Preprint typeset in MyNote style

28 May 2016

Deductor 2.0: How to use it

Zoltán Nagy

DESY
Notkestrasse 85
22607 Hamburg, Germany
E-mail: Zoltan.Nagy@desy.de

Davison E. Soper

Institute of Theoretical Science
University of Oregon
Eugene, OR 97403-5203, USA
E-mail: soper@uoregon.edu

Abstract: These notes explain how to use the parton shower event generator Deductor.

Keywords: perturbative QCD, parton shower.

Contents

1 Introduction

Welcome to the Monte Carlo event generator DEDUCTOR 2.0. In these notes, you will learn how to use the program. The program is organized as a main program, `deductor`, that calls certain modules. There is a kernel module which defines some basics of your calculation like the c.m. energy of a hadron-hadron collider and what parton distribution functions to use. Then there are user modules that perform the analysis of generated events. The kernel module and the user modules are created by users. We provide the main `deductor` code and, separately, a kernel module and a selection of user modules to get you started.

These notes begin with section 2, which explains how to obtain the code and get it running. Section 3 explains the structure of a simple user moduled for the Drell-Yan process. Section 4 explains the structure of a more complex user module for jet production. Section 5 provides some information about the elements of the DEDUCTOR C++ library.

These usage notes are a work in progress. More will be available in the future.

2 Quick start

Here are some brief instructions to help you get DEDUCTOR running and to begin to understand its organization.

2.1 Getting the code

The DEDUCTOR source code can be downloaded either from DESY

```
curl -O http://www.desy.de/~znagy/deductor/deductor-X.Y.Z.tar.gz
```

or from the University of Oregon

```
curl -O http://pages.uoregon.edu/soper/deductor/deductor-X.Y.Z.tar.gz
```

replacing `X.Y.Z` with the proper version number. The current version number is 2.0.1. After download, the code has to be unpacked:

```
tar zxvf deductor-X.Y.Z.tar.gz
```

DEDUCTOR needs some user written or user modified code. On our website we provide some simple examples for a quick start. So, as the next step download and unpack the package `deductor-user-X.Y.Z.tar.gz`. Here the current version number is still 2.0.0. Execute

```
curl -O http://www.desy.de/~znagy/deductor/deductor-user-X.Y.Z.tar.gz
```

or

```
curl -O http://pages.uoregon.edu/soper/deductor/deductor-user-X.Y.Z.tar.gz
```

followed by

```
tar zxvf deductor-user-X.Y.Z.tar.gz
```

2.2 Compiling the main code

Now, we have to configure the source and create the makefiles. The program is written in C++14 and it requires a C++ compiler that supports this standard. It is important that the compiler has to provide full support for C++14 features. DEDUCTOR was tested with `g++` version 5.2.0, 5.3.0, 6.1.0 and with `clang` version 3.8.0 on a MacOSX 10.11 platform and on a Linux cluster. DEDUCTOR doesn't compile with older version of `clang` or with the APPLE's version of `clang` that comes with XCODE. On MacOSX we used `gcc` from MacPorts. To configure and install, you should execute

```
cd deductor-X.Y.Z
mkdir build
cd build
cmake .. -DCMAKE_C_COMPILER=gcc-mp-5 -DCMAKE_CXX_COMPILER=g++-mp-5
make install
```

After this, you could remove `build`. This will install the program to the default installation directory, `$HOME`. This can be changed in the configuration step. For example, to install into `/usr/local`, you could use

```
cmake .. -DCMAKE_INSTALL_PREFIX=/usr/local -DCMAKE_C_COMPILER=gcc-mp-5 -DCMAKE_CXX_COMPILER↩
    =g++-mp-5
make install
```

The executable files are in the `$CMAKE_INSTALL_PREFIX/bin` directory. Make sure that this directory is in your `$PATH`. The program libraries are in `$CMAKE_INSTALL_PREFIX/lib`. If everything went well, then you have a working DEDUCTOR setup. You can check it by executing `deductor --help`, which gives a message about usage.

To be able to do a real calculation, DEDUCTOR needs some auxillary data and a user module. DEDUCTOR doesn't use a custom `config` file or other script languages to provide input. We think C++ is the best way to configure our calculations. In the user module one has to specify the part of the calculation that generates the hard scattering process that initiates events. This has to be a proper event generator based on simple partonic matrix elements. In the current version we have only a few hard processes implemented, but hard processes generators can be provided externally. The other important input is the analyzer routine. This is where the events are analyzed and histograms are filled. We provide a powerful but still simple and flexible interface.

2.3 Creating a user module

We can now look at the user routines that analyze events. In the directory `deductor-user-X.Y.Z`, there are subdirectories with several examples. Let us choose the Drell-Yan process,

```
cd Drell-Yan
ls
```

We see that included in this directory are three C++ files:

```
analyzer-ZpT.cc analyzer-ZpT.h  mod-dy.cc
```

You can use these code files to create a user module, which you might want to call `dy`. Just execute

```
deductor --new dy mod-dy.cc analyzer-ZpT.h analyzer-ZpT.cc
```

This command creates a directory `dy.bundle` in `Drell-Yan` and places some needed files there.[1] Included in `dy.bundle` is a directory `src` that contains the code source files. Later, results files will be included in `dy.bundle`. The idea is that both results and the source files that produced the results are together, so that the source files document the results. Later, you may want to archive the `dy.bundle` directory by simply giving it a different name, say `dyJan2016.bundle`. Then you can modify the code files `analyzer-ZpT.cc`, `analyzer-ZpT.h`, and `mod-dy.cc` and produce different results. The code that you used to produce the results in `dyJan2016.bundle` will be there in `dyJan2016.bundle/src`.

While developing the code, you can, if you wish, modify the code in `dy.bundle/src`, leaving the code files in `DrellYan` unchanged. Alternatively, you may find it convenient to modify the main code files in `DrellYan`. In that case, you can copy the modified files to `dy.bundle/src` with the command

```
deductor --new dy mod-dy.cc analyzer-ZpT.h analyzer-ZpT.cc --update
```

Once the code is copied to `dy.bundle/src`, you can compile it with

```
deductor --build dy
```

[1] MacOS treats `xx.bundle` directories specially. To look at them in a Finder window, use `control-click`.

This produces a shared object file `module.so` in `dy.bundle`. Now you are ready to calculate.

2.4 Calculating and analyzing events

Now we have everything to start a calculation. Let us fire it up by running

```
deductor --run dy
```

The program starts like this (with a little color added):

```
+------------------------------------------------------------------------+
|                        Deductor release 2.0.0                          |
|                        Z. Nagy and D.E. Soper                          |
|                                                                        |
|             A parton shower Monte Carlo event generator                |
|          not including an underlying event or hadronization            |
|             http://www.desy.de/~znagy/Site/deductor.html               |
|               http://pages.uoregon.edu/soper/deductor/                 |
|                                                                        |
| Please cite JHEP 1406 (2014) 097 [arXiv:1401.6364] if you use this     |
| package for scientific work.                                           |
|                                                                        |
| Deductor is provided without warranty under the terms of the GNU GPLv3. |
| See COPYING file for details.                                          |
+------------------------------------------------------------------------+

User bundle        : dy
Bundle path        : dy.bundle
Name of the run    : run1896-2016-Mar-26-172401@DESMacbook.home
```

Under the hood, DEDUCTOR builds some needed files and puts them into a directory `models` in a directory `.deductor` in your home directory. This takes about ten minutes. In later runs, the files will already be there, so the program will start producing results sooner.

Once DEDUCTOR starts producing results, it will store its results periodically in a directory `dy.bundle/output`. You can look at the results by opening a second terminal process and executing

```
deductor --results dy
```

This produces a directory `dy.bundle/result`. Inside of `result`, you will find a file `summary.tex`. This is a simple TEXfile with links to the files DEDUCTOR has produced that contain the results. If you typeset `summary.tex`, you will see the results in graphical form.

Perhaps the graphs suggest that you don't yet have enough events. In that case, you can wait until you have more events. Then you can run `deductor --results dy` again to update the results and typeset `summary.tex` again and see the improved results. When your results are good enough, you can stop the program with `command-period` or `control-c`.

It is simple to run the example code, but users should be able to create their own analyzer routines. In some applications they may want to change the underlying hard process. In the following sections, we discuss all the user interfaces.

If you work on a cluster you might want to start several jobs simultaneously on several nodes. Note that DEDUCTOR is multi-threaded and when your start it, it takes over all cores and threads of you CPUs in the computer. Thus on a cluster you should start only one DEDUCTOR job on every node. DEDUCTOR needs PDF tables and PDF evolution is calculated in the first run and

the tables are saved in `${HOME}/.deductor` directory. When you work on a cluster, at the first time just start only one job, wait until the PDF tables get generated and then you can start more jobs on other nodes.

3 Make your own user module

The usual event generators (*e.g.* PYTHIA) provide a routine that can be called inside a loop in user created code. Every time this function is invoked it returns with an event. It is the user's responsibility to manage the event loop. DEDUCTOR is organized in a event driven way. A stream of events is generated and the analyzer routines can be attached to this stream. The idea behind this concept is to make the user interface simple and flexible.

The user module consists of two parts, the main part and the analyzer part. In the main part, we set up the kernel, start the processes and make the connection to the analyzers. In the analyzer part we deal with the events. For example the event analyzer routines set up and fill histograms representing data from the generated events.

3.1 Main part of the Drell-Yan user module

In the main part of the user module we have to set up the kernel, the hard process, and the analyzers and then we have to start the shower process. Let us set up a Drell-Yan process as a simple example. A compact version of the main part of the user module should look something like this:

Listing 1: User module for Drell-Yan process — Drell-Yan/mod-dy.cc

```
/*   deductor headers  */
#include <deductor.h>
#include <proc-dyjets.h>

/*   local headers  */
#include "analyzer-ZpT.h"


/*    This function defines what Deductor should do.    */
void my_dy_module(const deductor_input&)
{
  /*  deductor object */
  deductor<hhc> kernel{qcd::ct14n};

  /*   Hard process   */
  auto hard = [=](const model_ref& mdl) -> dyjets
  {
    int proton = 2212, electron = 11;
    double Ecm = 13.0_TeV, scalefac = 1.0, mL = 2.0_TeV, mH = 2.1_TeV;

    return {mdl, Ecm, mL, mH, scalefac, proton, proton, {electron}};
  };

  /*   Drell-Yan Z pT distribution    */
  kernel("Drell-Yan", hard, new UserZpT{"ZpT"});
}


```

```
29 /*   Always export your module functions!    */
30 export_my_module drell_yan_module = {
31   {"Drell-Yan processes at 13TeV", my_dy_module}
32 };
```

This needs discussion.

1. We start with header files to include in lines 2 and 3. We have to include the main DEDUCTOR header. In line 3 we include the header file of the hard process, the Drell-Yan process. In DEDUCTOR there are some hard process implemented and one could also use a user supplied hard process. In line 6 we include the header of the analyzer routine. In this example we have only one analyzer, which calculates the p_T distribution Drell-Yan lepton pair. The analyzers are user defined. We will discuss how to define them in the next subsection.

2. In line 10, we begin the definition of our module for Drell-Yan calculations, which is a function that will tell DEDUCTOR what calculations to do. We call it `my_dy_module`. It has one argument and its type has to be `const deductor_input&`. Since in this example we don't use this argument, so we don't name it.

3. In line 13, we create an object called `kernel` of class `deductor<hhc>` where `hhc` specifies "hadron-hadron collisions". We specify that we want to use parton distributions based on evolution from the starting functions used in the CT14 NLO distributions (which is built into DEDUCTOR). This also fixes the strong coupling, α_s.

4. In line 16 we create a function object that will create the hard process. It is created as a C++14 lambda expression. The objects that we need is the return value of this lambda expression and its type is `dyjets`. There is one parameter, a reference to the model, the value of which is provided by the `deductor<hhc>` objects. In the body of the lambda expression, we specify the parameters needed by the `dyjets` object.

5. In line 18, we define the PDG codes for the particles involved in the hard process. The colliding hadrons are protons, and we want to generate final state electrons.

6. In line 19, we choose the c.m. energy of the hard process, 13 TeV. (`double Ecm = 13.0_TeV;`)

7. We choose the scale factor λ that defines the factorization scale μ_f (which is the scale for the starting parton distribution factor and is the starting scale for the shower): $\mu_f = \lambda Q$, where Q is the e^+e^- mass. (`double scalefac = 1.0;`)

8. We specify parameters for generating the hard process. We want to generate events in the region where the invariant mass Q of the lepton pair is between 2.0 TeV and 2.1 TeV (`double mL = 2.0_TeV, mH = 2.1_TeV;`).

9. In line 21, we return a `dyjets` process with the chosen parameters. For the final state leptons, we have a list, which, in this case, contains just one element: `electron`. Alternatively, we could define `muon` and `tau` and then replace `{electron}` by `{electron, muon, tau}`.

10. Line 25 does several things at once. In the file `analyzer-ZpT.h` that we included, we declared a class `UserZpT`, which is a user defined analyzer for Drell-Yan events. Here, we create a dynamically allocated (with `new`) instance of this class. (DEDUCTOR will take

care of deallocating this object.) Our `UserZpT` object is given a text name `"ZpT"`. It is important that the analyzers objects has to be created dynamically by the `new` operator. The `deductor<hhc>` object captures and owns this pointer.

11. Continuing in line 25, recall that we created a `deductor<hhc>` object `kernel`. Now, we use the `operator()` function of `kernel` to start the calculation. We supply to `kernel` a process name, `"Drell-Yan"`, the function `hard` that creates the hard process, and the analyzer.

12. Line 26 competes the definition of `my_dy_module`.

13. Finally, in line 30, we create an object (with a dummy name `drell_yan_module`) of type `export_my_module` defined in DEDUCTOR. This object lets `deductor` know the class and function names associated with our calculation. We supply a descriptive module name `"Drell-Yan-13TeV"` and the function, `my_dy_module`, that we have just defined.

14. Notice that the user has supplied text names for the bundle created by the `deductor --new` command, the module, the process, and the analyzer. These names are used, for instance, in the organization of the results files. They should be plain text names with letters, numbers, and hyphens so that they are suitable for use as directory names in your file system and as text input to TEX.

3.2 Analyzer part of the Drell-Yan user module

In the code for the Drell-Yan module in Listing 1, we created an instance of the analyzer class `UserZpT`. Now we need to declare and define this class. We derive `UserZpT` from the class `basic_user<hhc,...>`, which is in turn derived from a lower level class `user<hhc>`. The class `basic_user<hhc,...>` provides an interface to shower events and also provides powerful histograming functions. In order to derive the analyzer from class `basic_user`, the user has to specify only two virtual functions.

Let us use the Drell-Yan example to see how this works. We want to analyze the transverse momentum distribution of Z/γ-bosons. We call the analyzer `UserZpT`. The header file is

Listing 2: Definition of the class UserZpT — Drell-Yan/analyzer-ZpT.h

```
1  #ifndef __analyser_ZpT_h__
2  #define __analyser_ZpT_h__
3
4  /*   deductor headers  */
5  #include <user.h>
6
7  /*   It is not a library just a module. We can use using namespace...  */
8  using namespace std;
9  using namespace duct;
10 using namespace duct::distpoint;
11
12 /*   Narrowing the basic_user template   */
13 using DistHist1D = distbook<double, hist1d>;
14
15 /*   This is the declaration of the analyser class.    */
16 class UserZpT : public basic_user<hhc, DistHist1D>
17 {
18   /*   useful aliases  */
19   using Hist1D = Phys<DistHist1D>;
```

```
20
21  public:
22    /*   constructor   */
23    UserZpT(const char *n, const duration_type& p = 2.0min) { this->param(n, p);}
24
25    /*   User defined init function, called once at the beginning of the calculation. */
26    void initfunc();
27
28    /*   The analyser routine. It analyses the events and fills the histograms.   */
29    void userfunc(const shower_history<hhc>&);
30  };
31
32  #endif
```

This is a very simple analyzer class. It still needs some discussion.

1. This is a header file, thus we have to make sure it will be included only once. This is done in lines 1,2 and 32 in the usual way.

2. In line 5, we include the `deductor` header `user.h`. It brings all the necessary declarations and definitions that are needed for an analyzer based on the `basic_user` class.

3. In lines 8-10, we specify some namespaces that will be convenient.

4. In line 13, we give a convenient name to a histogramming class that we will use.

5. In line 16, we begin the declaration of our class `UserZpT`. It is derived from the class `basic_user<hhc, ...>`. The template parameter `hhc` specifies that we want to calculate something for processes in hadron-hadron collision. The remaining template parameter is histogramming class that we will use.

6. In lines 19, we define an alias that will be convenient for the code in `analyzer-ZpT.cc` in which we define what `UserZpT` does.

7. In lines 23, we define a constructor our class. (We used it in line 25 of `mod-dy.cc`.) It has two arguments. The first is the name of the analyzer and the second is a duration type variable that specifies the elapsed time between two backups of the result. This argument could be omitted: it is default value is 2 minutes.

8. As we mentioned earlier, we have to overload two virtual functions of the class `basic_user`. They are declared in lines 26 and 29. We will define them in the implementation file (`analyzer-ZpT.cc`). The function `initfunc()` is called at the start of the calculation and is used to set up the analysis. Then the function `userfunc(const shower_history<hhc>&)` is called for each event. The event history is passed to it, ready for analysis.

Now we need to define the two functions for our analyzer. We use

Listing 3: The analyzer code of the class UserZpT — Drell-Yan/analyzer-ZpT.cc

```
1  /*   local headers  */
2  #include "analyzer-ZpT.h"
3
4  void UserZpT::initfunc()
5  {
```

```
6     /*   Creating a histogram to calculate the pT distribution   */
7     unsigned nbins = 100;
8     double pTmin = 0.0_GeV, pTmax = 100.0_GeV;
9     auto bins = spacing<hist1d>::linear(nbins, pTmin, pTmax);
10    bool enable_norm = true;
11
12    Hist1D::phys(1, "$Z/\\gamma$ $p_T$ distribution", bins, enable_norm);
13  }
14
15  void UserZpT::userfunc(const shower_history<hhc>& history)
16  {
17    /*   Conversion factor from 1/GeV^2 to nb    */
18    constexpr double tonb = 389379.338;
19
20    /*   the shower stage after the shower evolution   */
21    for(auto& stage : history)
22    {
23      /*   momentum of the e+e- pair   */
24      auto& p = stage.state;
25      auto pee = p[-2].momentum + p[-3].momentum;
26
27      /*   Filling the histogram   */
28      if(pee.perp() < 100.0) {
29        weight_type w = weight(stage, tonb);
30
31        Hist1D::normfill(1, w[0]);
32        Hist1D::physfill(1, w[0], dirac{}, pee.perp());
33      }
34    }
35  }
```

The function `UserZpT::initfunc()` is called at the beginning of the run. Let's see what it does.

1. We are going to create a one dimensional histogram to accumulate the Z/γ transverse momentum distribution. In lines 7-9, we define the bins. We choose 100 equally spaced bins from $p_T = 0$ GeV to $p_T = 100$ GeV. There are more ways to create bins, but this simple method suffices for now.

2. We will want a normalized distribution, so in line 10 we define a `bool` variable `enable_norm` that we set to `true`. As we will see below, we use this mechanism to arrange that probabilities are stored in the bins and these probabilities sum to 1.

3. In line 12, we define the histogram that we want by using the function `phys(...)` with four arguments. The first is an integer, `1`. This is the ID of the histogram, it can be any integer number but unique for every histogram of this type. In the user function, we will use this ID to refer to our histograms. The second is a name for our histogram, which includes some special characters for LaTeX. The third is the set of bins, just defined. The fourth is an optional `bool` argument that defaults to `false`. In this case, we choose `true`, indicating that we want a normalized histogram.

Next, we define the function `UserZpT::userfunc(...)`, which is called for each event. Let's see what it does.

1. The argument of `userfunc(...)` is a constant reference to an object of type `shower_history<hhc>`, which contains the entire history of the event. We call the shower history `history`. Deductor will supply `history` when it calls `userfunc(...)`.

2. We will be interested in analyzing the partons in the final state of the shower. In contrast to other parton shower programs, DEDUCTOR can generate more than one final state in the same operation. For this reason, our `shower_history<hhc>` object can contain more than one end stage of the shower. In this example, there will be only one final state, but in order to match the most general structure of a `shower_history<hhc>` object, we need to iterate over the shower end stages that we have generated. Thus in line 21, we loop over all the possible showered stages and the loop variable is `stage`.

3. In line 24 we define the final state that we have found and call it `p`. The information about the final state partons is contained in `p`.

4. In line 25, we define the momentum of the lepton pair. With the DEDUCTOR labelling convention, this is `pee = p[-2].momentum + p[-3].momentum`. (The incoming partons carry labels -1 and 0. Outgoing QCD partons carry labels $1, 2, 3, \ldots$. Outgoing non-QCD partons carry labels $-2, -3, \ldots$.)

5. Our histogram covers the range $0 < P_{\mathrm{T}} < 100$ GeV, so we look only at events with P_{T} in this range. The `if` statement in line 28 takes care of this.

6. In DEDUCTOR, each event comes with a weight. In lines 29, we calculate the weight, `w`. Every stage of the shower evolution comes with several weights. The object `stage` has a member called `weight` and its type is `weight_set<hhc>`. Thus is a structure and it is defined by

```
struct weight_set<hhc>
{
  /*   weight of the unitary part of the shower   */
  struct {
    /*   this is the weight of the unitarized part   */
    double main;

    /*   ratio of NLO an LO MSbar PDFs at the hard state   */
    double pdf_ratio;
  } unitary;


  /*   threshold correction weights   */
  struct {
    /*   this is the main part of the threshold weight   */
    double main;

    /*   this is the delta part of the threshold weight   */
    double delta;
  } threshold;
};
```

   This way we have full access to the weight factors and we can use whichever combination of factors we want. In this simple example we just simply use a predefined function instead of assembling the weight factor.

7. In line 29, invoking of function `weight(stage, tonb)` returns a 4 element array of type `weight_type = std::array<double,4>`. This function is defined in the DEDUCTOR as

```
weight_type weight(const shower_stage<hhc>& stage, double tonb = 1.0)
```

```
{
  std::array<double,4> w;

  w[3] = stage.weight.unitary.main*color_weight(stage)*tonb;
  w[2] = w[3]*stage.weight.threshold.main;
  w[1] = w[2]*stage.weight.threshold.delta;
  w[0] = w[1]*stage.weight.unitary.pdf_ratio;

  return w;
}
```

Here `w[3]` is the weight of unitary (standard) shower. The first factor that we is `stage.weight.unitary.main`. This is the weight factor for a unitary shower (not including the color weight, which is included later). This factor is often just the weight from the hard scattering that initiates the shower, but when we use the LC+ color treatment, it can be different from that. Next, we multiply by the color weight of the final state, obtained from the overlap of the bra and ket color states at the end of the shower. Finally, we multiply by the conversion factor from $1/\mathrm{GeV}^2$ to nb.

The value of `w[2]` is the shower weight with main contributions of the threshold logarithms, `stage.weight.threshold.main`. The value of `w[1]` contains the contributions of Δ_{ak} and Δ_{bk}, `stage.weight.threshold.delta`. Finally, we have `weights.unitary.pdf_ratio`, which multiplies by the ratio of the NLO $\overline{\mathrm{MS}}$ parton distribution at the hard interaction to the enhanced LO $\overline{\mathrm{MS}}$ parton distribution that is used within DEDUCTOR.

In this little example we want to put into the histogram only `w[0]`, the full weight.

8. Our one dimensional histogram for the transverse momentum distribution is to be normalized. That is, we divide the cross section in each bin by a normalization factor that we supply. The normalization factor that we want is the sum of the cross sections in all of the bins. In line 31, we accumulate this factor using the `normfill` function for this histogram. We identify the histogram that we want by supplying its `ID`, namely `1`.

9. In line 32, we enter the event into histogram `1`. The argument `dirac{}` says that we want the standard way of entering events into a histogram, in which each event goes into a single bin according to the value of the independent variable, `pee.perp()`. Other choices are possible, but we don't discuss them here. Which bin the event goes into is defined by the argument `pee.perp()`. The weight `w[0]` is entered into this bin.

When you run this with `deductor --run dy` and then produce results with `deductor --result dy`, you will get a `result` directory in `dy.bundle`. There you will find a TEXfile `summary.tex`, which, when you typeset it, will contain a graph of the Drell-Yan p_{T} distribution. The data produced by the one dimensional histograms that you declared is in the subdirectory `Drell-Yan-13TeV/Drell-Yan/ZpT` in file `result-1`. The data for the particular plot is in the further subdirectory `plot-1` in file `table-1.dat`.

4 Make another user module

Let's try another user module. This one will be a little more complicated. We will make a user module and an analyzer that can perform a calculation of the one jet inclusive cross section.

4.1 Main part of the jets user module

We need a user module for our jet calculation:

Listing 4: User module for jet studies — Jets/mod-jets.cc

```
1  /*   Deductor headers  */
2  #include <deductor.h>
3  #include <proc-hhcjets.h>
4
5  /*   Analyser headers  */
6  #include "analyzer-jets.h"
7
8  auto my_jet_module(const deductor_input&)
9  {
10   /*  Using a previously creating the kernel   */
11   deductor<hhc> kernel{qcd::ct14n};
12
13   /*    Hard process     */
14   auto hard = [=](const model_ref& mdl) -> hhcjets
15   {
16     int proton = 2212;
17     double Ecm = 13.0_TeV, scalefac = 1.0;
18     auto pTlist = {
19       100.0, 300.0, 400.0, 600.0, 800.0, 1200.0, 1600.0,
20       2000.0, 2500.0, 3000.0, 3500.0, 4000.0, 4500.0, 5000.0
21     };
22
23     return {mdl, Ecm, pTlist, scalefac, proton, proton};
24   };
25
26   /*   Jet cross section pT distribution */
27   kernel("Jet Calculations", hard, {
28     new UserJets{"Jets kT R = 0.2", fj::JetDefinition{fj::antikt_algorithm, 0.2}},
29     new UserJets{"Jets kT R = 0.4", fj::JetDefinition{fj::antikt_algorithm, 0.4}},
30     new UserJets{"Jets kT R = 0.7", fj::JetDefinition{fj::antikt_algorithm, 0.7}}
31   });
32 }
33
34 /*   Always export your module functions!   */
35 export_my_module jets_module = {
36   {"Jets at LHC 13TeV", my_jet_module}
37 };
```

This needs discussion.

1. We begin in lines 2 and 3 with DEDUCTOR headers. The `proc-hhcjets.h` header is for the hard process generator for $2 \to 2$ QCD scattering in hadron-hadron collisions. s

2. In line 6, we include the header for our analyzer routine. (We can have more than one analyzer. Then we need more `#include` statements.)

3. In line 8, we begin the definition of our function to tell DEDUCTOR what calculations to do for the jet calculation that we want. Its single argument will be supplied by DEDUCTOR, but in this example we don't use it. We call this function `my_jet_module`.

4. In line 11, just as in as in Listing 1, we create an object called `kernel` of class `deductor<hhc>` where `hhc` specifies "hadron-hadron collisions." We specify that we want to use parton distributions based on evolution from the starting functions used in the CT14 NLO distributions (which is built into DEDUCTOR).

5. In line 14 we create the function object (a C++14 lambda expression) that will create the hard process. The value of the one parameter, `mdl` is provided by the class `deductor<hhc>`. The function returns an object of type `hhcjets`. In the body of the lambda expression, we specify the parameters needed by the `hhcjets` object.

6. In line 16, we specify the PDG code for our incoming hadrons, which are protons.

7. In line 17, we specify the c.m. energy that we want and a scale factor λ for the factorization scale μ_f that we want. We set $\mu_f = \lambda p_T$, where p_T is the transverse momentum of the outgoing partons at the hard interaction. The scale of the parton distributions at the hard interaction is μ_f. The renormalization scale for the α_s at the hard interaction is set to μ_f and the starting scale of the shower is also set to μ_f.

8. In line 18, we provide a list of transverse momenta. These tell the generator to generate hard scattering events with 150 GeV $< p_T <$ `Ecm`/2, with equal numbers of events with p_T between 100 GeV and 300 GeV, between 300 GeV and 400 GeV, ..., and between 5000 GeV and `Ecm`/2. You can choose what you want, but if you just set `auto pTlist = {100.0}`, you will get very few events with $p_T > 300$ GeV.

9. In line 23, we return an `hhcjets` object with the parameters that we have set.

10. Recall that we created a `deductor<hhc>` object `kernel`. Now, in line 33, we use the `operator()` function of `kernel` to start the calculation. We supply to `kernel` a process name, `"Jet Calculations"`, the function `hard` that creates the hard process, and the list of three analyzers.

11. In the file `analyzer-jets.h` that we included, we declared a class `UserJets`, which is a user defined analyzer for jet events. Here, we create a dynamically allocated (with `new`) instances of this class. Our analyzer will use FASTJET to find jets. In particular, we use the `fjcore` version and have included `fjcore.hh` and `fjcore.cc` in the `deductor-user` files. We specify the jet definition that we want FASTJET to use, namely the anti-k_T algorithm with $R = 0.2, 0.4, 0.7$. (We define the namespace `fj` to be an abbreviation for `fjcore` in the DEDUCTOR header file `fastjet.h`, which is included in `analyzer-jets.h`)

12. Line 32 competes the definition of `my_jet_module`.

13. Finally, in lines 36 and 37, we create an object `jet_module` of type `export_my_module`. This lets `deductor` know the class and function names associated with our calculation. We supply an indication of what type of module this is a plain text module name `"Jets at LHC 13TeV"` and the name, `my_jet_module`, of the function that we have just created.

4.2 Analyzer part of the jets user module

In the code for the jets module in Listing 4, we created an instances of the analyzer class `UserJets`. Lets now look at how we define `UserJets` . The header file of this is

Listing 5: Definition of the class UserJets — Jets/analyzer-jets.h

```
1  #ifndef __analyser_jets_h__
2  #define __analyser_jets_h__
3
4  /*   project headers  */
5  #include <user.h>
6
7  /*   other includes   */
8  #include "fastjet.h"
9
10 /*   It is not a library just a module. We can use using namespace...  */
11 using namespace std;
12 using namespace duct;
13 using namespace duct::distpoint;
14
15 /*   Narrowing the distbook template   */
16 using DistPlain = distbook<std::array<double, 2u>, hist1d>;
17 using DistRatio = distbook<std::array<double, 3u>, hist1d, sample_ratio_traits<std::↩
       array<double, 3u>>>;
18
19 /*      This is the declaration of the analyser class.   */
20 class UserJets : public basic_user<hhc, DistPlain, DistRatio>
21 {
22   /*   useful aliases  */
23   using Plain = Phys<DistPlain>;
24   using Ratio = Phys<DistRatio>;
25
26 public:
27   /*    constructors     */
28   UserJets(const char *n, fj::JetDefinition&& jd, const duration_type& d = 2.0min)
29   : jetDef{std::move(jd)} { this->param(n, d);}
30
31   UserJets(const char *n, const fj::JetDefinition& jd, const duration_type& d = 2.0min)
32   : jetDef{jd} { this->param(n, d);}
33
34   /*   User defined init function. It is called once at the beginning of the ↩
         calculation.  */
35   void initfunc();
36
37   /*   The analyser routine. It analyses the events and fill the histograms.   */
38   void userfunc(const shower_history<hhc>&);
39
40   /*   For fastjet analysis - select algorithm and parameters.    */
41   fj::JetDefinition jetDef;
42
43   /*   Predefined vectors of Pseudojets     */
44   mutable std::vector<fj::PseudoJet> theparticles;
45   mutable std::vector<fj::PseudoJet> incJets;
46 };
47
48 #endif
```

We discuss what is in this header file:

1. In line 5, we include the DEDUCTOR user header.

2. We will use FASTJET in the form of `fjcore`. In line 8, we include a header file `"fastjet↩.h"`. This file includes `"fjcore.hh"` and it defines a translation `make_pseudojets(...)` that

translates from a DEDUCTOR shower state record to a vector of FASTJET PseudoJets. This file also defines `fj` to be short for `fjcore`.

3. In lines 11-13 we declare that we are using some convenient namespaces.

4. In lines 16-17, we define an alias for a `basic_user` class definition: `DistPlain` defines the usual one-dimensional histograms with two doubles as sample type and `DistRatio` defines also a one-dimension histograms with three dimensional array as sample type and it also provides the ratios of first to third and second to third elements of the sample array.

5. In line 20 we define the class `UserJets` to be derived from the class `basic_user`. The template parameter `hhc` specifies that we want to calculate something for processes in hadron-hadron collision. The remaining template parameters are the histogramming classes that we use.

6. In lines 23-24, we define a useful alias for histogram classes.

7. In lines 28-32, we define a constructors for `UserJets`. The first parameter is just the text name for the user class. The second is the jet definition for FASTJET. The third (optional) parameter is a time interval: DEDUCTOR should save its results every time this interval elapses. We have two versions of this type of constructors, the first can move the `fj::↩ JetDefinition` class the other copies it.

8. We need an init function, declared in line 35.

9. We need an analyzer function `userfunc(...)`, which we declare in line 38.

10. In line 41, we declare an instance `jetDef` of the FASTJET class `JetDefinition`. This was initialized by the constructors.

11. In lines 44 and 45, we declare two data members of our class, both vectors of `PseudoJet` objects. These help us to avoid even-by-event allocation and deallocation of vectors of `PseudoJet` objects.

Now we need an implementation file for `UserJets`.

Listing 6: The analyzer code of the class UserJets — Jets/analyzer-jets.cc

```
1  /*   local headers  */
2  #include "analyzer-jets.h"
3
4  void UserJets::initfunc()
5  {
6    /*   Createing histograms to calculate the one jet cross sections   */
7    unsigned int nbins = 48;
8    double pTmin = 250.0_GeV, pTmax = 5050.0_GeV;
9    auto bins = spacing<hist1d>::linear(nbins, pTmin, pTmax);
10
11   /*   cross sections at the hard level   */
12   Plain::phys(1, "One jet inclusive cross section at the hard interaction", bins);
13
14   /*   cross sections after shower    */
15   Ratio::phys(1, "One jet inclusive cross section LO pdf", bins);
16   Ratio::phys(2, "One jet inclusive cross section NLO pdf", bins);
17 }
18
```

```
void UserJets::userfunc(const shower_history<hhc>& history)
{
  constexpr double tonb = 389379.338;

  /*   the shower stage after the shower evolution   */
  for(auto it = history.cbegin(); it != history.cend(); ++it)
  {
    /*   showered and hard stages    */
    auto& showered = *it;
    auto& hard = it.hard();

    /*   showered and hard events   */
    auto& p = it->state;
    auto& q = hard.state;


    //   One jet inclusive cross section after showering
    //  ------------------------------------------------
    std::array<double, 3u> w1, w2;

    w1[2] = showered.weights.unitary.main*color_weight(showered)*tonb;
    w1[1] = w1[2]*showered.weights.threshold.main;
    w1[0] = w1[1]*showered.weights.threshold.delta;

    w2[2] = w1[2]*showered.weights.unitary.pdf_ratio;
    w2[1] = w1[1]*showered.weights.unitary.pdf_ratio;
    w2[0] = w1[0]*showered.weights.unitary.pdf_ratio;


    /*   Converting deductor event to FastJet format   */
    make_pseudojets(theparticles, p);

    /*    Run fastjet algorithm   */
    fj::ClusterSequence clustSeq(theparticles, jetDef);

    /*    Extract inclusive jets sorted by pT (above pTjetmin)   */
    double pTjetmin = 200.0_GeV;
    incJets = clustSeq.inclusive_jets(pTjetmin);

    /*    Fill histogram, only count jets that have |y| < 2.0   */
    for(unsigned j = 0; j < incJets.size(); j++)
      if(abs(incJets[j].pseudorapidity()) < 2.0) {
        Ratio::physfill(1, w1, dirac{}, incJets[j].perp());
        Ratio::physfill(2, w2, dirac{}, incJets[j].perp());
      }


    //   One jet inclusive cross section before showering
    //  -------------------------------------------------
    std::array<double, 2u> wh;

    wh[1] = hard.weights.unitary.main*color_weight(hard)*tonb;
    wh[0] = wh[1]*hard.weights.unitary.pdf_ratio;


    /*   Fill histogram, only count jets that have |y| < 2.0
     *   The two jets are back-to-back, since we have only two
     *   partons in the final state.
     */
    if(abs(q[1].momentum.prapidity()) < 2.0)
```

```
79        Plain::physfill(1, wh, dirac{}, q[1].momentum.perp());
80
81      if(abs(q[2].momentum.prapidity()) < 2.0)
82        Plain::physfill(1, wh, dirac{}, q[2].momentum.perp());
83    }
84  }
```

The initialization function for `UserJets`, beginning in line 4, is pretty simple.

1. We will create one-dimensional histograms for $d\sigma/dp_{\rm T}$ at the hard interaction level and after showering. We define the binning for these in lines 7-9.

2. In lines 12-16, we create the one dimensional histogram objects that we will need, with label `1` for $d\sigma/dp_{\rm T}$ at the hard interaction and the this is a `Plain` histogram. With labels `1-2` for $d\sigma/dp_{\rm T}$ after showering with various level of threshold effects included. These histograms has the type of `Ratio`. Histogram `1` is with LO PDFS at the hard interaction while `2` is with NLO PDF at the hard level.

The event analysis function for `UserJets`, beginning in line 24 is also pretty simple.

1. In line 19, we define `history` as the shower history argument of `userfunc(...)`.

2. In line 21, we define the conversion factor that we will need to convert GeV^{-2} to nb.

3. We want to analyze the partons in the final state of the shower. Our `shower_history<hhc>` object can, in general, contain more than one end stage. Thus, we need to iterate over the end stages that we have generated. In line 24, we define an iterator `it` and loop over its values.

4. In line 27, we define a reference to the shower stage that corresponds to the stage after the shower evolution. The corresponding event is defined in line 31 and and call it `p`.

5. We are also interested in information about the shower stage at the start of the shower. This information is contained in the object `hard = it.hard()` defined in line 28. In line 32, we define the partonic state at the start of the shower, `hard.state`, and call it `q`.

6. We need the weights associated with the state at the end of the shower. We need them with LO PDFs (`w1`) and with NLO PDFs (`w2`) ath the hard interaction. These variables are declared in line 37 and they are three element standard arrays.

7. In lines 39-41 we assemble the `w1` weight array. Its third component (`w[2]`) is the weight of the standard shower. The second component (`w[1]`) is the shower weight with the main threshold contributions. The first component (`w[0]`) is full shower weight with all the threshold contributions.

8. In lines 43-45 the weight `w2` is just like `w1` but we use the NLO PDFs at hard level. Thus we have to multiply `w1` by `showered.weights.unitary.pdf_ratio`.

9. Starting in line 49, we want to calculate $d\sigma/dp_T$ for jets after showering. First, we use `make_pseudojets(theparticles, p)` to convert the DEDUCTOR description of the partons at the end of the shower, contained in `p`, to a vector of FASTJET PseudoJets, `theparticles`. The function to do this is defined in `"fastjet.h"`

10. In line 52, we apply the FastJet analysis based on `jetDef` to the starting PseudoJets in `theparticles`. This gives a FastJet `ClusterSequence` object that we call `clustSeq`.

11. In lines 55 and 56, we define `incJets` to be the FastJet jets with transverse momentum above 200 GeV.

12. In lines 59-63, we loop over all the jets with absolute value of rapidity less than 2 and then fill our three histograms, using the three weights that we have calculated together with the P_{T} of the jet, `inclJets[j].perp()`.

13. Starting from line 66 we focus on the hard part of the event. In line 68 we declare a two element standard array to be the weight, `wh`. Its second component (`wh[1]`) defines the weight for the hard cross section with LO PDFs, while the first component (`wh[0]`) is the same weight but with NLO PDFs.

14. In lines 78-82 we fill a histogram with the transverse momentum distribution of the partons produced at the hard interaction. There are two such partons, `q[1]` and `q[2]` in the Deductor labelling scheme in which outgoing QCD particles are labelled 1,2, ... For each of these partons, we check if its rapidity is in the allowed range and then fill histogram 1 using the hard scattering weight `wh`.

When compile this and run it and then produce results with `deductor --result jets`, you will get a `result` directory in `jets.bundle`. There you will find a TEX file `summary.tex`, which, when you typeset it, will contain graphs of the jet cross sections that you asked for. You can edit the TEX input file. You will be happier with the output if you replace `axis` in this file by `semilogyaxis` everywhere.

5 Basic elements of the C++ library

In this document we don't do a full documentation of the Deductor library we just show the most important features those are important for the users.

5.1 Namespace

To avoid collision with other C++ libraries all the exported symbols are contained in a common namespace

```
namespace duct {}
```

There is another publicly available subnamespace that is used in the histogramming routines:

```
namespace duct {
  namespace distpoint {}
}
```

In the user code one might not want to use the `duct::` and `duct::distpoint::` prefixes. This can be avoided by the `using namespace duct;` and `using namespace duct::distpoint;` statements.

5.2 Collision types

DEDUCTOR is a general purpose parton shower algorithm. It can calculate cross sections for processes in e^+e^- annihilation, DIS and hadron-hadron collisions. It can also calculate heavy particle decays. The shower algorithms for different process types are built on the same principles, but they have some structural differences and their implementations can also differ. In DEDUCTOR, the process type is selected at compiler time, not at run time. This means that the definition of the C++ classes, functions and objects depend on the process type. Most of the classes has a template variable that refers to the process type. For this we have introduced simple tags as

```
struct epa {};     // for e+e-
struct dis {};     // for DIS
struct hhc {};     // for hadron-hadron collisions
struct decay {};   // for decay processes
```

With these, the type of the event record in hadron-hadron collision is `event<hhc>`.

5.3 Three-vector and four-vector

The most basic data types in the DEDUCTOR for three-vectors and four-vectors. Unfortunately these are not part of the standard C++ library, so every code has its own implementation.

Listing 7: Template class threevector — bits/hep-threevector.h

```
template<typename _Tp>
class threevector
{
public:
  //  types
  typedef _Tp value_type;

  //  default constructor creates a null vector
  threevector();

  //  constructor
  threevector(const value_type& x, const value_type& y, const value_type& z);

  //  elements access (with obvious meaning)
  const value_type& X() const;
  const value_type& Y() const;
  const value_type& Z() const;

  value_type& X();
  value_type& Y();
  value_type& Z();

  //  magnitude and the transverse momentum
  value_type mag () const;   //  returns the magnitude
  value_type perp() const;   // returns the transverse momentum
  value_type mag2() const;   // returns the square x^2 + y^2 + z^2
  value_type perp2() const;  // returns the transverse momentum square x^2 + y^2

  //  azimuth and polar angles
  value_type phi() const;      //  return the azimuthal angle
```

```
31    value_type theta() const;    //  return the polar angle
32  };
33
34  //  return the dot product of two vector, same as a*b = a⃗ · b⃗
35  template<typename _Tp>
36  _Tp dot(const threevector<_Tp>& a, const threevector<_Tp>& b);
37
38  //  return the cross product of two vectors, a⃗ × b⃗
39  template<typename _Tp> threevector<_Tp>
40  cross(const threevector<_Tp>& a, const threevector<_Tp>& b);
```

All the possible arithmetic operators (=,+,-,*,+=,-=,*=) with all the possible arguments are defined. We provide six specialization of `class threevector<_Tp>`. The type `_Tp` can be `float`, `double`, `long double`, `std::complex<float>`, `std::complex<double>`, and `std::complex<long double>`.

The four-vectors are represent by `template<typename _Tp> class lorentzvector`. This is publicly inherited from `class threevector`.

Listing 8: Template class lorentzvector — bits/hep-lorentzvector.h

```
1   template<typename _Tp>
2   class lorentzvector : public threevector<_Tp>
3   {
4   public:
5     //  types
6     typedef _Tp value_type;
7     typedef threevector<_Tp> threevector_type;
8
9     //  constructors
10    lorentzvector();  //   constructs null vector
11    lorentzvector(const _Tp& x, const _Tp& y, const _Tp& z, const _Tp& t);
12    lorentzvector(const threevector_type& v, const value_type& t);
13
14    //   elements access, gives the reference to the timelike component
15    const value_type& T() const;
16    value_type& T();
17
18    //  member functions
19    value_type plus () const;      //   returns t + z
20    value_type minus() const;      //   returns t − z
21    value_type rapidity() const;   //   returns rapidity
22    value_type prapidity() const;  //   returns pseudo-rapidity
23    value_type mag2() const;       //   returns m² = t² − x² − y² − z²
24    value_type mag() const;        //   returns invariant mass, −√|m²| if m² < 0
25
26    //  returns the transverse component that is perpendicular to both a and b
27    lorentzvector transverse(const lorentzvector& a, const lorentzvector& b) const;
28  };
```

All the possible arithmetic operators (=,+,-,*,+=,-=,*=) with all the possible arguments are defined and all the comparison operators (==,!=). We provide six specialization of `class lorentzvector<_Tp>`. The type `_Tp` can be `float`, `double`, `long double`, `std::complex<float>`, `std::complex<double>`, and `std::complex<long double>`.