MCFM ntuple patch

Ivan Pogrebnyak, MSU

July 29, 2020

1 Introduction

The ability to save events, produced using a Monte Carlo generator, provides much greater flexibility in comparison to saving specific histograms internally populated by the events. Saving ntuples of generated events allows to adjust cuts, add new histograms, or change binning of already defined histograms without the need to rerun the event generator. Ntuples also allow users to utilize analysis software independent from the MC generator program. Having an intermediate format for event output allows for analysis code to be written in any desired programming language. Consequently, more analysis code, in whole or in part, may be reused from previous studies.

Unfortunately, `MCFM` has been difficult to setup for ntuple output, and the newer versions[*] saw the feature removed entirely. Here, I present an easy patch, that can be applied to any `MCFM` version, to allow outputting ntuples. As it is listed here, the code allows to write ntuples in `ROOT`[**] format. But any other format can be added with a few lines of code, as the patch provides a generic API.

The main feature of the patch is the decoupling of the code that formats and writes the ntuples from its interface. This is achieved by compiling the writing code into a shared library, containing functions with specific signatures, which comprise the API. The shared library is dynamically loaded using `libdl`, the standard Linux dynamic linking library. The shared library can be independently recompiled, if changes need to be made to the ntuple writing code or its dependencies, such as the installed version of `ROOT`, have changed. The output format can also be changed without recompiling `MCFM`.

[*] 10.1007/JHEP12(2019)034, arXiv: 1909.09117.
[**] 10.1016/j.cpc.2009.08.005, `http://root.cern.ch/`.

2 Code structure

The API functions have the following equivalent C declarations:

```
void* open_ntuple(const char* file_name)
void close_ntuple(void* ntuple)
void fill_ntuple(void* ntuple)
void add_ntuple_branch_double(void* ntuple, const char* name, double* ptr)
void add_ntuple_branch_float(void* ntuple, const char* name, float* ptr)
void add_ntuple_branch_int(void* ntuple, const char* name, int* ptr)
```

The `void* ntuple` is returned by `open_ntuple()` and needs to be passed to the other functions. It is a pointer to some implementation specific structure that represents the ntuple writer.

`add_ntuple_branch_.*()` functions are used to pass pointers to variables that will contain the values of ntuple branches. The program using the interface needs to allocate these variables and overwrite them for every event. Once the values of all the variables are set, `fill_ntuple()` needs to be called to add the event comprised of these values to the ntuple. This is similar to how trees are written in ROOT.

Once all the events have been written, `close_ntuple()` should be called to finish writing the ntuple file and to free the memory associated with the writer structure.

For ntuple output in ROOT format, these functions are defined in the `root_ntuples.cc` file, written in C++. This part cannot be written in Fortran, because ROOT does not provide Fortran bindings. Essentially, the reason for the particular structure of this patch is to allow this part to be written in an arbitrary language with arbitrary dependencies. The implementation is abstracted by encapsulating it in a shared library. In order to load the library and call the API functions from Fortran, the interface file, `ntuple_interface_dl.c`, written in C, needs to be compiled with MCFM. This part could probably be written in Fortran, but that would not add any tangible benefit, and it is easier to do in C. Finally, the MCFM specific Fortran interface is provided in the file `ntuple_interface.f`, which is a drop-in replacement for `src/User/mcfm_froot.f`.

3 How to apply the patch

Here are instructions on how to apply the patch to an existing MCFM source code. I use MCFM-8.0 as the specific example, but the steps should be very similar for all relatively recent versions.

1. Get the patch from https://github.com/ivankp/mcfm_ntuple_patch.

   ```
   cd /path/to/mcfm
   git clone https://github.com/ivankp/mcfm_ntuple_patch.git
   ```

2. Compile the shared library.

```
cd mcfm_ntuple_patch
make
```

3. Link (or copy) files to `src/User/`.

```
cd ../src/User
ln -s ../../ntuple/ntuple_interface_dl.c
ln -s ../../ntuple/ntuple_interface.f
cd ../..
```

4. Edit the `MCFM` `makefile` to compile the added files.

   Change the definition of the `USERFILES` variable:

```
# Check NTUPLES flag
ifeq ($(NTUPLES),FROOT)
  # USERFILES += mcfm_froot.o froot.co # <-- comment this line out
  USERFILES += ntuple_interface_dl.o ntuple_interface.o # <-- add this line
```

   Add a rule for `.c` files:

```
%.o: %.c
	gcc -Wall -O3 -c $< -o $(OBJNAME)/$@
```

   It can be placed near line 2376, below the comment *# for FROOT package*, but it doesn't really matter where.

5. Add an extra clause in `src/Need/mcfm_exit.f` near the end.

```
      call NTfinalize
  endif
else ! add this line
  call NTfinalize ! add this line
endif
```

6. `MCFM` can now be recompiled and run.

7. In order to run with the new ntuple output, a link to (or copy of) the library compiled in step 2 needs to be present in the run directory, and has to be called `ntuples.so`.

4 Outlook

If this patch is to be merged into a future release of `MCFM`, a few things need to be changed or added to fully incorporate it.

1. The ntuples shared library can be made to compile together with `MCFM`, if `ROOT` is installed.

2. Right now, the path to the ntuples shared library is hardcoded in `ntuple_interface.f` to look for an `ntuples.so` file in the current directory, i.e. the directory where `MCFM` is run.

   ```
   call load_ntuple_lib("./ntuples.so"//C_NULL_CHAR)
   ```

   The current approach can be retained for its flexibility. But this requires a link to the library to be present in the run directory. This should be documented.

   Alternatively, the default path could be to some directory where the shared library will be put after it is compiled.

   An optional parameter can be added to the runcard to specify where to look for the library in case the user wants to use a custom one. This option can be added in either case. The option should specify the path to the `.so` file rather then to a directory, for greater flexibility.

3. The `Fortran` interface file, `ntuple_interface.f`, can be cleaned up. I wrote it to imitate `src/User/mcfm_froot.f`, including dummy subroutines. This may not be necessary in `MCFM-9`.

5 Source code listings

5.1 root_ntuples.cc

```
#include <TFile.h>
#include <TTree.h>
#include <cstdlib>

extern "C"
void* open_ntuple(const char* file_name) {
  TFile* file = new TFile(file_name,"RECREATE");
  if (file->IsZombie()) exit(1);
  return new TTree("ntuple","");
}

extern "C"
void close_ntuple(void* ntuple) {
  TFile* file = reinterpret_cast<TFile*>(
    reinterpret_cast<TTree*>(ntuple)->GetDirectory());
  file->Write(0,TObject::kOverwrite);
  delete file;
}

extern "C"
void fill_ntuple(void* ntuple) {
  reinterpret_cast<TTree*>(ntuple)->Fill();
}

#define add_ntuple_branch(TYPE) \
extern "C" \
void add_ntuple_branch_##TYPE(void* ntuple, const char* name, TYPE* ptr) { \
  reinterpret_cast<TTree*>(ntuple)->Branch(name,ptr); \
}

add_ntuple_branch(double)
add_ntuple_branch(float)
add_ntuple_branch(int)
```

5.2 ntuple_interface_dl.c

```
#include <dlfcn.h>
#include <stdio.h>
#include <stdlib.h>

#define STR1(x) #x
#define STR(x) STR1(x)

static void* lib;

static void*(*f_open_ntuple)(char*);
static void(*f_close_ntuple)(void*);
static void(*f_fill_ntuple)(void*);

```

```
#define add_ntuple_branch(TYPE) \
static void(*f_add_ntuple_branch_##TYPE)(void*, char*, TYPE*);

add_ntuple_branch(double)
add_ntuple_branch(float)
add_ntuple_branch(int)

void load_ntuple_lib_(char* file_name) {
  if ((lib = dlopen(file_name,RTLD_LAZY)) == 0) {
    fflush(stdout);
    fprintf(stderr,"\nfailed to load %s\n",file_name);
    exit(1);
  }
  f_open_ntuple = dlsym(lib,"open_ntuple");
  f_close_ntuple = dlsym(lib,"close_ntuple");
  f_fill_ntuple = dlsym(lib,"fill_ntuple");

#undef add_ntuple_branch
#define add_ntuple_branch(TYPE) \
  f_add_ntuple_branch_##TYPE = dlsym(lib,STR(add_ntuple_branch_##TYPE));

  add_ntuple_branch(double)
  add_ntuple_branch(float)
  add_ntuple_branch(int)
}

void* open_ntuple_(char* file_name) {
  printf("opening ntuple file \"%s\"\n",file_name);
  void* ptr = f_open_ntuple(file_name);
  printf("%p\n",ptr);
  return ptr;
}

void close_ntuple_(void** ntuple) {
  printf("closing ntuple\n");
  fflush(stdout);
  return f_close_ntuple(*ntuple);
}

void fill_ntuple_(void** ntuple) {
  return f_fill_ntuple(*ntuple);
}

#undef add_ntuple_branch
#define add_ntuple_branch(TYPE) \
void add_ntuple_branch_##TYPE##_(void** ntuple, char* name, TYPE* ptr) { \
  printf("%p Adding branch %s\n",*ntuple,name); \
  f_add_ntuple_branch_##TYPE(*ntuple,name,ptr); \
}

add_ntuple_branch(double)
add_ntuple_branch(float)
add_ntuple_branch(int)
```

5.3 ntuple_interface.f

```
!--- J. Campbell, June 25th, 2008.
!--- Ivan Pogrebnyak, July 2020

!--- Generic C interface for ntuple output for MCFM
!--- In particular, allowing easy interface with any ROOT version

      subroutine bookfill(tag,p,wt) ! This routine is called by nplotter
        implicit none
        include 'types.f'
        include 'mxpart.f'
        include 'maxwt.f'

        integer  :: tag
        real(dp) :: p(mxpart,4)
        real(dp) :: wt

        ! TODO: something is not intitialized here

        if (.not.skipnt) then
          if (tag == 1) then
            call mcfm_ntuple_book
          else if (tag == 2) then
            call mcfm_ntuple_fill(p,wt)
          endif
        endif
      end subroutine

      subroutine NTfinalize
        ! This routine is called at the end of the program's execution;
        ! it should finalize the output and close opened files, if necessary
        use iso_c_binding
        implicit none

        real(c_double) pfill(105)
        type(c_ptr) :: ntuple
        common/ntuple_interface/ntuple,pfill

        write(*,*) " * * * finalizing"

        call close_ntuple(ntuple)
      end subroutine

      subroutine mcfm_ntuple_book
        use iso_c_binding
        implicit none
        include 'types.f'
        include 'npart.f'
        include 'mxdim.f'
        include 'scale.f'
        include 'facscale.f'
        include 'PDFerrors.f'
        include 'kpart.f'

        ! Extra definitions to facilitate dummy call to lowint
```

```
      real(dp):: scale_store, facscale_store
      real(dp):: dummy, wgt, r(mxdim), lowint
      integer:: i, imaxmom, ipdf
      common/iarray/imaxmom,ipdf
      integer:: lenocc
      character(len=255) :: runname
      common/runname/runname

      ! assume at most 10 final state particles and 60 PDF sets
      real(c_double) pfill(105)
      type(c_ptr) :: ntuple
      common/ntuple_interface/ntuple,pfill
      type(c_ptr) :: open_ntuple
      character(len=16) :: branch_name

      logical:: first
      data first/.true./
      save first

      ! Need to ascertain the correct size for momenta n-tuples when this routine
      ! is called for the first time, achieved via a dummy call to lowint
      if (first) then
        do i = 1, mxdim
          r(i) = 0.5_dp
        end do
        ! Be careful that dynamic scale choices aren't ruined
        ! (in versions 5.1 and before, this occured when calling lowint)
        scale_store = scale
        facscale_store = facscale
        dummy = lowint(r,wgt)
        scale = scale_store
        facscale = facscale_store

        imaxmom = npart
        if ((kpart == kreal).or.(kpart == ktota).or.(kpart == ktodk)) then
          imaxmom = imaxmom+1
        endif

        first = .false.
      endif

      ! determine if we need space in array to store PDF weights (ipdf)
      ipdf = 0
      if (PDFerrors) then
        ipdf = maxPDFsets
      endif

      ! open ntuple file
      call load_ntuple_lib("./ntuples.so"//C_NULL_CHAR)
      ntuple = open_ntuple(runname(1:lenocc(runname))//".root"//C_NULL_CHAR)

      ! create ntuple branches
      do i = 0, imaxmom-1
        write(branch_name,"(A2I1A)") "px", i+3, C_NULL_CHAR
        call add_ntuple_branch_double(ntuple,branch_name,pfill(i*4+1))
        write(branch_name,"(A2I1A)") "py", i+3, C_NULL_CHAR
```

```
111             call add_ntuple_branch_double(ntuple,branch_name,pfill(i*4+2))
112             write(branch_name,"(A2I1A)") "pz", i+3, C_NULL_CHAR
113             call add_ntuple_branch_double(ntuple,branch_name,pfill(i*4+3))
114             write(branch_name,"(A2I1A)") "E_", i+3, C_NULL_CHAR
115             call add_ntuple_branch_double(ntuple,branch_name,pfill(i*4+4))
116           end do
117
118           call add_ntuple_branch_double(
119          &    ntuple, 'wt_ALL'//C_NULL_CHAR, pfill(imaxmom*4+1))
120           call add_ntuple_branch_double(
121          &    ntuple, 'wt_gg' //C_NULL_CHAR, pfill(imaxmom*4+2))
122           call add_ntuple_branch_double(
123          &    ntuple, 'wt_gq' //C_NULL_CHAR, pfill(imaxmom*4+3))
124           call add_ntuple_branch_double(
125          &    ntuple, 'wt_qq' //C_NULL_CHAR, pfill(imaxmom*4+4))
126           call add_ntuple_branch_double(
127          &    ntuple, 'wt_qqb'//C_NULL_CHAR, pfill(imaxmom*4+5))
128
129           do i = 1, ipdf
130             write(branch_name,"(A3I2A)") "PDF", i, C_NULL_CHAR
131             call add_ntuple_branch_double(
132          &      ntuple, branch_name, pfill(imaxmom*4+5+i))
133           end do
134         end subroutine
135
136         subroutine mcfm_ntuple_fill(p,wt)
137           use iso_c_binding
138           implicit none
139           include 'types.f'
140           include 'mxpart.f'
141           include 'wts_bypart.f'
142           include 'PDFerrors.f'
143
144           real(dp):: p(mxpart,4)
145           real(dp):: wt
146
147           ! Extra common block to carry the information about maximum momenta entries
148           integer:: i, imaxmom, ipdf
149           common/iarray/imaxmom, ipdf
150
151           ! assume at most 10 final state particles and 60 PDF sets
152           real(c_double) pfill(105)
153           type(c_ptr) :: ntuple
154           common/ntuple_interface/ntuple,pfill
155
156           ! If the event weight is zero, don't bother to add the n-tuple
157           if (wt == 0._dp) return
158
159           do i = 0, imaxmom-1
160             pfill(4*i+1) = p(i+3,1)
161             pfill(4*i+2) = p(i+3,2)
162             pfill(4*i+3) = p(i+3,3)
163             pfill(4*i+4) = p(i+3,4)
164           end do
165
166           ! set up single precision variables for the event weights
```

```
          pfill(imaxmom*4+1) = wt
          pfill(imaxmom*4+2) = wt_gg
          pfill(imaxmom*4+3) = wt_gq
          pfill(imaxmom*4+4) = wt_qq
          pfill(imaxmom*4+5) = wt_qqb

          ! include PDF errors if necessary
          if (PDFerrors) then
            do i = 1, ipdf
              pfill(imaxmom*4+5+i) = wt*PDFwgt(i)/PDFwgt(0)
            end do
          endif

          call fill_ntuple(ntuple) ! add ntuple entry
        end subroutine

! ======================================================================
        ! dummy routines, as in dsw_dummy

        subroutine dswhbook(n,titlex,dx,xmin,xmax)
          implicit none
          include 'types.f'

          integer:: n
          character titlex*8
          real(dp)::dx,xmin,xmax

          call dsw_error
        end subroutine

        subroutine dswhfill(n,var,wgt)
          implicit none
          include 'types.f'

          integer:: n
          real(dp)::var,wgt

          call dsw_error
        end subroutine

        subroutine dsw_error
          implicit none

          write(6,*) 'This version of MCFM has not been compiled for'
          write(6,*) 'ROOT output only; DSW-style histograms are not'
          write(6,*) 'available.'
          write(6,*)

          stop
        end subroutine
```