

QCDLoop: a comprehensive framework for one-loop scalar integrals

Stefano Carrazza¹, R. Keith Ellis² and Giulia Zanderighi^{1,3}

¹ *Theoretical Physics Department, CERN, Geneva, Switzerland*

² *Institute for Particle Physics Phenomenology, Department of Physics,
Durham University, Durham DH1 3LE, UK*

³ *Rudolf Peierls Centre for Theoretical Physics, University of Oxford,
Oxford OX1 3NP, UK*

Abstract

We present a new release of the QCDLoop library based on a modern object-oriented framework. We discuss the available new features such as the extension to the complex masses, the possibility to perform computations in double and quadruple precision simultaneously, and useful caching mechanisms to improve the computational speed. We benchmark the performance of the new library, and provide practical examples of phenomenological implementations by interfacing this new library to Monte Carlo programs.

Program Summary

Name of the program: QCDLoop

Version: 2.0.0

Program obtainable from: <http://cern.ch/qcdloop>

Distribution format: compressed tar file from the GitHub git repository

E-mail: stefano.carrazza@cern.ch, keith.ellis@durham.ac.uk giulia.zanderighi@cern.ch

License: GNU Public License GPLv3

Computers: all

Operating systems: all with a c++11 compliant compiler with `quadmath` support, see Sect. 3.

Program language: c/c++, fortran 77/90, and python

Memory required to execute: $\lesssim 2$ MB

Other programs called: None

External files needed: None

Number of bytes in distributed program, including test data etc.: ~ 1.0 MB

Keywords: one-loop scalar integrals, tadpole, bubble, triangle, box, numerical evaluation, QCD, Feynman integrals

Nature of the physical problem: Computation of one-loop scalar integrals

Solution Method: Numerical evaluation of one-loop scalar integrals such as tadpole, bubble, triangle and box through analytic expressions.

Typical running time: detailed performance benchmark presented in Sect. 4

Contents

1	Introduction	4
2	One-loop scalar integral formalism	5
3	QCDLoop library documentation	6
3.1	Download and installation	6
3.2	The framework design	7
3.2.1	The namespace <code>ql</code> : types, typedefs and templates	9
3.2.2	Code examples	11
3.2.3	Caching mechanisms	12
3.2.4	Fortran and python wrappers	12
4	Validation and benchmarks	13
4.1	Performance tests	13
4.2	Phenomenological applications	14
4.2.1	MCFM interface	14
4.2.2	Ninja and GoSam interface	16
5	Conclusions	17

1 Introduction

The requirements of precision physics at the LHC and future experiments demand high precision theoretical predictions. In this context perturbative expansions in the coupling constant play a prominent role. The field of next-to-leading order (NLO) QCD corrections has undergone a revolution in the last 10-15 years, see e.g. Refs. [1, 2] and references therein. This revolution resulted in computational tools that allow one to obtain NLO results for generic processes in a semi- or fully automated way [3–6]. One longstanding bottleneck in NLO calculations has been the computation of virtual corrections. Recently, it was understood how to use algebraic methods to write any virtual amplitude as a product of coefficients (that can be computed essentially as products of tree level amplitudes) and of one-loop scalar master integrals. Still, for complicated final states often several CPU years are required to obtain distributions that are smooth both at low and high momentum scales. Practically, this means that one needs to run codes for several days on computer farms. Obviously any improvement in the performance of these tools would be welcome.

In more recent years, a NNLO (next-to-next-to-leading) revolution has also started, and now almost all $2 \rightarrow 2$ Standard Model processes are known to this accuracy. One of the ingredients required to achieve NNLO accuracy for $pp \rightarrow X$ is a pure NLO prediction $pp \rightarrow X + 1$ parton in the kinematic configuration where the parton becomes unresolved. Hence, one loop scalar integrals are a crucial ingredient for both NLO and NNLO calculations, and a fast computation of these integrals, that remains stable also in kinematic regions where external partons become soft or collinear, is required. In the original paper by two of us (Ellis and Zanderighi [7]) an algorithm was provided to calculate all the divergent one-loop integrals. The algorithm proceeds by defining a basis set of divergent integrals, some of which were available in the literature prior to ref. [7], and some of which were calculated *ab initio*. The results for all the divergent integrals in the basis set were given in ref. [7] and they were implemented in a fortran code, dubbed `QCDLoop`. For finite triangle- and box-integrals `QCDLoop` relied on `ff` [8]. One-loop scalar integrals have been implemented also in a number of other packages: `LoopTools` [9], `OneLoop` [10] and `Collier` [11]. In the case of unstable particles, calculations are often performed in the complex mass scheme [12]. So far, the `QCDLoop` library was limited to real masses in the propagators. Here we present an extension of the package to deal with complex masses. More generally, the aim of this paper is to present `QCDLoop 2.0`¹, a new library written in `c++` and based on the `QCDLoop 1.96` formalism documented in Ref. [7]. This new framework includes, new features such as the extension to complex masses, the possibility to switch precision from double to quadruple precision on the fly. Furthermore this new framework provides an abstract object-oriented inheritance mechanism which simplifies the implementation of caching algorithms which is useful when high performance is required.

The outline of this paper is the following. In Section 2 we present a short summary of the analytic expressions and relative diagrams implemented in `QCDLoop`. In Section 3 we describe the library structure and we introduce the main functionalities of `QCDLoop` and describe the standard user interface. In Section 4 we present a detailed performance benchmark followed by results obtained with the integration of `QCDLoop` in a few public Monte Carlo simulation codes. Finally, in Section 5 we summarize the features and advantages of the new `QCDLoop` package.

¹In the following sections the label “`QCDLoop`” refers to the new library

2 One-loop scalar integral formalism

The `QCDDLoop` library provides the numerical evaluation of one-loop scalar integrals such as tadpole, bubble, triangle and box through analytic expressions. This set of integrals constitutes a basis for one-loop scalar integrals. In Fig. 1 we provide a graphical representation for the definition of the following integrals:

- Tadpole:

$$I_1^D(m_1^2) = \frac{\mu^{4-D}}{i\pi^{\frac{D}{2}} r_\Gamma} \int d^D l \frac{1}{(l^2 - m_1^2 + i\epsilon)}, \quad (1)$$

- Bubble:

$$I_2^D(p_1; m_1^2, m_2^2) = \frac{\mu^{4-D}}{i\pi^{\frac{D}{2}} r_\Gamma} \int d^D l \frac{1}{(l^2 - m_1^2 + i\epsilon) ((l + q_1)^2 - m_2^2 + i\epsilon)}, \quad (2)$$

- Triangle:

$$I_3^D(p_1^2, p_2^2, p_3^2; m_1^2, m_2^2, m_3^2) = \frac{\mu^{4-D}}{i\pi^{\frac{D}{2}} r_\Gamma} \times \int d^D l \frac{1}{(l^2 - m_1^2 + i\epsilon) ((l + q_1)^2 - m_2^2 + i\epsilon) ((l + q_2)^2 - m_3^2 + i\epsilon)}, \quad (3)$$

- Box:

$$I_4^D(p_1^2, p_2^2, p_3^2, p_4^2; s_{12}, s_{23}; m_1^2, m_2^2, m_3^2, m_4^2) = \frac{\mu^{4-D}}{i\pi^{\frac{D}{2}} r_\Gamma} \times \int d^D l \frac{1}{(l^2 - m_1^2 + i\epsilon) ((l + q_1)^2 - m_2^2 + i\epsilon) ((l + q_2)^2 - m_3^2 + i\epsilon) ((l + q_3)^2 - m_4^2 + i\epsilon)}, \quad (4)$$

where $q_n \equiv \sum_{i=1}^n p_i$ and $s_{ij} = (p_i + p_j)^2$. The above expressions are in the Bjorken-Drell metric so that $l^2 = l_0^2 - l_1^2 - l_2^2 - l_3^2$. In this paper we consider momenta to be real, but the masses to be either real or complex. Near four dimensions we use $D = 4 - 2\epsilon$ and μ is a scale introduced so that the integrals preserve their natural dimensions, despite excursions away from $D = 4$. We have also removed the overall constant term which occurs in D -dimensional integrals

$$r_\Gamma \equiv \frac{\Gamma^2(1 - \epsilon)\Gamma(1 + \epsilon)}{\Gamma(1 - 2\epsilon)} = \frac{1}{\Gamma(1 - \epsilon)} + \mathcal{O}(\epsilon^3) = 1 - \epsilon\gamma + \epsilon^2 \left[\frac{\gamma^2}{2} - \frac{\pi^2}{12} \right] + \mathcal{O}(\epsilon^3). \quad (5)$$

The explicit expressions for all the divergent integrals in `QCDDLoop 2.0` are presented in detail in the original paper of Ellis and Zanderighi [7]. As noted in that paper, some of the results for the divergent integrals were new, but many of them were not. We refer the reader to Ref. [7] for details and the appropriate references. These expressions have been adjusted for performance optimization and the proper analytical continuation has been performed to deal with complex masses. The finite integrals for the bubble topology are taken from Refs. [13,14]. Finite integrals for the triangle topology with real masses are obtained from Refs. [15,16] following the `LoopTools` implementation. Expressions for complex masses can be derived from Refs. [13,15–17]. We

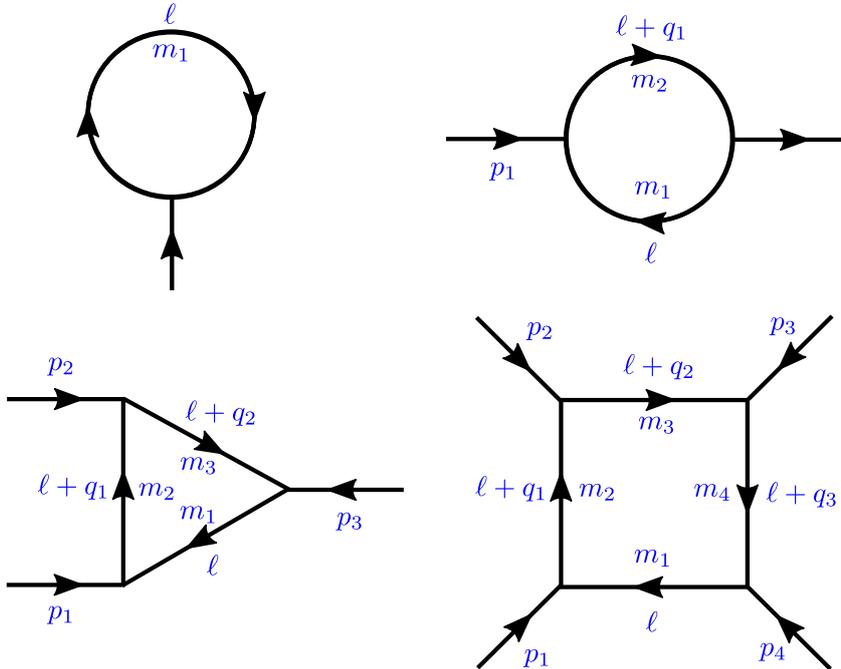


Figure 1: The notation for the one-loop tadpole, bubble, triangle and box integrals.

found that the expressions of Ref. [17], as implemented in `OneLoop`, had the best performance and hence this was the approach that we followed in our implementation. Finally, the finite box integrals are based on Ref. [17], following the `LoopTools` implementation. We invite the reader to examine the `QCDLoop 2.0` source code and documentation for further details.

3 QCDLoop library documentation

In this section we present the user manual for the `QCDLoop` library. First of all, we discuss how to download and install `QCDLoop`. After that, we illustrate the new framework design in `c++`, explaining how objects are organized and how to run a simple program. We conclude the discussion by presenting the available caching mechanisms and the `fortran` and `python` wrappers.

3.1 Download and installation

The `QCDLoop` library is available from the website:

<http://cern.ch/qcdloop>

The installation of the `QCDLoop` library can be easily performed using the standard `autotools` sequence:

```
1 ./configure
2 make
3 make install
```

which automatically installs QCDLoop in `/usr/local/`. Note that the QCDLoop library requires a `c++11` compliant compiler with `quadmath` support, such as `g++ 4.72`, `icpc 133` and or more recent versions of these compilers. The `configure` script will check for these and other system requirements before building the `makefiles`. In order to use a different installation path one can use the option:

```
1 ./configure --prefix=<path to the installation folder>
```

In this case, the QCDLoop installation path should be included to the environment variables `PATH` and `LD_LIBRARY_PATH`, adding to the local `.bashrc` file (or `.profile` file on Mac) the string:

```
1 export PATH=$PATH:<installation folder>/bin
2 export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:<installation folder>/lib
```

If the system provides more than one `c++` compiler we suggest to set the preferable choice when running `configure`:

```
1 ./configure CXX=<compiler name / path>
```

Finally, this package provides a `qcdloop-config` tool which simplifies the usage of the library when linking and compiling with user's codes. We provide the following flags: `--help`: shows the help message; `--prefix`: shows the installation prefix; `--incdir`: shows the path to the `qcdloop` header directory; `--libdir`: shows the path to the `qcdloop` library directory; `--cppflags`: gets compiler flags for use with the C preprocessor stage of `c++` compilation; `--ldflags`: gets compiler flags for use with the linker stage of any compilation; `--version`: returns `qcdloop` release version number

3.2 The framework design

The development of a new framework for QCDLoop is motivated by the following needs:

- generalization of the code to support complex masses;
- provision of the ability to provide double and quadruple precision results simultaneously;
- the removal of code dependencies such as the `ff` library [8];
- improvement of performance by implementing more sophisticated LRU cache algorithms;
- provision of a modern framework based on an object-oriented language, such as `c++`, which simplifies future developments and native integration with modern codes.

²<https://gcc.gnu.org/>

³<https://software.intel.com/>

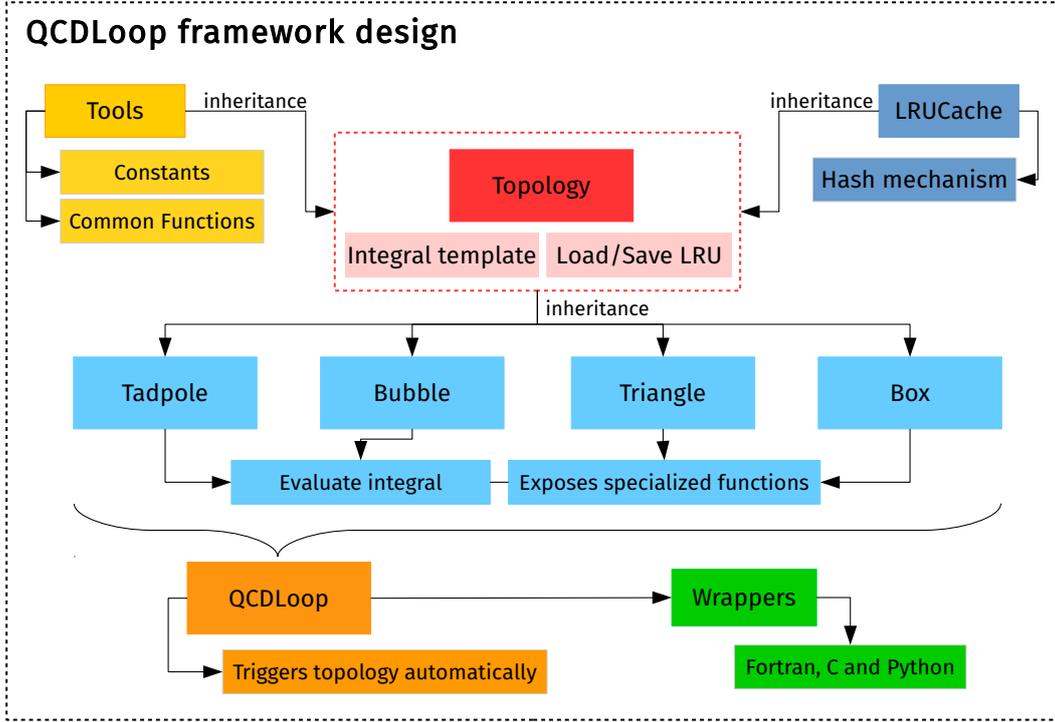


Figure 2: QCDLoop framework design.

The new QCDLoop framework design is presented schematically in Fig. 2. The core of the code is the `Topology` class which inherits common methods from `Tools` and `LRUCache` classes. `Topology` is a templated class which provides the public pure virtual `integral` method for the specialization of the one-loop scalar integrals, together with the basic mechanism for load and storage of cached results. The `Tools` templated class initializes real and complex constants based on the allocated precision type, and it provides a common set of functions which are shared by the specialization of topologies. It is important to highlight that this template class reduces to a minimum the duplication of code when a method or constant is required by several specializations. The `LRUCache` class implements a “latest recent used” (LRU) caching algorithm for dynamic caching sizes which are discussed in detail in Sec. 3.2.3.

The specialization of the one-loop scalar integrals are implemented in four classes, one for each topology: `Tadpole`, `Bubble`, `Triangle` and `Box`. These classes implement the public pure virtual `integral` method from `Topology`. For each topology the `integral` method triggers automatically the kinematics and selects the appropriate function call, when possible it also provides the public methods for the direct computation of the specific kinematics. A detailed summary of the function calls and the corresponding class methods are shown in Table 1 using the notation from Ref. [7].

On top of the specialized topologies we provide the `QCDLoop` class. This is a high-level user interface which detects the topology by checking the size of the `integral` arguments. This function verifies the consistency of the input arguments and calls the respective topology:

```

1  /*!
2  * Standard arguments to retrieve one-loop scalar integrals.
3  * output:
4  *   res: vector of dim(3) containing the coefficients in the Laurent series
5  *       res[0]: finite part (1/eps^0)
6  *       res[1]: single pole (1/eps^1)
7  *       res[2]: double pole (1/eps^2)
8  * input:
9  *   mu2: is the square of the scale mu
10 *   m: array containing the squares of the masses of the internal lines
11 *   p: array containing the four-momentum squared of the external lines
12 */
13 ql::QCDLoop<TOutput, TMass, TScale>::integral(vector<TOutput> &res,
14                                             TScale const& mu2,
15                                             vector<TMass> const& m,
16                                             vector<TScale> const& p) const;

```

The trigger mechanism employed by the `QCDLoop` class is the safest and simplest way to access all the library functionalities, however when maximum performance is required the user is invited to allocate the specific topology in order to remove the overhead due to the triggering procedure.

We conclude the description of the framework design by highlighting the availability of `c++` wrappers to `fortran`, `c` and `python`, further details about these interfaces are presented in Sect. 3.2.4. The native `c++` interface of the `QCDLoop` library is thread-safe only when the caching algorithms are switched off. It is particularly important to highlight that the `fortran` wrapper is not thread-safe by construction.

Further details about the code are fully documented using the `doxygen`⁴ syntax. The respective documentation is located in the `qcdloop/doc` folder.

3.2.1 The namespace `ql`: types, typedefs and templates

From a technical point of view all objects of the `QCDLoop` library are implemented in the `ql` namespace. In this namespace we define also aliases for double and quadruple precision real and complex types so that the primitive `ql` types are `double`, `qdouble`, `complex` and `qcomplex`:

```

1 namespace ql
2 {
3     typedef __float128 qdouble;           // quadruple precision real type
4     typedef __complex128 qcomplex;       // quadruple precision complex type
5     typedef std::complex<double> complex; // double precision complex type
6 }

```

The quadruple real and complex types are standard `quadmath` objects, and the double complex type corresponds to the `std::complex` type. Specialized mathematical operations are implemented for each type in the inline header `qcdloop/math.h`.

In order to allocate simultaneously double and quadruple precision objects, all classes presented in Fig. 2 are templated with three typenames: `TOutput` the output type, `TMass` the mass type and `TScale` the scale and momenta type. The accepted types for each typename is listed below:

⁴www.doxygen.org

Function	Short name	Method specialization
$I_1^D(m^2)$	Tadpole	TadPole<>::integral()
$I_2^D(s; m_1^2, m_2^2)$	Bubble finite (BB0)	Bubble<>::BB0()
$I_2^D(m^2; 0, m^2)$	Bubble BB1	Bubble<>::BB1()
$I_2^D(0; 0, m^2)$	Bubble BB2	Bubble<>::BB2()
$I_2^D(s; 0, 0)$	Bubble BB3	Bubble<>::BB3()
$I_2^D(s; 0, m^2)$	Bubble BB4	Bubble<>::BB4()
$I_2^D(0; m_1^2, m_2^2)$	Bubble BB5	Bubble<>::BB5()
$I_3^D(0, 0, 0; m_1^2, m_2^2, m_3^2)$	Triangle finite (TIN0)	Triangle<>::TIN0()
$I_3^D(0, 0, p_3^2; m_1^2, m_2^2, m_3^2)$	Triangle finite (TIN1)	Triangle<>::TIN1()
$I_3^D(0, p_2^2, p_3^2; m_1^2, m_2^2, m_3^2)$	Triangle finite (TIN2)	Triangle<>::TIN2()
$I_3^D(p_1^2, p_2^2, p_3^2; m_1^2, m_2^2, m_3^2)$	Triangle finite (TIN3)	Triangle<>::TIN3()
$I_3^D(0, 0, p_3^2; 0, 0, 0)$	Triangle T1	Triangle<>::T1()
$I_3^D(0, p_2^2, p_3^2; 0, 0, 0)$	Triangle T2	Triangle<>::T2()
$I_3^D(0, p_2^2, p_3^2; 0, 0, m^2)$	Triangle T3	Triangle<>::T3()
$I_3^D(0, p_2^2, m^2; 0, 0, m^2)$	Triangle T4	Triangle<>::T4()
$I_3^D(0, m^2, m^2; 0, 0, m^2)$	Triangle T5	Triangle<>::T5()
$I_3^D(m_2^2, s, m_3^2; 0, m_2^2, m_3^2)$	Triangle T6	Triangle<>::T6()
$I_4^D(p_1^2, p_2^2, p_3^2, p_4^2; s_{12}, s_{23}; 0, 0, 0, 0)$	Box finite (BIN0)	Box<>::BIN0()
$I_4^D(p_1^2, p_2^2, p_3^2, p_4^2; s_{12}, s_{23}; 0, 0, 0, m_4^2)$	Box finite (BIN1)	Box<>::BIN1()
$I_4^D(p_1^2, p_2^2, p_3^2, p_4^2; s_{12}, s_{23}; 0, 0, m_3^2, m_4^2)$	Box finite (BIN2)	Box<>::BIN2()
$I_4^D(p_1^2, p_2^2, p_3^2, p_4^2; s_{12}, s_{23}; 0, m_2^2, m_3^2, m_4^2)$	Box finite (BIN3)	Box<>::BIN3()
$I_4^D(p_1^2, p_2^2, p_3^2, p_4^2; s_{12}, s_{23}; m_1^2, m_2^2, m_3^2, m_4^2)$	Box finite (BIN4)	Box<>::BIN4()
$I_4^D(0, 0, 0, 0; s_{12}, s_{23}; 0, 0, 0, 0)$	Box B1	Box<>::B1()
$I_4^D(0, 0, 0, p_4^2; s_{12}, s_{23}; 0, 0, 0, 0)$	Box B2	Box<>::B2()
$I_4^D(0, p_2^2, 0, p_4^2; s_{12}, s_{23}; 0, 0, 0, 0)$	Box B3	Box<>::B3()
$I_4^D(0, 0, p_3^2, p_4^2; s_{12}, s_{23}; 0, 0, 0, 0)$	Box B4	Box<>::B4()
$I_4^D(0, p_2^2, p_3^2, p_4^2; s_{12}, s_{23}; 0, 0, 0, 0)$	Box B5	Box<>::B5()
$I_4^D(0, 0, m^2, m^2; s_{12}, s_{23}; 0, 0, 0, m^2)$	Box B6	Box<>::B6()
$I_4^D(0, 0, m^2, p_4^2; s_{12}, s_{23}; 0, 0, 0, m^2)$	Box B7	Box<>::B7()
$I_4^D(0, 0, p_3^2, p_4^2; s_{12}, s_{23}; 0, 0, 0, m^2)$	Box B8	Box<>::B8()
$I_4^D(0, p_2^2, p_3^2, m^2; s_{12}, s_{23}; 0, 0, 0, m^2)$	Box B9	Box<>::B9()
$I_4^D(0, p_2^2, p_3^2, p_4^2; s_{12}, s_{23}; 0, 0, 0, m^2)$	Box B10	Box<>::B10()
$I_4^D(0, m_3^2, p_3^2, m_4^2; s_{12}, s_{23}; 0, 0, m_3^2, m_4^2)$	Box B11	Box<>::B11()
$I_4^D(0, m_3^2, p_3^2, p_4^2; s_{12}, s_{23}; 0, 0, m_3^2, m_4^2)$	Box B12	Box<>::B12()
$I_4^D(0, p_2^2, p_3^2, p_4^2; s_{12}, s_{23}; 0, 0, m_3^2, m_4^2)$	Box B13	Box<>::B13()
$I_4^D(m_2^2, m_2^2, m_4^2, m_4^2; s_{12}, s_{23}; 0, m_2^2, 0, m_4^2)$	Box B14	Box<>::B14()
$I_4^D(m_2^2, p_2^2, p_3^2, m_4^2; s_{12}, s_{23}; 0, m_2^2, 0, m_4^2)$	Box B15	Box<>::B15()
$I_4^D(m_2^2, p_2^2, p_3^2, m_4^2; s_{12}, s_{23}; 0, m_2^2, m_3^2, m_4^2)$	Box B16	Box<>::B16()

Table 1: Summary of the function calls and the corresponding class methods in QCDLoop following the notation from Ref. [7]. The second column refers to the labels used in Figs. 3 and 4. Note that finite configurations do not require D dimension.

```

1 // Typenames and possible combinations (columns)
2 typename TOutput -> complex | qcomplex | complex | qcomplex
3 typename TMass   -> double   | qdouble   | complex | qcomplex
4 typename TScale  -> double   | qdouble   | double  | qdouble

```

Template classes are locked to these combinations. The compiler prevents the allocation of wrong combinations at compilation time. Further extensions for the typenames are possible if required.

3.2.2 Code examples

A simple example of code usage in c++ is presented in the code snippet given below. The code shows how to compute a tadpole double precision integral first by using the `QCDDLoop` trigger and then the direct allocation of the `TadPole` class. A similar example is then illustrated for the quadruple precision and complex mass calculation, see comments in the code. In order to change the topology it is sufficient to modify the content and size of the squared momenta and mass vectors and the initialization of the specific topology class if a direct computation is desired. Further examples are available and are build at the compilation time in the `examples/` folder.

```

1 #include <qcdloop/qcdloop.h>
2 using namespace ql;
3
4 int main() {
5     // double precision variables
6     double mu2 = ql::Pow(1.7,2);
7     std::vector<double> p = {};
8     std::vector<double> m = {5.0};
9     std::vector<complex> res(3);
10
11     // Trigger example - Tadpole double precision with real mass
12     ql::QCDDLoop<complex,double,double> auto_trigger;
13     auto_trigger.integral(res, mu2, m, p);
14
15     // Tadpole direct call - double precision with real mass
16     ql::TadPole<complex,double,double> tp;
17     tp.integral(res, mu2, m, p);
18
19     // quadruple precision and complex mass variables
20     qdouble mu2q = ql::Pow(1.7q,2);
21     std::vector<qdouble> pq = {};
22     std::vector<qcomplex> mq = { {5.0q,-1.0q} };
23     std::vector<qcomplex> resq(3);
24
25     // Trigger example - Tadpole quadruple precision with complex mass
26     ql::QCDDLoop<qcomplex,qcomplex,qdouble> auto_trigger_q;
27     auto_trigger_q.integral(resq, mu2q, mq, pq);
28
29     // Tadpole direct call - quadruple precision with complex mass
30     ql::TadPole<qcomplex,qcomplex,qdouble> tpq;
31     tpq.integral(resq, mu2q, mq, pq);
32
33     return 0; }

```

3.2.3 Caching mechanisms

We provide two caching algorithms for fast retrieval of previously computed one-loop scalar integrals. By default the `Topology` class implements and allocates a “last-used” LU cache, with dimension $N = 1$, where only the last computed result is stored. Such a caching mechanism is similar to the previous `QCDLoop 1.96` version, however a faster argument comparison algorithm parser is employed. Note that this approach is the fastest method when using a small cache with $N = 1$.

Due to the practical limitations of this approach we implement a dynamic size “last-recent-used” (LRU) algorithm in the `LRUCache` class. Such approach provides a simple and fast method to store the last N computed integrals, where N is chosen by the user. The algorithm first computes a key associated with the integral arguments by using the Murmur hash algorithm from the `std::Hash` function available from the `c++` standard library. We have verified explicitly that the rate of hash collisions is negligible in the context of one-loop scalar integral computations. Secondly, the result of the integral is stored in an unordered map so that the searching mechanism is based on a single key search. Performance results are presented and discussed in detail in Sec. 4.1.

In order to activate the different caching algorithms the user should call the `setCacheSize(int const& size)` method which is available from all inherited classes from `Topology` and `QCDLoop`. The code automatically selects the appropriate caching algorithm based on the `size` parameter:

```
1  ql::TadPole<complex,double,double> tp; // default cache size N = 1
2  tp.setCacheSize(10); // sets the cache to N=10
3  // perform calculation...
```

Note the possibility to switch off the caching algorithm by setting `size = 0`.

3.2.4 Fortran and python wrappers

The `QCDLoop` library provides wrappers to `fortran` (77/90) and `c` based on the same syntax of `QCDLoop 1.96` in [7]. Table 2 lists the available functions for different topologies and argument types. We enlarge the previous `qlIj` ($j = 1, 2, 3, 4$) syntax with extra functions identified by new prefixes: `qlIjc` computes integrals in double precision and complex masses, `qlIjq` computes integrals in quadruple precision and real masses, `qlIjqc` computes integrals in quadruple precision and complex masses. In parallel to these functions we included the new `qlcachesize(size)` function which provides the interface to modify the cache size. Further details of these wrappers are available in the header `qcdloop/wrapper.h`

We also provide a basic `python` interface to the library through `cython` which can be extended by the user easily. In order to build and install the `python` module for `QCDLoop` the user should first install the library following the instructions in Sect. 3.1 (and exporting the `PATH` and `LD_LIBRARY_PATH` environment variables) and then perform the following operations:

```
1  cd pywrap
2  python setup.py install
```

The last command invokes the `cython` compiler and installs the module to the system `PYTHONPATH`.

Integral	fortran function	Description	
-	qlinit()	initializes the library	
-	qlcachesize(size)	sets the cache size	
		Precision	Masses
I_1^D	ql1(m1,mu2,ep)	double	real
	ql1c(m1,mu2,ep)	double	complex
	ql1q(m1,mu2,ep)	quadruple	real
	ql1qc(m1,mu2,ep)	quadruple	complex
I_2^D	qlI2(p1,m1,m2,mu2,ep)	double	real
	qlI2c(p1,m1,m2,mu2,ep)	double	complex
	qlI2q(p1,m1,m2,mu2,ep)	quadruple	real
	qlI2qc(p1,m1,m2,mu2,ep)	quadruple	complex
I_3^D	qlI3(p1,p2,p3,m1,m2,m3,mu2,ep)	double	real
	qlI3c(p1,p2,p3,m1,m2,m3,mu2,ep)	double	complex
	qlI3q(p1,p2,p3,m1,m2,m3,mu2,ep)	quadruple	real
	qlI3qc(p1,p2,p3,m1,m2,m3,mu2,ep)	quadruple	complex
I_4^D	qlI4(p1,p2,p3,p4,s12,s23,m1,m2,m3,m4,mu2,ep)	double	real
	qlI4c(p1,p2,p3,p4,s12,s23,m1,m2,m3,m4,mu2,ep)	double	complex
	qlI4q(p1,p2,p3,p4,s12,s23,m1,m2,m3,m4,mu2,ep)	quadruple	real
	qlI4qc(p1,p2,p3,p4,s12,s23,m1,m2,m3,m4,mu2,ep)	quadruple	complex

Table 2: fortran and c wrapper functions.

The python wrapper contains the `qcdloop.QCDLoop` object which reflects exactly the class `ql::QCDLoop` from the library. One can obtain results by querying the python console with:

```

1 # TadPole computation in python
2 from qcdloop import QCDLoop as ql
3 m = [0.5]
4 mu2 = 1.7**2
5 out = ql.integral(mu2,m)

```

4 Validation and benchmarks

In this section we quantify and benchmark the performance of the new `QCDLoop` library in terms of computational time and then in terms of phenomenological results.

4.1 Performance tests

All the topologies implemented in `QCDLoop 2.0` have been validated successfully by direct comparison with `QCDLoop 1.96` [7] for configurations with real masses and `OneLoop 3.6` [10] for complex masses.

The performance benchmark is based on a common setup: all libraries compiled with `gcc-5.2.1` using `-O2` optimization flags on a `i7-6500U` CPU @ 2.50GHz. Kinematical configurations are constructed before the computation of the scalar integrals in order to avoid

copy-assignment operations during the computation loop. We use the native language of each library when performing the benchmark in order to avoid eventual overhead due to wrapper manipulation. We remove the initialization time of `OneLoop` from the results.

In Figure 3 we present four performance tests. In the upper left plot we compare the two versions of `QCDLoop`, *i.e.* the new `c++` library `QCDLoop 2.0` (red triangles) and the previous `fortran` library `QCDLoop 1.96` (yellow circles), to `OneLoop 3.6` (blue boxes). Computations are performed with disabled cache over 10^7 random configurations using real masses and double precision accuracy. Results are quoted in terms of average time in milliseconds for each topology and specific kinematics, following the notation of Table 1. The right inset shows the ratio to `QCDLoop 1.96` where we notice that, on average, the new library provides the best performance for tadpole, bubble and triangle integrals, but we obtain similar timings to `OneLoop` for box configurations. Overall we conclude that the new library provides an improvement in comparison to past releases, in particular when considering finite boxes computed in `QCDLoop 1.96` through the `ff` library. The upper right plot compares the LRU cache between the two versions of `QCDLoop`. We perform 10^7 trials of the same configuration for all topologies using real masses and double precision. The new library cache is 20 to 30% faster than the previous one for tadpoles, bubbles and triangles, meanwhile we observe a consistent speed-up of 70% for boxes. In the lower left plot we compare the performance of the new `QCDLoop` library when using the LRU cache with $N = 10$ in comparison to the direct computation of 10 configurations repeated 10^7 times. Results show a great performance improvement when using this caching mechanism. For some simple topologies like the tadpole the caching algorithm has similar performance in comparison to the direct computation, however when considering the most time consuming configurations, like BIN4, differences of a factor 40 are observed. Finally, in the lower right plot we compare the performance of the new `QCDLoop` library when computing results in double and quadruple precision. In this case the same configuration is repeated 10^7 times with double and quadruple variables. Differences are proportional to the complexity of the integral. We always observe a slowdown factor in the range 1.8 to 3 when using quadruple precision.

In Figure 4 we compare the performance of the new `QCDLoop` library when computing one-loop scalar integrals with real and complex masses. We observe a performance loss above 50% when activating complex masses for non finite integrals. Differences are smaller for finite triangles and boxes due to the fact that those implementations rely on complex objects for both mass types.

4.2 Phenomenological applications

In this section we test the new `QCDLoop` library in Monte Carlo environments. We first show results for real masses using simulations obtained with the `MCFM 7.0.1` [18,19] interface. We then test complex masses using interfaces to `Ninja` [20,21] and `GoSam` [22] in a `Sherpa 2.2.0` [23,24] simulation.

4.2.1 MCFM interface

The interface to the new `QCDLoop` library and `MCFM` is straightforward, thanks to the backward compatible `fortran` wrapper presented in Section 3.2.4. The only technical requirement in `MCFM` consists in editing the `makefile` and updating the links and paths to the new library.

Simulations are performed for the LHC setup at $\sqrt{s} = 13$ TeV, using NNPDF3.0 NLO [25] set of PDFs and the default parameters of the `MCFM 7.0.1` input card. Here we focus on the

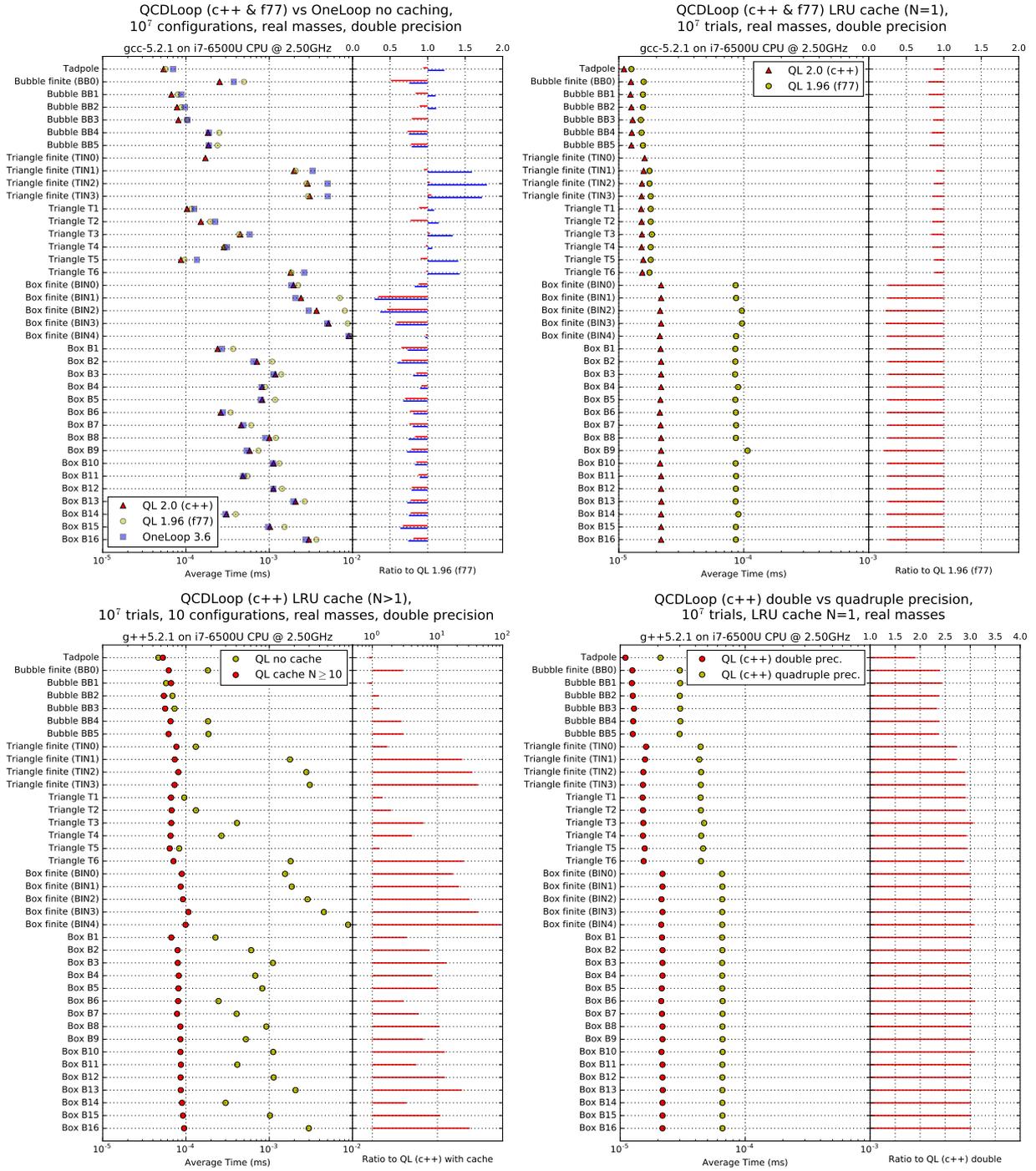


Figure 3: Performance comparisons. Upper left: QCDLoop 2.0 vs QCDLoop 1.96 vs OneLoop 3.6 performance for real masses and double precision for 10^7 configurations. Upper right: QCDLoop 2.0 vs QCDLoop 1.96 cache $N = 1$ with real masses and double precision, for 10^7 trials of the same configuration. Lower left: QCDLoop 2.0 cache $N > 1$ vs no cache for 10^7 trials of ten configurations. Lower right: QCDLoop 2.0 double vs quadruple precision performance for 10^7 trials of the same configuration.

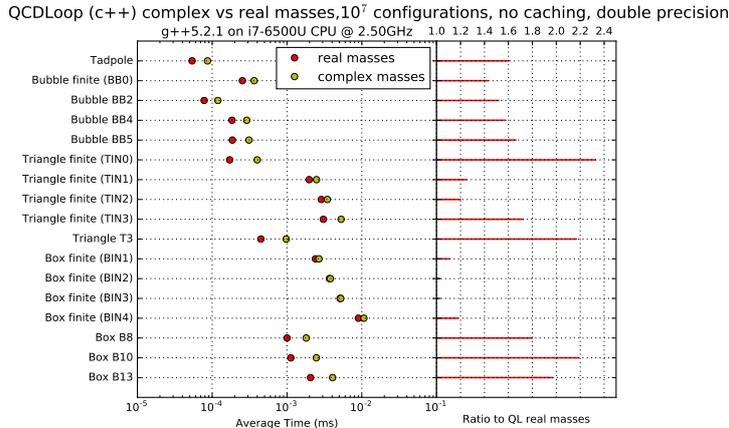


Figure 4: QCDLoop real vs complex masses performance.

predictions of three processes which cover a large range of topologies, including massive and massless loops, namely: WW ($nproc=61$), ZZ ($nproc=81$) and $\gamma\gamma\gamma\gamma$ ($nproc=289$) production. The aim of the results presented here is to show that fully compatible results are obtained when running the MC simulation with both versions of the library, *i.e.* QCDLoop 1.96 and 2.0.

In Figure 5 we plot the inclusive cross-section for WW , ZZ and $\gamma\gamma\gamma\gamma$ processes respectively for both versions of QCDLoop. Numerical results are in agreement for all processes. In terms of performance we observe 10% improvement with the new library for WW and ZZ production meanwhile 5% improvement for $\gamma\gamma\gamma\gamma$ production.

In Figure 6 we show differential distributions for the three processes described above, always comparing both versions of the QCDLoop library. In the top left panel we show the WW -pair transverse mass, m_T^{WW} , distribution. The top right panel presents the rapidity of the lepton-pair y_Z for the ZZ production. Finally, the bottom plot highlights the photon p_T^γ distribution in the $\gamma\gamma\gamma\gamma$ production. In all cases the agreement is very good; for the p_T^γ distribution we have performed a simulation of few hours (low statistics) in order to check that even with a low number of iterations the final agreement between both codes is excellent.

4.2.2 Ninja and GoSam interface

For the validation of complex masses in a Monte Carlo environment we considered the *Ninja* library [20, 21], which provides the integrand reduction via Laurent expansion method for the computation of one-loop integrals, and the *GoSam* [22] automated package.

First, we expanded the *Ninja* library with the new QCDLoop interface. We then verified the consistency of the new interface by comparing the output between *OneLoop* 3.6 and QCDLoop 2.0 for the examples provided by the *Ninja* test codes. Second, this new version of *Ninja* was linked to *GoSam* granting access of matrix elements to Monte Carlo simulation tools.

In order to provide quantitative results we used *Sherpa* 2.2.0 to simulate $H + 2j$ process at NLO, for the LHC setup at $\sqrt{s} = 13$ TeV with the NNPDF3.0 NLO PDF set. Such process is interesting for our tests because it calls several topologies with complex masses. We performed two simulations, the first with *OneLoop* and the second with QCDLoop. Table 3 shows the inclusive cross-section values obtained with both codes with 50M events, and Figure 7 presents the corresponding rapidity y_H and p_T^H distributions for the Higgs boson. This simulation shows

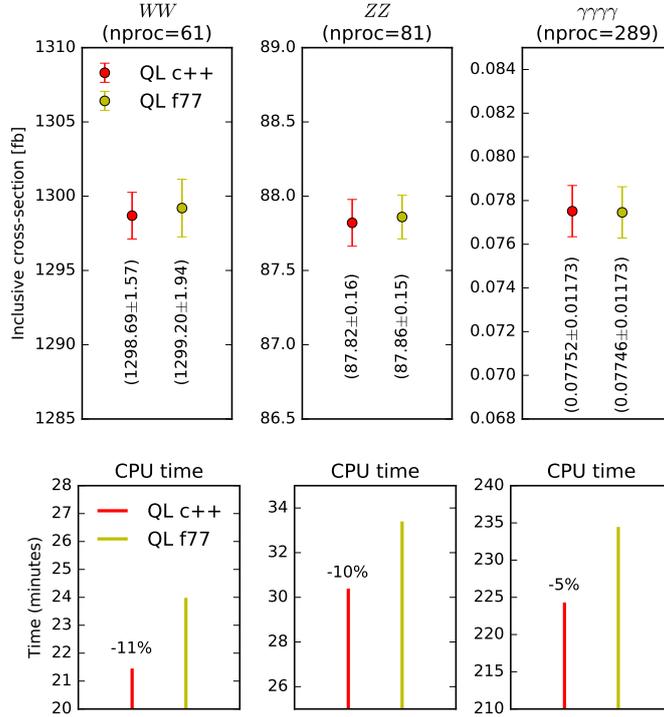


Figure 5: Examples of inclusive cross-sections obtained with MCFM 7.0.1 using both versions of QCDLoop for WW (nproc=61), ZZ (nproc=81) and $\gamma\gamma\gamma\gamma$ (nproc=289) production. Simulations performed for LHC @ 13 TeV, using NNPDF3.0 NLO.

that numerical results are in agreement for all distributions within Monte Carlo uncertainties. In terms of performance, both codes require ~ 10 CPU hours to complete the simulation.

Library	$H + 2j$ cross-section (Sherpa 2.2.0)
OneLoop 3.6	$5.5867 \pm (0.0121 = 0.21\%)$ pb
QCDLoop 2.0	$5.5838 \pm (0.0121 = 0.21\%)$ pb

Table 3: Inclusive cross-section for $H + 2j$ at NLO obtained with 5M events from Sherpa 2.2.0 interfaced with OneLoop 3.6 and QCDLoop 2.0 through the Ninja and GoSam interfaces.

Finally, other interfaces to Monte Carlo codes are possible, users are invited to interface their own code with QCDLoop.

5 Conclusions

In this work we presented a new object-oriented framework for the QCDLoop library. The new features compared with the fortran version are

- QCDLoop 2.0 calculates all integrals using native implementations. The reliance on the external library `ff` is no longer present.

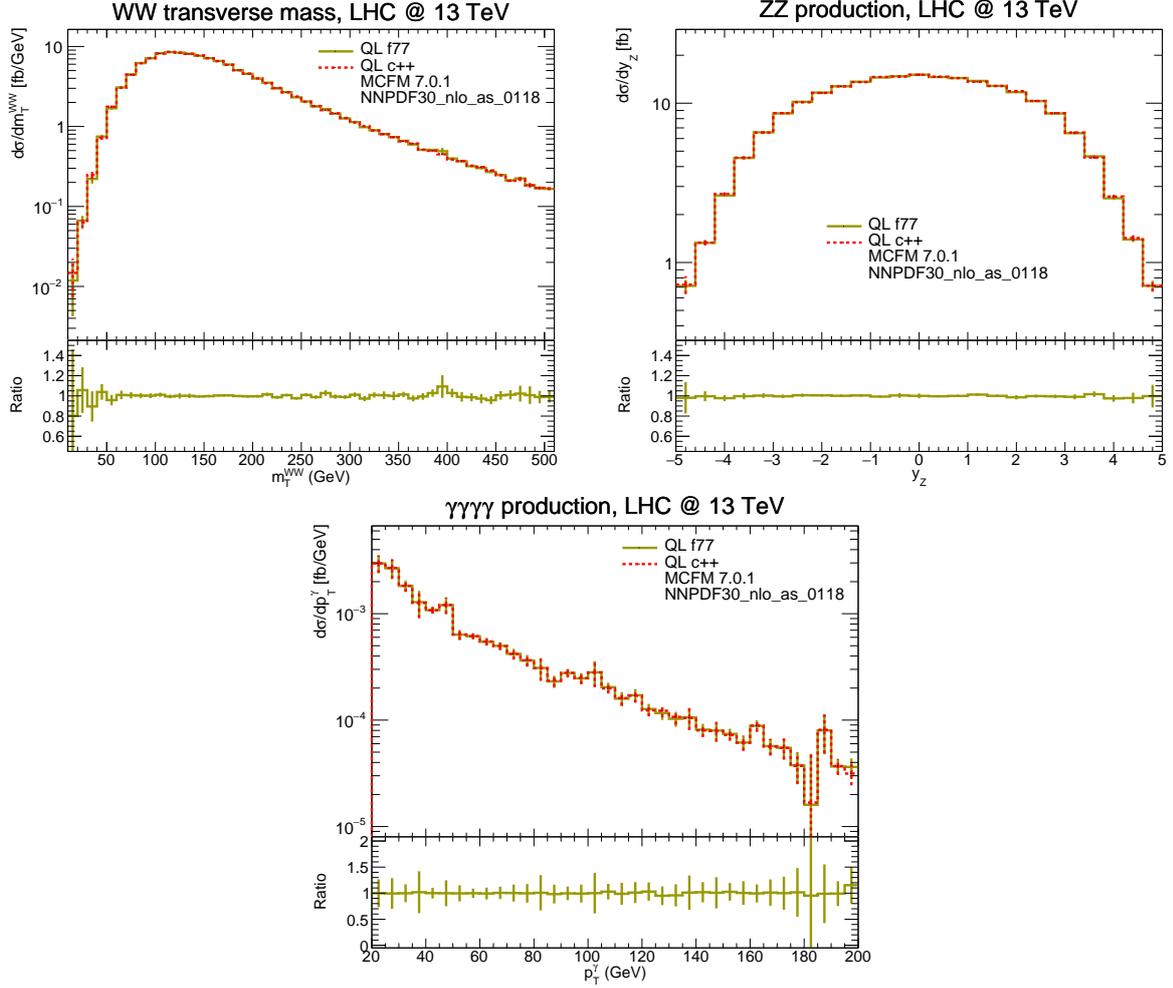


Figure 6: Examples of differential distributions obtained with MCFM 7.0.1 using both versions of QCDLoop, for the m_T^{WW} transverse mass in WW production, the average y_Z rapidity distribution in ZZ production and the average p_T^γ distribution in $4\text{-}\gamma$ production. Simulations performed for LHC @ 13 TeV, using NNPDF3.0 NLO.

- Full implementation of double and quadruple precision for the whole library, including the possibility of switching between the two dynamically. This can be useful in regions of phase space in which double precision is not sufficient. This can occur in corners of phase space, for example, in the context of NNLO calculations when unresolved regions of phase space are probed.
- Improvements in the evaluation time with respect to the Fortran version.
- Improvements in the caching algorithm. For certain applications, in which the same integrals are needed several times, this can lead to great improvements in evaluation time with respect to the Fortran version. In the new version one can adjust the size of the cache to yield the best performance.

The new QCDLoop library is publicly available from the webpage:

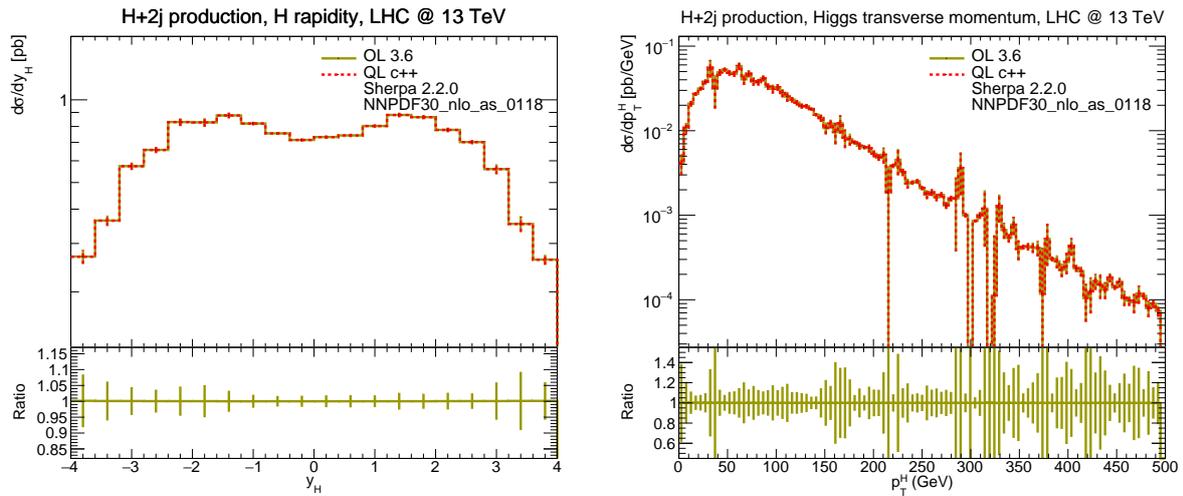


Figure 7: Examples of differential distribution obtained with Sherpa 2.2.0 using Oneloop and QCDLoop through the Ninja/GoSam interfaces. Simulations performed for LHC @ 13 TeV, using NNPDF3.0 NLO.

<http://cern.ch/qcdloop>

where instructions on how to install and run the code are also provided.

Acknowledgments

We thank Simone Alioli, Gavin Salam for interesting discussions about programming techniques, Fabrizio Caola for complex mass literature, Tiziano Peraro for discussions about the Ninja interface, Gionata Luisoni for the GoSam interface and the Sherpa setup presented in Sect. 4, and Ciaran Williams and Tobias Neumann for testing. S. C. and G. Z. are supported by the HICcup ERC Consolidator grant (614577). R. K. E. acknowledges support from the Alexander von Humboldt Foundation.

References

- [1] R.K. Ellis et al., Phys. Rept. 518 (2012) 141, 1105.4319.
- [2] J.R. Andersen et al., (2014), 1405.1067.
- [3] C.F. Berger et al., Phys. Rev. D78 (2008) 036003, 0803.4180.
- [4] G. Cullen et al., Eur. Phys. J. C72 (2012) 1889, 1111.2034.
- [5] F. Cascioli, P. Maierhofer and S. Pozzorini, Phys. Rev. Lett. 108 (2012) 111601, 1111.5206.
- [6] J. Alwall et al., JHEP 07 (2014) 079, 1405.0301.
- [7] R.K. Ellis and G. Zanderighi, JHEP 02 (2008) 002, 0712.1851.
- [8] G.J. van Oldenborgh, Comput. Phys. Commun. 66 (1991) 1.
- [9] T. Hahn and M. Rauch, Nucl. Phys. Proc. Suppl. 157 (2006) 236, hep-ph/0601248.
- [10] A. van Hameren, Comput. Phys. Commun. 182 (2011) 2427, 1007.4716.
- [11] A. Denner, S. Dittmaier and L. Hofer, (2016), 1604.06792.
- [12] A. Denner and S. Dittmaier, Nucl. Phys. Proc. Suppl. 160 (2006) 22, hep-ph/0605312.
- [13] A. Denner and S. Dittmaier, Nucl. Phys. B734 (2006) 62, hep-ph/0509141.
- [14] A. Denner, Fortsch. Phys. 41 (1993) 307, 0709.1075.
- [15] G. 't Hooft and M.J.G. Veltman, Nucl. Phys. B153 (1979) 365.
- [16] G.J. van Oldenborgh and J.A.M. Vermaseren, Z. Phys. C46 (1990) 425.
- [17] A. Denner, U. Nierste and R. Scharf, Nucl. Phys. B367 (1991) 637.
- [18] J.M. Campbell and R.K. Ellis, Phys. Rev. D62 (2000) 114012, hep-ph/0006304.
- [19] J.M. Campbell, H.B. Hartanto and C. Williams, JHEP 11 (2012) 162, 1208.0566.
- [20] P. Mastrolia, E. Mirabella and T. Peraro, JHEP 06 (2012) 095, 1203.0291, [Erratum: JHEP11,128(2012)].
- [21] T. Peraro, Comput. Phys. Commun. 185 (2014) 2771, 1403.1229.
- [22] H. van Deurzen et al., JHEP 03 (2014) 115, 1312.6678.
- [23] S. Höche and S. Prestel, Eur. Phys. J. C75 (2015) 461, 1506.05057.
- [24] S. Höche et al., Eur. Phys. J. C75 (2015) 135, 1412.6478.
- [25] NNPDF, R.D. Ball et al., JHEP 04 (2015) 040, 1410.8849.