Nikhef-10-002
arXiv:1005.1481

QCDNUM: Fast QCD Evolution and Convolution

QCDNUM Version 17.01/14

M. Botje[*]
Nikhef, Science Park, Amsterdam, the Netherlands

December 21, 2017

Abstract

The QCDNUM program numerically solves the evolution equations for parton densities and fragmentation functions in perturbative QCD. Un-polarised parton densities can be evolved up to next-to-next-to-leading order in powers of the strong coupling constant, while polarised densities or fragmentation functions can be evolved up to next-to-leading order. In addition to these evolution routines, a large set of tools is provided to solve n-fold coupled QCD evolution equations and to compute convolution integrals in the zero-mass or generalised mass schemes. Based on this toolbox and included in the software distribution are two add-on packages to calculate zero-mass structure functions in un-polarised deep inelastic scattering, and heavy flavour contributions to these structure functions in the fixed flavour number scheme.

[*]Nikhef, Science Park 105, 1098XG Amsterdam, the Netherlands; email m.botje@nikhef.nl

PROGRAM SUMMARY

Program Title: QCDNUM

Version: 17.01

Author: M. Botje

E-mail: `m.botje@nikhef.nl`

Program obtainable from: `http://www.nikhef.nl/user/h24/qcdnum`

Distribution format: gzipped tar file

Journal Reference:

Catalogue identifier:

Licensing provisions: GNU Public License

Programming language: FORTRAN-77

Computer: all

Operating system: all

RAM: Typically 3 Mbytes

Keywords: QCD evolution, DGLAP evolution equations, Parton densities, Fragmentation functions, Structure functions

Classification: 11.5 Quantum Chromodynamics, Lattice Gauge Theory

External routines/libraries: none, except the MBUTIL, ZMSTF and HQSTF packages that are part of the QCDNUM software distribution.

Nature of problem: Evolution of the strong coupling constant and parton densities, up to next-to-next-to-leading order in perturbative QCD. Computation of observable quantities by Mellin convolution of the evolved densities with partonic cross-sections.

Solution method: Parametrisation of the parton densities as linear or quadratic splines on a discrete grid, and evolution of the spline coefficients by solving (coupled) triangular matrix equations with a forward substitution algorithm. Fast computation of convolution integrals as weighted sums of spline coefficients, with weights derived from user-given convolution kernels. !

Restrictions: Accuracy and speed are determined by the density of the evolution grid.

Running time: Less than 10 ms on a 2 GHz Intel Core 2 Duo processor to evolve the gluon density and 12 quark densities at next-to-next-to-leading order over a large kinematic range.

Contents

1 Introduction

In perturbative quantum chromodynamics (pQCD), a hard hadron-hadron scattering cross section is calculated as the convolution of a partonic cross section with the momentum distributions of the partons inside the colliding hadrons. These parton distributions depend on the Bjorken-x variable (fractional momentum of the partons inside the hadron) and on a scale μ^2 characteristic of the hard scattering process. Whereas the x-dependence of the parton densities is non-perturbative, the μ^2 dependence can be described in pQCD by the DGLAP evolution equations [1]. The perturbative expansion of the splitting functions in these equations has recently been calculated up to next-to-next-to-leading order (NNLO) in powers of the strong coupling constant α_s [2, 3].

QCDNUM is a FORTRAN program (with a C++ interface)[1] that numerically solves the DGLAP evolution equations on a discrete grid in x and μ^2. Input to the evolution are the x-dependence of the parton densities at some input mass factorisation scale, and an input value of α_s at some input renormalisation scale. To study the scale uncertainties, the renormalisation scale can be varied with respect to the mass factorisation scale. All calculations in QCDNUM are performed in the $\overline{\text{MS}}$ scheme.

The program was originally developed in 1988 by members of the BCDMS collaboration [4] for a next-to-leading order (NLO) pQCD analysis of the SLAC and BCDMS structure function data [5]. This code was adapted by the NMC for use at low x [6]. A complete revision led to the version 16.12 which was used in the QCD fits by ZEUS [7], and in a global QCD analysis of deep inelastic scattering data by the present author [8].

QCDNUM17 is the NNLO upgrade of QCDNUM16. A new evolution algorithm, based on quadratic spline interpolation, yields large gains in accuracy and speed; on a 2012 MacBook it takes less than 4 ms to evolve over a large kinematic range the full set of parton densities at NNLO in the variable flavour number scheme. QCDNUM17 can evolve un-polarised parton densities up to NNLO, and polarised densities or fragmentation functions up to NLO. It is also possible to read an external pdf set into memory.

A large toolbox provides routines to solve user-defined coupled QCD evolution equations and to calculate convolution integrals in the zero-mass or generalised mass schemes. Using these tools the functionality of QCDNUM can be extended in add-on packages.

Based on the toolbox and included in the software distribution are the ZMSTF and HQSTF packages to compute un-polarised zero-mass structure functions and, in the fixed flavour number scheme, the contribution from heavy quarks to these structure functions.

This write-up is organised as follows. In Section 2 we summarise the formalism underlying the DGLAP evolution of parton densities. The QCDNUM numerical method is described in Section 3. Details about the program itself and the description of an example job can be found in Section 4. A subroutine-by-subroutine manual is given in Section 5, while Section 6 shows how to steer the program with data cards. The QCDNUM toolbox is presented in Section 7, and the ZMSTF and HQSTF packages in the Appendices D and E, respectively. A toolbox tutorial can be found in Appendix F.

The C++ interface [9] is described in Section 5.1 and in dedicated text boxes.

[1]I thank V. Bertone for providing a first working version of the interface.

2 QCD Evolution

In pQCD, the strong coupling constant $\alpha_{\rm s}$ evolves on the renormalisation scale $\mu^2_{\rm R}$. The starting value is specified at some input scale, which usually is taken to be $m^2_{\rm Z}$.

The parton density functions (pdf) evolve on the factorisation scale $\mu^2_{\rm F}$. The starting point of a pdf evolution is given by the x dependence of the pdf at some initial scale μ^2_0. The coupled evolution equations that are obeyed by the gluon and the quark densities can, to a large extent, be decoupled by writing them in terms of the *singlet* quark density $\}$? (sum of all active quarks and anti-quarks) and *non-singlet* densities (orthogonal to the singlet in flavour space). A nice feature of QCDNUM is that it automatically takes care of the singlet/non-singlet decomposition of a set of pdfs.

Another input to the QCD evolution is the number of active flavours $n_{\rm f}$ which specifies how many quark species (d,u,s,...) are participating in the QCD dynamics. In the *fixed flavour number scheme* (FFNS), $n_{\rm f}$ is kept fixed throughout the evolution. Input to an FFNS evolution are then the gluon density and $2n_{\rm f}$ (anti-)quark densities at the input scale μ^2_0. In the *variable flavour number scheme* (VFNS), the flavour thresholds $\mu^2_{\rm c,b,t}$ are introduced and 3 light quark densities (d,u,s) are, together with the corresponding anti-quark densities, specified below the charm threshold $\mu^2_{\rm c}$. The heavy quarks and anti-quarks (c,b,t) are dynamically generated by the QCD evolution equations at and above their thresholds. Both the FFNS and the VFNS are supported by QCDNUM.

The QCD evolution formalism is relatively simple when the renormalisation and factorisation scales are equal, but it becomes more complicated when $\mu^2_{\rm R} \neq \mu^2_{\rm F}$. QCDNUM supports a linear relationship between the two scales.

In the following sections we describe the evolution of $\alpha_{\rm s}$ and the pdfs, the renormalisation scale dependence, the singlet/non-singlet decomposition, and the flavour schemes.

2.1 Evolution of the Strong Coupling Constant

The evolution of the strong coupling constant reads, up to NNLO,

$$\frac{{\rm d}a_{\rm s}(\mu^2)}{{\rm d}\ln\mu^2} = -\sum_{i=0}^{2} \beta_i \, a_{\rm s}^{i+2}(\mu^2). \tag{2.1}$$

Here $\mu^2 = \mu^2_{\rm R}$ is the renormalisation scale and $a_{\rm s} = \alpha_{\rm s}/2\pi$. The β-functions in (2.1) depend on the number $n_{\rm f}$ of active quarks with pole mass $m < \mu$. In the $\overline{\rm MS}$ scheme they are given by [10, 11]

$$\begin{aligned} \beta_0 &= \frac{11}{2} - \frac{1}{3}\, n_{\rm f} \\ \beta_1 &= \frac{51}{2} - \frac{19}{6}\, n_{\rm f} \\ \beta_2 &= \frac{2857}{16} - \frac{5033}{144}\, n_{\rm f} + \frac{325}{432}\, n_{\rm f}^2. \end{aligned} \tag{2.2}$$

The leading order (LO) analytic solution of (2.1) can be written as

$$\frac{1}{a_{\rm s}(\mu^2)} = \frac{1}{a_{\rm s}(\mu^2_0)} + \beta_0 \, \ln\left(\frac{\mu^2}{\mu^2_0}\right) \equiv \beta_0 \, \ln\left(\frac{\mu^2}{\Lambda^2}\right). \tag{2.3}$$

In (2.3), the parameter Λ is defined as the scale where the first term on the right-hand side vanishes, that is, the scale where α_s becomes infinite. Beyond LO, the definition of a scale parameter is ambiguous so that it is more convenient to take $\alpha_s(m_Z^2)$ as a reference. The value of α_s at any other scale is then obtained from a numerical integration of (2.1),[2] instead of from approximate analytic solutions parameterised in terms of Λ.

In the evolution of α_s, the number of active flavours is set to $n_f = 3$ below the charm threshold $\mu_R^2 = \mu_c^2$ and is changed from n_f to $n_f + 1$ at the flavour thresholds $\mu_R^2 = \mu_{c,b,t}^2$. At NNLO, and sometimes also at NLO, there are small discontinuities in the α_s evolution at the flavour thresholds [12]; see Section 2.5 for details.

In Figure 1, we plot the evolution of α_s calculated at LO, NLO and NNLO.[3] Because

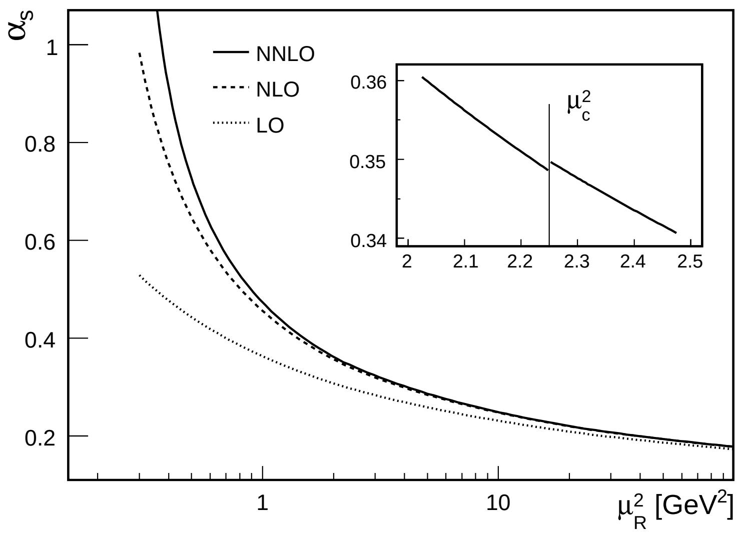


Figure 1 – The strong coupling constant $\alpha_s(\mu_R^2)$ evolved downward from $\alpha_s(m_Z^2) = 0.118$ at LO (dotted curve), NLO (dashed curve) and NNLO (full curve). The inset shows an enlarged view of the NNLO discontinuity in α_s at the charm threshold μ_c^2.

pQCD breaks down when α_s becomes large, QCDNUM will issue a fatal error when $\alpha_s(\mu^2)$ exceeds a pre-set limit. For a given value of $\alpha_s(m_Z^2)$, it is clear from the figure that such a limit will correspond to larger values of μ^2 at larger perturbative order.

2.2 The DGLAP Evolution Equations

y el término de gluon q qbar production?

The DGLAP evolution equations can be written as

$$\frac{\partial f_i(x,\mu^2)}{\partial \ln \mu^2} = \sum_{j=q,\bar{q},g} \int_x^1 \frac{dz}{z} P_{ij}\left(\frac{x}{z},\mu^2\right) f_j(z,\mu^2) \tag{2.4}$$

where f_i denotes an un-polarised parton number density, P_{ij} are the QCD splitting functions, x is the Bjorken scaling variable and $\mu^2 = \mu_F^2$ is the mass factorisation scale,

[2] I thank A. Vogt for providing his 4th order Runge-Kutta routine to integrate (2.1) up to NNLO.
[3] With the settings $\alpha_s(m_Z^2) = 0.118$ and $\mu_{c,b,t} = (1.5, 5, 188)$ GeV.

which we assume here to be equal to the renormalisation scale μ_R^2. The indices i and j in (2.4) run over the parton species *i.e.*, the gluon and n_f active flavours of quarks and anti-quarks. In the quark parton model, and also in LO pQCD, the parton densities are defined such that $f(x, \mu^2)\mathrm{d}x$ is, at a given μ^2, the number of partons which carry a fraction of the nucleon momentum between x and $x + \mathrm{d}x$. The distribution $xf(x, \mu^2)$ is then the parton momentum density.[4] Beyond LO there is no such intuitive interpretation. The definition of f then depends on the renormalisation and factorisation scheme in which the calculations are carried out ($\overline{\mathrm{MS}}$ in QCDNUM).[5]

Introducing a short-hand notation for the Mellin convolution,

$$[f \otimes g](x) = \int_x^1 \frac{\mathrm{d}z}{z} f\left(\frac{x}{z}\right) g(z) = \int_x^1 \frac{\mathrm{d}z}{z} f(z)\, g\left(\frac{x}{z}\right), \tag{2.5}$$

we can write (2.4) in compact form as (we drop the arguments x and μ^2 in the following)

$$\frac{\partial f_i}{\partial \ln \mu^2} = \sum_{j=q,\bar{q},g} P_{ij} \otimes f_j. \tag{2.6}$$

If the x dependencies of the parton densities are known at some scale μ_0^2, they can be evolved to other values of μ^2 by solving this set of $2n_f + 1$ coupled integro-differential equations. Fortunately, (2.6) can be considerably simplified by taking the symmetries in the splitting functions into account [10]:

$$\begin{aligned}
P_{\mathrm{gq}_i} &= P_{\mathrm{g}\bar{\mathrm{q}}_i} = P_{\mathrm{gq}} \\
P_{\mathrm{q}_i\mathrm{g}} &= P_{\bar{\mathrm{q}}_i\mathrm{g}} = \frac{1}{2n_f} P_{\mathrm{qg}} \\
P_{\mathrm{q}_i\mathrm{q}_k} &= P_{\bar{\mathrm{q}}_i\bar{\mathrm{q}}_k} = \delta_{ik} P^{\mathrm{v}}_{\mathrm{qq}} + P^{\mathrm{s}}_{\mathrm{qq}} \\
P_{\mathrm{q}_i\bar{\mathrm{q}}_k} &= P_{\bar{\mathrm{q}}_i\mathrm{q}_k} = \delta_{ik} P^{\mathrm{v}}_{\mathrm{q}\bar{\mathrm{q}}} + P^{\mathrm{s}}_{\mathrm{q}\bar{\mathrm{q}}}.
\end{aligned} \tag{2.7}$$

Inserting (2.7) in (2.6), we find after some algebra that the singlet quark density

$$q_s = \sum_{i=1}^{n_f} (q_i + \bar{q}_i) \tag{2.8}$$

obeys an evolution equation coupled to the gluon density

$$\frac{\partial}{\partial \ln \mu^2} \begin{pmatrix} q_s \\ g \end{pmatrix} = \begin{pmatrix} P_{\mathrm{qq}} & P_{\mathrm{qg}} \\ P_{\mathrm{gq}} & P_{\mathrm{gg}} \end{pmatrix} \otimes \begin{pmatrix} q_s \\ g \end{pmatrix}, \tag{2.9}$$

with P_{qq} given by

$$P_{\mathrm{qq}} = P^{\mathrm{v}}_{\mathrm{qq}} + P^{\mathrm{v}}_{\mathrm{q}\bar{\mathrm{q}}} + n_f(P^{\mathrm{s}}_{\mathrm{qq}} + P^{\mathrm{s}}_{\mathrm{q}\bar{\mathrm{q}}}). \tag{2.10}$$

Likewise, we find that the non-singlet combinations

$$q^{\pm}_{ij} = (q_i \pm \bar{q}_i) - (q_j \pm \bar{q}_j) \qquad \text{and} \qquad q_v = \sum_{i=1}^{n_f} (q_i - \bar{q}_i) \tag{2.11}$$

[4]In this section we use the number densities $f(x, \mu^2)$. In QCDNUM itself, however, we use $xf(x, \mu^2)$.

[5]In the DIS scheme f is defined such that the LO (quark-parton model) expression for the F_2 structure function is preserved at NLO. But this is true only for F_2 and not for F_L and xF_3.

evolve independently from the gluon and from each other according to

$$\frac{\partial\, q^{\pm}_{ij}}{\partial \ln \mu^2} = P_{\pm} \otimes q^{\pm}_{ij} \qquad \text{and} \qquad \frac{\partial\, q_{\rm v}}{\partial \ln \mu^2} = P_{\rm v} \otimes q_{\rm v}, \tag{2.12}$$

with splitting functions defined by

$$P_{\pm} = P^{\rm v}_{\rm qq} \pm P^{\rm v}_{\rm q\bar{q}} \qquad \text{and} \qquad P_{\rm v} = P^{\rm v}_{\rm qq} - P^{\rm v}_{\rm q\bar{q}} + n_{\rm f}(P^{\rm s}_{\rm qq} - P^{\rm s}_{\rm q\bar{q}}). \tag{2.13}$$

The evolution of the $q^{\pm}_{ij}$ is linear in the densities, so that any linear combination of the q^{+}_{ij} or q^{-}_{ij} also evolves according to (2.12).

The splitting functions can be expanded in a perturbative series in $\alpha_{\rm s}$ which presently is known up to NNLO. For the four splitting functions P_{ij} in (2.9) we may write

$$P_{ij}(x,\mu^2) = a_{\rm s}(\mu^2)\; P^{(0)}_{ij}(x) + a^2_{\rm s}(\mu^2)\; P^{(1)}_{ij}(x) + a^3_{\rm s}(\mu^2)\; P^{(2)}_{ij}(x) + {\rm O}(a^4_{\rm s}) \tag{2.14}$$

where we have set, as in the previous section, $a_{\rm s} = \alpha_{\rm s}/2\pi$. Note the separation in the variables x and μ^2 on the right-hand side of (2.14). We drop again the arguments x and μ^2 and write the expansion of the non-singlet splitting functions as

$$\begin{aligned} P_{\pm} &= a_{\rm s}\, P^{(0)}_{\rm qq} + a^2_{\rm s}\, P^{(1)}_{\pm} + a^3_{\rm s}\, P^{(2)}_{\pm} + {\rm O}(a^4_{\rm s}) \\ P_{\rm v} &= a_{\rm s}\, P^{(0)}_{\rm qq} + a^2_{\rm s}\, P^{(1)}_{-} + a^3_{\rm s}\, P^{(2)}_{\rm v} + {\rm O}(a^4_{\rm s}). \end{aligned} \tag{2.15}$$

Truncating the right-hand side to the appropriate order in $a_{\rm s}$, it is seen that at LO the three types of non-singlet obey the same evolution equations. At NLO, q^{-}_{ij} and $q_{\rm v}$ evolve in the same way but different from q^{+}_{ij}. At NNLO, all three non-singlets evolve differently.

It is evident from (2.7), (2.10) and (2.13) that several splitting functions depend on the number of active flavours $n_{\rm f}$. This number is set to 3 below $\mu^2_{\rm F} = \mu^2_{\rm c}$ and changed to $n_{\rm f} = (4, 5, 6)$ at and above the thresholds $\mu^2_{\rm F} = \mu^2_{\rm c,b,t}$. In case $\mu^2_{\rm F} \neq \mu^2_{\rm R}$, QCDNUM adjusts the $\mu^2_{\rm R}$ thresholds such that $n_{\rm f}$ changes in both the splitting and the beta functions when crossing a threshold; see also Section 2.5.

The LO splitting functions are given in Appendix B. Those at NLO can be found in [13] (non-singlet) and [14] (singlet).[6] The NNLO splitting functions and their parameterisations are given in [2] (non-singlet) and [3] (singlet). The DGLAP equations also apply to polarised parton densities and to fragmentation functions (time-like evolution), each with their own set of evolution kernels. For the polarised splitting functions up to NLO we refer to [15], and references therein. The time-like evolution of fragmentation functions at LO is described in [16]. The NLO time-like splitting functions can be found in [13] and [14]. In Appendix A we show which splitting functions actually enter in the space-like and time-like evolution (2.9) since this is not entirely obvious from [14].

2.3 Renormalisation Scale Dependence

In the previous section, we have assumed that the factorisation and renormalisation scales are equal. For $\mu^2_{\rm F} \neq \mu^2_{\rm R}$ we expand $a_{\rm s}$ in a Taylor series on a logarithmic scale

[6]Two well-known misprints in [14] are: (i) the lower integration limit in the definition of $S_2(x)$ must read $x/(1+x)$; (ii) in the expression for $\hat{P}^{(1,{\rm T})}_{\rm FF}$ the term $(10-18x-\frac{16}{3}x^2)$ must read $(-10-18x-\frac{16}{3}x^2)$.

around $\mu_{\rm R}^2$

$$a_{\rm s}(\mu_{\rm F}^2) = a_{\rm s}(\mu_{\rm R}^2) + a_{\rm s}'(\mu_{\rm R}^2)L_{\rm R} + \frac{1}{2}\, a_{\rm s}''(\mu_{\rm R}^2)L_{\rm R}^2 + \ldots \tag{2.16}$$

with $L_{\rm R} = \ln(\mu_{\rm F}^2/\mu_{\rm R}^2)$. Using (2.1) to calculate the derivatives in (2.16), we obtain

$$\begin{aligned}
a_{\rm s}(\mu_{\rm F}^2) &= a_{\rm s}(\mu_{\rm R}^2) - \beta_0 L_{\rm R}\, a_{\rm s}^2(\mu_{\rm R}^2) - (\beta_1 L_{\rm R} - \beta_0^2 L_{\rm R}^2)\, a_{\rm s}^3(\mu_{\rm R}^2) + {\rm O}(a_{\rm s}^4) \\
a_{\rm s}^2(\mu_{\rm F}^2) &= a_{\rm s}^2(\mu_{\rm R}^2) - 2\beta_0 L_{\rm R}\, a_{\rm s}^3(\mu_{\rm R}^2) + {\rm O}(a_{\rm s}^4) \\
a_{\rm s}^3(\mu_{\rm F}^2) &= a_{\rm s}^3(\mu_{\rm R}^2) + {\rm O}(a_{\rm s}^4).
\end{aligned} \tag{2.17}$$

To calculate the renormalisation scale dependence of the evolved parton densities, the powers of $a_{\rm s}$ in the splitting function expansions (2.14) and (2.15) are replaced by the expressions on the right-hand side of (2.17), with the understanding that these are truncated to order $a_{\rm s}$ when we evolve at LO, to order $a_{\rm s}^2$ when we evolve at NLO, and to order $a_{\rm s}^3$ when we evolve at NNLO.

2.4 Decomposition into Singlet and Non-singlets

In this section we describe the transformations between a flavour basis and a singlet/non-singlet basis, as is implemented in QCDNUM. For this purpose we write an arbitrary linear combination of quark and anti-quark densities as

$$|p\rangle = \sum_{i=1}^{n_{\rm f}} (\alpha_i |q_i\rangle + \beta_i |\bar{q}_i\rangle), \tag{2.18}$$

where the index i runs over the number of active flavours. To make a clear distinction between a coefficient and a pdf, we introduce here the ket notation $|f\rangle$ for $f(x,\mu^2)$.

Because a linear combination of non-singlets is again a non-singlet, it follows directly from the definition (2.11) that the coefficients of any non-singlet satisfy the constraint

$$\sum_{i=1}^{n_{\rm f}} (\alpha_i + \beta_i) = 0, \tag{2.19}$$

that is, a non-singlet is—by definition—orthogonal to the singlet in flavour space.

It is convenient to define $|q_i^{\pm}\rangle = |q_i\rangle \pm |\bar{q}_i\rangle$ and write the linear combination (2.18) as

$$|p\rangle = \sum_{i=1}^{n_{\rm f}} (b_i^+ |q_i^+\rangle + b_i^- |q_i^-\rangle). \tag{2.20}$$

The coefficients $b_i^{\pm}$, α_i and β_i are related by

$$b_i^{\pm} = \frac{\alpha_i \pm \beta_i}{2}, \qquad \alpha_i = b_i^+ + b_i^-, \qquad \beta_i = b_i^+ - b_i^-. \tag{2.21}$$

We define a basis of singlet, valence, and $2(n_{\rm f}-1)$ additional non-singlets by

$$|e_1^+\rangle = |q_{\rm s}\rangle, \qquad |e_1^-\rangle = |q_{\rm v}\rangle, \qquad |e_i^{\pm}\rangle = \sum_{j=1}^{i-1} |q_j^{\pm}\rangle - (i-1)\,|q_i^{\pm}\rangle \;\text{ for }\; 2 \le i \le n_{\rm f}. \tag{2.22}$$

In matrix notation, this transformation can be written as

$$|\boldsymbol{e}^{\pm}\rangle = \boldsymbol{U}|\boldsymbol{q}^{\pm}\rangle, \tag{2.23}$$

where $\boldsymbol{U}$ is the $n_{\mathrm{f}} \times n_{\mathrm{f}}$ sub-matrix of the 6×6 transformation matrix

$$\mathcal{U} = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & -1 & 0 & 0 & 0 & 0 \\ 1 & 1 & -2 & 0 & 0 & 0 \\ 1 & 1 & 1 & -3 & 0 & 0 \\ 1 & 1 & 1 & 1 & -4 & 0 \\ 1 & 1 & 1 & 1 & 1 & -5 \end{pmatrix}. \tag{2.24}$$

It is seen that the second to sixth row of (2.24) are orthogonal to the first row (singlet), so that they indeed represent non-singlets as defined by (2.19). In fact, all rows of $\boldsymbol{U}$ are orthogonal to each other, so that scaling by the row-wise norm yields a rotation matrix, which has the transpose as its inverse. By scaling back this inverse we obtain

$$\boldsymbol{U}^{-1} = \boldsymbol{U}^{\mathrm{T}}\boldsymbol{S}^{2}, \tag{2.25}$$

where $\boldsymbol{U}^{\mathrm{T}}$ is the transpose of $\boldsymbol{U}$ and $\boldsymbol{S}^2$ is the square of the diagonal scaling matrix:

$$S^2_{ij} = \delta_{ij}\left(\sum_{k=1}^{n_{\mathrm{f}}} U^2_{ik}\right)^{-1} = \begin{cases} \delta_{ij}/n_{\mathrm{f}} & \text{for } i = 1 \\ \delta_{ij}/i(i-1) & \text{for } i > 1. \end{cases} \tag{2.26}$$

Using (2.25) and (2.26) to invert any $n_{\mathrm{f}} \times n_{\mathrm{f}}$ sub-matrix of (2.24), it is straight forward to show by explicit calculation that

$$U^{-1}_{ij} = \begin{cases} \;1/n_{\mathrm{f}} & \text{for } j = 1 \\ -1/j & \text{for } j = i \neq 1 \\ \;1/j(j-1) & \text{for } j > i \\ \;0 & \text{otherwise.} \end{cases} \tag{2.27}$$

The inverse of the transformation (2.22) is thus given by

$$\begin{aligned} |q^{\pm}_1\rangle &= \frac{|e^{\pm}_1\rangle}{n_{\mathrm{f}}} + \sum_{j=2}^{n_{\mathrm{f}}} \frac{|e^{\pm}_j\rangle}{j(j-1)} \\ |q^{\pm}_i\rangle &= \frac{|e^{\pm}_1\rangle}{n_{\mathrm{f}}} - \frac{|e^{\pm}_i\rangle}{i} + \sum_{j=i+1}^{n_{\mathrm{f}}} \frac{|e^{\pm}_j\rangle}{j(j-1)} \qquad i > 1. \end{aligned} \tag{2.28}$$

We can now write the linear combination $|p\rangle$ on the $|e^{\pm}\rangle$ basis as

$$|p\rangle = \sum_{i=1}^{n_{\mathrm{f}}} (d^{+}_i|e^{+}_i\rangle + d^{-}_i|e^{-}_i\rangle), \tag{2.29}$$

where the coefficients $d^{\pm}_i$ are related to the $b^{\pm}_i$ of (2.20) by

$$d^{\pm}_i = \sum_{j=1}^{n_{\mathrm{f}}} b^{\pm}_j U^{-1}_{ji}, \qquad b^{\pm}_i = \sum_{j=1}^{n_{\mathrm{f}}} d^{\pm}_j U_{ji}. \tag{2.30}$$

Let the starting values of the DGLAP evolutions be given by the gluon density and $2n_\mathrm{f}$ arbitrary quark densities, that is, by $2n_\mathrm{f}+1$ functions of x at some input scale μ_0^2. We can arrange the input quark densities in a $2n_\mathrm{f}$-dimensional vector $|\boldsymbol{p}\rangle$. Likewise, we store the densities $|q_i^\pm\rangle$ in a vector $|\boldsymbol{q}\rangle$, the $|e_i^\pm\rangle$ in a vector $|\boldsymbol{e}\rangle$ and the $b^\pm$ coefficients of each input density in the rows of a $2n_\mathrm{f}\times 2n_\mathrm{f}$ matrix $\boldsymbol{B}$. The flavour decomposition of the input densities can then be written as $|\boldsymbol{p}\rangle = \boldsymbol{B}|\boldsymbol{q}\rangle$ and the singlet/non-singlet decomposition as

$$|\boldsymbol{p}\rangle = \boldsymbol{B}\boldsymbol{T}^{-1}|\boldsymbol{e}\rangle \quad \text{with} \quad \boldsymbol{T} \equiv \begin{pmatrix} \boldsymbol{U} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{U} \end{pmatrix}. \tag{2.31}$$

Provided that $\boldsymbol{B}^{-1}$ exists (*i.e.* the input densities are linearly independent), the starting values of the singlet and non-singlet densities are calculated from the inverse relation

$$|\boldsymbol{e}\rangle = \boldsymbol{T}\boldsymbol{B}^{-1}|\boldsymbol{p}\rangle. \tag{2.32}$$

2.5 Flavour Number Schemes

QCDNUM supports two evolution schemes, known as the fixed flavour number scheme (FFNS) and the variable flavour number scheme (VFNS, see below for the MFNS variant.)

In the FFNS we assume that n_f quark flavours have zero mass, while those of the remaining flavours are taken to be infinitely large. In this way, only n_f flavours participate in the QCD dynamics so that in the FFNS the value of n_f is simply kept constant for all μ^2, with $3 \le n_\mathrm{f} \le 6$. In the FFNS, the input scale μ_0^2 can be chosen anywhere within the boundaries of the evolution grid, although one should be careful with backward evolution in QCDNUM; see Section 3.4.

In the VFNS, the number of flavours changes from n_f to $n_\mathrm{f}+1$ when the factorisation scale is equal to the pole mass of the heavy quarks $\mu_h^2 = m_h^2$, $h = (\mathrm{c},\mathrm{b},\mathrm{t})$. A heavy quark h is thus considered to be infinitely massive below μ_h^2 and mass-less above μ_h^2. As a consequence, the heavy flavour distributions are zero below their respective thresholds and are dynamically generated by the QCD evolution equations at and above μ_h^2. Such an abrupt turn-on at a fixed scale is of course un-physical but this poses no problem since the parton densities themselves are not observable. The VFNS or FFNS parton densities evolved with QCDNUM are, in fact, valid input to structure function and cross section calculations that include mass terms and obey the kinematics of heavy quark production [17, 18, 19]. Such calculations are not part of QCDNUM itself, but can be coded in add-on packages; see the tools described in Section 7.

An important feature of VFNS evolution is that the input scale μ_0^2 cannot be above the lowest heavy flavour threshold μ_c^2. This is because otherwise heavy flavour contributions must be included in the input parton densities which clearly is in conflict with the dynamic generation of heavy flavour by the QCD evolution equations.

Another feature of the VFNS is the existence of discontinuities at the flavour thresholds in α_s and the parton densities; we will now turn to the calculation of these discontinuities. Because the beta functions (2.2) depend on n_f, it follows that the slope of the α_s evolution is discontinuous when crossing a threshold in the VFNS. Beyond LO there are not only discontinuities in the slope but also in α_s itself [12]. In N$^\ell$LO, the value of $\alpha_\mathrm{s}^{(n_\mathrm{f}+1)}$ is, at a flavour threshold, related to $\alpha_{\mathrm{s}}^{(n_{\mathrm{f}})}$ by, in the notation of [20],

$$a_{\mathrm{s}}^{(n_{\mathrm{f}}+1)}(\kappa\mu_h^2) = a_{\mathrm{s}}^{(n_{\mathrm{f}})}(\kappa\mu_h^2) + \sum_{n=1}^{\ell}\Big\{\left[a_{\mathrm{s}}^{(n_{\mathrm{f}})}(\kappa\mu_h^2)\right]^{n+1}\sum_{j=0}^{n} C_{n,j}\ln^j\kappa\Big\} \qquad \ell = 1,2. \tag{2.33}$$

Here μ_h^2 is the threshold defined on the factorisation scale and κ is the ratio $\mu_{\mathrm{R}}^2/\mu_{\mathrm{F}}^2$ at μ_h^2. For $a_{\mathrm{s}} = \alpha_{\mathrm{s}}/4\pi$, the coefficients C in (2.33) read

$$C_{1,0} = 0, \qquad C_{1,1} = \tfrac{2}{3}, \qquad C_{2,0} = \tfrac{14}{3}, \qquad C_{2,1} = \tfrac{38}{3}, \qquad C_{2,2} = \tfrac{4}{9}.$$

Note that there is always a discontinuity in α_{s} at NNLO. At NLO, a discontinuity only occurs when $\kappa \neq 1$, that is, when the renormalisation and factorisation scales are different. In case of upward evolution, $\alpha_{\mathrm{s}}^{(n_{\mathrm{f}}+1)}$ is computed directly from (2.33) while for downward evolution, $\alpha_{\mathrm{s}}^{(n_{\mathrm{f}}-1)}$ is evaluated by numerically solving the equation

$$a_{\mathrm{s}}^{(n_{\mathrm{f}})} - a_{\mathrm{s}}^{(n_{\mathrm{f}}-1)} - \Delta a_{\mathrm{s}}\left(a_{\mathrm{s}}^{(n_{\mathrm{f}}-1)}\right) = 0,$$

where the function $\Delta a_{\mathrm{s}}(a_{\mathrm{s}})$ is given by the second term on the right-hand side of (2.33).

In the VFNS at NNLO, not only α_{s} but also the parton densities have discontinuities at the flavour thresholds [21]:

$$\begin{aligned} g(x,\mu_h^2,n_{\mathrm{f}}+1) &= g(x,\mu_h^2,n_{\mathrm{f}}) + \Delta g(x,\mu_h^2,n_{\mathrm{f}}) \\ q_i^{\pm}(x,\mu_h^2,n_{\mathrm{f}}+1) &= q_i^{\pm}(x,\mu_h^2,n_{\mathrm{f}}) + \Delta q_i^{\pm}(x,\mu_h^2,n_{\mathrm{f}}) \qquad i = 1,\ldots,n_{\mathrm{f}} \\ h^{+}(x,\mu_h^2,n_{\mathrm{f}}+1) &= \Delta h^{+}(x,\mu_h^2,n_{\mathrm{f}}) \\ h^{-}(x,\mu_h^2,n_{\mathrm{f}}+1) &= \Delta h^{-}(x,\mu_h^2,n_{\mathrm{f}}) = 0, \end{aligned} \tag{2.34}$$

where $h = (\mathrm{c},\mathrm{b},\mathrm{t})$ for $n_{\mathrm{f}} = (3,4,5)$. Note that a heavy quark h becomes a light quark q_i above the threshold μ_h^2.

In QCDNUM, the flavour thresholds on the renormalisation scale are adjusted such that n_{f} changes by one unit in both the beta functions and the splitting functions when crossing a threshold. With this choice, the parton densities are continuous at LO and NLO while at NNLO the calculation of the discontinuities is considerably simplified (all terms proportional to powers of $\ln(m^2/\mu^2)$ in ref. [21] vanish). So we may write

$$\begin{aligned} \Delta g(x,\mu_h^2,n_{\mathrm{f}}) &= a_{\mathrm{s}}^2\left\{[A_{\mathrm{gq}}\otimes q_{\mathrm{s}}](x,\mu_h^2,n_{\mathrm{f}}) + [A_{\mathrm{gg}}\otimes g](x,\mu_h^2,n_{\mathrm{f}})\right\} \\ \Delta q_i^{\pm}(x,\mu_h^2,n_{\mathrm{f}}) &= a_{\mathrm{s}}^2\,[A_{\mathrm{qq}}\otimes q_i^{\pm}](x,\mu_h^2,n_{\mathrm{f}}) \\ \Delta h^{+}(x,\mu_h^2,n_{\mathrm{f}}) &= a_{\mathrm{s}}^2\left\{[A_{\mathrm{hq}}\otimes q_{\mathrm{s}}](x,\mu_h^2,n_{\mathrm{f}}) + [A_{\mathrm{hg}}\otimes g](x,\mu_h^2,n_{\mathrm{f}})\right\}. \end{aligned} \tag{2.35}$$

Here a_{s} stands for $a_{\mathrm{s}}^{(n_{\mathrm{f}}+1)}(\kappa\mu_h^2)$ as defined by (2.33). The convolution kernels A_{ij} can be found in Appendix B of [21].[7]

The discontinuities in the basis vectors $|e_i^{\pm}\rangle$ are calculated from

$$e_i^{\pm}(x,\mu_h^2,n_{\mathrm{f}}+1) = e_i^{\pm}(x,\mu_h^2,n_{\mathrm{f}}) + \Delta e_i^{\pm}(x,\mu_h^2,n_{\mathrm{f}}) + \lambda_i(n_{\mathrm{f}})\Delta h^{\pm}(x,\mu_h^2,n_{\mathrm{f}}), \tag{2.36}$$

[7]In the notation of [21], $A_{\mathrm{gq}} = A_{\mathrm{gq,H}}^{\mathrm{S},(2)}$ (eq. B.5), $A_{\mathrm{gg}} = A_{\mathrm{gg,H}}^{\mathrm{S},(2)}$ (B.7), $A_{\mathrm{qq}} = A_{\mathrm{qq,H}}^{\mathrm{NS},(2)}$ (B.4), $A_{\mathrm{hq}} = \tilde{A}_{\mathrm{Hq}}^{\mathrm{PS},(2)}$ (B.1) and $A_{\mathrm{hg}} = \tilde{A}_{\mathrm{Hg}}^{S,(2)}$ (B.3). For the latter we use a parameterisation provided by A. Vogt.

where the light component $\Delta e_i^\pm$ is given by (2.35), with $q_i^\pm$ replaced by $e_i^\pm$. With the definition (2.22) of the basis functions, the values of the coefficients $\lambda_i(n_\mathrm{f})$ are

n_f	λ_1	λ_2	λ_3	λ_4	λ_5	λ_6
3	1	0	0	-3		
4	1	0	0	0	-4	
5	1	0	0	0	0	-5

(2.37)

When the densities are evolved upward in μ^2, it is straight forward to calculate with (2.34) and (2.35) the parton densities at $n_\mathrm{f}+1$ from those at n_f. However, QCDNUM is capable to invert the relation between n_f and $n_\mathrm{f}+1$ so that it can also calculate the discontinuities in case of downward evolution. For this it is convenient to write the calculation of the singlet and gluon discontinuities in matrix form, similar to (2.9)

$$\begin{pmatrix} q_\mathrm{s} \\ g \end{pmatrix}^{(n_\mathrm{f}+1)} = \begin{pmatrix} q_\mathrm{s} \\ g \end{pmatrix}^{(n_\mathrm{f})} + a_\mathrm{s}^2 \begin{pmatrix} A_\mathrm{qq}+A_\mathrm{hq} & A_\mathrm{hg} \\ A_\mathrm{gq} & A_\mathrm{gg} \end{pmatrix} \otimes \begin{pmatrix} q_\mathrm{s} \\ g \end{pmatrix}^{(n_\mathrm{f})} . \qquad (2.38)$$

In Section 3.4 we will show how (2.38) is turned into an invertible matrix equation.

Note that the heavy quark non-singlets do not obey the DGLAP evolution equations over the full range in μ^2, because the heavy flavours are simply set to zero below their thresholds, instead of being evolved. The evolution of the set $|e_i^\pm\rangle$ thus proceeds in the VFNS as follows: The singlet/valence densities $|e_1^\pm\rangle$ and the light non-singlets $|e_{2,3}^\pm\rangle$ are evolved both upward and downward starting from some scale $\mu_0^2 < \mu_\mathrm{c}^2$. The heavy non-singlets $|e_{4,5,6}^\pm\rangle$ are dynamically generated from the DGLAP equations by upward evolution from the thresholds $\mu_{\mathrm{c,b,t}}^2$. At and below the thresholds, $|e_{4,5,6}^+\rangle$ is set equal to the singlet and $|e_{4,5,6}^-\rangle$ to the valence. This is equivalent to setting the heavy quark and anti-quark distributions to zero, except that at NNLO the heavy flavours do not evolve from zero but from the non-zero discontinuity given in (2.34). This is illustrated in Figure 2 where we plot the charm and bottom starting distributions, normalised to

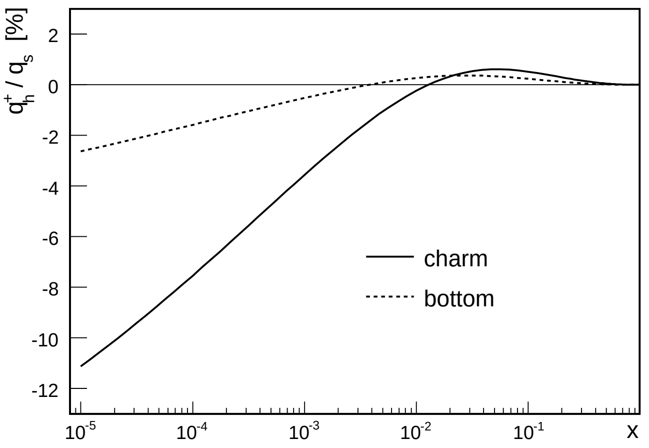


Figure 2 – The NNLO starting densities $q_h^+(x, \mu_h^2)$, normalised to the singlet density $q_\mathrm{s}(x, \mu_h^2)$, for charm (full curve) and bottom (dotted curve).

the singlet distribution. It is seen that the bottom discontinuity is less than 3% of the singlet over the whole range in x, while for charm it is much larger, exceeding 10% at low x. Note that the starting distributions are negative below $x \approx 10^{-2}$.

For the matching conditions in the time-like evolution of fragmentation functions we refer to [22]. By default, these are applied but can be switched-off, if desired.

QCDNUM also supports what we call the mixed flavour number scheme (MFNS) where the pdfs are evolved with a fixed number of light flavours, while α_s evolves with a variable number of flavours that change at given heavy quark mass thresholds. Thus n_f remains fixed in the splitting functions, but is variable in the β-functions, as is required in some heavy flavour calculations, see for instance [23].

3 Numerical Method

The DGLAP evolution equations are in QCDNUM numerically solved on a discrete $n \times m$ grid in x and μ^2. In such an approach the convolution integrals can be evaluated as weighted sums with weights calculated once and for all at program initialisation. Because of the convolutions, the total number of operations to solve a DGLAP equation is quadratic in n and linear in m. The accuracy of the solution depends, for a given grid, on the interpolation scheme chosen (linear or quadratic).

The advantage of this 'x-space' approach, compared to 'N-space' [20], is its conceptual simplicity and the fact that one is completely free to chose the functional form of the input distribution since it is fed into the evolution as a discrete vector of input values. A disadvantage is that accuracy and speed depend on the choice of grid and that each evolution will yield no less than $n \times m$ parton density values (typically 10^4) whether you want them or not.

The numerical method used in QCDNUM is based on polynomial spline interpolation of the parton densities on an equidistant logarithmic grid in x and a (not necessarily equidistant) logarithmic grid in μ^2. The order of the x-interpolation can in be set to $k = 2$ (linear) or 3 (quadratic). The interpolation in μ^2 is always quadratic. With such an interpolation scheme, the DGLAP evolution equations transform into a triangular set of linear equations in the interpolation coefficients. This leads to a very fast evolution of these coefficients from some input scale μ_0^2 to any other scale μ_i^2 on the grid. In the following sections we will describe the spline interpolation, the calculation of convolution integrals and the QCD evolution algorithm. Note that several features of the QCDNUM17 numerical method have been previously proposed in, for example, [24, 25].

3.1 Polynomial Spline Interpolation

To interpolate a function $h(y)$,[8] we sample this function on an $(n + 1)$-point grid

$$y_0 < y_1 < \ldots < y_{n-1} < y_n$$

[8]In QCDNUM, $h(y)$ represents a parton *momentum* density in the scaling variable $y = -\ln x$. However, for this section the identification of h with a parton density is not so relevant.

and parameterise it in each interval by a piece-wise polynomial of order k. Such a piece-wise polynomial is turned into a spline by imposing one or more continuity relations at each of the grid points. Usually—but not always—continuity is imposed at the internal grid points on the function itself and on all but the highest derivative, which is allowed to be discontinuous. Without further constraints at the end points, the spline has $k+n-1$ free parameters. Increasing the order k of the interpolation thus costs only *one* and not n extra parameters as is the case for unconstrained piece-wise polynomials.

It is convenient to write a spline function as a linear combination of so-called B-splines

$$h(y) = \sum_i A_i Y_i(y). \tag{3.1}$$

The basis Y_i of B-splines depends on the order k, on the distribution of the grid points along the y axis (equidistant in QCDNUM) and on the number of continuity relations we wish to impose at the internal grid points and at the two end points. For how to construct a B-spline basis and for more details on splines in general we refer to [26].

In Figure 3 are shown the B-splines for linear ($k = 2$) quadratic ($k = 3$) and cubic

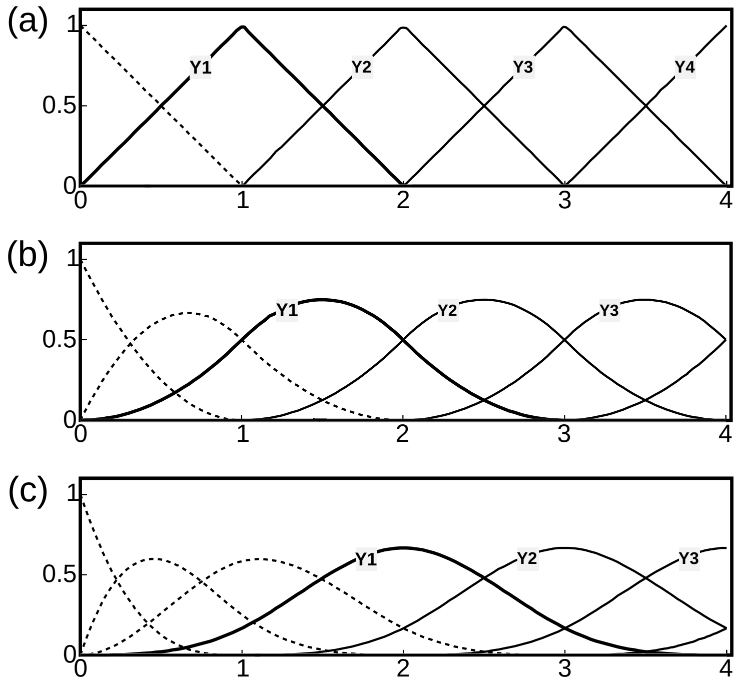


Figure 3 – B-spline bases generated on an equidistant grid. (a) Linear B-splines ($k = 2$). Removing the dashed spline enforces the boundary condition $h(y_0) = 0$; (b) Quadratic B-splines ($k = 3$). Removing the first two dashed splines enforces the boundary condition $h(y_0) = h'(y_0) = 0$; (c) Cubic B-splines ($k = 4$). Removing the first three dashed splines enforces the boundary condition $h(y_0) = h'(y_0) = h''(0) = 0$. Spline interpolation on such a basis is numerically unstable.

($k = 4$) interpolation on an equidistant grid. In case $h(y_0) = h(0) = 0$—which is always true for parton densities—we may remove the first B-spline in the plots of Figure 3. Removing the second B-spline in Figure 3b gives quadratic interpolation with an additional boundary condition $h'(y_0) = 0$.[9] With these boundary conditions—and because the grid is equidistant—the remaining B-splines possess translation invariance, that is, the basis can be generated by successively shifting the first spline one interval to the right (full curves in Figure 3a,b). Translation invariance greatly simplifies the evolution algorithm, as we will see later.

It is therefore tempting to extend the scheme to cubic interpolation by removing the first three B-splines in Figure 3c. This would yield a translation invariant basis with the boundary conditions $h(y_0) = h'(y_0) = h''(y_0) = 0$. However, it turns out that such a cubic spline interpolation tends to be numerically unstable. The cure is to drop the constraint $h''(y_0) = 0$ and impose a constraint on $h'(y_n)$ at the other end of the grid. But this does not fit in the evolution algorithm as it now stands so that we have abandoned cubic and higher order splines in QCDNUM.

If we number the B-splines $1, 2, \ldots, n$ from left to right as indicated in Figure 3 it is seen that for both $k = 2$ and 3 the following relation holds (translation invariance):

$$Y_i(y) = Y_1(y - y_{i-1}). \tag{3.2}$$

Furthermore, for linear interpolation ($k = 2$) we have $Y_i(y_i) = 1$ so that

$$\begin{aligned} h(y_0) &= 0 \\ h(y_i) &= A_i Y_i(y_i) = A_i \qquad 1 \le i \le n. \end{aligned} \tag{3.3}$$

Likewise, for quadratic interpolation ($k = 3$) we have $Y_{i-1}(y_i) = Y_i(y_i) = 1/2$ so that

$$\begin{aligned} h(y_0) &= 0 \\ h(y_1) &= A_1 Y_1(y_1) = A_1/2 \\ h(y_i) &= A_{i-1} Y_{i-1}(y_i) + A_i Y_i(y_i) = (A_{i-1} + A_i)/2 \qquad 2 \le i \le n. \end{aligned} \tag{3.4}$$

We denote $h(y_i)$ by h_i, the column vector of function values by $\boldsymbol{h} = (h_1, \ldots, h_n)^{\mathrm{T}}$, the corresponding vector of spline coefficients by $\boldsymbol{a}$ and write (3.3) and (3.4) as

$$\boldsymbol{h} = \boldsymbol{S}\, \boldsymbol{a} \tag{3.5}$$

where $\boldsymbol{S}$ is the identity matrix in case of linear interpolation and a lower diagonal band matrix for the quadratic spline. On a 5-point equidistant grid $y_0, \ldots, y_4$, for instance, we have in case of quadratic interpolation the vector $\boldsymbol{h} = (h_1, \ldots, h_4)^{\mathrm{T}}$ and the matrix

$$\boldsymbol{S} = \frac{1}{2} \begin{pmatrix} 1 & & & \\ 1 & 1 & & \\ & 1 & 1 & \\ & & 1 & 1 \end{pmatrix} \qquad \text{with inverse} \qquad \boldsymbol{S}^{-1} = 2 \begin{pmatrix} 1 & & & \\ -1 & 1 & & \\ 1 & -1 & 1 & \\ -1 & 1 & -1 & 1 \end{pmatrix}. \tag{3.6}$$

Note that $\boldsymbol{S}$ is sparse but $\boldsymbol{S}^{-1}$ is not. Thus, when a parton distribution $\boldsymbol{h_0}$ is given at some input scale μ_0^2, the corresponding vector $\boldsymbol{a_0}$ of spline coefficients is found by solving (3.5).[10] This vector is then evolved to other values of μ^2 using the DGLAP evolution equations as is described in the next two sections.

[9] A parton density parameterisation should thus behave like $h(y \to 0) \propto y^{\lambda}$ with $\lambda > 1$ because otherwise the condition $h'(0) = 0$ is violated and the spline might oscillate. All known pdf parameterisations fulfil this requirement but when the parameters are under control of a fitting program one should take precautions that λ will always stay above unity.

[10] Obtaining $\boldsymbol{a}$ from solving (3.5) by forward substitution (Appendix C) costs $\mathrm{O}(2n)$ operations. This is cheaper than the alternative of calculating $\boldsymbol{a} = \boldsymbol{S}^{-1}\boldsymbol{h}$ which costs $\mathrm{O}(n^2/2)$ operations.

3.2 Convolution Integrals

The Mellin convolution (2.5) calculated in QCDNUM is not that of a number density f and some kernel g but, instead, that of a momentum density $p = xf$ and a kernel $q = xg$. These convolutions differ by a factor x:

$$[p \otimes q](x) = x[f \otimes g](x). \tag{3.7}$$

This also true for multiple convolution: for $p = xf$, $q = xg$ and $r = xh$ we have

$$[p \otimes q \otimes r](x) = x[f \otimes g \otimes h](x). \tag{3.8}$$

A change of variable $y = -\ln x$ turns a Mellin convolution into a Fourier convolution:

$$[f \otimes g](x) = [u \otimes v](y) = \int_0^y \mathrm{d}z\; u(z)\; v(y-z) = \int_0^y \mathrm{d}z\; u(y-z)\; v(z), \tag{3.9}$$

where the functions u and v are defined by $u(y) = f(e^{-y})$ and $v(y) = g(e^{-y})$.

In the following we will denote by $h(y,t)$ a parton *momentum* density in the logarithmic scaling variables $y = -\ln x$ and $t = \ln \mu^2$. In terms of h, the DGLAP non-singlet evolution equation (2.12) is written as

$$\frac{\partial h(y,t)}{\partial t} = \int_0^y \mathrm{d}z\; Q(z,t)\; h(y-z,t) = \int_0^y \mathrm{d}z\; Q(y-z,t)\; h(z,t) \tag{3.10}$$

with a kernel $Q(y,t) = e^{-y}P(e^{-y},t)$. Here $P(x,t)$ is a non-singlet splitting function, as given in Section 2.2. To solve (3.10) we first have to evaluate the Fourier convolution

$$I(y,t) \equiv \int_0^y \mathrm{d}z\; Q(y-z,t)\; h(z,t). \tag{3.11}$$

Inserting (3.1) in (3.11) we find for the integrals at the grid points y_i (for clarity, we drop the argument t in the following)

$$I(y_i) = \sum_{j=1}^{i} A_j \int_0^{y_i} \mathrm{d}z\; Q(y_i - z)\; Y_j(z) \equiv \sum_{j=1}^{i} W_{ij} A_j \qquad (1 \le i \le n). \tag{3.12}$$

The summation is over the first i terms only, because B-splines with an index $j > i$ are zero in the integration domain $z \le y_i$, see Figure 3.

Eq. (3.12) defines the weights W_{ij} which are calculated as follows. Because $Y_j(y) = 0$ for $y < y_{j-1}$ the weights can be written as

$$W_{ij} = \int_{y_{j-1}}^{y_i} \mathrm{d}z\; Q(y_i - z) Y_j(z) = \int_0^{y_i - y_{j-1}} \mathrm{d}z\; Q(y_i - y_{j-1} - z) Y_1(z) \tag{3.13}$$

where we have used (3.2) in the second identity. From the property of equidistant grids

$$y_i + y_j = y_{i+j}$$

it follows that W_{ij} depends only on the difference $i-j$ (Toeplitz matrix):

$$W_{ij} = w_{i-j+1} \qquad \text{with} \qquad w_\ell \equiv \int_0^{y_\ell} \mathrm{d}z\; Q(y_\ell - z) Y_1(z) \qquad (1 \le \ell \le n). \tag{3.14}$$

The integrand only contributes in the region $k\Delta$ where Y_1 is non-zero so that in practical calculations the upper integration limit y_ℓ is replaced by $\min(y_\ell, k\Delta)$, with Δ the grid spacing. We remark that the calculation of the weights w_ℓ is a bit more complicated than suggested by (3.14) because singularities in the splitting functions have to be taken into account; for the relevant formula's we refer to Appendix B.

The weights can thus be arranged in a lower-triangular Toeplitz matrix, as is illustrated by the 4×4 example below:

$$W_{ij} = \begin{pmatrix} w_1 & & & \\ w_2 & w_1 & & \\ w_3 & w_2 & w_1 & \\ w_4 & w_3 & w_2 & w_1 \end{pmatrix}. \tag{3.15}$$

This matrix is fully specified by the first column, taking n instead of $n(n+1)/2$ words of storage. This is not only advantageous in terms of memory usage but also in terms of computing speed since frequent calculations like summing the perturbative expansion

$$\boldsymbol{W}(t) = a_\mathrm{s}(t)\{\, \boldsymbol{W}^{(0)} + a_\mathrm{s}(t)\boldsymbol{W}^{(1)} + \cdots \} \tag{3.16}$$

takes only $\mathrm{O}(n)$ operations instead of $\mathrm{O}(n^2/2)$. We write the vector of convolution integrals as $\boldsymbol{I}$ and express (3.12) in vector notation as

$$\boldsymbol{I} = \boldsymbol{W}\boldsymbol{a}. \tag{3.17}$$

Also multiple convolutions can be calculated as weighted sums. Let $f(x)$ be a number density and $K_{a,b}(x)$ be two convolution kernels. The vector of Mellin convolutions

$$I_i = x_i[f \otimes K_a \otimes K_b](x_i)$$

can be calculated from (3.17), using the weight table

$$\boldsymbol{W} = \boldsymbol{W}_a \boldsymbol{S}^{-1} \boldsymbol{W}_b. \tag{3.18}$$

Here $\boldsymbol{W}_a$ and $\boldsymbol{W}_b$ are the weight tables of K_a and K_b, respectively, and $\boldsymbol{S}$ is the transformation matrix defined by (3.5).

Another interesting convolution is that of two number densities f_a and f_b

$$I_i = x_i[f_a \otimes f_b](x_i).$$

This 'parton luminosity' [27] (times x) is calculated from the Fourier convolution

$$I(y_i) = \int_0^{y_i} \mathrm{d}z\; h_a(z)\, h_b(y_i - z). \tag{3.19}$$

Inserting the spline representation (3.1) gives an expression for the convolution integral as a weighted sum over the set of spline coefficients $\boldsymbol{a}$ of h_a and $\boldsymbol{b}$ of h_b,

$$I(y_i) = \sum_{j=1}^{i}\sum_{k=1}^{i} A_j B_k\, W_{ijk} \qquad \text{with} \qquad W_{ijk} \equiv \int_0^{y_i} \mathrm{d}z\; Y_j(z) Y_k(y_i - z).$$

To reduce the dimension of W_{ijk}, we use the translation invariance (3.2) and write

$$W_{ijk} = \int_0^{y_{i-j+1}} \mathrm{d}z\; Y_1(z)\; Y_k(y_{i-j+1} - z).$$

Because B-splines with index $k > i-j+1$ do not have their support inside the integration domain, we obtain an upper limit $k \leq i-j+1$. Again using translation invariance yields

$$W_{ijk} = \int_0^{y_{i-j-k+2}} \mathrm{d}z\; Y_1(z)\; Y_1(y_{i-j-k+2} - z).$$

We now have a compact expression for the convolution integral (3.19):

$$I(y_i) = \sum_{j=1}^{i}\sum_{k=1}^{i-j+1} A_j B_k\; w_{i-j-k+2} \qquad \text{with} \qquad w_\ell = \int_0^{y_\ell} \mathrm{d}z\; Y_1(z)\; Y_1(y_\ell - z). \tag{3.20}$$

Because Y_1 has a limited support, it turns out that only the first 3 (5) terms of w_ℓ are non-zero in case of linear (quadratic) interpolation. The operation count to calculate a convolution of parton densities is thus not more than O($5n$), for quadratic splines.

3.3 Rescaling Variable in Convolution Integrals

Calculating structure functions for heavy quarks leads to convolution not in x, but in the so-called rescaling variable χ. In this section we describe how QCDNUM handles such convolution integrals.

The general expression for a structure function can be written as [28]

$$\mathcal{F}_i(x, Q^2) = \sum_j x \int_\chi^1 \frac{\mathrm{d}z}{z} f_j(z, \mu^2)\; C_{ij}\left[\frac{\chi}{z}, \mu^2, Q^2, m_h^2, \alpha_\mathrm{s}(\mu^2)\right]. \tag{3.21}$$

Here the index i labels the structure function (*e.g.* F_2, F_L, xF_3, F_2^c, ...) and j labels a parton number density like the gluon, the singlet and various non-singlets. The coefficient function C_{ij} depends on x, on the scale variables μ^2 and Q^2, on one or more quark masses m_h^2 and on the strong coupling constant α_s. The variable $\chi = ax$, $a \geq 1$, is a so-called *rescaling* variable which takes into account the kinematic constraints of heavy quark production, for instance,

$$\chi = ax = \left(1 + \frac{4m_h^2}{Q^2}\right) x. \tag{3.22}$$

We have $0 \leq \chi \leq 1$ so that the range of x in (3.21) is restricted to $0 \leq x \leq 1/a$. In the zero-mass limit $a = 1$, $\chi = x$, and (3.21) reduces to the Mellin form $x[f \otimes C](x)$.

To calculate the structure function, we first have to evaluate the convolution integrals (for clarity we drop α_s and the indices i, j)

$$\mathcal{F}(x, Q^2) = x \int_\chi^1 \frac{\mathrm{d}z}{z} f(z, \mu^2)\, C\left(\frac{\chi}{z}, \mu^2, Q^2, m_h^2\right). \tag{3.23}$$

As in Section 3.2 we denote by $h(y, t)$ a parton momentum density in the logarithmic scaling variables $y = -\ln x$ and $t = \ln \mu^2$. In terms of these, and provided that χ is proportional to x, (3.23) can be written as a weighted sum of spline coefficients

$$\mathcal{F}(y_i, Q^2) = \sum_{j=1}^{i} W_{ij} A_j \tag{3.24}$$

with $W_{ij} = w_{i-j+1}$ and

$$w_\ell = e^{-b} \int_0^{y_\ell - b} \mathrm{d}z\, Y_1(z) D(y_\ell - b - z, t, Q^2, m_h^2) \qquad (1 \leq \ell \leq n). \tag{3.25}$$

Here $D(y, t, Q^2, m_h^2) = e^{-y} C(e^{-y}, e^t, Q^2, m_h^2)$ and $b = \ln(a)$. It is understood that the integral (3.25) is set to zero in case $y_\ell - b \leq 0$.

In the massive schemes, $b > 0$ depends on t which implies that the weights must be stored in 2-dimensional y-t tables unless, of course, $\chi = x$ ($b = 0$) so that we are back to the integrals of Section 3.2 which are functions of y only.

3.4 DGLAP Evolution

We denote by the vector $\boldsymbol{h_0}$ a *non-singlet* quark density at the input scale $t_0 = \ln \mu_0^2$. The derivative of $\boldsymbol{h_0}$ with respect to the scaling variable t is given by the DGLAP evolution equation (3.10) which can be written in vector notation as, from (3.5) and (3.17)

$$\frac{\mathrm{d}\boldsymbol{h_0}}{\mathrm{d}t} = \frac{\mathrm{d}\boldsymbol{S}\boldsymbol{a_0}}{\mathrm{d}t} = \boldsymbol{W_0}\, \boldsymbol{a_0} \qquad \text{or} \qquad \frac{\mathrm{d}\boldsymbol{a_0}}{\mathrm{d}t} \equiv \boldsymbol{a_0'} = \boldsymbol{S}^{-1}\boldsymbol{W_0}\, \boldsymbol{a_0}. \tag{3.26}$$

Likewise we have at t_1

$$\boldsymbol{a_1'} = \boldsymbol{S}^{-1}\boldsymbol{W_1}\, \boldsymbol{a_1}. \tag{3.27}$$

We have indexed the weight matrices above by a subscript because they depend on t through multiplication by powers of a_s, see (3.16).

Assuming that $\boldsymbol{a}(t)$ is quadratic in t, we can relate $\boldsymbol{a_0}$, $\boldsymbol{a_1}$, $\boldsymbol{a_0'}$ and $\boldsymbol{a_1'}$ by

$$\boldsymbol{a_1} = \boldsymbol{a_0} + (\boldsymbol{a_0'} + \boldsymbol{a_1'})\Delta_1 \tag{3.28}$$

with $\Delta_1 = (t_1 - t_0)/2$. If $t_1 > t_0$, Δ_1 is positive and we perform forward evolution. If $t_1 < t_0$, Δ_1 is negative and we perform backward evolution.

Inserting (3.26) and (3.27) in (3.28) we obtain a relation between the known spline coefficients $\boldsymbol{a_0}$ and the unknown coefficients $\boldsymbol{a_1}$

$$(\boldsymbol{1} - \boldsymbol{S}^{-1}\boldsymbol{W_1}\Delta_1)\, \boldsymbol{a_1} = (\boldsymbol{1} + \boldsymbol{S}^{-1}\boldsymbol{W_0}\Delta_1)\, \boldsymbol{a_0}. \tag{3.29}$$

Multiplying both sides from the left by $\boldsymbol{U_1} \equiv \boldsymbol{S}/\Delta_1$ gives

$$(\boldsymbol{U_1} - \boldsymbol{W_1})\,\boldsymbol{a_1} = (\boldsymbol{U_1} + \boldsymbol{W_0})\,\boldsymbol{a_0}. \tag{3.30}$$

Eq. (3.30) is more convenient than (3.29) because matrix multiplication $\boldsymbol{S}^{-1}\boldsymbol{W}$ is replaced by matrix addition.[11] Note that $\boldsymbol{U}$ is a lower diagonal band matrix so that $\boldsymbol{U} \pm \boldsymbol{W}$ is still lower triangular with, in fact, the Toeplitz structure (3.15) preserved. All this leads to a very simple and fast evolution algorithm, starting from $\boldsymbol{a_0}$:

1. At t_0, calculate $\boldsymbol{a_0}$ from (3.5), $\boldsymbol{W_0}$ from (3.16) and $\boldsymbol{U_1}$ as defined above. Then construct the vector $\boldsymbol{b_1} \equiv (\boldsymbol{U_1} + \boldsymbol{W_0})\,\boldsymbol{a_0}$.
2. Subsequently, at t_1,
   (a) Calculate $\boldsymbol{W_1}$ and the lower triangular matrix $\boldsymbol{V_1} = \boldsymbol{U_1} - \boldsymbol{W_1}$;
   (b) Solve the equation $\boldsymbol{V_1}\boldsymbol{a_1} = \boldsymbol{b_1}$ by forward substitution, see Appendix C;
   (c) Calculate $\boldsymbol{U_2}$ and $\boldsymbol{b_2} = (\boldsymbol{U_1} + \boldsymbol{U_2})\boldsymbol{a_1} - \boldsymbol{b_1}$ for the next evolution to t_2.[12]
3. Repeat step 2 at t_2 and so on.

With this algorithm each evolution step consists of a few vector manipulations which have an operation count $\mathrm{O}(n)$ and solving one triangular matrix equation which has an operation count $\mathrm{O}(n^2/2)$. The total operation count only very weakly depends on the order k of the interpolation chosen: quadratic interpolation is almost for free.

The algorithm can also be used for the coupled evolution of the singlet quark ($\boldsymbol{a}_\mathrm{s}$) and gluon ($\boldsymbol{a}_\mathrm{g}$) spline coefficients, provided we make the following replacements in the formalism:

$$\boldsymbol{a} \to \begin{pmatrix} \boldsymbol{a}_\mathrm{s} \\ \boldsymbol{a}_\mathrm{g} \end{pmatrix} \qquad \boldsymbol{S} \to \begin{pmatrix} \boldsymbol{S} & \\ & \boldsymbol{S} \end{pmatrix} \qquad \boldsymbol{W} \to \begin{pmatrix} \boldsymbol{W}_\mathrm{qq} & \boldsymbol{W}_\mathrm{qg} \\ \boldsymbol{W}_\mathrm{gq} & \boldsymbol{W}_\mathrm{gg} \end{pmatrix}.$$

In Appendix C is shown how the coupled triangular equations are solved by extending the forward substitution algorithm. The operation count is $4 \times \mathrm{O}(n^2/2)$ so that for m grid points in t we have in total $\mathrm{O}(2n^2m)$ operations for the singlet-gluon evolution and $\mathrm{O}(n^2m/2)$ operations for each non-singlet evolution.

Finally, let us express in vector notation the NNLO parton density discontinuities at the flavour thresholds. The relation between the singlet and gluon distributions at n_f and $n_\mathrm{f}+1$ as given by (2.38) can be written as

$$\begin{pmatrix} \boldsymbol{S} & \\ & \boldsymbol{S} \end{pmatrix} \begin{pmatrix} \boldsymbol{a}_\mathrm{s} \\ \boldsymbol{a}_\mathrm{g} \end{pmatrix}^{(n_\mathrm{f}+1)} = \begin{pmatrix} \boldsymbol{S} + \boldsymbol{A}_\mathrm{qq} + \boldsymbol{A}_\mathrm{hq} & \boldsymbol{A}_\mathrm{hg} \\ \boldsymbol{A}_\mathrm{gq} & \boldsymbol{S} + \boldsymbol{A}_\mathrm{gg} \end{pmatrix} \begin{pmatrix} \boldsymbol{a}_\mathrm{s} \\ \boldsymbol{a}_\mathrm{g} \end{pmatrix}^{(n_\mathrm{f})}. \tag{3.31}$$

It is easy to solve this linear equation for $\boldsymbol{a}^{(n_\mathrm{f}+1)}$ when $\boldsymbol{a}^{(n_\mathrm{f})}$ is known (forward evolution) or for $\boldsymbol{a}^{(n_\mathrm{f})}$ when $\boldsymbol{a}^{(n_\mathrm{f}+1)}$ is known (backward evolution). Likewise, we may write for the non-singlet discontinuities

$$\boldsymbol{S}\,\boldsymbol{a}_\mathrm{ns}^{(n_\mathrm{f}+1)} = (\boldsymbol{S} + \boldsymbol{A}_\mathrm{qq})\ \boldsymbol{a}_\mathrm{ns}^{(n_\mathrm{f})} + \lambda\left(\boldsymbol{A}_\mathrm{hq}\,\boldsymbol{a}_\mathrm{s}^{(n_\mathrm{f})} + \boldsymbol{A}_\mathrm{hg}\,\boldsymbol{a}_\mathrm{g}^{(n_\mathrm{f})}\right), \tag{3.32}$$

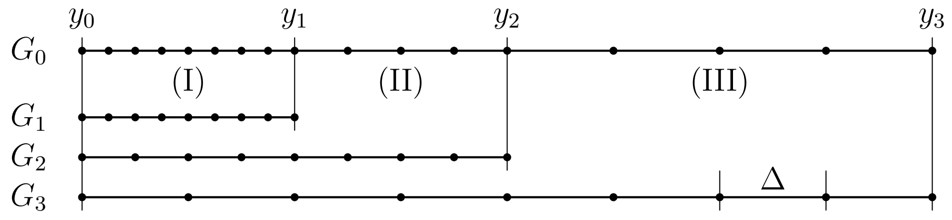
[11] In fact, adding a matrix with band structure (3.6) to a lower triangular matrix with structure (3.15) takes only *two* additions irrespective of the dimension of the matrices.

[12] Using (3.30) it is a simple exercise to establish this relation between $\boldsymbol{b}$, $\boldsymbol{U}$ and $\boldsymbol{a}$. Note that $\boldsymbol{b}$ in step (2c) is calculated much faster than $\boldsymbol{b}$ in step (1).

where λ is defined by (2.36). Also this equation can easily be inverted.

It can be seen from (3.5) and (3.28) that $h(y,t)$ is, by construction, a spline in both the variables y and t. However, it turns out that it is technically more convenient to represent the pdfs by their *values* on the grid, instead of by their spline coefficients. Polynomial interpolation of order k in y and quadratic in t is then done locally on a $k \times 3$ mesh around the interpolation point. The NNLO discontinuities are preserved by storing, at the flavour thresholds, the pdf values for both $n_\mathrm{f} - 1$ and n_f, and by prohibiting the interpolation mesh to cross a flavour threshold. Note, however, that the interpolation routine yields a single-valued function of t, so that one has to calculate $h(y, t_{\mathrm{c,b,t}} - \epsilon)$ to view the discontinuity.[13]

In QCDNUM it is possible to evolve on multiple equidistant y-grids which allow for a finer binning at low y (large x) where the parton densities are rapidly varying. This is illustrated below by a grid G_0 which is built-up from three equidistant sub-grids G_1, G_2 and G_3 with spacing $\Delta/4$, $\Delta/2$ and Δ, respectively.



On such a multiple grid, the parton densities are first evolved on the grid G_1 and the results are copied to the region (I) of G_0. The evolution is then repeated on the grids G_2 and G_3 followed by a copy to the regions (II) and (III) of G_0, respectively. We refer to Section 4.3 for spectacular gains in accuracy that can be achieved by employing these multiple grids.

3.5 Backward Evolution

As remarked above, the evolution algorithm can—at least in principle—handle both forward and backward evolution in μ^2 simply by changing the sign of Δ in (3.28). This works very well for linear spline interpolation but it turns out that backward evolution of quadratic splines can sometimes lead to severe oscillations. This is illustrated in Figure 4 where is shown a non-singlet quark density evolved downward from $\mu_0^2 = 5$ to $\mu^2 = 2$ GeV^2 in the quadratic interpolation scheme (dotted curve). In QCDNUM this numerical instability is handled as follows: (i) evolve downward from μ_0^2 to μ^2 in the linear interpolation scheme (which is stable); (ii) then take μ^2 as the starting scale and evolve *upward* to μ_0^2 in the quadratic interpolation scheme (also stable); (iii) calculate the difference Δf between the newly evolved pdf and the original one at μ_0^2; (iv) subtract Δf from the starting value at μ_0^2 used in (i) and repeat the procedure.

The full curve in the top plot of Figure 4 shows the result of downward evolution in the linear interpolation scheme, that is, without iterations. Oscillations are absent but the evolution is not very accurate as is evident from the difference between the dotted and

[13]Do not take ϵ too small because QCDNUM may snap to the threshold, see Section 5.4.

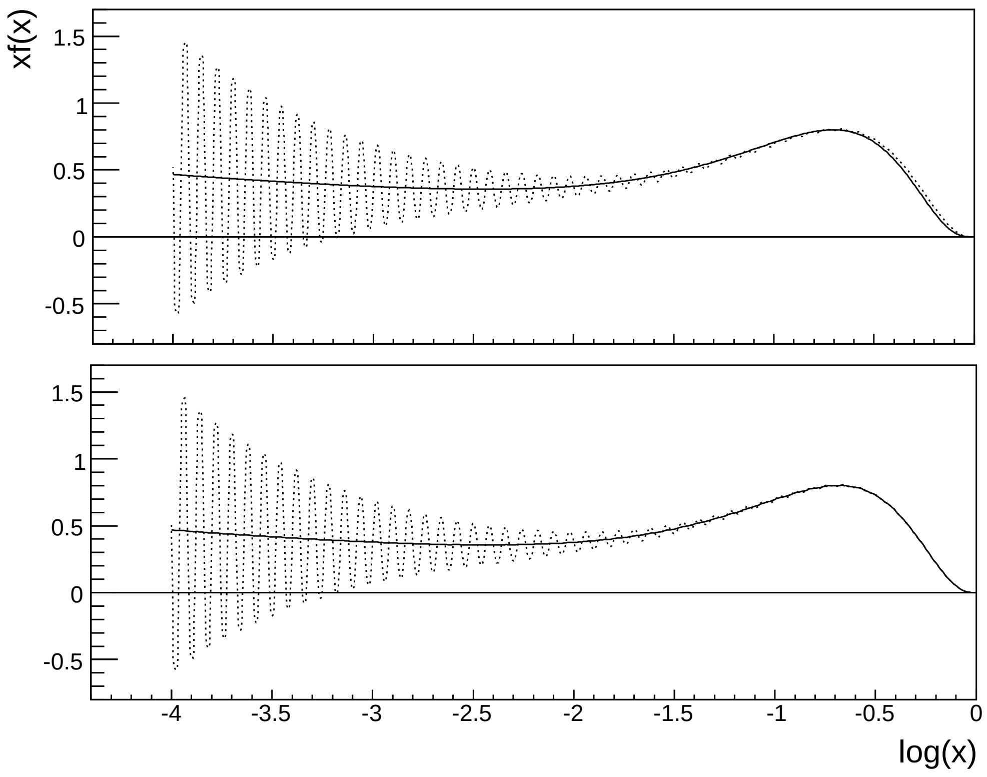


Figure 4 – A non-singlet parton density $xf(x)$ versus $\log(x)$ evolved downward from $\mu_0^2 = 5$ to $\mu^2 = 2$ GeV^2 in the quadratic interpolation scheme showing large oscillations (dotted curve). The full curve in the top plot shows the result of downward evolution in the linear interpolation scheme. The full curve in the bottom plot shows an improved result obtained by iteration, as described in the text.

full curve at large x. One iteration already much improves the precision as can be seen from the good match at large x between the two curves in the bottom plot. It turns out that one iteration (QCDNUM default), perhaps two, are sufficient while more iterations tend to spoil the convergence. Clearly best is to limit the range of downward evolution by keeping μ_0^2 low or, if possible, to set it at the lowest grid point to avoid downward evolution altogether.

QCDNUM checks for quadratic spline oscillation as follows. We denote the values of the quadratic B-spline at $(\frac{1}{2}\Delta, \Delta, \frac{3}{2}\Delta)$ by $(b_1, b_2, b_3) = (\frac{1}{8}, \frac{1}{2}, \frac{3}{4})$. It is easy to show that quadratic interpolation mid-between the grid points is given by $\boldsymbol{u} = \boldsymbol{D}\boldsymbol{a}$, where $\boldsymbol{D}$ is a lower diagonal Toeplitz band matrix, of bandwidth 3, which is characterised by the vector (b_1, b_3, b_1). Likewise, the linear interpolation of the spline at the mid-points is calculated from $\boldsymbol{v} = \boldsymbol{E}\boldsymbol{a}$, where $\boldsymbol{E}$ is the lower diagonal Toeplitz band matrix $(\frac{1}{2}b_2, b_2, \frac{1}{2}b_2)$. The maximum deviation $\epsilon = \|\boldsymbol{u} - \boldsymbol{v}\| = \|(\boldsymbol{D} - \boldsymbol{E})\boldsymbol{a}\|$ should be small; for pdfs sampled on a reasonably dense grid, $\epsilon \approx 0.1$ or less. For each pdf evolution, ϵ is computed at the input scale, and at the lower and upper end of the μ^2 grid. An error condition is raised when it exceeds a given limit, indicating that the spline oscillates, or that the x-grid is not dense enough.

4 The QCDNUM Program

4.1 Source Code

The QCDNUM source code can be downloaded from the web site

`http://www.nikhef.nl/user/h24/qcdnum`

Unpacking the tar file produces a directory `qdcnum-xx-yy-nn` with `xx-yy` the version number and `nn` the update number (see Appendix G). Sub-directories contain the source code, example jobs and write-up. See the `README` file for how to build QCDNUM with a simple script or with AUTOTOOLS.[14]

The code comes with a utility package MBUTIL (including write-up) which is a collection of general-purpose routines (some developed privately, some taken from CERNLIB and some taken from public source code repositories like NETLIB). Because QCDNUM uses several of these routines, MBUTIL must also be compiled and linked to your application program. Apart from this, QCDNUM is completely stand-alone. To calculate structure functions, the ZMSTF and HQSTF add-on packages are provided, see Sections D.3 and E.2.

Before compiling QCDNUM you may want to set several parameters which control the size of internal arrays. These parameters can be found in the include file `qcdnum.inc`:

`mxg0`	Maximum number of multiple x-grids [5].
`mxx0`	Maximum number of points in the x-grid [300].[15]
`mqq0`	Maximum number of points in the μ^2-grid [150].[13]
`mst0`	Maximum number of table sets in a workspace [30].
`mpl0`	Maximum number of evolution parameter lists [30].
`mce0`	Maximum number of coupled evolutions [20].
`mbf0`	Maximum number of fast convolution scratch buffers [10].
`mky0`	Maximum number of data-card keys [50].
`mqs0`	Size of the QCDNUM user store [100].
`nwf0`	Size of the QCDNUM dynamic store in words [1200000].

The first 9 parameters are simply dimensions of book-keeping arrays which you may want to adjust to your needs. More important is the parameter `nwf0` that defines the size of an internal store that contains the weight tables and the tables of parton densities. How many words are needed depends on the size of the tables which, in turn, depends on the size of the current x-μ^2 grid. It also depends on how many different sets of tables (un-polarised pdfs, polarised pdfs, fragmentation functions, *etc.*) you want to store. Note that QCDNUM is very user-friendly by always gracefully grinding to a halt if it runs out of memory, with a message that tells you how large `nwf0` should be.

[14] AUTOTOOLS is mandatory if you want to use the C++ interface.

[15] For technical reasons the maximum number of grid points is about 10 less than `mxx0` and `mqq0`.

C++ QCDNUM offers the possibility to call its FORTRAN routines via a C++ interface [9]. To learn about the interface we refer to the code shown in Figure 6 below, to the general remarks made in Section 5.1 and to the information given in text boxes like this one.

4.2 Application Program

To illustrate the use of QCDNUM, we present in Figure 5 the listing of a simple application program. For a detailed description of the subroutine calls, and for additional routines not included in the example, we refer to Section 5.

C++ In Figure 6 we show the C++ version of the example program.

The first step in a QCDNUM based analysis is initialisation (`qcinit`), setting up the x-μ^2 grid (`gxmake`, `gqmake`) and the calculation of the weight tables (`fillwt`). The weights depend on the grid definition and the interpolation order so that `fillwt` must be called after the grid has been defined. The weight tables are calculated for LO, NLO and NNLO as well as for all possible flavour settings in the range $3 \leq n_{\mathrm{f}} \leq 6$ so that you do not have to call `fillwt` again when you set or re-set QCDNUM parameters further downstream. Although the weight calculation is fast (typically about 10–20 s) it may become a nuisance in semi-interactive use of QCDNUM so that there is a possibility to dump the weights to disk and read them back in the next QCDNUM run.

In the example code, the weight calculation is followed by setting the perturbative order (`setord`) and the input value of α_{s} at some renormalisation scale μ_{R}^2 (`setalf`). The call to `setcbt` sets the VFNS mode and defines the thresholds on the factorisation scale μ_{F}^2.

The second step is to evolve the parton densities from input specified at the scale μ_0^2. It is important to note that QCDNUM evolves parton *momentum* densities $xf(x)$, although all theory in this write-up is expressed in terms of parton *number* densities $f(x)$. The evolution is done by calling the routine `evolfg` which evolves $2n_{\mathrm{f}} + 1$ input parton densities (quarks plus gluon) in the FFNS, VFNS or MFNS scheme. The routine internally takes care of the proper decomposition of the input quark densities into singlet and non-singlets. In the VFNS the input scale μ_0^2 must lie below the charm threshold μ_{c}^2 so that, as a consequence, μ_{c}^2 must lie above the lower boundary of the μ^2 grid.

The flavour composition of each of the input quark densities is given by a table of weights `def(-6:6,12)`. In the example program, six light quark input densities are defined: three valence densities $x(q - \bar{q})$ and three anti-quark densities $x\bar{q}$. This is sufficient input to run evolutions in the VFNS scheme. One is completely free to define the flavour composition of the input quark densities as long as they form a linearly independent set (QCDNUM checks this). Note that the flavours are ordered according to the PDG convention $\mathrm{d}, \mathrm{u}, \mathrm{s}, \ldots$ and not $\mathrm{u}, \mathrm{d}, \mathrm{s}, \ldots$ as often is the case in other programs.

The x dependence of these momentum densities at μ_0^2 must be coded for each identifier

```
C       ----------------------------------------------------------------
        program example
C       ----------------------------------------------------------------
        implicit double precision (a-h,o-z)
        data ityp/1/, iord/3/, nfin/0/          !unpolarised, NNLO, VFNS
        data as0/0.364/, r20/2.D0/              !alphas
        external func                           !input parton dists
        dimension def(-6:6,12)                  !flavor decomposition
        data def  /
C--     tb  bb  cb  sb  ub  db   g   d   u   s   c   b   t
C--     -6  -5  -4  -3  -2  -1   0   1   2   3   4   5   6
     +  0., 0., 0., 0., 0.,-1., 0., 1., 0., 0., 0., 0., 0.,   !dval
     +  0., 0., 0., 0.,-1., 0., 0., 0., 1., 0., 0., 0., 0.,   !uval
     +  0., 0., 0.,-1., 0., 0., 0., 0., 0., 1., 0., 0., 0.,   !sval
     +  0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0.,   !dbar
     +  0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0.,   !ubar
     +  0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0.,   !sbar
     +  78*0.    /
        data xmin/1.D-4/, nxin/100/, iosp/3/            !x grid, splord
        dimension qq(2),wt(2)                           !mu2 grid
        data qq/2.D0,1.D4/, wt/1.D0,1.D0/, nqin/60/     !mu2 grid
        data q2c/3.D0/, q2b/25.D0/, q0/2.0/             !thresh and mu20
        data x/1.D-3/, q/1.D3/, qmz2/8315.25D0/         !output scales
C       ----------------------------------------------------------------
        call qcinit(6,' ')                       !initialise
        call gxmake(xmin,1,1,nxin,nx,iosp)       !x-grid
        call gqmake(qq,wt,2,nqin,nq)             !mu2-grid
        call fillwt(ityp,id1,id2,nw)             !calculate weights
        call setord(iord)                        !LO, NLO, NNLO
        call setalf(as0,r20)                     !input alphas
        iqc  = iqfrmq(q2c)                       !mu2c
        iqb  = iqfrmq(q2b)                       !mu2b
        call setcbt(nfin,iqc,iqb,0)              !thresholds in the VFNS
        iq0  = iqfrmq(q0)                        !starting scale
        call evolfg(ityp,func,def,iq0,eps)       !evolve all pdfs
        csea = 2.D0*fvalxq(ityp,-4,x,q,0)        !charm sea at x,Q2
        asmz = asfunc(qmz2,nfout,ierr)           !alphas(mz2)
        end
C       ----------------------------------------------------------------
        double precision function func(id,x)     !momentum density xf(x)
C       ----------------------------------------------------------------
        implicit double precision (a-h,o-z)
        if(id.eq.0) func = gluon(x)              !0 = always gluon
        if(id.eq.1) func = dvalence(x)           !1 = defined in def
        ..                                       ..
        if(id.eq.6) func = strangebar(x)         !6 = defined in def
        return
        end
```

Figure 5 – Listing of a QCDNUM application program evolving a complete set of parton densities in the VFNS at NNLO. The array `def` defines the light quark valence ($xq - x\bar{q}$) and anti-quark ($x\bar{q}$) distributions as an input to the evolution. The x dependence of the input densities is coded in the function `func`. After evolution, the pdfs are interpolated to some x and μ^2 and $\alpha_s(m_Z^2)$ is calculated.

C++

```
#include <iostream> //not shown <iomanip>, <cmath>, <fstream>
#include "QCDNUM/QCDNUM.h"
using namespace std;

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 ==  6) f = xsbar(xb);
  return f;
  }

int main() {
  int    ityp = 1, iord = 3, nfin = 0;
  double as0 = 0.364, r20 = 2.0, xmin[] = {1.e-4};
  int    iwt[] = {1}, ng = 1, nxin = 100, iosp = 3, nqin = 60;
  double qq[] = { 2e0, 1e4}, wt[] = { 1e0, 1e0};
  double q2c = 3, q2b = 25, q0 = 2;
  double x = 1e-3, q = 1e3, qmz2 = 8315.25, pdf[13];
  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., 0.,   // 1=dval
      0., 0., 0., 0.,-1., 0., 0., 0., 1., 0., 0., 0., 0.,   // 2=uval
      0., 0., 0.,-1., 0., 0., 0., 0., 0., 1., 0., 0., 0.,   // 3=sval
      0., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0.,   // 4=dbar
      0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0.,   // 5=ubar
      0., 0., 0., 1., 0., 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., 0.};  //12=zero
  int nx, nq, id1, id2, nw, nfout, ierr ; double eps;
  int lun = 6 ; string outfile = " ";
  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::fillwt(ityp,id1,id2,nw);              //compute weights
  QCDNUM::setord(iord);                         //LO, NLO, NNLO
  QCDNUM::setalf(as0,r20);                      //input alphas
  int iqc = QCDNUM::iqfrmq(q2c);                //charm threshold
  int iqb = QCDNUM::iqfrmq(q2b);                //bottom threshold
  QCDNUM::setcbt(nfin,iqc,iqb,999);             //set VFNS thresholds
  int iq0 = QCDNUM::iqfrmq(q0);                 //start scale
  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 6 – The QCDNUM example program in C++. Note the different indexing of the array `pdf` and that the arguments of `func` are passed as pointers.

in an if-then-else block in the function `func`. The sum rules

$$\begin{aligned}
\int_0^1 xg(x)\mathrm{d}x + \int_0^1 xq_\mathrm{s}(x)\mathrm{d}x &= 1, \\
\int_0^1 [d(x) - \bar{d}(x)]\mathrm{d}x &= 1, \\
\int_0^1 [u(x) - \bar{u}(x)]\mathrm{d}x &= 2
\end{aligned} \tag{4.1}$$

cannot be reliably evaluated by QCDNUM since it has no information on the x-dependence of the pdfs below the lowest grid point in x. These sum rules should therefore be built into the parameterisation of the input densities. The evolution does, of course, conserve the sum rules once they are imposed at μ_0^2. The easiest way to evolve with a symmetric strange sea is to include $xs_\mathrm{v} = x(s - \bar{s})$ in the collection of input densities and set it to zero for all x at the input scale μ_0^2. In the VFNS at LO or NLO, the generated heavy flavour densities $h = (\mathrm{c}, \mathrm{b}, \mathrm{t})$ are always symmetric $(xh - x\bar{h} = 0)$ but this is not true anymore at NNLO, which generates a small asymmetry.

After the parton densities are evolved, the results can be accessed by `fvalxq`. This routine transforms the parton densities from the internal singlet/non-singlet basis to the flavour basis and returns the gluon, a quark, or an anti-quark momentum density, interpolated to x and μ^2. Also here the flavours $\mathrm{d}, \mathrm{u}, \mathrm{s}, \ldots$ are indexed according to the PDG convention. The last call in the example program evolves the input value of α_s to the scale m_Z^2. This evolution is completely stand-alone and does not make use of the μ^2 grid. The function `asfunc` can thus be called at any point after the call to `qcinit`. We refer to Section 5 for more ways to access the QCDNUM pdfs, and for ways to change the renormalisation scale with respect to the factorisation scale.

C++ From the example code you may notice that the FORTRAN array `pdf(-6:6)` is declared as `pdf[13]` in C++, with an index that counts from zero. Thus `pdf(-4)` in FORTRAN becomes `pdf[2]` in C++. For more on the FORTRAN versus C++ correspondence see Section 5.1.

QCDNUM has an extensive checking mechanism which maintains internal consistency and verifies that all subroutine arguments supplied by the user are within their allowed ranges. Error messages might pop-up unexpectedly when the renormalisation scale is changed with respect to the factorisation scale because the low end of the μ^2 grid may then map onto values of $\mu_\mathrm{R}^2 < \Lambda^2$.

Another QCDNUM feature is that $n_\mathrm{f} = (4, 5, 6)$ and not $(3, 4, 5)$ at the heavy flavour thresholds μ_h^2. This implies, first of all, that parton evolution in the VFNS must start from $\mu_0^2 < \mu_\mathrm{c}^2$ and not from $\mu_0^2 \leq \mu_\mathrm{c}^2$, simply because the number of flavours must be $n_\mathrm{f} = 3$ at the starting scale. There is, however, no restriction on the starting (renormalisation) scale of α_s so that it may very well coincide with a flavour threshold, either before or after varying the renormalisation scale with respect to the factorisation scale. If this happens at NNLO, the input value of α_s is assumed to include the discontinuity.

4.3 Validation and Performance

The CPU time that is needed to evolve a pdf on a discrete grid grows quadratic with the number of grid points in x. With linear (quadratic) interpolation the accuracy increases linearly (quadratic) with the number of grid points. It follows that an r-fold gain in accuracy will cost a factor of r^2 in CPU for linear interpolation but only a factor of r for quadratic interpolation. This reduction in cost motivated the inclusion of quadratic splines in QCDNUM.

To investigate the performance of the two interpolation schemes, we compare results from QCDNUM to those from the N-space evolution program PEGASUS [20]. In this comparison a default set of initial distributions [29] is evolved at NNLO from $\mu^2 = 2$ to $\mu^2 = 10^4$ GeV^2 with $n_\mathrm{f} = 4$ flavours. The dashed curve in the top plot of Figure 7 shows

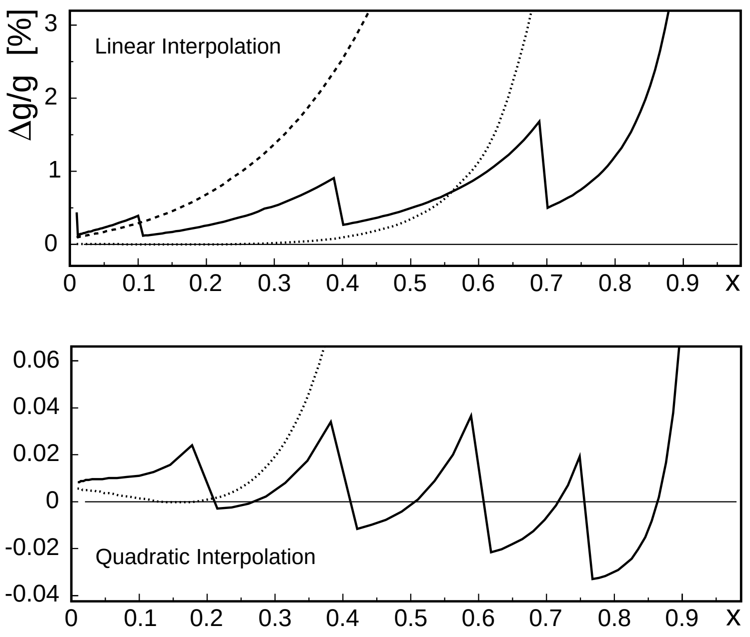


Figure 7 – The relative difference $\Delta g/g$ (in percent) of gluon densities evolved from $\mu^2 = 2$ to $\mu^2 = 10^4$ GeV^2 by QCDNUM and PEGASUS. Top: Evolution with linear splines on a 200 point single grid down to $x = 10^{-5}$ (dashed curve) and on multiple grids (full curve). Bottom: Evolution with quadratic splines on a 100 point single grid (dotted curve, also shown in the top plot) and on multiple grids (full curve). Note the different vertical scales in the two plots.

the relative difference $\Delta g/g$ versus x for QCDNUM evolution with linear splines on a single 200 point grid extending down to $x = 10^{-5}$. The accuracy at low x is satisfactory (few permille) but deteriorates rapidly to $\Delta g/g > 2\%$ for $x > 0.35$.

The precision is much improved by evolving on multiple grids (Section 3.4) as shown by the full curve in the top plot of Figure 7. Here the 200 grid points are re-distributed over five sub-grids with lower limits as given in Table 1. For each successive grid the point density is twice that of the previous grid. It is seen from Figure 7 that the precision is now better than 2% for $x < 0.85$.

The dotted curves in Figure 7 (top and bottom) correspond to evolution with quadratic

Table 1 – Lower x limits of multiple grids used in the evolution with linear and quadratic splines.

	n	x_1	x_2	x_3	x_4	x_5
Linear interpolation	200	10^{-5}	0.01	0.10	0.40	0.70
Quadratic interpolation	100	10^{-5}	0.20	0.40	0.60	0.75
Relative point density		1	2	4	8	16

splines on a single 100 point grid. There is a large improvement in accuracy (more than a factor of 10) compared to linear splines even though the number of grid points is reduced from 200 to 100. However, also here the precision deteriorates with increasing x, reaching a level of 2% at $x = 0.65$. A five-fold multiple grid with lower limits as listed in Table 1 yields a precision $\Delta g/g < 5 \times 10^{-4}$ over the entire range $x < 0.9$ as can be seen from the full curve in the lower plot of Figure 7. Note that this is for evolution up to $\mu^2 = 10^4$ GeV^2; at lower μ^2 the accuracy is even better since it increases (roughly linearly) with decreasing $\ln(\mu^2)$. To fully validate the QCDNUM evolution with PEGASUS,[16] we have made additional comparisons in the FFNS with $n_\mathrm{f} = 3$, 5 or 6 flavours, in the VFNS with and without backward evolution, and with the renormalisation scale set different from the factorisation scale. This for both un-polarised evolution up to NNLO and polarised evolution up to NLO.

As remarked in Section 3.4, the quadratic spline evolution is not more expensive in CPU time than linear spline evolution. On the contrary: QCDNUM runs 4 times *faster* since we need only 100 instead of 200 grid points. With the multiple grid definition given in Table 1 for quadratic splines, the density of the first grid ($x > 10^{-5}$) is 12 points per decade. It follows that for evolution down to $x = 10^{-6}$ (10^{-4}) a grid with $100+12 = 112$ ($100 - 12 = 88$) points should be sufficient.

To investigate the execution speed we did mimic a QCD fit by performing 1000 NNLO evolutions in the VFNS (13 pdfs), using a 60 point μ^2 grid and the 5-fold 100 point x-grid given in Table 1. After each evolution, the proton structure functions F_2 and F_L were computed at NNLO for 1000 interpolation points in the HERA kinematic range. For this test, QCDNUM, MBUTIL, and ZMSTF were compiled with the GFORTRAN compiler, using level 2 optimisation and without array boundary check. The computations took 18.5 CPU seconds on a 2 GHz Intel Core 2 Duo processor under Mac OS-X: 8.5 s for the evolutions and 10 s for the structure functions.

5 Subroutine Calls

In this section we describe all QCDNUM routines, listed in Table 2.

[16]Similar bench-marking between HOPPET [24] and PEGASUS is given in [29] and [30], where also pdf reference tables can be found. We do not provide here benchmark tables for QCDNUM, but a program that generates such tables and compares them with PEGASUS is available upon request from the author.

Table 2 – Subroutine and function calls in QCDNUM.

Subroutine or function	Description
Initialisation	
`QCINIT ( lun, 'filename' )`	Initialise
`SETLUN ( lun, 'filename' )`	Redirect output
`NXTLUN ( lmin )`	Get next free lun
`SET|GETVAL ( 'opt', val )`	Set|Get parameters
`SET|GETINT ( 'opt', ival )`	Set|Get parameters
`QSTORE ( 'action', i, val )`	Store user data
Grid	
`GXMAKE ( xmi, iwt, n, nxin, *nxout, iord )`	Define x grid
`IXFRMX ( x )`	Get i_x from x
`XFRMIX ( ix )`	Get x from i_x
`XXATIX ( x, ix )`	Verify grid point
`GQMAKE ( qarr, wt, n, nqin, *nqout )`	Define $\mu_{\rm F}^2$ grid
`IQFRMQ ( q2 )`	Get i_μ from $\mu_{\rm F}^2$
`QFRMIQ ( iq )`	Get $\mu_{\rm F}^2$ from i_μ
`QQATIQ ( q2, iq )`	Verify grid point
`GRPARS ( *nx, *x1, *x2, *nq, *q1, *q2, *io )`	Get grid definitions
`GXCOPY ( *array, n, *nx )`	Copy x grid
`GQCOPY ( *array, n, *nq )`	Copy μ^2 grid
Weights	
`FILLWT ( itype, *idmi, *idma, *nw )`	Fill weight tables
`DMPWGT ( itype, lun, 'filename' )`	Dump weight tables
`READWT ( lun, 'fn', *idmi, *idma, *nw, *ie )`	Read weight tables
`NWUSED ( *nwtot, *nwuse, *ndummy )`	Memory words used
Parameters	
`SET|GETORD ( iord )`	Set|Get order
`SET|GETALF ( alfs, r2 )`	Set|Get $\alpha_{\rm s}$ start value
`SETCBT ( nfix, iqc, iqb, iqt )`	Set $n_{\rm f}$ or thresholds
`MIXFNS ( nfix, r2c, r2b, r2t )`	Set MFNS parameters
`GETCBT ( *nfix, *q2c, *q2b, *q2t )`	Get $n_{\rm f}$ or thresholds
`NFLAVS ( iq, *ithresh )`	Get $n_{\rm f}$ at i_μ
`SET|GETABR ( ar, br )`	Set|Get $\mu_{\rm R}^2$ scale
`RFROMF ( fscale )`	Convert $\mu_{\rm F}^2$ to $\mu_{\rm R}^2$
`FFROMR ( rscale )`	Convert $\mu_{\rm R}^2$ to $\mu_{\rm F}^2$
`SET|GETCUT ( xmi, q2mi, q2ma, dummy )`	Set|Get cuts
`CPYPAR ( *array, n, iset )`	Copy parameter list
`KEYPAR ( iset )`	Get parameter key
`USEPAR ( iset )`	Activate parameters
`PUSHCP`	Push on a stack
`PULLCP`	Pull from a stack
Evolution	
`ASFUNC ( r2, *nf, *ierr )`	Evolve $\alpha_{\rm s}(\mu_{\rm R}^2)$
`ALTABN ( iset, iq, n, *ierr )`	Returns $a_{\rm s}^n(\mu_{\rm F}^2)$

continued on next page

continued from previous page

`EVOLFG ( iset, func, def, iq0, *eps )`	Evolve all pdfs
`PDFCPY ( iset1, iset2 )`	Copy pdf set
`EXTPDF ( fun, iset, n, offset, *eps )`	Pdfs from outside
`NPTABS ( iset )`	Number of pdf tables
Interpolation	
`BVALXQ|IJ ( iset, id, x|ix, qmu2|iq, ichk )`	Get one basis pdf
`FVALXQ|IJ ( iset, id, x|ix, qmu2|iq, ichk )`	Get one flavour pdf
`ALLFXQ|IJ ( iset, x|ix, qmu2|iq, *pdf, n, ichk )`	Get all flavour pdfs
`SUMFXQ|IJ ( iset, c, isel, x|ix, qmu2|iq, ichk )`	Linear combination
`SPLCHK ( iset, id, iq )`	Check spline
`FSPLNE ( iset, id, x, iq )`	Spline interpolation
`FFLIST ( iset, c, is, x, q, *f, n, ichk )`	Make list of pdfs
`FTABLE ( iset, c, is, x, nx, q, nq, *f, ichk )`	Make table of pdfs
Datacards	
`QCARDS ( mycards, 'filename', iprint )`	Process datacard file
`QCBOOK ( 'action', 'key' )`	Manage keycards

Output arguments are prefixed with an asterisk (∗).

In the following we will prefix output variables with an asterisk (∗). We use the FORTRAN convention that integer variable and function names start with the letters `I`–`N`. Character variables are given in quotes as in `'opt'`. Other variables and functions are in double precision unless otherwise stated. Note that floating point numbers should be entered in double precision format:

```
      ix = ixfrmx ( x )       ! ok
      ix = ixfrmx ( 0.1D0 )   ! ok
      ix = ixfrmx ( 0.1   )   ! wrong!
```

Unlike FORTRAN, QCDNUM is case insensitive so that character arguments like `'ALIM'` or `'Alim'` are both valid inputs.

Most QCDNUM functions will, upon error, generate an error message. The inclusion of function calls in `print` or `write` statements can then cause program hang-up in case the function tries to issue a message. Thus:

```
      write(6,*) 'Glue = ', fvalxq(1,0,x,q,1) ! not recommended

      glue = fvalxq(1,0,x,q,1)                ! OK
      write(6,*) 'Glue = ', glue              ! OK
```

5.1 C++ interface

In this section we describe the correspondence between the QCDNUM calls in FORTRAN and C++. For this we also refer to the listings of the example program shown in Figures 5 and 6 of Section 4.2. A few more remarks and C++ code examples can be found in the subroutine-by-subroutine descriptions given in the next sections.

1. Unlike FORTRAN, C++ code is case-sensitive. We have therefore adopted the convention that the C++ 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 C++ there is no implicit type declaration so that 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`.[17]

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

4. The type of a character input argument 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);
```

5. A FORTRAN array index starts at one unless you specify the index range, as is done for the array `pdf(-6:6)`. However, this is not possible in C++ where arrays always start at index zero. Thus you 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 become one-dimensional arrays in the C++ wrappers; see the `def` array in the listings of Figure 5 and 6. Best is to provide a pointer $k(i,j)$ that maps the indices of a FORTRAN array `A(n,m)` onto those of a C++ array `A[n*m]` with $k(i+1,j) = k(i,j)+1$, $k(i,j+1) = k(i,j)+n$ and $k(1,1) = 0$.

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

```
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. 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 port this to C++ the corresponding function must have its input arguments passed as pointers, as is shown for the input function `func` in the listing of Figure 6.

In this QCDNUM version wrappers are available for the out-of-the-box routines listed in Table 2 (except the steering by data cards) and for the ZMSTF (Table 5) and HQSTF packages (Table 6). The toolbox will be interfaced in a future version.

5.2 Pdf Sets

The QCDNUM internal memory is a linear array that is dynamically partitioned into sets of tables. Its size is given by the parameter `nwf0` in the file `qcdnum.inc`; if you run out of space (error message), you should increase the value of `nwf0` and recompile QCDNUM.

Apart from sets of weight tables, which are managed internally, and a base pdf set ($\texttt{id} = 0$), the memory can hold up to 24 additional pdf sets as given below.[18]

Id	Pdf set	Number of pdf tables	Created/filled by
0	Base set	5 (for internal use)	Managed internally
1	Unpolarised pdfs	13	`EVOLFG`
2	Polarised pdfs	13	`EVOLFG`
3	Fragmentation functions	13	`EVOLFG`
4	User-defined evolution	13	Presently disabled
5–24	External pdfs	$13 + n$	`PDFCPY`, `PDFEXT`, `EVPCOPY`

The base set ($\texttt{id} = 0$) contains the list of active evolution parameters (see Section 5.6), a set of up-to-date α_s tables, and 5 pdf scratch tables for internal use. The pdf sets 1–24 inherit the parameters and α_s tables from the base set at the moment when they are filled by one of routines given in the table above.[19] Each pdf set in internal memory thus remembers its own evolution parameters and α_s tables.

[18]Apart from the base set, the maximum number of sets is $\texttt{mset} = \texttt{mst0}-6 = 24$, where `mst0` can be adjusted in `qcdnum.inc`. The call `getint('mset',mset)` gives you the current value of `mset`.

[19]An exception is the copy routine `pdfcpy` where the parameters and α_s tables are inherited from the source set instead of from the base set.

Note that the pdf sets in QCDNUM contain the gluon and (anti)quark tables for all 6 flavours (13 pdfs). The (anti)quarks are stored as the singlet/non-singlet basis functions $|e^{\pm}\rangle$, defined by (see also Section 2.4)[20]

$$\begin{pmatrix} e_1^{\pm} \\ e_2^{\pm} \\ e_3^{\pm} \\ e_4^{\pm} \\ e_5^{\pm} \\ e_6^{\pm} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ -1 & 1 & & & & \\ 1 & 1 & -2 & & & \\ 1 & 1 & 1 & -3 & & \\ 1 & 1 & 1 & 1 & -4 & \\ 1 & 1 & 1 & 1 & 1 & -5 \end{pmatrix} \begin{pmatrix} d^{\pm} \\ u^{\pm} \\ s^{\pm} \\ c^{\pm} \\ b^{\pm} \\ t^{\pm} \end{pmatrix} \quad \text{with} \quad q_i^{\pm} \equiv q_i \pm \bar{q}_i. \tag{5.1}$$

Apart from the 13 basis pdfs, one can import any number of additional pdfs (photon, for instance) into a set of *external* pdfs.

5.3 Initialisation

```
call QCINIT ( lun, 'filename' )
```

Initialise QCDNUM and define the output stream. Should be called before anything else.

`lun`	Output logical unit number. When set to `6`, QCDNUM messages appear on the standard output. When set to `-6`, the QCDNUM banner printout is suppressed on the standard output.
`'filename'`	Output file name. Irrelevant when `lun` is set to `6` or `-6`.

```
call SETLUN ( lun, 'filename' )
```

Redirect the QCDNUM messages. The parameters are as for `qcinit` above. This routine can be called at any time after `qcinit`.

```
lun = NXTLUN ( lmin )
```

Returns a free logical number `lun` $\geq$ `10`, or `lun` $\geq$ `lmin`, whichever limit is larger. This routine can be called before or after `qcinit`. Returns `0` if there is no free logical unit. Handy if you want to open a file on a unit that is guaranteed to be free.

```
call SETVAL|GETVAL ( 'opt', val )
```

Set or get QCDNUM floating point parameters.

`'null'`	Result of a calculation that cannot be performed. Default, `null` = `1.D11`.
`'epsi'`	The tolerance level in the floating point comparison $\lvert x-y\rvert < \epsilon$, which QCDNUM uses to decide if x and y are equal. Default, `epsi` = `1.D-9`.

[20] Note that $e_2 = u - d$, instead of $d - u$. The reason for this is purely cosmetic: $u - d$ is positive.

`'epsg'`	Required numerical accuracy of the Gauss integration in the calculation of weight tables. Default, `epsg` = `1.D-7`.
`'elim'`	Allowed difference between a quadratic and a linear spline interpolation mid-between the grid points in x. Default, `elim` = `0.5`; larger values may indicate spline oscillation. To disable the check, set `elim` < `0`.
`'alim'`	Maximum allowed value of $\alpha_s(\mu^2)$. When α_s exceeds the limit, a fatal error condition is raised. Default, `alim` = `10`.[21]
`'qmin'`	Smallest possible lower boundary of the μ^2 grid. Default, `qmin` = `0.1` GeV^2.
`'qmax'`	Largest possible upper boundary of the μ^2 grid. Default, `qmax` = `1.D11` GeV^2.

These parameters can be set and re-set at any time after `qcinit`.

```
call SETINT|GETINT ( 'opt', ival )
```

Set or get QCDNUM integer parameters.

`'iter'`	This parameter is only relevant when you evolve in the quadratic interpolation scheme. It then sets the number of iterations to perform when evolving backwards, see Section 3.5. When set negative, you will quadratically evolve backward (not recommended since this is prone to spline oscillations). When set to zero, you will evolve backward in the linear interpolation scheme, without iterations (numerically stable but less accurate). A value larger than zero gives the number of iterations to perform. Default, `iter` = `1`. Note again that all this is irrelevant when QCDNUM runs in the linear interpolation scheme.
`'tlmc'`	Switch the time-like matching conditions off (`0`) or on ($\neq$ `0`). Default is on.
`'nopt'`	Set the number of perturbative terms (`nopt`) in the `evdglap` evolution. In QCD this is 1/2/3 at LO/NLO/NNLO but it may also be something different like 2/3/4 when LO QED corrections are included in the evolution. The number of perturbative terms at different orders is encoded in an integer such as `123` (default) or `234`; see `evdglap` in Section 7.5 for more on this.
`'ichk'`	Check that pdfs are evolved with the current (active) set of evolution parameters (`1` = default) or switch the check off (`0`, not recommended).

And then for `getint` only:

`'vers'`	Returns the current 6-digit version number like `170007`. Can be used to check if QCDNUM is initialised because the version is set to `0` if not.
`'lunq'`	Returns the QCDNUM logical unit number. Useful if you want to write messages on the same output stream as QCDNUM.
`'mset'`	Maximum number of pdf sets allowed in internal memory.
`'mxg0'`...`'nwf0'`	Value of a fixed parameter in `qcdnum.inc` (see Section 4.1).

For instance, `getint('nwf0',nwords)` will give you the size of the internal memory.

[21] When you raise `alim` > `10` then α_s will at some point be limited by internal cuts in QCDNUM.

```
call QSTORE ( 'action', i, val )
```

QCDNUM reserves 100 words of memory for the user to store and retrieve data.[22] This is useful to share data between different parts of a user program. For instance when a function is passed as an argument to a QCDNUM routine one can use `qstore` to transfer some parameter values to this function, other than via the brackets.

C++ In FORTRAN one often transfers data via a common block but not so in C++.

The `'action'` argument can be set to any word starting with the letters `W`, `R`, `L` or `U`.

`'write'`	Write `val` into `qstore(i)`.
`'read'`	Read `val` from `qstore(i)`.
`'lock'`	Make the store read-only.
`'unlock'`	Allow to write into the store (default).

5.4 Grid

A proper choice of the grid in x and μ^2 is important because it determines the speed and accuracy of the QCDNUM calculations.[23] The grid definition also sets the size of the weight and pdf tables and thus the amount of space needed in the internal store.

The x grid must be strictly equidistant in the variable $y = -\ln x$ but in QCDNUM you can generate multiple equidistant grids (Section 3.4) to obtain a finer binning at low y (large x). Multiple grids are generated when the x-range is subdivided into regions with different densities, as is described below.

The μ^2 grid does not need to be equidistant. So you can either enter a fully user-defined grid or let QCDNUM generate one by an equidistant logarithmic fill-in of a given set of intervals in μ^2.

```
call GXMAKE ( xmin, iwt, n, nxin, *nxout, iord )
```

Generate a logarithmic x-grid.

`xmin`	Input array containing `n` values of x in ascending order: `xmin(1)` defines the lower end of the grid while the other values define the approximate positions where the point density will change according to the values set in `iwt`. The list may or may not contain $x = 1$ which is ignored anyway.
`iwt`	Input integer weights. The point density between `xmin(1)` and `xmin(2)` will be proportional to `iwt(1)`, that of the next region will be proportional to `iwt(2)` and so on. The weights should be given in ascending order and must always be an integer multiple of the previous weight. Thus, to give an example, the triplets {`1,1,1`} and {`1,2,4`} are allowed but {`1,2,3`} is not.

[22]The store size can be set by the parameter `mqs0` in `qcdnum.inc`.
[23]We refer to Section 4.3 for recommended grids in the linear and quadratic interpolation schemes.

`n`	The number of values specified in `xmin` and `iwt`. This is also the number of sub-grids used internally by QCDNUM.
`nxin`	Requested number of grid points (not including the point $x = 1$). Should of course be considerably larger than `n` for an x-grid to make sense.
`nxout`	Number of generated grid points. This may differ slightly from `nxin` because of the integer arithmetic used to generate the grid.
`iord`	you should set `iord` = 2 (3) for linear (quadratic) spline interpolation.

With this routine, you can define a (logarithmic) grid in x with higher point densities at large x, where the parton distributions are strongly varying. Thus

```
          xmin = 1.D-4
          iwt  = 1
          call gxmake(xmin,iwt,1,100,nxout,iord)
```

generates a logarithmic grid with exactly 100 points in the range $10^{-4} \leq x < 1$, while

```
          xmin(1) = 1.D-4
          iwt(1)  = 1
          xmin(2) = 0.7D0
          iwt(2)  = 2
          call gxmake(xmin,iwt,2,100,nxout,iord)
```

generates a 100-point grid with twice the point density above $x \approx 0.7$.

A call to `gxmake` invalidates the weight tables and the pdf store.

```
    ix = IXFRMX ( x )      x = XFRMIX ( ix )      L = XXATIX ( x, ix )
```

The function `ixfrmx` returns the index of the closest grid point at or below x. Returns zero if x is out of range (note that $x = 1$ is outside the range) or if the grid is not defined. The inverse function is `x = xfrmix(ix)`. Also this function returns zero if `ix` is out of range or if the grid is not defined. To verify that x coincides with a grid point, use the logical function `xxatix`, as in

```
          logical xxatix
          ix = xfrmix(x)          !x is at or above grid point ix
          if(xxatix(x,ix)) then   !x is at grid point ix
```

Note that QCDNUM snaps to the grid, that is, x is considered to be at a grid point i if $|y - y_i| < \epsilon$ with $y = -\ln x$ and, by default, $\epsilon = 10^{-9}$.

C++ In the C++ interface a logical function returns an `int` with `0` ($\neq$`0`) evaluating as `false` (`true`).

```
        int gridpoint = QCDNUM::xxatix(x,ix);
        if(gridpoint) { ... }
```

```
call GQMAKE ( qarr, wgt, n, nqin, *nqout )
```

Generate a logarithmic μ_F^2 grid on which the parton densities are evolved.[24]

qarr	Input array containing n values of μ^2 in ascending order: qarr(1) and qarr(n) define the lower and upper end of the grid, respectively. The lower end of the grid should be above 0.1 GeV^2. If $n > 2$ then the additional points specified in qarr are put into the grid. In this way, you can incorporate a set of starting values μ_0^2, or thresholds $\mu_{c,b,t}^2$.
wgt	Input array giving the relative grid point density in each region defined by qarr. The weights are not restricted by integer multiples as in gxmake but can be set to any value in the range $0.1 \le w \le 10$. With these weights, you can generate a grid with higher density at low values of μ^2 where α_s is changing rapidly.
n	The number of values specified in qarr and wgt ($n \ge 2$).
nqin	Requested number of grid points. The routine generates these grid points by a logarithmic fill-in of the regions defined above. When nqin = n the grid is not generated but taken from qarr. This allows you to read-in your own μ^2 grid.
nqout	Number of generated grid points. This may differ slightly from nqin because of the integer arithmetic used to generate the grid.

A call to gqmake invalidates the weight tables and the pdf store.

```
iq = IQFRMQ ( q2 )     q2 = QFRMIQ ( iq )     L = QQATIQ ( q2, iq )
```

The function iqfrmq returns the index of the closest grid point at or below μ^2. The inverse function is qfrmiq. To verify that μ^2 coincides with a grid point, use the logical function qqatiq. As described above for the corresponding x grid routines, a value of zero is returned if q2 and/or iq are not within the range of the current grid, or if the grid is not defined.

```
call GRPARS ( *nx, *xmi, *xma, *nq, *qmi, *qma, *iord )
```

Returns the current grid definitions

nx	Number of points in the x grid not including $x = 1$.
xmi	Lower boundary of the x grid.
xma	Upper boundary of the x grid. Is always set to xma = 1.
nq	Number of points in the μ^2 grid.
qmi	Lower boundary of the μ^2 grid.
qma	Upper boundary of the μ^2 grid.
iord	Order of the spline interpolation (2 = linear, 3 = quadratic).

[24] Note that α_s is evolved (without using a grid) on μ_R^2 which may or may not be different from μ_F^2.

```
call GXCOPY ( *array, n, *nx )
```

Copy the x grid to a local array

`array`	Local array containing on exit the x grid but not the value $x = 1$.
`n`	Dimension of `array` as declared in the calling routine.
`nx`	Number of grid points copied to the local array. A fatal error occurs if `array` is not large enough to contain the current grid.

```
call GQCOPY ( *array, n, *nq )
```

As above, but now for the μ^2 grid.

5.5 Weights

In this section we describe routines to calculate the weight tables, to dump these to disk and to read them back. The weight tables are calculated for all orders (LO,NLO,NNLO) and all number of flavours $n_f = (3, 4, 5, 6)$, irrespective of the current QCDNUM settings. There is thus no need to re-calculate the weights when one or more of these settings are changed later on.

Weight tables can be created for un-polarised pdfs, polarised pdfs and fragmentation functions and these can all exist simultaneously in memory.

```
call FILLWT ( itype, *idmin, *idmax, *nwds )
```

Partition the pdf store and fill the weight tables used in the calculation of the convolution integrals. Both the x and μ^2 grid must have been defined before the call to `fillwt`.

`itype`	Select un-polarised pdfs (1), polarised pdfs (2) or fragmentation functions (3). Any other input value will select un-polarised pdfs (default).
`idmin`	Returns, on exit, the identifier of the first pdf table. Always `idmin` = 0.
`idmax`	Identifier of the last pdf table in the store. Always `idmax` = 12.
`nwds`	Total number of words used in memory.

You can create more than one set of tables tables by calling `fillwt` with different values of `itype`. For instance, the sequence

```
        call fillwt(1,idmin,idmax,nw)   !Unpolarised pdfs
        call fillwt(2,idmin,idmax,nw)   !Polarised pdfs
```

makes both the un-polarised and the polarised weights available. If there is not enough space in memory to hold all the tables, `fillwt` returns with an error message telling how much memory it needs. You should then increase the value of `nwf0` in the include file `qcdnum.inc`, and recompile QCDNUM.[25] Note that `fillwt` acts as a do-nothing when the pdf type already exists in memory:

[25] This you may have to repeat several times because `fillwt` proceeds in stages and will report the memory needs of the current stage, but not beyond.

```
      call fillwt(1,idmin,idmax,nw)   !Unpolarised pdfs
      call fillwt(1,idmin,idmax,nw)   !Do nothing
```

```
call DMPWGT( itype, lun, 'filename' )
```

Dump the weight tables (not the pdf tables) of a given pdf type [1–3] to disk. Fatal error if `itype` does not exist. Additional information about the QCDNUM version, grid definition and partition parameters is also dumped, to protect against corruption of the dynamic store when the weights are read back in future QCDNUM runs. The dump is un-formatted so that the output file cannot be exchanged across machines. You can use the function `nxtlun` of Section 5.3 to find a free logical unit number.

```
call READWT( lun, 'fname', *idmin, *idmax, *nwds, *ierr )
```

Read the weight tables from a disk file. Both the x and μ^2 grid must have been defined before the call to `readwt`. Like in `fillwt`, you will get a fatal error message if there is not enough space in memory to hold all the tables. Otherwise, `ierr` is set as follows:

0	Weights are successfully read in.
1	Read error or input file does not exist.
2	Input file was written with another QCDNUM version.
3	Key mismatch (should never occur).
4	Incompatible x-μ^2 grid definition.

When successful (`ierr` = 0), the routine returns on exit the parameters `idmin`, `idmax` and `nwds` as does the subroutine `fillwt`. Upon failure (`ierr` $\neq$ 0), the routine acts as a do-nothing in which case you should create the weights from scratch, as in

```
    lun = nxtlun(0)      !find free logical unit number
    call readwt(lun,'polarised.wgt',idmin,idmax,nw,ierr)
    if(ierr.ne.0) then
      call fillwt(2,idmin,idmax,nw)
      call dmpwgt(2,lun,'polarised.wgt')
    endif
```

The code above thus automatically maintains an up-to-date weight file on disk.

```
call NWUSED( *nwtot, *nwuse, *ndummy )
```

Returns the size `nwtot` of the QCDNUM store (the parameter `nwf0` in `qcdnum.inc`) and the number of words used (`nwuse`). The parameter `ndummy` is reserved for later use.

5.6 Parameters

In this section we describe the QCDNUM routines to set evolution parameters like the perturbative order, flavour thresholds, α_s, *etc.* All these parameters have reasonable defaults but you can change them at any point in the code. A call to one of the routines below will update the internal parameter list which we call *active* because they steer to a great extent the behaviour of QCDNUM. In particular, the internal set of α_s tables and pointer tables for the grid indexing do depend on one or more of these parameters.

When you evolve (Section 5.7) or import (Section 5.8) a pdf set, the parameters and α_s tables are copied to that set. Each pdf set thus remembers its own evolution parameters.

```
call SETORD|GETORD ( iord )
```

Set (or get) the order of the QCDNUM calculations to 1, 2 or 3 for LO, NLO and NNLO, respectively. Default, `iord` = 2.

```
call SETALF|GETALF ( alfs, r2 )
```

Set or get for the α_s evolution the starting value `alfs` and the starting renormalisation scale `r2`. Default $\alpha_s(m_Z^2) = 0.118$.

```
call SETCBT( nfix, iqc, iqb, iqt )
```

Select the FFNS or VFNS mode, and set thresholds on μ_F^2.

`nfix`	Number of flavours in the FFNS mode. If not set to 3, 4, 5 or 6, QCDNUM runs in the VFNS mode.
`iqc,b,t`	Grid indices of the quark mass thresholds $\mu^2_{c,b,t}$. This input is ignored when QCDNUM runs in the FFNS mode, that is, when `nfix` is set to 3, 4, 5 or 6. There are some restrictions, dictated by the evolution and interpolation routines: `iqc` $\geq$ 2, `iqb` $\geq$ `iqc+2` and `iqt` $\geq$ `iqb+2`.

A threshold index value of zero (or larger than the number of grid points) means 'beyond the upper edge of the grid'. For instance, setting (`iqc,b,t`) = (`0,0,0`) is equivalent to setting FFNS with 3 flavours, while the setting (`2,4,0`) puts the top quark threshold beyond the evolution range. By default, QCDNUM runs in the FFNS with $n_f = 3$.

```
call MIXFNS ( nfix, r2c, r2b, r2t )
```

Select the MFNS mode, and set thresholds on μ_R^2.

`nfix`	Fixed number of flavours [3–6] in the pdf evolution.
`r2c,b,t`	Thresholds defined on the renormalisation scale μ_R^2. When crossing a threshold, n_f changes in the β-functions but not in the splitting functions.

To put a threshold below the evolution range, set it to a value ≤ 0. For instance,

```
call mixfns ( 3, r2c ,  r2b,  r2t )  !nf(pdf)= 3   nf(as)= 3,4,5,6
call mixfns ( 4, 0.D0,  r2b,  r2t )  !nf(pdf)= 4   nf(as)= 4,5,6
call mixfns ( 5, 0.D0, 0.D0,  r2t )  !nf(pdf)= 5   nf(as)= 5,6
call mixfns ( 6, 0.D0, 0.D0, 0.D0 )  !nf(pdf)= 6   nf(as)= 6
```

To put the threshold above the evolution range, set it to a large value (please note that non-zero thresholds must be in ascending order, as is shown in the calls below):

```
call mixfns ( 3, 1.D9, 2.D9, 3.D9 )  !nf(pdf)= 3   nf(as)= 3
call mixfns ( 4, 0.D0, 2.D9, 3.D9 )  !nf(pdf)= 4   nf(as)= 4
call mixfns ( 5, 0.D0, 0.D0, 3.D9 )  !nf(pdf)= 5   nf(as)= 5
call mixfns ( 6, 0.D0, 0.D0, 0.D0 )  !nf(pdf)= 6   nf(as)= 6
```

These calls are equivalent to calling `setcbt` with `nf` = `3,4,5,6`, respectively.

```
call GETCBT( *nfix, *q2c, *q2b, *q2t )
```

Return the current threshold settings.

When `nfix` = `0` on return, QCDNUM runs in the VFNS and the routine returns the threshold values (not the indices) on the μ_F^2 scale.

When `nfix` = `+`(`3`,`4`,`5`,`6`) on return, QCDNUM runs in the FFNS and the values of `q2c`,`b`,`t` are irrelevant.

When `nfix` = `-`(`3`,`4`,`5`,`6`) on return, QCDNUM runs in the MFNS and the routine returns the threshold values on the μ_R^2 scale.

```
nf = NFLAVS( iq, *ithresh )
```

Get the number of flavours at a μ^2 grid point.

`iq`	Grid point in μ^2. The function returns `nf` = `0` if `iq` is outside the μ^2 grid.
`ithresh`	Threshold indicator that is set to `+1` (`-1`) if `iq` is at a threshold with the larger (smaller) number of flavours, to `0` otherwise.

Note that `nflavs` will return (`4`,`5`,`6`) at the thresholds (`iqc`,`b`,`t`), as is the QCDNUM convention, unless you set `iq` negative in which case it will return (`3`,`4`,`5`).

```
nf = nflavs( iqc,ithresh)  !nf = 4 and ithresh =  1
nf = nflavs(-iqc,ithresh)  !nf = 3 and ithresh = -1
```

```
call SETABR|GETABR ( ar, br )
```

Define the relation between the factorisation scale μ_F^2 and the renormalisation scale μ_R^2

$$\mu_\mathrm{R}^2 = a_\mathrm{R}\,\mu_\mathrm{F}^2 + b_\mathrm{R}.$$

Default: `ar` = `1` and `br` = `0`.

```
rscale2 = RFROMF( fscale2 )     fscale2 = FFROMR( rscale2 )
```

Convert the factorisation scale μ_F^2 to the renormalisation scale μ_R^2 and *vice versa*.

```
call SET|GETCUT ( xmi, q2mi, q2ma, dummy )
```

Restrict the kinematic range of a pdf evolution to a part of the x-μ^2 grid.[26]

`xmi`, `q2mi`, `q2ma`	Re-define the x-μ^2 range. To release a cut, enter a value of zero (or outside the current grid boundaries). Acts as a do-nothing if the cuts would lead to an empty kinematic domain.
`dummy`	Not used at present.

A call to `getcut` returns the values of the current cuts.

Restricting the evolution to the kinematic range of the data is a nice way to save CPU time in the χ^2 minimisation stage of a QCD fit. When the fit has converged, you can release the cuts and evolve, just once, over the full grid to obtain the final pdf set.

```
call CPYPAR ( *array, n, iset )
```

Copy the evolution parameters stored in a pdf set to a local array.

`array` Double precision array, dimensioned to at at least `array(13)` in the calling routine. On exit, the array will be filled with the following parameter values:[27]

1	`iord`	2	`alfas`	3	`r2alf`	4	`nfix`	5	`q2c`	6	`q2b`
7	`q2t`	8	`ar`	9	`br`	10	`xmin`	11	`qmin`	12	`qmax`

In `array(13)` is returned the type of pdf stored in `iset`:

`1` = unpolarised, `2` = polarised, `3` = time-like, `4` = user, `5` = external.

`n` Dimension of `array` as declared in the calling routine. Fatal error if `n` < `13`.

`iset` Pdf set identifier in the range `0`–`24`. When `iset` = `0`, the internal set of parameters is returned (this is like calling all of `getord`, `getalf`, *etc.*). Fatal error if `iset` does not exist.

```
key = KEYPAR ( iset )
```

Give the parameter key of a pdf set (`iset` = `0` selects the internal parameters). Useful to quickly check if parameter lists are (un)equal since then the keys are (un)equal:

[26]You can also use `setcut` to define the kinematic domain of an external pdf set (see Section 5.8).

[27]Note that integer parameters are returned as double precision numbers so that you should take care of the type conversion yourself, like `iord = array(1)` or `iord = int(array(1))`, *etc.*

```
    if( keypar(iset).eq.keypar(0) ) !pdfs are evolved with current params
```

`call USEPAR ( iset )`

Copy the parameters of a pdf set back into the internal (active) parameter list.

`iset`	Pdf set identifier in the range 1–24. Fatal error if `iset` does not exist.

This routine activates the parameters of `iset` which may be necessary because QCDNUM blocks access to pdfs that are not evolved with the current (active) set of parameters.

`call PUSHCP|PULLCP`

The routine `pushcp` pushes the current (active) parameter set on a LIFO stack (5-deep), and `pullcp` pulls them out again.[28] In this way you can *temporarily* activate your favourite set of parameters, as is shown in the example code below.

```
          if(keypar(iset).eq.keypar(0)) then
            call ftable(iset,...)       !no need to activate iset
          else
            call pushcp                 !save the current parameters
            call usepar(iset)           !activate the parameters of iset
            call ftable(iset,...)       !make table of pdfs from iset
            call pullcp                 !restore the current parameters
          endif
```

C++ Be aware of the brackets in the wrappers `QCDNUM::pushcp()` and `QCDNUM::pullcp()`.

Note that frequent activation of parameter sets (with or without a push/pull) can become quite expensive in CPU since these calls (partially) re-initialise QCDNUM.

5.7 Evolution

`alphas = ASFUNC( r2, *nf, *ierr )`

Standalone evolution of α_s on the renormalisation scale μ_R^2 (without using the μ^2 grid or weight tables). QCDNUM internally keeps track of α_s so that there is no need to call this function; it is just a user interface that gives access to $\alpha_s(\mu_R^2)$.

`r2`	Renormalisation scale μ_R^2 where α_s is to be calculated.
`nf`	Returns, on exit, the number of flavours at the scale `r2`.

[28] Because of the LIFO stack, a routine that does a push/pull can call another routine that does a push/pull and so on, up to a nesting of 5-deep which should be more than enough.

`ierr = 1` Too low value of `r2`. Internally, there is a cut `r2` > 0.1 GeV^2 and also a cut on the slope, to avoid getting too close to Λ^2.

The input scale and input value of α_s, the order of the evolution and the flavour thresholds are those set by default or by the routines described in Section 5.6. Note that although α_s is evolved on the renormalisation scale the result, in the VFNS, may still depend on the relation between μ^2_R and μ^2_F. This is because the position of the heavy flavour thresholds depends on this relation.[29]

It is important to realise, however, that $\alpha_\mathrm{s}(\mu^2_\mathrm{R})$ returned by `asfunc` is *not* the right expansion coefficient in a QCDNUM perturbative series because α_s must then be given at the factorisation scale μ^2_F instead of at the renormalisation scale μ^2_R. The relation between $\alpha_\mathrm{s}(\mu^2_\mathrm{R})$ and $\alpha_\mathrm{s}(\mu^2_\mathrm{F})$ is given by the truncated Fourier series (2.17) where each power of α_s is computed seperately. Note that the truncation is different for the splitting function expansion $(\alpha_\mathrm{s}, \alpha^2_\mathrm{s}, \alpha^3_\mathrm{s})$ and the structure function expansion $(1, \alpha_\mathrm{s}, \alpha^2_\mathrm{s})$.

To keep track of all this, QCDNUM maintains tables of $\alpha^n_\mathrm{s}(\mu^2_\mathrm{F})$ in terms of powers of $a_\mathrm{s} \equiv \alpha_\mathrm{s}/2\pi$ and keeps them always up-to date. The following function gives you access to these internal tables, or to those stored in one of the pdf sets `1-24`.

```
asn = ALTABN ( iset, iq, n, *ierr )
```

Returns the value of $(\alpha_\mathrm{s}/2\pi)^n$ at the factorisation scale μ^2_F.

`iset`	Select the internal α_s table (`0`), or the one from a pdf set (`1`–`24`). Fatal error if `iset` does not exist or is not evolved with the current (active) set of parameters.
`iq`	Index of a μ^2 grid point.
`n`	Power of α_s which should be set as follows for the different perturbative series.

`n` = `1`, `2`, `3`, ...	for the series	α_s,	α^2_s,	α^3_s,	...
`n` = `0`, `-1`, `-2`, ...	for the series	1,	α_s,	α^2_s,	...

`ierr` Set, on exit, to `1` if `iq` is close to or below the value of Λ^2, and to `2` if `iq` is outside the grid boundaries. Upon error, the function returns a `null` value.

To have access to the NNLO discontinuities at the thresholds, you can set `iq` positive (takes n_f above threshold, QCDNUM default) or negative (n_f below threshold), thus:

```
      asn = altabn ( iset,  iqc, n, ierr )    !result for nf = 4
      asn = altabn ( iset, -iqc, n, ierr)     !result for nf = 3
```

In other words, by prefixing `iq` with a minus sign you effectively change the QCDNUM convention $n_\mathrm{f} = (4, 5, 6)$ at the thresholds to the alternative $n_\mathrm{f} = (3, 4, 5)$.

Note that the QCDNUM expansion parameter is $\alpha_\mathrm{s}/2\pi$ but that many convolution kernels found in the literature are defined for an expansion in $\alpha_\mathrm{s}/4\pi$, in which case you must be careful to account somewhere for the missing factors of 2^n.

[29]If $\mu^2_\mathrm{R} = a\,\mu^2_\mathrm{F} + b$ then the flavour thresholds are similarly related: $\mu^2_{h,\mathrm{R}} = a\,\mu^2_{h,\mathrm{F}} + b$. In this way, n_f changes simultaneously in both the splitting and the β-functions, as required (see Section 2.5).

```
call EVOLFG( iset, func, def, iq0, *epsi )
```

Evolve a set of parton *momentum* densities from an input scale μ_0^2. If QCDNUM runs in the FFNS, the gluon and $2n_\mathrm{f}$ quark densities must be given as an input at μ_0^2. In the VFNS, the gluon and $2n_\mathrm{f} = 6$ light quark densities must be given at $\mu_0^2 < \mu_\mathrm{c}^2$.

Here and in the following the parton densities are written on the flavour basis (note the PDG convention) with an indexing defined by

$$\begin{array}{ccccccccccccc} -6 & -5 & -4 & -3 & -2 & -1 & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline \bar{t} & \bar{b} & \bar{c} & \bar{s} & \bar{u} & \bar{d} & g & d & u & s & c & b & t \end{array} \tag{5.2}$$

`iset`	Pdf set identifier: un-polarised (`1`), polarised (`2`) or time-like (`3`).
`func`	User defined function `func(j,x)` (see below) that returns the input parton momentum density $xf_j(x)$ at `iq0`. Must be declared `external` in the calling routine. The index `j` runs from 0 (gluon) to $2n_\mathrm{f}$.
`def`	Input array dimensioned in the calling routine to `def(-6:6,12)` which contains in `def(i,j)` the contribution of parton species `i` to the input distribution `j`, that is, `def(i,j)` specifies the flavour decomposition of all input distributions `j`. The indexing of `i` is given in (5.2). Internally, QCDNUM constructs from `def` a $2n_\mathrm{f} \times 2n_\mathrm{f}$ sub-matrix of coefficients and tries to invert this matrix. If that fails, the $2n_\mathrm{f}$ input densities are not linearly independent in flavour space and an error condition is raised; see also (2.32) and (F.8) in Appendix F.5.
`iq0`	Grid index of the starting value μ_0^2. When evolving in the FFNS the input scale can be anywhere inside the range of the μ^2 grid or cuts. In the VFNS, however, μ_0^2 should be below the charm threshold.
`epsi`	Maximum deviation of the quadratic spline interpolation from linear interpolation mid-between the grid points (see Section 3.4). By definition, `epsi` $= 0$ when QCDNUM is run in the linear interpolation scheme but for quadratic interpolation a large value `epsi` $>$ `elim` may indicate spline oscillation and will cause a fatal error message. The value of `elim` can be set by a call to `setval`. When `elim` $\leq$ `0` the error message—and program stop—is suppressed so that you can investigate the cause of oscillation with the routines `splchk` and `fsplne`, as is described Section 5.9.

The input function `func` must be coded as follows

```
      double precision function func(ipdf,x)
      implicit double precision (a-h,o-z)
      if(ipdf.eq.0) then
        func = xgluon(x)                    !0 = gluon  xg(x)
      elseif(ipdf.eq.1) then
        func = my_favourite_quark_dstn_1(x)  !1 = quarks xq1(x)
      elseif(ipdf.eq.2) then
        func = my_favourite_quark_dstn_2(x)  !2 = quarks xq2(x)
      elseif(ipdf.eq.3) then
        ..
      endif
```

```
          return
          end
```

C++ The C++ function prototype is (see also the listing Figure 6 in Section 4.2):

```
          double func( int* ipdf, double* x )
```

Because `evolfg` will call `func` only at the grid points x_i, it is possible to feed tabulated values into the evolution routine as is illustrated by the following code

```
          double precision function pdfinput(ipdf,x)
          implicit double precision (a-h,o-z)
          common /input/ table(0:12,nxx)    !table with input values
          ix       = ixfrmx(x)
          pdfinput = table(ipdf,ix)
          return
          end
```

Here is code that evolves both un-polarised and polarised pdfs.

```
          call fillwt(1, idmin, idmax, nw)           !unpolarised
          call fillwt(2, idmin, idmax, nw)           !polarised
            ..
          call evolfg(1, func1, def1, iq01, epsi1)   !unpolarised
          call evolfg(2, func2, def2, iq02, epsi2)   !polarised
```

5.8 Pdf Import

Additional pdf sets with identifiers `5–24` can be stored in QCDNUM memory by (i) copying an internal set to another location, (ii) importing a pdf set from some external source and (iii) importing a pdf set from a toolbox workspace (see Section 7.5).

As mentioned in Section 5.2, all pdf sets in QCDNUM contain a list of evolution parameters, α_s tables, and the full set of gluon, singlet and non-singlet basis tables (13 pdfs). Apart from the 13 basis pdfs, one can import any number of additional pdfs into the set like, for instance, the photon.

```
call PDFCPY ( iset1, iset2 )
```

Copy an internal pdf set to another location.

`iset1` Identifier `1–3` of a pdf set previously evolved with `evolfg`. Fatal error if `iset1` does not exist.

`iset2` Identifier `5–24` of the target set. If `iset2` does not exist it is created (with 13 pdf tables), otherwise it is overwritten. Fatal error if the internal memory is not large enough to hold `iset2`.

Here is example code that stores LO/NLO/NNLO unpolarised pdfs in memory.

```
      do iord = 1,3
        call setord(iord)
        call evolfg(1, func, def, iq0, epsi)
        call pdfcpy(1, 4+iord)
      enddo
```

`call EXTPDF ( fun, iset, n, offset, *epsi )`

Import a pdf set from an external source.

`fun`	User function (see below), declared `external` in the calling routine.
`iset`	Pdf set identifier in the range 5–24. If `iset` does not exist it is created (with $13 + n$ pdf tables), otherwise it is overwritten. Fatal error if there is not enough memory to hold `iset` or if an existing set cannot hold $13 + n$ pdfs.
`n`	Number of extra pdfs to be imported, beyond the 13 quark and gluon pdfs.
`offset`	Relative offset δ at the thresholds μ_h^2 which is used to catch discontinuities at the thresholds, if any, by sampling the pdfs at $\mu_h^2(1 \pm \delta)$. Usually you can set δ to a small value like 10^{-3}, but this depends on the external representation of the discontinuities, and on how accurate the thresholds are set in QCDNUM.[30]
`epsi`	Maximum deviation of the quadratic spline interpolation from linear interpolation mid-between the grid points, as is described for the routine `evolfg`.

The function `fun` provides the interface between QCDNUM and the external repository:

```
      double precision function fun ( ipdf, x, qmu2, first )
      implicit double precision (a-h,o-z)
      logical first
      if(first) some initialisation, if any
      if(i .eq. -6) fun = xTopBar(x,qmu2)
      if(i .eq. -5) ...
```

C++ The C++ prototype is `double fun( int* i, double* x, double* q, bool* f )`.

The index `ipdf` runs from `-6` to `6`, indexed according to (5.2) in the PDG convention. For the extra pdfs, if any, the index is `7` ≤ `ipdf` ≤ `6+n`. The first time `func` is called by `extpdf` the flag `first` is set to `true` and you can initialise the function, if needed.

C++ If arguments are passed other than through the function brackets you can do this via a common block in FORTRAN or make use of `qstore`. Here is a C++ example that uses `extpdf` to copy a pdf set (as does `pdfcpy`). The pdf set identifier is passed to `fun` via `qstore`; note that `iset` must be declared `static` in the body of `fun`.

```
    double fun( int* i, double* x, double* q, bool* first ){
      static int iset;  double val;
      if(*first) { QCDNUM::qstore("Read",1,val); iset = int(val); }
      return QCDNUM::fvalxq(iset,*i,*x,*q,1);
      }
    ...
```

[30] If there is a sharp step at μ_h^2 you can set $\delta = 0$ to let QCDNUM bracket μ_h^2 as tightly as possible.

```
    QCDNUM::qstore("Write",1,double(iset1));
    QCDNUM::extpdf(fun,iset2,0,offset,epsi); //this copies iset1 to iset2
```

It is important that the perturbative order, flavour scheme, positions of the thresholds and the input value of α_s are set correctly in QCDNUM before the call to `extpdf`. Otherwise the pdf set may inherit incorrect evolution parameters and α_s tables, or might suffer from corrupted threshold discontinuities if the thresholds are set wrongly. Note also that the kinematic cuts, if any, will be applied.

```
ntabs = NPTABS(iset)
```

Returns the number of pdf tables in `iset` = `1`–`24`. Can be used to check if `iset` exists, because `nptabs` returns `0` if not.

5.9 Pdf Interpolation

There are several routines to access the flavour *momentum* densities $|xg\rangle$, $|xq\rangle$ and $|x\bar{q}\rangle$, or one of the basis pdfs $|xe^{\pm}\rangle$ as defined by (5.1). The pdfs can be interpolated to an (x, μ^2) point inside the grid, or just be returned at a given grid point.

Access is limited to pdf sets that are evolved with the current (active) set of evolution parameters; this restriction protects you against an unintentional mix of incompatible pdf sets in a QCD calculation. If you want to use pdfs in memory that are evolved with a different set of parameters then you have to first activate these parameters as is described in Section 5.6. The code below shows how to *temporarily* activate the parameters of a pdf set by first saving and then restoring the current parameters.

```
      call   PUSHCP                        !save current parameters
      call   USEPAR(iset)                  !activate iset
      pdf = FVALXQ (iset, id, x, q, ichk)  !get pdf value
      call   PULLCP                        !restore current params
```

Below we give the indexing of the flavour pdfs (note the PDG convention)

$$\begin{array}{cccccccccccccc|c} -6 & -5 & -4 & -3 & -2 & -1 & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & \cdots \\ \hline \bar{t} & \bar{b} & \bar{c} & \bar{s} & \bar{u} & \bar{d} & g & d & u & s & c & b & t & f_1 & \ldots \end{array} \quad (5.3)$$

and that of the basis pdfs

$$\begin{array}{ccccccccccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & \cdots \\ \hline g & q_s & e_2^+ & e_3^+ & e_4^+ & e_5^+ & e_6^+ & q_v & e_2^- & e_3^- & e_4^- & e_5^- & e_6^- & f_1 & \ldots \end{array} \quad (5.4)$$

Here f_i is an imported pdf beyond the gluon or quarks with an index $6+i$ on the flavour basis, and $12+i$ on the internal basis.

```
pdf = BVALXQ ( iset, id, x, qmu2, ichk )
```

Returns one of the basis pdfs $|xg\rangle$ or $|xe^{\pm}\rangle$, interpolated to x and μ^2.

`iset`	Pdf set identifier of un-polarised (`1`), polarised (`2`), fragmentation function (`3`), or external pdfs (`5`–`24`).
`id`	Basis pdf identifier running from `0` to `12+n`, indexed as given in (5.4). Here `n` is the number of extra pdfs in `iset`, beyond the gluon and the quarks.
`x`, `qmu2`	Input value of x and μ^2.
`ichk`	Input flag that steers the error checking.

`0`	Return a `null` value when `x` or `qmu2` are outside the grid boundaries.
`+1`	Fatal error message when `x` or `qmu2` are outside the grid boundaries.
`-1`	As above, but do not check the input values of `iset` and `id`.

Thus `ichk` = `-1` makes the routine faster but see the remark at the end of this section.

```
pdf = FVALXQ ( iset, id, x, qmu2, ichk )
```

Returns $|xg\rangle$, $|xq\rangle$ or $|x\bar{q}\rangle$. The arguments are the same as for `bvalxq` except that the pdf identifier runs from `-6` to `6+n`, indexed according to (5.3) . Note that `fvalxq` is slower than `bvalxq` because a linear combination of basis pdfs has to be taken.

```
call ALLFXQ ( iset, x, qmu2, *pdf, n, ichk )
```

Returns all flavour-pdf values in one call. The arguments are as given above, except

`pdf`	Output array, dimensioned to `pdfs(-6:6+n)` in the calling routine.
`n`	Number of extra pdfs (those beyond the gluon and quarks) to be returned in `pdfs`, provided that they exist in `iset` (fatal error if not).

```
pdf = SUMFXQ ( iset, c, isel, x, qmu2, ichk )
```

Return the gluon or a weighted sum of quark densities, depending on the selection flag `isel`. The arguments are as given above, except

`c`	Input array, dimensioned to `c(-6:6)` in the calling routine, containing the coefficients of a linear combination of quarks and antiquarks, indexed according to (5.3). Note that `c(0)` is ignored since it does not correspond to a quark.
`isel`	Selection flag that determines what is actually returned.

`0`	Return the gluon density $|xg\rangle$.
`1`	Return the linear combination of (anti)quarks as specified in `c`.
`2`–`8`	Return a specific singlet/non-singlet quark component (see below).
`12+i`	Return the extra pdf $|xf_i\rangle$, if present in `iset` (fatal error if not).

To describe the setting of `isel` = `1`–`8` we write the quark linear combination (2.29) as

$$|xp\rangle \;=\; \overbrace{d_1^+|xe_1^+\rangle}^{\rm si} \;+\; \overbrace{\sum_{i=2}^{n_{\rm f}} d_i^+|xe_i^+\rangle}^{\rm ns^+} \;+\; \overbrace{d_1^-|xe_1^-\rangle}^{\rm va} \;+\; \overbrace{\sum_{i=2}^{n_{\rm f}} d_i^-|xe_i^-\rangle}^{\rm ns^-}. \qquad (5.5)$$

The `isel` flag selects specific combinations of the singlet and nonsinglet components in (5.5), as is needed in structure function calculations where, depending on the perturbative order, each component may convolute with a different coefficient function.

`isel` = `1` :	si + ns$^+$ + va + ns$^-$	`2` :	si	`3` :	ns$^+$ + va + ns$^-$	`4` :	ns$^+$
`5` :	va + ns$^-$	`6` :	ns$^-$	`7` :	va	`8` :	$d_1^+|xg\rangle$

Note that `isel` = `8` returns a weighted gluon which is useful since a singlet convolution $C_{\rm s} \otimes d_1^+|e_1^+\rangle$ is usually accompanied by a gluon convolution $C_{\rm g} \otimes d_1^+|g\rangle$.

There are also routines that return the value of a pdf at a given grid point (`ix`,`iq`). They have the same argument list as their interpolation equivalents, except that `x` and `qmu2` must be replaced by the grid indices `ix` and `iq`, as shown below.

```
pdf  =  bvalij( iset, id, ix, iq, ichk )
pdf  =  fvalij( iset, id, ix, iq, ichk )
call    allfij( iset, ix, iq, pdf, n, ichk )
pdf  =  sumfij( iset, c, isel, ix, iq, ichk )
```

These routines are of course faster since no interpolation is required. They have the additional feature that a negative `iq` will return the pdfs at the flavour thresholds for $n_{\rm f} = (3, 4, 5)$, instead of the QCDNUM standard $n_{\rm f} = (4, 5, 6)$. Here is the un-polarised gluon discontinuity at the charm threshold:

```
disc = fvalij(1,0,ix,iqc,ichk) - fvalij(1,0,ix,-iqc,ichk)
```

Note that a routine like `bvalij` is very fast because it simply transfers a value stored in the internal memory. The CPU time is thus spent in computing a memory address and checking the argument list of the call. Setting `ichk` = `-1` switches off part of the checking so that you can optimise calls in a loop, as in the example below.

```
pdf(1) =   bvalij( iset, id, ix(1), iq(1),  1 ) !check arguments
do i = 2,n
  pdf(i) = bvalij( iset, id, ix(i), iq(i), -1 ) !no check
enddo
```

For `bvalij` this gives about a factor of two in speed but the gain is much less for the other routines where most of the time is spent in computing weighted sums of pdfs.

The following two routines can help you to investigate the location and cause of oscillating splines, in case you get complaints from `evolfg` or `extpdf`.

```
epsi = SPLCHK ( iset, id, iq )
```

Returns $\epsilon = \|\boldsymbol{u} - \boldsymbol{v}\|$ of a singlet/non-singlet basis pdf `id`, indexed according to (5.4), at a grid point `iq`. Here $\boldsymbol{u}$ and $\boldsymbol{v}$ are the vectors of quadratic and linear interpolation mid-between the grid points in x, as is described in Section 3.4. By definition, $\epsilon = 0$ for linear interpolation, and should be a small number (like 0.05, say) for quadratic interpolation. A large value indicates that the spline oscillates at `iq`, which can then be further investigated with the routine `fsplne` below.

```
pdf = FSPLNE ( iset, id, x, iq )
```

This routine is identical to `bvalxq`, except that the local polynomial interpolation in x is replaced by spline interpolation, as used in the QCDNUM evolution and convolution routines (note that `fsplne` does not interpolate in μ^2). This function is provided as a diagnostic tool to investigate quadratic spline oscillations, if any, which may not be visible in the local interpolation used by `bvalxq`. You do not need `fsplne` or `splchk` to *detect* spline oscillations, since that is done automatically by `evolfg` and `extpdf`.

5.10 Lists and Tables

Instead of calling an interpolation routine in a loop over x and μ^2, you can gain speed by passing to QCDNUM a list of these points so that the program can optimise the loop internally. Here are two fast routines to generate a list or a table of interpolated pdfs.

```
call FFLIST ( iset, c, isel, x, q, *f, n, ichk )
```

Generate a x-μ^2 list of interpolated pdfs.

`iset`	Pdf set identifier [1–24].
`c`	Input array, dimensioned to `c(-6:6)` in the calling routine, filled with the coefficients of a linear combination of (anti)quarks. The indexing is given in (5.3). Note that the value of `c(0)` is ignored.
`isel`	Selection flag [0–8,12+i], as is described for the routine `sumfxq` above.
`x`, `q`	List of interpolation points, dimensioned to at least `n` in the calling routine.
`f`	Output array of pdf values, dimensioned to at least `n` in the calling routine.
`n`	Number of items in `x`, `q` and `f`.
`ichk`	If non-zero, QCDNUM insists that all `x-q` points are within the grid boundaries or cuts. If zero, a `null` value is returned for pdfs outside the boundaries.

```
call FTABLE ( iset, c, isel, x, nx, q, nq, *table, ichk )
```

Generate a pdf table in x and μ^2. The parameters are as for `fflist`, except,

`x`, `nx`	x-grid, dimensioned to at least `x(nx)` in the calling routine, and filled with x-values in strictly ascending order (no equal values allowed).
`q`, `nq`	As above, but now for the μ^2-grid.
`table`	Output table of pdfs, dimensioned to `table(nx,nq)` in the calling routine. The first dimension must be set to `nx`, the second dimension can be larger than `nq`.

This routine is fast since it partially scales with the sum `nx` + `nq` instead of with `nx` $\times$ `nq`.

C++ As mentioned in Section 5.1 the table is stored in a one-dimensional array by the C++ wrapper. To access the table it is best to use the pointer function `kij` below.

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

int iset = 1, isel = 1, n = 30, m = 15;
double c[13], x[n], q[m], table[n*m];
QCDNUM::ftable(iset,c,isel,x,n,q,m,table,1);
double tij = table[ kij(i,j,n) ];
```

6 Program Steering with Datacards

It is possible to steer QCDNUM via a datacard input file. A keycard in such a file consists of a keyword followed by a parameter list. Such a card will, upon reading, cause a call to the corresponding QCDNUM routine. As an example we show here a datacard file that steers the example program given in Section 4.2 (the syntax will be described later).

```
' SETLUN   6                          '
' GXMAKE   3  100  1  1.D-4           '
' GQMAKE       60  2  2.    1.D4      '
' FILLWT   1                          '
' SETORD   3                          '
' SETALF   0.364  2.                  '
' SETCBT   0      3.  25.   1.D11     '
```

Note that each card is read as a character string and should be embedded in single quotes and that the parameter lists can be given in free format. The corresponding calls in the example program can now be replaced by a call to `qcards`:

```
..
external mycards    !to be explained later
..
call qcinit( 6, ' ' )
call qcards( mycards, 'example.dcards', 0 )
ityp = 1
q0   = 3.5
call evolfg( ityp, func, def, iqfrmq(q0), eps )
```

```
          ..
```

In the above we have used so-called predefined keycards that can execute QCDNUM routines prior to a pdf evolution; the call to `evolfg` (and beyond) is, in this example, made in the FORTRAN code itself. We will show in Section 6.3 how to define user keycards that can drive a call to `evolfg` or to any other routine.

C++ At present there is no C++ interface to the datacard routines described in this section. A solution is to package all datacard handling in a FORTRAN subroutine and interface that routine to C++.

6.1 Datacard File

The input datacard file is a normal text file. Datacards in this file are strings of up to 120 characters long, embedded in single quotes (`'  ...  '`). Blank or unquoted records are allowed on the file as long as the first word of that record is not a valid keyword.

List-directed (*i.e.* free-format) read from a string is not permitted in FORTRAN77 [31],[31] so that the format of each word in a datacard is classified by QCDNUM as logical (`L`), integer (`I`), floating point (`F`), exponential (`E` or `D`) or character (`A`).[32] After classification the words in a datacard are transferred to memory by a formatted read.

Note that cards are only processed if the first word is a keyword recognised by QCDNUM. Thus it is easy to add comment (or blank) lines to the file, or to comment-out data-cards by inserting some non-blank characters in front of the keyword.

```
        ' This line is a comment but the next line is a keycard '
        ' SETORD 3 '
        ' And the keycard below is commented out '
        ' C-- SETORD 2 '
```

Inline comments can be put after the closing quote.

```
        ' SETLUN 6 '   This is an inline comment
```

If a parameter is a string with embedded blanks, you must put it in double quotes (`''`).

```
        ' MYCARD   Paul  Dirac  ''Paul Dirac'' '
```

This card has three parameters: first name (`A4`), last name (`A5`) and full name (`A10`).

In the examples above we have put the keywords in upper case for readability but note that QCDNUM is case-insensitive so that it does not matter if character strings are put in upper, lower, or mixed case. File names are an exception to this since they are passed directly to the operating system which usually is case-sensitive.

A datacard file is processed by a call to `qcards`.

[31] Most systems allow for a list-directed read from strings but this is not guaranteed to be portable.

[32] The string formatter recognises any number in `I`, `F`, `E` or `D` format. Anything else is classified as a character word (`A`), except that a single uppercase `T` or `F` is classified as logical `true` or `false`. We refer to the MBUTIL write-up for a full description of the string formatter.

```
call QCARDS ( mycards, 'filename', iprint )
```

Process all known keycards in a text file.

`mycards`	Subroutine, declared external in the calling routine, that is called when a user-defined card is encountered, see Section 6.3 for how it should be coded. Must be supplied as a dummy routine when there are no user cards.
`filename`	Input file name.
`iprint`	No listing (`0`), listing and processing (`1`) or listing without processing (`-1`). The latter is useful to get in a dummy run a clean datacard listing from which you can check the correct QCDNUM formatting of the parameters. If you set `iprint` to ±2 also the format descriptor will be printed.

A fatal error message is issued if the datacard file does not exist, cannot be read or if there is a problem with decoding a keycard. On exit, the input file will be closed.

6.2 Predefined Keycards

In Table 3 we list the QCDNUM predefined keycards. Optional parameters are given in

Table 3 – QCDNUM predefined keycards.

SETLUN	lun	<filename>				
SETVAL	chopt	dval				
SETINT	chopt	ival				
GXMAKE	iosp	nx	nxlim	xmi(1)	...	xmi(nxlim)
GQMAKE		nq	nqlim	qsq(1)	...	qsq(nqlim)
FILLWT	itype	<filename>				
SETORD	iord					
SETALF	alfs	r2				
SETCBT	nfix	q2c	q2b	q2t		
MIXFNS	nfix	r2c	r2b	r2t		
SETABR	ar	br				
SETCUT	xmi	q2mi	q2ma			
QCSTOP						

brackets. Most keycards have exactly the same parameter lists as the subroutine calls given in Table 2 but some of them have a slightly different syntax as is given below.

`SETLUN lun <filename> :` When `lun` is set to `6` you can omit the `filename` parameter; in all other cases a `filename` must be given.

`GXMAKE iosp nx nxlim xmi(1) ... xmi(nxlim) :` Here the order of the parameters differs from that in the FORTRAN call: we first put the spline order, followed by the requested number of grid points, the number of sub-grids, and then the subgrid limits.

```
        ' GXMAKE  3  100  1  1.D-4         '    single 100 pt x-grid
        ' GXMAKE  3  100  3  1.D-4 0.1 0.7 '    3-fold 100 pt x-grid
        ' GXMAKE  3  100  2  1.D-4 0.1 0.7 '    2-fold without xmi = 0.7
```

The point density increases by a factor of two for each subgrid generated.

`GQMAKE nq nqlim qsq(1) ... qsq(nqlim) :` Also here the order of the parameters differs from that in the FORTRAN call. You can inject up to 20 points into the grid (including the end points). The point densities are set equal for all regions in μ^2.

```
        ' GQMAKE 60 2  2.        1.D4 '   60 points in range 2-10^4 GeV2
        ' GQMAKE 60 4  2. 3. 25. 1.D4 '   idem with q2c and q2b inserted
```

`FILLWT itype <filename> :` If a `filename` is given, an up-to-date disk file of weights is maintained, if not, the weights are calculated from scratch.

```
        ' FILLWT  1   ../weights/unpolarised.wt '
        ' FILLWT  2   ../weights/polarised.wt   '
```

`SETCBT nfix q2c q2b q2t :` Here you have to specify the threshold values, and not the grid indices as in the FORTRAN call to `setcbt`. All three thresholds must be given.

`QCSTOP :` An input file may contain other data in addition to the keycards (how to read these data is up to you). In this case you can put a `QCSTOP` card to terminate the search for keycards, which would otherwise continue until an end-of-file is reached.

6.3 User-defined Keycards

Suppose that we want, via a datacard, to run from a scale of $\mu_0^2 = 3.5$ GeV^2 an evolution of un-polarised pdfs (`itype` = 1).

```
        '  EVOLFG  3.5  1 '
```

Since this card is non-standard we have to do two things to make it active: (i) add `EVOLFG` to the list of known keys by a call to `qcbook` and (ii) provide the FORTRAN code to be executed when the key is encountered in a datacard file.

```
call QCBOOK ( 'action', 'key' )
```

Add or delete keys from the list of known keys.[33]

`'action'`	First character must be set to `'A'` (add) or `'D'` (delete). You can also set it to `'L'` to get a list of known keys on the QCDNUM output stream.
`'key'`	Name of the key. A new key should not be longer than 7 characters and not be present in the list of known keys.

By default, `mky0` = `50` key-names can be held in memory of which 13 are already taken by the predefined keys. The value of `mky0` can be changed in `qcdnum.inc`, if necessary.

Thus, in our example program, we have first to make a call to `qcbook`.

```
      call qcbook( 'Add', 'EVOLFG' )
```

The routine `qcards` will now recognise `EVOLFG` as a user-defined keyword and will call the subroutine `mycards` to process the card. This subroutine should be coded as follows.

```
      subroutine mycards ( key, nk, pars, np, fmt, nf, ierr )
      implicit double precision (a-h,o-z)
      character*(*) key, pars, fmt
      ..
```

The first 6 arguments are presented to you by QCDNUM, the `ierr` flag you set yourself.

`key`	Character string containing the name of the key in upper case.
`nk`	Number of characters in `key`. Always `1` ≤ `nk` ≤ `7`.
`pars`	Character string containing the parameter list.
`np`	Number of characters in `pars` (`0` = no parameters for this key).
`fmt`	Character string with the FORTRAN format descriptor of `pars`.
`nf`	Number of characters in `fmt` (`0` = no format string).
`ierr`	Should be set, on exit, to `0` if all OK, to `1` if the parameters cannot be read, to `2` if the card cannot be processed, and to `3` if the `key` is unknown.

The `pars` character string acts as an internal file from which the parameters can be read in the format given by `fmt`. This is illustrated in the following code which processes the `EVOLFG` keycard (not all error handling shown).

```
      subroutine mycards ( key, nk, pars, np, fmt, nf, ierr )
      implicit double precision (a-h,o-z)
      character*(*) key, pars, fmt
      external func
      common /pass/ def(-6:6,12)

      if(key .eq. 'EVOLFG') then
        read(unit=pars,fmt=fmt,err=100,end=100) q0, itype
```

[33]Note that you can also delete predefined keys. This is useful when you want to inhibit a keycard-driven call or want to replace a key by deleting it and then add it again (it comes back as a user-defined key). In this way you can taylor the predefined keys to your needs, if necessary.

```
          call evolfg(itype,func,def,iqfrmq(q0),eps)
        else
          ierr = 3
        endif
        return

  100   ierr = 1
        return
        end
```

It is easy to handle optional parameters by checking on read errors. In the modified if-block below we also process `EVOLFG` with only `q0` as input, and `itype` = `1` as default.

```
        if(key .eq. 'EVOLFG') then
          read(unit=pars,fmt=fmt,err=10,end=10)   q0, itype
          call evolfg(itype,func,def,iqfrmq(q0),eps)
          return
   10     read(unit=pars,fmt=fmt,err=100,end=100) q0
          call evolfg(  1  ,func,def,iqfrmq(q0),eps)
        else
          ierr = 3
        endif
```

Note that user keys are strictly local in the sense that they can only be defined in the program that calls `qcards` to read the datacard file. It thus does not make sense to define keys in an add-on package: packages should provide subroutines and not keys. Of course you can easily define your own key that calls a package (or any other) routine.

7 Tools

In this section we describe a large set of tools that can be used to evolve a set of pdfs, or to compute convolution integrals needed for the calculation of structure functions or parton luminosities. With these tools you can write QCDNUM add-on packages that will extend the functionality of the program. For instance, the ZMSTF and HQSTF structure function packages (see Appendices D and E) are entirely based on the QCDNUM toolbox.

C++ At present there are no C++ wrappers available for the toolbox routines.

All toolbox routines operate on a workspace which is nothing else than a sufficiently large linear array that you declare in your application program. This workspace is partitioned into sets of tables after which you can call toolbox routines that act on these tables in various ways. It is important to note that the toolbox inherits from the QCDNUM main program the x-μ^2 grid and also the evolution parameters like the perturbative order, flavour thresholds *etc.*, as set by the routines described in Section 5.6. Note also that, like in the main program, pdfs can only be accessed when they are evolved with the current set of parameters, see Section 5.9.

In the next section we show how to partition the toolbox workspace, followed by a description of the table indexing (Section 7.2), weight filling routines (7.3), combined weights (7.4), coupled DGLAP evolution (7.5), pdf access (7.6), pdf tranformations (7.7), convolution tools (7.8) and the fast convolution engine (7.9). In Appendix F you can find a tutorial with many examples of how to use the toolbox.

All toolbox routines are listed in Table 4.

Table 4 – Routines in the QCDNUM toolbox.

Subroutine or function	Description
	Workspace
`MAKETAB ( w, nw, itypes, np, new, *iset, *nwu )`	Create tables
`SETPARW ( w, iset, upars, n )`	Store user information
`GETPARW ( w, iset, *upars, n )`	Read user information
`DUMPTAB ( w, iset, lun, 'filename', 'key' )`	Dump to disk
`READTAB ( w, nw, ..., *iset, *nwu, *ierr )`	Read from disk
	Identifiers
`IDSPFUN ( 'pij', iord, iset )`	Internal weight table
`IPDFTAB ( iset, id )`	Internal pdf table
	Weights
`MAKEWTA ( w, id, afun, achi )`	Regular piece $A(x)$
`MAKEWTB ( w, id, bfun, achi, nodelta )`	Singular piece $[B(x)]_+$
`MAKEWRS ( w, id, rfun, sfun, achi, nodelta )`	Product $R(x)[S(x)]_+$
`MAKEWTD ( w, id, dfun, achi )`	δ-Function $D(x)\delta(1-x)$
`MAKEWTX ( w, id )`	Weights for $x[f_a \otimes f_b]$
	Combined weights
`SCALEWT ( w, c, id )`	Scale weight table
`COPYWGT ( w, id1, id2, iadd )`	Copy weight table
`WCROSSW ( w, ida, idb, idc, iadd )`	Convolution of weights
`WTIMESF ( w, fun, id1, id2, iadd )`	Multiply by $f(\mu^2, n_f)$
	Evolution
`EVFILLA ( w, id, func )`	Fill table of coefficients α
`EVGETAA ( w, id, iq, *nf, *ithresh )`	Get coefficients
`EVDGLAP ( w, iw, ia, if, ..., *nf, *e )`	Coupled evolution
	Access to pdfs
`EVPDFIJ ( w, id, ix, iq, ichk )`	Pdf at a grid point
`EVPLIST ( w, id, x, qmu2, *pdf, n, ichk )`	List of interpolated pdfs
`EVTABLE ( w, id, x, nx, q, nq, *table, ichk )`	Table of interpolated pdfs
`EVPCOPY ( w, id, def, n, iset )`	Copy to internal memory
	Evolution parameters
`CPYPARW ( w, *array, n, iset )`	Copy parameter list
`KEYPARW ( w, iset )`	Get parameter key
`USEPARW ( w, iset )`	Activate parameters
	Transformations
`EFROMQQ ( qvec, *evec, nf )`	Transform from $q, \bar{q}$ to $e^{\pm}$
`QQFROME ( evec, *qvec, nf )`	Transform from $e^{\pm}$ to $q, \bar{q}$
	Convolution
`FCROSSK ( w, idw, idum, idf, ix, iq )`	Convolution $x[f \otimes K]$

continued on next page

continued from previous page

`FCROSSF ( w, idw, idum, ida, idb, ix, iq )`	Convolution $x[f_a \otimes f_b]$
`STFUNXQ ( stfun, x, qmu2, stf, n, ichk )`	Interpolation
Fast convolution	
`FASTINI ( x, qmu2, n, ichk )`	Pass a list of x and μ^2
`FASTCLR ( ibuf )`	Clear buffer
`FASTINP ( w, idf, coef, ibuf, iadd )`	Store a pdf in a buffer
`FASTEPM ( idum, idf, ibuf )`	Store $\|g, e^{\pm}\rangle$ in a buffer
`FASTSNS ( iset, def, isel, ibuf )`	Store singlet or non-singlet
`FASTSUM ( iset, coef, ibuf )`	Store weighted sum of $\|e^{\pm}\rangle$
`FASTFXK ( w, idw, ibuf1, ibuf2 )`	Convolution $x[f \otimes K](x)$
`FASTFXF ( w, idw, ibuf1, ibuf2, ibuf3 )`	Convolution $x[f_a \otimes f_b](x)$
`FASTKIN ( ibuf, fun )`	Scale by a kinematic factor
`FASTCPY ( ibuf1, ibuf2, iadd )`	Copy or accumulate result
`FASTFXQ ( ibuf, *f, n )`	Interpolate final result

Output arguments are prefixed with an asterisk (`*`).

7.1 Toolbox Workspace

The first step in using the toolbox is to declare a large double precision array `w(nw)` in your application program.[34] This toolbox workspace must be partitioned into one or more sets of tables that, depending on your application, will contain evolution or convolution weights, collections of pdfs, and tables of perturbative expansion coefficients.

All tables in the workspace depend on the x-μ^2 grid as defined by upstream calls to `gxmake` and `gqmake`; a downstream re-definition of the grid invalidates all your toolbox workspaces and makes them inaccessible (see Section 7.10 for more on toolbox errors).

Internally these tables have up to 6 dimensions but you are exposed to only a few of these, depending on what the table is used for. This leads to six different type of table:

`itype` = `1`	Weight tables that depend only on x. Identifiers run from `101`–`199`;
`itype` = `2`	Weight tables that depend on x and n_f. Identifiers run from `201`–`299`;
`itype` = `3`	Weight tables that depend on x and μ^2. Identifiers run from `301`–`399`;
`itype` = `4`	Weight tables that depend on x, μ^2 and n_f. Identifiers run from `401`–`499`;
`itype` = `5`	Pdf tables that depend on x and μ^2. Identifiers run from `501`–`599`;
`itype` = `6`	Tables that depend on μ^2 only. Identifiers run from `601`–`699`.

The n_f dependence of weight tables should not be confused with that of the pdfs. For pdfs it means that they are evolved with a certain number of active flavours that may, in the VFNS, depend on μ^2. For the weight tables it means that n_f is a parameter of the convolution kernel so that there are four look-up tables, one for each $n_\mathrm{f} = 3, 4, 5, 6$.

Below we describe a routine (`maketab`) that creates a set of tables. Additional calls to `maketab` will create additional sets, numbered in sequence, up to a maximum of `30`

[34]How large should `w` be? Just put `nw` = `1` and get the answer from the `maketab` error message.

sets. Organising tables into sets can much simplify your bookkeeping because the same table identifiers can be used in different sets, as is done internally in QCDNUM for the un-polarised (1), polarised (2) and time-like sets (3) where the gluon always has id = 0, the singlet id = 1, *etc.*

In the following we describe the routines that partition the toolbox workspace, store extra information into a set of tables, dump a set of tables to disk, and read them back.

```
call MAKETAB ( w, nw, itypes, np, new, *iset, *nwused )
```

Add a set of tables to a workspace `w`.

`w`	Double precision array declared in the calling routine.
`nw`	Dimension of `w` as declared in the calling routine.
`itypes`	Integer array dimensioned to `itypes(6)` in the calling routine which contains in `itypes(i)` the number of tables ($\leq$ 99) of type `i` to be generated. When `itypes(i)` = 0 then no tables of type `i` will be generated.
`np`	Number of words reserved to store user information (see `setparw` below).
`new`	If `new` = 0 the set is added to those already present in the workspace and the `iset` identifier is incremented up to a maximum of `iset` $\leq$ `mst0` = 30 sets.[35] If `new` = 1, then the existing sets, if any, are overwritten by the new set.
`iset`	Gives, on exit, the identifier that QCDNUM has assigned to the set of tables.
`nwused`	Gives, on exit, the total number of words used in the workspace. If `nwused` is negative, then the workspace is not sufficiently large (fatal error) and should be re-dimensioned in the calling routine to at least `-nwused`.

Note that `maketab` initialises all tables in the set to zero.[36]

By default, the weight tables (type 1–4) can be used in both the evolution (`evdglap`) and convolution routines (`fcrossk`, `fcrossf`, `fastfxk`, `fastfxf`). However, if your convolution kernels are not splitting functions, you can flag the tables as such by putting a negative number in `itypes(i)`. Such type-`i` tables can then only be used in a convolution routine and not in `evdglap` (fatal error if you try) but have the advantage that they are much smaller. This can lead to considerable savings in space and filling time, in particular when the tables are type-3 or 4.

```
call SETPARW ( w, iset, upars, n )
```

Store extra information such as quark masses or other parameters that you want to dump to disk, along with the tables.[37]

`w`	Workspace, partitioned by a previous call to `booktab`.

[35] For more than 30 sets please update the parameter `mst0` in `qcdnum.inc`, and recompile QCDNUM.

[36] You can use also the routine `scalewt` (Section 7.4) to explicitly zero a table, if needed.

[37] When you evolve a pdf set with `evdglap` (Section 7.5) the evolution parameters are automatically stored so that you do not have to do this explicitly by a call to `setparw`. See also Section 7.6.

`iset`	Table set identifier.
`upars`	Array, dimensioned to at least `upars(n)` in the calling routine. On entry the array must be filled with the extra information you want to store.
`n`	Number of items to store, `n` $\leq$ `np`, where `np` is set in the call to `maketab`.

The user data can be retrieved by a call to `getparw(w,iset,upars,n)`.

```
call DUMPTAB ( w, iset, lun, 'filename', 'key' )
```

Dump the table set `iset` to disk. Fatal error if `iset` does not exist. You can use the function `nxtlun` of Section 5.3 to find a free logical unit number. Apart from the tables, information is written about the QCDNUM version, the x-μ^2 grid definition and the current spline interpolation order. The `key` text string can be used to stamp the file with some identifier like a package name and version number. The dump is un-formatted so that the file cannot be exchanged across machines. Note that a disk file can contain only one set of tables so that different sets must be dumped on different files.

```
call READTAB ( w, nw, lun, 'fn', 'key', new, *iset, *nwu, *ierr )
```

Read a set of tables from disk into the workspace `w(nw)`. Like in `maketab`, the input flag `new` controls the overwriting of existing tables. The parameter `iset` is, on exit, set to the identifier that QCDNUM has assigned to the set of tables read in. The total number of words used in the workspace is returned in `nwu`. You will get a fatal error message if `w(nw)` is not large enough to contain the tables or if the maximum number of table sets `mst0` = `30` is exceeded. On exit, the error flag is set as follows (non-zero means that nothing has been read in so that it is up to you to take the appropriate action).

`0`	Set of tables successfully read in.
`1`	Read error or input file does not exist.
`2`	File written by another QCDNUM version.
`3`	Key mismatch.
`4`	Incompatible x-μ^2 grid definition.

QCDNUM insists that the key written on the file matches the key entered as an argument to `readtab`.[38] Thus if, for instance, the key is set to a package name and version number then the user of the package cannot read obsolete files written by earlier versions, or read files written by another package. If you don't want to use keys, just enter an empty string as a key in the calls to `dumptab` and `readtab`.

Note that the table set identifier `iset` is dynamically allocated so that it usually is not preserved in a write-read sequence: if you dumped `iset` = `2` to disk, it may very well be read back as `iset` = `5`. Please be aware of this when addressing tables in your toolbox workspace (see the next section).

[38]Note that the key matching is case insensitive and that leading and trailing blanks are ignored.

7.2 Table Identifiers

A table in the workspace `w` is characterised by three numbers:

`iset`	The table set number assigned by QCDNUM in the call to `maketab` or `readtab`;
`itype`	The index `1`–`6` that differentiates between the various type of table;
`n`	The table number in the range `1`–`99`.

These numbers serve to build a so-called *global* identifier that uniquely identifies a table.

$$\texttt{id\_global} = \texttt{1000*iset} + \texttt{100*itype} + \texttt{n}. \tag{7.1}$$

Thus `id` = `3206` refers to the 6th type-2 table of set 3 in the toolbox workspace.

If you store tables on disk please note that the `iset` identifier is dynamically allocated so that only the last three digits of a global identifier are guaranteed to be preserved between disk dump and a disk read. You have thus to construct, after a read, the global identifier with (7.1) using the value of `iset` returned by `readtab`, and not use the table identifier as it was before the dump.

Most toolbox routines allow you to use tables in the internal memory, in addition to those in the toolbox workspace. To address the internal tables, QCDNUM provides two functions that return the identifiers of weight tables (`idspfun`) and pdf tables (`ipdftab`).[39]

```
id = IDSPFUN ( 'pij', iord, iset )
```

Returns the global identifier of a weight table in internal memory (negative number).

`'pij'` Name of the splitting function. Valid input strings are

`PQQ, PQG, PGQ, PGG, PPL, PMI, PVA, AGQ, AGG, AQQ, AHQ, AHG`

`iord`	Select LO (`1`), NLO (`2`) or NNLO (`3`).
`iset`	Select un-polarised (`1`), polarised (`2`) or fragmentation function (`3`).

```
id = IPDFTAB ( iset, id )
```

Returns the global identifier of a pdf table in internal memory (negative number).

`iset`	Pdf set identifier as is defined in Section 5.2 [`1`–`24`].
`id`	Pdf identifier of an $|e^{\pm}\rangle$ basis function, indexed according to (5.4) [`0`–`98`].

Both functions return `-(1000*iset+100*itype+n)` with `iset`, `itype` and `n` the internal table identifiers (which are hidden for you). The minus sign tells QCDNUM to address the internal memory instead of the toolbox workspace. Upon error, the functions return an error code that is understood by downstream toolbox routines which will then generate the appropriate error message.

[39]The internal α_s tables can be accessed by the function `altabn` described in Section 5.7.

7.3 Weight Tables

To calculate convolution integrals, the convolution kernels must first be turned into weight tables. Note that the kernels presented to QCDNUM must be defined by convolution with a parton *number* density f, and not with a momentum density xf.[40]

The convolution kernels may contain singularities ('plus' prescriptions), as is described in Appendix B. To deal with such singularities, we formally decompose a kernel into a regular part (A), a singular part (B), a product (RS) and a delta function

$$C(x) = A(x) + [B(x)]_+ + R(x)[S(x)]_+ + D(x)\delta(1-x). \tag{7.2}$$

For each term in (7.2) there exists a separate filling routine that will *add* its contribution to the weight table of C.

The generalised mass (GM) form of a convolution integral is given by (3.23) in Section 3.3:

$$\mathcal{F}(x, Q^2) = x \int_\chi^1 \frac{\mathrm{d}z}{z} f(z, \mu^2)\, C\left(\frac{\chi}{z}, \mu^2, Q^2, m_h^2\right), \tag{7.3}$$

where $\chi = ax$ is a so-called rescaling variable and the factor $a \geq 1$ is a function of Q^2. In the code examples below we will assume, for definiteness, that the relation between μ^2 and Q^2 is given by

$$Q^2 = \alpha\mu^2 + \beta \tag{7.4}$$

and that the rescaling variable is defined by

$$\chi = ax = \left(1 + \frac{4m_h^2}{Q^2}\right) x. \tag{7.5}$$

To the table generating routines must be supplied a function that defines the rescaling variable (`achi`), together with one or more functions that provide the interface to the convolution kernel (`cfun`). These functions must be coded as follows.

```
      double precision function achi(qmu2)
      implicit double precision (a-h,o-z)
      common /pass/ alfa, beta, hmass, ....
      Q2   = alfa*qmu2 + beta
      achi = 1.D0 + 4.D0*hmass*hmass/Q2
      return
      end
```

```
      double precision function cfun(chiz,qmu2,nf)
      implicit double precision (a-h,o-z)
      common /pass/ alfa, beta, hmass, ....
      Q2   = alfa*qmu2 + beta
      cfun = some_function_of(chiz,qmu2,Q2,nf,hmass,...)  !chiz = chi/z
      return
      end
```

[40]The weight table conversion from number to momentum density is done internally in QCDNUM.

Although one should take out as many μ^2-dependent factors (*e.g.* powers of α_s) as possible from the convolution kernel, it is clear from the above that quark mass parameters and the relation between μ^2 and Q^2 may enter via the rescaling variable χ and that such a dependence can never be factored out of the convolution integral. Therefore the weight tables of the GM schemes will, in general, depend on x and μ^2 and must be stored in type-3 or 4 tables. The code below, for example, fills a type-4 table with the regular part of a convolution kernel.

```
      external cfun, achi
      call MakeWtA(w,6402,cfun,achi)  !fill table 402 of set 6
```

When you work in a mass-less scheme where $\chi = x$ and $a = 1$ then (7.3) reduces to the familiar Mellin form $\mathcal{F} = x[f \otimes C]$. In this case you may code for the `achi` function

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

Here your weight table will most likely depend on x only, or on x and n_f, and can thus be stored in type-1 or 2 tables.

QCDNUM calculates by Gauss quadrature (CERNLIB routine D103) the integrals that define the weights. In case the default accuracy of $\epsilon = 10^{-7}$ cannot be reached (fatal error message), this limit can be raised by a call to `setval('epsg',value)`. Note, however, that problems with the Gauss integration will most likely be caused by problems with the integrand—such as near-singular behaviour somewhere in the integration domain—and that this cannot be cured by relaxing the required accuracy.

We emphasise once more that convolution integrals found in the literature must, if necessary, be brought into the general form (7.3) by modifying the published convolution kernel. An example of such a modification can be found in Appendix E.1.

```
call MAKEWTA ( w, id, afun, achi )
```

Calculate the weights for the regular contribution $A(x)$ to a convolution kernel and add these to table `id` in the workspace `w`.

`w`	workspace declared in the calling routine and previously partitioned by `maketab`.
`id`	Weight table identifier, given in the global format (7.1).
`afun`	Name of a function (see above) that returns the regular piece of the convolution kernel. Should be declared `external` in the calling routine.
`achi`	Name of a function (see above), declared `external` in the calling routine, that returns the value a of the rescaling variable $\chi = ax$. QCDNUM insists that always `achi` $\geq$ `1`; you will get a fatal error if not.

```
call MAKEWTB ( w, id, bfun, achi, nodelta )
```

Calculate the weights for the singular contribution $[B(x)]_+$ to a convolution kernel and add these to a table in the workspace `w`. The arguments and the coding of `bfun` and `achi` are as for `makewta`. Thus, if a kernel has both a regular and a singular part, then do

```
        call MakeWtA(w,1201,afun,achi)   !put regular  part in id = 1201
        call MakeWtB(w,1201,bfun,achi,0) !add singular part to id = 1201
```

It is seen from Appendix B, equation (B.3), that a '+' prescription generates a $\delta(1-x)$ contribution. By default, `makewtb` includes this contribution, unless you set `nodelta` = `1`. In that case the $\delta(1-x)$ contribution is not calculated and must be entered, perhaps combined with other such contributions, via a call to `makewtd`, see below.

```
call MAKEWRS ( w, id, rfun, sfun, achi, nodelta )
```

Calculate the weights for the product contribution $R(x)[S(x)]_+$ to a convolution kernel and add these to a table in the workspace `w`. The arguments and the coding of `rfun`, `sfun` and `achi` are as for `makewta`.

```
call MAKEWTD ( w, id, dfun, achi )
```

Calculate the weights for the $\delta(1-x)$ contribution to a convolution kernel and add these to a table in the workspace `w`. The delta function is multiplied by the function `dfun`. The arguments and the coding of `dfun` and `achi` is as for `makewta`.

```
call MAKEWTX ( w, id )
```

Calculate the weights (3.20) for the convolution $x[f_a \otimes f_b](x)$.

`w`	workspace declared in the calling routine and previously partitioned by `maketab`.
`id`	Table identifier. Because the weight table depends only on x, it can be stored in a type-1 table, but equally well in types-2, 3 or 4, if desired.

7.4 Combined Weights

Sometimes it is necessary to combine weight tables into another weight table. An example of this is the coefficient function

$$C^{(2,1)}_{2,+} = C^{(0)}_{2,\mathrm{q}} \otimes P^{(1)}_{+} + C^{(1)}_{2,+} \otimes P^{(0)}_{\mathrm{qq}} - \beta_0\, C^{(1)}_{2,+} \tag{7.6}$$

taken from expression (D.7) in Appendix D. Here we see convolutions of coefficient and splitting functions, multiplication by a β-function, and, of course, the addition or subtraction of terms. Below we will describe a set of routines that allow you to calculate expressions like (7.6) and store the result in a combined weight table.

One feature of these routines is that you can combine tables of different type, provided that this does not lead to a loss of information. Thus you can copy a type-1 table to a type-3 table but not the other way around. All routines check that you use a correct combination of types, and issue a fatal error condition if this is not the case.

```
call SCALEWT ( w, c, id )
```

Multiply the contents of a weight table `id` by a constant `c`. Obviously, `id` cannot refer to a table in internal memory. You can use this routine to explicitly set a table to zero.

```
call COPYWGT ( w, id1, id2, iadd )
```

Copy the contents of table `id1` to `id2`. You can copy weight tables (type-1–4), but also pdf tables (type-5) or tables with expansion coefficients (type-6).

`w`	Workspace declared in the calling routine.
`id1`	Input weight table identifier, given in the global format (7.1).
`id2`	Output table identifier with `id2` $\neq$ `id1`. The output table type may be different from the input table type, as is described above.
`iadd`	If set to `0` copy `id1` to `id2`, if set to `+1` (`-1`) add (subtract) `id1` to (from) `id2`.

You can use this routine to import a weight table from internal memory into the toolbox workspace by setting `id1` to an identifier that is generated by `idspfun`. Internal splitting function tables are type-1 or type-2 (you can see this from the second but last digit of the `idspfun` identifier) so that you can avoid errors simply by always importing to type-2. Importing pdf tables from internal memory does not make sense and is not allowed.

```
call WCROSSW ( w, ida, idb, idc, iadd )
```

This routine generates a weight table for the convolution of two kernels K_a and K_b. The weight table is calculated with (3.18) from two input tables $\boldsymbol{W}_a$ and $\boldsymbol{W}_b$.

`w`	Workspace declared in the calling routine.
`ida`	Table identifier containing the weights of kernel K_a.
`idb`	As above for the weights of kernel K_b.
`idc`	Output table identifier. Cannot be set equal to `ida` or `idb`.
`iadd`	If set to `0` store the result of the convolution in `idc`, if set to `+1` (`-1`) add (subtract) the result to (from) the contents of `idc`.

Both `ida` and `idb` can refer to splitting functions in internal memory, with identifiers constructed with `idspfun`. The table types of `ida` and `idb` may be different, but the type of `idc` must be such that it can contain either input table. Thus if `ida` is type-2 (x, n_f) and `idb` is type-3 (x, μ^2), then `idc` must be type-4 (x, μ^2, n_f).

```
call WTIMESF ( w, fun, id1, id2, iadd )
```

Multiply a weight table by a function of μ^2 and n_f and store the result in another table.

`w`	Workspace declared in the calling routine.

`fun`	Double precision function `fun(iq,nf)` declared `external` in the calling routine.
`id1`	Input identifier of a weight table in the workspace `w`, or in internal memory.
`id2`	Identifier of the output table. It is allowed to have `id1` = `id2` (in-place modification of a table), unless `id1` is an internal splitting function table. The table type of `id2` must be such that no information is lost, fatal error otherwise.
`iadd`	Store the result in `id2` in case `iadd` = `0` or add (subtract) the result to (from) `id2` in case `iadd` = `+1` (`-1`).

The routine loops over `iq` and `nf` and calls `fun(iq,nf)` with the following argument ranges, depending on the *output* table type:

type	variables	`iq` range	`nf` range
1	x	`1`–`1`	`3`–`3`
2	x, n_f	`1`–`1`	`3`–`6`
3	x, μ^2	`1`–`nq`	`3`–`3`
4	x, μ^2, n_f	`1`–`nq`	`3`–`6`

QCDNUM checks that the output table type matches the μ^2 and n_f dependence of both `id1` and `fun`, and will produce a fatal error if this is not the case.

As an example, we give below the code to construct a table corresponding to the combination of convolution kernels (7.6).

```
      external beta0   !beta function
        ..
      call WcrossW ( w, idC2Q0, idSpfun('PPL',2,1), idC2P21,  0 )
      call WcrossW ( w, idC2P1, idSpfun('PQQ',1,1), idC2P21, +1 )
      call WtimesF ( w, beta0 , idC2P1            , idC2P21, -1 )
```

7.5 Coupled DGLAP Evolution

In this section we describe routines to solve n coupled evolution equations

$$\frac{\partial f_i(x,\mu^2)}{\partial \ln \mu^2} = \sum_{j=1}^{n} [P_{ij} \otimes f_j](x,\mu^2).$$

The routines operate on a toolbox workspace `w` that should contain the $n \times n$ weight tables, tables of expansion coefficients, and also the n pdf tables.

After partitioning the workspace (Section 7.1) and computing the weight tables (Section 7.3), look-up tables of perturbative expansion coefficients must be filled with the routine `evfilla` below. Such a look-up table (type-6) should contain one of the coefficients a_i versus μ^2 of the perturbative expansion

$$P_{ij} = a_0\, P_{ij}^{(0)} + a_1\, P_{ij}^{(1)} + a_2\, P_{ij}^{(2)} + \cdots$$

Examples of expansion coefficients are powers α^n, α_s^m or products $\alpha^n \alpha_\mathrm{s}^m$.

```
call EVFILLA ( w, id, func )
```

Fill a type-6 look-up table with a perturbative expansion coefficient.

`w`	Toolbox workspace with at least one type-6 table.
`id`	Type-6 table identifier, in the global format (7.1).
`func`	User supplied function (see below), declared `external` in the calling routine.

The function `func` is called by `evfilla` in a loop over the μ^2 grid points `iq`, and should be coded as follows.

```
      double precision function func(iq,nf,ithresh)
      implicit double precision (a-h,o-z)
      func = value_of_expansion_coefficient_at_iq
      return
      end
```

Passed to `func` are `iq`, the number of flavours `nf`, and a threshold indicator `ithresh` that is set to `0` if `iq` is not at a threshold, and to `+1` (`-1`) if `iq` is at a threshold with the larger (smaller) number of flavours. Thus, at the charm threshold `iqc` the function is called twice, once with `nf` = `3` and `ithresh` = `-1`, and once with `nf` = `4` and `ithresh` = `+1`. To fill the tables with powers of $a_\mathrm{s}(\mu^2)$, properly truncated, you can use the routine `altabn` that is described in Section 5.7.

QCDNUM cannot keep track of your coupling constant so that you must yourself update the tables when it changes (in a fit), when the flavour thresholds change, or when the relation between the renormalisation and the factorisation scale changes.

```
alfa = EVGETAA ( w, id, iq, *nf, *ithresh )
```

Returns the value of the expansion coefficient at the μ^2 grid point `iq`, as is stored in the look-up table `id`. Also returned are the value `nf` of the number of flavours at `iq`, and the threshold indicator `ithresh` as described above. By default QCDNUM always takes the larger number of flavors $(4,5,6)$ at $(\mu_\mathrm{c}^2, \mu_\mathrm{b}^2, \mu_\mathrm{t}^2)$ but you can force the routine to take the smaller number of flavours by prefixing `iq` with a minus sign, thus:

```
    alfa = EVGETAA(w,id, iqc,iord,nf,ithresh)  !nf = 4, ithresh =  1
    alfa = EVGETAA(w,id,-iqc,iord,nf,ithresh)  !nf = 3, ithresh = -1
```

Upon error (non-existing table, `iq` out of range, *etc.*) a `null` value is returned for `alfa`.

After filling the weight tables (Section 7.3) and the tables of expansion coefficients, a set of pdfs can be simultaneously evolved with the routine `evdglap`. This routine can only evolve with a fixed number of flavours so that in the VFNS you have to implement yourself the loop over the flavour thresholds, as will be explained later.

```
call EVDGLAP ( w, idw, ida, idf, !start, m, n, !iqlim, ,*nf, *epsi )
```

Coupled fixed-flavour n-fold DGLAP evolution of the pdfs f_i, $i = 1, \ldots, n$.

`w`	Toolbox workspace, previously filled with weights and expansion coefficients. The workspace should also contain the pdf tables to be evolved.
`idw`	Integer array that contains in `idw(i,j,k)` the weight table identifier of P_{ij} at order k. In the calling routine it must be dimensioned to `idw(m,m,nk)` with `m` $\geq$ `n`. The third dimension `nk` should be at least as large as the maximum number of perturbative terms in the `evdglap` evolution (see below).
`ida`	As above, but now `ida(i,j,k)` contains the identifier of the look-up table of the expansion coefficient that multiplies P_{ij} at order k.
`idf`	Integer array, with in `idf(i)` the identifier of the pdf f_i to be evolved. The array must be dimensioned to at least `n` in the calling routine. It is required that all pdfs reside in the same table set (fatal error otherwise).[41]
`start`	Double precision array that in the calling routine should be dimensioned to `start(m,nx)` with the second dimension `nx` at least as large as the number of x-grid points. On entry, `start(i,j)` must be filled with the start value of $f_i(x_j)$. On exit, the array contains the pdfs at the end point of the evolution.
`m`	First two dimensions of `idw` and `ida` and the first dimension of `start`, as declared in the calling routine.
`n`	The number of pdfs to evolve simultaneously, `n` $\leq$ `m`.
`iqlim`	Integer array, declared `iqlim(2)` in the calling routine. On entry, `iqlim(1)` must be set to the start point of the evolution and `iqlim(2)` to the requested end point. If `iqlim(2)` $\geq$ `iqlim(1)` there is upward evolution, downward evolution otherwise. On exit, `iqlim(2)` is set to the actual endpoint of the evolution that might have been limited by QCDNUM to the next threshold.
`nf`	On exit, `abs(nf)` is set to the number of flavours used in the evolution. See below for the meaning of a negative `nf`.
`epsi`	Maximum difference between quadratic and linear interpolation in-between the grid points, as is described for the routine `evolfg`.

The order of the evolution (`iord`), the flavour thresholds (`iqc`,`b`,`t`) and the kinematic cuts are those set by upstream calls to `setord`, `setcbt` and `setcut`, respectively. All evolution parameters are stored along with the evolved pdfs.

By default, the number of perturbative terms (`nopt`) in the evolution is set equal to 1/2/3 at LO/NLO/NNLO. But this may not always be the case: for QCD-QED evolution, for instance, this may be 2/3/4, depending on the truncation of the $\alpha_s^n\alpha^m$ terms in the expansion. You can set the number of perturbative terms `nopt` by call to `setint` upstream of `evdglap`, for instance,

```
        call setint ( 'nopt', 234 )  !2/3/4 terms at LO/NLO/NNLO
```

[41]This is because the evolution parameters are stored as an attribute of a *set* and not of a table.

The number of digits of `nopt` defines the maximum order that `evdglap` can handle. Thus if you coded a QCD-QED evolution at LO, the setting of `nopt` = `2` means that `evdglap` can only be run at LO (error message otherwise), with two perturbative terms.

The weight tables must all reside in the toolbox workspace so that internal weight tables can only be used when they are first copied to the workspace with `copywgt`. Be aware, however, that P_{qg} in QCDNUM contains a factor $2n_{\mathrm{f}}$, see (B.2), which may not be what you want. It might very well be that splitting functions are missing, either because they don't exist like $P_{\mathrm{q\bar{q}}}$ at LO, or simply because they have never been calculated at higher orders. In this case you should set the table identifier to zero which causes `evdglap` to skip over the entry in the $n \times n$ splitting function matrix.

The `evdglap` routine cannot, like `evolfg`, transparently handle VFNS evolution, simply because it has no information about the flavour composition of the pdfs to be evolved. Instead, it automatically restricts the evolution to a fixed number of flavours by, if necessary, limiting the end point of the evolution to the next flavour threshold, cut, or grid boundary above (below) μ_0^2 in case of upward (downward) evolution.

In the VFNS you thus have to chain yourself the evolutions in the different threshold regions. Several features of the `evdglap` interface do facilitate such a chaining: after the evolution the actual endpoint is passed via `iqlim(2)` and the values of the pdfs at this endpoint via the `start` array. The number of flavours is passed via `nf`, but it is pre-pended by a minus sign when you hit the limits of the μ^2 grid, or cuts, indicating that no further evolution is possible.

These features make the VFNS chaining quite easy as is shown below by code for upward evolution (for downward evolution set `iqlim(2)` $\leq$ `0`, also inside the while loop).

```
      iqlim(1) = iq0
      iqlim(2) = 99999
      call evdglap(w,idw,ida,idf,start,m,n,iqlim,nf,epsi)
      do while(nf.gt.0)
        iqlim(1) = iqlim(2)
        iqlim(2) = 99999
        start    = start + discontinuity (code not shown)
        call evdglap(w,idw,ida,idf,start,m,n,iqlim,nf,epsi)
      enddo
```

This method of carrying the pdfs over the threshold via the start array causes a small bias when you evolve on multiple grids. This is because the pdfs returned by `evdglap` at the end of the evolution are a *composite* of the pdfs evolved on the individual sub-grids.

The bias is eliminated when you run `evdglap` in the so-called internal transfer mode where the pdfs are, subgrid-by-subgrid, carried over the threshold internally. You can select this mode by setting n negative. On entry, only the discontinuity should be passed via the start array which, on exit, is set to zero by `evdglap`.

The VFNS code now becomes (note that the first call to `evdglap` remains the same).

```
      iqlim(1) = iq0
      iqlim(2) = 99999
      call evdglap(w,idw,ida,idf,start,m,+n,iqlim,nf,epsi)
```

```
          do while(nf.gt.0)
            iqlim(1) = iqlim(2)
            iqlim(2) = 99999
            start    = discontinuity (code not shown)
            call evdglap(w,idw,ida,idf,start,m,-n,iqlim,nf,epsi)
          enddo
```

The bias is usually quite small (about a permille or less) but it is of course always better to remove it simply by running `evdglap` in the internal transfer mode.

7.6 Pdf interpolation

In this section we describe routines to access the pdfs evolved with `evdglap`. QCDNUM insists that these pdfs are evolved with the current parameters (fatal error otherwise). If this is not the case, you may have to first *activate* the parameters via

```
        call  USEPARW ( w, iset )
```

You can also copy the parameters to a local array or ask for the parameter key

```
        call  CPYPARW ( w, *array, n, iset )
        key = KEYPARW ( w, iset )
```

as is described in Section 5.6 for parameters stored in the internal memory.

```
pdfij = EVPDFIJ ( w, id, ix, iq, ichk )
```

Returns the value of a pdf at the grid point (`ix`,`iq`).

`w`	Workspace, with pdfs previously evolved with `evdglap`.
`id`	Pdf identifier in the global format (7.1). You can also access a (basis) pdf in internal memory, provided that `id` is constructed with the function `ipdftab`.
`ix`, `iq`	Grid point.
`ichk`	Yes (`1`) or no (`0`) check that (`ix`, `iq`) is within the grid boundaries or cuts.

You can set `iq` negative to return at the thresholds (`iqc,b,t`) the pdf for the lower number of flavours $n_f = (3, 4, 5)$ instead of for $n_f = (4, 5, 6)$. This allows you to view discontinuities at the thresholds, if there are any. For instance,

```
        disc = EvPdfij(w,id,ix,iqc,ichk) - EvPdfij(w,id,ix,-iqc,ichk)
```

To make the routine run faster in a loop, you can set `ichk = -1` to skip the check on the identifier `id`. However, you should always check the identifier on entry of the loop:

```
      pdf(1) =   EvPdfij( w, id, ix(1), iq(1),  1 )  !check id
      do i = 2,n
        pdf(i) = EvPdfij( w, id, ix(i), iq(i), -1 )  !do not check id
      enddo
```

```
call EVPLIST ( w, id, x, qmu2, *pdf, n, ichk )
```

Generate a list of interpolated pdfs.

`w`	Workspace, with pdfs previously evolved with `evdglap`.
`id`	Pdf identifier in the global format (7.1). You can also access a (basis) pdf in internal memory, provided that `id` is constructed with the function `ipdftab`.
`x`, `qmu2`	Arrays, dimensioned to at least `x(n)` and `qmu2(n)` in the calling routine.
`pdf`	Array, dimensioned to at least `pdf(n)` in the calling routine. On exit the array is filled with the list of interpolated pdfs.
`n`	Number of interpolation points.
`ichk`	As above (without the option to set `ichk` = `-1`).

```
call EVTABLE ( w, id, x, nx, q, nq, *table, ichk )
```

As above but now fill an `nx` $\times$ `nq` table of interpolated values. In the calling routine the arrays must be dimensioned `x(nx)`, `q(nq)` and `table(nx,nq)`. This routine is about a factor of two faster than `evplist`.

```
call EVPCOPY ( w, idf, def, n, iset )
```

With this routine you can export 13 gluon and (anti)quark tables plus n additional tables from the toolbox workspace to the QCDNUM internal memory.[42] Such an export is useful because you can then use any routine that works with internal pdfs such as the built-in interpolation routines or the structure function packages ZMSTF and HQSTF.

For this export it is required that you supply the flavour composition of the input quark densities so that QCDNUM can transform them to the singlet/non-singlet basis (5.1).

In the description below, n_{f} is the largest number of flavours encountered in the `evdglap` evolution. You do not have to actually enter this number since it is known to QCDNUM.

`w`	Workspace, with pdfs previously evolved with `evdglap`.
`idf`	Integer array, dimensioned to `idf(0:12+n)` in the calling routine. The gluon identifier must be stored in `idf(0)`, the $2n_{\text{f}}$ quark identifiers in `idf(i)`, `i` $= 1, \ldots, 2n_{\text{f}}$ and the extra n identifiers in `idf(12+i)`. The identifiers should be given in the global format (7.1) and all pdfs must be in the same table set.

[42]You can also use `pdfext` for this, but `evpcopy` is more convenient and also faster.

`def`	Double precision array, dimensioned `def(-6:6,12)` in the calling routine. In `def(i,j)` should be stored the contribution of flavour `i`, indexed as given in (5.2), to the quark density `j` $= 1, \ldots, 2n_\mathrm{f}$. The $2n_\mathrm{f} \times 2n_\mathrm{f}$ submatrix of (anti)quark coefficients must be invertible, see also `evolfg` in Section 5.7 and the tutorial in Appendix F.5.
`n`	The number of additional pdfs to be exported.
`iset`	Pdf set identifier [5–24] to which the pdfs should be copied. If `iset` does exist its contents are overwritten, provided that it can hold $13 + n$ pdfs (fatal error if not).[43] Otherwise the set is created, provided that there is enough space in the QCDNUM internal memory (fatal error if not).

The evolution parameters associated with the pdfs are also copied to internal memory.

7.7 Transformations

The routines in this section allow you to make pdf transformations from the singlet/nonsinglet basis (5.1) to the flavour basis, and *vice versa.*

We can write an arbitrary pdf in two ways as

$$|f\rangle = \sum_{i=1}^{n_\mathrm{f}} (\alpha_i |q_i\rangle + \beta_i |\bar{q}_i\rangle) = \sum_{i=1}^{n_\mathrm{f}} (d_i^+ |e_i^+\rangle + d_i^- |e_i^-\rangle), \tag{7.7}$$

where the first term on the right-hand side represents the pdf written on the flavour basis and the second term that on the singlet/non-singlet basis. To translate the coefficients α and β into d^+ and d^-, and *vice versa*, the routines `efromqq` and `qqfrome` are provided.

For convenience we show here again the indexing (5.3) of 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

, (7.8)

and the indexing (5.4) of the singlet/non-singlet basis

0	1	2	3	4	5	6	7	8	9	10	11	12
g	q_s	e_2^+	e_3^+	e_4^+	e_5^+	e_6^+	q_v	e_2^-	e_3^-	e_4^-	e_5^-	e_6^-

. (7.9)

```
call EFROMQQ ( qvec, *evec, nf )
```

Transform the coefficients of a linear combination of quarks and anti-quarks from the flavour basis to the singlet/non-singlet basis.

`qvec`	Input array, dimensioned `qvec(-6:6)`, filled with the coefficients α and β of a linear combination of quarks and antiquarks, indexed according to (7.8).
`evec`	Output array, dimensioned to `evec(12)`, filled with the singlet/non-singlet coefficients d^+ and d^-, indexed according to (7.9).
`nf`	Active number of flavours. This parameter is needed to construct the appropriate $2n_\mathrm{f} \times 2n_\mathrm{f}$ transformation matrix that acts on `qvec(-nf:nf)`.

[43] You can call `nptabs(iset)` beforehand to check if the pdf set exists and has enough pdf tables.

```
call QQFROME ( evec, *qvec, nf )
```

Transform the coefficients of a linear combination of basis vectors from the singlet/non-singlet basis to the flavour basis. The arguments are as for `efromqq`.

7.8 Convolution Tools

With the convolution tools you can calculate convolution integrals like

$$x[f \otimes K](x) \quad \text{and} \quad x[f_a \otimes f_b](x),$$

where f is a parton *number* density, and K is a convolution kernel; see Section 3.2 for how convolution integrals are computed and where the factor x in front comes from. Such convolutions are the building blocks to compute structure functions in deep inelastic scattering, or parton luminosities in hadron-hadron scattering.

```
val = FCROSSK ( w, idw, idum, idf, ix, iq )
```

Calculate the convolution $x[f \otimes K](x)$ at a grid point in x and μ^2.

`w`	Toolbox workspace.
`idw`	Weight table identifier in the global format (7.1). You can use a weight table from internal memory, provided its identifier is constructed by a call to `idspfun`.
`idum`	Not used.
`idf`	Pdf identifier in the global format (7.1). You can use a pdf from internal memory, provided its identifier is constructed by a call to `ipdftab`. The input pdf must have been evolved with the current (active) set of evolution parameters.
`ix`, `iq`	Indices of an x-μ^2 grid point.

```
val = FCROSSF ( w, idw, idum, ida, idb, ix, iq )
```

Calculate the convolution $x[f_a \otimes f_b](x)$ at a grid point in x and μ^2.

`w`	Toolbox workspace.
`idw`	Identifier of a weight table in `w`, previously filled by a call to `makewx`.
`idum`	Not used.
`ida`, `idb`	Pdf identifiers in the global format (7.1). You can use a pdf from internal memory, provided its identifier is constructed by a call to `ipdftab`. The pdfs must have been evolved with the current (active) set of evolution parameters.
`ix`, `iq`	Indices of an x-μ^2 grid point.

The convolution routines above can be used to build a structure function or a parton luminosity at a grid point (`ix`,`iq`) in x and μ^2. The idea is to pack this calculation into a function `stfun(ix,iq)` which is then passed to the routine `stfunxq` that takes care of the interpolation to any value of x and μ^2.

```
call STFUNXQ ( stfun, x, qmu2, stf, n, ichk )
```

Interpolate a function `stfun(ix,iq)` to a list of x and μ^2 values. Note that *any* function of the grid points `ix` and `iq` can be interpolated by this routine.

`stfun`	Double precision function, declared external in the calling routine.
`x`, `qmu2`	List of interpolation points, dimensioned to at least `n` in the calling routine.
`stf`	Contains, on exit, the list of interpolated results.
`n`	Number of items in `x`, `qmu2` and `stf`.
`ichk`	If set to `0` the routine returns a `null` value if x or μ^2 are outside the boundaries of the grid (or cuts); if set non-zero it will insist that all interpolation points are inside the grid boundaries (or cuts).

Note that the interpolation is done in μ^2 and not in Q^2. Note also that `stfunxq` should not be called in a loop over each interpolation point individually but must, instead, be given the *list* of points. In this way, the loop is optimised internally leading to large gains in speed (factor 10 or more, usually).

This scheme of calculating structure functions or parton luminosities is fairly straightforward and therefore suitable for prototyping and debugging. However, there is a considerable amount of overhead so that it is recommended to ultimately move the computation to a fast calculation scheme that is described in the next section. By this you will gain at least an order of magnitude in speed.

7.9 Fast Convolution Engine

The convolution routines provided up to now are slow because there is quite a lot of overhead when the calculation is repeated at more than one interpolation point. In this section we describe a set of routines that does bulk calculations on selected points in the x-μ^2 grid. In this way, loops are internally optimised, redundant calculations are eliminated, and user interface checks are reduced to a minimum. This usually leads to large speed gains of more than an order of magnitude.

The engine works as follows. First, you have to pass a list of interpolation points in x and μ^2 so that the engine can construct an interpolation mesh. Next you should copy pdfs onto the mesh in one or more scratch buffers. A set of fast routines then allows you to operate on these buffers. The final result is accumulated in an end-buffer which is passed to an interpolation routine that produces the list of interpolated structure functions or parton luminosities (or whatever else you want to calculate with the engine).

One feature of the engine is that the calculations can be chained so that all kind of convolution integrals can be computed, such as

$$x[f \otimes K](x), \quad x[f \otimes K_a \otimes K_b](x), \quad x[f_a \otimes f_b](x), \quad x[f_a \otimes f_b \otimes K](x), \quad \textit{etc.}$$

In principle, the output buffer of any fast routine can serve as the input buffer of any other fast routine. There is, however, a little complication related to the amount of information stored in a buffer. For interpolation purposes, it is sufficient to store results only at the mesh points; such a buffer is called *sparse*. A convolution routine, on the other hand, does not only need the values at the mesh points x_i, but also the values at all points $x_j > x_i$. An input buffer with such a storage pattern is called *dense*; a dense buffer is of course more expensive to generate than a sparse buffer. Usually you do not have to worry about sparse and dense buffers, because QCDNUM has reasonable defaults for what kind of buffer is accepted as input, and what kind of buffer is generated on output. You can always override the output default and force a routine to generate a dense or a sparse buffer, as needed.

To guarantee internal consistency, the engine does not accept pdfs that were evolved with other than the current (active) set of parameters nor does it accept a parameter change after you have initialised (or cleared) the buffers. Thus if you want to activate your favourite set of parameters (see Section 7.7), then this must be done *before* the call to `fastini` or `fastclr(0)`.

```
call FASTINI ( x, qmu2, n, ichk )
```

Pass a list of interpolation points and, at the first call, generate the set of scratch buffers. The routine will also link the engine to the current (active) set of evolution parameters.

`x`	Array, dimensioned to at least `n` in the calling routine, filled with x values.
`qmu2`	As above, but for μ^2 (not Q^2).
`n`	Number of entries in `x` and `qmu2` ($<$ `5000`).
`ichk`	If non-zero, `fastini` insists that all interpolation points are within the grid boundaries or cuts.

By default, `10` scratch buffers[44] (`ibuf` = `1`–`10`) are generated at the first call (or cleared if they exist) provided, of course, that there is enough space for the buffers in the QCDNUM internal memory (error message if not). The number of interpolation points is limited to 5000 which means that longer lists have to be processed in batches of 5000. Appendix F.6 of the toolbox tutorial and the example program `longlist.f` show a compact while-loop that does this.

```
call FASTCLR ( ibuf )
```

Clear a scratch buffer. Setting `ibuf` = `0` will clear all buffers and re-link the engine to the current (active) set of evolution parameters.

```
call FASTINP ( w, idf, coef, ibuf, iadd )
```

Copy a pdf from the toolbox workspace `w` or from internal memory into a scratch buffer.

`w`	Workspace with pdfs previously evolved by `evdglap`.

[44]This number can be changed by setting `mbf0` in `qcdnum.inc`.

`idf`	Pdf identifier in the global format (7.1). You can also read a pdf from internal memory, provided its identifier is constructed with `ipdftab`.
`coef`	Array, dimensioned `coef(3:6)` in the calling routine, containing an n_f-dependent factor by which the pdf will be multiplied.
`ibuf`	Output scratch buffer identifier [1–10].
`iadd`	Store (0), add (1) or subtract (-1) the weighted pdf to `ibuf`.

By default, `fastinp` generates a dense buffer; a sparse buffer is generated when you prefix `ibuf` with a minus sign.

```
      call FastInp(w, coef, 1502,  1, 0)      !1=dense  buffer
      call FastInp(w, coef, 1502, -1, 0)      !1=sparse buffer
```

Repeated calls to this routine[45] allow you to store n_f-dependent linear combinations of pdfs from the workspace or the internal memory.

Alternatively you can use the input routines `fastepm`, `fastsns` and `fastsum` below, which do work only for pdfs in internal memory. You may find them quite handy but note that everything you can do with these routines, you can also do with `fastinp`.

```
call FASTEPM ( idum, idf, ibuf )
```

Copy the gluon density or one of the basis pdfs $|e^{\pm}\rangle$ to a scratch buffer.

`idum`	Not used
`idf`	Global pdf identifier constructed with `ipdftab`.
`ibuf`	Output scratch buffer identifier [1–10].

By default, `fastepm` generates a dense buffer; a sparse buffer is generated when you prefix `ibuf` with a minus sign.

```
call FASTSNS ( iset, def, isel, ibuf )
```

Decompose a given linear combination of quarks and anti-quarks into singlet and non-singlet components and copy a specific component to a scratch buffer.

`iset`	Identifier of a pdf set in internal memory [1–24].
`def`	Input array, dimensioned to `def(-6:6)` and filled with the coefficients of a linear combination of quarks and anti-quarks, indexed according to (7.8). The value of `def(0)` corresponds to the gluon and is ignored by this routine.
`isel`	Selection flag [0–7], see below.
`ibuf`	Output scratch buffer identifier [1–10].

[45]Note that once you have selected a sparse output in one of these calls, the output buffer will remain flagged as sparse until you start a new accumulation by setting `iadd` = `0`.

The `isel` flag selects the gluon (0), the singlet component $q_{\rm s}$ (1), the non-singlet component $q^+_{\rm ns}$ (2), the valence component $q_{\rm v}$ (3), the non-singlet component $q^-_{\rm ns}$ (4), the sum $q_{\rm v} + q^-_{\rm ns}$ (5), all non-singlets $q_{\rm v} + q^-_{\rm ns} + q^+_{\rm ns}$ (6) or all quarks (7).

It is important to note that the gluon and singlet pdfs transferred by `fastsns` are weighted by the value of `def` averaged over the number of active flavours $n_{\rm f}$, with `def(0)` not included in the average. Take for example the charge-squared weighted sum of quarks and antiquarks

$$xq = \sum_{n_{\rm f}} e_i^2 \; (xq_i + x\bar{q}_i),$$

defined by `def(i)` = `4/9` and `1/9` for up-type and down-type quarks, respectively. Then for `isel` = `0` or `1` the `fastsns` routine will import the components $\langle e^2\rangle xg$ or $\langle e^2\rangle xq_{\rm s}$ into the buffer, with $\langle e^2\rangle = (2/9, 5/18, 11/45, 5/18)$ for $n_{\rm f} = (3, 4, 5, 6)$. Setting `isel` = `0` or `1` in `fastsns` is thus not the same as calling `fastepm` for the gluon or the singlet.

By default, `fastsns` generates a dense buffer; a sparse buffer is generated when you prefix `ibuf` with a minus sign.

```
call FASTSUM ( iset, coef, ibuf )
```

Copy a linear combination of basis pdfs $|e^{\pm}\rangle$ to a scratch buffer.

`iset`	Identifier of a pdf set in internal memory [1–24].
`coef`	Array of coefficients dimensioned `coef(0:12,3:6)` in the calling routine.
`ibuf`	Output scratch buffer identifier [1–10].

The array `coef(i,nf)` is indexed according to (7.9). Here is code that fills `coef` by transforming a set of quark coefficients from flavour space to singlet/non-singlet space:

```
      dimension qvec(-6:6), coef(0:12,3:6)
      do nf = 3,6
        coef(0,nf) = 0.D0                    !no gluon, thank you
        call efromqq(qvec, coef(1,nf), nf)  !quark coefficients
      enddo
```

By masking-out coefficients, you can copy the singlet component, or various combinations of non-singlets; this is exactly what `fastsns` does. To copy the gluon distribution, you must set all coefficients to zero, except `coef(0,nf)`.

By default, `fastsum` generates a dense buffer; a sparse buffer is generated when you prefix `ibuf` with a minus sign.

```
call FASTFXK ( w, idw, ibuf1, ibuf2 )
```

Calculate the convolution $x[f \otimes K](x)$ at all selected grid points.

`w`	Workspace, previously filled with weights.

`idw`	Array of weight table identifiers declared `idw(4)` in the calling routine. Identifiers must be given in the global format (7.1) and cannot refer to those in internal memory. See below for what should be stored in `idw(i)`.
`ibuf1`	Input buffer, previously filled by `fastinp`, `fastepm`, `fastsns` or `fastsum`.
`ibuf2`	Output buffer with `ibuf2` $\neq$ `ibuf1`.

You can either convolute with a given weight table or with a perturbative expansion of weight tables, depending on what you put in the array `idw`.

- To convolute with a given weight table, set `idw(1)` to the identifier of that weight table and set `idw(2)`, `(3)` and `(4)` to zero.
- To convolute with a perturbative expansion in $\alpha_{\rm s}$,
  - Store the (LO,NLO,NNLO) weight table identifiers in `idw(1)`, `idw(2)` and `idw(3)`. Set the identifier to zero if no such table exists (*e.g.* for $F_{\rm L}$ at LO);
  - Declare in `idw(4)` the leading power of $\alpha_{\rm s}$, that is, multiply (LO,NLO,NNLO) by $(1, \alpha_{\rm s}, \alpha_{\rm s}^2)$ if `idw(4)` $=$ `0` and by $(\alpha_{\rm s}, \alpha_{\rm s}^2, \alpha_{\rm s}^3)$ if `idw(4)` $=$ `1`.

  Note that the expansion is summed up to the current perturbative order. Thus you do not have to specify the identifiers in `idw(2)` and `(3)` when you run in LO.

The routine only accepts a dense buffer as input (otherwise fatal error) and will, by default, generate a sparse buffer as output. If you prefix `ibuf2` with a minus sign, the output buffer will be dense. The output table can then serve as an input to another convolution so that you can calculate multiple convolutions in a chain. For example,[46]

```
          call fastSum( 1, coef,  1 )      ! 1 = f
          call fastFxK( w, idK1,  1, -2 )  ! 2 = f * K1         (dense)
          call fastFxK( w, idK2,  2,  3 )  ! 3 = f * K1 * K2    (sparse)
```

```
call FASTFXF ( w, idx, ibuf1, ibuf2, ibuf3 )
```

Calculate the convolution $x[f_a \otimes f_b](x)$ at all selected grid points.

`w`	Workspace, previously filled with weights.
`idx`	Identifier of a weight table, previously filled by a call to `makewtx`.
`ibuf1, 2`	Input buffers, filled with pdfs. It is allowed to have `ibuf1` $=$ `ibuf2`.
`ibuf3`	Output buffer with `ibuf3` $\neq$ `ibuf1` or `ibuf2`.

The routine accepts only dense buffers as input, and generates a sparse buffer as output, unless the `ibuf2` is prefixed by a minus sign, thus,

[46]It is more efficient, however, to first calculate with `wcrossw` a weight table for $K_3 = K_1 \otimes K_2$, and use that table to convolute f with K_3.

```
        call fastSum( 1, coefa, 1 )          ! 1 = fa
        call fastSum( 1, coefb, 2 )          ! 2 = fb
        call fastFxF( w, idwX,  1,  2, -3 )  ! 3 = fa * fb      (dense)
        call fastFxK( w, idwK,  3,  4 )      ! 4 = fa * fb * K  (sparse)
```

`call FASTKIN ( ibuf, fun )`

Multiply the contents of a buffer by a kinematic factor.

`ibuf`	Identifier of the input buffer.
`fun`	Double precision function, declared `external` in the calling routine, that should return the kinematic factor.

The routine `fastkin` loops over the selected grid points and passes these to `fun` via the argument list, together with the current number of flavors and a threshold indicator:

```
            double precision function fun ( ix, iq, nf, ithresh )
```

`ix,iq`	Grid point indices.
`nf`	Number of flavors at `iq`. This number is bi-valued at the thresholds so that at the charm threshold, for instance, `nf` can be either `3` or `4`.[47]
`ithresh`	Set to `0` if `iq` is not at a threshold and to `+1` (`-1`) if `iq` is at a threshold with the upper (lower) number of flavours. This variable can be used to take NNLO discontinuities into account, as is shown in the example function below which returns $\alpha_s/2\pi$ as the kinematic factor.

```
        double precision function fun(ix,iq,nf,ithresh)
          ..
        if(ithresh.ge.0) then
          fun = altabn(0, iq,1,ierr)  !alfas/2pi with discontinuity
        else
          fun = altabn(0,-iq,1,ierr)  !without discontinuity
        endif
          ..
```

`call FASTCPY ( ibuf1, ibuf2, iadd )`

Copy or accumulate a result in an output buffer.

`ibuf1`	Identifier of the input buffer.
`ibuf2`	Identifier of the output buffer with `ibuf2` $\neq$ `ibuf1`.
`iadd`	Store (`0`), add (`1`) or subtract (`-1`) the result to `ibuf2`.

The type of output buffer (sparse or dense) is the same as that of the input buffer, except that once you have used a sparse input buffer, the output buffer is flagged as sparse and will remain so until you set `iadd` = `0` to start a new accumulation in `ibuf2`.

[47]You may wonder when QCDNUM returns the value `3`, and when the value `4`. This depends on the interpolation point μ^2 to which `iq` is associated: if μ^2 is below (above) μ_c^2, then `nf` = `3` (`4`).

```
call FASTFXQ ( ibuf, *f, n )
```

Interpolate the contents of `ibuf` to the list of x and μ^2 values that was previously passed to QCDNUM by the call to `fastini`.

`ibuf`	Identifier of an input buffer.
`f`	Array dimensioned to at least `n` in the calling routine that will contain, on exit, the interpolated values.
`n`	Number of interpolations requested.

The routine works through the list of interpolation points given in the call to `fastini` and exits when it reaches the end of that list or when the number of interpolations is equal to `n`, whatever happens first. Best is to have a sparse input buffer because a dense buffer, although allowed, contains a lot of mesh points that are not used by `fastfxq`.

The fast convolution engine is designed for structure function, cross-section or parton luminosity calculations but can also be used for simple tasks like efficient pdf interpolation, as is illustrated by the following code (see also the tutorial Appendix F.6).

```
        dimension xx(150), qq(150), pdf(150)

        idg = ipdftab(iset,0)        !gluon in internal memory
        call fastini(xx,qq,150,ichk) !pass list of interpolation points
        call fastepm(idum,idg,-1)    !copy gluon to buffer #1 (sparse!)
        call fastfxq(1,pdf,150)      !interpolate gluon
```

7.10 Error Messages in Add-On Packages

QCDNUM error messages refer to the QCDNUM routine and not to the calling routine. This may become confusing for the user of an add-on package who expects error messages to be issued by the package routines and not by what is inside.

One solution would be that a package catches errors before QCDNUM does, but this would duplicate a good checking mechanism which is already in place. An easier solution is to pass a string to QCDNUM which contains the name of the package routine so that it will be printed together with the error message. For this, the routines `setumsg` and `clrumsg` are provided. For instance one of the first calls in the `zmfillw` routine of the ZMSTF package is

```
         call setUmsg('ZMFILLW')
```

so that, upon error, the user gets additional information:

```
         ------------------------------------------------------------------
         Error in MAKETAB ( W, NW, ITYPES, NP, NEW, ISET, NWDS ) ---> STOP
         ------------------------------------------------------------------
         No x-grid available
         Please call GXMAKE
```

```
        Error was detected in a call to ZMFILLW
```

The last call in `zmfillw` is

```
        call clrUmsg
```

that wipes the additional message. This is important because downstream QCDNUM errors would otherwise appear to have always come from `zmfillw`.

8 Acknowledgements

I am of course indebted to the original authors of QCDNUM, in particular to M. Virchaux who introduced me to the program in 1991.[48] I thank M. Cooper-Sarkar for using preliminary versions of QCDNUM17 in her QCD fits and providing important feedback during the development phase of the first release.

I greatly benefited from the many clarifying discussions with A. Vogt and thank him for the code of the NNLO splitting and coefficient functions. I am grateful to him and to M. Cooper-Sarkar, E. Laenen and R. Thorne for comments on the manuscript.

The C++ interface would not exist without the help of V. Bertone who wrote the first version of the interface and got it going.

This work is part of the research programme of the Foundation for Fundamental Research on Matter (FOM), which is financially supported by the Netherlands Organisation for Scientific Research (NWO).

[48]It was sad to hear that Marc Virchaux passed away in November 2004.

A Space-like and time-like singlet evolution

In this Appendix we specify which splitting functions from ref. [14] enter in the space-like and time-like singlet evolution. For this purpose (2.9) is written as

$$\frac{\partial \boldsymbol{V}}{\partial \ln \mu^2} = \boldsymbol{M} \otimes \boldsymbol{V} \quad \text{with} \quad \boldsymbol{V} = \begin{pmatrix} F \\ G \end{pmatrix} \quad \text{and } \boldsymbol{M} = \begin{pmatrix} P_{\rm qq} & P_{\rm qg} \\ P_{\rm gq} & P_{\rm gg} \end{pmatrix}.$$

For space-like evolution F and G are the singlet and gluon pdfs while for time-like evolution they stand for the corresponding fragmentation functions. Here we are concerned with the expansion of the splitting functions up to NLO:

$$\boldsymbol{M} = a_{\rm s}\,\boldsymbol{M}^{(0)} + a_{\rm s}^2\,\boldsymbol{M}^{(1)} \quad \text{with} \quad a_{\rm s} \equiv \frac{\alpha_{\rm s}}{2\pi}.$$

The following four functions are defined in [14]

$$\begin{array}{ll} p_{\rm FF} = (1+x^2)/(1-x) & p_{\rm GF} = x^2 + (1-x)^2 \\ p_{\rm FG} = [1+(1-x)^2]/x & p_{\rm GG} = 1/(1-x) + 1/x - 2 + x - x^2. \end{array}$$

The four LO splitting functions are then written as

$$\begin{array}{ll} P_{\rm FF}^{(0)} = C_{\rm F}\,[p_{\rm FF}]_+ & P_{\rm GF}^{(0)} = 2T_{\rm R}n_{\rm f}\,p_{\rm GF} \\ P_{\rm FG}^{(0)} = C_{\rm F}\,p_{\rm FG} & P_{\rm GG}^{(0)} = 2C_{\rm G}x^{-1}[xp_{\rm GG}]_+ - \tfrac{2}{3}T_{\rm R}n_{\rm f}\,\delta(1-x) \end{array}$$

with the colour factors and the regularisation prescription given by

$$C_{\rm F} = \tfrac{4}{3}, \quad C_{\rm G} = 3, \quad T_{\rm R} = \tfrac{1}{2} \quad \text{and} \quad [f(x)]_+ \equiv f(x) - \delta(1-x)\int_0^1 f(y)\mathrm{d}y.$$

The NLO splitting functions for space-like (S) and time-like (T) processes are

$$\begin{aligned} P_{\rm FF}^{(1,\rm U)} &= \hat{P}_{\rm FF}^{(1,\rm U)} - \delta(1-x)\int_0^1 \mathrm{d}x\; x\left[\hat{P}_{\rm FF}^{(1,\rm T)} + \hat{P}_{\rm FG}^{(1,\rm T)}\right] \\ P_{\rm GF}^{(1,\rm U)} &= \hat{P}_{\rm GF}^{(1,\rm U)} \\ P_{\rm FG}^{(1,\rm U)} &= \hat{P}_{\rm FG}^{(1,\rm U)} \\ P_{\rm GG}^{(1,\rm U)} &= \hat{P}_{\rm GG}^{(1,\rm U)} - \delta(1-x)\int_0^1 \mathrm{d}x\; x\left[\hat{P}_{\rm GG}^{(1,\rm T)} + \hat{P}_{\rm GF}^{(1,\rm T)}\right], \end{aligned}$$

where U = {S,T}. The functions $\hat{P}_{\rm AB}^{(1,\rm U)}$ are given in Eqs. (11) and (12) of [14].

The space-like splitting function matrices in QCDNUM are

$$\boldsymbol{M}^{(0,\rm S)} = \begin{pmatrix} P_{\rm FF}^{(0)} & P_{\rm GF}^{(0)} \\ P_{\rm FG}^{(0)} & P_{\rm GG}^{(0)} \end{pmatrix} \qquad \boldsymbol{M}^{(1,\rm S)} = \begin{pmatrix} P_{\rm FF}^{(1,\rm S)} & P_{\rm GF}^{(1,\rm S)} \\ P_{\rm FG}^{(1,\rm S)} & P_{\rm GG}^{(1,\rm S)} \end{pmatrix}.$$

The LO time-like matrix is the transpose of the space-like matrix [16]. The NLO matrix is also transposed [33]. Accounting for factors $2n_{\rm f}$, we have

$$\boldsymbol{M}^{(0,\rm T)} = \begin{pmatrix} P_{\rm FF}^{(0)} & 2n_{\rm f}P_{\rm FG}^{(0)} \\ \frac{1}{2n_{\rm f}}P_{\rm GF}^{(0)} & P_{\rm GG}^{(0)} \end{pmatrix} \qquad \boldsymbol{M}^{(1,\rm T)} = \begin{pmatrix} P_{\rm FF}^{(1,\rm T)} & 2n_{\rm f}P_{\rm FG}^{(1,\rm T)} \\ \frac{1}{2n_{\rm f}}P_{\rm GF}^{(1,\rm T)} & P_{\rm GG}^{(1,\rm T)} \end{pmatrix}.$$

B Singularities

In this appendix we denote by $f(x)$ a parton *momentum* density and not a number density. In terms of f the convolution integrals in the evolution equations read

$$I(x) = \int_x^1 \mathrm{d}z\, P(z)\, f\left(\frac{x}{z}\right). \tag{B.1}$$

The splitting functions of the LO splitting matrix $P_{ij}^{(0)}$ in (2.14) can be written as

$$\begin{aligned}
P_{\mathrm{qq}}^{(0)}(x) &= \frac{4}{3}\left[\frac{1+x^2}{(1-x)_+} + \frac{3}{2}\,\delta(1-x)\right] \\
P_{\mathrm{qg}}^{(0)}(x) &= 2n_{\mathrm{f}}\,\frac{1}{2}\left[x^2+(1-x)^2\right] \\
P_{\mathrm{gq}}^{(0)}(x) &= \frac{4}{3}\left[\frac{1+(1-x)^2}{x}\right] \\
P_{\mathrm{gg}}^{(0)}(x) &= 6\left[\frac{x}{(1-x)_+} + \frac{1-x}{x} + x(1-x) + \left(\frac{11}{12}-\frac{n_{\mathrm{f}}}{18}\right)\delta(1-x)\right].
\end{aligned} \tag{B.2}$$

The '+' prescription in (B.2) is defined by

$$[f(x)]_+ = f(x) - \delta(1-x)\int_0^1 f(z)\mathrm{d}z \tag{B.3}$$

so that

$$\int_x^1 f(z)[g(z)]_+\,\mathrm{d}z = \int_x^1 [f(z)-f(1)]\,g(z)\,\mathrm{d}z - f(1)\int_0^x g(z)\,\mathrm{d}z. \tag{B.4}$$

For reference we give the expressions for I_{qq} and I_{gg} obtained from (B.2) and (B.3)

$$\begin{aligned}
I_{\mathrm{qq}}^{(0)}(x) &= \frac{4}{3}\int_x^1 \mathrm{d}z\,\frac{1}{1-z}\left[(1+z^2)f\left(\frac{x}{z}\right) - 2f(x)\right] + \frac{4}{3}\,f(x)\left[\frac{3}{2}+2\ln(1-x)\right] \\
I_{\mathrm{gg}}^{(0)}(x) &= 6\int_x^1 \mathrm{d}z\,\frac{1}{1-z}\left[zf\left(\frac{x}{z}\right) - f(x)\right] + 6\int_x^1 \mathrm{d}z\left[\frac{1-z}{z}+z(1-z)\right]f\left(\frac{x}{z}\right) + \\
&\quad 6\,f(x)\left[\ln(1-x)+\frac{11}{12}-\frac{n_{\mathrm{f}}}{18}\right].
\end{aligned} \tag{B.5}$$

To write down a generic expression we decompose a splitting (or coefficient) function into a regular part (A), singular part (B), product of the two (RS) and a delta function

$$P(x) = A(x) + [B(x)]_+ + R(x)[S(x)]_+ + K(x)\delta(1-x) \tag{B.6}$$

where, of course, not all terms have to be present. The following functions are defined in the logarithmic scaling variable $y = -\ln(x)$:

$$h(y) = f(e^{-y}),\ Q(y) = e^{-y}P(e^{-y}),\ \bar{A}(y) = e^{-y}A(e^{-y}) \tag{B.7}$$

with similar definitions for $\bar{B}$ and $\bar{S}$. Furthermore, $\hat{R}(y) = R(e^{-y})$ and $\hat{K}(y) = K(e^{-y})$ without a factor e^{-y} in front. With these definitions (B.1) can be written as

$$\begin{aligned}
I(y) &= \int_0^y \mathrm{d}u\; Q(u)\; h(y-u) = I_1(y) + I_2(y) + I_3(y) + I_4(y) \qquad \text{with} \\
I_1(y) &= \int_0^y \mathrm{d}u\; \bar{A}(u)\; h(y-u); \\
I_2(y) &= \int_0^y \mathrm{d}u\; \bar{B}(u)\; \left[h(y-u) - h(y)\right] - h(y) \int_0^x \mathrm{d}z\; B(z); \\
I_3(y) &= \int_0^y \mathrm{d}u\; \bar{S}(u)\; \left[\hat{R}(u)h(y-u) - \hat{R}(0)h(y)\right] - \hat{R}(0)h(y) \int_0^x \mathrm{d}z\; S(z); \\
I_4(y) &= \hat{K}(y)h(y), \qquad\qquad (B.8)
\end{aligned}$$

where the last integrals of I_2 and I_3 are still expressed in the variable $x = \exp(-y)$ to avoid integration extending to infinity in our expressions. Note that we are free to swap the arguments u and $y-u$ in (B.8).

The four integrals in (B.8) are tabulated by calls to the toolbox weight routines `makewta`, `makewtb`, `makewrs` and `makewtd`, respectively. In these calls the functions (B.6) are passed as arguments; the transformations (B.7) are done internally in QCDNUM.

C Triangular Systems in the DGLAP Evolution

For the non-singlet evolution we have to solve the equation (see Section 3.4)

$$\boldsymbol{V}\boldsymbol{a} = \boldsymbol{b}. \tag{C.1}$$

The matrix $\boldsymbol{V}$ is a lower triangular Toeplitz matrix, that is, a matrix with the elements V_{ij} depending only on the difference $i-j$ as is shown in the 4×4 example (3.15). This matrix is uniquely determined by storing the first column in a one-dimensional vector $\boldsymbol{v}$ so that $V_{ij} = v_{i-j+1}$ for $i \geq j$, and zero otherwise. Eq. (C.1) is, like any other lower triangular system, iteratively solved by forward substitution

$$\begin{aligned} a_1 &= b_1/v_1 \\ a_i &= \frac{1}{v_1}\left[b_i - \sum_{j=1}^{i-1} v_{(i-j+1)}\, a_j\right] \quad \text{for } i \geq 2. \end{aligned} \tag{C.2}$$

There is no recursion relation between a_{i-1} and a_i so that in each iteration the sums must be accumulated, giving an operation count of $n(n+1)/2$ for a system of n equations. This is as expensive (or cheap) as multiplying the triangular matrix by a vector.

The substitution algorithm can be extended to solve the coupled singlet-gluon equation

$$\begin{pmatrix} \boldsymbol{V}_{\mathrm{qq}} & \boldsymbol{V}_{\mathrm{qg}} \\ \boldsymbol{V}_{\mathrm{gq}} & \boldsymbol{V}_{\mathrm{gg}} \end{pmatrix} \begin{pmatrix} \boldsymbol{f} \\ \boldsymbol{g} \end{pmatrix} \equiv \begin{pmatrix} \boldsymbol{a} & \boldsymbol{b} \\ \boldsymbol{c} & \boldsymbol{d} \end{pmatrix} \begin{pmatrix} \boldsymbol{f} \\ \boldsymbol{g} \end{pmatrix} = \begin{pmatrix} \boldsymbol{r} \\ \boldsymbol{s} \end{pmatrix}, \tag{C.3}$$

where $\boldsymbol{a}$ is a short-hand notation for $\boldsymbol{V}_{\mathrm{qq}}$, *etc.* These matrices are all lower triangular $n \times n$ Toeplitz matrices. Writing out this equation in components it is easy to see that for the first elements f_1 and g_1 we have to solve the 2×2 matrix equation

$$\begin{pmatrix} a_1 & b_1 \\ c_1 & d_1 \end{pmatrix} \begin{pmatrix} f_1 \\ g_1 \end{pmatrix} = \begin{pmatrix} r_1 \\ s_1 \end{pmatrix} \to \begin{pmatrix} f_1 \\ g_1 \end{pmatrix} = \frac{1}{a_1 d_1 - b_1 c_1} \begin{pmatrix} d_1 & -b_1 \\ -c_1 & a_1 \end{pmatrix} \begin{pmatrix} r_1 \\ s_1 \end{pmatrix}. \tag{C.4}$$

For $i \geq 2$ we have to accumulate the sums

$$\begin{aligned} R_i &= r_i - \sum_{j=1}^{i-1} \left[a_{(i+1-j)}\, f_j + b_{(i+1-j)}\, g_j\right] \\ S_i &= s_i - \sum_{j=1}^{i-1} \left[c_{(i+1-j)}\, f_j + d_{(i+1-j)}\, g_j\right] \end{aligned} \tag{C.5}$$

and solve, for each i, the equations

$$\begin{pmatrix} a_1 & b_1 \\ c_1 & d_1 \end{pmatrix} \begin{pmatrix} f_i \\ g_i \end{pmatrix} = \begin{pmatrix} R_i \\ S_i \end{pmatrix} \tag{C.6}$$

The operation count of this algorithm is four times that of (C.2), plus some little overhead to solve the 2×2 matrix equations for each i. It is straight-forward to generalise the algorithm to $n \times n$ coupled equations (Section 7.5).

D Zero Mass Structure Functions

D.1 General Formalism

The zero-mass structure functions $F_2(x,Q^2)$, $F_\mathrm{L}(x,Q^2)$ and $xF_3(x,Q^2)$ in un-polarised deep inelastic scattering are calculated from (3.21) with $\chi = x$. The Wilson coefficients are functions of x (and sometimes n_f) only. We set, for the moment, the physical scale Q^2 equal to the factorisation and renormalisation scale μ^2 and write the singlet/gluon contribution to F_2 and F_L as (there is no contribution to xF_3 since this structure function is a pure non-singlet)

$$\frac{1}{x}\mathcal{F}_i^{(\mathrm{s})}(x,Q^2) = [C_{i,\mathrm{s}} \otimes q_\mathrm{s}](x,\mu^2) + [C_{i,\mathrm{g}} \otimes g](x,\mu^2) \qquad i = 2,\mathrm{L}. \tag{D.1}$$

Likewise, non-singlet contributions to the structure functions are given by

$$\frac{1}{x}\mathcal{F}_i^{(\mathrm{ns})}(x,Q^2) = [C_{i,\mathrm{ns}} \otimes q_\mathrm{ns}](x,\mu^2) \qquad i = 2,\mathrm{L},3 \tag{D.2}$$

where the label 'ns' stands for the non-singlet indices '+', '−' and 'v' as defined by (2.11). To be precise on notation: $\mathcal{F}_2 = F_2$, $\mathcal{F}_\mathrm{L} = F_\mathrm{L}$ and $\mathcal{F}_3 = xF_3$ in (D.1) and (D.2). A structure function is calculated by adding the singlet/gluon and non-singlet parts, weighted by the appropriate combination of electroweak couplings; we refer to [32] for how to compute neutral and charged current cross sections and structure functions in deep inelastic charged lepton and neutrino scattering.

Like the splitting functions, the coefficient functions are expanded in powers of α_s,

$$C_{i,j}^{\mathrm{N}^\ell\mathrm{LO}} = \sum_{k=0}^{\ell} a_\mathrm{s}^k\, C_{i,j}^{(k)} \qquad i = 2,\mathrm{L},3 \qquad j = \mathrm{g},\mathrm{s},+,-,\mathrm{v} \tag{D.3}$$

where $\ell = (0,1,2)$ denotes $(\mathrm{LO},\mathrm{NLO},\mathrm{NNLO})$ and $a_\mathrm{s} = \alpha_\mathrm{s}/2\pi$. The LO coefficient functions are either zero or trivial delta functions:

$$\begin{array}{lll} C_{2,\mathrm{g}}^{(0)} = 0 & C_{2,\mathrm{s}}^{(0)} = \delta(1-x) & C_{2,\mathrm{ns}}^{(0)} = \delta(1-x) \\ C_{\mathrm{L},\mathrm{g}}^{(0)} = 0 & C_{\mathrm{L},\mathrm{s}}^{(0)} = 0 & C_{\mathrm{L},\mathrm{ns}}^{(0)} = 0 \\ C_{3,\mathrm{g}}^{(0)} = 0 & C_{3,\mathrm{s}}^{(0)} = 0 & C_{3,\mathrm{ns}}^{(0)} = \delta(1-x). \end{array} \tag{D.4}$$

The NLO coefficient functions can be found in [10]. For those at NNLO we refer to [34, 35, 36, 37] and the parameterisations given in [38] and [39].

The LO coefficient functions for F_L are zero so that the longitudinal structure function vanishes at LO. An alternative, which we call F'_L, is calculated from the expansion

$$C_{\mathrm{L},j}^{\mathrm{N}^\ell\mathrm{LO}} = \sum_{k=1}^{\ell+1} a_\mathrm{s}^k\, C_{\mathrm{L},j}^{(k)}. \tag{D.5}$$

In this way, $C_{\mathrm{L},j}^{(1)}$ is used already at LO (giving a non-zero F_L) and $C_{\mathrm{L},j}^{(2)}$ at NLO. At NNLO the 3-loop coefficient function $C_{\mathrm{L},j}^{(3)}$ is taken from [40]. As stated in [40], this 3-loop calculation applies only to electromagnetic current exchange so that Z^0 or $\mathrm{W}^\pm$ contributions to F'_L at NNLO are, at present, not available.

D.2 Renormalisation and Factorisation Scale Dependence

To calculate the *renormalisation* scale dependence ($\mu_\mathrm{R}^2 \neq \mu_\mathrm{F}^2$) we replace, in the expansion of the coefficient functions, the powers of a_s by the Taylor series given in (2.17). If the expansion (D.3) is used, the truncation of the right-hand side of (2.17) is to order a_s in NLO and a_s^2 in NNLO. If, for F'_L, the expansion (D.5) is used, the truncation is to order a_s in LO, a_s^2 in NLO and a_s^3 in NNLO, like for the splitting functions.

To calculate the *factorisation* scale dependence ($Q^2 \neq \mu_\mathrm{F}^2$), the coefficient functions in (D.3) and (D.5) are replaced by [38, 39]

$$C_{i,j}^{(0)} \to C_{i,j}^{(0)} \qquad \text{and} \qquad C_{i,j}^{(k)} \to C_{i,j}^{(k)} + \sum_{m=1}^{k} C_{i,j}^{(k,m)} L_\mathrm{F}^m \qquad k \geq 1, \tag{D.6}$$

where $L_\mathrm{F} = \ln(Q^2/\mu_\mathrm{F}^2)$ and $\mu_\mathrm{F}^2 = \mu_\mathrm{R}^2$. To write compact expressions for the $C_{i,j}^{(k,m)}$, we introduce the following vector notation. In the non-singlet sector we have a one-dimensional vector $\boldsymbol{C}_i = C_{i,\mathrm{ns}}$ and a 1×1 matrix $\boldsymbol{P} = P_\mathrm{ns}$. In the singlet/gluon sector we have a 2-dimensional row-vector and a 2×2 matrix that are given by

$$\boldsymbol{C}_i = (C_{i,\mathrm{s}}\ C_{i,\mathrm{g}}) \quad \text{and} \quad \boldsymbol{P} = \begin{pmatrix} P_\mathrm{qq} & P_\mathrm{qg} \\ P_\mathrm{gq} & P_\mathrm{gg} \end{pmatrix}.$$

In this vector notation, the functions $C_{i,j}^{(k,m)}$ in (D.6) are written as

$$\begin{aligned}
\boldsymbol{C}_i^{(1,1)} &= \boldsymbol{C}_i^{(0)} \otimes \boldsymbol{P}^{(0)} \\
\boldsymbol{C}_i^{(2,1)} &= \boldsymbol{C}_i^{(0)} \otimes \boldsymbol{P}^{(1)} + \boldsymbol{C}_i^{(1)} \otimes \Big[\boldsymbol{P}^{(0)} - \beta_0\, \boldsymbol{I}\Big] \\
\boldsymbol{C}_i^{(2,2)} &= \frac{1}{2}\, \boldsymbol{C}_i^{(1,1)} \otimes \Big[\boldsymbol{P}^{(0)} - \beta_0\, \boldsymbol{I}\Big] \\
\boldsymbol{C}_i^{(3,1)} &= \boldsymbol{C}_i^{(0)} \otimes \boldsymbol{P}^{(2)} + \boldsymbol{C}_i^{(1)} \otimes \Big[\boldsymbol{P}^{(1)} - \beta_1\, \boldsymbol{I}\Big] + \boldsymbol{C}_i^{(2)} \otimes \Big[\boldsymbol{P}^{(0)} - 2\beta_0\, \boldsymbol{I}\Big] \\
\boldsymbol{C}_i^{(3,2)} &= \frac{1}{2}\Big\{\boldsymbol{C}_i^{(1,1)} \otimes \Big[\boldsymbol{P}^{(1)} - \beta_1\, \boldsymbol{I}\Big] + \boldsymbol{C}_i^{(2,1)} \otimes \Big[\boldsymbol{P}^{(0)} - 2\beta_0\, \boldsymbol{I}\Big]\Big\} \\
\boldsymbol{C}_i^{(3,3)} &= \frac{1}{3}\boldsymbol{C}_i^{(2,2)} \otimes \Big[\boldsymbol{P}^{(0)} - 2\beta_0\, \boldsymbol{I}\Big].
\end{aligned} \tag{D.7}$$

For F_2, F_L and xF_3, the coefficients are calculated up to $\boldsymbol{C}_i^{(2,2)}$. For F'_L, on the other hand, all coefficients in (D.7) are computed. Note, however, that quite some convolutions are trivial because the LO coefficient functions are either zero or δ-functions, see (D.4).

As mentioned above, the expression (D.6) applies only when $\mu_\mathrm{F}^2 = \mu_\mathrm{R}^2$. It is therefore not possible to vary both scales μ_R^2 and Q^2 at the same time.

D.3 The ZMSTF Package

The ZMSTF package is a QCDNUM add-on with routines that calculate the structure functions F_2, F_L and xF_3 in un-polarised deep inelastic scattering. The structure functions are computed as a convolution of the parton densities with zero-mass coefficient functions, using the fast convolution engine described in Section 7.9.

QCDNUM insists that the pdfs used for the structure function calculation are evolved with the *current* (active) set of evolution parameters, otherwise a fatal error condition is raised. If this happens you must first activate the parameters of the pdfs by a call to `usepar(iset)`, as is described in Section 5.6.

The list of subroutines is given in Table 5. Note that error messages are, in most

Table 5 – Subroutine and function calls in ZMSTF.

Subroutine or function	Description
`ZMWORDS ( *ntotal, *nused )`	Words available, used
`ZMFILLW ( *nused )`	Fill weight tables
`ZMDUMPW ( lun, 'filename' )`	Dump weight tables
`ZMREADW ( lun, 'filename', *nused, *ierr )`	Read weight tables
`ZMDEFQ2 ( a, b )`	Define Q^2
`ZMABVAL ( *a, *b )`	Retrieve a and b coefficients
`ZMQFRMU ( qmu2 )`	Convert μ_F^2 to Q^2
`ZMUFRMQ ( Q2 )`	Convert Q^2 to μ_F^2
`ZSWITCH ( iset )`	Switch pdf set
`ZMSTFUN ( istf, def, x, Q2, *f, n, ichk )`	Structure functions

Output arguments are prefixed with an asterisk (`*`).

cases, issued by the underlying QCDNUM routines and not by the ZMSTF routine itself. However, the calling ZMSTF routine is mentioned in the error message so that you know where it came from.

```
call ZMWORDS ( *ntotal, *nused )
```

`ntotal`	Number of words available in the ZMSTF workspace (`nzmstor` in `zmstf.inc`).
`nused`	Number of words used (set to `0` before the call to `zmfillw` or `zmreadw`).

```
call ZMFILLW ( *nused )
```

Fill the weight tables. The tables are calculated for all flavours $3 \leq n_f \leq 6$ and for all orders LO, NLO, NNLO. On exit, the number of words occupied by the workspace is returned in `nused`. If you get an error message that the internal workspace is too small to contain the weight tables, you should increase the value of the parameter `nzmstor` in the include file `zmstf.inc` and recompile ZMSTF.

This routine (or `zmreadw` below) should be called after an x-μ^2 grid is defined in QCDNUM and before the first call to `zmstfun`. The routine also needs the splitting function weight tables so that `fillwt` or `readwt` must have been called before (fatal error if not).

```
call ZMDUMPW ( lun, 'filename' )
```

Dump the weights in memory via logical unit number `lun` to a disk file. The dump is un-formatted so that the weight file cannot be exchanged across machines.

```
call ZMREADW ( lun, 'filename', *nused, *ierr )
```

Read weights from a disk file via logical unit number `lun`. On exit, `nused` contains the number of words read into the workspace (fatal error if not enough space, see above) and the flag `ierr` is set as follows.

0	Weights are successfully read in.
1	Read error or input file does not exist.
2	Incompatible QCDNUM version.
3	Incompatible ZMSTF version.
4	Incompatible x-μ^2 grid definition.

These errors will not generate a program abort so that one should check the value of `ierr`, and take the appropriate action if it is non-zero.

```
call ZMDEFQ2 ( a, b )
```

Define the relation between the factorisation scale μ_F^2 and Q^2

$$Q^2 = a\mu_F^2 + b.$$

The Q^2 scale can only be varied when the renormalisation and factorisation scales are set equal in QCDNUM. The default setting is `a` = `1` and `b` = `0`. The ranges are limited to `0.1` ≤ `a` ≤ `10` and `-100` ≤ `b` ≤ `100`.

A call to `zmabval(a,b)` reads the coefficients back from memory. To convert between the scales use:

```
          Q2   = zmqfrmu(qmu2)
          qmu2 = zmufrmq(Q2)
```

```
call ZSWITCH ( iset )
```

By default, the structure functions are calculated from the un-polarised parton densities, evolved with QCDNUM (`iset` = `1`). With this routine you can switch to the custom evolution (`4`), or to one of the external pdf sets (`5`–`24`). Switching to polarised pdfs (`2`) or to fragmentation functions (`3`) does not make sense and will produce an error message.

```
call ZMSTFUN ( istf, def, x, Q2, *f, n, ichk )
```

Calculate a structure function for a linear combination of parton densities.

`istf`	Structure function index (1,2,3,4) = $(F_\mathrm{L}, F_2, xF_3, F'_\mathrm{L})$.
`def(-6:6)`	Coefficients of the quark linear combination for which the structure function is to be calculated. The indexing of `def` is given in (5.2).
`x`, `Q2`	Input arrays containing a list of x and Q^2 (not μ^2) values.
`f`	Output array containing the list of structure functions.
`n`	Number of items in `x`, `Q2` and `f`.
`ichk`	If set to zero, `zmstfun` will return a `null` value when x or μ^2 are outside the grid boundaries; otherwise you will get a fatal error message. A μ^2 point that is close or below the QCD scale Λ^2 is considered to be outside the grid boundary.

To calculate a structure function for more than one interpolation point, it is recommended to not execute `zmstfun` in a loop but to pass the entire list of interpolation points in a single call. The loop is then internally optimised for greater speed.

Another way to calculate structure functions is by calling the routine `zmslowf`, with the same argument list as `zmstfun`. This routine was used for prototyping and runs quite slow but provides the possibility to calculate the quark and gluon contributions separately, order by order. This is achieved by setting `ichk` to one of the values given below; a positive (negative) value switches the boundary check on (off).

Contribution	LO	NLO	NNLO
Quark and gluon	± 101	± 102	± 103
Quark only	± 201	± 202	± 203
Gluon only	± 301	± 302	± 303

E Heavy Quark Structure Functions

E.1 General Formalism

A NLO calculation of the heavy quark contributions to the F_2 and $F_{\rm L}$ structure functions in deep inelastic charged lepton-proton scattering is given in [17]. Only electromagnetic exchange contributions are taken into account. In this calculation, a heavy flavour h is not taken to be a constituent of the incoming proton but is, instead, assumed to be exclusively produced in the hard scattering process. Quarks with pole mass $m < m_h$ are taken to be mass-less so that the input light quark densities should have been evolved in the FFNS with $n_{\rm f} = (3, 4, 5)$ for $h = ({\rm c}, {\rm b}, {\rm t})$ [41].

A heavy flavour contribution to F_2 or F_L is calculated from

$$F_k^h(x, Q^2) = \frac{\alpha_s}{2\pi} \left\{ e_h^2\, g \otimes \mathcal{C}_{k,g}^{(0)} + \frac{\alpha_s}{2\pi} \left(e_h^2\, g \otimes \mathcal{C}_{k,g}^{(1)} + e_h^2\, q_{\rm s} \otimes \mathcal{C}_{k,q}^{(1)} + q_{\rm p} \otimes \mathcal{D}_{k,q}^{(1)} \right) \right\}, \qquad \text{(E.1)}$$

where e_h is the charge of the heavy quark (in units of the positron charge), g is the gluon density, $q_{\rm s}$ is the singlet density and

$$q_{\rm p} = \sum_{i=1}^{n_{\rm f}} e_i^2\, (q_i + \bar{q}_i)$$

is the charge-weighted proton quark distribution for $n_{\rm f}$ light flavours. The first term in (E.1) is the LO contribution from the photon-gluon fusion process $\gamma^* g \to h\bar{h}$. The last three terms correspond to the NLO sub-process $\gamma^* g \to h\bar{h}g$ and $\gamma^* q \to h\bar{h}q$.[49] For the heavy quark coefficient functions $\mathcal{C}$ and $\mathcal{D}$ in (E.1) we refer to [17].[50]

In terms of a number density $f(x, \mu^2)$, the convolution integrals in (E.1) are defined by

$$f \otimes \mathcal{C} = \int_{ax}^{1} \frac{dz}{z}\, z f(z, \mu^2)\, \mathcal{C}(x/z, Q^2, \mu^2, m_h^2) \qquad \text{(E.2)}$$

where $a = 1 + 4m_h^2/Q^2$ and μ^2 is the factorisation (equals renormalisation) scale which is usually set to $\mu^2 = Q^2$ or $\mu^2 = Q^2 + 4m_h^2$. The kinematic domain where the heavy quarks contribute is restricted by the requirement that the square of the γ^*p centre of mass energy must be sufficient to produce the $h\bar{h}$ pair: $W^2 = M^2 + Q^2(1-x)/x \geq M^2 + 4m_h^2$ so that the lower integration limit $ax \leq 1$ in (E.2). It turns out that the dependence of the coefficient functions on the relation between Q^2 and μ^2 cannot be factorised so that each setting of the scale parameters needs its own set of weight tables. To calculate the renormalisation scale dependence, the powers of $a_{\rm s} = \alpha_{\rm s}/2\pi$ in (E.1) are replaced by the Fourier expansion (2.17), truncated to $a_{\rm s}$ in LO, and to $a_{\rm s}^2$ in NLO. Note that you can vary either $\mu_{\rm R}^2$ or Q^2 with respect to $\mu_{\rm F}^2$, but not both at the same time.

[49] In the LO and the first two NLO terms the virtual photon couples to the heavy quark, hence the factor e_h^2 in (E.1). The last NLO term describes the process where the virtual photon couples to a light quark which subsequently branches into a $h\bar{h}$ pair via an intermediate gluon: hence the appearance of the charge weighted sum, $q_{\rm p}$, of light quark distributions.

[50] Some of these coefficient functions are given as interpolation tables (taken from code provided by S. Riemersma) since they are too complex to be cast into analytic form. Note that in [17] the coefficient functions are convoluted with parton momentum densities and not with number densities [41].

The convolution integral (E.2) is not of the general form (3.21): (i) the factor x in front is missing; (ii) the pdf is $xf(x)$ and not $f(x)$ and (iii) the argument of C is x/z and not χ/z. This mismatch is cured by presenting to QCDNUM the modified kernel

$$C_{\text{modified}}(\chi, \mu^2, Q^2, m_h^2) \equiv \frac{a}{\chi}\, C_{\text{published}}\left(\frac{\chi}{a}, \mu^2, Q^2, m_h^2\right), \text{ with } \chi \equiv ax.$$

To make the heavy quark calculation available in QCDNUM17 (as it was in QCDNUM16) we provide the add-on package HQSTF described below.

E.2 The HQSTF Package

The HQSTF package calculates up to NLO the heavy flavour contributions to the F_2 or F_L structure functions from pdfs evolved in the FFNS scheme with n_f light flavours. The list of subroutines is given in Table 6. We will only describe here the routines `hqfillw`

Table 6 – Subroutine and function calls in HQSTF.

Subroutine or function	Description
`HQWORDS ( *ntotal, *nused )`	Words available, used
`HQFILLW ( istf, qmass, aq, bq, *nused )`	Fill weight tables
`HQDUMPW ( lun, 'filename' )`	Dump weight tables
`HQREADW ( lun, 'filename', *nused, *ierr )`	Read weight tables
`HQPARMS ( *qmass, *aq, *bq )`	Retrieve parameters
`HQQFRMU ( qmu2 )`	Convert μ_F^2 to Q^2
`HQMUFRQ ( Q2 )`	Convert Q^2 to μ_F^2
`HSWITCH ( iset )`	Switch pdf set
`HQSTFUN ( istf, icbt, def, x, Q2, *f, n, ichk )`	Structure functions

Output arguments are prefixed with an asterisk (*).

and `hqstfun`, the other ones being similar to those in the ZMSTF package.

```
call HQFILLW ( istf, qmass, aq, bq, *nused )
```

Fill the weight tables. To be called before anything else.

`istf`	Select structure function: `1` = F_L, `2` = F_2 and `3` = both.
`qmass(3)`	Input array with the c, b, t quark masses in GeV. If a quark mass is set to $m_h < 1$ GeV, no tables will be generated for that quark.
`aq`, `bq`	Defines the relation $Q^2 = a\mu_F^2 + b$.
`nused`	Gives, on exit, the number of words used in the workspace.

You will get a fatal error if the workspace is not large enough to hold all tables. In that case you can increase the value of `nhqstor` in the include file `hqstf.inc` and recompile HQSTF. The values of the mass and scale parameters can be retrieved at any time after the call to `hqfillw` (or `hqreadw`) by a call to `hqparms(qmass,aq,bq)`.

```
call HQSTFUN ( istf, icbt, def, x, Q2, *f, n, ichk )
```

Calculate the heavy quark contribution to a structure function.

`istf`	Calculate F_L (1) or F_2 (2).
`icbt`	Select contribution from charm (1), bottom (2) or top (3).
`def(-6:6)`	Coefficients of the quark linear combination for which the structure function is to be calculated. The indexing of `def` is given in (7.8).
`x`, `Q2`	Input arrays containing a list of x and Q^2 (not μ^2) values.
`f`	Output array containing the list of structure functions.
`n`	Number of items in `x`, `Q2` and `f`.
`ichk`	If set to zero, `hqstfun` will return a `null` value when x or μ^2 are outside the grid boundaries;[51] otherwise you will get a fatal error message.

The routine checks that for `icbt` = (1,2,3) = (c,b,t) the pdfs were evolved in the FFNS with $n_f = (3, 4, 5)$ and issues an error message if that is not the case. To relax the check you can prefix `icbt` by a minus sign: both the FFNS and the MFNS are then allowed with any number of fixed flavours. The VFNS does not make sense and is not allowed.

Here is a snippet of code that, in combination with ZMSTF, calculates the d,u,s contribution, the charm contribution and the total F_2 (neglecting bottom and top) in charged lepton-proton scattering (the pdfs should have been evolved with $n_f = 3$ flavours).

```
      dimension x(100),Q2(100),F2dus(100),F2c(100),F2p(100)
      dimension proton(-6:6)
      data proton /4.,1.,4.,1.,4,.1.,0.,1.,4.,1.,4.,1.,4./ !divide by 9
        ..
      call zmstfun(2,    proton, x, Q2, F2dus, 100, ichk)
      call hqstfun(2, 1, proton, x, Q2, F2c  , 100, ichk)
      do i = 1,100
        F2p(i) = F2dus(i) + F2c(i)
      enddo
```

[51]For technical reasons a cut $Q^2 > 0.5$ GeV^2 is also imposed.

F 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 `Toolbox00.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.

F.1 How to partition a workspace

In this section we describe how to set-up the toolbox workspace `w(nw)` for our evolution of unpolarised and polarised pdfs at LO. The toolbox routines are described in Section 7.3.

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 ($P_\mathrm{qq}, P_\mathrm{qg}, P_\mathrm{gq}, P_\mathrm{gg}$) and 13 pdf tables (one gluon and up to 6 flavours of quark and antiquark). Note from eqs. (**??**) and (B.2) in Appendix B that there are two splitting functions that depend only on x (type-1) and two that depend on x and n_f (type-2).[52]

There is of course nothing against putting all these tables into one set:[53]

```
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.

$$\underbrace{\boxed{\alpha_s}}_{1}\ \overbrace{\underbrace{\boxed{\text{weights}}}_{2}\underbrace{\boxed{\text{pdfs}}}_{3}}^{\text{unpolarised}}\ \overbrace{\underbrace{\boxed{\text{weights}}}_{4}\underbrace{\boxed{\text{pdfs}}}_{5}}^{\text{polarised}}$$

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

[52]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.

[53]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. (F.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 hard-wire 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.

F.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:

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

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. (**??**) and (B.2) in Appendix B:

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

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 (F.2) and will assume from now on that this has been done.

From (F.2) it is seen that the splitting functions in the second column of the matrix in (F.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 7.3. 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 7.3 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`.

F.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`.[54]

[54]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 5.7) 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 routine `altabn` (Section 5.7), 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 = ALTABN( 0, -iq, 1, ierr )  !alfas/2pi
else
  AlfasFun = ALTABN( 0,  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 `altabn` 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`.

F.4 Singlet/gluon and non-singlet evolution

We can use the $n \times n$ `evdglap` routine (Section 7.5) both for the non-singlet and the singlet/gluon evolution by setting $n = 1$ or 2, respectively. Because P_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 (7.8) and (7.9) as QCDNUM. For convenience we show them here again for the flavour basis

$$\begin{array}{ccccccccccccc} -6 & -5 & -4 & -3 & -2 & -1 & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline \bar{t} & \bar{b} & \bar{c} & \bar{s} & \bar{u} & \bar{d} & g & d & u & s & c & b & t \end{array}, \tag{F.3}$$

and for the singlet/non-singlet basis

$$\begin{array}{ccccccccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\ \hline g & q_\mathrm{s} & e_2^+ & e_3^+ & e_4^+ & e_5^+ & e_6^+ & q_\mathrm{v} & e_2^- & e_3^- & e_4^- & e_5^- & e_6^- \end{array}. \tag{F.4}$$

The singlet/non-singlet basis functions $e^\pm$ are in QCDNUM defined by[55]

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

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 7.5.

[55]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`.[56] 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
```

[56]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 (F.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 $n_\mathrm{f} = 3$. To handle all flavours 3–6 in the FFNS you can simply set the upper limits in the loops above to n_f and $n_\mathrm{f} + 6$, 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.[57]

```
      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
      ..
```

[57]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.

F.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

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

In matrix notation this reads

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

We adopt the convention (F.5) for the basis pdfs so that we can use the routine `efromqq` (Section 7.7) 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 $2n_\mathrm{f} \times 2n_\mathrm{f}$ 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	×	L	L	L	C	B	T
2	T	B	C	L	L	L	×	L	L	L	C	B	T
3	T	B	C	L	L	L	×	L	L	L	C	B	T
4	T	B	C	L	L	L	×	L	L	L	C	B	T
5	T	B	C	L	L	L	×	L	L	L	C	B	T
6	T	B	C	L	L	L	×	L	L	L	C	B	T
7	T	B	C	C	C	C	×	C	C	C	C	B	T
8	T	B	C	C	C	C	×	C	C	C	C	B	T
9	T	B	B	B	B	B	×	B	B	B	B	B	T
10	T	B	B	B	B	B	×	B	B	B	B	B	T
11	T	T	T	T	T	T	×	T	T	T	T	T	T
12	T	T	T	T	T	T	×	T	T	T	T	T	T

(F.8)

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

F.6 Your own interpolation routine

The toolbox routines `evplist` and `evtable` can be used to interpolate the pdfs stored in the toolbox 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 toolbox workspace into the QCDNUM internal memory and then call the built-in interpolation routines such as `pdflst`, *etc.* For this we can use `pdfext` (Section 5.8) or, better, `evpcopy` (Section 7.5) 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 toolbox workspace.

Second, we can provide our own interpolation routine which can be done very easily with the fast convolution engine described in Section 7.9. This may be attractive because you can write the code such that it can interpolate *any* pdf in the toolbox 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 7.9 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 (F.6) as

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

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 (F.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 . . .

F.7 How to compute a structure function

F.8 Make it robust and user-friendly

G QCDNUM17 Releases and Updates

QCDNUM17 versions are labelled as `qcdnum-17-rr/uu` where `rr` is the release number, and `uu` is the update number of a given release.[58] Here is an up-to-date list of all releases and updates.

`17-rr/uu`	`dd-mm-yy`	– Description
`17-00/07`	`26-02-16`	– Correct error in the NLO evolution of singlet fragmentation functions (splitting function matrix was not transposed). Implement NLO matching conditions in the time-like evolution.
`17-00/06`	`10-07-12`	– In the MFNS, the function `asfunc` evolved α_s in the FFNS. The QCDNUM internal α_s tables were not affected by this bug.
		– HQSTF: the routine `hqstfun` did not accept the MFNS. Preceding `icbt` by a minus sign now allows for both the FFNS and the MFNS, with any number of flavours.
`17-00/05`	`10-04-12`	– Access to version number and `qcdnum.inc` parameters (via `getint`).
		– New routine `mixfns` to set the mixed flavour number scheme.
		– New routine `set|getcut` to set (get) evolution cuts on the grid.
		– New function `lpassc` to check if a point passes the cuts.
		– New routines `pdflst` and `pdftab` for fast pdf interpolations.
		– Remove the `mpt0` limit on the number of interpolations in QCDNUM, ZMSTF and HQSTF. In `fastini` the `mpt0` limit still exists.
		– Increase storage sizes `nwf0`, `nzmstor` and `nhqstor`.
		– ZMSTF: New routine `zmwords` gives access to storage size/use.
		– ZMSTF: Possibility to separately calculate gluon or quark contributions to structure functions, order by order (with `zmslowf`).
		– HQSTF: New routine `hqwords` gives access to storage size/use.
		– HQSTF: When using external pdfs (`iset` = 5–9), the availability of unpolarised pdfs (`iset` = 1) was imposed. Bug now fixed.
`17-00/04`	`18-07-11`	– Increase storage sizes `nwf0`, `nzmstor` and `nhqstor`.
		– Set $Q^2 = 0.25$ GeV^2 in the HQSTF coefficient functions when the input value of $Q^2 < 0$. This avoids problems in `hqfillw` if the low end of the μ^2 grid maps onto negative Q^2.
`17-00/03`	`30-03-11`	– Rename splitting/coefficient functions in QCDNUM and ZMSTF to avoid name clashes with QCDNUM16 and—more important—with LHAPDF (which has QCDNUM16 inside).
`17-00/02`	`07-10-10`	– Comments from the CPC referees included in the write-up.
`17-00/01`	`03-09-10`	– Adjust internal cuts on the α_s evolution to allow for evolution to lower Q^2. The cuts in the original release were set too tight.
`17-00/00`	`08-05-10`	– Initial release.

[58]Updates are bug fixes or changes in the code that do not require modification of user programs. Releases, on the other hand, may affect user code by adding extra functionality to existing QCDNUM routines or by replacing an old routine with a new one. In the latter case, a call to the old routine will always generate an error message pointing to the replacement, the description of which can then be found in the write-up.

References

[1] V.N. Gribov and L.N. Lipatov, Sov. J. Nucl. Phys. **15**, 438 (1972);
L.N. Lipatov, Sov. J. Nucl. Phys. **20**, 94 (1975);
G. Altarelli and G. Parisi, Nucl. Phys. **B126**, 298 (1977);
Y. Dokshitzer, Sov. Phys. JETP **46**, 641 (1977).

[2] S. Moch, J.A.M. Vermaseren and A. Vogt, Nucl. Phys. **B688**, 101 (2004), hep-ph/0403192.

[3] A. Vogt, S. Moch and J.A.M. Vermaseren, Nucl. Phys. **B691**, 129 (2004), hep-ph/0404111.

[4] A. Ouraou, Ph. D. Thesis, Université de Paris-XI (1988);
M. Virchaux, Ph. D. Thesis, Université de Paris-VII (1988).

[5] M. Virchaux and A. Milsztajn, Phys. Lett. **B274**, 221 (1992).

[6] NMC, M. Arneodo et al., Phys. Lett. **B309**, 222 (1993).

[7] ZEUS Collab., M. Derrick et al., Phys. Lett. **B345**, 576 (1995);
ZEUS Collab., J. Breitweg et al., Eur. Phys. J. **C7**, 609 (1999);
ZEUS Collab., S. Chekanov et al., Phys. Rev. **D67**, 012007 (2003).

[8] M. Botje, Eur. Phys. J. **C14**, 285 (2000).

[9] V. Bertone and M. Botje, 'A C++ interface to QCDNUM', arXiv:1712.08162 [hep-ph] (2017).

[10] W. Furmanski and R. Petronzio, Z. Phys. **C11**, 293 (1982).

[11] O.V. Tarasov, A.A. Vladimirov and A.Yu Sharkov, Phys. Lett. **B93**, 429 (1980);
S.A. Larin and J.A.M. Vermaseren, Phys. Lett. **B303**, 224 (1993).

[12] K.G. Chetyrkin, B.A. Kniehl and M. Steinhauser, Phys. Rev. Lett. **79**, 2184 (1997), hep-ph/9706430.

[13] G. Gurci, W. Furmanski and R. Petronzio, Nucl. Phys. **B175**, 27 (1980).

[14] W. Furmanski and R. Petronzio, Phys. Lett. **97B**, 437 (1980).

[15] R. Mertig and W.L. van Neerven, Z. Phys. **C70**, 637 (1996) hep-ph/9506451;
W. Vogelsang, Nucl. Phys. **B475**, 47 (1996), hep-ph/9603366.

[16] P. Nason and B.R. Webber, Nucl. Phys. **B421**, 473 (1994); Erratum Nucl. Phys. **B480**, 755 (1996).

[17] E. Laenen et al., Nucl. Phys. **B392**, 162 (1993);
S. Riemersma et al., Phys. Lett. **B347**, 143 (1995).

[18] W.K. Tung et al., J. Phys. **G28**, 983 (2002);
S. Kretzer et al., Phys. Rev. **D69**, 114005 (2004).

[19] R.S. Thorne, Phys. Rev. **D73**, 054019 (2006), hep-ph/0601245 and references therein.

[20] A. Vogt, Comput. Phys. Commun. **170**, 65 (2005), hep-ph/0408244.

[21] M. Buza et al., Eur. Phys. J. **C1**, 301 (1998), hep-ph/9612398.

[22] M. Cacciari, P. Nason and C. Oleari, JHEP **0510**, 034 (2005), arXiv:hep-ph/0504192.

[23] M. Glück, E. Reya and M. Stratmann, Nucl. Phys. **B422**, 37 (1994).

[24] G.P. Salam and J. Rojo, Comput. Phys. Commun. **180**, 120 (2009), ArXiv:0804.3755.

[25] M. Miyama and S. Kumano, Comput. Phys. Commun. **94**, 185 (1996), hep-ph/9508246;
P.G. Ratcliffe, Phys. Rev. **D63**, 116004 (2001), hep-ph/0012376;
C. Pascaud and F. Zomer, hep-ph/0104013 (2001);
A. Cafarella and C. Coriano, Comput. Phys. Commun. **160**, 213 (2004), hep-ph/0311313;
A. Cafarella, C. Coriano and M. Guzzi, Comput. Phys. Commun. **179**, 665 (2008), ArXiv:0803.0462.

[26] C. de Boor, 'A Practical Guide to Splines', Applied Mathematical Sciences 27, Springer-Verlag New York Inc. (1978);
L.L. Schumaker, 'Spline Functions: Basic Theory', Krieger Publishing Company, Malabar Florida (1993);
R. Kress, 'Numerical Analysis', Springer-Verlag New York Inc. (1998).

[27] E. Eichten et al., Rev. Mod. Phys. **56**, 579 (1984).

[28] R.S. Thorne and W.K. Tung, in Proc. workshop 'HERA and the LHC', H. Jung and A. De Roeck eds., DESY-PROC-2009-02, arXiv:0903.3861, pp. 332–351 (2009).

[29] G. Salam and A. Vogt in the QCD/SM working group report of the workshop 'Physics at TEV Colliders', Les Houches, May 2001, FERMILAB-CONF-02-410, hep-ph/0204316.

[30] S. Alekhin et al., in Proc. workshop 'HERA and the LHC' Part A, H. Jung and A. De Roeck eds., DESY-PROC-2005-01, CERN-2005-014, hep-ph/0601012, pp. 119–159 (2006).

[31] C.G. Page 'Professional Programmer's Guide to Fortran77', (1988 updated 2005), `http://www.star.le.ac.uk/~cgp/prof77.pdf`.

[32] D. Roberts, 'The Structure of the Proton', Cambridge University Press (1990);
U.F. Katz, 'Deep Inelastic Positron-Proton Scattering in the High-Momentum-Transfer-Regime of Hera', Springer Tracts in Modern Physics (2000);
A.M. Cooper-Sarkar, R.C.E. Devenish and A. De Roeck, Int. J. Mod. Phys. **A13**, 3385 (1998), hep-ph/9712301.

[33] M. Botje, 'Erratum for the time-like evolution in QCDNUM', arXiv:1602.08383 (2016).

[34] W.L. van Neerven and E.B. Zijlstra, Phys. Lett. **B272**, 127 (1991).

[35] E.B. Zijlstra and W.L. van Neerven, Phys. Lett. **B273**, 476 (1991).

[36] E.B. Zijlstra and W.L. van Neerven, Phys. Lett. **B297**, 377 (1992).

[37] J. Sanchez Guillen et al., Nucl. Phys. **B353**, 337 (1991).

[38] W.L. van Neerven and A. Vogt, Nucl. Phys. **B568**, 263 (2000), hep-ph/9907472.

[39] W.L. van Neerven and A. Vogt, Nucl. Phys. **B588**, 345 (2000), hep-ph/0006154.

[40] S. Moch, J.A.M. Vermaseren and A. Vogt, Phys. Lett. **B606**, 123 (2005), hep-ph/0411112.

[41] E. Laenen, private communication.

List of Tables

Index