A C++ interface to QCDNUM

V. Bertone*, M. Botje†
NIKHEF, Science Park, Amsterdam, the Netherlands

December 25, 2017

Abstract

In this document we report on the recent development of a C++ interface to the FORTRAN-based evolution program QCDNUM. A short description of the interface is given with a few basic examples of its usage.

1 Introduction

The QCDNUM program [1] numerically solves the DGLAP evolution equations in perturbative QCD up to next-to-next-to-leading order (NNLO) in the strong coupling α_s for unpolarised parton distribution functions (PDFs) and up to next-to-leading order (NLO) for polarised PDFs and fragmentation functions. QCDNUM also provides the users with a set of flexible tools for the fast computation of customised evolutions and physical observables. Based on these tools, and included in the QCDNUM distribution, are packages for the computation of the unpolarised deep-inelastic-scattering structure functions in the zero-mass scheme and heavy-flavour contributions to these structure functions in the massive scheme up to order α_s^2.

Thanks to its capabilities, QCDNUM has been employed in a large number of phenomenological studies, see for example the recent results of Refs. [2, 3, 4]. QCDNUM is currently interfaced to a number of codes routinely used for the analysis and the interpretation of experimental data, such as `xFitter` [5] and `Alpos` [6].

The QCDNUM code is written in FORTRAN77. This language usually guarantees an excellent performance, thanks to its simplicity and the static management of memory. Indeed, speed is one of the strongest points of QCDNUM that mostly derives from the use of FORTRAN77 but also from an optimal memory-access strategy made possible by an in-house dynamic memory manager. However, it is a fact that in today's large-scale HEP codes FORTRAN is abandoned in favour of object-oriented languages like C++ with its built-in dynamic memory allocation and support for modularity. It is therefore

*email valerio.bertone@cern.ch
†email m.botje@nikhef.nl

important that standard tools, such as PDF evolution codes, are not only available in FORTRAN but also in C++.

Instead of re-writing QCDNUM entirely in C++ (a huge effort which might yield degraded performance in terms of speed) we have decided to provide an interface that allows users to call the existing QCDNUM routines from a C++ program. In this note we present the C++ interface that has been made available in the beta release 17-01/14 which can be downloaded from the web-site given in Ref. [7]. The interface is also documented in the QCDNUM write-up.

2 The C++ interface

Since release 17-01/10, the QCDNUM library can be built using AUTOTOOLS [8] which is a standard suite of tools to make source code packages portable on different platforms. AUTOTOOLS provides some support to facilitate the simultaneous use of multiple languages, such as the automatic selection of the linker and its flags and macros to convert names of C++ identifiers into the FORTRAN format.

This allowed us to write C++ wrappers to the original FORTRAN routines so that, from version 17-01/14 onward,[1] it is possible to call QCDNUM from a C++ code, provided that the library is built with AUTOTOOLS, and not locally with the `makelib` script.

```
bash> cd qcdnum-17-01-14
bash> ./configure --prefix=<installation-dir>
bash>  make
bash>  make install
```

The QCDNUM includes, library and the `qcdnum-config` script can then be found in, respectively, the sub-directories `/include`, `/lib` and `/bin` of `<installation-dir>`. To make these accessible from everywhere, care should be taken to properly set-up the search paths, for instance on OS-X,

```
bash> export PATH=$PATH:<installation-dir>/bin
bash> export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:<installation-dir>/lib
```

Now one can run test programs that are stored in the `testjobsCxx` folder.

```
bash> cd qcdnum-17-01-14/run
bash> ./runtest exampleCxx         (without extension .cc)
```

The program `exampleCxx.cc` evolves the set of unpolarised parton densities at NNLO in the variable flavour number scheme (VFNS) and prints the value of the charm sea $2x\bar{c}$ at some x and μ^2, as well as the value of $\alpha_s(m_Z^2)$. The output should look like this.

[1] At present, wrappers are available for the out-of-the-box routines listed in Table 2 of the QCDNUM write-up, except that the steering by data cards is only possible in FORTRAN. Also available are wrappers for the ZMSTF and HQSTF structure function packages. Those for the QCDNUM toolbox (Table 4) are under development and their validation will be announced on the QCDNUM web site and mailing list.

A C++ interface to the toolbox is available from QCDNUM-18 onward

```
x, q, CharmSea = 1.0000e-03  1.0000e+03  1.8708e+00
as(mz2)        = 1.1807e-01
```

3 Example code

To illustrate how the interface is used in practice, we show in Figure 1 a listing of the `exampleCxx` program. For the sake of clarity we have omitted some include directives, type declarations and variable initialisations; for a full listing the reader is referred to the code stored in the `testjobsCxx` directory of the QCDNUM distribution. We will not describe here the function calls in detail (see the QCDNUM write-up for this) but present below some features of the interface that are important to know.

1. Unlike FORTRAN, C++ code is case-sensitive. We have therefore adopted the convention that the wrappers will have the same name (and argument list) as their FORTRAN counterparts, but written in lower case. Furthermore, to avoid possible name-conflicts with other codes, all the wrappers are assigned to the namespace `QCDNUM`. We thus have

```
call SUB(arguments)    →    QCDNUM::sub(arguments);
```

2. In the FORTRAN description of the QCDNUM routines we use the implicit type declaration that, unless otherwise stated, all variables are in `double precision` except those with names starting with the letter `I`–`N` (or `i`–`n`) which are of type `integer`. In C++ each variable must be explicitly typed, for example,

```
implicit double precision (a-h,o-z)        double x;
ix = IXFRMX(x)                        →    int ix = QCDNUM::ixfrmx(x);
```

3. The wrapper for logical functions returns an `int` and not a `bool`.[2]

```
logical gridpoint                    double x; int ix;
gridpoint = XXATIX(x,ix)    →        int gridpoint = QCDNUM::xxatix(x,ix);
if(gridpoint) then ...               if(gridpoint) { ...
```

4. The type of character input arguments should be `string`. String literals are delimited by double quotes in C++ and by single quotes in standard FORTRAN77.

```
character*50 file
file = 'example.log'                 string file = "example.log";
call QCINIT(20,file)          →      QCDNUM::qcinit(20,file);
call SETVAL('Alim',5.0D0)            QCDNUM::setval("Alim",5);
```

[2] In C++ any nonzero (zero) value evaluates as true (false) in logical expressions.

```
  ...
#include "QCDNUM/QCDNUM.h"
  ...

double func(int* ipdf, double* x) {
  int i = *ipdf;
  double xb = *x;
  double f = 0;
  if(i ==  0) f = xglu(xb);
  if(i ==  1) f = xdnv(xb);
  if(i ==  2) f = xupv(xb);
  ...
  return f;
  }

int main() {
  ...
  double def[] =                          //input flavour composition
  // tb  bb  cb  sb  ub  db   g   d   u   s   c   b   t
    { 0., 0., 0., 0., 0.,-1., 0., 1., 0., 0., 0., 0.,   // 1=dval
      0., 0., 0., 0.,-1., 0., 0., 0., 1., 0., 0., 0.,   // 2=uval
      0., 0., 0.,-1., 0., 0., 0., 0., 0., 1., 0., 0.,   // 3=sval
      0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0.,   // 4=dbar
      0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0.,   // 5=ubar
      0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0.,   // 6=sbar
      ...                                                  // more not shown
      0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.};  //12=zero
  ...
  QCDNUM::qcinit(lun,outfile);                    //initialise
  QCDNUM::gxmake(xmin,iwt,ng,nxin,nx,iosp);       //x-grid
  QCDNUM::gqmake(qq,wt,2,nqin,nq);                //mu2-grid
  QCDNUM::setcbt(nfin,iqc,iqb,999);               //set VFNS thresholds
  QCDNUM::fillwt(ityp,id1,id2,nw);                //compute weights
  QCDNUM::setord(iord);                           //LO, NLO, NNLO
  QCDNUM::setalf(as0,r20);                        //input alphas
  QCDNUM::evolfg(ityp,func,def,iq0,eps);          //evolve all pdf's
  QCDNUM::allfxq(ityp,x,q,pdf,0,1);               //interpolate all pdf's

  double csea = 2 * pdf[2];                       //charm sea at x,mu2
  double asmz = QCDNUM::asfunc(qmz2,nfout,ierr); //alphas(mz2)

  cout << scientific << setprecision(4);
  cout << "x, q, CharmSea = " << x << "  " << q << "  " << csea << endl;
  cout << "as(mz2)        = " << asmz << endl;

  return 0;
  }
```

Figure 1 – A QCDNUM example program in C++. Only the relevant parts of the code are shown.

5. In FORTRAN an array index starts at one, unless the index range is specified as in `pdf(-6:6)`, for example. Here the gluon has index `0` and the (anti)quarks have a (negative) positive index according to the PDG convention [9]. However, this is not possible in C++ where arrays always start with index zero. Thus one should account for index shifts between the FORTRAN and C++ arrays as is shown below.

```
dimension pdf(-6:6)                    double x, q, pdf[13];
call ALLFXQ(1,x,q,pdf,0,1)      →      QCDNUM::allfxq(1,x,q,pdf,0,1);
gluon = pdf(0)                         double gluon = pdf[6];
```

6. Two-dimensional arrays in FORTRAN become one-dimensional arrays in the C++ wrappers. This can best be handled by providing a pointer function $k(i,j)$ that maps the indices of a FORTRAN array `A(n,m)` onto those of a C++ array `A[n*m]` such that $k(i+1,j) = k(i,j)+1$, $k(i,j+1) = k(i,j)+n$ and $k(1,1) = 0$.

```
inline int kij(int i, int j, int n) { return i-1 + n*(j-1); }
```

Here is an example of how to use such a pointer.

```
dimension c(-6:6),x(8),q(5),f(8,5)     double c[13],x[8],q[5],f[8*5];
call FTABLE(1,c,0,x,8,q,5,f,1)    →    QCDNUM::ftable(1,c,0,x,8,q,5,f,1);
fij = f(i,j)                           double fij = f[ kij(i,j,8) ];
```

7. Particular care has to be taken when passing functions as arguments. An example is the evolution routine `evolfg` where the PDF values $f_i(x)$ at the input scale μ_0^2 are, in FORTRAN, entered via the user-defined function `func(i,x)` which should be declared `external` in the calling routine. To the best of our understanding this can only be ported to C++ if the corresponding function has its input arguments passed as pointers, as is shown for the input function `func` in the listing of Figure 1.

This is then more or less all one needs to know about the C++ interface to QCDNUM.

4 Acknowledgements

We are grateful to the `xFitter` developers team, particularly to Ringaile Plačakytė, Voica Radescu, and Sasha Glazov who have encouraged the development of the interface. V. B. is supported by the European Research Council Starting Grant 'PDF4BSM'.

References

[1] M. Botje, Comput. Phys. Commun. **182**, 490 (2011), arXiv:1005.1481.

[2] A. Vafaee and A. Khorramian, Eur. Phys. J. **A53**, 220 (2017), arXiv:1711.06573 [hep-ph].

[3] V. Andreev *et al.* [H1 Collaboration], arXiv:1709.07251 [hep-ex].

[4] F. Hautmann *et al.*, arXiv:1708.03279 [hep-ph].

[5] S. Alekhin *et al.*, Eur. Phys. J. **C75**, 304 (2015), arXiv:1410.4412 [hep-ph].

[6] D. Britzger *et al.*, unpublished, `http://www.desy.de/~ britzger/alpos`.

[7] `https://www.nikhef.nl/~h24/qcdnum`.

[8] `https://www.gnu.org/software/autoconf/manual/autoconf.html`

[9] `http://pdg.lbl.gov/2007/reviews/montecarlorpp.pdf`.