Analysis description for LHC result reinterpretations

Philippe Gras^1*, Prosper Harrison*^{2}*, Sezen Sekmen*^{3}
^{1} IRFU, CEA, Université Paris-Saclay, Gif-sur-Yvette, France
^{2} Florida State University, Tallahassee, USA
^{3} Kyungpook National University

Abstract
LHADA is a language to describe LHC analysis which has a wide range of applications. In this work, this language is investigated for its usage in the context of LHC result reinterpretation. It would be employed to describe in an unambiguous and concise manner a data analysis including all the details needed for a reinterpretation of the result in the context of a physics theory not considered in the original analysis. A specialisation of the language dedicated to reinterpretation is introduced. The specialisation defines extra syntax rules and constitutes a subset of the language. Three different analyses used as benchmarks are described with this language. Automatic generation of code reproducing the analysis on Monte-Carlo samples for the purpose of result reinterpretation is investigated. We demonstrate that programs that generates code to be used in a result reinterpretation tool can be easily developed and a prototype is presented. In addition, the generated code can be used to validate the accuracy of the analysis description.

1. Introduction

The need for a standard to describe analyses of LHC data in an unambiguous way together with the definition of its requirements has been studied at the 2015 session of Les Houches PhysTeV workshop [1]. The study includes a proposal for this standard. Other choices are possible, for instance CUTLAND [2] is another description language that fulfils similar requirements. In this work, we are investigating the standard proposed in Les Houches proceedings in the context of analysis reinterpretation. For convenience we will call this standard LHADA, although it's one implementation of Les Houches analysis description accord (Lhada), which refers to the language requirement only. Three questions are addressed: the coverage of the language, that is its ability to implement a large spectrum of analyses, the completeness of the analysis description, and the capacity to validate this description. The first question is addressed by implementing the description of example analyses with different levels of complexity. The two others were addressed by developing two machine interpreters. The interpreters generate c++ code that reproduces the analysis on an input samples. One interpreter produces a module, so-called "Rivet analysis", for the Rivet [3] framework, while the second produces a standalone code based on the ROOT [4] framework. The Rivet based code is meant to be used for result reinterpretation. It can also be used to validate an analysis description by reproducing reference numbers provided by the analysis authors. In particular the cut flow, that is the acceptances of the subsequent selections (the "cuts") of the analysis, is well suited for such validation [5]. The completeness of the description is validated at the same time.

2. Describing analyses LHADA

In order to test the suitability of the LHADA language to describe a LHC analysis three different analyses have been considered. The first analysis is the *Search for new physics in the all-hadronic final state with the MT2 variable* from Ref. [6]. The two other analyses are the *Search for squarks and gluinos in final states with jets and missing transverse momentum at* $\sqrt{s} = 13\,TeV$ *with the ATLAS detector* [7] and *Search for dark matter at* $\sqrt{s} = 13\,TeV$ *in final states containing an energetic photon and large missing transverse momentum with the ATLAS detector* [8], which are used for the comparison of the reinterpretation tools performed in Contribution `\ref{sec:recast}`.

We have described the three analyses with the LHADA language and the descriptions can be found in the analysis descriptions database [9] respectively under `lhada/analyses/CMS-PAS-SUS-16-015`, `lhada/analyses/ATLASSUSY1605.03814`, and `lhada/analyses/ATLASEXOT1704.0384`. No particular difficulty has been encountered. The LHADA17 language subset has been used and a cut flow table has been included [editor's note: to be added] in the description to allow the validation of the descriptions using code generated with the interpreter.

3. Generating a Rivet analysis from Lhada

LHADA is a multipurpose and flexible language. In the case of LHADA2RIVET, the interpreter that generates a RIVET analysis, we have chosen to limit ourselves to the analysis reinterpretation use case and to specify accurately the analysis description language understood by the program. For this purpose, we have derived a subset of the LHADA language, called LHADA17. In this section, we will first draw the specifications of this sublanguage. We will then present the automatic generation of Rivet analyses.

3.1 Describing the description language

The Extended Backus-Naur Form [10, 11] (EBNF) notation has been used to specify the syntax and grammar of LHADA17. The syntax is given in Appendix A. The following rules which have not been included in the EBNF syntax to simplify the notation apply:

- A hash sign (#) can be used to include comments in the LHADA files: all characters of a line starting from a hash sign are comments and ignored for the interpretation based on the EBNF description.
- If the last non-space character of a line is a backslash (\), then the line is merged with the following line before being interpreted according to the EBNF description.
- An entity (`function`, `object` or `cut`) should be declared before being used. For instance if a function is used in a "cut" definition, the corresponding `function` block should appear before the `cut` line. This rule is meant to simplify the parsing and also to avoid circular definitions.

A specificity of LHADA is the usage of programming languages to describe algorithms, via the LHADA `functions` while the main structure of the analysis is described with the dedicated language. A reference implementation of the algorithm is provided in a "commonly used" programming language.

The implementation is given in a code source file, which can group implementations of different LHADA `functions` and can be shipped along with the LHADA description file or provided as an http link. In order to ease machine interpretation LHADA17 includes the following restrictions for the reference `function` implementations.

- the implementation is written in c++11 [12];
- the implementation must depend only on the code provided in the file and libraries from the restricted set defined below; the file should be compilable with c++11 compliant compilers.
- the allowed types for the function parameters are: `int`, `float`, `double`, `std::string`, `LHADAParticle`, and `FourMomentum`. Parameters are passed by copy (no modifier) or by constant reference (with `const &` modifier, like `const LHADAParticle&`); this rules exclude the use of a templated function; function templates are allowed for auxiliary functions;
- `#include` statements can be used to include header files from the allowed libraries;
- the file, where the functions is defined should be compilable with c++11 compliant compilers;
- self-contained function are encouraged but not mandatory; by self contained we mean that the file is compilable with c++11 compliant compilers after having removed all code except the function and the `#include` that precedes it;
- the function can use the random object to draw pseudo-random numbers, whose scope is global to all functions and which provides the methods described in Table 3;

The `LHADAParticle` and `Momentum` types are two classes which provides the methods respectively described in Tables 1 and 2. The restricted set of libraries includes the libraries that comes with the c++ standard (`std` libraries) and a common library provided in the LHADA repository. The header file coming with the latest library will be callded `lhadatools-vXX.h` where `XX` is a string specifying the library version. The set of libraries can evolve without requiring a revision of the LHADA17 language standard and will be defined by a list stored in the LHADA repository.

The LHADA language does not explicitly specify how the arguments listed in a `function` block are matched to the arguments of the reference implementation of the function. To prevent confusion, LHADA17 requires that the arguments appear in the `function` block in the order of the c++ function argument list of its reference implementation and with the same name. If the names differ, the arguments should be matched according to their order, though in such case the file can simply be considered as invalid.

An `object` block defines an entity, typically a collection of particles, starting from the input defined by the `take` statement that is transformed by a sequence of `apply`, `select`, and `reject` statements. The `apply` statement specifies a function that transforms the entity. In LHADA17, the function must take as first argument the entity to transform. This argument is specified in the function definition, but not on the `apply` statement, where it is implicit. Collections are filtered with `select` and `reject` statements. A condition

Table 1: Definition of the LHADAParticle type.

Name of the property in the LHADA file	c++ method	Description
mass	mass()	Mass
e	e()	Energy
px	px()	momentum x-component
py	py()	momentum y-component
pz	pz()	momentum z-component
pt	pt()	absolute transverse momentum
eta	eta()	pseudorapidity
rapidity	rapidity()	rapidity
charge	charge()	charge
spin	spin()	spin
pdgid	pdgid()	PDG particle id
phi	phi()	momentum phi coordinate
x	x()	particle production vertex x-coordinate
y	y()	particle production vertex y-coordinate
z	z()	particle production vertex z-coordinate

Table 2: Definition of the FourMomentum type.

Name of the property in the LHADA file	c++ method	Description
mass	mass()	Mass
e	e()	Energy
px	px()	momentum x-component
py	py()	momentum y-component
pz	pz()	momentum z-component
pt	pt()	absolute transverse momentum
eta	eta()	pseudorapidity
rapidity	rapidity()	rapidity

Table 3: Definition of the random object interface: list of provided methods.

c++ method	Description
uniform(double x)	Returns a pseudorandom number following a uniform distribution over the $[0, \mathrm{x}]$ interval.
gauss(double mean, double sigma)	Returns a pseudorandom number following a Gaussian distribution.
poisson(int mean)	Returns a pseudorandom number following a Poisson distribution.
breitWigner(double mean, double gamma)	Returns a pseudorandom number following a Breit-Wigner distribution.
exp(double tau)	Return a pseudorandom number following the $\exp(-t/\mathrm{tau})$ distribution.
landau(double mean, double sigma)	Returns a pseudorandom number following a Landau distribution.
binomial(int ntot, double prob)	Returns a pseudorandom number in the $[0, \mathrm{ntot}]$ interval following a Binomial distribution.

to respectively keep or reject a collection element is provided in the form of a boolean expression. In addition to arithmetic and boolean operations, the expression can contain calls to functions. In the examples given in Ref. [1], the functions take the collection element as an implicit argument, but this might not always be the case. In LHADA17 the following rule applies: if the number of passed arguments is less by one with respect to the expected one, then the collection element to filter is assumed to be implicitly passed as first argument. A mismatch between the expected argument type and the collection element type is considered as ill-formed. While the `apply` statement is valid for an entity which is a single object (e.g. missing transverse momentum), the `select` and `reject` statements are restricted to collections. Blocks without a `take` statement are also allowed. In this case there is no implicit argument in the `apply` statement. It is strongly discourage to use this form when the one with a `take` statement can be used.

Three extensions to LHADA are introduced in LHADA17. The first is the backslash line continuation marker described above. The second extension is the `uid` attribute of the `object` block that is defined in the EBNF language description. This attribute, whose name stands for "universal identifier", identifies the `object`. In LHADA17, object blocks with a `take external` statement include a `uid` and no `apply` or `cut` statement. The `uid` attribute is reserved to this type of `objects`. Finally, we have introduced two aliases for the keyword `object`: `variable` and `collection`. The `object` block can represent several types of entities. Providing the possibility to use a name reflecting the type of entities, `collection` when dealing with a collection of physics object, `variable` when dealing with a single observable, like an event shape variable, should help in writing more intelligible analysis descriptions. The choice of the name is left to the discretion of the analysis description author.

3.2 Automatised generation of Rivet analysis code

The LHADA language will play a role for LHC result reinterpretations only if it is interfaced to commonly used reinterpretation frameworks. The interface can be done in two different ways. The first approach is to interpret the analysis description at run time. The second one, which is adopted here, is to generate code from the description.

A prototype application, called LHADA2RIVET, that produces a RIVET analysis from its description in LHADA17 has been developed. The analysis produces a cut flow table using. Special care has been taken to produce code in the RIVET style using facilities provided by the framework, like the projections or the `CutFlow` class.

Detector response simulation has been partly addressed. The random object was introduced in LHADA17 to allow the definition of the detector resolution and efficiency using a c++ function. Such functions will be added to the central database along with the defined objects. In addition, the object `uid` provides the identification of the objects whose detector response is already included in RIVET permitting to use the Rivet built-in detector response simulation in such case. Support for detector response simulation is expected to be included soon in the LHADA2RIVET interpreter based on these features.

The code produced by the LHADA2RIVET interpreter for the first analysis considered in the previous section can be found in appendix B. Each LHADA cut block is mapped to a c++ method. Rivet built-in tools, as Projections and Cutflow are used, leading to a clean code that is well integrated in the framework. The RIVET interface to the fastjet [13] library is used. The anti-k_T jet used in the analysis is identified as such by the interpreter based on the name of the function specified in the `apply` statement of the LHADA `object` block. This identification based on a name convention will be replaced by use the common library introduced in Sec. 3.1 in future update of the interpreter.

With this prototype we have investigated the different aspects that a result reinterpretation code generator based on a LHADA analysis description should cover. We can conclude from this exercise that the development of such a generator that takes as input a description compliant with the LHADA17 specifications can be done with a limited effort. The code produced by such a generator can be used to validate the analysis description using analysis cut flow, which needs to be provided by the analysis authors.

4. The LHADA2TNM interpreter

Two key design features of LHADA are human readability and analysis framework independence. As noted above, framework independence can be tested by attempting to implement tools that automatically translate analyses described using LHADA into analyses that can be executed in different analysis frameworks. Human readability demands that the number of rules and syntactical elements be kept as low as possible. But, since we also demand that LHADA be sufficiently expressive to capture the details of LHC analyses, it pays to follow Einstein's advice: "Everything should be made as simple as possible, but not simpler". In the prototype of the LHADA2TNM translator, we have tried to place the burden where it properly belongs, namely, on the translator. For example, it is expected that physicists will write LHADA files so that blocks appear in a natural order. However, the LHADA2TNM translator does not rely on the order of blocks within a LHADA file. The appropriate ordering of blocks is handled by the translator. Given a `cut` block called `signal`, which makes use of another called `preselection`, LHADA2TNM places the code for `preselection` before the code for `signal` in the resulting C++ file.

Another example of placing the burden on the translator rather than on the author of a LHADA file, concerns statements that span multiple lines. Many computer languages have syntactical elements to identify such statements. However, since LHADA is a keyword-value language, continuation marks are not needed because it is possible to identify when the value associated with a statement ends.

The LHADA2TNM translator is a `Python` program that translates a LHADA file to a C++ program that can be executed within the TNM n-tuple analysis framework. The translator extracts all the blocks from a LHADA file and places them within a data structure that groups the blocks according to type. The `object` and `cut` blocks are ordered according to their dependencies on other object or cut blocks. It is assumed that a standard, extensible, type is available to model all analysis objects and that an adapter exists to translate input types, e.g., DELPHES, ATLAS, CMS, etc., types to the standard type. In order to determine where a statement ends, LHADA2TNM looks ahead to the next record in the LHADA file during translation.

In the current version, instances of the standard, extensible, type as well as functions are placed in the global namespace of the C++ program so that the object and cut code blocks that need them can access them without the need to pass objects between code blocks. The name of a function defined in LHADA is assumed to be identical with that of a function, which, ultimately, will be accessed from an online code repository. However, this assumption can be relaxed if warranted in a later iteration of LHADA; for example, the appropriate function can be specified by its DOI. While the technical details of the automatic access of codes from an online repository need to be worked out, we see no insurmountable hurdles.

5. Conclusion

The sustainability of the LHADA language to be used in the context of analysis reinterpretation has been studied. Three questions have been addressed: the analysis range the language can cover, the completeness of the analysis description, and the capacity to validate the analysis description. The analysis coverage question has been tackled by taking three different LHC results and implementing their description. The language turns out to be very flexible and no difficulty has been encountered in this exercise giving confidence that the coverage covers the need. Nevertheless, it will be wise to extend the exercise with more sophisticated analyses. The two other questions have been addressed by developing the prototypes of two applications that interpret analysis descriptions written in LHADA. The development of an application generating reinterpretation code out of a LHADA analysis description can easily be done provided that the syntax used by the description is well-defined and not too flexible. A specialisation, LHADA17, of the LHADA language that fulfils this requirement has been set up. The restrictions introduced by LHADA17 have not been a limitation for the description of the analysis considered in the coverage test.

References

[1] G. Brooijmans *et. al.*, in *9th Les Houches Workshop on Physics at TeV Colliders (PhysTeV 2015) Les Houches, France, June 1-19, 2015*, 2016. 1605.02684.

[2] S. Sekmen and G. Unel, 1801.05727.

[3] A. Buckley, J. Butterworth, L. Lonnblad, D. Grellscheid, H. Hoeth, J. Monk, H. Schulz, and F. Siegert, *Comput. Phys. Commun.* **184** (2013) 2803–2819, [1003.0694].

[4] R. Brun and F. Rademakers, *Nucl. Instrum. Meth.* **A389** (1997) 81–86.

[5] S. Kraml *et. al.*, *Eur. Phys. J.* **C72** (2012) 1976, [1203.2489].

[6] C. Collaboration,, **CMS** Collaboration.

[7] M. Aaboud *et. al.*,, **ATLAS** Collaboration *Eur. Phys. J.* **C76** (2016), no. 7 392, [1605.03814].

[8] M. Aaboud *et. al.*,, **ATLAS** Collaboration *Eur. Phys. J.* **C77** (2017), no. 6 393, [1704.03848].

[9] tech. rep., The version these proceedings refers to is tagged Houches2017Proceedings.

[10] International Organization for Standardization, *ISO/IEC 14977* **1996** (1996).

[11] Wikipedia contributors,, 2018. [Online; accessed 5-February-2018].

[12] International Organization for Standardization, *ISO/IEC 14882* **2017** (2017).

[13] M. Cacciari, G. P. Salam, and G. Soyez, *Eur. Phys. J.* **C72** (2012) 1896, [1111.6097].

Appendices

A LHADA17 language syntax

```
(* Definition of the LHADA file structure *)

lhada file = info block, {function block}, {object block}, {cut block}, {table block}


(* Syntax of an info block *)

info block = "info analysis", EOL,
             {space, analysis info}

analysis info = analysis info key, text

analysis info key = "id" | "doc" | "experiment" | "publication" | "sqrtS" | "lumi" |
                    "arXiv" | "hepdata"


(* Syntax of a function block *)

function block = "function", space, function name, EOL,
                 {space}, "#", {char}, EOL,
                 {arg statement}
                 indent, "code", space, extented file path, EOL

function name = identifier
arg statement = indent, "arg", space, arg name, {space, arg name}, EOL
arg name = identifier

(* Syntax of an object block *)

object block = internal objecct | external object

internal object block = object, space, object name, EOL,
                        indent, "take", space, internal input, EOL,
                        {object optional statement}

object optional statement = apply statement | cut statement

internal input = "Particles" |  defined object

defined object = identifier (* the object must be defined in an object block in
                               the preceding text *)
```

```
external object block = object, space, object name, EOL,
                        indent, "take", space, "external", EOL,
                        "uid", uid

uid = identifier

apply statement = indent, "apply", space, function call

cut statement  = indent, "cut", expression

(* Syntax of a cut block *)
cut block = "cut",  space, cut name, EOL
             {"select", space, expression, EOL}

cut name = identifier

(* Syntax of a table block*)

table block = "table", space, table name, EOL,
               indent, "type", space, table type, EOL,
               indent, "columns", {space, column name}, EOL (* header *)
               {indent, "entry", {space, cell content}, EOL} (* lines *)
               (* The number of fields in table lines must match
                  with the number of columns defined in the header*)
               [indent, "hepdata", space, extended file path]

table name = identifier
table type = "events" | "limits" | "cutflow" | "corr" | "bkg"
cell content = (char - space}

(* Syntax for a function call *)

function call =  defined function, "(", arg list, ")"
defined function = function name (* The function must be defined
                                    with a function block before in
                                    the file *)
arg list = "" | arg, { ", ", arg}
arg = arg name, {space}, "=", {space}, expression, { ",", expression }


(* Miscellaneous *)

extended file path = ? http, https, or ftp url as defined in RFC 2386 ? |
                     ? unix-like path name ?


(* Syntax of a mathematical expression *)
(* Note: we have not expressed the operator precedence rules within the grammar below *)
(* The same precedence as the one of c/c++ language applied *)

expression = primary expression, {primary expression}

primary expression = real number | variable | unary operation | binary operation |
                     function call | "(", expression, ")"

variable = identifier

operator = "+" | "-" | "*" | "/" | "**" | "^" | "and" | "or" | "&&" | "||" | "<" | ">" |
           "<=" | ">=" | "!="
unary operator = "-" | "not" | "!"
unary operation = unary operator, ( real number | variable | unary operation |
                                    function call | "(" expression ")" )
```

```
binary operation = expression, binary operator, expression

(* Syntax of identifiers*)
identifier = ( letter | "_" ), {letter | "_" | digit }

(* Definition of character sets *)

letter = "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L" | "M" |
         "N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z" |
         "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m" |
         "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z"

digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

(* symbols include all 7-bit ASCII characters other than letters, digits and spaces *)
symbol =  "!" | """ | "#" | "$" | "%" | "&" | "'" | "(" | ")" | "*" | "+" | "," | "-" |
          "." | "/" | ":" | ";" | "<" | "=" | ">" | "?" | "@" | "[" | "\" | "]" | "^" |
          "_" | "`" | "{" | "|" | "}" | "~"

space char = " " | (? ISO 6429 character Horizontal tab ?)

char = letter | digit | symbol | space char

sign = "+" | "-"

exp10 sign = "e" | "E"

EOL = ( ?  ISO 6429 character Line Feed ?)

(* Definition of numbers *)

natural number = digit, {digit}

integer = [sign], ( digit - "0" ) , { digit }

real number = decimal, { exp10 sign, integer }

decimal = ( ( integer, [ "." ] ) | ( { integer }, ".", natural number) )

(* Definition of space *)

space = {space char}
indent = space (* Alias used for a space at beginning of a line *)
```

B Example of code produced by the LADHA2RIVET interpreter

```
// -*- C++ -*-
#include <iostream>
#include "Rivet/Analysis.hh"
#include "Rivet/Tools/Cutflow.hh"

#include <math.h>
#include "Rivet/Projections/FastJets.hh"
#include "Rivet/Projections/FastJets.hh"
#include "Rivet/Projections/MissingMomentum.hh"
namespace Rivet {


  class CMS_PAS_SUS_16_015: public Analysis {
  public:
```

```
///Cut ids
enum {kmt2_cut, kdeltaphi_etmiss_jet, kveto, kpreselection,
      kone_jet_selection, kat_least_two_jet_selection,
      k1j_loose_1, k1j_loose_2, k1j_medium_1, k1j_medium_2,
      k1j_medium_3, k1j_medium_4, k1j_medium_5, k1j_medium_6,
      k1j_medium_7, k1j_medium_8} CutIds;
/// Constructor
CMS_PAS_SUS_16_015():      Analysis("CMS_PAS_SUS_16_015"),
  cutflow("CutFlow", {"kmt2_cut", "kdeltaphi_etmiss_jet",
        "kveto", "kpreselection", "kone_jet_selection",
        "kat_least_two_jet_selection", "k1j_loose_1",
        "k1j_loose_2", "k1j_medium_1", "k1j_medium_2",
        "k1j_medium_3", "k1j_medium_4", "k1j_medium_5",
        "k1j_medium_6", "k1j_medium_7", "k1j_medium_8"})
{  }


/// Book histograms and initialise projections before the run
void init() {

  FinalState fs;
  VisibleFinalState visfs(fs);

  addProjection(FastJets(fs, FastJets::ANTIKT, 0.4), "jets_eta47");
  addProjection(FastJets(fs, FastJets::ANTIKT, 0.4), "bjets");
  addProjection(MissingMomentum(fs), "met");
}

template<typename P>
double  scalar_pt_sum(const std::vector<P>& momenta){
    double ht  = 0;
    for(const auto& p: momenta){
      ht += p.pt();
    }
    return ht;
}


double  btag_eff(const Particles& bjets){
    return 1.;
}


template<typename P>
double  mt2(const P& particle1, const P& particle2, const P& met){
    return P();
}


template<typename P1, typename P2>
double  dphi(const P1& p1, const P2& p2){
    double r = acos(p1.px()*p2.px() + p1.px()+p2.py() )
      / sqrt((p1.px()*p1.px() + p1.py()*p1.py())
             * (p1.px()*p1.px() + p1.py()*p1.py()));
    return isnan(r) ? 0 : r;
}


bool cut_mt2_cut(){
    bool r = true;
    return cutflow.fill(kmt2_cut, r);
}
```

```
bool cut_deltaphi_etmiss_jet(){
    bool r = true;
    r &= (dphi(met, jets[1 - 1]) > 0.3)  && (dphi(met, jets[2 - 1]) > 0.3)
      && (dphi(met, jets[3 - 1]) > 0.3)  && (dphi(met, jets[4 - 1]) > 0.3);
    return cutflow.fill(kdeltaphi_etmiss_jet, r);
}

bool cut_veto(){
    bool r = true;
    return cutflow.fill(kveto, r);
}

bool cut_preselection(){
    bool r = true;
    r &= cut_mt2_cut();
    r &= cut_deltaphi_etmiss_jet();
    r &= cut_veto();
    return cutflow.fill(kpreselection, r);
}

bool cut_one_jet_selection(){
    bool r = true;
    r &= cut_preselection();
    r &= jets.size() == 1;
    r &= jets[1 - 1].pt() > 200;
    return cutflow.fill(kone_jet_selection, r);
}

bool cut_at_least_two_jet_selection(){
    bool r = true;
    r &= cut_preselection();
    r &= jets.size()> 1;
    r &= cut_mt2_cut()> 200;
    return cutflow.fill(kat_least_two_jet_selection, r);
}

bool cut_1j_loose_1(){
    bool r = true;
    r &= cut_one_jet_selection();
    r &= ht> 575;
    return cutflow.fill(k1j_loose_1, r);
}

bool cut_1j_loose_2(){
    bool r = true;
    r &= cut_at_least_two_jet_selection();
    r &= bjets.size() < 3;
    r &= ht> 575;
    r &= ht < 1000;
    r &= cut_mt2_cut()> 200;
    return cutflow.fill(k1j_loose_2, r);
}

bool cut_1j_medium_1(){
    bool r = true;
    r &= cut_one_jet_selection();
    r &= bjets.size() < 1;
    r &= ht> 1000;
    return cutflow.fill(k1j_medium_1, r);
}
```

```
bool cut_1j_medium_2(){
    bool r = true;
    r &= cut_one_jet_selection();
    r &= bjets.size()> 0;
    r &= ht> 575;
    return cutflow.fill(k1j_medium_2, r);
}

bool cut_1j_medium_3(){
    bool r = true;
    r &= cut_at_least_two_jet_selection();
    r &= jets.size() < 4;
    r &= bjets.size() < 1;
    r &= ht> 575;
    r &= ht < 1000;
    r &= cut_mt2_cut()> 800;
    return cutflow.fill(k1j_medium_3, r);
}

bool cut_1j_medium_4(){
    bool r = true;
    r &= cut_at_least_two_jet_selection();
    r &= jets.size() < 4;
    r &= bjets.size()> 0;
    r &= bjets.size() < 3;
    r &= ht> 575;
    r &= ht < 1000;
    r &= cut_mt2_cut()> 600;
    return cutflow.fill(k1j_medium_4, r);
}

bool cut_1j_medium_5(){
    bool r = true;
    r &= cut_at_least_two_jet_selection();
    r &= jets.size() < 4;
    r &= bjets.size() < 2;
    r &= ht> 1000;
    r &= ht < 1500;
    r &= cut_mt2_cut()> 800;
    return cutflow.fill(k1j_medium_5, r);
}

bool cut_1j_medium_6(){
    bool r = true;
    r &= cut_at_least_two_jet_selection();
    r &= jets.size() < 4;
    r &= bjets.size() == 2;
    r &= ht> 1000;
    r &= ht < 1500;
    r &= cut_mt2_cut()> 400;
    return cutflow.fill(k1j_medium_6, r);
}

bool cut_1j_medium_7(){
    bool r = true;
    r &= cut_at_least_two_jet_selection();
    r &= jets.size() < 4;
    r &= bjets.size() < 2;
    r &= ht> 1500;
    r &= cut_mt2_cut()> 400;
    return cutflow.fill(k1j_medium_7, r);
}
```

```
  bool cut_1j_medium_8(){
      bool r = true;
      r &= cut_at_least_two_jet_selection();
      r &= jets.size() < 4;
      r &= bjets.size() == 2;
      r &= ht> 1500;
      r &= cut_mt2_cut()> 200;
      return cutflow.fill(k1j_medium_8, r);
  }
  /// Perform the per-event analysis
  void analyze(const Event& event) {

     const FastJets& jets_eta47Proj = applyProjection<FastJets>(event, "jets_eta47");
     Jets jets_eta47 = jets_eta47Proj.jetsByPt();
     Jets jets;
     for(const auto& p: jets_eta47){
       if((p.eta() < 2.4)){
         jets.push_back(p);
       }
     }

     const FastJets& bjetsProj = applyProjection<FastJets>(event, "bjets");
     Jets bjets = bjetsProj.jetsByPt();
     const MissingMomentum metProj = applyProjection<MissingMomentum>(event, "met");
     met = metProj.missingMomentum();
     ht = scalar_pt_sum(jets);
  }


  /// Normalise histograms etc., after the run
  void finalize() {
     std::cout << "Analsyis cut flow:\n"
               << "-----------------\n\n"
               << cutflow << "\n";
  }


protected:

  //@{
  /** Collections and variables
   */
  //@}
   /** Analysis objects
    * @{
    */
   Jets jets_eta47;

   Jets jets;

   Jets bjets;

   FourMomentum met;

   double  ht;
   /** @}
   */    //@{
  /** Histograms
   */
  //@}
```

```
    ///Tracks the event counts after each cut
    Cutflow cutflow;

  };
  DECLARE_RIVET_PLUGIN(CMS_PAS_SUS_16_015);
}
```