Nikhef

QCDNUM Status and Plans

Michiel Botje

Nikhef, Amsterdam

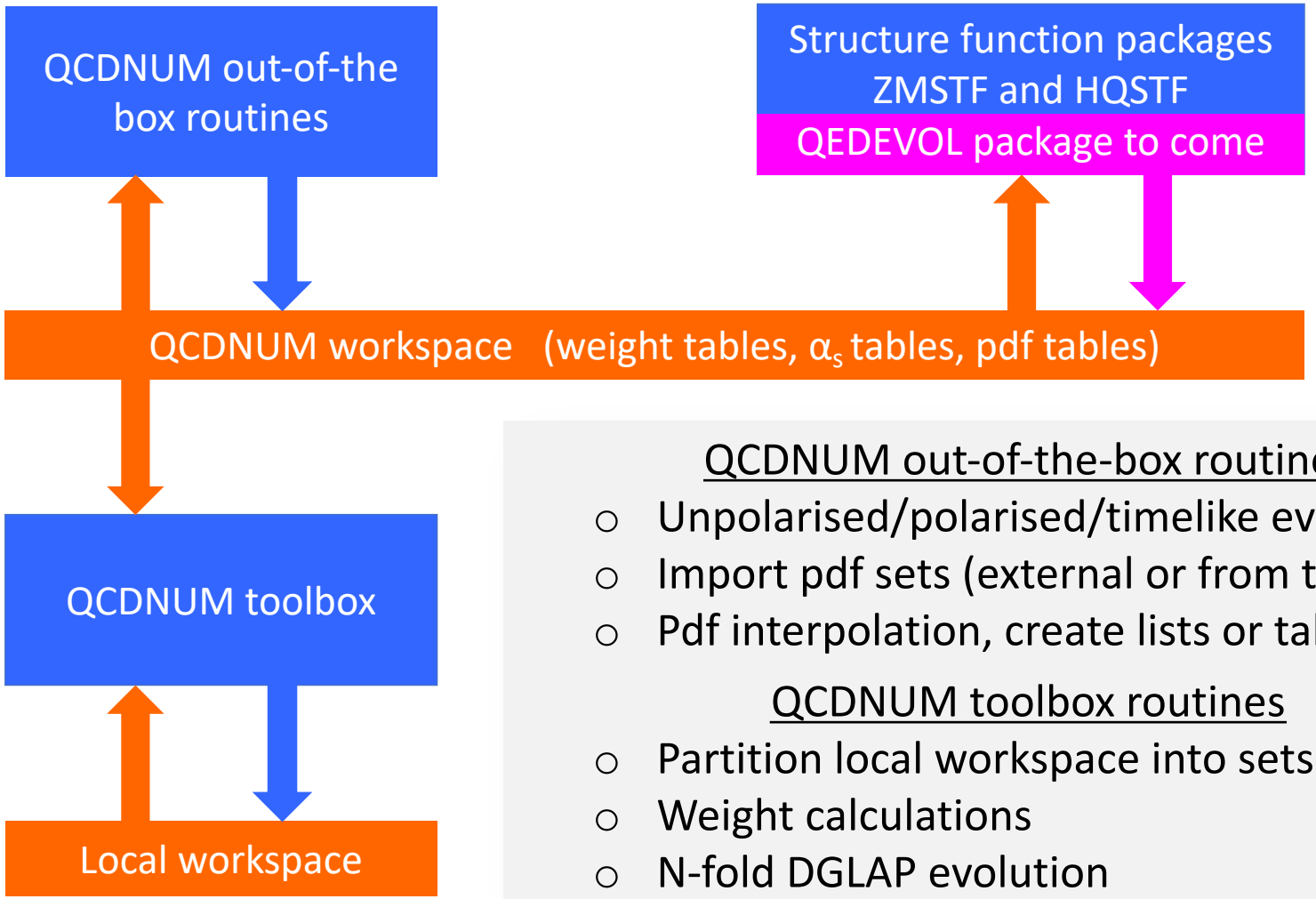
xFitter external workshop

DESY February 27, 2020

QCDNUM releases

- 17-01/15: Released March 17, 2019
  - Out-of-the-box evolution routine with intrinsic heavy flavours
  - New out-of-the-box singlet/non-singlet evolution routine
  - New routine to set cuts in the kinematic plane
  - More flexibility in setting thresholds
  - Evolution start scale can be anywhere in μ^2
  - Pdf access not anymore restricted to those with current parameters
- 17-01/15: Update October 31, 2019
  - Few minor fixes

QCDNUM program structure



QCDNUM out-of-the-box routines

- Unpolarised/polarised/timelike evolution
- Import pdf sets (external or from toolbox)
- Pdf interpolation, create lists or tables

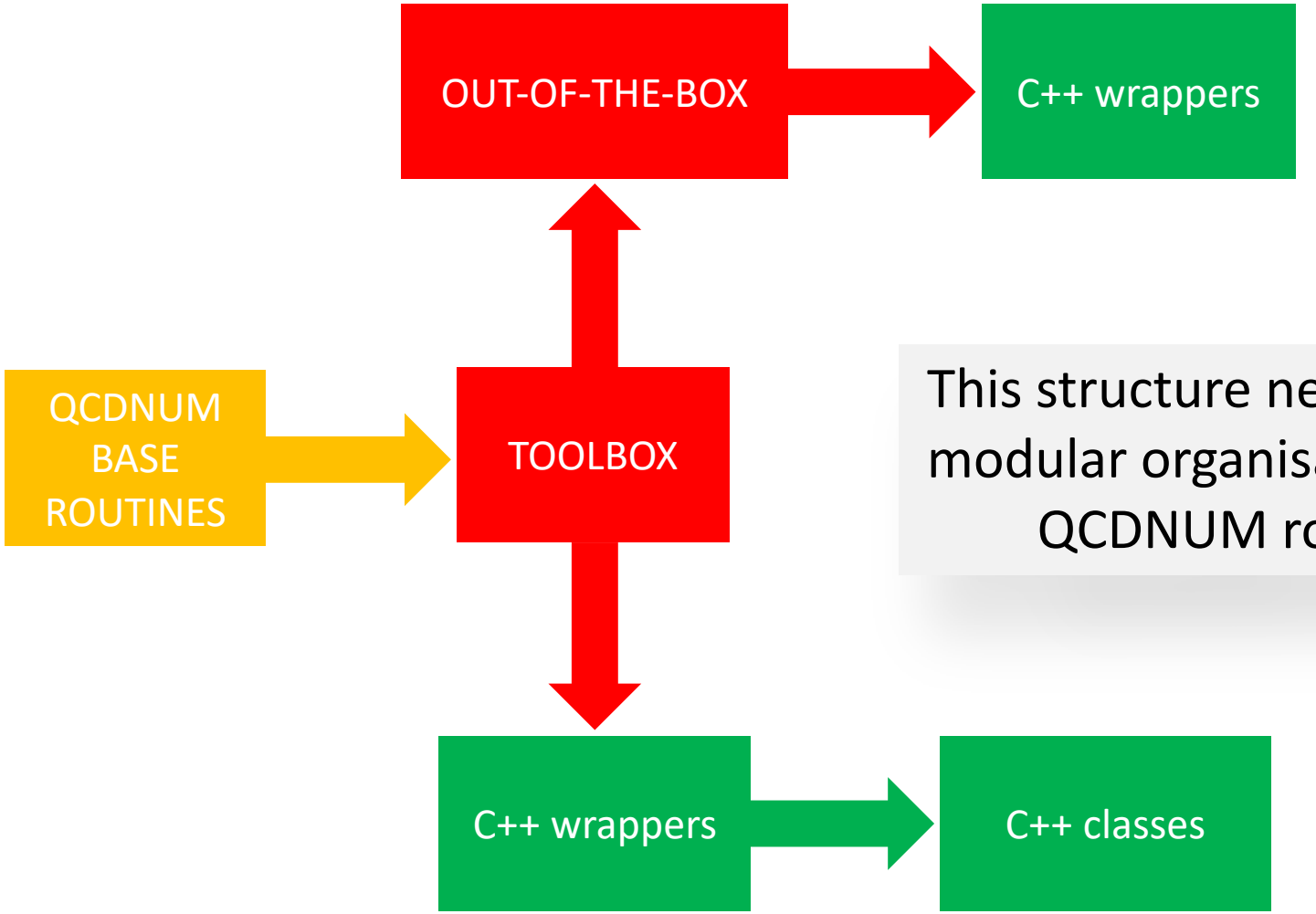
QCDNUM toolbox routines

- Partition local workspace into sets of tables
- Weight calculations
- N-fold DGLAP evolution
- Convolution tools, fast convolution engine

Why upgrade QCDNUM ?

- The QCDNUM code is written in Fortran77
- Its key feature is an in-house dynamic memory manager that eliminates the use of multi-dimensional Fortran arrays
- This in-house memory management allows for very fast code
- The memory manager has been developed over many years and starts to hinder the maintenance of QCDNUM
- A way-out could be to abandon F77 and goto C++ dynamic memory
- My choice is to re-vamp the manager and provide a C++ interface to it
- Modularisation/encapsulation in object-oriented style will make code maintenance much easier, also by other people than me

Long-term program structure



This branch exists

This structure needs a more modular organisation of the QCDNUM routines

This branch does not exist

Modular organisation of the code

- A module is defined as a table-set in a workspace, together with associated routines that manage this table-set (create object, setters, getters, …)
- A module in Fortran is just the equivalent of a class in C++
- The list of module types (classes) in QCDNUM is not very large:
  - Memory manager
  - x-grid
  - μ^2-grid
  - Convolution weight tables
  - Evolution parameter tables
  - α_s tables
  - Pdf tables
- Everything will be stored in modules $\Longrightarrow$ get rid of Fortran common blocks
- Fortran: all modules reside in one large workspace (allocated at compilation)
- C++: each module sits in a separate workspace (allocated dynamically)

First step in the symbiosis of Fortran and C++

```
C-----------------------------------------------------------------------
CXXHDR    int imb_hdsize();
C-----------------------------------------------------------------------
CXXHFW  #define fimb_hdsize FC_FUNC(imb_hdsize,IMB_HDSIZE)
CXXHFW    int fimb_hdsize();
C-----------------------------------------------------------------------
CXXWRP  //-----------------------------------------------------------------
CXXWRP    int imb_hdsize()
CXXWRP    {
CXXWRP     return fimb_hdsize();
CXXWRP    }
C-----------------------------------------------------------------------
```

C++

```
C     ============================
      integer function imb_HdSize()
C     ============================

C--   Return header size

C--   Author: Michiel Botje h24@nikhef.nl   02-12-19

      implicit double precision (a-h,o-z)

      include 'wspace0.inc'

      imb_HdSize = nwHeader0

      return
      end
```

Fortran

- Fortran code and C++ wrapper code now sit together in one file
- Release script extracts the C++ code and puts it into a directory in the release tree
- Maintenance godsend

Most basic ingredient: memory module

- ✓ Routine to convert 1-dim double precision array into a workspace (formatting)
- ✓ Routines to create table-sets and populate them with one or more n-dim tables
- ✓ Routines to clone, copy, disk dump and read tables and table-sets
- ✓ Object fingerprinting (equal fingerprint = equal object structure)
- ✓ Easy and fast navigation through linked-list structure
- ✓ Each object has a tag-field to store attributes
- ✓ Can build object hierarchies by storing addresses (pointers) in the tag-fields
- ✓ Hooks to create very fast iterators and address functions

Memory module exists (in MBUTIL) with full documentation …

… and a C++ interface …

7 Workspaces ✓

The C++ prototypes of these routines (without the scope resolution operator `MBUTIL::`) are:

```
int iaddr  = imb_wsinit(double *w, int nw, int nt)
void         smb_setwsn(double *w, int nw)
int iaddr  = imb_wtable(double *w, int *imin, int *imax, int ndim)
int iaddr  = imb_newset(double *w)
int iaddr  = imb_wclone(double *obj1, double *w2)
void         smb_tbcopy(double *table1, double *table2, int itag)
void         smb_tsdump(string fname, int key, double *tbset, int &ierr)
int iaddr  = imb_tsread(string fname, int key, double *w, int &ierr)
int marker = imb_marker(string otype)
int nwords = imb_tbsize(int *imin, int *imax, int ndim)
int nwords = imb_hdsize()
```

✓

… and code examples in Fortran and C++

```
integer function iP3(w, ia, i, j, k)
double precision w(*)
dimension kk(5)
save kk
ifp = imb_FingerPrint(w,ia)
if(kk(1).ne.ifp) call K3(w, ia, kk)
ip  = kk(2)+kk(3)*i+kk(4)*j+kk(5)*k
iP3 = ia + ip
return
end
```

```
int iP3(double *tb, int i, int j, int k) {
static int kk[5];
int ifp = imb_FingerPrint(tb);
if( kk[0] != ifp ) { K3( tb, kk ); }
int ip  = kk[1]+kk[2]*i+kk[3]*j+kk[4]*k;
int ia  = int(*(tb+1));
return ia + ip;
}
```

✓

Presently I am trying to encapsulate the C++ interface into a C++ class that creates and manages an arbitrary set of tables in a dynamic C++ array

Up to now it all seems to work ✓

Whats Next

- Freeze the current status in a new release qcdnum-17-01-16
- The QCDNUM part of 16 will be the same as 15, except that a few compiler complaints are fixed as a bonus
- I am currently writing the workspace C++ class
- This class serves as a proof of principle but is not essential (basic memory management will be encapsulated anyway)
- Next step is then to code the grid modules
- Will keep a keen eye on thread support (my dream) via OpenMP
- Let me know if new functionality is needed beyond version 15