Preprint typeset in MyNote style

**17 January 2014**

# Deductor version 1: 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.1.0.2.

Keywords: perturbative QCD, parton shower.

# Contents



# 1 Introduction

Welcome to the Monte Carlo event generator DEDUCTOR. 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. Then section 3 explains how to make your own kernel module. Section 4 explains the structure of a simple user module. Section 5 explains the structure of a more complex user module. Section 6 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 form DESY

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

or from the University of Orgeon

```
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 1.0.0. 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`:

```
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++11 and it requires a C++ compiler that supports this standard. It is important that the compiler has to provide full support for C++11 features. DEDUCTOR was tested with `g++-4.8.1` and `clang-3.3` on linux and MacOSX platforms. To configure and install, you should execute

```
mkdir bld-duct
cd bld-duct
cmake ../deductor-X.Y.Z/
make install
```

After this, you could remove `bld-duct`. 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 ../deductor-X.Y.Z/
```

```
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 the following message:

```
Usage: deductor --module my-module --cflags="opt1 opt2 ..."
                                   --cxxflags="opt1 opt2 ..."
                                   --ldflags="opt1 opt2 ..."
                                   source1 source2 ...
                                   -Lpath1 -llibrary1 -Lpath2 -llibrary2 ...

Options of the --module mode:
  my-module                        The name of the module. It will create the
                                   loadable module file my-module.so.
  --cflags="opt1 opt2 ..."         C compiler options
  --cxxflags="opt1 opt2 ..."       C++ compiler options
  --ldflags="opt1 opt2 ..."        linker options
  source1 source2 ...              C or C++ source files. Recognized extensions
                                   are .c, .cc, .c++, .cxx, .cpp, .C
  -Lpath                           library search path
  -llibrary                        link the liblibrary.so (e.g.: -lfastjet)
-------------------------------------------------------------------------------

Usage: deductor --kernel my-module

Options of --kernel mode:
  my-module              The module file contains the user defined functions.
-------------------------------------------------------------------------------

Usage: deductor --calculate my-module -n|--name name

Options of --calculate mode:
  my-module              The module file contains the user defined functions.
  -n, --name name        Name of the run.
-------------------------------------------------------------------------------

Usage: deductor --add my-module [-r dir] name1 [name2] [name3] ...

Options of --add mode:
  my-module            The module file contains the user defined functions.
  -r directory         Save the result to this directory (default: ./result)
-------------------------------------------------------------------------------
```

We will cover the `deductor --module`, `deductor --kernel`, `deductor --calculate`, and `deductor --add` commands below. Here we note only that each module created with `deductor --module` should be given a different name appropriate to its function.

To be able to do a real calculation, DEDUCTOR needs some auxilary 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 scttering 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 analyser routine. This is where the events are analysed and histograms are filled. We provide a powerful but still simple and flexible interface.

## 2.3 Getting Deductor ready

Now we need to create a kernel module. In `deductor-user-X.Y.Z`, execute

```
deductor --module kernelname kernel.cc
```

Here `kernelname` can be anything you want. This command creates some files in a "bundle" directory `kernelname.bundle` in `deductor-user-X.Y.Z`. On MacOSX in a Finder window it appears as a single file with a little Lego block icon. The essential function of `deductor --module` is to create a module, which is a dynamically loadable object file, `module.so`, that will be in `kernelname.bundle`.

Deductor needs some tables to operate. The tables are specific to the collider energy that you want to use and to some other choices, parton distribution functions (PDF) and $\alpha_s$. You can specify whatever sort of collider you want. The module that you just made from `kernel.cc` will create everything that you need for a 7 TeV LHC and an 8 TeV LHC and put it into a directory `.deductor` in your home directory. Execute

```
deductor --kernel kernelname
```

Now DEDUCTOR will work to build what it needs. Go get some lunch, followed by a cappuccino. In a couple of hours DEDUCTOR will be done. Fortunately, you don't need to do this again until you decide that you need a 14 TeV LHC.

## 2.4 Creating a user module

We can now look at the user rountines 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:

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

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

```
deductor --module mod-dy mod-dy.cc analyser-ZpT.cc
```

This command creates the needed files in `mod-dy.bundle` in `Drell-Yan`.

## 2.5 Calculating and analysing events

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

```
deductor --calculate mod-dy -n runname
```

Here `runname` identifies a particular run. You might like `run1`. Later, you might calculate some more with run name `run2`. The program starts like this:

```
+--------------------------------------------------------------------+
```

```
|                       Deductor release 1.0.0                               |
|                       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 JJJJ nnnn (2014) ppp [arXiv:xxxx.yyyy] 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.                                              |
+----------------------------------------------------------------------------+

Name of the run    : runname
User module        : mod-dy

Trying to find the main_calc() function : OK

Process has been created and started.
  Process Name  : Drell-Yan
  Process ID    : 1

    Analyser 'ZpT' of run 'runname' : 100000 events in 00:00:52
    Analyser 'ZpT' of run 'runname' : 200000 events in 00:01:47
```

DEDUCTOR keeps informing you that it has generated a certain number of events, in steps of 100000 in this case. This means that DEDUCTOR has put results for these events in a file in a subdirectory `output` in `mod-dy.bundle`. The results are there, but you don't want to read them in this form.

Instead, open another shell in the `Drell-Yan` directory and excecute

```
deductor --add mod-dy runname
```

This gives output that is something like

```
Trying to find the main_add() function in mod-dy : OK

Analyser 'ZpT' has collected 200000 MC events.
```

The `deductor --add` command creates a subdirectory `result` in the directory `Drell-Yan`. Here you will find files with results in tabular form, suitable for examining with, for example, GNUPLOT. For your convenience, DEDUCTOR also generates some draft LaTeX figures. You can find them in the `result` directory, too. You need the `pdflatex` and `pgfplots` packages to typeset them.

The `mod-dy.cc` code that you used specified that DEDUCTOR should work until you tell it to stop. You can just execute `deductor --add mod-dy runname` from time to time and, when you think you have enough events, stop your run with control-C.

It is possible to start the same calculation on many computers, for example on a cluster where each node shares the same filesystem. It important to give a unique name for every run when the calculations are started. The results can be combined by listing all run names when `deductor --add` is executed. Supposing that you have generated results from runs with names `run1, run2, run3`, the combined result can be obtained by

```
deductor --add mod-dy run1 run2 run3
```

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

# 3 Make your own kernel module

DEDUCTOR has very complicated splitting kernels. It is hard to define a good and efficient apporoximation for these functions that can be used in a veto algorithm efficiently. To achieve speed and efficiency DEDUCTOR has to explore the splitting kernels. This is done in the `deductor --kernel` stage. Fortunately we have to do this only once for every collider setup.

## 3.1 The simplest case

Let assume you want to create kernels for the LHC at 7 TeV and 8 TeV energies using the default PDF and $\alpha_s$. The code for this would be

Listing 1: Simple kernel example — kernel.cc

```
1  /*   Deductor headers  */
2  #include <deductor.h>
3
4  using namespace duct;
5
6  extern "C" {
7
8    std::function<void()> main_kern = [=]() -> void
9    {
10     /*  The kernel name is "LHC-7TeV" and it is defined with our standard PDFs.   */
11     deductor<hhc> kernel_7TeV("LHC-7TeV", deductor_attr<hhc>(7.0_TeV));
12
13     /*  The kernel name is "LHC-8TeV" and it is defined with our standard PDFs.   */
14     deductor<hhc> kernel_8TeV("LHC-8TeV", deductor_attr<hhc>(8.0_TeV));
15   };
16 }
```

This file is very simple but there are some rules:

1. In line 2 we include the main DEDUCTOR header file.
2. Every kernel module has to consist of only one function object called `main_kern`. Tthis object has to have C linkage.
3. The type of `main_kern` has to be an instance of the type `std::function<void()>`. Such a function object can be easily created by using C++11 lambda expression. (That is why you see the strange `[=]()-> void` syntax.)
4. We can create the desired kernels in the body of the lambda expression. The type of the DEDUCTOR kernel for hadron-hadron collisions is `duct::deductor<hhc>`. In lines 11 and 14 we create kernels for 7 TeV and 8 TeV, respectively.
5. Every kernel has a unique identifier, a name. This is the first argument of the constructor. This name will be the directory name in the `$HOME/.deductor/` folder where the run-time

data are saved. Please try to use something simple and straightforward because different systems handle filenames differently. We would discourage putting whitespace or special characters (like `!,*,~,^,?,/,#,$,|`) into this name.

6. Note that `kernel_7TeV` and `kernel_8TeV` are temporary names local to the lambda expression.

7. The second argument of the constructor is an instance of type `deductor_attr<hhc>`. It is a very simple class with few simple public members. Here we would like to highlight only one of its constructors and one if its public members.

```
1  struct deductor_attr<hhc>
2  {
3    /*   constructor   */
4    explicit deductor_attr(double ecm = 14.0_TeV,
5                           double (*as)(double) = __herapdflo_as,
6                           void (*pdf)(double, double *) = __herapdflo);
7
8    /*   model and pdf input  */
9    std::map<int, void (*)(double, double *)> pdf_at_1GeV;
10 };
```

Most users will want to do calculations for the LHC or for the Tevatron. In these cases the attribute class can be easily created by the constructor defined in line 4. This constructor has three parameters:

1. The collider energy `ecm` in GeV units. The default argument is `14_TeV=14000`.
2. The strong coupling $\alpha_s/(2\pi)$, which is provided as a pointer to the corresponding function. This function has only one argument, the renromalization scale in $\mathrm{GeV}^2$. Our default $\alpha_s$ is a 1-loop running coupling with $\alpha_s(M_Z^2) = 0.118$. This running $\alpha_s$ is given by the function `__herapdflo_as` and it is declared in the header file `bits/mc-default-pdf.h`.
3. The parton distribution functions (PDF). This is provided as a pointer to the PDF at a starting scale of 1 GeV. The evolution of the parton distribution functions is done internally. The default PDF is a custom LO HERAPDF fit, which is given by the function `__herapdflo`. It is declared in `bits/mc-default-pdf.h`.

In Listing 1 we simply changed the collider energy and used the default $\alpha_s$ and PDFs.

### 3.2 More advanced cases

The class `deductor_attr<hhc>` has a public member `pdf_at_1GeV`. This is for providing an alternative PDF for the proton or for creating PDF inputs for other kinds of hadrons. With the constructor we can provide PDF input for the proton. This is adapted for the anti-proton and neutron. If the colliding objects are photons or pions, then we should be able to provide the PDFs. For example for photon with default $\alpha_s$ it can be done by

```
duct::deductor_attr<hhc> attr(7.0_TeV);
attr.pdf_at_1GeV[22] = my_photon_pdf;
```

Note we refer to the photon by its PDG number, which is 22. In DEDUCTOR we use the PDG scheme for labeling the particles. The gluon has two labels, the PDG one, `gluon=21`, and the more convenient choice `gluon=0`.

It is useful to show a complete, realistic non-trivial kernel module with user defined $\alpha_s$ and PDFs.

Kernel module with user defined $\alpha_s$ and PDFs

```
#include <deductor.h>

static double __herapdf_func(double *p, double x) {
  return p[0]*std::pow(x, p[1])*std::pow(1.0-x, p[2])*(1.0+p[3]*x+p[4]*x*x);
}

/*   User defined PDF  */
void my_beautiful_pdf(double x, double *f)
{
  constexpr double p[7][5] = {
    {2.9784,    0.6278,    4.3263,    0.0000,   12.7383},
    {1.8555,    0.6584,    3.2857,    0.0000,    0.0000},
    {0.0872,   -0.1920,    2.6728,    0.0000,    0.0000},
    {0.0872,   -0.1920,    4.8723,    0.0000,    0.0000},
    {3.0061,    0.1195,    4.3692,   -0.4241,    0.0000},
    {0.1435,   -0.1920,   17.3391,    0.0000,    0.0000}
  };

  f[-3] = f[3] = __herapdf_func(p[5], x);          //  s and s̄ quarks
  f[-2] = __herapdf_func(p[2], x);                 //  ū qaurk
  f[-1] = __herapdf_func(p[3], x);                 //  d̄ quark
  f[ 0] = __herapdf_func(p[4], x);                 //  gluon
  f[ 1] = __herapdf_func(p[1], x) + f[-1];         //  d quark
  f[ 2] = __herapdf_func(p[0], x) + f[-2];         //  u quark
  f[4]=f[-4]=f[5]=f[-5]=0.0;                       //  c and b quarks
}

/*  User defined alpha_s   */
double my_beautiful_as(double q2)
{
  constexpr double thr[7] = {0.0, 0.0, 0.0, 0.0, 1.69, 20.25, 30625};
  constexpr double lmd[7]  = {0.0304811, 0.0304811, 0.0304811, 0.0304811, 0.021846, 0.0↩
      119477, 0.00293037};

  unsigned nf = 6;
  while(q2 < thr[nf] && nf > 3) --nf;

  //  computing the LO alpha_s
  double b0 = beta0(nf);
  double t = 0.5*b0*log(q2/lmd[nf]);

  return 1.0/t;
}

using namespace duct;

extern "C" {

  std::function<void()> main_kern = [=]() -> void
```

```
  {
    /*  The kernel "My-Beautiful-LHC-7TeV" is defined with new alpha-s and PDFs.  */
    deductor_attr<hhc> attr7(7.0_TeV, my_beautiful_as, my_beautiful_pdf);
    deductor<hhc> kernel_7TeV("My-Beautiful-LHC-7TeV", attr7);

    /*  The kernel "My-Beautiful-LHC-8TeV" is defined with new alpha-s and PDFs. */
    deductor_attr<hhc> attr8(8.0_TeV, my_beautiful_as, my_beautiful_pdf);
    duct::deductor<hhc> kernel_8TeV("My-Beautiful-LHC-8TeV", attr8);
  };
}
```

### 3.3 Kernel module for a multi-core computer

If you have a powerful computer with many CPU cores you can speed up the the scanning process. It is possible to create the kernels simultaneous by using the C++11 standard thread library. With this, the simple kernel module would look

```
/*   Deductor headers  */
#include <deductor.h>

using namespace duct;

extern "C" {

  std::function<void()> main_kern = [=]() -> void
  {
    /*  The kernel name is "LHC-7TeV" and it is defined with our standard PDFs.   */
    std::thread th7TeV([=]() -> void {
                    deductor<hhc> kernel_7TeV("LHC-7TeV", deductor_attr<hhc>(7.0_TeV));
                  });

    /*  The kernel name is "LHC-8TeV" and it is defined with our standard PDFs.   */
    std::thread th8TeV([=]() -> void {
                    deductor<hhc> kernel_8TeV("LHC-8TeV", deductor_attr<hhc>(8.0_TeV));
                  });

    th7TeV.join();
    th8TeV.join();
  };
}
```

With the code in Listing 1, DEDUCTOR will already construct the 7 TeV and 8 TeV kernels using multiple threads. The modification listed above causes DEDUCTOR to create the 7 TeV and 8 TeV kernels at the same time.

## 4 Make your own user module

The usual event generators (*e.g.* PYTHIA) provide a routine that can be called in a loop and for every invocation of this function it returns with an event. It is the user's responsibility to manage this event loop. DEDUCTOR is organized in a event driven way. A stream of events is generated and the analyser routines can be attached and detached 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 analyser part. In the main part, we set up the kernel, start the processes and make the connection to the analysers. In the analyser part we deal with the events. For example the event analyser routines set up and fill histograms representing data from the generated events.

## 4.1 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 analysers 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 like something like this:

Listing 2: User module for Drell-Yan process — Drell-Yan/mod-dy.cc

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

/*   Analyser headers  */
#include "analyser-ZpT.h"

extern "C" {

  std::function<void(const char *, const char *)>
  main_calc = [=](const char *dir, const char *rname) -> void
  {
    /*  Using an already created kernel. */
    duct::deductor<hhc> kernel("LHC-8TeV");

    /*   Some parameters of the shower   */
    kernel.kT_cutoff(1.0_GeV);
    kernel.max_color_suppression(0u);

    /*   Drell-Yan process    */
    {
      /*   Hard process  */
      auto hard = [=](const model *mdl, double Ecm) -> duct::dyjets {
        int proton = 2212, electron = 11;
        double mL = 400.0_GeV, mH = 5.0_TeV;
        return duct::dyjets(mdl, Ecm, mL, mH, proton, proton, {electron});
      };

      /*   Creating the analyser object.   */
      UserZpT *ZpT = new UserZpT;
      unsigned long nstore = 100000UL;
      ZpT->param(dir, rname, "ZpT", nstore);

      /*  Create the shower process and start the calculation.   */
      unsigned pid = kernel("Drell-Yan", hard, ZpT);

      /*  Here we can amuse ourself with fancy messages like these...  */
      std::cout<<"Process "<<kernel.name(pid)<<" has been started."<<std::endl;
    }
  };

  /*   The list of all analysers in the main_calc function.   */
```

```
43    duct::main_add_list main_add = {
44      duct::to_main_add<UserZpT>("ZpT")
45    };
46  }
```

This needs discussion.

1. We start again with includes in lines 2, 3 and 6. 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 but one could also use an external library. In line 6 we include the header of the analyser routine. In this example we have only one, which calculates the $p_T$ distribution Drell-Yan lepton pair. The analysers are user defined. We will discuss this in the next subsection.

2. Every user module has to contain a function object called `main_calc`. Its type has to be `std::function<void(const char *, const char *)>`. It is created as a C++11 lambda expression. The objects that we need are created in the body of the lambda expression. There are two `const char *` arguments of this function. Their values are provided by the `deductor --calculate` program. The argument `const char *dir` is the name of the directory where the backup files are saved during the calculation and the argument `const char *rname` is the name of the run. This is the only information that `deductor --calculate` can provide.

3. In line 14, we create an instance of the previously defined "LHC-8TeV" kernel. This constructor simply reads the previously saved kernel from the hard disk. If the kernel doesn't exist then the constructor will give an error message and the program terminates.

4. The shower needs a cutoff parameter. The DEDUCTOR shower doesn't allow splitting with smaller transverse momentum than 1 GeV. It cannot go lower because the PDF functions are not available under this scale. Thus, the $k_\perp > 1$ GeV is hardwired in the kernel. But one might want to use a higher transverse momentum cutoff in the calculation. This can be done by defining a $k_\perp$ veto for the current kernel. In line 17, this is defined as 1 GeV (which is the hard wired cutoff, so that line 17 doen't do anything). One can set this to 2 GeV or some other value. If the the transverse momentum cutoff is set to smaller than 1 GeV, then it is completely ignored. It is also possible to impose a virtuality cutoff by `kernel.virt_cutoff(3.0_GeV)`.

5. The kernel needs a maximum value of the color suppression index that can be generated in the shower. Here we want the leading color approximation, so in line 18 we set the maximum value of the color suppression index to zero.

6. Now that the kernel is set up and configured, we have to create a process. First we need the hard process. One would expect that can be done by creating an object that represents the hard process and can generate hard events. In DEDUCTOR we have a slightly different approach. Since the kernel is defined by the collider energy, $\alpha_s$, PDFs and underlying physics model (represented by a list of the available particles), the hard process has to be matched to the kernel parameters and properties. The hard process has be able to generate events with $\alpha_s$, PDFs or particle masses of the kernel.

   In lines 23-27 we deal with the hard process. Instead of creating a hard process object, we define a *hard process factory*. We create a functor that can create the desired hard process. This functor has to have two parameters, a constant pointer to the model and the collider energy. Thus this lambda expression is always something like this

```
auto hard = [=](const model *mdl, double Ecm) -> my_beautiful_hard_process_type {
  //  create the object of your hard process
  my_beautiful_hard_process_type my_precious{...};
  return my_precious;
}
```

In our example we want to create a Drell-Yan process. In line 26 we crate a `dyjets` object using the model descriptor `mdl` and collider energy `Ecm` parameters provided by the kernel. We want to generate events in the region where the invariant mass of the lepton pair is between 400 GeV and 5 TeV (`double mL = 400.0_GeV, mH = 5.0_TeV;`). The colliding hadrons are protons (PDG key 2212). The last argument of the constructor is a list of the leptons that we want to generate in the final state. In this example, we want to generate events with electron-positron pairs only. Thus this list consists of only one element, the electron (PDG key equal to 11).

7. We have to be able to analyse the events and for that we need analysers. We will discuss in detail how to define analysers in the next subsection. Every analyser has to be a class derived for a certain abstract class. In line 30, we create a analyser object that can produce the transverse momentum distribution of the $Z/\gamma$ bosons. There are some strict rules for creating the analyser objects:

   - It is very important that every analyser has to be dynamically allocated.
   - Always use the `new` operator, never use `malloc` or other allocator for this purpose!
   - Never allocate an analyser in an array, thus something like `ZpT = new UserZpT[3];` is forbidden.
   - Never deallocate the analysers! The statement `delete ZpT;` mustn't appear in your code.

   In every C++ book it is said that all memory allocated by the `new` operator has to be deallocated. That is true but here it is forbidden to deallocate the memory at the end of the block or at the end of the lambda expression. The analyser object will be owned by the kernel and when the kernel gets destroyed the analyser will be destroyed too.

8. Our analyser is inherited form class `basic_user` or `basic_user_set` and that needs some setup. In line 32 we deal with this. The anlyser wants to know the backup directory `dir` and the name `rname` of the run to be able to save the results periodically. This information will be supplied when the analyser is called. The third argument of function is the name of the analyser, (`"ZpT"`). This has to be unique for every instance of any analyser type. The fourth argument is a number of events, here `nstore = 100000`. After each `nstore` events, the analyser will store the results.

9. By default, the analyser object will never tell DEDUCTOR to stop generating events. If you want just `nstop` events, you can use an optional fifth argument to the `param` function: `ZpT->param(dir, rname, "ZpT", nstore, nstop);`.

10. Now we have everything to start the calculation. It can be done by the `operator()(...)` function of the kernel. In line 33 we start the calculation. The first argument is a name for the process, in this case `"Drell-Yan"`. (Currently, DEDUCTOR does not use the process name.) The second argument is the hard process factory (`hard`) and the last argument is the pointer to the analyser (`ZpT`). The calculation immediately starts and `operator()(...)` returns an the process ID (`pid`).

11. Finally, in line 43, we supply a list of the analysers that our `deductor --add` should use. Here there is only one element in the list. We give its type, `UserZpT`, and its name `"ZpT"`. Later in this manual, we will see a user module with more analysers.

## 4.2 Analyser part of the Drell-Yan user module

In the code for the Drell-Yan module in Listing 2, we created an instance of the analyser class `UserZpT`. Now we need to define this class. The analyser part of the user module deals with events. Every analyser has to derived from the abstract class `user<hhc>`. It has four virtual functions. This class doesn't do anything but provide the interface to the shower process. In this section we don't want to discuss it in detail. Instead we use the derived class `basic_user<hhc,...>` which is a higher level interface. It provides powerful histograming and it is supported by the command line program `deductor --add`. In order to derive the analyser 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/$\gamma$-bosons. We can call the analyser `UserZpT`. Now, the header file of this is

Listing 3: Definition of the class UserZpT — Drell-Yan/analyser-ZpT.h

```
1  #ifndef __analyser_zpt_h__
2  #define __analyser_zpt_h__
3
4  /*   deductor main user headers  */
5  #include <user.h>
6
7  /*   It is not a library just a module. We can use using namespace...  */
8  using namespace duct;
9  using namespace duct::distpoint;
10
11 /*  This is the declaration of the analyser class.   */
12 struct UserZpT : public basic_user<hhc, double, hist1d>
13 {
14   /*   The init function is called once at the beginning of the calculation.   */
15   void initfunc();
16
17   /*   The analyser routine. It analyses the events and fill the histograms.   */
18   void userfunc(const shower_history<hhc>&);
19 };
20 #endif
```

This is a very simple analyser 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 20 in the usual way.

2. The main deductor user header is `user.h`. It brings all the necessary declarations and definitions that are needed to define the analyser based on the `basic_user` class.

3. The class `UserZpT` is derived from class `basic_user<hhc, double, hist1d>`. What are the template arguments? The `basic_user<...>` is a rather complicated and abstract structure (have a look in file bits/mc-basic-user.h) and we will discuss it in detail later. The first template

parameter (`hhc`) specifies that we want to calculate something for processes in hadron-hadron collision. The second parameter tells that the sample type in the histogramming is a `double`. The last argument (`hist1d`) specifies that we are doing simple 1-dimensional binning. The class `hist1d` is contained in the namespace `distpoint`.

4. As we mentioned earlier, we have to overload two virtual functions of the class `basic_user`. They are declared in line 15 and 18. We will define them in the implementation file (`Drell-Yan/analyser-ZpT.cc`).

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

Listing 4: The analyser code of the class UserZpT — Drell-Yan/analyser-ZpT.cc

```
1  /*   local headers  */
2  #include "analyser-ZpT.h"
3
4  void UserZpT::initfunc()
5  {
6    /*   Create a histogram to calculate the pT distribution.   */
7    auto bins = spacing<hist1d>::linear(100, 0.0_GeV, 100.0_GeV);
8    phys(1, "$Z/\\gamma$ $p_T$ distribution", bins, true);
9  }
10
11 void UserZpT::userfunc(const shower_history<hhc>& s)
12 {
13   /*   The shower stage after the shower evolution.   */
14   const shower_stage<hhc>& sb = s.back();
15   const event<hhc>& p = sb.state;
16
17   /*   Momentum of the ee pair.   */
18   auto pee = p[-2].momentum + p[-3].momentum;
19
20   if(pee.mag() > 400.0_GeV && std::abs(pee.rapidity()) < 2.0) {
21     /*   Calculate the weight of the event.   */
22     constexpr double tonb = 389379.338;     //   1/GeV^2 ==> nanobarn
23     double w = sb.weight*color_weight(sb)*tonb;
24
25     /*   Fill the historgam.   */
26     if(pee.perp() < 100.0) {
27       physfill(1, w, dirac(), pee.perp());
28       normfill(1, w);
29     }
30   }
31 }
```

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

1. We are going to create one histogram in one dimension. In line 7 we define the bins. We choose 100 equally spaced bins from $p_{\mathrm{T}} = 0$ GeV to $p_{\mathrm{T}} = 100$ GeV. There are more ways to create bins, which we will describe in later sections.

2. In line 8, 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. In the user function, we will use this ID to refer to our histograms. The second is a name for our histogram. 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`. This indicates that we want a normalized histogram. The normalization will be defined in the user function that comes next.

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 an object of type `shower_history<hhc>`, which contains the entire history of the event. We call the shower history `s`.

2. In line 14, we define `sb` to be the record of the final stage of the shower. The shower history essentially is a list of the shower stages at each step in shower evolution. The front element of this list is the hard event and the back element is the event after the shower evolution.

3. In line 15, we define `p` to be a sequential container of variables describing each of the partons at shower stage `sb`, namely at the end of the shower. The `event<hhc>` has special labeling convention. Outgoing QCD partons are labeled by `1`,`2`,`3`,..., the incoming partons are labeled by `-1`,`0` and the outgoing non-QCD particles are labeled by `-2`,`-3`,`-4`,.... Thus `p[n]` describes the variables for the parton numbered `n`. The momentum of this parton is `p[n].momentum`, which is a Lorentz vector of `double` components.

4. In line 18, we define the momentum of the lepton pair. With the DEDUCTOR labelling convention, this is `pee = p[-2].momentum + p[-3].momentum`.

5. We want to examine events with the lepton pair mass greater than 400 GeV and rapidity between $-2$ and 2. The `if` statement in line 20 takes care of this.

6. In line 23, we calculate the weight of the event. The variable `sb.weight` contains all factors for the weight except the color weight, which is calculated by `color_weight(sb)`. The weight has units of cross section, which in DEDUCTOR is $\mathrm{GeV}^{-2}$. For our histogram, we would like to use nanobarns, so we multiply by the proper conversion factor.

7. Now we need to fill our histogram for $p_\perp(e^+e^-) =$ `pee.perp()`. The `if` statement in line 26 selects the right events to go into the histogram.

8. In line 27, we enter this 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` is entered into this bin.

9. When we defined the histogram, we said that we wanted to normalize it. That means that when `deductor --add` prepares the final results, it will divide the cross section in each bin that was accumulated by `physfill(...)` by an accumulated normalization factor. The instruction `normfill(1, w)` in line 28 collects for histogram `1` the normalization factor that we want. In this case, we just want the total cross section that has contributed to the histogram.

When you run this and then run `deductor --add`, you will get output that starts something like this:

```
#  This result was calculated by Deductor-1.0.0.
#  The number of collected MC events is 500000.
#  It was created on Friday, 2014 January 10 at 22:00:55 CET.

#  $Z/\gamma$ $p_T$ distribution
 0.0000e+00  5.0000e-01  1.0000e+00  1.138649e-02  3.563727e-04  9.236616e-07  2.890863e-08
 1.0000e+00  1.5000e+00  2.0000e+00  2.764588e-02  5.608064e-04  2.242609e-06  4.549211e-08
 2.0000e+00  2.5000e+00  3.0000e+00  3.433764e-02  6.283939e-04  2.785439e-06  5.097475e-08
 3.0000e+00  3.5000e+00  4.0000e+00  3.801704e-02  6.596057e-04  3.083908e-06  5.350663e-08
 4.0000e+00  4.5000e+00  5.0000e+00  3.783220e-02  6.535876e-04  3.068915e-06  5.301844e-08
 ...
```

The first three columns give the bins: left edge, center, right edge. Column 4 is the cross section in the corresponding bin, normalized by the chosen normalization factor. With the normalization that we used, we have

$$\sum_{n=1}^{100} C_4(n) \times \Delta p_\perp(n) = 1 \,, \tag{4.1}$$

where $\Delta p_\perp(n) = 1$ GeV is the bin size. Column 5 is the statistical error on column 4, ignoring the (normally very small) stastical error on the normalization factor. Column 6 is the cross section in the corresponding bin, before we normalized it. With the user function that we used, its units are nb/GeV. Column 7 is the statistical error on column 6.

# 5 Make another user module

Let's try another user module. This one will be a little more complicated. We will make a user module that can perform some jet calculations. There will be several analysers and we will look at one of them.

## 5.1 Main part of the jets user module

In the main part of the user module we have to set up the kernel, the hard process, and the analysers 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 like something like this:

Listing 5: User module for jet studies — Jets/mod-jets.cc

```
1  /*   Deductor headers  */
2  #include <deductor.h>
3  #include <proc-hhcjets-lc.h>
4
5  /*   Analyser headers  */
6  #include "analyser-jets.h"
7  #include "analyser-xperp.h"
8  #include "analyser-angle.h"
9  #include "analyser-Ninjet.h"
10 #include "analyser-Zinjet.h"
11
12
13 /*   deductor and standard namespaces   */
```

```
using namespace duct;
using namespace std;

extern "C" {

  std::function<void(const char *, const char *)>
  main_calc = [=](const char *dir, const char *rname) -> void
  {
    /*  Using a previously creating the kernel   */
    deductor<hhc> kernel("LHC-8TeV");

    /*   Set parameters for the kernel.  */
    kernel.kT_cutoff(1.0_GeV);
    kernel.max_color_suppression(0u);

    /*   Jet production    */
    {
      /*   Create the hard process. */
      auto hard = [=](const model *mdl, double Ecm) -> hhcjets_lc
      {
        int proton = 2212;
        auto pTlist = {100.0, 300.0, 400.0, 600.0};

        // We use a list of typical PT values that we want.
        return hhcjets_lc(mdl, Ecm, pTlist, proton, proton);
      };

      /*   Create the analyser objects. */
      unsigned long nstore = 100000UL, nstop = 0UL;
      double R = 0.4;
      fjcore::JetAlgorithm jetalg = kt_algorithm;

      /*   Jet cross section pT distribution */
      UserJets *Jets = new UserJets(jetalg, R);
      Jets->param(dir, rname, "Jets", nstore, nstop);

      /*   Azimuthal decorelation   */
      UserAngle *Angle = new UserAngle(jetalg, R);
      Angle->param(dir, rname, "Angle", nstore, nstop);

      /*   Number of partons in a jet  */
      UserNinjet *Ninjet = new UserNinjet(jetalg, R);
      Ninjet->param(dir, rname, "Ninjet", nstore, nstop);

      /*   Momentum fraction distribution of the partons in a jet  */
      UserZinjet *Zinjet = new UserZinjet(jetalg, R);
      Zinjet->param(dir, rname, "Zinjet", nstore, nstop);

      /*   x_perp distribution   */
      UserXperp *Xperp = new UserXperp(jetalg, R = 0.1);
      Xperp->param(dir, rname, "Xperp", nstore, nstop);


      /* Creates the shower process and start the calculation. */
      unsigned pid = kernel("jets", hard, {Jets, Xperp, Angle, Ninjet, Zinjet});

      /*  Here we can amuse ourself with fancy messages like these...  */
      std::cout<<"Process \'"<<kernel.name(pid)
      <<"\' has been created and started with process id, pid = "<<pid<<".\n";
    }
```

```
74    };
75
76
77    /*   List of all analysers.*/
78    main_add_list main_add = {
79      to_main_add<UserJets>("Jets"),
80      to_main_add<UserAngle>("Angle"),
81      to_main_add<UserNinjet>("Ninjet"),
82      to_main_add<UserZinjet>("Zinjet"),
83      to_main_add<UserXperp>("Xperp")
84    };
85  }      //  extern "C"
```

This needs discussion.

1. We begin in lines 2 and 3 with DEDUCTOR headers. The `proc-hhcjets-lc.h` header is for the hard process generator for $2 \to 2$ QCD scattering in hadron-hadron collisions using the leading color approximation. Also available is the same thing with full color, with header `proc-hhcjets.h`.

2. In lines 6 - 10, we include headers for five analyser routines.

3. In lines 14 - 16, we have `using namespace` declarations that make the following code a little easier to read.

4. In lines 19 and 20, we begin the definition of the `maincalc` function as in Listing 2. In the following lines, the body of the function defines what `maincalc` should do.

5. In line 23, we create an instance of the previously defined "LHC-8TeV" kernel, just as in Listing 2.

6. In line 26, we define the $k_\perp$ cutoff for the shower to be 1 GeV, the smallest possible value.

7. In line 27, we set the maximum allowed value of the color suppression index to zero, thus using the leading color approximation.

8. In lines 32 - 39, we set up the hard process generator using `hhcjets_lc`, which generates $2 \to 2$ QCD scattering in the leading color approximation. The `hhcjets_lc` generator needs some arguments that we supply here.

9. In line 34, we define the arguments that tell the identity of the colliding hadrons.

10. In line 35, we provide a list of transverse momenta. These tell the generator to generate hard scattering events with $p_\perp > 100$ GeV, with equal numbers of events with $p_\perp$ between 100 GeV and 300 GeV, between 300 GeV and 400 GeV, between 400 and 600 GeV, and above 600 GeV. You can choose what you want, but if you just set `auto pTlist = 100.0`, you will get very few events with $p_\perp > 600$ GeV.

11. Next, we need to declare some analyser objects. In line 42, we define `nstore = 100000UL`, which will tell DEDUCTOR to store results every 100000 events. We also define `nstop` to tell DEDUCTOR when to stop. A value `137000UL` would tell DEDUCTOR to stop after 137000 events. The value `0UL` tells DEDUCTOR never to stop. This is an optional parameter for `param(...)` below and `0UL` is the default value, so we could have left `nstore` out of the code.

12. If we give non-zero values of `nstop` for the various analyser objects, here is what will happen. The main DEDUCTOR program is like a factory that produces events. It has several event producing machines (treads) producing events at once. Newly produced events go to the Order Fulfillment Department, which sends accumulated events to the customers, the analysers. When a given customer reaches `nstop` events, it produces its final report and tells the Order Fulfillment Department that it does not want any more events. The event factory continues to produce events as long as there are customers, but after a certain amount of time with no customers it will reluctantly turn itself off.

13. Our analysers will use FASTJET to find jets. In particular, we use the `fjcore` version. For this reason, in lines 43 and 44 we define a jet radius parameter `double R = 0.4` and a jet algorithm to use, `fjcore::JetAlgorithm jetalg = kt_algorithm`.

14. In line 47, we create an instance of our first analyser class, `UserJets`. The constructor takes two arguments, `jetalg` and `R`.

15. In line 48, we use the `param(...)` function to set the parameters `nstore` and `nstop` for `UserJets` and to give it a name, `"Jets"`.

16. The other analysers are declared similarly, except for `UserXperp`, for which we redefine `R` to `0.1` before passing it as an argument for the constructor.

17. In line 68, we start the calculation by calling the kernel with a name, the hard process generator `hard`, and a list of the analysers.

18. In lines 78 - 83, we supply the list of the analysers that `deductor --add` should use. For each analyser, we need to supply its type (*e.g.* `UserJets`) and its name (*e.g.* `"Jets"`) with the syntax shown.

## 5.2 Analyser part of the jets user module

In the code for the jets module in Listing 5, we created an instances of the several analyser classes. Lets now look at how we define one of these, `UserJets`. The header file of this is

Listing 6: Definition of the class UserJets — Jets/analyser-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 /*   Using namespace...  */
11 using namespace std;
12 using namespace duct;
13 using namespace duct::distpoint;
14 using namespace fjcore;
15
16 /*   Narrowing the basic_user template   */
17 using UserPlain = basic_user<hhc, double, hist1d>;
```

```
18  using UserVoid = basic_user<hhc, double, void>;
19
20  /*
21   *   This is the declaration of the analyser class.
22   */
23  class UserJets : public basic_user_set<UserVoid, UserPlain>
24  {
25  public:
26    UserJets(JetAlgorithm jet_algorithm=kt_algorithm, double Rparam=0.4,
27             RecombinationScheme recomb_scheme=E_scheme, Strategy strategy=Best)
28    : jetDef(jet_algorithm, Rparam, recomb_scheme, strategy) {}
29
30    /*   User defined init function. */
31    void initfunc();
32
33    /*   The analyser routine.  */
34    void userfunc(const shower_history<hhc>&);
35
36    /*  For fastjet analysis - select algorithm and parameters.
37     *  The UserJets constructor above initializes jetDef.
38     */
39    JetDefinition jetDef;
40
41    /*  Predefined vectors of Pseudojets  */
42    std::vector<PseudoJet> theparticles;
43    std::vector<PseudoJet> incJets;
44  };
45
46  #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`. 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 stage record to a vector of FASTJET PseudoJets.

3. In lines 17 and 18, we define aliases for two `basic_user` class definitions. `UserPlain` defines one-dimensional histograms and `UserVoid` defines zero-dimensional histograms (*i.e.* cross sections).

4. In line 23 we define the class `UserJets` to be derived from the class `basic_user_set`, which in turn is based on `UserPlain` and `UserVoid`.

5. In lines 26 and 27, we define the constructor for `UserJets` with four parameters. The parameters are related to FASTJET. All of the parameters are optional, with default values as shown. When we created an instance of `UserJets` in Listing 5, we in fact used the default values of all four parameters. However, it would have been easy to switch jet algorithms or $R$.

6. In line 28, we initialize an instance `jetDef` of the FASTJET class `JetDefinition` with the given FASTJET parameters.

7. We need an init function, declared in line 31.

8. We need an analyser function `userfunc(...)`, which we declare in line 34.

9. In line 39, we declare an instance `jetDef` of the FastJet class `JetDefinition`. This was initialized in line 28.

10. In lines 42 and 43, 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 7: The analyser code of the class UserJets — Jets/analyser-jets.cc

```
1  /*   local headers  */
2  #include "analyser-jets.h"
3
4  void UserJets::initfunc()
5  {
6    /*   Create an object to calculate the total cross section   */
7    UserVoid::phys(1,"Total cross section");
8    UserVoid::phys(2,"Total cross section (hard)");
9
10   /*   Create a histogram to calculate the one jet cross section   */
11   unsigned int nbins = 60;
12   double pTmin = 200.0_GeV, pTmax = 800.0_GeV;
13   auto bins = spacing<hist1d>::linear(nbins, pTmin, pTmax);
14
15   UserPlain::phys(1, "One jet inclusive cross section", bins);
16   UserPlain::phys(2, "One jet inclusive cross section at hard interaction",bins);
17
18   cout << "UserJets is using jet algorithm " << jetDef.jet_algorithm()
19        << " with R = " << jetDef.R() << endl;
20 }
21
22 void UserJets::userfunc(const shower_history<hhc>& s)
23 {
24   /*   the shower stage before the shower evolution */
25   auto& sf = s.front();
26   auto& pf = sf.state;
27
28   /*   the shower stage after the shower evolution   */
29   auto& sb = s.back();
30   auto& p = sb.state;
31
32   /*   Calculate the weight of the event   */
33   constexpr double tonb = 389379.338;
34   double wf = sf.weight*color_weight(sf)*tonb;   //  before shower
35   double w = sb.weight*color_weight(sb)*tonb;    //  after shower
36
37   /*   Total cross sections   */
38   UserVoid::physfill(1, w);
39   UserVoid::physfill(2, wf);
40
41   /*   Converting deductor event to a FastJet format   */
42   make_pseudojets(theparticles, p);
43
44   /*   Run fastjet algorithm   */
45   ClusterSequence clustSeq(theparticles, jetDef);
```

```
46
47    /*   Extract inclusive jets sorted by pT (above pTjetmin = 200GeV)   */
48    incJets = clustSeq.inclusive_jets(200.0_GeV);
49
50    /*   Fill histogram,  only count jets that have |y| < 2.0  */
51    for(unsigned j = 0; j < incJets.size(); j++)
52      if(abs(incJets[j].rap()) < 2.0)
53        UserPlain::physfill(1, w, dirac(),incJets[j].perp());
54
55    /*   One jet inclusive cross section before showering   */
56    if(abs(pf[1].momentum.rapidity()) < 2.0)
57      UserPlain::physfill(2, wf, dirac(), pf[1].momentum.perp());
58
59    if(abs(pf[2].momentum.rapidity()) < 2.0)
60      UserPlain::physfill(2, wf, dirac(), pf[2].momentum.perp());
61  }
```

The initialization function for `UserJets`, beginning in line 6 is pretty simple.

1. In line 7, we create a zero dimensional histogram for the total cross section within the cuts. We use `UserVoid` from `"analyser-jets.h"`. This historgram has an appropriate label and the index `1`.

2. In line 8, we create a zero dimensional histogram with label `2` for the total cross section at the level of the hard interaction.

3. We will create one-dimensional histograms for $d\sigma/dp_{\mathrm{T}}$ at the hard interaction level and after showering. We define the binning for these in lines 13-15.

4. In lines 15 and 16, we create the histogram objects that we will need, with labels `2` for $d\sigma/dp_{\mathrm{T}}$ at the hard interaction and `1` for $d\sigma/dp_{\mathrm{T}}$ after showering.

5. In lines 18 and 19 we write information about the jet algorithm, just as a check.

The event analysis function for `UserJets`, beginning in line 22 is also pretty simple.

1. In line 22, we define `s` as the shower history argument of `userfunc(...)`.

2. In line 25, we let `sf` denote the variables that describe the shower stage just after the hard interaction. In line 28, we let `pf` denote the sequential container that describes the variables describing each of the partons at shower stage `sf`.

3. In line 29, we let `sb` denote the variables that describe the shower stage at the end of the shower. In line 30, we let `p` denote the sequential container that describes the variables describing each of the partons at shower stage `sb`.

4. In lines 34 and 35, we define `wf` and `w` be the weights for the event just after the hard scattering and at the end of the shower, respectively. The weight includes the weight that is calculated as the shower history is generated and a separate color weight that we calculate from the information in `sf` and `sb`. We also include a conversion factor to convert from cross sections in $\mathrm{GeV}^{-2}$ to cross sections in nb.

5. In lines 38 and 39, we fill the zero dimensional histograms for the total cross sections after (for `1`) and before (for `2`) showering. These are the cross sections for events for which the

parton transverse momentum at the hard process was greater than the lowest $p_T$ generated, which was defined in `mod-jets.cc` to be 100 GeV. It is a check on the calculation that these cross sections come out the same since the Sudakov factors in the shower are defined to preserve probabilities.

6. Starting in line 42, 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"`:

```
#ifndef __analyser_fastjet_h__
#define __analyser_fastjet_h__ 1

#include <event.h>       //   deductor header
#include "fjcore.hh"     //   fastjet header

inline fjcore::PseudoJet make_pseudojet(const duct::lorentzvector<double>& p) {
  return fjcore::PseudoJet(p.X(), p.Y(), p.Z(), p.T());
}

template<class _Tag>
void make_pseudojets(std::vector<fjcore::PseudoJet>& pj, const duct::event<_Tag>& p)
{
  int np =  static_cast<int>(p.template number_of<duct::qcd_out>());
  pj.resize(np);
  for(int i = 0; i < np; i++)
    pj[i] = make_pseudojet(p[i+1].momentum);
}
#endif
```

7. In line 45, we apply the FASTJET analysis based on `jetDef` to the starting PseudoJets in `theparticles`. This gives a FASTJET `ClusterSequence` object that we call `clustSeq1`.

8. In line 48, we set the vector `incJets` of PseudoJets to consist of all of the found jets with $p_T > 200$ GeV.

9. In lines 51 - 53, we loop over all of the jets in `incJets` that have rapidities in the range $-2 < y < 2$. For each of these, we fill histogram `1` with the weight `w` according to the transverse momentum of the jet.

10. In lines 56 and 60 we fill histogram `2` with the jet cross section before showering. Here we need the set of parton variables before the showering, `pf`. There are only two final state partons, with indices `1` and `2`. Thus can use simply `pf[1].momentum` and `pf[2].momentum` to fill the histogram.

To compile this, you need

```
deductor --module mod-jets mod-jets.cc analyser-jets.cc analyser-angle.cc
          analyser-Ninjet.cc analyser-Zinjet.cc analyser-xperp.cc fjcore.cc
```

# 6 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.

## 6.1 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.

## 6.2 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>`.

## 6.3 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 8: Template class threevector — bits/hep-threevector.h

```
1 template<typename _Tp>
2 class threevector
3 {
4 public:
5   //  types
6   typedef _Tp value_type;
```

```
7
8     //  default constructor creates a null vector
9     threevector();
10
11    //  constructor
12    threevector(const value_type& x, const value_type& y, const value_type& z);
13
14    //  elements access (with obvious meaning)
15    const value_type& X() const;
16    const value_type& Y() const;
17    const value_type& Z() const;
18
19    value_type& X();
20    value_type& Y();
21    value_type& Z();
22
23    //  magnitude and the transverse momentum
24    value_type mag () const;   //  returns the magnitude
25    value_type perp() const;   // returns the transverse momentum
26    value_type mag2() const;   // returns the square x^2 + y^2 + z^2
27    value_type perp2() const;  // returns the transverse momentum square x^2 + y^2
28
29    //  azimuth and polar angles
30    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 = a · b
35  template<typename _Tp>
36  _Tp dot(const threevector<_Tp>& a, const threevector<_Tp>& b);
37
38  //  return the cross product of two vectors, a × b
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 9: 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 m^2 = t^2 - x^2 - y^2 - z^2
24    value_type mag() const;        //   returns invariant mass, -sqrt(|m^2|) if m^2 < 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>`.