Preprint typeset in MyNote style

6 January 2017

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. Most precisely, these notes apply to version 2.0.2. 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 user defined kernel module which specifies 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 analysis modules that perform the analysis of generated events. The kernel module and the analysis modules are created by users. We provide the main `deductor` code and, separately, 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 2.0.2. 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 also 2.0.2. 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 your 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
```

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

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

```
deductor --build dy
```

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 2.0.2                            |
|                        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 --result dy
```

This produces a directory `dy.bundle/result`. Inside of `result`, you will find a file `summary.tex`. This is a simple TEX file 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 your 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

```
1  /*   Deductor headers  */
2  #include <deductor.h>
3  #include <proc-dyjets.h>
4
5  /*   Analyser headers  */
6  #include "analyzer-ZpT.h"
7
8  /*   deductor and standard namespaces   */
9  using namespace duct;
10 using namespace std;
11 using namespace std::literals::chrono_literals;
12
13 /* This function defines what Deductor should do. */
14
15 template<class _Ordering>
16 void my_DY_module(const deductor_input& info)
17 {
18   /* deductor object */
19   deductor<hhc> kernel{qcd::ct14n};
```

```
20
21    /* We create a function object that can create the desired hard process. */
22    auto hard = [=](const model_ref& mdlr) -> dyjets
23    {
24      double Ecm = 13.0_TeV;
25      double scalefactor = 1.0;
26      double mL = 1800.0_GeV, mH = 2300.0_GeV;
27      int proton = 2212, electron = 11;
28
29      return dyjets(mdlr, Ecm, mL, mH, scalefactor, proton, proton, {electron});
30    };
31
32    /*  Create the shower process, start the calculation and return the process ID.*/
33    shower_engine_opts opts;
34    opts.max_color_suppression = 0u;    // The default is 0u.
35    opts.allow_color_branching = false; // The default is true.
36    opts.threshold.psi_min = 0.01;      // The default is 0.01.
37
38    kernel("Drell-Yan-pT", opts, hard, new UserZpT{"ZpT"});
39    cout << "Calculation started." << endl;
40  }
41
42  /*   Always export your module functions. */
43  export_my_module drell_yan_module = {
44    {"DY-13TeV",  my_DY_module<ordering::lambda>},
45  };
```

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 lines 9-11, we make some declarations for namespaces that we might want to use without fully specifying them.

3. In lines 15-16, 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&`. In this example we don't use this argument.

4. In line 19, 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.

5. In line 22 we create a function object that will create the hard process. It is created as a C++14 lambda expression. The object 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.

6. In line 24, we choose the c.m. energy of the hard process. (`double Ecm = 13.0_TeV;`)

7. In line 25, 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. In line 26, 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 1.8 TeV and 2.3 TeV (`double mL = 1.8_GeV, mH = 2300_GeV;`).

9. In line 27, 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.

10. In line 29, we return a `dyjets` process with the chosen parameters. For the final state leptons, we have a list that contains just one element: `{electron}`. Alternatively, we could define `muon` and `tau` and then replace `{electron}` by `{electron, muon, tau}`.

11. In lines 33-36, we specify some options for `deductor`. The shower generation uses the LC+ approximation for color evolution. This generates color states with color suppression index (roughly, the power of $1/N_c$) greater than zero. We generally want to turn this off and revert to the leading color (LC) approximation if the color suppression index is greater than some value that we specify. Here we set `opts.max_color_suppression` to zero so that the whole calculation is in the leading color approximation. This is the default, so one could omit this line. If `opts.max_color_suppression` is greater than zero, then setting `opts.allow_color_branching = true` provides a mechanism for reducing fluctuations in the color weights generated, thus making the calculation faster for a given statistical error. Here `opts.max_color_suppression` is zero, so we set `opts.allow_color_branching = false`.

12. In line 36, we specify the option `opts.threshold.psi_min = 0.01`. This omits the Δ term in the Sudakov exponent for threshold effects when the parameter ψ is less than 0.01. The value 0.01 is the default choice, but you can change it. If you leave out this line, you get the default choice.

13. Line 38 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 analyzer objects be created dynamically by the `new` operator. The `deductor<hhc>` object captures and owns this pointer.

14. Continuing in line 38, 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 options `opts`, the function `hard` that creates the hard process, and the analyzer.

15. Line 40 competes the definition of `my_DY_module`.

16. Finally, in lines 43-45, 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 `"DY-13TeV"` and the function, `my_DY_module`, that we have just defined. The template argument `<ordering::lambda>` specifies the type of shower ordering variable.

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

```
#ifndef __analyzer_ZpT_h__
#define __analyzer_ZpT_h__

/*   deductor headers  */
#include <user.h>

using namespace std;
using namespace duct;
using namespace duct::distpoint;

/*   Narrowing the basic_user template   */
using DistHist1d = distbook<double, hist1d>;
using DistRatio4 = distbook<std::array<double,4>, hist1d,
                    sample_ratio_traits<std::array<double,4>>>;

/*
 *   This is the declaration of the analyzer class.
 */
class UserZpT : public basic_user<hhc, DistHist1d, DistRatio4>
{
  /*   useful aliases  */
using Hist1D = Phys<DistHist1d>;
using Hist1DR4 = Phys<DistRatio4>;

public:
  /*   constructors   */
  UserZpT() = default;
  UserZpT(const char *n) {this->name(n);}

  /*   User defined init function, called once
   *   at the beginning of the calculation.
   */
  void initfunc();

  /*   The analyzer routine. It analyzes the events and fills the histograms. */
  void userfunc(const shower_history<hhc>&);
};

#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 39 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 7-9, we specify some namespaces that will be convenient.

4. In lines 12 and 13, we give convenient names to two histograming classes that we will use.

5. In line 19, 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 parameters are histograming classes that we will use.

6. In lines 22 and 23, we define two aliases that will be convenient for the code in `analyzer-ZpT.cc` in which we define what `UserZpT` does.

7. In lines 27 and 28, we define constructors our class. (We used the second in line 38 of `mod-dy.cc`.) Its argument is the name of the analyzer.

8. As we mentioned earlier, we have to overload two virtual functions of the class `basic_user`. They are declared in lines 33 and 36. 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, Full", bins, enable_norm);
13   Hist1D::phys(2, "$Z/\\gamma$ $p_T$ distribution, No Delta", bins, enable_norm);
14   Hist1D::phys(3, "$Z/\\gamma$ $p_T$ distribution, ThresholdMain", bins, enable_norm);
15   Hist1D::phys(4, "$Z/\\gamma$ $p_T$ distribution, Std", bins, enable_norm);
16   Hist1DR4::phys(5, "$Z/\\gamma$ $p_T$ distribution - ratios", bins);
17 }
18
19
20 void UserZpT::userfunc(const shower_history<hhc>& history)
21 {
22   /*   Conversion factor from 1/GeV^2 to nb    */
23   constexpr double tonb = 389379.338;
24
25   /*   the shower stage after the shower evolution   */
26   for(auto it = history.cbegin(); it != history.cend(); ++it)
```

```
27    {
28      /*   momentum of the e+e- pair   */
29      auto& p = it->state;
30      auto pee = p[-2].momentum + p[-3].momentum;
31
32      /*   Calculate the weights   */
33
34      auto& weights = it->weights;
35      // The weight for a probability conserving shower
36      // including factor to use NLO pdfs at hard scale.
37      double wStd = weights.unitary.main;
38             wStd*= weights.unitary.pdf_ratio*color_weight(*it)*tonb;
39      // The main threshold factor, omitting P^{reg} term and the Delta term.
40      double wThresholdMain = wStd*weights.threshold.main;
41      // Now include the P^{reg} term.
42      double wNoDelta = wThresholdMain*weights.threshold.reg;
43      // Now include the Delta term also, giving the full result.
44      double wFull = wNoDelta*weights.threshold.delta;
45
46      std::array<double,4> myweights4;
47      myweights4[0] = wFull;
48      myweights4[1] = wNoDelta;
49      myweights4[2] = wThresholdMain;
50      myweights4[3] = wStd;
51
52      /*   Fill the histograms   */
53      if(pee.mag() > 2000.0 && pee.mag() < 2100.0 && pee.perp() < 100.0) {
54        Hist1D::normfill(1, wFull);
55        Hist1D::physfill(1, wFull, dirac{}, pee.perp());
56        Hist1D::normfill(2, wNoDelta);
57        Hist1D::physfill(2, wNoDelta, dirac{}, pee.perp());
58        Hist1D::normfill(3, wThresholdMain);
59        Hist1D::physfill(3, wThresholdMain, dirac{}, pee.perp());
60        Hist1D::normfill(4, wStd);
61        Hist1D::physfill(4, wStd, dirac{}, pee.perp());
62        Hist1DR4::physfill(5, myweights4, dirac{}, pee.perp());
63      }
64    }
65  }
```

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

1. We are going to create two kinds of one dimensional histograms to accumulate the Z/γ transverse momentum distribution. In lines 7-9, 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, but this simple method suffices for now.

2. We will want a normalized distribution for some of our histograms, 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 one of the histograms 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.

4. In lines 13-15, we define three more histograms of this type.

5. In line 16, we define a different kind of histogram. It will be made from histograms similar to 1 through 4, but with cross sections instead of normalized probabilities. We will create ratios of unnormalized histograms 1, 2, and 3 to unnormalized histogram 4. We will see in the user function how to do this.

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. In line 23, we specify a conversion factor that we will need.

3. 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 26, we loop over all the possible shower end stages and call the loop variable `it`.

4. In line 29 we define the final state that we have found and call it `p`. The information about the final state partons is contained in `p`.

5. In line 30, 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 $-2, -3, \ldots$.)

6. In DEDUCTOR, each event comes with a weight. In fact, there are several choices for how the weight is defined. In line 34, we define a variable `weights` that contains the information that we need.

7. In lines 37 and 38, we calculate the simplest weight, which we call `wStd`. This is the event weight for a "unitary" or probability preserving shower. The first factor in `wStd` is `weights.unitary.main`, which can contain factors from the hard scattering cross section and certain factors related to the color choices if we don't use the leading color approximation. The second factor, `weights.unitary.pdf_ratio` is the ratio of the NLO parton distribution at the hard scale to the leading order parton distribution used internally in the shower. The next factor, `color_weight(*it)`, is the color overlap function of the final state, which can be highly non-trivial if we work beyond the leading color approximation. Finally there is a factor to convert the units to nanobarns.

8. In lines 40, 42, and 44, we calculate the weight `wfull` with threshold corrections. There are three types of terms in the Sudakov exponent for the threshold corrections and we multiply the corresponding exponentials: `weights.threshold.main` is the main factor, containing the

double logarithms and a trivial color factor; `weights.threshold.reg` is the factor associated with the non-singular parts of the splitting kernel; `weights.threshold.delta` is the factor associated with the term containing the function Δ. (The cut `opts.threshold.psi_min` in `mod-dy.cc` applies to this contribution.) We also calculate weights corresponding to the factors separately so that we can see what each factor does.

9. In lines 46-50, we create a four component array that contains our separate weights.

10. Our histogram covers the range $0 < P_T < 100$ GeV for events with 2000 GeV $< Q <$ 2100 GeV, so we look only at events in this range. The `if` statement in line 51 takes care of this.

11. In lines 54 and 55, we fill the first histogram, using the full weight `wFull`. This is to be a normalized histogram giving the probability to have the dimuon p_T in each bin. Thus we need to divide the cross section in each bin by the cross section in all of the bins. The `normfill` function accumulates the normalizing cross section. Then the `physfill` function accumulates the cross section for each of the bins. 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 `wFull` is entered into this bin.

12. In lines 56-61, we fill three other similar histograms.

13. In line 62, we fill four histograms with the weights `myweights4[0]`, `myweights4[1]`, `myweights4[2]`, and `myweights4[3]`. This will create the four histograms and, most importantly, three histograms containing the ratios of the contents of the first three histograms to the last one.

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 TEX file `summary.tex`, which, when you typeset it, will contain graphs of your histograms.[2] The data produced by the histograms that you declared is in the subdirectory `DY-13TeV/Drell-Yan-pT/ZpT`. The data for the one dimensional histograms are in subdirectory `plot-0` in the files `table-1.dat`, `table-2.dat`, `table-3.dat`, and `table-4.dat`. The data for the ratio histograms is in subdirectory `plot-1` in file `table-5.dat`. This table has 17 columns: columns 1-3 have the bins, while columns 10-11 have histogram 0 with its error, columns 12-13 have histogram 1 with its error, columns 14-15 have histogram 2 with its error, and columns 16-17 have histogram 3 with its error. Then columns 4-5 have the ratio of histograms 0 and 3, columns 6-7 have the ratio of histograms 1 and 3, and columns 8-9 have the ratio of histograms 2 and 3.

Notice that the user has supplied text names for the bundle created by the `deductor --new` command and then, in the user code, has supplied other names. 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.

[2] You can edit this file. One trick that is sometimes useful is to change "axis" to "semilogyaxis."

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

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

/*   Analyser headers  */
#include "analyzer-jets.h"

template<class _Ordering>
void my_jet_module(const deductor_input& info)
{
  /*  Deductor object  */
  deductor<hhc> kernel{qcd::ct14n};

  /*  Hard process     */
  auto hard = [=](const model_ref& mdl) -> hhcjets
  {
    int proton = 2212;
    double Ecm = 13.0_TeV;
    double scalefac = 1.0;
    auto pTlist = {250.0, 700.0, 1000.0, 1500.0, 2000.0, 2500.0, 3000.0, 3500.0,
      4000.0, 4500.0, 5050.0};

    return {mdl, Ecm, pTlist, scalefac, proton, proton};
  };

  /*   Jet definition    */
  fj::JetDefinition jetDef{fj::antikt_algorithm, 0.4};

  /*   Jet cross section pT distribution */
  UserJets *Jets = new UserJets{"Jets", jetDef};

  /* Create the shower process, start the calculation.   */

  shower_engine_opts opts;
  opts.max_color_suppression = 0u;    // The default is 0u.
  opts.allow_color_branching = false; // The default is true.
  opts.threshold.psi_min = 0.01;      // The default is 0.01.

  kernel("JetCalculations", opts, hard, {Jets});
  cout << "Calculation started." << endl;
}

/*   Always export your module functions!   */
export_my_module jetcalc_module = {
  {module_type::process, "Jets-13TeV", my_jet_module<ordering::lambda>}
};
```

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.

2. In line 7, we include the header for our analyzer routine. (We can have more than one analyzer. Then we need more `#include` statements.)

3. In lines 8 and 9, 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 12, 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 15 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 17, we specify the PDG code for our incoming hadrons, which are protons.

7. In line 18, we specify the c.m. energy that we want.

8. In line 19, we specify a scale factor λ for the factorization scale μ_f that we want. We set $\mu_\mathrm{f} = \lambda p_\mathrm{T}$, 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.

9. In line 20, we provide a list of transverse momenta. These tell the generator to generate hard scattering events with 250 GeV $< p_\mathrm{T} <$ 5050 GeV, with equal numbers of events with p_T between 250 GeV and 700 GeV, between 700 GeV and 1000 GeV, ..., and between 4500 GeV and 5050 GeV. You can choose what you want, but if you just set `auto pTlist↩ = {250.0}`, you will get very few events with $p_\mathrm{T} > 350$ GeV.

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

11. 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. Additionally, we include a DEDUCTOR header file `fastjet.h` in `analyzer-jets.h`. This header file defines the namespace `fj` to be an abbreviation for `fjcore`. In line 27, we specify the jet definition that we want FASTJET to use, namely the anti-k_T algorithm with $R = 0.4$.

12. In the file `analyzer-jets.h` that we included, we declared a class `UserJets`, which is a user defined analyzer for jet events. In line 30, we create a dynamically allocated (with `new`) instance, called `Jets`, of this class. We supply `Jets` with the jet definition `jetDef` that we have just created.

13. In lines 34-37, we specify options, `opts`, for the calculation, just as we did in Listing 1.

14. Recall that we created a `deductor<hhc>` object `kernel`. Now, in line 39, we use the `operator()` function of `kernel` to start the calculation. We supply to `kernel` a process name, `"JetCalculations"`, the options, the function `hard` that creates the hard process, and a list of analyzers, which here contains one element, the analyzer `Jets`. (We can have several analyzers. If we do, each is called for each event.)

15. Line 41 competes the definition of `my_jet_module`.

16. Finally, in lines 44-46, we create an object `jetcalc_module` of type `export_my_module`↩. This lets `deductor` know the class and function names associated with our calculation. We supply a plain text module name `"Jets-13TeV"` and the name, `my_jet_module`, of the function that we have just created. The template argument `<ordering::lambda>` specifies the type of shower ordering variable.

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 __analyzer_jets_h__
2  #define __analyzer_jets_h__
3
4  /*   project headers  */
5  #include <user.h>
6
7  /*   other includes   */
8  #include "fastjet.h"
9
10 /*   Define some namespaces  */
11 using namespace std;
12 using namespace duct;
13 using namespace duct::distpoint;
14
15 /*   Narrowing the distbook template   */
16 using DistVoid = distbook<double, void>;
17 using DistHist1d = distbook<double, hist1d>;
18
19 /*
20  *   This is the declaration of the analyzer class.
21  */
22 class UserJets : public basic_user<hhc, DistVoid, DistHist1d>
23 {
24   /*   useful aliases  */
25   using Hist0D = Phys<DistVoid>;
26   using Hist1D = Phys<DistHist1d>;
27
28 public:
29   /*    constructors     */
30   UserJets(const char *n, fj::JetDefinition&& jd, const duration_type& d = 2.0min)
31   : jetDef{std::move(jd)} { this->param(n, d);}
32
33   UserJets(const char *n, const fj::JetDefinition& jd, const duration_type& d = 2.0min)
34   : jetDef{jd} { this->param(n, d);}
35
```

```
36
37    /*   User defined init function, called once at the begining of the calculation.  */
38    void initfunc();
39
40    /*   The analyser routine. It analyses the events and fill the histograms.   */
41    void userfunc(const shower_history<hhc>&);
42
43    /*   For fastjet analysis - select algorithm and parameters.    */
44    fj::JetDefinition jetDef;
45
46    /*   Predefined vectors of Pseudojets     */
47    mutable std::vector<fj::PseudoJet> theparticles;
48    mutable std::vector<fj::PseudoJet> incJets;
49  };
50
51  #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 from the `deductor-user-2.0.2/Jets` directory, `"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 and 17, we define aliases for a `basic_user` class definitions: `DistVoid` defines a zero-dimensional histogram for holding just a single cross section; `DistHist1d` defines a normal one-dimensional histogram.
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 25 and 26, we define useful aliases for histogram classes.
7. In lines 20-34, we define 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. If we leave this out, the default is 2 minutes. 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 38.
9. We need an analyzer function `userfunc(...)`, which we declare in line 41.
10. In line 44, we declare an instance `jetDef` of the FASTJET class `JetDefinition`. This is initialized by the constructors.
11. In lines 47 and 48, 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: Definition of the class UserJets — Jets/analyzer-jets.cc

```
#include "analyzer-jets.h"

void UserJets::initfunc()
{
  /*   Create objects to calculate the total cross sections   */
  Hist0D::phys(1,"Total cross section");
  Hist0D::phys(2,"Total cross section (hard)");

  /*   Create histograms to calculate the one jet cross sections   */
  unsigned int nbins = 48;
  double pTmin = 250.0_GeV, pTmax = 5050.0_GeV;
  auto bins = spacing<hist1d>::linear(nbins, pTmin, pTmax);

  Hist1D::phys(1, "One jet inclusive cross section, full threshold effects", bins);
  Hist1D::phys(2, "One jet inclusive cross section, no Delta", bins);
  Hist1D::phys(3, "One jet inclusive cross section, ThresholdMain", bins);
  Hist1D::phys(4, "One jet inclusive cross section, Std., no threshold effects", bins);
  Hist1D::phys(5, "One jet inclusive cross section at hard interaction", bins);

  cout << "UserJets is using jet algorithm " << jetDef.jet_algorithm()
       << " with R = " << jetDef.R() << endl;
}

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

  /*   the shower stage after the shower evolution   */
  for(auto it = history.cbegin(); it != history.cend(); ++it)
  {
    /*   showered and hard events   */
    auto& p = it->state;
    auto& hard = it.hard();
    auto& phard = hard.state;

    // Weights after showering
    auto& weights = it->weights;
    // The weight for a probability conserving shower
    // including factor to use NLO pdfs at hard scale.
    double wStd  = weights.unitary.main;
           wStd *= weights.unitary.pdf_ratio*color_weight(*it)*tonb;
    // The main threshold factor, omitting P^{reg} term and the Delta term.
    double wThresholdMain = wStd*weights.threshold.main;
    // Now include the P^{reg} term.
    double wNoDelta = wThresholdMain*weights.threshold.reg;
    // Now include the Delta term also, giving the full result.
    double wFull = wNoDelta*weights.threshold.delta;

    // Weights before showering.
    // If there are multiple branches from the same hard start,
    // we count the hard contribution only once.
    auto hweights = hard.weights;
    double hweight = 0.0;
    if (newHard)
```

```
56      {
57        hweight  = hweights.unitary.main*color_weight(hard)*tonb;
58        hweight *= hweights.unitary.pdf_ratio;
59      }
60      newHard = false;
61
62      /*   Total cross sections   */
63      Hist0D::physfill(1, wFull);
64      Hist0D::physfill(2, hweight);
65
66      //   One jet inclusive cross section after showering
67      //  -------------------------------------------------
68
69      /*   Converting deductor event to FastJet format   */
70      make_pseudojets(theparticles, p);
71
72      /*    Run fastjet algorithm   */
73      fj::ClusterSequence clustSeq(theparticles, jetDef);
74
75      /*    Extract inclusive jets sorted by pT (above pTjetmin)   */
76      double pTjetmin = 250.0_GeV;
77      incJets = clustSeq.inclusive_jets(pTjetmin);
78
79      /*    Fill histogram, only count jets that have |y| < 2.0   */
80      for(unsigned j = 0; j < incJets.size(); j++)
81        if(abs(incJets[j].rap()) < 2.0)
82        {
83          Hist1D::physfill(1, wFull,          dirac{}, incJets[j].perp());
84          Hist1D::physfill(2, wNoDelta,       dirac{}, incJets[j].perp());
85          Hist1D::physfill(3, wThresholdMain, dirac{}, incJets[j].perp());
86          Hist1D::physfill(4, wStd,           dirac{}, incJets[j].perp());
87        }
88
89      //   One jet inclusive cross section before showering
90      //  -------------------------------------------------
91
92      /*   Fill histogram, only count jets that have |y| < 2.0
93       *   The two jets are back-to-back, since we have only two
94       *   partons in the final state.
95       */
96
97      if(abs(phard[1].momentum.rapidity()) < 2.0)
98      {
99        Hist1D::physfill(5, hweight, dirac{}, phard[1].momentum.perp());
100     }
101     if(abs(phard[2].momentum.rapidity()) < 2.0)
102     {
103       Hist1D::physfill(5, hweight, dirac{}, phard[2].momentum.perp());
104     }
105   }
106 }
```

The initialization function for `UserJets` is fairly simple.

1. In lines 6 and 7, we create zero-dimensional histograms for the jet cross section within the cuts at the hard interaction level and after showering.

2. 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 10-12.

3. In lines 14-18, we create the one dimensional histogram objects that we will need.

In the analysis function, we learn how to incorporate a jet analysis with the help of FastJet and we learn how to get access to the parton configuration and weights at the start of the shower.

1. In line 24, we define `history` as the shower history argument of `userfunc(...)`.

2. In line 26, we define the conversion factor that we will need to convert GeV^{-2} to nb.

3. In line 27, we define a `bool` variable `newhard`, initialized to `true`. The purpose of this will become apparent below.

4. 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 30, we define an iterator `it` and loop over its values.

5. In line 33, we define a reference `p` to the shower state after the shower evolution.

6. 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 34. In line 35, we define the partonic state at the start of the shower, `hard.state`, and call it `phard`.

7. We need the weights associated with the state at the end of the shower. We calculate the one jet cross section with the full threshold correction, with just a standard shower that has no threshold correction, or with intermediate choices for what threshold terms to include. The weights that we need are defined as in Listing 3.

8. We also need the weight associated with the state at the start of the shower. In line 53 we define the weights that we need.

9. We call the hard event weight that we want `hweight`. We have to be careful because there can be more than one final state corresponding to the same hard state. In that case, iterating over `it` will produce `hard` given by `it.hard()` more than once. Before entering the loop over `it`, we set `newHard = true`. Now in line 54 we declare `hweight` and initialize it to `0.0`. Then if `newHard` is `true`, we calculate `hweight` in lines 57 and 58. But if `newHard` is `false`, we leave `hweight` with its initial value of `0.0`. In either case, we set `newHard = false` in line 60, so that we will obtain a non-zero value for `hweight` only once.

10. In lines 66 and 67, we fill our zero-dimensional histograms that will accumulate the total generated cross sections.

11. Next, we want to calculate the one jet inclusive cross section after showering and fill our histograms. We begin in line 70 by making FastJet pseudojets from the final state particles `p` given by Deductor. This uses the function `make_pseudojets` defined in `fastjet.h`.

12. In lines 73, 76, and 77, we use FastJet to create a vector `incJets` of the jets in the final state with $P_{\text{T}} > 250$ GeV.

13. In line 80 - 87, we fill the histograms for jets with rapidity in the range $|y| < 2$.

14. Finally, we want to fill our histogram for the one jet inclusive cross section before showering. We use the weight `hweight`. At the hard interaction, there are two partons. Their momenta are `phard[1].momentum` and `phard[2].momentum`. We fill the histogram in lines 97 - 104.

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 have 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 DEDUCTOR are those 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
  value_type theta() const;    //  return the polar angle
};

//  return the dot product of two vector, same as a*b = a · b
template<typename _Tp>
_Tp dot(const threevector<_Tp>& a, const threevector<_Tp>& b);

//  return the cross product of two vectors, a × b
template<typename _Tp> threevector<_Tp>
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

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

  //  constructors
  lorentzvector();  //   constructs null vector
  lorentzvector(const _Tp& x, const _Tp& y, const _Tp& z, const _Tp& t);
  lorentzvector(const threevector_type& v, const value_type& t);

  //   elements access, gives the reference to the timelike component
  const value_type& T() const;
  value_type& T();

  //  member functions
  value_type plus () const;      //   returns t + z
  value_type minus() const;      //   returns t - z
  value_type rapidity() const;   //   returns rapidity
  value_type prapidity() const;  //   returns pseudo-rapidity
  value_type mag2() const;       //   returns m^2 = t^2 - x^2 - y^2 - z^2
  value_type mag() const;        //   returns invariant mass, -sqrt(|m^2|) if m^2 < 0

  //  returns the transverse component that is perpendicular to both a and b
  lorentzvector transverse(const lorentzvector& a, const lorentzvector& b) const;
};
```

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>`.