E The Toolbox by Examples

In this tutorial we will show how to write an application program, based on the QCDNUM toolbox, that can evolve polarised and unpolarised pdfs at LO, and hold both of these pdfs in memory. Of course, QCDNUM itself does these evolutions already up to NNLO but the aim here is not to develop useful software, but to learn how to use the toolbox.

The reader who wants to write code is advised to start from `tutorial00.f` which can be found in the `testjobs` directory of the QCDNUM distribution. This example program (see also Section 4.2) provides a basic set-up of QCDNUM as is needed by the toolbox. A kick-start with `Toolbox00.f` also will enable you to run the evolutions with QCDNUM, and directly compare the results with those from your own code.

E.1 How to partition a workspace

In this section we describe how to set-up the local workspace `w(nw)` for our evolution of unpolarised and polarised pdfs at LO. The toolbox routines are described in Section 6.3.

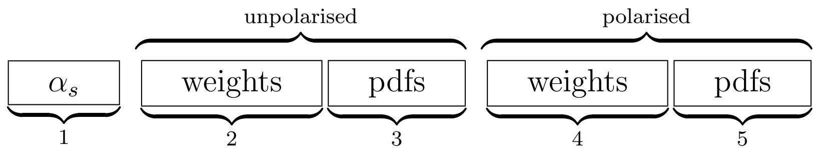
We set α_s common for both type of evolution so that we will need a single α_s table. In addition we need, for each type of evolution, four weight tables ($P_{qq}, P_{qg}, P_{gq}, P_{gg}$) and 13 pdf tables (one gluon and up to 6 flavours of quark and antiquark). Note from eqs. (A.2) and (A.3) in Appendix A that there are two splitting functions that depend only on x (type-1) and two that depend on x and n_f (type-2).[42]

There is of course nothing against putting all these tables into one set:[43]

```
parameter (nw = ...)
dimension w(nw)
dimension itypes(6)
data itypes / 4, 4, 0, 0, 26, 1 /

call MAKETAB(w, nw, itypes, 0, 0, iset, nwused) !returns iset = 1
```

However, the code will become much more flexible if we put the α_s, unpolarised and polarised tables into separate sets. At this point we envisage to dump the weight tables to disk which implies that they should be separated from the pdfs by storing them in their own sets. Thus we arrive at a memory layout with five sets, as shown below.



Here is the code that partitions the local workspace in the manner shown above; note that each call to `maketab` will generate a new set of tables.

[42]In fact, you can put *all* the splitting functions into type-2 tables, or even type-3 or 4 if desired; it won't do harm but should be avoided because it is a waste of memory and CPU.

[43]To find out how much space is needed simply put `nw = 1` and let the `maketab` error message give you the answer.

```
      parameter (nw = ....)
      dimension w(nw)
      dimension itypes_alf(6),itypes_pij(6),itypes_pdf(6)
      data itypes_alf / 0, 0, 0, 0, 0, 1 /
      data itypes_pij / 2, 2, 0, 0, 0, 0 /
      data itypes_pdf / 0, 0, 0, 0,13, 0 /

      call MAKETAB(w, nw, itypes_alf, 0, 0, iset_alfa, nwused) !set 1
      call MAKETAB(w, nw, itypes_pij, 0, 0, iset_piju, nwused) !set 2
      call MAKETAB(w, nw, itypes_pdf, 0, 0, iset_pdfu, nwused) !set 3
      call MAKETAB(w, nw, itypes_pij, 0, 0, iset_pijp, nwused) !set 4
      call MAKETAB(w, nw, itypes_pdf, 0, 0, iset_pdfp, nwused) !set 5
```

Please bear in mind that the table set identifiers are assigned by QCDNUM and *returned* in the pointers iset_alfa, iset_piju, *etc.* The α_s table is simply addressed by

```
      id_alfa = 1000*iset_alfa + 601
```

while the pdfs can be mapped onto, for instance, the indices (0–12) by the function

```
      id_pdf_unpolarised(i) = 1000*iset_pdfu + 501 + i
```

To map the splitting function tables onto the indices $(1, 2, 3, 4)$—see Eq. (E.1) in the next section—it is best to introduce a little pointer array like that shown below.

```
      dimension ipoint(4)
      data ipoint / 101, 201, 102, 202 /

      id_pij_unpolarised(i) = 1000*iset_piju + ipoint(i)
```

The weight tables must be filled before they can be used so that it makes sense to introduce weight filling routines that also take care of the partitioning:

```
      subroutine FillWtU( w, nw, iset_piju ) !iset_piju is out, not in
      implicit double precision (a-h,o-z)
      dimension w(nw)
      dimension itypes(6)
      data itypes / 2, 2, 0, 0, 0, 0 /

      call MAKETAB(w, nw, itypes, 0, 0, iset_piju, nwused)
      ..
      code to fill the unpolarised weight tables
      ..
      return
      end
```

Now we are in a position to maintain up-to-date weight files on disk, as is shown by the code below. Note that we shuffled the calls to make the code more readable; the table set identifiers will therefore be different from those above but this does not matter since they are stored in pointers (in fact, you should never hardwire them in your code).

```
parameter (nw = ....)
dimension w(nw)
dimension itypes_alf(6),itypes_pdf(6)
data itypes_alf / 0, 0, 0, 0, 0, 1/
data itypes_pdf / 0, 0, 0, 0,13, 0/
character*24 key
data key /'MyAddOn v1.0 22-Oct-2014'/

call READTAB(w, nw, lun, 'unp.wt', key, 0, iset_piju, nwused, ierr)
if(ierr.ne.0) then
  call FillWtU(w, nw, iset_piju)
  call DUMPTAB(w, iset_piju, lun, 'unp.wt', key)
endif

call READTAB(w, nw, lun, 'pol.wt', key, 0, iset_pijp, nwused, ierr)
if(ierr.ne.0) then
  call FillWtP(w, nw, iset_pijp)
  call DUMPTAB(w, iset_pijp, lun, 'pol.wt', key)
endif

call MAKETAB(w, nw, itypes_alf, 0, 0, iset_alfa, nwused)
call MAKETAB(w, nw, itypes_pdf, 0, 0, iset_pdfu, nwused)
call MAKETAB(w, nw, itypes_pdf, 0, 0, iset_pdfp, nwused)
```

By default, the weights are now read from disk, unless QCDNUM finds reason to reject them (read error, change of the x-μ^2 grid, new QCDNUM version, new memory layout, *etc.*), in which case `readtab` returns `ierr` $\neq$ `0`. This causes a jump into the `if`-block and the weights are calculated from scratch, followed by an update of the disk file.

Of course a disk file can also become obsolete if you have made changes in the weight calculation itself. In this case you can simply change the version number or the date (or whatever) in the key variable. Such a change will then raise the error flag and force a weight calculation from scratch followed by a disk dump, with the new key. Needless to say that this is a very easy and user-friendly way to manage the weight calculations.

In `Toolbox01.f` you can find the code presented in this section.

E.2 How to calculate weight tables

We will now further develop the routine `FillWtU` to calculate the weight tables for the unpolarised splitting functions at LO. The same code will work in the polarised case, provided that we feed-in the polarised splitting functions.

For the singlet-gluon evolution the LO splitting functions can be arranged in a 4×4 matrix as follows:

$$P_{ij} = \begin{pmatrix} P_{\rm qq} & P_{\rm qg} \\ P_{\rm gq} & P_{\rm gg} \end{pmatrix}, \qquad \texttt{id} = \begin{pmatrix} \texttt{1} & \texttt{2} \\ \texttt{3} & \texttt{4} \end{pmatrix} \to \begin{pmatrix} \texttt{101} & \texttt{201} \\ \texttt{102} & \texttt{202} \end{pmatrix}, \tag{E.1}$$

where in the second matrix we have indicated our choice of P_{ij} identifiers and in the third matrix the mapping to the table identifiers that we are going to use. The splitting functions are given by the Eqs. (A.2) and (A.3) in Appendix A:

$$\begin{aligned}
P_{\rm qq}(x) &= \underbrace{\left[\frac{4}{3}(1+x^2)\right]}_{\texttt{PQQR}} \times \underbrace{\left[\frac{1}{(1-x)_+}\right]}_{\texttt{PQQS}} + \underbrace{\left[2\right]}_{\texttt{PQQD}} \delta(1-x) \\
P_{\rm qg}(x) &= \underbrace{n_f\left[x^2+(1-x)^2\right]}_{\texttt{PQGA}} \\
P_{\rm gq}(x) &= \underbrace{\frac{4}{3}\left[\frac{1+(1-x)^2}{x}\right]}_{\texttt{PGQA}} \\
P_{\rm gg}(x) &= \underbrace{\left[6x\right]}_{\texttt{PGGR}} \times \underbrace{\left[\frac{1}{(1-x)_+}\right]}_{\texttt{PGGS}} + \underbrace{6\left[\frac{1-x}{x}+x(1-x)\right]}_{\texttt{PGGA}} + \underbrace{\left[\frac{11}{2}-\frac{n_f}{3}\right]}_{\texttt{PGGD}} \delta(1-x).
\end{aligned} \tag{E.2}$$

Here we have separated the regular and singular components, and have indicated the names of the FORTRAN functions of these components. These functions all have x, μ^2 and n_f as (dummy) arguments, for instance:

```
double precision function PQQD(x,qmu2,nf)
implicit double precision (a-h,o-z)
PQQD = 2.D0
return
end
```

We will not further discuss here the simple task of programming all the splitting function ingredients in (E.2) and will assume from now on that this has been done.

From (E.2) it is seen that the splitting functions in the second column of the matrix in (E.1) do depend on n_f, while those in the first column do not. This is reflected in the type-1 and type-2 table identifiers shown in the third matrix.

Apart from properly encoding the splitting functions, we also have to establish the relation $\chi = ax$ between the rescaling variable χ and the Bjorken variable x, see Sections 3.3 and 6.3. In our case $\chi = x$, $a = 1$ as defined by the function

```
double precision function ACHI(qmu2)
implicit double precision (a-h,o-z)
ACHI = 1.D0
return
end
```

Now we can write the complete code of the weight filling routine `FillWtU` which looks as follows (see Section 6.3 for a description of the toolbox routines used):

```
      subroutine FillWtU( w, nw, iset_piju ) !out: iset_piju
      implicit double precision (a-h,o-z)
      dimension w(nw)
      dimension itypes(6)
      data itypes / 2, 2, 0, 0, 0, 0 /

      external ACHI,PQQR,PQQS,PQQD,PQGA,PGQA,PGGR,PGGS,PGGA,PGGD

      call MAKETAB( w, nw, itypes, 0, 0, iset_piju, nwused )

      idPQQ = 1000*iset_piju + 101
      idPQG = 1000*iset_piju + 201
      idPGQ = 1000*iset_piju + 102
      idPGG = 1000*iset_piju + 202

      call MAKEWRS( w, idPQQ, PQQR, PQQS, ACHI, 0 )
      call MAKEWTD( w, idPQQ, PQQD, ACHI )

      call MAKEWTA( w, idPQG, PQGA, ACHI )

      call MAKEWTA( w, idPGQ, PGQA, ACHI )

      call MAKEWRS( w, idPGG, PGGR, PGGS, ACHI, 0 )
      call MAKEWTA( w, idPGG, PGGA, ACHI )
      call MAKEWTD( w, idPGG, PGGD, ACHI )

      return
      end
```

We leave it as an exercise to dig out the polarised LO splitting functions from the literature, or from the QCDNUM `pij` library, and write the weight filling routine `FillWtP`. The (unpolarised) code of this section you can find in `Toolbox02.f`.

E.3 How to fill the α_s table

At this point we have partitioned the workspace and filled the weight tables. The next step is to fill the α_s table and for this we have to use the toolbox routine `evfilla`.

```
    external AlfasFun
    id_as = 1000*iset_alfa + 601
    call EVFILLA( w, id_as, AlfasFun )
```

Here `AlfasFun` is a function—provided by us—that should return $\alpha_s/2\pi$ versus `iq`.[44]

[44] If we upgrade to beyond LO, additional tables and functions must be provided to store $(\alpha_s/2\pi)^n$.

It would seem that the QCDNUM function asfunc (Section 5.5) is a good way to get α_s but this is not so. The reason is that asfunc gives α_s at the *renormalisation* scale μ_R^2, but we will need it at the *factorisation* scale μ_F^2. The relation between $\alpha_s(\mu_R^2)$ and $\alpha_s(\mu_F^2)$ is given by the truncated Taylor expansion described in Section 2.3 and this is taken care of in internal QCDNUM tables. Thus we have to get the α_s values stored in these tables by calling the toolbox routine getalfn (Section 6.7), instead of using asfunc. As a bonus, we will now also have the proper renormalisation scale dependence of our evolutions and, in fact, of any calculation that uses α_s. Thus we write

```
double precision function AlfasFun( iq, nf, ithresh )
implicit double precision (a-h,o-z)
if( ithresh .eq. -1 ) then
  AlfasFun = GETALFN( -iq, 1, ierr )  !alfas/2pi
else
  AlfasFun = GETALFN(  iq, 1, ierr )  !alfas/2pi
endif
return
end
```

In this code we have used the threshold indicator ithresh to properly take care of discontinuities in α_s at the flavour thresholds. These are of course absent in our LO calculations but not anymore if we would upgrade the program to NLO or NNLO. Note also that getalfn does not return α_s but $\alpha_s/2\pi$.

The next thing to worry about is how to keep the α_s table up to date. It should clearly be updated when the input value $\alpha_s(\mu_0^2)$ changes, for instance in the iterations of a fit, but also when we change the order of the calculations, the flavour threshold settings or the relation between μ_R^2 and μ_F^2. Here is a routine that checks the current parameter values returned by calls to getalf(as,r2), getord(io), getcbt(nf,qc,qb,qt) and getabr(ar,br); the code is quite trivial and not all of it is shown.

```
logical function AtabInvalid()
implicit double precision (a-h,o-z)
save asL, r2L, ioL, nfL, qcL, qbL, qtL, arL, brL

call GETALF( as, r2 )
..
call GETABR( ar, br)
if( as.eq.asL .and. r2.eq.r2L ... .and. br.eq.brL ) then
  AtabInvalid = .false.
else
  AtabInvalid = .true.
  asL = as
  ..
  brL = br
endif
return
end
```

Now we can write the first lines of our evolution code.

```
      subroutine EvolSGNS( w, iset_alfa, ... )
      implicit double precision (a-h,o-z)
      logical   AtabInvalid
      external  AlfasFun
      dimension w(*)

      id_as = 1000*iset_alfa + 601
      if( AtabInvalid() ) call EVFILLA( w, id_as, AlfasFun )
      ..
```

In this way the α_s tables will always be up to date in our evolution routines.

You can find the code we have developed up to now in `Toolbox03.f`.

E.4 Singlet/gluon and non-singlet evolution

We can use the $n \times n$ `evdglap` routine (Section 6.5) both for the non-singlet and the singlet/gluon evolution by setting $n = 1$ or 2, respectively. Because $P_{\rm qq}$ is shared between the two evolutions we can compactly wrap everything into *one* user routine. Note, however, that this is only possible at LO because at higher orders the non-singlet splitting functions proliferate, and so will the code.

In what follows we will adopt the same pdf indexing (6.7) and (6.8) as QCDNUM. For convenience we show them here again for the flavour basis

−6	−5	−4	−3	−2	−1	0	1	2	3	4	5	6
$\bar{t}$	$\bar{b}$	$\bar{c}$	$\bar{s}$	$\bar{u}$	$\bar{d}$	g	d	u	s	c	b	t

, (E.3)

and for the singlet/non-singlet basis

0	1	2	3	4	5	6	7	8	9	10	11	12
g	$q_{\rm s}$	e_2^+	e_3^+	e_4^+	e_5^+	e_6^+	$q_{\rm v}$	e_2^-	e_3^-	e_4^-	e_5^-	e_6^-

. (E.4)

The singlet/non-singlet basis functions $e^\pm$ are in QCDNUM defined by[45]

$$\begin{pmatrix} e_1^\pm \\ e_2^\pm \\ e_3^\pm \\ e_4^\pm \\ e_5^\pm \\ e_6^\pm \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ -1 & 1 & & & & \\ 1 & 1 & -2 & & & \\ 1 & 1 & 1 & -3 & & \\ 1 & 1 & 1 & 1 & -4 & \\ 1 & 1 & 1 & 1 & 1 & -5 \end{pmatrix} \begin{pmatrix} d^\pm \\ u^\pm \\ s^\pm \\ c^\pm \\ b^\pm \\ t^\pm \end{pmatrix} \quad \text{with} \quad q_i^\pm \equiv q_i \pm \bar{q}_i. \tag{E.5}$$

The main purpose of our evolution routine is to hide details like the filling of the α_s table, the setting of the table identifiers, and the threshold loop described in Section 6.5.

[45]Note that $e_2 = u - d$, instead of $d - u$. The reason for this is purely cosmetic: $d - u$ is negative which would make e_2 look shocking on a plot.

Input to the evolution routine are the table-set identifiers `iseta`, `isetp` and `isetf`, a pdf table identifier `idfin`, the starting point `iq0` and an array `start` with start values. The order of the calculation is not passed as an argument but should, if the code is upgraded to beyond LO, be taken from QCDNUM via a call to `getord`. The code below will automatically perform a singlet/gluon evolution when `idfin` = `1` and a non-singlet evolution when `idfin` > `1`.[46] Note that inside the routine the start array is saved to a buffer because the start values are not preserved by `evdglap`.

A robust user routine should of course include checks on the input, like verifying that `idfin` is in the range 1–12, that QCDNUM runs in LO, and that `iq0` is within the grid boundaries but we will not clutter the code below with such checks.

```
      subroutine EvolSGNS( w, iseta, isetp, isetf, idfin, iq0, start )
      implicit double precision (a-h,o-z)
      logical   AtabInvalid
      external  AlfasFun
      dimension w(*), start(2,*)
      parameter (nxmax = 200) !change this when you use a larger xgrid
      dimension sbuf(2,nxmax)
      dimension idw(2,2,1), ida(2,2,1), idf(2), iqlim(2)

      idalf      = 1000*iseta + 601
      idw(1,1,1) = 1000*isetp + 101          !PQQ
      idw(1,2,1) = 1000*isetp + 201          !PQG
      idw(2,1,1) = 1000*isetp + 102          !PGQ
      idw(2,2,1) = 1000*isetp + 202          !PGG
      ida(1,1,1) = idalf
      ida(1,2,1) = idalf
      ida(2,1,1) = idalf
      ida(2,2,1) = idalf
      idf(1)     = 1000*isetf + 501 + idfin   !quark table
      idf(2)     = 1000*isetf + 501          !gluon table

      if( AtabInvalid() ) call EVFILLA( w, idalf, AlfasFun )

      n = 1                             !non-singlet evolution
      if( idfin.eq.1 ) n = 2            !singlet/gluon evolution

      call GRPARS( nx, xmi, xma, nq, qmi, qma, iosp )
      do i = 1,n
        do j = 1,nx
           sbuf(i,j) = start(i,j)       !copy start array
        enddo
      enddo
      iqlim(2) = iq0
```

[46]Note that LO DGLAP only makes a distinction between singlet/gluon and non-singlet evolution; the evolution routine does not care *which* non-singlet is evolved.

```
      nf       = 1
      do while( nf.gt.0 )            !upward evolution
        iqlim(1) = iqlim(2)
        iqlim(2) = 99999
        call EVDGLAP( w, idw, ida, idf, sbuf, 2, n, iqlim, nf, eps )
      enddo

      do i = 1,n
        do j = 1,nx
           sbuf(i,j) = start(i,j)    !copy start array
        enddo
      enddo
      iqlim(2) = iq0
      nf       = 1
      do while( nf.gt.0 )            !downward evolution
        iqlim(1) = iqlim(2)
        iqlim(2) = -99999
        call EVDGLAP( w, idw, ida, idf, sbuf, 2, n, iqlim, nf, eps )
      enddo

      return
      end
```

This routine can be used for both unpolarised and polarised evolution, simply by providing the correct table-set identifiers for the splitting functions and the pdfs. This flexibility clearly shows the advantage of organising tables into sets.

To test-run the evolution, you can of course fill the start array with any suitable smooth function of x but we propose here to take the starting values from an evolution previously run with the QCDNUM. In this way we can directly compare the pdfs from `EvolSGNS` with those from `evolfg` and check that our code is correct. Here is a routine that fills the start array with QCDNUM basis pdfs.

```
      subroutine SetStart( iset, idf, iq0, start )
      implicit double precision (a-h,o-z)
      dimension start(2,*)
      i = 1                    !quarks
      if( idf.eq.0 ) i = 2     !gluon
      call GRPARS(nx, xmi, xma, nq, qmi, qma, iord)
      do ix = 1,nx
        start(i,ix) = FSNSIJ(iset, idf, ix, iq0, 1)
      enddo
      return
      end
```

Here `iset` $=$ `1` (2) for unpolarised (polarised) evolution and `idf` is the gluon/singlet/non-singlet basis pdf identifier as defined by (E.4). The code that runs the singlet/gluon and the light quark non-singlet evolution now may look as follows.

```
parameter (nxmax = 200) !change this when you use a larger xgrid
dimension start(2,nxmax)
..
iset    = 1             !1= unpolarised   2 = polarised
isetp   = iset_piju
isetf   = iset_pdfu
if(iset.eq.2) then
  isetp = iset_pijp
  isetf = iset_pdfp
endif
call SetStart(iset, 1, iq0, start)       !singlet
call SetStart(iset, 0, iq0, start)       !gluon
call EvolSGNS(w, iset_alfa, isetp, isetf, 1, iq0, start)
do idf = 2,3
  call SetStart(iset, idf, iq0, start)   !nonsinglet e^+
  call EvolSGNS(w, iset_alfa, isetp, isetf, idf, iq0, start)
enddo
do idf = 7,9
  call SetStart(iset, idf, iq0, start)   !nonsinglet e^-
  call EvolSGNS(w, iset_alfa, isetp, isetf, idf, iq0, start)
enddo
..
```

This code correctly evolves the gluon, singlet and the *light* quark pdfs in the VFNS, and also in the FFNS with $n_f = 3$. To handle all flavours 3–6 in the FFNS you can simply set the upper limits in the loops above to n_f and $n_f + 6$, respectively, see `Toolbox04.f`.

The heavy quarks in the VFNS evolve only upward from their thresholds, starting from the singlet pdf for $e^+_{4,5,6}$ and the valence pdf for $e^-_{4,5,6}$. The code above clearly cannot do this and we leave it as an exercise to extend the evolution program so that it can fully handle the FFNS and VFNS. For this, note that the heavy flavour basis functions are not evolved below their thresholds or perhaps even not at all, depending on the FFNS/VFNS settings. To avoid that the heavy quarks in memory are partially undefined, it is a good idea to initialise their basis pdfs to the singlet or valence before they are evolved (this precaution might save you some headaches later). Here is the initialisation code.[47]

```
id(i) = 1000*isetf + 501 + i              !statement function
.. code to evolve singlet/gluon
do i = 4,6                                !loop over c,b,t
   call COPYWGT( w, id(1), id(i), 0 )     !copy singlet
enddo
.. code to evolve valence
do i = 10,12                              !loop over c,b,t
    call COPYWGT( w, id(7), id(i), 0 )    !copy valence
enddo
..
```

[47]The routine `copywgt` can copy tables of all types including type-5 (pdfs) and type-6 (α_s).

A fully generalised evolution routine can be found in the example program `Toolbox05.f`. In this program we have packaged the code into three user interface routines

```
      MyWeight( w, nw, iset, key )
      MyEvolve( w, iset, iq0, nfmax, idb )
      CompareF( w, iset, idf, iq, nprint, dif )
```

for weight calculation, evolution and comparison of the evolved pdfs with QCDNUM, respectively. In these routines `iset = 1 (2)` for unpolarised (polarised) evolution; we refer to the comments in the code for the meaning of the other parameters.

The introduction of a user interface raises several issues. First of all, we have up to now communicated common variables via parameter lists in brackets, but we cannot do this anymore for variables that have to be hidden for the user. One solution is to put them in common blocks but we have, instead, chosen the alternative to pass their values via setter/getter routines (`SetGetI` in `Toolbox05`).

Second, we have now to worry about the robustness of the code. Any sensible user would call the routines in the order given above—weights-evolution-comparison—but a robust program must handle, in one way or another, all the possible orderings of these calls. If you try this out with `Toolbox05` then you will find that QCDNUM itself is already quite robust and that we do not need to take action at this point. In the last Section of this tutorial we will add more robustness to the code, and also make it more user-friendly.

E.5 How to construct the singlet/non-singlet basis pdfs

Input to `EvolSGNS` are the gluon, singlet and non-singlet pdfs at μ_0^2. Up to now we have taken the start values from basis pdfs previously evolved by QCDNUM but ultimately we would like to take them from user-defined parameterisations. In the quark sector these parameterisations then represent a set of arbitrary (but independent) linear combinations of quarks and antiquarks. Our task is now to construct the singlet/non-singlet input basis pdfs from the set of pdfs provided by the user.

For definiteness we write for the set of input quark pdfs

$$|f_i\rangle \equiv \sum_{j=1}^{n_f} \alpha_{ij}\,|q_j\rangle + \beta_{ij}\,|\bar{q}_j\rangle = \sum_{j=1}^{2n_f} d_{ij}\,|e_j\rangle, \quad i = 1,\ldots,2n_f. \tag{E.6}$$

In matrix notation this reads

$$|\boldsymbol{f}\rangle = \boldsymbol{D}\,|\boldsymbol{e}\rangle \quad \to \quad |\boldsymbol{e}\rangle = \boldsymbol{D}^{-1}\,|\boldsymbol{f}\rangle. \tag{E.7}$$

We adopt the convention (E.5) for the basis pdfs so that we can use the routine `efromqq` (Section 6.7) to build the matrix $\boldsymbol{D}$, and `smb_dminv` from MBUTIL to invert this matrix.

This is done in the routine `GetDinv` below. Input to this routine is the active number of flavours `nf` and an array `abmat(-6:6,12)` where the user should specify in `abmat(i,j)` the contribution of (anti)quark flavour (`i`) to the input pdf (`j`).

```
      subroutine GetDinv( abmat, dinv, nf )
      implicit double precision (a-h,o-z)
      dimension abmat(-6:6,12), d(12), dmat(12,12), dinv(12,12)
C--   Build dmat
      do i = 1,2*nf
        call EFROMQQ( abmat(-6,i), d, nf )
        do j = 1,12
            dmat(i,j) = d(j)
        enddo
      enddo
C--   Invert dmat
      call InvertD( dmat, dinv, nf, ierr )
      if( ierr.ne.0 ) stop 'GetDinv : singular matrix'
      return
      end
```

Before inversion by `smb_dminv`, the `dmat` array must be copied into an $2n_f \times 2n_f$ working matrix. This is hidden in the routine `InvertD` (as an exercise, try to write it yourself).

```
      subroutine InvertD( dmat, dinv, nf ,ierr )
      implicit double precision (a-h,o-z)
      dimension dmat(12,12), dinv(12,12), work(12,12), iw(12)

C--   Indexing   e1+ e2+ e3+ e4+ e5+ e6+ e1- e2- e3- e4- e5- e6-
C--               1   2   3   4   5   6   7   8   9  10  11  12

C--   Initialise dinv to zero (code not shown)
      ..
C--   Copy dmat to a 2nf x 2nf working matrix
      do i = 1,2*nf
        do j = 1,nf
            work(i,j   ) = dmat(i,j  )
            work(i,j+nf) = dmat(i,j+6)
        enddo
      enddo
C--   Invert working matrix
      call SMB_DMINV( 2*nf, work, 12, iw, ierr )
      if( ierr.ne.0 ) return
C--   Copy inverted working matrix to dinv
      do i = 1,nf
        do j = 1,2*nf
          dinv(i  ,j) = work(i   ,j)
          dinv(i+6,j) = work(i+nf,j)
        enddo
      enddo
      return
      end
```

The input coefficients stored in the array `abmat(i,j)` are schematically shown below.

i	-6	-5	-4	-3	-2	-1	0	1	2	3	4	5	6
j	$\bar{\mathrm{t}}$	$\bar{\mathrm{b}}$	$\bar{\mathrm{c}}$	$\bar{\mathrm{s}}$	$\bar{\mathrm{u}}$	$\bar{\mathrm{d}}$	g	d	u	s	c	b	t
1	T	B	C	L	L	L	$\times$	L	L	L	C	B	T
2	T	B	C	L	L	L	$\times$	L	L	L	C	B	T
3	T	B	C	L	L	L	$\times$	L	L	L	C	B	T
4	T	B	C	L	L	L	$\times$	L	L	L	C	B	T
5	T	B	C	L	L	L	$\times$	L	L	L	C	B	T
6	T	B	C	L	L	L	$\times$	L	L	L	C	B	T
7	T	B	C	C	C	C	$\times$	C	C	C	C	B	T
8	T	B	C	C	C	C	$\times$	C	C	C	C	B	T
9	T	B	B	B	B	B	$\times$	B	B	B	B	B	T
10	T	B	B	B	B	B	$\times$	B	B	B	B	B	T
11	T	T	T	T	T	T	$\times$	T	T	T	T	T	T
12	T	T	T	T	T	T	$\times$	T	T	T	T	T	T

(E.8)

When $n_f(\mu_0^2) = 3$, the 6×6 matrix formed by the light quark coefficients L must be invertible; all other coefficients are ignored. For $n_f(\mu_0^2) = 4$, the 8×8 matrix formed by the L and C coefficients must likewise be invertible, and so on for $n_f = 5$ and 6.

Now we can put `GetDinv` into a routine that gets the start values of the gluon and quark pdfs from a user-given subroutine and transforms the latter into start values for the quark basis pdfs $|e^{\pm}\rangle$. The result is returned in the array `stval(i,ix)`, with `i` the basis pdf identifier, indexed according to (E.4).

```
      subroutine UsrStart( usub, abmat, iq0, stval )
      implicit double precision (a-h,o-z)
      external usub
      dimension abmat(-6:6,12), dinv(12,12), pdfusr(0:12), stval(0:12,*)

      nf = NFLAVOR(iq0)
      call GetDinv( abmat, dinv, nf )
      call GRPARS( nx, xmi, xma, nq, qmi, qma, iord )
      do ix = 1,nx
        call usub( ix, pdfusr )
        stval(0,ix) = pdfusr(0)
        do i = 1,nf
          stval(i  ,ix) = 0.D0
          stval(i+6,ix) = 0.D0
          do j = 1,2*nf
            stval(i  ,ix) = stval(i  ,ix) + dinv(i  ,j)*pdfusr(j)
            stval(i+6,ix) = stval(i+6,ix) + dinv(i+6,j)*pdfusr(j)
          enddo
        enddo
      enddo

      return
      end
```

The user-supplied subroutine usub should, as a function of ix, return the values of the gluon in pdfusr(0) and those of the $2n_f$ quark densities in pdfusr(1), . . . , pdfusr(2nf).

Note that in UsrStart the transformation matrix dinv is calculated at every call; if we care about efficiency (yes, of course we do) we could be a bit smarter and calculate the transformation matrix only when necessary:

```
      save nflast, dinv
      data nflast /0/
      ..
      nf = NFLAVOR(iq0)
      if( nf.ne.nflast ) then
        call GetDinv( abmat, dinv, nf )
        nflast = nf
      endif
```

The array stval(0:12,nx) filled by UsrStart cannot be directly fed into our evolution routine EvolSGNS so that we must first make a copy into the start array. Here is the subroutine that does it.

```
      subroutine CpyStart( idf, stval, start )
      implicit double precision (a-h,o-z)
      dimension stval(0:12,*), start(2,*)
      i = 1                     !copy quqarks
      if(idf.eq.0) i = 2        !copy gluon
      call GRPARS( nx, xmi, xma, nq, qmi, qma, iord )
      do ix = 1,nx
        start(i,ix) = stval(idf,ix)
      enddo
      return
      end
```

The upgraded evolution code can be found in Toolbox06.f.

E.6 Your own interpolation routine

The toolbox routines evplist and evtable can be used to interpolate the pdfs stored in the local workspace. But these routines cannot transform them to the flavour basis (d,u,s,. . .) simply because the flavour composition of the pdfs in the local memory is not known to QCDNUM. So how can we then get interpolated pdfs in flavour space?

First of all we can copy the pdfs from the local workspace into the QCDNUM internal memory and then call the built-in interpolation routines such as pdflst, *etc.* For this we can use pdfinp (Section 5.6) or, better, evfcopy (Section 6.5) to import the pdfs as an external pdf set into QCDNUM. An additional advantage of this is that we can then also use the structure function packages ZMSTF and HQSTF, or any other package that works with internal pdfs. Note, however, that QCDNUM can only store the gluon and 6 flavours of quark and antiquark so that other types of pdf, such as that of the photon, have to remain in the local workspace.

Second, we can provide our own interpolation routine which can be done very easily with the fast convolution engine described in Section 6.8. This may be attractive because you can write the code such that it can interpolate *any* pdf in the local workspace. The interpolation will also be faster since there is no copy step to QCDNUM internal memory.

The idea behind the fast engine is that it constructs an interpolation mesh from a given list of x-μ^2 interpolation points. Calculations are then performed at these mesh points, and *only* at these points, followed by an interpolation of the final result. This mechanism usually leads to very fast code since it can very much reduce redundant calculations. Please read the introduction part of Section 6.8 to understand the sparse (for interpolation) and dense (for convolution) storage schemes in the fast engine.

In our code we will use the engine to store the appropriate linear combination of pdfs in a fast buffer, immediately followed by an interpolation without any kind of calculation in between. In fact, a basic interpolation routine is quite straight forward:

```
subroutine Interpolate( w, isetf, id, x, q, f, n, ichk )
implicit double precision (a-h,o-z)
dimension w(*), x(*), q(*), f(*)
dimension coef(3:6)
data coef /4*1.D0/
idf = 1000*isetf + 501 + id        !pdf table identifier
call FASTINI(x, q, n, ichk)        !create interpolation mesh
call FASTINP(w, idf, coef, -1, 0) !sparse storage in buffer 1
call FASTFXQ(1, f, n)              !interpolate buffer 1
return
end
```

This routine does not (yet) make transformations to the flavour basis but efficiently interpolates a pdf in `w` to a list of `n` points, and returns the result in the array `f`.

A little drawback is that the number of interpolation points is limited by the parameter `mpt0` (set by default to 5000 in the file `qcdnum.inc`). Instead of raising this limit it is better to run through the list of interpolation points in batches of `mpt0` points, so that you will never hit the limit. The wrapper code below can handle an arbitrary long list of points.

```
call GETINT( 'mpt0', mpt0 )
nlast = 0
ntodo = min(n,mpt0)
do while( ntodo.gt.0 )
  i1    = nlast+1
  call Interpolate(w, iset, id, x(i1), q(i1), f(i1), ntodo, ichk)
  nlast = nlast+ntodo
  ntodo = min(n-nlast,mpt0)
enddo
```

Let us now modify the routine `Interpolate` so that it returns the gluon or an (anti)quark density. To see how this works we again write down the transformation (E.6) as

$$|f\rangle \equiv \sum_{i=1}^{n_f} \alpha_i\,|q_i\rangle + \beta_i\,|\bar{q}_i\rangle = \sum_{i=1}^{2n_f} d_i\,|e_i\rangle, \tag{E.9}$$

where $|f\rangle$ is now a single quark or antiquark pdf. We thus set *one* coefficient α_i or β_i to one, and the rest to zero, and calculate the coefficients d_i with the routine `efromqq`.

```
subroutine getcoefs( idf, coefd )       !idf = +-[1,2,3,4,5,6]
implicit double precision (a-h,o-z)
dimension qvec(-6:6), temp(12), coefd(3:6,12)
data qvec /13*0.D0/
qvec(idf) = 1.D0                    !select idf (and nothing else)
do nf = 3,6
  call EFROMQQ(qvec, temp, nf)
  do i = 1,12
    coefd(nf,i) = temp(i)
  enddo
enddo
qvec(idf) = 0.D0                    !deselect idf
return
end
```

The d_i depend on n_f so that `coefd` is a 2-dimensional array with n_f the first dimension (which runs fastest in FORTRAN) and the basis pdf identifier [1–12] the second dimension.

Now we have to store the linear combination on the right-hand side of (E.9) into a fast buffer. This is easy to do with the `fastinp` routine since it has the capability to multiply an input pdf with an n_f-dependent constant and then to either *put* or *add* the result into the buffer. Thus we build the linear combination $\sum d_i|e_i\rangle$ in a loop:

```
call FASTINP(w, id(1), coefd(3,1), -1, 0)    !0 = store
do i = 2,12
  call FASTINP(w, id(i), coefd(3,i), -1, 1)  !1 = add
enddo
```

In this snippet, `id(i)` is the table identifier of the basis pdf `i` = `1`,...,`12`.

Putting it all together, and setting the the gluon (`idf` = `0`) coefficients to one for all n_f, we arrive at the following interpolation routine.

```
subroutine Interpolate( w, isetf, idf, x, q, f, n, ichk )
implicit double precision (a-h,o-z)
dimension w(*), x(*), q(*), f(*)
dimension coefg(3:6), coefd(3:6,12)
data coefg /4*1.D0/
```

```
      id(i) = 1000*isetf + 501 + i                  !statement function

      call FASTINI(x, q, n, ichk)                   !interpolation mesh
      if( idf.eq.0 ) then
        call FASTINP(w, id(0), coefg, -1, 0)        !put gluon in buf1
      else
        call getcoefs(idf, coefd)                   !get coefficients
        call FASTINP(w, id(1), coefd(3,1), -1, 0)   !put c1*id1 in buf1
        do i = 2,12
          call FASTINP(w, id(i), coefd(3,i), -1, 1) !add ci*idi to buf1
        enddo
      endif
      call FASTFXQ( 1, f, n )                       !interpolate buf1
      return
      end
```

The routine `Interpolate` now returns a list of interpolated gluon or (anti)quark pdfs for `idf` = -6, . . . ,0, . . . ,6.

It always is tempting to call an interpolation routine in a loop

```
      do i = 1,100
         xi = ...
         qi = ...
         call Interpolate(w, iset, idf, xi, qi, fi, 1, ichk)
      enddo
```

but note that this is *very* slow because it completely counteracts the idea of bulk processing in the fast engine. Here is the correct way to obtain the same result.

```
      dimension x(100), q(100), f(100)
      do i = 1,100
        x(i) = ...
        q(i) = ...
      enddo
      call Interpolate(w, iset, idf, x, q, f, 100, ichk)
```

In `toolbox07.f` the routine `Interpolate` is compared to its QCDNUM equivalent `pdflst`.

More to come . . .