calculation as required by the analysis and the result is stored in the variable `HT`. In addition, the amount of hadronic energy located on the same hemisphere as the missing momentum is stored in the variable `HTsameHemisphere`. All the calculation relies on a collection with the relevant jets stored in the `Jets` container,

```
double HT          = 0.;
double HTsameHemisphere = 0.;
for(unsigned int i=0; i<Jets.size(); i++)
{
  double Et = Jets[i]->momentum().Et();
  HT += Et;
  if(Jets[i]->dphi_0_pi(pTmiss) < 3.1415)
    HTsameHemisphere += Et;
}
double HTratio = HTsameHemisphere / HT;
```

An associated histogram, whose name is declared as "`HTratio`" in the method `Initialize` can then be filled using the syntax introduced in Section 2.3.5,

```
Manager()->FillHisto("HTratio", HTratio);
```

The figure that is extracted from the corresponding output Saf file is shown in Figure 2, with the CMS results from Figure 2 of Ref. [8] superimposed. Both the shape and the normalization of the $H_T^{\rm ratio}$ are in good agreement with the CMS results.

3.2 Designing a phenomenological analysis for probing hadronic monotop states

The LHC sensitivity to the observation of a monotop state – a topology where a single, hadronically decaying, top quark is produced in association with missing energy – has been recently investigated by means of a phenomenological analysis relying on a cut-and-count technique [9]. It exploits the presence of three final-state jets (including a b-tagged jet) compatible with a top quark decay and that lie in a different hemisphere than a large amount of missing momentum. More into details, events are selected as follows.

- Selected events contain two or three light jets (allowing one for initial or final state radiation) with a transverse momentum greater than 30 GeV, as well as one b-tagged jet with a transverse momentum larger than 50 GeV. Jet pseudorapidity is demanded to satisfy $|\eta^j| < 2.5$ and the ratio of the hadronic and electromagnetic calorimetric energy of each jet is constrained to be above 30%.
- Events featuring isolated charged leptons with a transverse momentum $p_T^\ell > 10$ GeV and a pseudorapidity $|\eta^\ell| < 2.5$ are vetoed. Lepton isolation is enforced by imposing that the sum of the transverse momentum of all tracks in a cone of $R = 0.4$ centered on the lepton is smaller than $0.2 p_T^\ell$.
- At least 250 GeV of missing transverse energy is required.
- We select the pair of light jets whose invariant mass is the closest to the W-boson mass. This quantity is then constrained to lie in the $[50, 105]$ GeV range.
- The missing momentum is constrained to be well separated from the momentum of the reconstructed top quark ($\Delta\phi \in [1, 5]$, the azimuthal angular distance being normalized in the $[0, 2\pi]$ range).
- The missing momentum is constrained to be well separated from the momentum of the hardest jet ($\Delta\phi \in [0.5, 5.75]$, the azimuthal angular distance being normalized in the $[0, 2\pi]$ range).
- The reconstructed top mass is constrained to lie in the $[140, 195]$ GeV range.

As in Section 3.1, all these cuts can be easily implemented following the syntax of Section 2, so that we again restrict ourselves to the presentation of a few illustrative samples of code.

First, we show how to select the jet candidates relevant for the analysis and store them in a container named `TheJets`,

```
for (unsigned int i=0;
     i<event.rec()->jets.size();i++)
{
  const RecJetFormat *myj = &(event.rec()->jets[i]);
  double abseta = fabs(myj->Eta());
  double pt = myj->Pt();
  double HEEE  = myj->HEoverEE();
  if(abseta<2.5 && pt > 30 && HEEE>0.3)
    TheJets.push_back(myj);
}
```

In those (self-explanatory) lines of code, we have not yet split the jets into b-tagged and non-b-tagged ones.

Second, we focus on the reconstruction of the W-boson and the top quark. Assuming that all selected light jets are stored in a vector named `ljets` and the b-tagged jet is represented by the variable `bjet`, one possible implementation of a code deriving the four-momenta of the reconstructed W-boson and top quark would be

```
TLorentzVector w, top;
double mjj=9999;
for(unsigned int i=0; i<ljets.size(); i++)
```

Cut	**SII.v-400**	**SII.v-600**
Jet selection and lepton veto	3176 ± 51.2	774 ± 25.2
Missing energy requirement	949 ± 30.0	305 ± 16.8
W-boson reconstructed mass	515 ± 22.4	163 ± 12.5
Separation of the reconstructed top quark from the missing momentum	501 ± 22.1	158 ± 12.3
Separation of the hardest jet from the missing momentum	497 ± 22.0	156 ± 12.3
Reconstructed top mass	311 ± 17.5	96 ± 9.7

Table 10. Cut-flow charts for the two considered monotop scenarios **SII.v-400** and **SII.v-600** (the numbers indicating the choice for the invisible state mass in GeV). We present the predicted number of events after each of the cuts detailed in the text, for an integrated luminosity of 20 fb^{-1} of LHC collisions at a center-of-mass energy of 8 TeV.

```
for(unsigned int j=i+1; j<ljets.size(); j++)
{
  TLorentzVector w_tmp =
    ljets[i]->momentum()+ljets[j]->momentum();

  if( fabs(w_tmp.M()-80.) < fabs(mjj-80.) )
  {
    w=w_tmp;
    mjj=w.M();
  }
}

top = w + bjet->momentum();
```

In the lines above, the double `for`-loop derives the pair of light jets that form the system which is the most compatible with a W-boson, the associated reconstructed mass being stored in the `mjj` variable. The W-boson reconstructed four-momentun is saved in an instance of the `TLorentzVector` class (named `w`) and the top quark four-momentum is then derived by adding the four-momentum of the b-tagged jet (stored in the `top` variable). The reconstructed top mass could further be histogrammed via the standard command,

```
Manager()->FillHisto("Mtreco", top.M());
```

where an histogram named `"Mtreco"` has been initialized appropriately. Moreover, the selection cut on this variable could be implemented via

```
bool cutcondition = (top.M()>140) && (top.M()<195);
if(!Manager()->ApplyCut(cutcondition,"Mtop"))
   return;
```

assuming that the cut named `"Mtop"` has been correctly initialized.

We illustrate the above analysis in the context of the **SII.v** monotop scenario of Ref. [9]. In this setup, the monotop system arises from the flavor-changing interaction of an up quark with a novel invisible vector boson whose mass have been fixed to either 400 GeV or 600 GeV. Using the publicly available FeynRules [27, 28] monotop model [29], we generate a UFO library [30] that we link to the MadGraph 5 event generator [31]. We then simulate parton-level events that include the decay of the top quark, perform parton showering with the Pythia 6 package [26] and simulate the detector response with

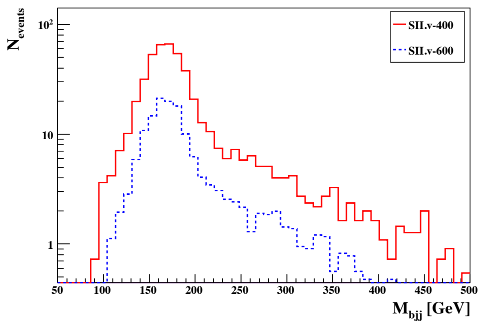

Fig. 3. Distribution in the reconstructed top mass M_{bjj} for the two new physics scenarios investigated in Section 3.2, **SII.v-400** and **SII.v-600** (the numbers indicating the choice for the invisible state mass in GeV). All selection cuts, but the one on M_{bjj}, have been applied. The curves are normalized to 20 fb^{-1} of simulated LHC collisions at a center-of-mass energy of 8 TeV.

the above-mentionned modified version of the Delphes 3 program [20, 21], using the CMS detector description of Ref. [32]. From our analysis implementation, we derive, in the context of the two considered new physics models, the distribution in the reconstructed top mass M_{bjj} on Figure 3 and the cut-flow charts of Table 10.

4 Conclusion

Acknowledgements

CW thanks Jon So and Paul Wymant for helpful discussions, the LPSC de Grenoble for hospitality, and funding from the French ANR – Project DMAstroLHC, ANR-12-BS05-0006.

References

1. K. Cranmer, I. Yavin, JHEP **1104**, 038 (2011). DOI 10.1007/JHEP04(2011)038

2. E. Conte, B. Fuks, G. Serret, Comput.Phys.Commun. **184**, 222 (2013). DOI 10.1016/j.cpc.2012.09.009
3. M. Drees, H. Dreiner, D. Schmeier, J. Tattersall, J.S. Kim, arXiv:1312.2591 [hep-ph].
4. S. Kraml, S. Kulkarni, U. Laa, A. Lessa, W. Magerl, et al., arXiv:1312.4175 [hep-ph].
5. M. Papucci, K. Sakurai, A. Weiler, L. Zeune, arXiv:1402.0492 [hep-ph].
6. E. Conte, B. Fuks, arXiv:1309.7831 [hep-ph].
7. B. Dumont, B. Fuks, S. Kraml, et al., in preperation.
8. S. Chatrchyan, et al., Eur.Phys.J. **C73**, 2677 (2013). DOI 10.1140/epjc/s10052-013-2677-2
9. J.L. Agram, J. Andrea, M. Buttignol, E. Conte, B. Fuks, Phys.Rev. **D89**, 014028 (2014). DOI 10.1103/PhysRevD.89.014028
10. URL http://www.datacite.org
11. H. Plothow-Besch, Comput.Phys.Commun. **75**, 396 (1993). DOI 10.1016/0010-4655(93)90051-D
12. W. Giele, et al., hep-ph/0204316.
13. J. Beringer, et al., Phys.Rev. **D86**, 010001 (2012). DOI 10.1103/PhysRevD.86.010001
14. E. Boos, M. Dobbs, W. Giele, I. Hinchliffe, J. Huston, et al., hep-ph/0109068.
15. J. Alwall, A. Ballestrero, P. Bartalini, S. Belov, E. Boos, et al., Comput.Phys.Commun. **176**, 300 (2007). DOI 10.1016/j.cpc.2006.11.010
16. URL http://cepa.fnal.gov/psm/stdhep/c++/
17. M. Dobbs, J.B. Hansen, Comput.Phys.Commun. **134**, 41 (2001). DOI 10.1016/S0010-4655(00)00189-2
18. URL http://www.jthaler.net/olympicswiki/
19. R. Brun, F. Rademakers, Nucl.Instrum.Meth. **A389**, 81 (1997). DOI 10.1016/S0168-9002(97)00048-X
20. J. de Favereau, et al., JHEP **1402**, 057 (2014). DOI 10.1007/JHEP02(2014)057
21. G. Brooijmans, R. Contino, B. Fuks, F. Moortgat, P. Richardson, et al., in preperation.
22. L. Randall, D. Tucker-Smith, Phys.Rev.Lett. **101**, 221803 (2008). DOI 10.1103/PhysRevLett.101.221803
23. C. Lester, D. Summers, Phys.Lett. **B463**, 99 (1999). DOI 10.1016/S0370-2693(99)00945-4
24. H.C. Cheng, Z. Han, JHEP **0812**, 063 (2008). DOI 10.1088/1126-6708/2008/12/063
25. Y. Bai, H.C. Cheng, J. Gallicchio, J. Gu, JHEP **1207**, 110 (2012). DOI 10.1007/JHEP07(2012)110
26. T. Sjostrand, S. Mrenna, P.Z. Skands, JHEP **0605**, 026 (2006). DOI 10.1088/1126-6708/2006/05/026
27. N.D. Christensen, C. Duhr, Comput.Phys.Commun. **180**, 1614 (2009). DOI 10.1016/j.cpc.2009.02.018
28. A. Alloul, N.D. Christensen, C. Degrande, C. Duhr, B. Fuks, (2013)
29. J. Andrea, B. Fuks, F. Maltoni, Phys.Rev. **D84**, 074025 (2011). DOI 10.1103/PhysRevD.84.074025
30. C. Degrande, C. Duhr, B. Fuks, D. Grellscheid, O. Mattelaer, et al., Comput.Phys.Commun. **183**, 1201 (2012). DOI 10.1016/j.cpc.2012.01.022
31. J. Alwall, M. Herquet, F. Maltoni, O. Mattelaer, T. Stelzer, JHEP **1106**, 128 (2011). DOI 10.1007/JHEP06(2011)128
32. J.L. Agram, J. Andrea, E. Conte, B. Fuks, D. GelÃĺ, et al., Phys. Lett. **B725**, 123 (2013). DOI 10.1016/j.physletb.2013.06.052